

UNIVERSITY OF VERONA

DOCTORAL THESIS

Taming Strings in Dynamic Languages

An Abstract Interpretation-based Static Analysis Approach

Author:

Vincenzo ARCERI

Supervisor:

Prof. Isabella MASTROENI

Thesis referees:

Prof. Sergio MAFFEIS

Prof. Xavier RIVAL

Defence committee members:

Prof. Roberto BRUNI

Prof. Sergio MAFFEIS

Prof. Isabella MASTROENI

Local committee members:

Prof. Isabella MASTROENI

Prof. Graziano PRAVADELLI

Prof. Roberto SEGALA

*A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy*

at the

Department of Computer Science

April 21, 2020



UNIVERSITÀ
di **VERONA**

Dipartimento
di **INFORMATICA**

Declaration of Authorship

I, Vincenzo ARCERI, declare that this thesis titled, “Taming Strings in Dynamic Languages - An Abstract Interpretation-based Static Analysis Approach” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: April 21, 2020

“Agire senza obiettivo, giocare per distruggere.”

Alessandro Pecile

UNIVERSITY OF VERONA

Abstract

School of Science and Engineering
Department of Computer Science

Doctor of Philosophy

Taming Strings in Dynamic Languages An Abstract Interpretation-based Static Analysis Approach

by Vincenzo ARCERI

In the recent years, dynamic languages such as JavaScript, Python or PHP, have found several fields of applications, thanks to the multiple features provided, the agility of deploying software and the seeming facility of learning such languages. In particular, strings play a central role in dynamic languages, as they can be implicitly converted to other type values, used to access object properties or transformed at run-time into executable code. In particular, the possibility to dynamically generate code as strings transformation breaks the typical assumption in static program analysis that the code is an immutable object, indeed static. This happens because program's essential data structures, such as the control-flow graph and the system of equation associated with the program to analyze, are themselves dynamically mutating objects. In a sentence: *"You can't check the code you don't see"*. For all these reasons, dynamic languages still pose a big challenge for static program analysis, making it drastically hard and imprecise.

The goal of this thesis is to tackle the problem of statically analyzing dynamic code by treating the code as any other data structure that can be statically analyzed, and by treating the static analyzer as any other function that can be recursively called. Since, in dynamically-generated code, the program code can be encoded as strings and then transformed into executable code, we first define a novel and suitable string abstraction, and the corresponding abstract semantics, able to both keep enough information to analyze string properties, in general, and keep enough information about the possible executable strings that may be converted to code. Such string abstraction will permit us to distill from a string abstract value the executable program expressed by it, allowing us to recursively call the static analyzer on the synthesized program.

The final result of this thesis is an important first step towards a sound-by-construction abstract interpreter for real-world dynamic string manipulation languages, analyzing also string-to-code statements, that is the code that standard static analysis *"can't see"*.

Contents

Declaration of Authorship	v
Abstract	ix
1 Introduction	1
1.1 Why is it important to analyze dynamic code?	1
1.2 JavaScript overview	3
1.2.1 eval in the wild	4
1.3 Contributions and structure of the thesis	5
2 Mathematical background	9
2.1 Basic notions and notation	9
2.2 Posets, semi-lattices and lattices	11
2.3 Fix-point theory	14
2.4 Galois connections	14
2.5 Abstract interpretation	16
2.5.1 Concrete objects, abstract objects and Galois connections	17
2.5.2 Fix-point computations	19
2.5.3 Fix-point extrapolation and interpolation	20
2.5.4 Abstract domains collectively	22
2.5.5 Making abstract interpretations complete	22
2.6 Strings, languages and finite state automata	24
2.6.1 Regular expressions	31
3 A dynamic imperative core language: μJS	35
3.1 μ JS syntax and semantics	35
3.2 Semantics over CFGs and static analysis of μ JS	39
4 Towards a string abstract domain for dynamic languages	47
4.1 An example of complete shell	47
4.2 Making JavaScript string abstract domains complete	49
4.2.1 Completing SAFE string abstract domain	49
4.2.2 Completing TAJIS string abstract domain	51
4.3 What we gain from using a complete abstract domain?	54
4.4 Can we use complete shells for dynamic code analysis?	56
5 The finite state automata domain	59
5.1 $DFA_{/\equiv}$ abstract domain	59
5.2 Characterization of substrings languages	63
5.2.1 Substring language between two fixed indexes	63
5.2.2 Substring language after a fixed initial index	66
5.2.3 Substring language to an unbounded final index	66

6	A sound abstract interpreter for dynamic code	69
6.1	μ JS with eval	69
6.2	Dyn: An abstract domain for μ JS	70
6.2.1	Abstract semantics of μ JS	73
6.3	Towards an analysis for dynamic code	82
6.4	The analyzer architecture	83
6.5	Approximating eval executable code	84
6.5.1	StmSyn: Extracting the executable language	84
6.5.2	CFGGen: Control-flow graph generation	87
6.5.3	Abstracting sequences of eval nested calls	90
6.6	Evaluating the analyzer	91
6.6.1	Limitations	94
6.6.2	Comparison with TAJs	94
7	An abstract domain for objects in dynamic languages	99
7.1	Object concrete semantics	100
7.2	An abstract domain for objects	102
7.2.1	Normalization	103
7.2.2	Objects-related abstract semantics	105
7.2.3	Widening	107
8	Conclusions	111
8.1	Related works	111
8.2	Future directions	113
A	Proofs	117
	Bibliography	138

List of Figures

1.1	(a) Object property lookup with <code>eval</code> , (b) Object property lookup without <code>eval</code>	4
1.2	JavaScript obfuscated malware example.	5
2.1	Example of Hasse diagram.	11
2.2	(a) Example of join semi lattice and (b) meet semi lattice	12
2.3	Example of DFA.	26
2.4	Left quotient and right quotient algorithms.	27
2.5	Suffix and prefix algorithms.	28
2.6	(a) ϵ -NFA A . (b) DFA A' s.t. $\mathcal{L}(A) = \mathcal{L}(A')$	30
3.1	μ JS syntax.	36
3.2	μ JS implicit type conversion functions.	37
3.3	Example of <code>if</code> CFG.	40
3.4	Example of <code>while</code> CFG.	40
3.5	Ty abstract domain for μ JS.	43
4.1	(a) Type abstract domain for PHP. (b) Complete shell of type abstract domain w.r.t. the sum operation.	48
4.2	(a) SAFE, (b) TAJs string abstract domains recasted for μ JS.	49
4.3	SAFE concat abstract semantics.	50
4.4	Absolute complete shell of ρ_{SF} w.r.t. <code>concat</code>	51
4.5	TAJS numerical abstract domain.	52
4.6	Complete shell of ρ_{TJ} relative to ρ_{TJ_N} w.r.t. <code>toNum</code>	53
5.1	(a) A_1 (b) A_2 (c) $\text{Min}(A_1 \sqcup_{\text{DFA}} A_2)$	60
5.2	(a) A_1 , $\mathcal{L}(A_1) = \{\epsilon, a\}$ (b) A_2 , $\mathcal{L}(A_2) = \{a, aa\}$ (c) $A_1 \nabla_{\text{DFA}/=}^1 A_2$	63
6.1	Coalesced sum abstract domain for μ JS	72
6.2	(a) A_1 , $\mathcal{L}(A_1) = \{abc, hello\}$. (b) A_2 , $\mathcal{L}(A_2) = \{abc, hello\} \cup \{(abb)^n \mid n > 0\}$	78
6.3	(a) A , $\mathcal{L}(A) = \{ddd, abc, bc\}$. (b) A' , $\mathcal{L}(A') = \{bcd, aaab\}$	78
6.4	(a) A , $\mathcal{L}(A) = \{a^n \mid n > 0\} \cup \{b\}$ (b) A' , $\mathcal{L}(A') = \{cd^n \mid n \in \mathbb{N}\}$ (c) $A'' = \text{CC}^\#(A, A')$	79
6.5	A potentially malicious obfuscated JavaScript program.	81
6.6	A_d abstract value of <code>d</code> before <code>eval</code> call of the program in Figure 6.5	81
6.7	FA A_{ds} abstract value of <code>ds</code> at line 15 of Example 6.9.	83
6.8	Analyzer architecture and call execution structure.	84
6.9	FA $A_{ds}^{\text{pStm}} = \text{StmSyn}(A_{ds})$	87
6.10	(a) $\text{CFG}_{\mu\text{JS}}(\{a:=a+1; \mid \mid b:=b+1;\})$, (b) $\text{CFG}_{\mu\text{JS}}(\{(a:=a+1;)\})$	88
6.11	μ JS program of $\{r_{ds}\}$	90
6.12	Control-flow graph G_{ds} generated by <code>CFGGen</code> module	91
6.13	A_{str} s.t. $\mathcal{L}(A_{\text{str}}) = \{x = 5^n; \mid n > 0\}$	94

7.1	Motivating example.	100
7.2	μ JS program example.	103
7.3	(a) Abstract value of o after line 1 of the fragment reported in Figure 7.2 (b) Abstract value of o after line 10. (c) Normal form of o after line 10.	103
7.4	Example of materialization.	106
7.5	(a) μ JS fragment, (b) Value of o after while-loop.	107
7.6	(a) μ JS fragment, (b) Value of o after while-loop.	108

Dedicated to Patrizia, Mattia and Chiara.

Chapter 1

Introduction

Dynamic languages, such as JavaScript, PHP or Python, in the last years have faced an important growth in a very wide range of fields and applications, thanks to the several features that dynamic languages provide and the agility offered by these languages in deploying applications. Dynamic language common features are implicit type conversion, dynamic typing, the multiple usages of strings (e.g., used to access object-properties) and string-to-code statements, only to cite few. In particular, the possibility to transform strings into executable code at run-time is one of the most challenging features to statically analyze, since it breaks the standard assumption in static analysis that the code we aim to analyze is, indeed, static.

In this chapter, we introduce the problem of statically analyzing string manipulation programs, also those that may turn strings into executable code. In particular, we first motivate the need and the importance of analyzing string-to-code statements in dynamic languages, presenting also the difficulties and the challenges that these languages pose from the point of view of static program analysis. Then, we go into details of a particular dynamic language, namely JavaScript, chosen as representative of the class of dynamic languages, giving an overview of its dynamic features that we aim to analyze in this thesis.

The goal of this thesis is to tackle the problem of statically analyzing dynamic code. Hence, in the last part of this chapter, we present an overview of the contribution of this thesis, presenting the several steps we have intended to take in order to reach our goal.

1.1 Why is it important to analyze dynamic code?

String-to-code statements allow developers to transform strings into executable code at run-time. If from the one hand this practice permits developers to simplify writing programs, on the other hand it introduces statically unpredictable executions in deployed applications, which may make programs harder to understand and error-prone. As we have already mentioned before, programs that turn strings into executable code are hard to statically analyze. This happens because of 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. In a sentence: "*You can't check code you don't see*" [Bessey et al., 2010].

Despite several tools proposed in the last years to reason about dynamic code, static analysis is still extremely hard if not even impossible. 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. Hence, when reaching such a statement, an analysis may continue by accepting to lose any previously gathered information. Let us show this situation on a simple but expressive enough JavaScript example. In JavaScript, it is possible to

transform strings into executable code by calling the global function `eval`. Consider the code on the left, where there is a variable `x` independent from what is dynamically executed in `y`.

```

1 x = 1;
2 a = 1;
3 y = "a++;";
4 while (x<10) {
5   y = y + y;
6   eval(y);
7   x++;
8 }
```

Suppose we are interested in analyzing the interval of the variable `x` inside the loop, i.e., at line 5. Executing the code, we can observe that the interval of `x` at line 5 is precisely $[1, 9]$, and this would be the result of any interval analysis on the code without line 6. Unfortunately, the presence of `eval` makes it impossible for the analysis to know whether there is any “hidden” (dynamically generated) modification of `x`, and therefore it cannot properly compute the interval of `x`. This is a very simple use of `eval`, but anyway it is not even suitable for code rewriting techniques removing (when possible) `eval` by replacing it with equivalent code without `eval` [Jensen, Jonsson, and Møller, 2012]. Indeed, in the example, the `eval` parameter is not *hard-coded* but dynamically generated.

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`. Moreover, it should be clear that any analysis of dynamically-generated code cannot be independent from string analysis. Unfortunately, existing static analyzers for dynamic languages, may fail to precisely analyze strings in dynamic contexts. For instance, in the example, existing static analyzers such as TAJIS [Jensen, Møller, and Thiemann, 2009], SAFE [Lee et al., 2012] and JSAI [Kashyap et al., 2014] lose precision on the `eval` input value, losing any information about it. Namely, the issue of analyzing dynamic languages, even if tackled by sophisticated tools as the cited ones, still lacks formal approaches for handling the more dynamic features of string manipulation, such as dynamic code generation.

For all these reasons, we believe that the analysis of dynamic code is not something that can be ignored forever.

In this thesis, we choose a representative of the class of dynamic languages, JavaScript. This choice is guided by the fact that JavaScript is the most popular language and it seems to be the most misunderstood dynamic language. For example, let us consider the 2017 Stack Overflow survey¹. Here, we can observe that JavaScript is, for the seventh year in a row, the most common used programming language (67.8% of the questions posted are tagged with `#javascript`). Moreover, looking at the Stack Overflow ranking of the most common web frameworks and libraries, at the first three positions we find jQuery (first position), React (second position) and AngularJS (third position), all libraries for JavaScript. Even the GitHub surveys confirm the language as the most common and popular programming language, with more than 1.5 million pull requests in 2016 and 323.938 active repositories in 2014².

At the first impact, JavaScript can offer a wide range of features and functionalities, since it supports several programming paradigms, but given its dynamic nature, the resulting applications can open potential security holes, leading to leakage of data or compromise of data integrity. The behavior of JavaScript is provided by the informal ECMA specification that can lead to misunderstandings and unexpected behaviors during the program execution. More dangerous, JavaScript provides string-to-code primitives that allow programmers (but also malicious agents)

¹<https://insights.stackoverflow.com/survey/2019#technology>

²<https://octoverse.github.com/>

to execute arbitrary code in web applications. String-to-code statements constitute a serious security problem and very few solutions have been proposed in order to solve this issue.

In the next section, we go deeper into some of the most unexpected, weird, and critical behaviors of a dynamic language such as JavaScript (according to the formal semantics reported in [Maffeis, Mitchell, and Taly, 2008]) explaining the challenges behind static analysis of JavaScript programs and applications.

1.2 JavaScript overview

JavaScript is a dynamically typed scripting language, born in 1995 as a client-side scripting language to interact at run-time with the HTML object during the web navigation. Nowadays JavaScript is implemented in every major browser and a must-requirement for web developers. The JavaScript interpreter does not provide a *static semantics*, i.e. a type system, and during the program execution it assigns a type to each variable. For example, let `var x = 5` be a typical JavaScript assignment. The type of `x` is not provided by the programmer when the variable is declared but rather it is derived at run-time by the interpreter. Hence, in JavaScript it is legal, for example, to write `x = true` after the previous declaration. Similarly to other typical scripting languages, in JavaScript dynamic typing occurs together with *implicit type conversion*. Let us consider the expression `5 + true`. Obviously the sum operation cannot be performed in this form because of the second operand. Hence, unlike strongly-typed languages (e.g., Java), the interpreter is not stuck with a *type error* but rather it implicitly converts `true` value to `1`, making the sum feasible and returning `6`. Now consider the expression `5 / 0`. Unlike other typical programming languages, the division by zero is allowed in JavaScript and returns the JavaScript global object `Infinity`. This makes the division a *polymorphic* function, since it may return either a number or `Infinity`, in the division-by-zero case. Most of the JavaScript operators and functions have two or more return-types and this may compromise readability and data consistency [Pradel and Sen, 2015]. Implicit type conversion is a key feature of the world of scripting languages since it lightens the development process, often sacrificing code readability and making the code bug-prone.

JavaScript is also an object-oriented programming language based on *prototypal inheritance*. Any object has an internal property called `__proto__` that points to the *constructor prototype*, from which it inherits its properties. The property lookup is performed searching the property on the *prototype chain* and it is very simple: if the object contains the property, the execution returns the value of its property, otherwise it is recursively searched in its prototype object. This operation stops when the root of the prototype chain, i.e., `Object.prototype`, is reached. JavaScript prototypal inheritance makes very easy to perform some object modifications to any object built with a specific constructor, even for the standard built-in objects such as `Number` or `String`, leading programs to be error-prone and more unreadable.

Further, the possibility of dynamically building code instructions as the result of text manipulation is a key aspect in dynamic programming languages such as JavaScript. By using reflection, programs can turn text, which can be built at run-time, into executable code. These features are often used in code protection and tamper resistant applications, employing camouflage for escaping attack or detection, 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

<pre>obj = {a: 20, b: 30}; p = getPropName(); eval("result = obj." + p)</pre> <p style="text-align: center;">(A)</p>	<pre>obj = {a: 20, b: 30}; p = getPropName(); result = obj[p];</pre> <p style="text-align: center;">(B)</p>
---	--

FIGURE 1.1: (a) Object property lookup with `eval`, (b) Object property lookup without `eval`.

generation. In JavaScript it is possible to transform strings into executable code by calling the global function `eval`.

While the use of `eval` 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, data-base corruption, and malware propagation. The most suggested best practice for JavaScript-based web application developers is

"The eval function is the most misused feature of JavaScript. Avoid it." [Crockford, 2008]

Indeed, most of the `eval` usages are often not necessary and can be replaced by a more clear and secure JavaScript semantic-equivalent statement, such as in the case of JSON deserialization and library loading [Richards et al., 2011].

For example, let us consider the program reported in Figure 1.1a. The `getPropName` function may return "a" or "b" and then the code stores into the variable `result` the value of the `obj` property calling the function `eval`. Here the call to `eval` is completely unnecessary and dangerous. The `getPropName` function may retrieve some data from users and then access them into the `eval` code, causing a potential code injection. In this case, JavaScript best practice suggests to avoid `eval` and use the typical object property access, shown in Figure 1.1b.

1.2.1 eval in the wild

An important survey on `eval` usage (and other string-to-code statements such as `setInterval` and `Function`) has been presented in [Richards et al., 2011], showing that `eval` is still popular in benevolent web applications. In the recent years, string obfuscation and the use of `eval` to hide malicious intents have also become very popular in JavaScript malware [Xu, Zhang, and Zhu, 2012]. Its classical usage is depicted in Figure 1.2. Let `NUCLEAR_BOMB` be a function with malicious intents (e.g., creates an `ActiveXObject` to open a shell). In the example, the string value `NUCLEAR_BOMB` is obfuscated and it is manipulated by string manipulation operations in order to obtain the plain string. Finally, when the string is deobfuscated, it is transformed into executable code by `eval`, in order to perform the attack. We wonder: *How popular is eval in JavaScript malware in order to hide malicious actions?* In order to answer this question, we have performed a simple investigation on a JavaScript malware collection [Petraek, 2018]. We have focused on the benchmark related to malware collected in the year 2017, consisting of 192 JavaScript malware samples. Focusing on the explicit `eval` calls (i.e., explicit in the program code and not obfuscated) we have found that:

- 52% of the samples have no explicit `eval` calls;

```
s = "qxfohdu_erpe()";
i = 0; f = "";

while(i++ < s.length) {
  c = s.charAt(i);
  if(!"(_)".includes(c)){
    c = String.fromCharCode(
      c.charCodeAt(0) - 3);
  }
  f += c;
}

eval(f.toUpperCase()); // NUCLEAR_BOMB()
```

FIGURE 1.2: JavaScript obfuscated malware example.

- 33% of the sample have 1 explicit eval call;
- 15% of the samples have >1 explicit eval calls.

Further, we have analyzed dynamic eval calls (i.e., potentially obfuscated eval calls), discovering that 23.5% of the JavaScript samples has at least an obfuscated eval call. Summarizing, more than 50% of the JavaScript samples has at least an eval call (implicit or explicit). Moreover, even if it is not a common practice to use nested eval calls, we have found that a subset of this benchmark uses this strategy to highly obfuscate malicious intents (almost the 10%).

In this simple analysis of eval usage in JavaScript malware, it should be clear that using string-to-code primitives to hide a malicious attack is a common practice and the lack of static analyses of these kind of statements could increase the trend. Moreover, eval (dynamic code in general) can be used also for protecting benevolent code: there already exist obfuscator tools³ that may transform a *removable* eval (replaceable with an eval-free equivalent code fragment) [Jensen, Jonsson, and Møller, 2012] in an eval that can not be removed, providing also malware with powerful obfuscation techniques against existing JavaScript analyzers.

1.3 Contributions and structure of the thesis

In this thesis, we focus on the problem of statically analyzing the dynamic features concerning strings reported in previous sections. The contribution of this thesis is twofold:

- We first focus on the problem of statically reasoning about program strings properties. In particular, we first present the string analyses of existing abstract interpretation-based static analyzers, systematically improving the precision of their string abstractions w.r.t. an operation of interest. Then, we will discuss the need of designing a novel string abstract domain, preparing the ground for the core contribution of this thesis, namely the eval analysis. We aim at a strings abstraction collecting, as faithfully as possible, the set of possible values that a string variable may receive before eval executes it. It surely

³<https://www.daftlogic.com/projects-online-javascript-obfuscator.htm>,
<http://www.danstools.com/javascript-obfuscate/>

has to approximate the set of possible string values, hence it has to be a language, it has also to keep enough information allowing us to extract code from it, but it has also to keep enough information for analyzing properties of string variables that are never executed by an `eval` during computation. In particular, we will formally present the finite state automata abstract domain. Then, we focus on the characterization of the abstract semantics of the most common string manipulation operations, inspired by the JavaScript semantics, taking into account also implicit type conversion. Since strings plays a crucial role also in objects, in dynamic languages, we will exploit finite state automata domain also in analyzing objects.

- Then, we exploit the finite state automata abstract domain and the corresponding string analysis in order to tackle the problem of soundly analyzing dynamic code. The idea behind our approach is that of treating code as any other dynamic structure that can be statically analyzed by abstract interpretation [Cousot and Cousot, 1977], and to treat the abstract interpreter as any other program function that can be recursively called on a piece of code. In particular, once we have designed the novel string abstract domain, since we have to analyze the code potentially executed by an `eval` call, we need to extract from the abstract argument of `eval` (i.e., from an automaton), an abstraction of the code that this collection may contain. Hence, once we have extracted an approximated code representation from an automaton, the idea is to recursively call the abstract interpreter, for the performed analysis, on this approximated code. The result is a first step towards a static analyzer for dynamic code containing non-trivial usage of `eval` that still have some limitations (as we will explain in the final discussion) but which pose the basis for studying more general solutions to the problem.

Structure of the thesis. In Chapter 2 we introduce the basic mathematical notions needed to understand the work and the notations adopted in this thesis. In particular, we introduce basic notions about ordered algebraic structures, abstract interpretation and finite state automata.

In Chapter 3 we introduce in detail μ JS, the dynamic core language on which we will present the main contributions of the thesis. Its syntax and semantics is inspired by the real JavaScript semantics. Moreover, in this chapter we show how to perform static analysis by abstract interpretation of μ JS programs, analyzing their control-flow graphs. This is the starting point for the abstract interpreter for `eval` that we will introduce in Chapter 6.

In Chapter 4 we present the main string abstractions integrated in real JavaScript static analyzers. In particular, we formally discuss the completeness property w.r.t. some common string operations and we will systematically improve the precision of the abstractions for those operations. At the end of this chapter, we discuss the importance of completeness of string abstractions in abstract interpretation and we motivate the need of a novel string abstract domain for dynamic languages.

In Chapter 5 we present the finite state automata abstract domain, formally defining it and introducing some novel operations on that. This is the core of the value abstract domain that we will use for analyzing μ JS programs.

In Chapter 6 we present a sound abstract interpreter for μ JS programs with `eval`, built upon the finite state automata abstract domain. First, we define the abstract semantics of the string manipulation operations of μ JS, allowing us to analyze string values. Then, upon the string analysis, we define a sound analysis for `eval`. We

evaluate our approach on real-world examples, from a precision point of view, also comparing the interpreter with an existing JavaScript static analyzer.

In Chapter 7 we report the last part of the thesis contribution, exploiting again finite state automata abstract domain for analyzing objects properties. We extend μ JS syntax and semantics with objects, then we formally introduce a novel abstract domain for objects, built upon the finite state automata abstract domain described in Chapter 5.

Finally, in Chapter 8 we conclude the thesis, discussing the main related works and discussing the approach and the contribution of this thesis and its future directions. Long proofs of the main results of the thesis are reported in Appendix A.

Publications. Most of the results presented in the thesis have been already published in the following papers:

- Vincenzo Arceri and Isabella Mastroeni [2019]. “Static Program Analysis for String Manipulation Languages”. In: *Proceedings Seventh International Workshop on Verification and Program Transformation*, Genova, Italy, 2nd April 2019. Ed. by Alexei Lisitsa and Andrei Nemytykh. Vol. 299. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 19–33. DOI: [10.4204/EPTCS.299.5](https://doi.org/10.4204/EPTCS.299.5)
- Vincenzo Arceri et al. [2019]. “Completeness of Abstract Domains for String Analysis of JavaScript Programs”. In: *Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings*, pp. 255–272. DOI: [10.1007/978-3-030-32505-3_15](https://doi.org/10.1007/978-3-030-32505-3_15)
- Vincenzo Arceri, Michele Pasqua, and Isabella Mastroeni [2019]. “An abstract domain for objects in dynamic programming languages”. In: *8th International Workshop on Numerical and Symbolic Abstract Domains - NSAD’19*.
- Vincenzo Arceri and Isabella Mastroeni [2020]. “A sound abstract interpreter for dynamic code”. In: *SAC ’20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*. Ed. by Chih-Cheng Hung et al. ACM, pp. 1979–1988. DOI: [10.1145/3341105.3373964](https://doi.org/10.1145/3341105.3373964)

Chapter 2

Mathematical background

In this chapter, we introduce the basic notions that we will use in this thesis. We will first recall notions for sets, relations and functions. Then, we will present the basic background about ordered sets, following [Bancerek and Rudnicki, 2002], useful for introducing abstract interpretation (that will be presented after), the main formal framework that we will exploit for statically reasoning about programs. Most of the results and definitions are taken from [Cousot and Cousot, 1977; Cousot and Cousot, 1976; Nielson, Nielson, and Hankin, 1999]. Finally, we will present the background about strings, formal languages and finite state automata (taken from [Davis, Sigal, and Weyuker, 1994]), fixing the notation and reporting the main operations on finite state automata that we will use in the subsequent chapters.

2.1 Basic notions and notation

A set is a collection of objects, without ordering. When an object x is a member of a set X we write $x \in X$, otherwise we write $x \notin X$. We can extensionally represent a finite set of elements x_0, x_1, \dots, x_n as $\{x_0, x_1, \dots, x_n\}$. We intensionally represent a subset of a set Y of elements satisfying a property ϕ as $\{x \in Y \mid \phi(x)\}$. We often omit Y when it is clear from the context. The predicate ϕ is a first-order logic predicate with the following notation: \neg (negation), \vee (disjunction), \wedge (conjunction), \Rightarrow (implication), \Leftrightarrow (double-implication/logical equivalence), \forall (universal quantifier), \exists (existential quantifier). The cardinality of a set X , namely the number of elements of X , is denoted by $|X|$ and the empty set is denoted by \emptyset . A set X is subset of Y , denoted by $X \subseteq Y$, when any element of X belong also to Y . The empty set is subset of every set. The set of any possible subset of X is denoted by $\wp(X)$ and it is defined as $\{Y \mid Y \subseteq X\}$. Let X, Y be two sets. Then, the following operations between sets can be performed.

- (Union) $X \cup Y \triangleq \{x \mid x \in X \vee x \in Y\}$;
- (Intersection) $X \cap Y \triangleq \{x \mid x \in X \wedge x \in Y\}$;
- (Set difference) $X \setminus Y \triangleq \{x \mid x \in X \wedge x \notin Y\}$;

We can extend union and intersection to a family of sets \mathcal{X} : $\bigcup \mathcal{X} \triangleq \bigcup_{X \in \mathcal{X}} X \triangleq \{x \mid \exists X \in \mathcal{X}. x \in X\}$ and $\bigcap \mathcal{X} \triangleq \bigcap_{X \in \mathcal{X}} X \triangleq \{x \mid \forall X \in \mathcal{X}. x \in X\}$.

Given a set X , a set $\mathcal{P} \subseteq \wp(X)$ is a partition of X if the following conditions hold: (i) $\forall P \in \mathcal{P}. P \neq \emptyset$, (ii) $\bigcup_{P \in \mathcal{P}} P = X$, (iii) $\forall P_1, P_2 \in \mathcal{P}. P_1 = P_2 \vee P_1 \cap P_2 = \emptyset$. For example, if $X = \{1, 2, 3, 4, 5\}$, the set $\wp = \{\{1\}, \{2, 3\}, \{4, 5\}\}$ is a partition of X .

The cartesian product of two sets X and Y is denoted by $X \times Y$ and it is the set of all pairs where the first element belongs to X and the second one belongs to Y , formally $X \times Y \triangleq \{\langle x, y \rangle \mid x \in X \wedge y \in Y\}$. The cartesian product definition can

be extended to n sets, with $n > 2$, denoted by $X_1 \times X_2 \dots X_n$ and it is defined as $X_1 \times X_2 \times \dots \times X_n \triangleq \{ \langle x_1, x_2, \dots, x_n \rangle \mid x_1 \in X_1 \wedge x_2 \in X_2 \wedge \dots \wedge x_n \in X_n \}$.

A relation R between the sets X_1, X_2, \dots, X_n is a subset of the cartesian product $X_1 \times X_2 \times \dots \times X_n$. The elements x_i , with $i \in [1, n]$ are in relation R if $\langle x_1, x_2, \dots, x_n \rangle \in R$. The relation $R \subseteq X_1 \times X_2$, for some set X_1 and X_2 is called binary and in order to denote the membership of an element to R we write $x_1 R x_2$ and $\langle x_1, x_2 \rangle \in R$ and to denote that an element does not belong to R we write $x_1 \not R x_2$ and $\langle x_1, x_2 \rangle \notin R$. Given two relation $R_1 \subseteq X_1 \times X_2$ and $R_2 \subseteq X_2 \times X_3$ we defined the composition between R_1 and R_2 , denoted by $R_2 \circ R_1$, and defined as $R_2 \circ R_1 \triangleq \{ \langle x_1, x_3 \rangle \mid \exists x_2 \in X_2. \langle x_1, x_2 \rangle \in R_1 \wedge \langle x_2, x_3 \rangle \in R_2 \}$.

A binary relation R on X (i.e., $R \subseteq X \times X$) can have some important properties, listed in the following:

- (Reflexivity) $\forall x \in X. x R x$;
- (Irreflexivity) $\forall x \in X. x \not R x$;
- (Symmetry) $\forall x, y \in X. x R y \Rightarrow y R x$;
- (Anti-symmetry) $\forall x, y \in X. x R y \wedge y R x \Rightarrow x = y$;
- (Transitivity) $\forall x, y, z \in X. x R y \wedge y R z \Rightarrow x R z$
- (Totality) $\forall x, y \in X. x R y \vee y R x$;

A binary relation is an equivalence relation if it is reflexive, symmetric and transitive. An equivalence relation R on X induces a partition on X . Each element of the partition induced by R is called equivalent class, usually denoted by $[x]_R$, for $x \in X$, and it is defined as $[x]_R \triangleq \{ y \in X \mid x R y \}$. Given an equivalence class $[x]_R$, the element x is called representative of the equivalence class. A binary relation is a preorder if it is reflexive and transitive. A binary relation is a partial order if it is reflexive, anti-symmetric and transitive.

A function f from the set X to the set Y is a relation $f \subseteq X \times Y$ such that:

- $\forall x \in X. \exists y \in Y. \langle x, y \rangle \in f$;
- $\langle x, y \rangle \in f \wedge \langle x, y' \rangle \in f \Rightarrow y = y'$;

Hence, a function maps any element of X to a single element of Y . The set of functions from X to Y are denoted by $X \rightarrow Y$ and we denote an element f of it as $f : X \rightarrow Y$. Sometimes we use the lambda notation to refer to a function: $\lambda x. f(x)$. Given $f : X \rightarrow Y$, X and Y are called domain and co-domain of f , respectively. We denote the domain of a function f as $dom(f)$. If an element $y \in Y$ is in relation with $x \in X$ we write $y = f(x)$ and y is called image of x . The composition of a function $f : X \rightarrow Y$ with $g : Y \rightarrow Z$ is the function $g \circ f : X \rightarrow Z$ where $\forall x \in X. g \circ f(x) = g(f(x))$. Given $f : S^n \rightarrow T$, $s \in S^n$ and $i \in [1, n]$, we denote by $f_s^i = \lambda z. f(s[z/i]) : S \rightarrow T$ a generic i -th unary restriction of f . Given a function $f : X \rightarrow X$ we define the iterates of f from $x \in X$ as follows:

$$f^0(x) = x$$

$$f^{n+1}(x) = f^n \circ f(x)$$

A function $f : X \rightarrow Y$ is injective if $\forall x, x' \in X. f(x) = f(x') \Rightarrow x = x'$ and it is also called *one-to-one* function. A function $f : X \rightarrow Y$ is surjective if $\forall y \in Y \exists x \in$

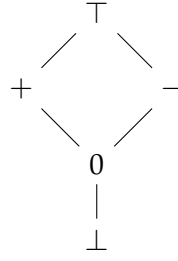


FIGURE 2.1: Example of Hasse diagram.

X . $f(x) = y$ and it is also call *onto* function. A bijective function is a function that is both injective and surjective. Given a function $f : X \rightarrow Y$, its additive lift is the function $\bar{f} : \wp(X) \rightarrow \wp(Y)$ that maps a subset $X' \subseteq X$ to the set of images of elements of X' , that is $\bar{f}(X') \triangleq \{ f(x) \mid x \in X' \}$. In the thesis, we will often abuse notation denoting the additive lift of f on X' with $f(X')$.

A partial function f from a set X to the set Y is a relation $f \subseteq X \times Y$ s.t.

- $\langle x, y \rangle \in f \wedge \langle x, y' \rangle \in f \Rightarrow y = y'$

In partial functions, for some elements of the domain their behavior is not defined. The set of the partial functions from X to Y is denoted by $X \rightharpoonup Y$.

2.2 Posets, semi-lattices and lattices

In the previous section, we have defined a set as a collection of unstructured elements. In this section, we define several algebraic structures embedding orders between the elements contained.

Definition 2.1 (Poset). A set X with a partial order relation \leq_X is said partial order set, denoted by $\langle X, \leq_X \rangle$, and it is called poset.

When it is clear from the context where the order relation \leq_X is defined on, the subscript is omitted. A typical example of poset is $\langle \mathbb{N}, \leq \rangle$, where $\forall x, y \in \mathbb{N}. x \leq y \Leftrightarrow \exists z \in \mathbb{N}. x + z = y$. We can use graphical notation to represent a poset, called *Hasse diagram*. Given a poset $\langle X, \leq \rangle$, each elements of $x \in X$ is represented, in the corresponding Hasse diagram, as a node of the diagram and an edge between $x \in X$ and $y \in X$ if y covers x , that is $y \leq x$ and $\nexists z \in X. x \leq z \wedge z \leq y$. An example of Hasse diagram is reported in Figure 2.1, that corresponds to the poset $\langle P, \leq \rangle$ where $P = \{\perp, 0, +, -, \top\}$ and order \leq is defined as follows: $\perp \leq \perp, \perp \leq 0, \perp \leq +, \perp \leq -, \perp \leq \top, 0 \leq 0, 0 \leq +, 0 \leq -, 0 \leq \top, + \leq +, + \leq \top, - \leq -, - \leq \top, \top \leq \top$. Depending on the type of order, we obtain different types of posets.

Definition 2.2 (Chain). A chain of a poset $\langle P, \leq \rangle$ is a subset $C \subseteq P$ s.t.

$$\forall x, y \in C. x \leq y \vee y \leq x$$

We say that a poset $\langle P, \leq \rangle$ is a chain if P is a chain for $\langle P, \leq \rangle$, namely \leq is a total order. For example $\langle \mathbb{N}, \leq \rangle$, where \leq is the typical order relation between natural, is a chain.

Definition 2.3 (Ascending chain condition). A poset $\langle P, \leq \rangle$ satisfies the ascending chain condition (for short ACC) iff any infinite sequence $p_0 \leq p_1 \leq \dots \leq p_n \leq \dots$ of elements of P is not strictly increasing, that is $\exists k \geq 0. \forall j \geq k. p_k = p_j$.



FIGURE 2.2: (a) Example of join semi lattice and (b) meet semi lattice

Definition 2.4 (Descending chain condition). A poset $\langle P, \leq \rangle$ satisfies the descending chain condition (for short DCC) iff any infinite sequence $p_0 \geq p_1 \geq \dots \geq p_n \geq \dots$ of elements of P is not strictly decreasing, that is $\exists k \geq 0. \forall j \geq k. p_k = p_j$.

Definition 2.5 (Upper bounds, least upper bound, maximum). Let $\langle P, \leq \rangle$ be a poset, and $X \subseteq P$. An element m is an upper bound of X if $\forall x \in X. x \leq m$. If m also belongs to X it is called maximal. If the set of upper bounds of X has the smallest element, we call this element least upper bound of X (lub for short) and it is denoted by $\bigvee X$. If the lub belongs to P , then the lub is called maximum (or top) element.

Given a poset $\langle P, \leq \rangle$ and $X \subseteq P$, for duality, we can define the notion of lower bound, minimal element of a set X , greatest lower bound (glb for short), denoted by $\bigwedge X$, and minimum. We denote the bottom and the top element of a poset by \perp and \top , respectively. It is worth noting that the bottom and the top elements are unique by anti-symmetry property of the order relation. Given two elements $x, y \in P$, we denote by $x \vee y$ and $x \wedge y$ the elements $\bigvee \{x, y\}$ and $\bigwedge \{x, y\}$, respectively.

More complex algebraic structures can be defined, such as the ones defined below.

Definition 2.6 (Directed set). Let $\langle P, \leq \rangle$ be a poset. P is a directed set if $\forall S \subseteq P$ s.t. $S \neq \emptyset$ and S is finite, then S have least upper bound in P .

For duality, we can define when a poset $\langle P, \leq \rangle$ is a co-directed set, namely if $\forall S \subseteq P$ s.t. $S \neq \emptyset$ and S is finite, then S have greatest lower bound in P .

Definition 2.7 (Complete partial order). Let $\langle P, \leq \rangle$ be a poset. P is a complete partial order on directed sets (cpo for short) if $\perp \in P$ and for each directed set D in P we have that $\bigvee D$ exists and $\bigvee D \in P$.

Definition 2.8 (Join semi lattice). A join semi lattice $\langle P, \leq, \vee \rangle$ is a poset $\langle P, \leq \rangle$ such that $\forall x, y \in P$ have lub $x \vee y$.

Definition 2.9 (Meet semi lattice). A meet semi lattice $\langle P, \leq, \wedge \rangle$ is a poset $\langle P, \leq \rangle$ such that $\forall x, y \in P$ have glb $x \wedge y$.

For example, Figure 2.2a reports a join semi lattice and Figure 2.2b reports a meet semi lattice. Merging the definitions of join semi lattice and meet semi lattice we obtain the notion of lattice, as reported below.

Definition 2.10 (Lattice). A lattice $\langle P, \leq, \vee, \wedge \rangle$ is both a join semi lattice and a meet semi lattice.

Given a set X , a typical example of lattice, is $\wp(X)$, namely the powerset of X , where the glb operator is the set intersection and the lub is the set union. When we talk about lattices, it is not guaranteed that the lub or the glb of a subset of it always exist. We can characterize this important property in the definition of complete lattice.

Definition 2.11 (Complete lattice). A complete lattice is a lattice $\langle P, \leq, \vee, \wedge \rangle$ such that $\forall X \subseteq P. \bigvee X \in P$.

For example, given a set X , $\wp(X)$ is a complete lattice. The notion of complete lattice plays a crucial role when we will talk about abstract interpretation in the next sections. Another characterization of the notion of complete lattice is given by the following theorem.

Theorem 2.12. *Let P be a poset s.t. $P \neq \emptyset$. The following assertions are equivalent:*

- P is a complete lattice;
- $\forall X \subseteq P. \bigvee X \in P$;
- P has the top element \top and $\forall S \subseteq P. S \neq \emptyset$, then $\bigwedge S$ exists in P .

In the following, we denote a complete lattice L by $\langle L, \leq, \bigvee, \bigwedge, \top, \perp \rangle$.

Definition 2.13 (Moore family). Let L be a complete lattice. $X \subseteq L$ is a Moore family of L if $\mathcal{M}(X) \triangleq \{ \bigwedge S \mid S \subseteq X \}$, where $\bigwedge \emptyset = \top \in \mathcal{M}(X)$.

For each subset $X \subseteq L$, $\mathcal{M}(X)$ is called Moore closure of X in L , namely $\mathcal{M}(X)$ is the smallest subset of L which contains X and it is a Moore family of L .

Let $\langle P, \leq_P \rangle$ and $\langle Q, \leq_Q \rangle$ be two posets. A function can have interesting properties when they are defined on ordered structures.

- (Monotone) $f : P \rightarrow Q$ is *monotone* if $\forall x, y \in P. x \leq_P y \Rightarrow f(x) \leq_Q f(y)$;
- (Order-embedding) $f : P \rightarrow Q$ is *order-embedding* if $\forall x, y \in P. x \leq_P y \Leftrightarrow f(x) \leq_Q f(y)$;
- (Order-isomorphism) $f : P \rightarrow Q$ is *order-isomorphism* if it is surjective and order-embedding;
- (Extensive) a function $f : P \rightarrow P$ is *extensive* if $\forall x \in P. x \leq_P f(x)$;
- (Reductive) a function $f : P \rightarrow P$ is *reductive* if $\forall x \in P. f(x) \leq_P x$;
- (Idempotent) a function $f : P \rightarrow P$ is *idempotent* if $\forall x \in P. f(f(x)) = f(x)$;
- (Additive) a function $f : P \rightarrow Q$ is *additive* if $\forall X \subseteq P. f(\bigvee_P X) = \bigvee_Q f(X)$
- (Co-additive) a function $f : P \rightarrow Q$ is *co-additive* if $\forall X \subseteq P. f(\bigwedge_P X) = \bigwedge_Q f(X)$

An important property of functions over cpo's is to be (Scott) continuous.

Definition 2.14 (Continuous function). Let $\langle P, \leq_P \rangle$ and $\langle Q, \leq_Q \rangle$ be two cpo's. A function $f : P \rightarrow Q$ is continuous if $\forall D \subseteq P$ directed:

$$f\left(\bigvee_P D\right) = \bigvee_Q f(D) = \bigvee_Q \{ f(d) \mid d \in D \}$$

For duality, a function $f : P \rightarrow Q$ is (Scott) co-continuous if it preserves existing greatest lower bound of co-directed subsets, that is, $\forall D \subseteq P$ co-directed:

$$f\left(\bigwedge_P D\right) = \bigwedge_Q f(D) = \bigwedge_Q \{ f(d) \mid d \in D \}$$

2.3 Fix-point theory

Given a poset $\langle P, \leq \rangle$ and a function $f : P \rightarrow P$, a fix-point of the function f is an element $x \in P$ such that $f(x) = x$. In general, a function may have zero or more fix-points. Moreover, we define the following sets:

- set of fix-points: $\text{fp}(f) \triangleq \{ x \in P \mid f(x) = x \}$;
- set of pre-fix-points: $\text{prefp}(f) \triangleq \{ x \in P \mid x \leq f(x) \}$;
- set of post-fix-points: $\text{postfp}(f) \triangleq \{ x \in P \mid f(x) \leq x \}$;

Note that, by reflexivity, $\text{fp}(f) \subseteq \text{prefp}(f)$ and $\text{fp}(f) \subseteq \text{postfp}(f)$ and, by anti-symmetry, $\text{fp}(f) = \text{prefp}(f) \cap \text{postfp}(f)$. Given a poset $\langle P, \leq \rangle$ and a fix-point x of $f : P \rightarrow P$ is the least fix-point of f if $\forall y \in P. f(y) = y \Rightarrow x \leq y$ and it is the greatest fix-point if $\forall y \in P. f(y) = y \Rightarrow y \leq x$. We denote by $\text{lfp}(f)$ and $\text{gfp}(f)$ the least fix-point and the greatest fix-point of f , respectively. It is important to know, especially when we will talk about static analysis by abstract interpretation, when and if a function admits fix-points. For this reason we recall the following theorem.

Theorem 2.15 (Knaster-Tarski fix-point theorem [Tarski, 1955]). *Let $\langle L, \leq, \vee, \wedge, \perp, \top \rangle$ be a complete lattice $f : L \rightarrow L$ a monotone function. The set of the fix-points of f is a complete lattice. Moreover, the least and greatest fix-points are*

$$\text{lfp}(f) = \bigwedge \text{postfp}(f) \qquad \text{gfp}(f) = \bigvee \text{prefp}(f)$$

Knaster-Tarski theorem guarantees the existence of fix-points but does not give any constructive method about how to compute such fix-points. The following fix-point characterization gives us a constructive method to compute them.

Theorem 2.16 (Kleene fix-point theorem). *Let $\langle D, \leq, \vee, \wedge, \perp, \top \rangle$ be a cpo and $f : D \rightarrow D$ be a continuous function. Then, f has a least fix-point and it is the least upper bound of the following increasing chain.*

$$\perp \leq f(\perp) \leq f^2(\perp) \leq \dots$$

that is $\text{lfp}(f) = \bigvee \{ f^n(\perp) \mid n \in \mathbb{N} \}$.

The above theorem ensures that the least fix-point is reached in at most a countable numbers of steps. The dual result holds for the greatest fix-point, that is, under the dual assumptions, f has the greatest fix-point and it can be computed as follows.

$$\text{gfp}(f) = \bigwedge \{ f^n(\top) \mid n \in \mathbb{N} \}$$

2.4 Galois connections

In this section, we recall important notions of other algebraic structures that play a crucial role in abstract interpretation.

Definition 2.17 (Galois connection). Let $\langle C, \leq_C \rangle$ and $\langle A, \leq_A \rangle$ be two posets and $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$. We say that (C, α, γ, A) is a Galois connection (for short GC) if

$$\forall c \in C, \forall a \in A. \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$$

A GC (C, α, γ, A) is denoted as

$$C \xrightleftharpoons[\alpha]{\gamma} A$$

In this case, α [γ] is called *left adjoint* [*right adjoint*] of γ [α]. They, when we have a GC, it is worth noting that since the adjoints are monotone, preserve the relative precision relation, meaning that if an element of C contains more information of another element of C , then this relation is preserved when the elements are mapped in A by α . Moreover, the GC conditions guarantee the existence of the best abstraction $\alpha(c)$, for each $c \in C$.

Theorem 2.18. *If $C \xleftrightarrow[\alpha_1]{\gamma_1} A$ and $C \xleftrightarrow[\alpha_2]{\gamma_2} A$, then $\alpha_1 = \alpha_2 \Leftrightarrow \gamma_1 = \gamma_2$.*

An important consequence of Theorem 2.18 is that we can refer to a GC between C and A simply by its left adjoint or its right adjoint. Moreover, we can determinate one adjoint knowing the other one.

Proposition 2.19. *Each adjoint can be uniquely determined by using the other one, as follows.*

$$\begin{aligned} \forall c \in C. \alpha(c) &= \bigwedge_A \{ a \in A \mid c \leq_C \gamma(a) \} \\ \forall a \in A. \gamma(a) &= \bigvee_C \{ c \in C \mid \alpha(c) \leq_A a \} \end{aligned}$$

Theorem 2.20. *$C \xleftrightarrow[\alpha]{\gamma} A$ iff α is additive iff γ is co-additive.*

The consequence of the above theorem is that whenever we have an additive (or a co-additive) function between two posets then we have a GC between them. Hence, given an additive function $f : C \rightarrow A$, we can construct the corresponding GC by defining its right adjoint $f^+ \triangleq \lambda a. \bigvee_C \{ c \mid f(c) \leq_A a \}$. Dually, given a co-additive function $f : A \rightarrow C$ we can construct the corresponding GC by defining its left adjoint $f^- \triangleq \lambda c. \bigwedge_A \{ a \mid c \leq_C f(a) \}$.

Definition 2.21 (Galois insertion). *Let $C \xleftrightarrow[\alpha]{\gamma} A$ be a GC. If $\alpha \circ \gamma$ is the identity function, that is $\alpha \circ \gamma = \lambda a. a$, then we have a Galois insertion (GI for short) that it is denoted as follows*

$$C \xleftrightarrow[\alpha]{\gamma} \twoheadrightarrow A$$

When we have a GC, it is always possible to obtain a GI collecting all the elements $a \in A$ having the same image under the function γ . This process is called *reduction*, since, informally speaking, removes all the elements in A that are already "represented" in A by another element. Moreover, given $C \xleftrightarrow[\alpha]{\gamma} A$, the following statements are equivalent:

- $C \xleftrightarrow[\alpha]{\gamma} \twoheadrightarrow A$;
- α is surjective;
- γ is injective;

Given two GCs, it is possible to compose them as stated in the following.

Proposition 2.22. *Let $C \xleftrightarrow[\alpha_1]{\gamma_1} B$ and $B \xleftrightarrow[\alpha_2]{\gamma_2} A$ be two GCs. The composition of them is defined as $C \xleftrightarrow[\alpha_2 \circ \alpha_1]{\gamma_1 \circ \gamma_2} A$.*

It is possible to equivalently define GCs also in terms of *upper closure operators*.

Definition 2.23. Let $\langle P, \leq_P \rangle$ be a poset. A function $\rho : P \rightarrow P$ is an upper closure operator (uco for short) if ρ is extensive, monotone and idempotent.

Dually, we can define lower closure operator (lco for short) requiring that the closure is reductive. Given a GC $C \xleftrightarrow[\alpha]{\gamma} A$, the function $\gamma \circ \alpha$ is an uco on C and $\alpha \circ \gamma$ is a lco on A . This means that α is allowed to lose information when it is mapped to an element of A , but this is forbidden to γ . Given a element $c \in C$, $\alpha(c)$ is the most precise element that represents c , that is, as we have already mentioned before, $\alpha(c)$ is the best abstraction. The next theorem relates the notions of Galois insertion, upper closure operator and Moore family.

Theorem 2.24. Let C and A be two lattices, and $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$. The following statements are equivalent:

- $C \xleftrightarrow[\alpha]{\gamma} A$;
- A is isomorphic to a Moore family of C ;
- if ρ is an uco on C and there exists an isomorphism $\iota : \rho(C) \rightarrow A$ (therefore $\iota^{-1} : A \rightarrow \rho(C)$), then $C \xleftrightarrow[\iota \circ \rho]{\iota^{-1}} A$.

The above theorem tells us that a Galois insertion can be represented also as an upper closure operator on C . Depending on the context of application, it could be more convenient to represent a Galois insertion as an upper closure operator, since the first is related to the representation of the objects of A , namely the names of elements of A , while the latter is not and it talks about the elements of A independently from their names.

2.5 Abstract interpretation

In this section, we introduce the formal notions about abstract interpretation [Cousot and Cousot, 1977; Cousot and Halbwachs, 1978], that is the formal framework that we will use along all this thesis. The first and main application of abstract interpretation is static analysis, and it is used to approximate concrete behaviors of a system into an abstract version of them. Informally speaking, we call the concrete/real behavior of a system *concrete semantics* and an abstract approximation of it is called *abstract semantics*.

Why do we need abstractions? Given a system, or a program, the *meaning*, namely its concrete semantics, can be represented as a mathematical object like a set. This set can be infinite and, in general, it is undecidable to compute any possible behavior of a program. Hence, due to Rice's Theorem, it is undecidable to reason about (non-trivial) program properties on its concrete semantics. The idea behind abstract interpretation is to relate concrete and abstract worlds and to reason in the abstract world implying some reason about the concrete one. Hence, the fact that a certain degree of abstraction is added permits to gain decidability, sacrificing, on the other hand, precision of the observed concrete objects. In particular, abstract interpretation consists in observing the semantics at certain level of abstraction, watching at only the properties of interest and discarding any other concrete detail not interesting or relevant for the property. After that, abstract interpretation permits to calculate, in a

constructive way, the abstract semantics of the system. A crucial property of abstract interpretation is that it does not permit *false negatives*, namely it is sound-by-design: any possible concrete behavior also holds in the abstract.

2.5.1 Concrete objects, abstract objects and Galois connections

Abstract interpretation is a formal framework for approximating mathematical objects. In particular, abstract interpretation does not focus on what the objects are, but rather on the relation between them. Once the relation between concrete objects and abstract objects is established, the idea is to fully transfer computations on abstract objects, with the goal of reasoning about concrete objects.

We denote the concrete object domain as C , that can be a set of numbers, functions, heap locations, etc. The abstract object domain is denoted by A and it is the domain on which the concrete objects are abstracted. Concrete and abstract domains are modeled as posets, namely $\langle C, \leq_C \rangle$ and $\langle A, \leq_A \rangle$. We say that $c \in C$ is more precise than $c' \in C$ if $c \leq c'$. These domains are related by the monotone function $\alpha : C \rightarrow A$ and the function $\gamma : A \rightarrow C$, called *abstraction* and *concretization* functions, respectively. These functions must preserve the relative precision of objects, that is $c \leq c' [a \leq a']$ implies $\alpha(c) \leq_A \alpha(c') [\gamma(a) \leq_C \gamma(a')]$. Anyway, this is not sufficient to guarantee that a concrete object has best abstraction in A . As we have already mentioned in the previous section, this is guaranteed when a Galois connection exists, and this also corresponds to the optimal case of abstract interpretation.

$$\langle C, \leq_C \rangle \begin{array}{c} \xleftarrow{\gamma} \\ \xrightarrow{\alpha} \end{array} \langle A, \leq_A \rangle$$

The Galois connection condition

$$\forall c \in C, a \in A. \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$$

ensures soundness and to have, for each $c \in C$ the best abstraction $\alpha(c)$, namely the most precise correct abstraction in A w.r.t. \leq_A . In this ideal setting, abstract interpretation inherits all the properties about Galois connections. For example, we can determine the abstraction function knowing the concretization function and viceversa (Proposition 2.19) or we can always obtain, from a Galois connection, a Galois insertion, in order to remove the abstract objects that have the same image under the concretization function (i.e., *useless* abstract object).

Example 2.25 (Interval abstract domain [Cousot and Cousot, 1977]). A typical example of abstraction of integers is the interval abstract domain. Informally speaking, the idea of the interval domain is to abstract an integer property (namely an integer set) in the least interval that contains any integer of the property. Given the concrete domain $\wp(\mathbb{Z})$, the interval domain is defined as

$$\text{Ints} \triangleq \{\perp\} \cup \{[a, b] \mid a, b \in \mathbb{Z} \cup \{-\infty, +\infty\}, a \leq b\}$$

where the order \leq is the typical order between integers enriched with $\forall z \in \mathbb{Z}. -\infty \leq z \wedge z \leq +\infty$. The element \perp is the bottom element while the interval $[-\infty, +\infty]$ is the top element. The (partial) order $\sqsubseteq_{\text{Ints}}$ between intervals is defined as:

$$\forall i \in \mathbb{Z}. \perp \sqsubseteq_{\text{Ints}} i \wedge \forall [a, b], [c, d] \in \text{Ints}. [a, b] \sqsubseteq_{\text{Ints}} [c, d] \Leftrightarrow c \leq a \wedge b \leq d$$

The least upper bound operator $\sqcup_{\text{Ints}} : \text{Ints} \times \text{Ints} \rightarrow \text{Ints}$ and the greatest lower bound operator $\sqcap_{\text{Ints}} : \text{Ints} \times \text{Ints} \rightarrow \text{Ints}$ are defined as follows:

$$\begin{aligned} \forall i \in \text{Ints}. \perp \sqcup_{\text{Ints}} i &\triangleq i \wedge i \sqcup_{\text{Ints}} \perp \triangleq i \wedge \\ \forall [a, b], [c, d] \in \text{Ints}. [a, b] \sqcup_{\text{Ints}} [c, d] &\triangleq [\min_{\leq} \{a, c\}, \max_{\leq} \{b, d\}] \end{aligned}$$

$$\forall i \in \text{Ints}. \perp \sqcap_{\text{Ints}} i \triangleq \perp \wedge i \sqcap_{\text{Ints}} \perp \triangleq \perp \wedge \forall [a, b], [c, d] \in \text{Ints}$$

$$\text{Let } i = \max_{\leq} \{a, c\} \text{ and } j = \min_{\leq} \{b, d\} \quad [a, b] \sqcap_{\text{Ints}} [c, d] \triangleq i \leq j ? [i, j] : \perp$$

It is possible to prove that $\langle \text{Ints}, \sqsubseteq_{\text{Ints}}, \sqcup_{\text{Ints}}, \sqcap_{\text{Ints}}, \perp, [-\infty, +\infty] \rangle$ is a complete lattice. Let consider the abstraction function $\gamma_{\text{Ints}} : \text{Ints} \rightarrow \wp(\mathbb{Z})$ defined as follows:

$$\gamma(i) \triangleq \begin{cases} \emptyset & \text{if } i = \perp \\ \{n \in \mathbb{Z} \mid i \leq n \leq j\} & \text{if } i = [i, j] \text{ s.t. } i, j \notin \{-\infty, +\infty\} \\ \{n \in \mathbb{Z} \mid n \geq i\} & \text{if } i = [i, +\infty] \\ \{n \in \mathbb{Z} \mid n \leq i\} & \text{if } i = [-\infty, i] \\ \mathbb{Z} & \text{otherwise} \end{cases}$$

From Proposition 2.19, having γ_{Ints} , we can uniquely determine the corresponding abstraction function $\alpha_{\text{Ints}} : \wp(\mathbb{Z}) \rightarrow \text{Ints}$ in terms of γ_{Ints} . It possible to prove that $\wp(\mathbb{Z}) \xleftrightarrow[\alpha_{\text{Ints}}]{\gamma_{\text{Ints}}} \text{Ints}$.

Interval abstract domain is one of the first numerical domains proposed in abstract interpretation. Several other complex numerical abstract domains have been proposed from the origin of abstract interpretation, for example congruence [Granger, 1989], octagons [Miné, 2006], polyhedra [Chen, Miné, and Cousot, 2008], octahedron [Clariso and Cortadella, 2007] and pentagons [Logozzo and Fähndrich, 2010].

Once the concrete and abstract elements are related, the next step is to move concrete computations on the abstract domain. As we have already mentioned before, in general, a function $f : C \rightarrow C$ is not computable. Hence, we build a computable function $f^\# : A \rightarrow A$ that must correctly approximate f , namely, if we aim at correct abstract computations, it must be *sound*.

Definition 2.26 (Soundness). Consider $\langle C, \leq_C \rangle \xleftrightarrow[\alpha]{\gamma} \langle A, \leq_A \rangle$, a concrete function $f : C \rightarrow C$ and an abstract function $f^\# : A \rightarrow A$. The function $f^\#$ is sound/correct approximation of f in A if the following condition holds:

$$\forall c \in C. \alpha(f(c)) \leq_A f^\#(\alpha(c))$$

or equivalently

$$\forall a \in A. f(\gamma(a)) \leq_C \gamma(f^\#(a))$$

Among all the sound abstract functions $f^\#$ w.r.t. f , we would like to have the best one, that is the one that loses less information computing the abstraction of f . This property is given by the notion of best correct approximation.

Definition 2.27 (Best correct approximation). Given $\langle C, \leq_C \rangle \xleftrightarrow[\alpha]{\gamma} \langle A, \leq_A \rangle$ and a concrete function $f : C \rightarrow C$. The function $\alpha \circ f \circ \gamma : A \rightarrow A$ is the best correct approximation (bca for short) of f in A .

Since the best correct approximation of f depends on the abstraction and concretization functions, it is often useless since it needs to pass through the concrete domain, leading, as we have already mentioned before, to undecidability of the computation.

We can strengthen the soundness conditions, reported in Definition 2.26, by requiring the equality. Doing so, we obtain two notions of completeness.

Definition 2.28 (Completeness). Consider $\langle C, \leq_C \rangle \xleftrightarrow[\alpha]{\gamma} \langle A, \leq_A \rangle$, a concrete function $f : C \rightarrow C$ and an abstract function $f^\# : A \rightarrow A$. We say that:

- $f^\#$ is backward complete for f if: $\forall c \in C. \alpha(f(c)) = f^\#(\alpha(c))$;
- $f^\#$ is forward complete for f if: $\forall a \in A. f(\gamma(a)) = \gamma(f^\#(a))$.

Backward completeness property focuses on complete abstractions of the inputs, while forward completeness focuses on complete abstractions of the outputs, both w.r.t. an operation of interest f . While the notion of backward completeness is well known in abstract interpretation [Arceri et al., 2019], the notion of forward completeness is less known. If the backward completeness property is guaranteed, no loss of information arises during the input abstraction process, w.r.t. an operation of interest. When forward completeness is guaranteed, no loss of information arises during the output abstraction process, w.r.t. an operation of interest.

2.5.2 Fix-point computations

The most interesting operations we can do on (concrete) mathematical objects (numbers, strings, etc.) are the ones expressible as fix-point computations, that is the goal is to compute a least (or greatest) fix-point of a certain function. In general, computing the least fix-point of a function on concrete objects may be undecidable. The goal is, given a complete lattice $\langle C, \leq, \perp, \top, \wedge, \vee \rangle$, a monotone function $f : C \rightarrow C$ and the abstraction $\alpha : C \rightarrow A$ (A is the abstract domain), to compute $\alpha(\text{lfp}f)$ without computing $\text{lfp}f$, that is without passing through concrete computations. Depending on the relation between $\alpha(\text{lfp}f)$ and $\text{lfp}f^\#$ we obtain different desirable properties.

Definition 2.29 (Fix-point soundness and completeness). Consider $C \xleftrightarrow[\alpha]{\gamma} A$, a concrete function $f : C \rightarrow C$ and an abstract function $f^\# : A \rightarrow A$: We say that $f^\#$ is

- fix-point sound if $\alpha(\text{lfp}f) \leq_A \text{lfp}f^\#$;
- fix-point complete if $\alpha(\text{lfp}f) = \text{lfp}f^\#$

Theorem 2.30 (Fix-point approximation). Consider $C \xleftrightarrow[\alpha]{\gamma} A$, a concrete function $f : C \rightarrow C$ and an abstract function $f^\# : A \rightarrow A$. Then the following fact holds:

$$\forall c \in C. \alpha(f(c)) \leq_A f^\#(\alpha(c)) \Rightarrow \alpha(\text{lfp}f) \leq_A \text{lfp}f^\#$$

that is, $f^\#$ is fix-point sound.

Theorem 2.31 (Tarski fix-point transfer). Let consider the complete lattices C and A , the concrete function $f : C \rightarrow C$ and the abstract function $f^\# : A \rightarrow A$ and suppose that f and $f^\#$ are monotone. If there exists an abstraction $\alpha : C \rightarrow A$ s.t. α is co-additive and satisfies $\alpha \circ f \leq_C f^\# \circ \alpha$ and for each post-fix-point $a \in A$ there exists a post-fix-point $c \in C$ s.t. $\alpha(c) = a$, then $\alpha(\text{lfp}f) = \text{lfp}f^\#$.

2.5.3 Fix-point extrapolation and interpolation

In the previous section we have shown how to transfer a least-fix-point computation from the concrete world to the abstract one. Nevertheless, we can still have infinite computations also in the abstract domain. Termination of the fix-point computations is guaranteed, on a complete lattice L either when:

- L is a finite lattice;
- L is ACC (Definition 2.3).

In the absence of one of these properties, fix-point computations may diverge. In this context, these kinds of abstract domains are equipped with an extrapolation operator to enforce convergence and hence termination of fix-point computations.

Definition 2.32 (Widening [Cousot and Cousot, 1977; Cortesi, 2008]). Let $\langle P, \leq \rangle$ be a poset. A widening $\nabla : P \times P \rightarrow P$ satisfies the following conditions:

- $\forall x, y \in P. x \leq (x \nabla y) \wedge y \leq (x \nabla y)$;
- for any increasing chain $x_0 \leq x_1 \leq x_2 \leq \dots$, the following increasing chain

$$\begin{aligned} y_0 &\triangleq x_0 \\ y_{n+1} &\triangleq y_n \nabla x_{n+1} \end{aligned}$$

is not strictly increasing.

Hence, a widening operator is an over-approximation of the least upper bound and is needed to enforce termination of infinite increasing chain. Widening can also be used in converging fix-point computations in order to accelerate convergence when the computation is on too long ascending chains.

Example 2.33 (Interval widening). The interval abstract domain Ints previously reported is not ACC, meaning that fix-point computations may diverge. Hence, Ints is equipped with the widening operator $\nabla_{\text{Ints}} : \text{Ints} \times \text{Ints} \rightarrow \text{Ints}$, introduced in [Cousot and Cousot, 1976], defined as follows.

$$\begin{aligned} \forall i \in \text{Ints}. i \nabla_{\text{Ints}} \perp &\triangleq \perp \nabla_{\text{Ints}} i \triangleq i \\ \forall [a, b], [c, d] \in \text{Ints}. [a, b] \nabla_{\text{Ints}} [c, d] &\triangleq [(c < a ? -\infty : a), (d > b ? +\infty : b)] \end{aligned}$$

It has been proved that ∇_{Ints} satisfies the condition reported in Definition 2.32 [Cousot and Cousot, 1976]. For example, $[0, 1] \nabla_{\text{Ints}} [0, 2] = [0, +\infty]$. It is worth noting that widening operators, in general, are not monotone. This is also the case of the intervals widening ∇_{Ints} .

Next definition shows how to involve widening operator in the (upward) iteration sequence over a poset.

Definition 2.34 (Upward iteration with widening). Let $\langle P, \leq_P \rangle$ a poset, $f : P \rightarrow P$ a function and $\nabla : P \times P \rightarrow P$ a widening. The iteration sequence with ∇ for f , starting from the bottom element $\perp \in P$ is recursively defined as follows, for each $n \in \mathbb{N}$:

$$\begin{aligned} x_0 &\triangleq \perp \\ x_{n+1} &\triangleq \begin{cases} x_n & \text{if } f(x_n) \leq_P x_n \\ x_n \nabla f(x_n) & \text{otherwise} \end{cases} \end{aligned}$$

Any iteration sequence of a function f , equipped with a widening as shown above, is increasing but stationary after a finite number of steps. Moreover, its limit, denoted by x^∇ , is a post-fix-point of f , that is an over-approximation of $\text{lfp}(f)$, namely $\text{lfp}(f) \leq_P x^\nabla$.

Clearly, fix-point computations exclusively equipped with a widening may lead to a drastic loss of information. In order to retrieve some precision lost by the widening, also a interpolation operator, called narrowing, is integrated to improve the precision of over-approximation made by the widening.

Definition 2.35 (Narrowing). Let $\langle P, \leq \rangle$ be a poset. A narrowing $\Delta : P \times P \rightarrow P$ satisfies the following condition:

- $\forall x, y \in P. x \leq y \Rightarrow x \leq (x \Delta y) \leq y$;
- for any decreasing chain $x_0 \geq x_1 \geq x_2 \geq \dots$, the following decreasing chain

$$\begin{aligned} y_0 &\triangleq x_0 \\ y_{n+1} &\triangleq y_n \Delta x_{n+1} \end{aligned}$$

is not strictly decreasing.

Example 2.36 (Interval narrowing). The abstract domain Ints can be also equipped with the narrowing operator $\Delta_{\text{Ints}} : \text{Ints} \times \text{Ints} \rightarrow \text{Ints}$, reported in [Cortesi, 2008], defined as follows.

$$\begin{aligned} \forall i \in \text{Ints}. i \Delta_{\text{Ints}} \perp &\triangleq \perp \Delta_{\text{Ints}} i \triangleq \perp \\ \forall [a, b], [c, d] \in \text{Ints}. [a, b] \Delta_{\text{Ints}} [c, d] &\triangleq [(a = -\infty ? c : a), (b = +\infty ? d : b)] \end{aligned}$$

It had been proved that Δ_{Ints} satisfies the condition reported in Definition 2.35 [Cortesi, 2008]. As for widening, narrowing may be not monotone.

Definition 2.37 (Downward iteration with narrowing). Let $\langle P, \leq_P \rangle$ a poset, $f : P \rightarrow P$ a function and $\Delta : P \times P \rightarrow P$ a narrowing. The iteration sequence with Δ for f , starting from x^∇ is recursively defined as follows, for each $n \in \mathbb{N}$:

$$\begin{aligned} y^0 &\triangleq x^\nabla \\ y^{n+1} &\triangleq \begin{cases} y^n & \text{if } f(y^n) = y^n \\ y^n \Delta f(x^n) & \text{otherwise} \end{cases} \end{aligned}$$

Hence, any iteration sequence, equipped with a narrowing and starting with the post-fix-point x^∇ of f , is decreasing but stationary after a finite number of steps. Moreover, its limit y^∇ is a post-fix-point of f , that is an over approximation of $\text{lfp}(f)$:

$$\text{lfp}(f) \leq y^\nabla \leq x^\nabla$$

A final remark is about the usefulness of infinite domain equipped with widening and narrowing. Among the years, there has always been the feeling that, given an infinite abstract domain with widening and narrowing, it were possible to find an equivalent finite abstract domain (hence without widening and narrowing operators) able to produce the same results. It is trivial to show a counterexample to this, as it had done in [Cousot and Cousot, 1992b], showing that, in general, finite abstract domains are less precise than infinite abstract domains equipped with widening and narrowing.

2.5.4 Abstract domains collectively

In the previous sections, we have presented abstract domains individually. In particular, we have shown that we can characterize abstract domains as adjoints or upper closure operators. In the following, we present the lattice of abstract domains, namely the lattice of any possible abstract domain for a given concrete domain C .

Definition 2.38 (Lattice of abstract domains [Cousot and Cousot, 1977]). Let C a complete lattice. The lattice of abstract domains \mathcal{L}_C of C is

$$\langle \text{uco}(C), \sqsubseteq, \sqcup, \sqcap, \lambda x. \top, \lambda x. x \rangle$$

where $\text{uco}(C)$ is the set of all the possible abstract domains of the concrete domain C . \mathcal{L}_C is a complete lattice, where $\lambda x. x$ is the bottom element and $\lambda x. \top$ is the top element. Let $\rho, \eta \in \text{uco}(C)$, $\{\rho_i\}_{i \in I} \subseteq \text{uco}(C)$. The partial order, least upper bound and greatest lower bound are defined as follows

- $\rho \sqsubseteq \eta \Leftrightarrow \forall c \in C. \rho(c) \sqsubseteq \eta(c)$
- $(\sqcap_{i \in I} \rho_i)(c) = \bigwedge_{i \in I} \rho_i(c)$, for $c \in C$
- $(\sqcup_{i \in I} \rho_i)(c) = c \Leftrightarrow \forall i \in I. \rho_i(c) = c$

The partial order on \mathcal{L}_C corresponds the comparison between abstract domains: indeed A_1 is *more precise* than A_2 , meaning that A_2 is an abstraction of A_1 , if $A_1 \sqsubseteq A_2$. The least upper bound $A_1 \sqcup A_2$ gives as result the most concrete domain that is an abstraction of A_1 and A_2 and it is the least w.r.t. \sqsubseteq while the greatest lower bound $A_1 \sqcap A_2$ is the most abstract domain which is most concrete than A_1 and A_2 .

Several abstract domains combinations have been proposed, such as cartesian product, reduced product [Cousot and Cousot, 1979] or complement [Cortesi et al., 1997]. Further, we will also exploit an abstract domain combination for the abstract interpreter of our core dynamic language, since we will deal with different abstract domains.

2.5.5 Making abstract interpretations complete

We conclude the background about abstract interpretation presenting the interesting work reported in [Giacobazzi, Ranzato, and Scozzari, 2000]. The author of this work proposed a constructive methodology to derive, from a incomplete abstract domain, w.r.t. an operation of interest, a novel abstract domain, that it is complete for that operation. We will use the notions reported in this section in Chapter 4, when we will talk about completeness of string operations and complete domains for JavaScript.

As reported in [Giacobazzi, Ranzato, and Scozzari, 2000], it is worth noting that completeness is a property related to the underlying abstract domain. Starting from this fact, the authors proposed a constructive method to manipulate the underlying incomplete abstract domain in order to get a complete abstract domain w.r.t. a certain operation. In particular, given two abstract domains A and B and an operator $f : A \rightarrow B$, the authors gave two different notions of completion of abstract domains w.r.t. f : the one that *adds* the minimal number of abstract points to the input abstract domain A or the other that *removes* the minimal number of abstract points from the output abstract domain B . The first approach captures the notion of *complete shell* of A , while the latter defines the *complete core* of B , both w.r.t. an operator f .

Complete shell vs complete core. In this thesis, we will focus on the construction of complete shells of abstract domains, rather than complete cores. This choice is guided by the fact that a complete core for an operation f removes abstract points from a starting abstract domain, and so, even if it is complete for f , the complete core could worsen the precision of other operations. On the other hand, complete shells augment the starting abstract domain (adding abstract points), and consequently they cannot compromise the precision of other operations.

Below, we provide two important theorems proved in [Giacobazzi, Ranzato, and Scozzari, 2000] that give a constructive method to compute abstract domain complete shells, defined in terms of an upper closure operator ρ . Precisely, the latter theorems present two notions of complete shells: *i. complete shells of ρ relative to η* (where η is an upper closure operator), meaning that they are complete shells of operations defined on ρ that return results in η , and *ii. absolute complete shells of ρ* , meaning that they are complete shells of operations that are defined on ρ and return results in ρ .

Theorem 2.39 (Complete shell of ρ relative to η). *Let C and D be two posets and $f : C^n \rightarrow D$ be a continuous function. Given $\rho \in \text{uco}(C)$, then $\mathcal{S}_f^\rho : \text{uco}(D) \rightarrow \text{uco}(C)$ is the following domain transformer:*

$$\mathcal{S}_f^\rho(\eta) = \mathcal{M}(\rho \cup (\bigcup_{\substack{i \in [0, n) \\ x \in C^n, y \in \eta}} \max(\{ z \in C \mid (f_x^i)(z) \leq_D y \})))$$

and it computes the complete shell of ρ relative to η .

As already mentioned above, the idea under the complete shell of ρ (input abstraction) relative to η (output abstraction) is to refine ρ adding the minimum number of abstract points to make ρ complete w.r.t. an operator f . From Theorem 2.39, this is obtained adding to ρ the maximal elements in C , whose f image is dominated by elements in η , at least in one dimension i . Clearly, the so-obtained abstraction may not be an upper closure operator for C . Hence, Moore closure operator is applied. On the other hand, absolute complete shells are involved in the case in which the operator f of interest has same input and output abstract domain, i.e., $f : C^n \rightarrow C$. In this case, given $\rho \in \text{uco}(C)$, absolute complete shells of ρ can be obtained as the greatest fix-point of the domain transformer \mathcal{S}_f^ρ (see Theorem 2.39), as stated by the following theorem.

Theorem 2.40 (Absolute complete shell of ρ). *Let C be a poset and $f : C^n \rightarrow C$ be a continuous function. Given $\rho \in \text{uco}(C)$, then $\overline{\mathcal{S}}_f^\rho : \text{uco}(C) \rightarrow \text{uco}(C)$ is the following domain transformer:*

$$\overline{\mathcal{S}}_f^\rho = \text{gfp}(\lambda \eta. \mathcal{S}_f^\rho(\eta))$$

and it computes the absolute complete shell of ρ .

For example, the sign abstract domain is complete for the product operation, but it is not complete w.r.t. the sum. Indeed, the sign of $e_1 + e_2$ cannot be defined by simply knowing the sign of e_1 and e_2 . In [Giacobazzi, Ranzato, and Scozzari, 2000], authors computed the absolute complete shell of the sign domain w.r.t. the sum operation, and they showed it corresponds to the interval abstract domain [Cousot and Cousot, 1977].

Final remarks. In this section, we have reported the background notions about abstract interpretation. As we have already mentioned before, the most popular application of abstract interpretation is static analysis. Since its birth, program static analysis by abstract interpretation have been applied in several research fields, such as for example type checking [Cousot, 1997], information flow [Giacobazzi and Mastroeni, 2018], malware detection [Preda et al., 2008], heap analysis [Balakrishnan and Reps, 2006], termination analysis [Cousot and Cousot, 2012], only to cite few. In Chapter 3, we will introduce the core language that we will use in this thesis and we will explain how to exploit abstract interpretation for static program analysis.

2.6 Strings, languages and finite state automata

A *symbol* is a primitive abstract data type that we do not formally define. Letters, numeric characters and punctuation characters are examples of symbols. An *alphabet* is a finite set of symbols and it is denoted by Σ . A *string* is a sequence of zero or more characters and it is denoted by σ . The empty string is denoted by ϵ . For example, if $\Sigma = \{a, b, c\}$, the string $caaabc$ is a string built upon the alphabet Σ . The *length* of a string is the number of symbols that occurs in σ and it is denoted by $|\sigma|$. For example, $|caaabc| = 6$. The *concatenation* operation is the end-to-end join between two strings. Given two strings σ and σ' its concatenation is denoted by $\sigma \cdot \sigma'$. Sometimes we omit the dot and we refer to the concatenation of two strings by $\sigma\sigma'$. For example, the concatenation of abc and cba is $abccba$. Let $\sigma = \sigma_0\sigma_1 \dots \sigma_n$ be a string. We denote by σ_i the character at the i -th position of σ . Given a string σ , the string $\sigma_0 \dots \sigma_i$, $i \in \{0, \dots, n\}$ is called *prefix* of σ , the string $\sigma_i \dots \sigma_n$, $i \in \{0, \dots, n\}$ is called *suffix* of σ and the string $\sigma_i \dots \sigma_j$, $i, j \in \{0, \dots, n\}$ with $i \leq j \leq |\sigma|$ is called *substring* of σ . For example, if we consider the string $\sigma = \text{helloworld}$, then hel is a prefix, ld is a suffix and low is a substring from i to j of σ . It is worth noting that ϵ is always prefix, suffix and substring of any string. Given an alphabet Σ , the Kleene-closure of Σ , denoted by Σ^* , is the set of any string of finite length over the alphabet Σ . For example, if $\Sigma = \{a, b\}$, then $aaba \in \Sigma^*$. If Σ is not the empty set or composed only by the empty string, then Σ^* is always an infinite set. Moreover, we use the following notations, with $i \in \mathbb{N}$: $\Sigma^i \triangleq \{ \sigma \in \Sigma^* \mid |\sigma| = i \}$ and $\Sigma^{<i} \triangleq \bigcup_{j < i} \Sigma^j$, corresponding to the set of strings of length i and the set of strings of length less than i , respectively.

Definition 2.41 (Formal language). Given an alphabet Σ , a formal language (for short language) \mathcal{L} is a set of strings over Σ , namely $\mathcal{L} \subseteq \Sigma^*$.

For example, $\Sigma = \{a, b, c\}$, then Σ^* , $\{\epsilon\}$, $\{a^n \mid n \in \mathbb{N}\}$ are examples of languages over Σ . The language not containing any string is the *empty language* and it is denoted by \emptyset .

In the following, we define standard language operations. Let Σ be an alphabet and $\mathcal{L}, \mathcal{L}'$ be languages over Σ .

- $\overline{\mathcal{L}} \triangleq \Sigma^* \setminus \mathcal{L}$ is the *complement language* of \mathcal{L} ;
- $\mathcal{L} \cup \mathcal{L}' \triangleq \{ \sigma \mid \sigma \in \mathcal{L} \vee \sigma \in \mathcal{L}' \}$ is the *union language* of \mathcal{L} and \mathcal{L}' ;
- $\mathcal{L} \cap \mathcal{L}' \triangleq \{ \sigma \mid \sigma \in \mathcal{L} \wedge \sigma \in \mathcal{L}' \}$ is the *intersection language* of \mathcal{L} and \mathcal{L}' ;
- $\mathcal{L} \cdot \mathcal{L}' \triangleq \{ \sigma\sigma' \mid \sigma \in \mathcal{L}, \sigma' \in \mathcal{L}' \}$ is the *concatenation language* of \mathcal{L} with \mathcal{L}' .
Note that $\mathcal{L} \cdot \emptyset = \emptyset$ and $\mathcal{L} \cdot \{\epsilon\} = \mathcal{L}$.

- $\mathcal{L} \setminus \mathcal{L}' \triangleq \{ \sigma \mid \sigma \in \mathcal{L} \wedge \sigma \notin \mathcal{L}' \}$ is the *subtraction language* between \mathcal{L} and \mathcal{L}' .

The notions of prefix, suffix and substring can be also lifted from strings to languages, as reported in the following definitions.

Definition 2.42 (Suffixes and prefixes [Davis, Sigal, and Weyuker, 1994]). Let $\mathcal{L} \subseteq \Sigma^*$ be a language. The suffixes of \mathcal{L} are $\text{SU}(\mathcal{L}) \triangleq \{ \sigma \in \Sigma^* \mid \exists \sigma' \in \Sigma^* . \sigma' \sigma \in \mathcal{L} \}$, and the prefixes of \mathcal{L} are $\text{PR}(\mathcal{L}) \triangleq \{ \sigma' \in \Sigma^* \mid \exists \sigma \in \Sigma^* . \sigma \sigma' \in \mathcal{L} \}$.

For example, consider the language $\mathcal{L} = \{abc, def\}$. Then, its suffixes are $\text{SU}(\mathcal{L}) = \{\epsilon, abc, bc, c, f, ef, def\}$ and its prefixes are $\text{PR}(\mathcal{L}) = \{\epsilon, a, ab, abc, d, de, def\}$. Finally, we report two other important operations between languages.

Definition 2.43 (Left quotient [Davis, Sigal, and Weyuker, 1994]). Let $\mathcal{L}, \mathcal{L}' \subseteq \Sigma^*$ be two languages. The left quotient of \mathcal{L} w.r.t. \mathcal{L}' is $\text{LQ}(\mathcal{L}, \mathcal{L}') \triangleq \{ \sigma \in \Sigma^* \mid \exists \sigma' \in \mathcal{L}' . \sigma' \sigma \in \mathcal{L} \}$.

Definition 2.44 (Right quotient [Davis, Sigal, and Weyuker, 1994]). Let $\mathcal{L}, \mathcal{L}' \subseteq \Sigma^*$ be two languages. The right quotient of \mathcal{L} w.r.t. \mathcal{L}' is $\text{RQ}(\mathcal{L}, \mathcal{L}') \triangleq \{ \sigma \in \Sigma^* \mid \exists \sigma' \in \mathcal{L}' . \sigma \sigma' \in \mathcal{L} \}$.

For example, let $\mathcal{L} = \{xab, yac\}$ and $\mathcal{L}' = \{x, y\}$ be two languages. The left quotient of \mathcal{L} w.r.t. \mathcal{L}' is $\text{LQ}(\mathcal{L}, \mathcal{L}') = \{ab, ac\}$. Consider now $\mathcal{L} = \{xab, yab\}$ and $\mathcal{L}' = \{b, ab\}$. The right quotient of \mathcal{L} w.r.t. \mathcal{L}' is $\text{RQ}(\mathcal{L}, \mathcal{L}') = \{xa, ya, x, y\}$.

We introduce a particular subset of suffixes, namely the suffixes starting from a given position. This language will be useful in Chapter 5 when we will define, given a language, its substring language between two indexes.

Definition 2.45 (Suffixes starting from a given position). Let $\mathcal{L} \subseteq \Sigma^*$ be a language and $i \in \mathbb{N}$. The suffixes of \mathcal{L} starting from i are $\text{SU}(\mathcal{L}, i) \triangleq \{ y \in \Sigma^* \mid \exists x \in \Sigma^* . xy \in \mathcal{L}, |x| = i \}$.

For instance, let $\mathcal{L} = \{abc, hello\}$, then $\text{SU}(\mathcal{L}, 2) = \{c, llo\}$. Moreover, we can rewrite the suffixes starting from a position as composition of prefix and left quotient languages, namely $\text{SU}(\mathcal{L}, i) = \text{LQ}(\mathcal{L}, \text{PR}(\mathcal{L}) \cap \Sigma^i)$, as shown below.

$$\begin{aligned}
\text{SU}(\mathcal{L}, i) &= \{ y \in \Sigma^* \mid \exists x \in \Sigma^* . xy \in \mathcal{L}, |x| = i \} && \text{[Def. 2.45]} \\
&= \{ y \in \Sigma^* \mid \exists x \in \Sigma^* . xy \in \mathcal{L}, |x| = i, x \in \text{PR}(\mathcal{L}) \} && \text{[Def. 2.42]} \\
&= \{ y \in \Sigma^* \mid \exists x \in \Sigma^* . xy \in \mathcal{L}, x \in \text{PR}(\mathcal{L}) \cap \Sigma^i \} && \text{[Def. } \Sigma^i \text{]} \\
&= \{ y \in \Sigma^* \mid \exists x \in \text{PR}(\mathcal{L}) \cap \Sigma^i . xy \in \mathcal{L} \} && \text{[Def. of } \cap \text{]} \\
&= \text{LQ}(\mathcal{L}, \text{PR}(\mathcal{L}) \cap \Sigma^i) && \text{[Def. 2.43]}
\end{aligned}$$

It is also worth noting the following fact:

$$\text{SU}(\text{SU}(\mathcal{L}, i)) = \{ y \in \Sigma^* \mid y \in \text{SU}(\mathcal{L}, j), j \geq i \} \quad (2.1)$$

Definition 2.46 (Substrings/Factors [Davis, Sigal, and Weyuker, 1994]). Let $\mathcal{L} \subseteq \Sigma^*$ be a language. The set of its substrings/factors is $\text{FA}(\mathcal{L}) \triangleq \{ \sigma' \in \Sigma^* \mid \exists \sigma, \sigma'' \in \Sigma^* . \sigma \sigma' \sigma'' \in \mathcal{L} \}$

It is worth noting that the set of the factors of \mathcal{L} can be rewritten as composition of prefix and suffix operators, namely $\text{FA}(\mathcal{L}) = \text{PR}(\text{SU}(\mathcal{L}))$.

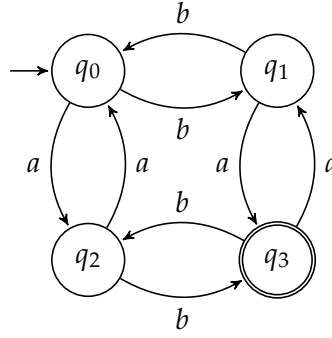


FIGURE 2.3: Example of DFA.

Definition 2.47 (Deterministic finite state automaton). A deterministic finite state automaton (DFA for short) is a tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ such that:

- Q is a finite set of states;
- Σ is an alphabet;
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function;
- $q_0 \in Q$ is the initial state;
- $F \subseteq Q$ is a set of final state;

We can graphically represent a DFA as a graph. An example is reported in Figure 2.3, where edges correspond to the transition function δ , the double-circled state is the final state (q_3) and the state with the incoming transition without the source is the initial state (q_0). The behavior of a DFA is given by the transition function δ , that takes as input a single symbol and a state and returns as output the state reached by reading the input symbol. From the transition function δ , we can uniquely obtain its transitive-closure $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$ as follows:

$$\hat{\delta}(q, \sigma) = \begin{cases} q & \text{if } \sigma = \epsilon \\ \delta(\hat{\delta}(q, \sigma_0 \dots \sigma_{n-1}), \sigma_n) & \text{otherwise} \end{cases}$$

Given $p, q \in Q$, we write $p \rightsquigarrow q$ if there exists a path in the automaton from p to q . A string σ is accepted by a DFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ if $\hat{\delta}(q_0, \sigma) \in F$. Informally speaking, the string is accepted if, reading the string starting from the initial state q_0 , a final state is reached. For example, the string $abbb$ is accepted by the DFA reported in Figure 2.3, whereas the string $aabb$ is not accepted.

Definition 2.48 (Language accepted/recognized by an automaton). Given a DFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$, the language accepted by A , denoted by $\mathcal{L}(A)$, is the set of strings accepted/recognized by A , that is

$$\mathcal{L}(A) \triangleq \{ \sigma \in \Sigma^* \mid \hat{\delta}(q_0, \sigma) \in F \}$$

At this point, we are able to characterize the class of languages recognized by DFA.

Definition 2.49 (Regular language). A language \mathcal{L} is regular if there exists a DFA A such that $\mathcal{L} = \mathcal{L}(A)$.

Algorithm 1: LQ : DFA × DFA → DFA	Algorithm 2: RQ : DFA × DFA → DFA
Data: $A_1, A_2 \in \text{DFA}$, $A_1 = \langle Q_1, \Sigma, \delta_1, q_0^1, F_1 \rangle, A_2 = \langle Q_2, \Sigma, \delta_2, q_0^2, F_2 \rangle$ Result: LQ(A_1, A_2) 1 $I_{LQ} \leftarrow \emptyset$ 2 foreach $q \in Q_1$ do 3 $A_f \leftarrow (Q_1, \Sigma, \delta_1, q_0^1, \{q\});$ 4 if $A_2 \cap A_f \neq \emptyset$ then 5 $I_{LQ} \leftarrow I_{LQ} \cup \{q\};$ 6 end 7 end 8 return Min($\langle Q_1, I_{LQ}, \Sigma, \delta_1, F_1 \rangle$);	Data: $A_1, A_2 \in \text{DFA}$, $A_1 = \langle Q_1, \Sigma, \delta_1, q_0^1, F_1 \rangle, A_2 = \langle Q_2, \Sigma, \delta_2, q_0^2, F_2 \rangle$ Result: RQ(A_1, A_2) 1 $F_{RQ} \leftarrow \emptyset$ 2 foreach $q \in Q_1$ do 3 $A_i \leftarrow (Q_1, q, \Sigma, \delta_1, F_1);$ 4 if $A_2 \cap A_i \neq \emptyset$ then 5 $F_{RQ} \leftarrow F_{RQ} \cup \{q\};$ 6 end 7 end 8 return Min($\langle Q_1, q_0^1, \Sigma, \delta_1, F_{RQ} \rangle$);

FIGURE 2.4: Left quotient and right quotient algorithms.

We denote by DFA the set of deterministic finite state automata. Interesting and desirable properties are guaranteed by regular languages, as reported by the following theorem.

Theorem 2.50 (Regular languages closure properties [Davis, Sigal, and Weyuker, 1994]). *Regular languages are closed under union, intersection, complement, concatenation and factors, right quotient, left quotient, suffix and prefix operations.*

Given $A_1, A_2 \in \text{DFA}$, we denote by $A_1 \cup A_2$ and $A_1 \cap A_2$ the union and the intersection between automata, abusing notation. Moreover, we introduce the operators LQ, RQ : DFA × DFA → DFA, corresponding to the left quotient and right quotient transformers on automata, respectively, and SU, PR : DFA → DFA, corresponding to the suffix and prefix transformers on automata, respectively. In Figure 2.4, we report the left quotient and right quotient algorithms on automata. Algorithm 1 [Davis, Sigal, and Weyuker, 1994] computes the left quotient between two automata, A_1 and A_2 . For each state q of A_1 , we build a new automaton A_f , equal to A_1 , except that the only final state is q (line 3). If A_f recognizes strings of A_2 , i.e., $A_f \cap A_2 \neq \emptyset$, the algorithm collects q in I_{LQ} (lines 4-5). Finally, the result is an automaton equals to A_1 , except that the set of initial states is I_{LQ} . We denote by Min the minimization operation on automata (that we will shortly define). Dually, Algorithm 2 computes the right quotient between two automata.

In Figure 2.5, we reported the suffix and prefix algorithms on automata. In particular, Algorithm 3 [Davis, Sigal, and Weyuker, 1994] computes the suffix automata of A. For each state q , the algorithm checks if there exists a path from q to a final state (line 3). If this happens (line 4), the state q is collected into I_{SU} . Finally, the result is the (minimum) automaton equal to A, except that the set of the initial states is I_{SU} . Dually, Algorithm 4 computes the prefix automata of A. As far as factor automaton is concerned, it can be computed as composition of prefix and suffix operators on automata, as for languages, namely $\text{FA}(\mathcal{L}(A)) = \text{PR}(\text{SU}(A))$. Given $A_1, A_2 \in \text{DFA}$, the following results hold [Davis, Sigal, and Weyuker, 1994].

$$\begin{aligned} \text{RQ}(\mathcal{L}(A_1), \mathcal{L}(A_2)) &= \text{RQ}(A_1, A_2) & \text{LQ}(\mathcal{L}(A_1), \mathcal{L}(A_2)) &= \text{LQ}(A_1, A_2) \\ \text{FA}(\mathcal{L}(A_1)) &= \text{FA}(A_1) & \text{SU}(\mathcal{L}(A_1)) &= \text{SU}(A) & \text{PR}(\mathcal{L}(A_1)) &= \text{PR}(A_1) \end{aligned}$$

Algorithm 3: SU : DFA \rightarrow DFA	Algorithm 4: PR : DFA \rightarrow DFA
Data: $A \in \text{DFA}$, $A = \langle Q, \Sigma, \delta, q_0, F \rangle$	Data: $A \in \text{DFA}$, $A = \langle Q, \Sigma, \delta, q_0, F \rangle$
Result: $\text{SU}(A)$	Result: $\text{PR}(A)$
1 $I_{\text{SU}} \leftarrow \emptyset$	1 $F_{\text{PR}} \leftarrow \emptyset$
2 foreach $q \in Q$ do	2 for $q \in Q$ do
3 if $\exists p \in F. q \rightsquigarrow p$ then	3 if $q_0 \rightsquigarrow q$ then
4 $I_{\text{SU}} \leftarrow I_{\text{SU}} \cup \{q\};$	4 $F_{\text{PR}} \leftarrow F_{\text{PR}} \cup \{q\};$
5 end	5 end
6 end	6 end
7 return $\text{Min}(\langle Q, \Sigma, \delta, I_{\text{SU}}, F \rangle);$	7 return $\text{Min}(\langle Q, \Sigma, \delta, q_0, F_{\text{PR}} \rangle);$

FIGURE 2.5: Suffix and prefix algorithms.

It is worth noting that, in the definition of a DFA, δ is a function. We can extend δ to be a relation, rather than a function, introducing non-determinism in recognizing strings.

Definition 2.51 (Non-deterministic finite state automaton). A non-deterministic finite state automaton (NFA for short) is a tuple $\langle Q, \Sigma, \delta, I, F \rangle$ such that:

- Q is a finite set of states;
- Σ is an alphabet;
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation;
- $I \subseteq Q$ is the set of the initial states;
- $F \subseteq Q$ is a set of final state;

This definition can be further enriched allowing a NFA to read *no symbol*, meaning that when a transition without symbols is met, the automaton directly goes to the next state. This class of finite state automata is called non-deterministic finite state automaton with ϵ -transition (ϵ -NFA). Despite the extension of DFA with non-determinism, the classes of languages both recognized by NFA and ϵ -NFA do not change, as stated by the following theorem.

Theorem 2.52 (Equivalence between DFA, NFA and ϵ -NFA [Hopcroft, Motwani, and Ullman, 2007]). *The classes of languages recognized by DFA, NFA and ϵ -NFA coincide and it is the class of regular languages.*

The above theorem tells us that non-determinism, also enriched with ϵ -transitions, does not add expressiveness to finite state automata. Hence, given a NFA, or a ϵ -NFA, we can obtain an equivalent DFA recognizing the same language. Before providing the algorithm to do that, we introduce three operations used by it:

- ϵ -closure(q): set of states reachable from q with ϵ -transition (included q);
- ϵ -closure(S): set of states reachable from some $q \in S$ with ϵ -transition;
- move(S, a): set of states to which there is a transition labeled with $a \in \Sigma$ from some $q \in S$.

Algorithm 5: Subset construction algorithm

Data: ϵ -NFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$
Result: DFA $A' = \langle Q', \Sigma, \delta', q'_0, F' \rangle$ s.t. $\mathcal{L}(A) = \mathcal{L}(A')$

- 1 $\delta' \leftarrow \emptyset; F' \leftarrow \emptyset;$
- 2 $q'_0 \leftarrow \epsilon\text{-closure}(q_0);$
- 3 $Q' \leftarrow q'_0;$
- 4 **while** there is some unmarked state P in Q' **do**
- 5 mark $P;$
- 6 **foreach** $a \in \Sigma$ **do**
- 7 $R \leftarrow \epsilon\text{-closure}(\text{move}(P, a));$
- 8 **if** $R \notin Q'$ **then**
- 9 $Q' \leftarrow Q' \cup R;$
- 10 **end**
- 11 $\delta' \leftarrow \delta' \cup (P, a, R);$
- 12 **end**
- 13 **end**
- 14 **foreach** $S \in Q'$ **do**
- 15 **if** $S \cap F \neq \emptyset$ **then**
- 16 $F' \leftarrow F' \cup S;$
- 17 **end**
- 18 **end**
- 19 $A' \leftarrow \langle Q', \Sigma, \delta', q'_0, F' \rangle;$
- 20 **return** $A';$

The algorithm to convert a ϵ -NFA A into an equivalent DFA A' is called *subset construction* [Rabin and Scott, 1959] and its pseudo-code is reported in Algorithm 5. The algorithm starts computing the power-state $q'_0 \in \wp(Q)$ (line 2), that merges all the states reachable from the initial state q_0 of A only with ϵ -transition. At line 3, this is the only state of the resulting DFA (line 3). Then, for each unmarked state P in Q' , it is first marked and then, for each symbol a of the alphabet, is computed the ϵ -closure of the power-state, namely R reached by reading the symbol a from any state of P (lines 5-7). If R is not in Q' yet then it is added to it as an unmarked state (lines 8-10) and the transition (P, a, R) is added to δ . This operation is repeated for any computed macro-state of Q' and it terminates when any macro-state is marked. Finally, lines 14-17 build the novel set of final states as follows: for each macro-state $S \in Q'$, if it contains a final state of the NFA A , then S is also a final state of the DFA A' . For example, let consider the ϵ -NFA A reported in Figure 2.6a, that recognizes the language $\{ \{a, b\}^n abb \mid n \in \mathbb{N} \}$. Applying the subset construction algorithm reported in Algorithm 5 we obtain the DFA A' reported in Figure 2.6b, recognizing the same language.

The subset construction algorithm, in the worst case has exponential complexity [Hopcroft, Motwani, and Ullman, 2007], since starting from a ϵ -NFA with n states, the resulting DFA may have 2^n states. Moreover, it is worth noting that Algorithm 5 also works to convert NFA in DFA. We introduce the notation $\text{Det}(A)$ to denote the determinized automaton of A , that implements Algorithm 5.

Theorem 2.53 (Minimum DFA). *For each regular language \mathcal{L} , there uniquely exists a DFA A , with the minimum number of states, s.t. $\mathcal{L} = \mathcal{L}(A)$, modulo state renaming.*

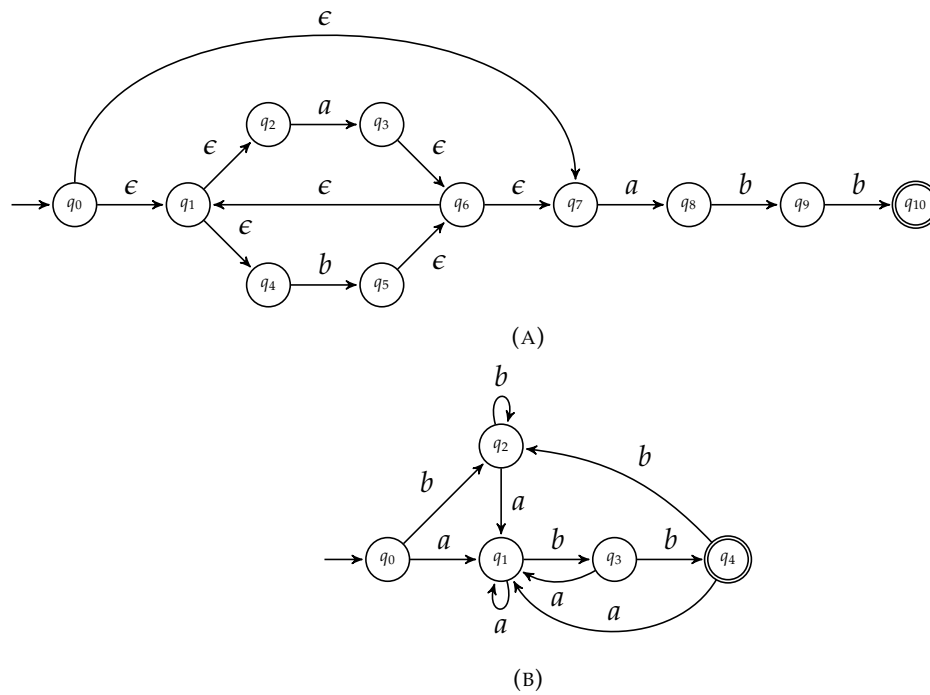


FIGURE 2.6: (a) ϵ -NFA A. (b) DFA A' s.t. $\mathcal{L}(A) = \mathcal{L}(A')$.

Hence, given an automaton A that recognizes a language \mathcal{L} , we can obtain an equivalent DFA A' s.t. $\mathcal{L}(A) = \mathcal{L}(A') = \mathcal{L}$, and A has the minimum number of states. For example, the DFA A reported in Figure 2.6b is also the minimum DFA recognizing the language $\{ \{a, b\}^n abb \mid n \in \mathbb{N} \}$. Several algorithms have been proposed to minimize a DFA. In this thesis, we report two minimization algorithms, namely Hopcroft's algorithm [Hopcroft, 1971] and Brzozowski's algorithm [Brzozowski, 1962]. Before introducing these two approaches, we need to define two classes of states in a non-minimized DFA:

- *unreachable states*: states that cannot be reached starting from the initial state of the DFA;
- *non-distinguishable states*: states that cannot be distinguished from another state of the DFA.

Since unreachable states cannot be reached by q_0 , they cannot recognize any string and they can be removed by the DFA without affecting the recognized language. The algorithm to detect unreachable states is reported in Algorithm 6. The idea is to perform a visit of the DFA, starting from the initial state q_0 , collecting in RS the states met in the visit reading some symbol of the alphabet Σ (lines 3-12). In T are saved the states met while visiting states in NS and are computed in lines 4-9. Then, at line 10, are computed the states that have not been visited yet ($T \setminus RS$) and at line 11 are saved in RS the new reached states. The process at lines 3-12 terminates when no more new reached states are met (line 12). Unreachable states are all the states not met during this visit, namely $Q \setminus RS$ (line 13). Given a DFA A , we denote by $\text{Reach}(A)$ the automaton A without unreachable states.

Hopcroft's minimization algorithm takes care of merging non-distinguishable states of a DFA. The idea behind is to partition the states of a DFA into classes by their behavior. Algorithm 7 represent the pseudo-code of Hopcroft's minimization algorithm. In particular, the algorithm returns as result P , that is the set of equivalence

Algorithm 6: Unreachable states detection algorithm.

Data: DFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$
Result: Set of unreachable states US

```

1 RS  $\leftarrow \{q_0\}$ ;
2 NS  $\leftarrow \{q_0\}$ ;
3 repeat
4   T  $\leftarrow \emptyset$ ;
5   for  $q \in NS$  do
6     foreach  $a \in \Sigma$  do
7       T  $\leftarrow T \cup \{ p \in Q \mid \exists q \in Q. \delta(q, a) = p \}$ ;
8     end
9   end
10  NS  $\leftarrow T \setminus RS$ ;
11  RS  $\leftarrow RS \cup NS$ ;
12 until NS  $\neq \emptyset$ ;
13 US  $\leftarrow Q \setminus RS$ ;
14 return US;
```

classes of states of the input automaton. States in the same equivalence class are undistinguished states (i.e., have the same behavior). Similarly to the subset construction algorithm, the minimum DFA of A can be constructed from the equivalence classes obtained from Algorithm 7, creating a single state for each equivalence class and creating a transition between two equivalence classes S and R when there exists a transition in (s, a, r) in A s.t. $s \in S$ and $r \in R$.

Hopcroft's algorithm works only on DFA. Another minimization algorithm had been proposed by Brzozowski [Brzozowski, 1962] and also works with ϵ -NFA and NFA, as described by Theorem 2.54, where $\text{Reverse}(A)$ denotes the automaton recognizing the reversal language recognized by A .

Theorem 2.54 (Brzozowski's minimization algorithm [Brzozowski, 1962]). *Given a DFA A ,*

$$A' = \text{Reach}(\text{Det}(\text{Reverse}(\text{Reach}(\text{Det}(\text{Reverse}(A)))))$$

is the minimum DFA s.t. $\mathcal{L}(A) = \mathcal{L}(A')$.

The complexity of Brzozowski's algorithm may be exponential in the worst case but in practice performs better than the worst case. Hopcroft's algorithm performs better than the Brzozowski one, since its worst case running time is $O(|\Sigma| |Q| \log(|Q|))$ but its average-case complexity is $O(|Q| \log \log |Q|)$. Given a DFA A , in the thesis, we denote by $\text{Min}(A)$ the minimum DFA of A . Given a language $\mathcal{L} \in \Sigma^*$, we abuse notation denoting with $\text{Min}(\mathcal{L})$ the minimum DFA recognizing \mathcal{L} .

2.6.1 Regular expressions

A different way to characterize regular languages is *regular expressions*, introduced and defined for the first time by Kleene in the 1950s. A regular expression is a sequence of symbols that identifies a pattern, that is the set of strings that fulfill a certain constraint expressed by the regular expression.

Definition 2.55 (Regular expression). Given an alphabet Σ , the following are constant regular expressions:

Algorithm 7: Hopcroft's minimization algorithm.

```

Data: DFA  $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ 
Result:  $P$ , equivalence classes of  $A$ 
1  $P \leftarrow \{F, Q \setminus F\}$ ;
2  $W \leftarrow \{F\}$ ;
3 while  $W \neq \emptyset$  do
4   select  $A$  from  $W$ ;
5    $W \leftarrow W \setminus A$ ;
6   foreach  $a \in \Sigma$  do
7      $X \leftarrow \{q \in Q \mid \delta(q, a) \in A\}$ ;
8     foreach  $Y \in P$  s.t.  $X \cap Y \neq \emptyset$  and  $Y \setminus X \neq \emptyset$  do
9        $P \leftarrow (P \setminus Y) \cup \{X \cap Y\} \cup \{Y \setminus X\}$ ;
10      if  $Y \in W$  then
11         $W \leftarrow (W \setminus Y) \cup \{X \cap Y\} \cup \{Y \setminus X\}$ ;
12      else
13        if  $|X \cap Y| \leq |Y \setminus X|$  then
14           $W \leftarrow W \cup \{X \cap Y\}$ ;
15        else
16           $W \leftarrow W \cup \{Y \setminus X\}$ ;
17        end
18      end
19    end
20  end
21 end
22 return  $P$ ;

```

- (empty set) \emptyset is the empty set denoting the empty set \emptyset ;
- (empty string) ϵ is the empty string denoting $\{\epsilon\}$;
- (character) $a \in \Sigma$ denotes $\{a\}$.

If r and s are regular expressions, the following are also regular expressions

- (concatenation) rs denotes the set containing the strings that are the concatenation between a string in r and a string in s ;
- (disjunction) $r \parallel s$ denotes the set containing the strings in r and s ;
- (Kleene star) r^* denotes the set containing all the strings obtained by concatenating zero or more times strings in r .

To avoid ambiguity, Kleene star has highest priority, then concatenation and then disjunction.

For the sake of clarity, we can use parentheses. For example, given the alphabet $\Sigma = \{a, b\}$, the regular expression $ab^*c \mid a$ denotes the set of strings $\{ab^n c \mid n \in \mathbb{N}\} \cup \{a\}$. Given a regular expression r , we denote by $\mathcal{L}(r)$ the string set denoted by r , and we call it the *language recognized by* r . Given the regular expressions r , s and t , it is easy to prove that they respect the following properties:

- $r \parallel s = s \parallel r$;

- $(r \parallel s) \parallel t = r \parallel (s \parallel t)$;
- $(rs)t = r(st)$;
- $r(s \parallel t) = rs \parallel rt$;
- $(r \parallel s)t = rt \parallel st$;
- $\emptyset^* = \epsilon$;
- $(r^*)^* = r^*$;
- $(\epsilon \parallel rr^*) = (\epsilon \parallel r)^* = r^*$;
- $(r^*s^*)^* = (r \parallel s)^*$

Next theorem tells us that the class of languages recognized by regular expressions is the one of regular languages.

Theorem 2.56 (McNaughton-Yamada [McNaughton and Yamada, 1960]). *Let r be a regular expression. There exists an ϵ -NFA A s.t. $\mathcal{L}(A) = \mathcal{L}(r)$.*

The proof of Theorem 2.56 is recursively done on the structure of a regular expression and also gives an algorithm to transform a regular expression to a ϵ -NFA [McNaughton and Yamada, 1960], and hence to a DFA for Theorem 2.52. In this thesis, we will often refer to a regular language, or a finite state automata, by its corresponding regular expression. Regular expressions will play a crucial role when we will define the analysis of string-to-code statements (i.e., eval) in Chapter 6.

Chapter 3

A dynamic imperative core language: μJS

In this chapter, we define a dynamic imperative core language, that we call μJS , inspired by the JavaScript language. We have decided to adopt a core language rather than a real-world one, such as JavaScript, in order to focus the attention on the main dynamic features we want to analyze and not on other features, that would only make the exposition more complicated. This is the core language that we will use for the rest of the thesis and it is expressive enough to handle implicit type conversion, dynamic typing and string manipulation operations. In the next chapters, we will further augment the syntax and semantics of μJS , with string-to-code statements and object expressions.

In the next, we first define the formal syntax and semantics of μJS , then we introduce the basic notions in order to statically analyze μJS programs by abstract interpretation. This will place the ground for the main contribution of the thesis, that will be presented in the next chapters.

3.1 μJS syntax and semantics

μJS is a dynamic imperative core language and its syntax is reported in Figure 3.1 and it is expressive enough to write arithmetic expressions (AE), boolean expressions (BE) and string expressions (SE). For the sake of simplicity, in the definition of μJS we have not considered some JavaScript features, not related to strings, such as assignment expressions. As far as statements are concerned, we can express no-op statement (`skip`), variable assignments ($x = e$) and `if` and `while` control standard structures. A μJS program is an element of `STMT` rule reported in Figure 3.1, ending with a semicolon. It is worth noting that in μJS any kind of expression can be used also in expressions of a different type. For example, the μJS program `x = 5 + true;` is a legal program. In this sense, μJS takes into account of implicit type conversion [Arceri and Maffei, 2017]. We will shortly explain what implicit type conversion means when we will define the μJS semantics. We denote by `ID` the set of μJS identifiers, ranged over the meta-variable x . Primitive values are represented by the set $\text{VAL} = \text{INT} \cup \text{BOOL} \cup \text{STR} \cup \{\text{NaN}\} \cup \{\uparrow\}$ such that:

- $\text{INT} = \mathbb{Z}$ is the set of signed integers;
- $\text{BOOL} = \{\text{true}, \text{false}\}$ is the set of booleans;
- $\text{STR} = \Sigma^*$ is the set of strings over a fixed alphabet Σ ;
- `NaN` is a special value denoting not-a-number;

$ \begin{aligned} a \in \text{AE} ::= & x \mid n \mid e + e \mid e - e \mid e * e \mid e / e \\ & \mid \text{length}(e) \mid \text{indexOf}(e, e) \\ b \in \text{BE} ::= & x \mid \text{true} \mid \text{false} \mid e \ \&\& \ e \mid e \ \ \ \ e \mid ! \ e \\ & \mid e < e \mid e == e \\ s \in \text{SE} ::= & x \mid "s" \mid \text{substr}(e, e, e) \mid \text{charAt}(e, e) \\ & \mid \text{concat}(e, e) \\ e \in \text{E} ::= & a \mid b \mid s \mid \text{NaN} \mid (e) \\ \text{st} \in \text{STMT} ::= & \text{st} ; \text{st} \mid \text{skip} \mid x = e \mid \text{if} (b) \{ \text{st} \} \text{else} \{ \text{st} \} \\ & \mid \text{while} (b) \{ \text{st} \} \\ P \in \mu\text{JS} ::= & \text{st} ; \\ \text{where } x \in \text{ID} & \text{ (identifiers), } n \in \mathbb{Z} \text{ and } s \in \Sigma^* \end{aligned} $

FIGURE 3.1: μJS syntax.

- \uparrow denotes an error value. In particular, in expression semantics, if an operand evaluates to \uparrow , it is propagated in the evaluation of the main expression (e.g., $\uparrow + 5 = \uparrow$). For the sake of readability, in the following definition of the concrete semantics, these cases are implied.

We denote by $\Sigma_{\text{INT}}^* \subset \text{STR}$ the set of numerical strings, namely strings corresponding to integers. Σ_{INT}^* is defined by the regular expression $\{+, -, \epsilon\} \cdot \{0, 1, \dots, 9\}^+$. Moreover, the function $\mathcal{I} : \Sigma_{\text{INT}}^* \rightarrow \mathbb{Z}$ maps numeric strings to the corresponding integers. Dually, we define the function $\mathcal{S} : \mathbb{Z} \rightarrow \Sigma_{\text{INT}}^*$ that maps each integer to its minimal numeric string representation (e.g., 1 is mapped to the string "1", and not "+1").

Program states $\text{STATE} : \text{ID} \rightarrow \text{VAL}$ are ranged over the meta-variable ζ and are partial functions that maps from identifiers to primitive values. State updates and lookups are defined as follows:

$$\zeta[x \leftarrow v](y) = \begin{cases} v & \text{if } x = y \\ \zeta(y) & \text{otherwise} \end{cases}$$

In order to define the behaviors of the μJS syntax elements, we define its formal big-step semantics. In particular, we define the function $\llbracket P \rrbracket : \text{STATE} \rightarrow \text{STATE}$ that takes as input a μJS program and a program state and returns the output state, containing the *effects* of the input program execution on the input state. The big-step concrete semantics is inductively defined on the structure of STMT . Hence, abusing notation, we define the function $\llbracket \text{st} \rrbracket : \text{STATE} \rightarrow \text{STATE}$ as follows.

$$\begin{aligned}
\llbracket \text{skip} \rrbracket \zeta &= \zeta \\
\llbracket x = e \rrbracket \zeta &= \zeta[x \leftarrow \llbracket e \rrbracket \zeta] \\
\llbracket \text{if}(e) \{ \text{st}_1 \} \text{else} \{ \text{st}_2 \} \rrbracket \zeta &= \begin{cases} \llbracket \text{st}_1 \rrbracket \zeta & \text{if } \text{toBool}(\llbracket e \rrbracket \zeta) = \text{true} \\ \llbracket \text{st}_2 \rrbracket \zeta & \text{if } \text{toBool}(\llbracket e \rrbracket \zeta) = \text{false} \end{cases} \\
\llbracket \text{while}(e) \{ \text{st} \} \rrbracket \zeta &= \text{lfp}(\lambda t. \zeta \cup \llbracket \text{if}(e) \{ \text{st} \} \text{else} \{ \text{skip} \} \rrbracket t)
\end{aligned}$$

$$\begin{aligned}
\text{toString}(v) &= \begin{cases} v & v \in \text{STR} \\ \text{"NaN"} & v = \text{NaN} \\ \text{"true"} & v = \text{true} \\ \text{"false"} & v = \text{false} \\ \mathcal{S}(v) & v \in \text{INT} \end{cases} & \text{toInt}(v) = \begin{cases} v & v \in \text{INT} \\ 1 & v = \text{true} \\ 0 & v = \text{false} \vee v = \text{NaN} \\ \mathcal{I}(v) & v \in \text{STR} \wedge v \in \Sigma_{\text{INT}}^* \\ \text{NaN} & v \in \text{STR} \wedge v \notin \Sigma_{\text{INT}}^* \end{cases} \\
\text{toBool}(v) &= \begin{cases} v & v \in \text{BOOL} \\ \text{true} & v \in \text{INT} \setminus \{0\} \vee v \in \text{STR} \setminus \{\epsilon\} \\ \text{false} & v = 0 \vee v = \epsilon \vee v = \text{NaN} \end{cases}
\end{aligned}$$

FIGURE 3.2: μ JS implicit type conversion functions.

The semantics uses the concrete semantics of expressions, like the assignment rule, defined, abusing notation again, as the function $\llbracket e \rrbracket : \text{STATE} \rightarrow \text{VAL}$, that takes as input an expression and a state and returns the primitive value to which the expression evaluates to in the input state. We will later formally define the expression semantics. All the big-step semantics rules are standard expect for the `if` statement, which contains the first example of *implicit type conversion*, meaning that language operations allow operands of any type and it applies an implicit conversion when a specific type is needed. For example, the statement `if(5+2){ x=1; }else{ x=2; }` is a legal statement in μ JS, but the boolean guard of the `if` statement does not normally evaluate to an boolean value. Hence, when the boolean guard evaluates to a final value, the semantics relies on the function `toBool` that converts the boolean guard final value to a boolean value. In our example, the boolean guard evaluates to 7 and it is implicitly converted to `true`. In order to deal with implicit type conversion, we define the auxiliary functions $\text{toBool} : \text{VAL} \rightarrow \text{BOOL}$, $\text{toInt} : \text{VAL} \rightarrow \text{INT} \cup \{\text{NaN}\}$ and $\text{toString} : \text{VAL} \rightarrow \text{STR}$, that convert a primitive value to a boolean, an integer (if possible) and a string, respectively. Their definitions are reported in Figure 3.2. Note that all the functions behave like the identity when applied to values not needing conversion, e.g., `toInt` on integers. Then, `toString` maps any input value to its string representation; `toInt` returns the integer corresponding to a value, when it is possible: For `true` and `false` it returns respectively 1 and 0, for strings in Σ_{INT}^* it returns the corresponding integer, while all the other values are converted to `NaN`. For instance, $\text{toInt}(\text{"42"}) = 42$, $\text{toInt}(\text{"42hello"}) = \text{NaN}$. Finally, `toBool` returns `false` when the input is `false`, 0, `NaN` or the empty string, and `true` for all the other primitive values.

Expression semantics. We now define the concrete semantics for boolean, arithmetic and string expressions. As we have already mentioned before, the expressions semantics is captured by the function $\llbracket e \rrbracket : \text{STATE} \rightarrow \text{VAL}$. The evaluation of a variable returns the value of the corresponding identifier, if it is defined in the current state, otherwise \uparrow is returned.

$$\llbracket x \rrbracket \xi = \begin{cases} \xi(x) & \text{if } x \in \text{dom}(\xi) \\ \uparrow & \text{otherwise} \end{cases}$$

The semantics for addition, subtraction, multiplication and division is reported below, where $\square \in \{+, -, *, /\}$ and $\blacksquare \in \{+, -, *, /\}$, corresponding to the standard integer operations.

$$\llbracket e_1 \square e_2 \rrbracket \zeta = \begin{cases} \llbracket e_1 \rrbracket \zeta \blacksquare \llbracket e_2 \rrbracket \zeta & \text{if } \text{toInt}(\llbracket e_1 \rrbracket \zeta) \in \text{INT} \wedge \text{toInt}(\llbracket e_2 \rrbracket \zeta) \in \text{INT} \\ \text{NaN} & \text{otherwise} \end{cases}$$

If both evaluations of arithmetic expression operands are implicitly converted to an integer value, then the corresponding standard integer expression is applied to the implicitly converted values, otherwise, NaN is returned. For example, `5 + true` returns 6, while `5 + "hello"` returns NaN, since the string "hello" cannot be converted to an integer value by `toInt`.

The semantics of `length`, reported below, evaluates its input expression, then implicitly converts the result to a string and finally computes its length.

$$\llbracket \text{length}(e) \rrbracket \zeta = |\text{toString}(\llbracket e \rrbracket \zeta)|$$

The semantics of `indexOf` returns the position of the first occurrence of a given substring, if present, otherwise it returns -1. Let us suppose that $\text{toString}(\llbracket e_1 \rrbracket \zeta) = \sigma$ and $\text{toString}(\llbracket e_2 \rrbracket \zeta) = \delta$

$$\llbracket \text{indexOf}(e_1, e_2) \rrbracket = \begin{cases} \min\{i \mid \sigma_i \dots \sigma_j = \delta\} & \exists i, j \in \mathbb{N}. \sigma_i \dots \sigma_j = \delta \\ -1 & \text{otherwise} \end{cases}$$

Logical boolean expressions behave as usual and their semantics is only enriched with implicit type conversion, as follows.

$$\begin{aligned} \llbracket e_1 \ \&\& \ e_2 \rrbracket &= \text{toBool}(\llbracket e_1 \rrbracket \zeta) \ \&\& \ \text{toBool}(\llbracket e_2 \rrbracket \zeta) \\ \llbracket e_1 \ \|\| \ e_2 \rrbracket &= \text{toBool}(\llbracket e_1 \rrbracket \zeta) \ \|\| \ \text{toBool}(\llbracket e_2 \rrbracket \zeta) \\ \llbracket !e \rrbracket &= \neg \text{toBool}(\llbracket e \rrbracket \zeta) \end{aligned}$$

The semantics of less (`<`) and equals (`==`), if both the operands have the same type, returns a result, otherwise an error occurs. Their definitions are reported below.

$$\llbracket e_1 < e_2 \rrbracket \zeta = \begin{cases} \llbracket e_1 \rrbracket \zeta < \llbracket e_2 \rrbracket \zeta & \text{if } \llbracket e_1 \rrbracket \zeta \in \text{INT} \wedge \llbracket e_2 \rrbracket \zeta \in \text{INT} \\ |\llbracket e_1 \rrbracket \zeta| < |\llbracket e_2 \rrbracket \zeta| & \text{if } \llbracket e_1 \rrbracket \zeta \in \text{STR} \wedge \llbracket e_2 \rrbracket \zeta \in \text{STR} \\ \text{toInt}(\llbracket e_1 \rrbracket \zeta) < \text{toInt}(\llbracket e_2 \rrbracket \zeta) & \text{if } \llbracket e_1 \rrbracket \zeta \in \text{BOOL} \wedge \llbracket e_2 \rrbracket \zeta \in \text{BOOL} \\ \uparrow & \text{otherwise} \end{cases}$$

$$\llbracket e_1 == e_2 \rrbracket \zeta = \begin{cases} \llbracket e_1 \rrbracket \zeta == \llbracket e_2 \rrbracket \zeta & \text{if } \llbracket e_1 \rrbracket \zeta \wedge \llbracket e_2 \rrbracket \zeta \text{ have the same type} \\ \uparrow & \text{otherwise} \end{cases}$$

The semantics of `<` successfully evaluates to a primitive value when the operands have the same type: if both operands are integers, then the standard less operation between integer is applied, when they are strings, then the comparison is done between their lengths and when the operands are booleans, they are converted to integers and the integer comparison is applied (e.g., `false < true = 0 < 1 = true`). The behavior of `==` behaves similarly to `<`, since it applies the standard equal operation only when the operands have the same type. When types do not coincide, \uparrow is returned.

As far as string expressions are concerned, μJS includes `substr`, `charAt` and `concat`. Let us first focus on the `substr` semantics and suppose that $\text{toString}(\llbracket e_1 \rrbracket \zeta) = \sigma$, $\text{toInt}(\llbracket e_2 \rrbracket \zeta) = i$ and $\text{toInt}(\llbracket e_3 \rrbracket \zeta) = j$. Before applying the `substr` semantics, two checks are performed on i and j :

1. if i or j are equal to `NaN`, then \uparrow is returned;
2. if i or j are negative integers, then they are treated as 0;

At this point, the semantics of `substr` is applied, after having performed the checks above.

$$\llbracket \text{substr}(e_1, e_2, e_3) \rrbracket \zeta = \begin{cases} \llbracket \text{substr}(\sigma, j, i) \rrbracket \zeta & \text{if } j < i \\ \sigma_i \dots \sigma_j & \text{if } i \leq j < |\sigma| \\ \sigma_i \dots \sigma_{n-1} & \text{if } i \leq j, j \geq |\sigma| = n \end{cases}$$

If the first index is greater than the second one, the indexes are swapped (first case), otherwise if the second index is less than the input string length, the corresponding substring is extracted from σ . Finally, if the second index is greater than the input string length, the suffix starting from i is returned (third case). For instance, $\text{substr}(\text{"abc"}, 1, 4) = \text{"bc"}$.

The semantics of `charAt` is similar to the `substr` one. Let $\text{toString}(\llbracket e_1 \rrbracket \zeta) = \sigma$ and $\text{toInt}(\llbracket e_2 \rrbracket \zeta) = i$.

$$\llbracket \text{charAt}(e_1, e_2) \rrbracket \zeta = \begin{cases} \uparrow & \text{if } i = \text{NaN} \\ \sigma_i & \text{if } i \in \text{INT} \wedge 0 \leq i \leq |\sigma| \\ \epsilon & \text{otherwise} \end{cases}$$

Finally, the semantics of `concat` relies on the standard concatenation operation between strings, as reported by the following semantics.

$$\llbracket \text{concat}(e_1, e_2) \rrbracket \zeta = \text{toString}(\llbracket e_1 \rrbracket \zeta) \text{toString}(\llbracket e_2 \rrbracket \zeta)$$

3.2 Semantics over CFGs and static analysis of μJS

In this section, we focus on the problem of performing static program analysis (by using abstract interpretation), defining a collecting semantics for μJS programs. In particular, we follow the notation and the notions reported in [Seidl, Wilhelm, and Hack, 2012; Miné, 2013], recalling the static analysis process and the necessary semantic transformer corresponding to the statements of μJS . The approach we use is quite standard, but we recall it here for fixing also the notation used in the rest of the thesis.

Given a μJS program P , we suppose to annotate each statement with a label $\ell \in \text{Lab}$, that is not part of the μJS syntax, corresponding to the statement program point in P . Let ℓ_i a special label identifying the initial program point and ℓ_f a special label identifying the final/exit program point. We refer to the labels of P with Lab_P .

In order to analyze a program $P \in \mu\text{JS}$, we have to build a corresponding control flow graph [Seidl, Wilhelm, and Hack, 2012] (CFG for short), which embeds the control structure in the graph and leaves in the blocks (or equivalently on the edges) only the manipulation of the states (assignments) and the guards. We follow [Seidl, Wilhelm, and Hack, 2012] for the construction of the control flow graph, where each

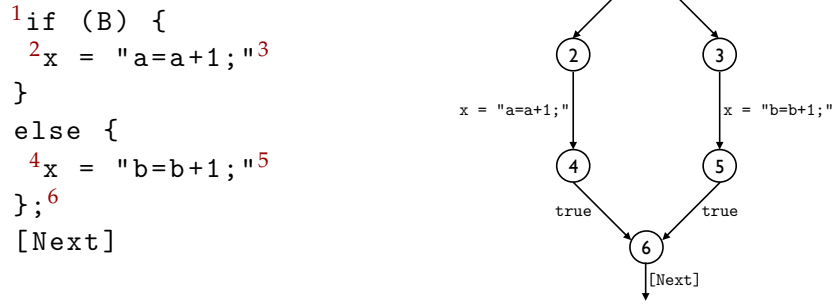


FIGURE 3.3: Example of if CFG.

node is a program point, and each edge is labeled with a statement or a boolean guard. Formally, given a program $P \in \mu\text{JS}$, we define the corresponding CFG $G_P \triangleq \text{CFG}_{\mu\text{JS}}(P) \triangleq \langle \text{Nodes}_P, \text{Edges}_P, \text{In}_P, \text{Out}_P \rangle$ as the CFG whose nodes are the program points, namely $\text{Nodes}_P \triangleq \text{Lab}_P$, the input node (without incoming edges) is the entry program point, i.e., $\text{In}_P \triangleq l_i$, the output node (without outgoing edges) is the last program point, i.e., $\text{Out}_P \triangleq l_f$, and the edges $\text{Edges}_P \in \text{Nodes}_P \times \mu\text{JS-CFG} \times \text{Nodes}_P$ are inductively defined on P by the function

$$\begin{aligned}
\text{Edges}(\ell_1 \text{ skip } \ell_2) &= \{ \langle \ell_1, \text{skip}, \ell_2 \rangle \} \\
\text{Edges}(\ell_1 x = e \ell_2) &= \{ \langle \ell_1, x = e, \ell_2 \rangle \} \\
\text{Edges}(\ell_1 \text{ if}(e) \{ \ell_2 \text{ st}_1 \ell_3 \} \text{ else } \{ \ell_4 \text{ st}_2 \ell_5 \} \ell_6) &= \{ \langle \ell_1, e, \ell_2 \rangle, \langle \ell_1, !e, \ell_4 \rangle, \langle \ell_3, \text{true}, \ell_6 \rangle \} \\
&\quad \cup \{ \langle \ell_5, \text{true}, \ell_6 \rangle \} \\
&\quad \cup \text{Edges}(\ell_2 \text{ st}_1 \ell_3) \cup \text{Edges}(\ell_4 \text{ st}_2 \ell_5) \\
\text{Edges}(\ell_1 \text{ while}(e) \{ \ell_2 \text{ st } \ell_3 \} \ell_4) &= \{ \langle \ell_1, e, \ell_2 \rangle, \langle \ell_1, !e, \ell_4 \rangle, \langle \ell_3, \text{true}, \ell_1 \rangle \} \\
&\quad \cup \text{Edges}(\ell_2 \text{ st } \ell_3) \\
\text{Edges}(\ell_1 \text{ st}_1; \ell_2 \text{ st}_2 \ell_3) &= \text{Edges}(\ell_1 \text{ st}_1 \ell_2) \cup \text{Edges}(\ell_2 \text{ st}_2 \ell_3)
\end{aligned}$$

In Figures 3.3 and 3.4 we report the if statement transformation and while statement transformation to CFG, respectively.

From this construction, it is clear that the language of CFG, namely the grammar of edge labels, is slightly different from μJS , and in particular it is generated by the grammar:

$$\mu\text{JS-CFG} \ni l ::= \text{skip} \mid x = e \mid e$$

At this point, given a CFG G , we denote by $\text{Nodes}(G)$ its set of nodes, by $\text{Edges}(G) \subseteq$

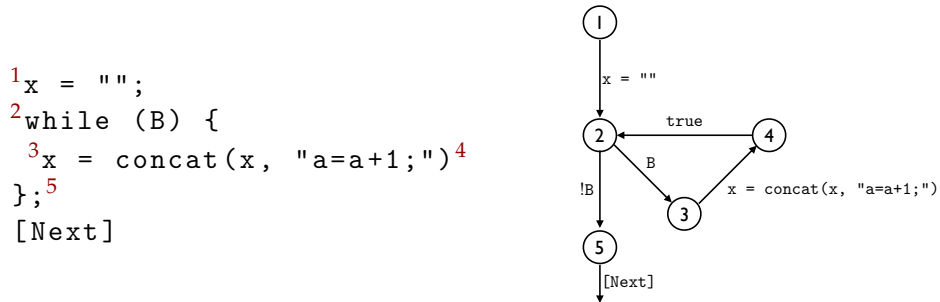


FIGURE 3.4: Example of while CFG.

$Nodes(\mathbb{G}) \times \mu\text{JS-CFG} \times Nodes(\mathbb{G})$ its set of edges, by $In(\mathbb{G})$ its (unique by construction) input node and by $Out(\mathbb{G})$ its (unique by construction) output node. Finally, given a CFG \mathbb{G} , we can define the set of its (finite) paths (from the input node to the exit node). CFG paths can be seen as computations on a CFG, i.e., any possible legal sequence of statements on \mathbb{G} . Formally,

$$Paths(\mathbb{G}) \triangleq \{ l_0 l_1 \dots l_k \mid \forall i \leq k. \langle l_i, l_i, l_{i+1} \rangle \in Edges(\mathbb{G}), l_0 = In(\mathbb{G}), l_{k+1} = Out(\mathbb{G}) \}$$

Note that the CFG reported in Figure 3.3 has only two paths, while the CFG Figure 3.4 has infinite paths, since it contains a cycle (i.e., nodes 2-3-4).

Our aim is to analyze programs in μ JS by analyzing their CFG. Hence, first of all we have to specify the semantic transformation associated with each possible edge of the CFG. In other words, we have to provide the semantics of the edge labels. In particular, we have to formalize how each statement transforms a current state, which is in general represented as a store, namely as an association between identifiers and values. It is well known that static program analysis works computing (abstract) collecting semantics, namely for each program point p and for each variable x , it computes the set of values that the variable x can have in any computation at the program point p . Let $VAL \triangleq \wp(INT) \cup \wp(BOOL) \cup \wp(STR) \cup \wp(\{NaN\}) \cup \wp(\{\uparrow\})$ be the set of the possible collecting values. We define the set of collecting memories $\mathbb{M} \triangleq ID \rightarrow VAL$, ranged over the meta-variable m that associates with each variable a set of values. We define two particular memories, m_\emptyset and m_\top , that associate each variable with \emptyset and each variable with any possible value, respectively. The update of memory m for a variable x with set of values v is denoted by $m[x/v]$. Finally, lub and glb of memories are computed point-wisely, that is $m_1 \sqcup m_2(x) = m_1(x) \cup m_2(x)$ and $m_1 \sqcap m_2(x) = m_1(x) \cap m_2(x)$.

The collecting (input/output) semantics of statements¹ $st \in \mu\text{JS-CFG}$ is defined as the function $\llbracket st \rrbracket : \mathbb{M} \rightarrow \mathbb{M}$. As far as expression semantics is concerned, we abuse notation defining the function $\llbracket e \rrbracket : \mathbb{M} \rightarrow VAL$, defined as additive lift of the previously reported expression semantics. For example, if $m = \{x \mapsto \{1,2\}, y \mapsto \{3,4\}\}$ then, $\llbracket x + y \rrbracket m = \{4,5,6\}$.

$$\begin{aligned} \llbracket skip \rrbracket m &= m \\ \llbracket x = e \rrbracket m &= m[x/\llbracket e \rrbracket m] \\ \llbracket e \rrbracket m &= m \sqcap \bigsqcup \{ m' \mid toBool(\llbracket e \rrbracket m') = \{true\} \} \end{aligned}$$

The semantics of `skip` simply returns the input collecting memory without altering it. The semantics of assignment replaces the value of x in the input memory m with the evaluation of e . The semantics of expressions returns as output the projection of the input memory where the expression is true. This is defined as the glb between the input memory and the set of any memory where e evaluates to `true`. Since μ JS provides implicit type conversion, the evaluation of e is implicitly converted to a boolean by `toBool`, that is the additive lift of the value-to-bool function previously defined on single values. Once we have defined the collecting semantics of a single edge, we can define the collecting semantics of a CFG path of \mathbb{G} . Let $\pi \in Paths(\mathbb{G}_p)$, $\pi = l_0 l_1 \dots l_k$, and $m \in \mathbb{M}$ then $\llbracket \pi \rrbracket m \triangleq \llbracket l_k \rrbracket \circ \dots \circ \llbracket l_1 \rrbracket \circ \llbracket l_0 \rrbracket m$ [Seidl, Wilhelm, and Hack, 2012]. Note that, given a program P , by construction of $\mathbb{G}_p =$

¹The collecting semantics defined in this chapter is an abstraction for the more typical one (i.e., the one collecting sets of concrete memories). Nevertheless, the choice is not restrictive for our purpose and we adopted this semantics for the sake of simplicity.

Algorithm 8: Procedure $\text{ANALYZE}(\mathbb{G}_P, \mathfrak{s}_0)$, static analysis on CFG of P

Data: $\mathbb{G}_P = \langle \text{Nodes}_P, \text{Edges}_P, \text{In}_P, \text{Out}_P \rangle$ and a flow-sensitive input store \mathfrak{s}_0

Result: \mathfrak{s} fix-point of the collecting memories for each program point (result of the analysis)

```

1  $\mathfrak{s} \leftarrow \mathfrak{s}_0$ ;
2  $\mathfrak{s}' \leftarrow \emptyset$ ;
3 while  $\mathfrak{s} \neq \mathfrak{s}'$  do
4    $\mathfrak{s}' \leftarrow \mathfrak{s}$ ;
5   foreach  $\langle \ell_1, st, \ell_2 \rangle \in \text{Edges}_P$  do
6      $\mathfrak{s} \leftarrow \mathfrak{s}[\mathfrak{s}_{\ell_2} / (st) \mathfrak{s}_{\ell_1} \sqcup \mathfrak{s}_{\ell_2}]$ ;
7   end
8 end
9 return  $\mathfrak{s}$ ;

```

CFG $_{\mu\text{JS}}(P)$, it is well known (and it can be easily proved by induction) that

$$\forall m \in \mathbb{M}. \exists \Pi \in \text{Paths}(\mathbb{G}_P). (\downarrow P \downarrow) m = \bigcup_{\pi \in \Pi} (\downarrow \pi \downarrow) m \quad (3.1)$$

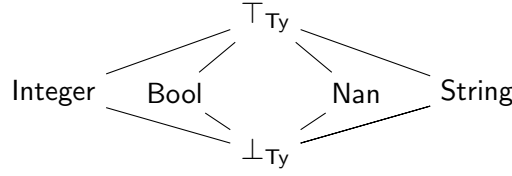
where the $(\downarrow P \downarrow) m$ is the collection of the executions of P on the concrete memories collected in m .

At this point, we use this semantic transformer for analyzing μJS programs by computing the fix-point of the collecting semantics for each program point. In particular, we rewrite the standard fix-point algorithm for static analysis reported in [Nielson, Nielson, and Hankin, 1999] in our notation. First of all we define another important element, which is the collection of stores for each program point, that we will call *flow-sensitive store* $\mathbb{S} \triangleq \text{Lab}_P \rightarrow \mathbb{M}$ associating with each program point a (collecting) memory. Hence, a store $\mathfrak{s} \in \mathbb{S}$ is a sequence of memories, one for each program point. We use \mathfrak{s}_ℓ to denote $\mathfrak{s}(\ell)$, namely the memory at the program point ℓ . Given a store \mathfrak{s} , the update of memory \mathfrak{s}_ℓ with a new memory m is denoted $\mathfrak{s}[\mathfrak{s}_\ell / m]$ and provides a new store \mathfrak{s}' such that $\mathfrak{s}'_\ell = m$ while $\forall \ell' \neq \ell$ we have $\mathfrak{s}'_{\ell'} = \mathfrak{s}_{\ell'}$. Finally, let \mathfrak{s}_\emptyset be the initial flow sensitive store where all the memories associate with all the variables the empty set, i.e., $\forall \ell \in \text{Lab}_P. \mathfrak{s}_\emptyset(\ell) = m_\emptyset$. Then the analysis algorithm is Algorithm 8, whose result is a store \mathfrak{s} such that for each $\ell \in \text{Lab}_P$, we have that \mathfrak{s}_ℓ is the fix-point collecting memory for the program point ℓ .

For instance, let us consider the control-flow graph G reported in Figure 3.3 and let us suppose that the if guard B evaluates to the value set $\{\text{true}, \text{false}\}$, namely it is statically unknown. Applying $\text{ANALYZE}(G, \mathfrak{s}_\emptyset)$ the result is the following flow-sensitive store \mathfrak{s} :

$$\mathfrak{s} = \left\{ \begin{array}{l} \mathfrak{s}_1, \mathfrak{s}_2, \mathfrak{s}_3 \mapsto \mathfrak{s}_\emptyset \\ \mathfrak{s}_4 \mapsto \{x \mapsto \{\text{"a=a+1;"}\}\} \\ \mathfrak{s}_5 \mapsto \{x \mapsto \{\text{"b=b+1;"}\}\} \\ \mathfrak{s}_6 \mapsto \{x \mapsto \{\text{"a=a+1;"}, \text{"b=b+1;"}\}\} \end{array} \right\}$$

In this case, Algorithm 8 converges and returns as result the flow-sensitive store \mathfrak{s} that corresponds to the variables values potentially holding at each program point. In general, it is well-known that Algorithm 8, in presence of loops may diverge on concrete memories, as it happens, for example, for the CFG reported in Figure 3.4. This means that we need abstraction for guaranteeing loop analysis convergence,

FIGURE 3.5: Ty abstract domain for μJS .

as it is usual in static program analysis. In order to show how static program analysis by abstract interpretation based on control-flow graphs works, we consider a particular value abstraction for VAL , that is the *types abstraction* [Cousot, 1997]. The type abstract domain is reported in Figure 3.5, where \sqsubseteq_{Ty} , \sqcup_{Ty} and \sqcap_{Ty} , respectively the partial order, the least upper bound and the greatest lower bound on Ty , can be derived from the Hasse diagram.

The abstraction function $\alpha_{\text{Ty}} : \text{VAL} \rightarrow \text{Ty}$ and concretization function $\gamma_{\text{Ty}} : \text{Ty} \rightarrow \text{VAL}$ are defined as follows.

$$\alpha_{\text{Ty}}(v) \triangleq \begin{cases} \perp_{\text{Ty}} & \text{if } v = \emptyset \vee v = \{\uparrow\} \\ \text{Integer} & \text{if } v \in \wp(\text{INT}) \\ \text{Bool} & \text{if } v \in \wp(\text{BOOL}) \\ \text{String} & \text{if } v \in \wp(\text{STR}) \\ \text{Nan} & \text{if } v = \{\text{NaN}\} \\ \top_{\text{Ty}} & \text{otherwise} \end{cases} \quad \gamma_{\text{Ty}}(a) \triangleq \begin{cases} \{\uparrow\} & \text{if } a = \perp_{\text{Ty}} \\ \wp(\text{INT}) & \text{if } a = \text{Integer} \\ \wp(\text{BOOL}) & \text{if } a = \text{Bool} \\ \wp(\text{STR}) & \text{if } a = \text{String} \\ \{\text{NaN}\} & \text{if } a = \text{Nan} \\ \text{VAL} & \text{if } a = \top_{\text{Ty}} \end{cases}$$

It is trivial to prove that $\text{VAL} \xleftrightarrow[\alpha_{\text{Ty}}]{\gamma_{\text{Ty}}} \text{Ty}$. Given this value abstraction, we can use it to abstract also collecting memories. The set of abstract memories $\mathbb{M}^{\text{Ty}} \triangleq \text{ID} \rightarrow \text{Ty}$, ranged over \mathfrak{m}^{Ty} , associates with each variable the corresponding type. The abstraction function $\alpha_{\mathfrak{m}^{\text{Ty}}} : \mathbb{M} \rightarrow \mathbb{M}^{\text{Ty}}$ is defined as $\alpha_{\mathfrak{m}^{\text{Ty}}}(\mathfrak{m}) \triangleq \{x \mapsto \alpha_{\text{Ty}}(v) \mid x \mapsto v \in \mathfrak{m}\}$, namely abstracts each collecting value (associated with a variable) to its corresponding type, and the corresponding concretization function $\gamma_{\mathfrak{m}^{\text{Ty}}} : \mathbb{M}^{\text{Ty}} \rightarrow \mathbb{M}$ can be derived from α_{Ty} (see Proposition 2.19). We can easily prove that $\mathbb{M} \xleftrightarrow[\alpha_{\mathfrak{m}^{\text{Ty}}}{\gamma_{\mathfrak{m}^{\text{Ty}}}}$.

Similarly, the set of abstract flow-sensitive store $\mathbb{S}^{\text{Ty}} \triangleq \text{Lab} \rightarrow \mathbb{M}^{\text{Ty}}$, ranged over \mathfrak{s}^{Ty} , associates with each program point the corresponding abstract memory. The abstraction function $\alpha_{\mathfrak{s}^{\text{Ty}}} : \mathbb{S} \rightarrow \mathbb{S}^{\text{Ty}}$ abstracts each collecting memory to an abstract memory, namely $\alpha_{\mathfrak{s}^{\text{Ty}}}(\mathfrak{s}) \triangleq \{\ell \mapsto \alpha_{\mathfrak{m}^{\text{Ty}}}(\mathfrak{s}_\ell) \mid \mathfrak{s}_\ell \in \mathfrak{s}\}$ and the corresponding concretization function $\gamma_{\mathfrak{s}^{\text{Ty}}} : \mathbb{S}^{\text{Ty}} \rightarrow \mathbb{S}$ can be derived from $\alpha_{\mathfrak{s}^{\text{Ty}}}$ (see Proposition 2.19). We can easily prove that $\mathbb{S} \xleftrightarrow[\alpha_{\mathfrak{s}^{\text{Ty}}}{\gamma_{\mathfrak{s}^{\text{Ty}}}}$.

The idea is to use the same algorithm reported in Algorithm 8, replacing collecting memories and stores, and the corresponding operations on them, with the abstract versions of them. We denote by $\text{ANALYZE}^{\text{Ty}}$ the Algorithm 8 that uses the type abstract domain. Before doing that, we need to define the abstract semantics of CFG labels, namely $\mu\text{JS-CFG}$.

$$\begin{aligned} \langle \text{skip} \rangle^{\text{Ty}} \mathfrak{m}^{\text{Ty}} &= \mathfrak{m}^{\text{Ty}} \\ \langle x = e \rangle^{\text{Ty}} \mathfrak{m}^{\text{Ty}} &= \mathfrak{m}^{\text{Ty}}[x / \langle e \rangle^{\text{Ty}} \mathfrak{m}^{\text{Ty}}] \\ \langle e \rangle^{\text{Ty}} \mathfrak{m}^{\text{Ty}} &= \mathfrak{m}^{\text{Ty}} \end{aligned}$$

Clearly, we need also to define the abstract semantics of expressions and we abuse

notation denoting it by $\langle e \rangle^{\text{Ty}} : \mathbb{M}^{\text{Ty}} \rightarrow \text{Ty}$ and we suppose that it is defined as the best correct approximation of the collecting semantics (see Definition 2.27).

The abstract semantics of `skip` and assignment is similar to the collecting one. The semantics of expressions, also used in the control-flow graph as boolean guard, simply propagate the input abstract memory. The domain Ty loses any information about the concrete value and it is able only to track the type of each value. In the case of booleans, the abstract semantics is not able to know if the boolean is `true` or `false` and it can only return the input abstract memory (like the expression `e` evaluates both to `true` and `false`).

Finally, we can use this abstraction and the corresponding abstract semantics in Algorithm 8, in order to answer about types of the input μJS program. For example, let us suppose to analyze the CFG reported in Figure 3.4. The algorithm converges and returns as result the following abstract flow-sensitive store \mathfrak{s}^{Ty} :

$$\mathfrak{s}^{\text{Ty}} = \left\{ \begin{array}{l} \mathfrak{s}^{\text{Ty}_1} \mapsto \mathfrak{s}^{\text{Ty}_\emptyset} \\ \mathfrak{s}^{\text{Ty}_2}, \mathfrak{s}^{\text{Ty}_3}, \mathfrak{s}^{\text{Ty}_4}, \mathfrak{s}^{\text{Ty}_5} \mapsto \{x \mapsto \text{String}\} \end{array} \right\}$$

It is possible to prove that Algorithm 8, for any $P \in \mu\text{JS}$ always converges, since Ty is a finite height abstract domain (consequently also \mathbb{M}^{Ty} and \mathbb{S}^{Ty} domains) hence, any fix-point computation converges. Moreover, the abstract store computed by Algorithm 8 on the type abstract domain is sound. Let \mathfrak{s} be the fix-point collecting flow-sensitive store, associating for each program point of a program $P \in \mu\text{JS}$ collecting memories, and $\mathfrak{s}^{\text{Ty}} = \text{ANALYZE}^{\text{Ty}}(\mathbb{G}_P, \alpha_{\mathbb{M}^{\text{Ty}}}(\mathfrak{m}))$. Then $\forall \ell \in \text{Lab}_P. \mathfrak{s}_\ell \subseteq \gamma_{\mathbb{M}^{\text{Ty}}}(\mathfrak{s}^{\text{Ty}}_\ell)$. The type analysis we have shown here is able to track, for each variable program, if it has constant type during program execution.

Unfortunately, the proposed static analysis algorithm may not be sufficient to avoid divergence of the analysis. This may happen, for example, when the abstraction integrated into the analysis is not ACC. For example, let us suppose to use the same value abstract domain reported in Figure 3.5 replacing the integer abstraction with the interval abstract domain Ints described in Example 2.25. We denote this abstract domain by ITy , with the abstraction function defined as

$$\alpha_{\text{ITy}}(v) \triangleq v \in \wp(\mathbb{Z}) ? \alpha_{\text{Ints}}(v) : \alpha_{\text{Ty}}(v)$$

The least upper bound $\sqcup_{\text{ITy}} : \text{ITy} \times \text{ITy} \rightarrow \text{ITy}$ of ITy is defined as follows, and the greatest lower bound $\sqcap_{\text{ITy}} : \text{ITy} \times \text{ITy} \rightarrow \text{ITy}$ can be dually defined.

$$v_1 \sqcup_{\text{ITy}} v_2 \triangleq \begin{cases} v_1 \sqcup_{\text{Ints}} v_2 & \text{if } v_1 \in \text{Ints}, v_2 \in \text{Ints} \\ v_1 \sqcup_{\text{Ty}} v_2 & \text{if } v_1 \notin \text{Ints}, v_2 \notin \text{Ints} \\ v_1 & \text{if } v_2 = \perp_{\text{Ty}} \\ v_2 & \text{if } v_1 = \perp_{\text{Ty}} \\ \top_{\text{Ty}} & \text{otherwise} \end{cases}$$

Similarly to Ty , we can lift this abstraction to memories and stores, obtaining $\mathbb{M}^{\text{ITy}} \triangleq \text{ID} \rightarrow \text{ITy}$, ranged over \mathbb{m}^{ITy} , and $\mathbb{S}^{\text{ITy}} \triangleq \text{Lab} \rightarrow \mathbb{M}^{\text{ITy}}$, ranged over $\mathfrak{s}^{\text{ITy}}$. Now, let us suppose to analyze the following μJS program with Algorithm 8, using the abstraction just defined.

Algorithm 9: Procedure $\text{ANALYZE}(G_P, \mathfrak{s}_0)$, static analysis on CFG of P with widening

Data: $G_P = \langle \text{Nodes}_P, \text{Edges}_P, \text{In}_P, \text{Out}_P \rangle$ and a flow-sensitive input store \mathfrak{s}_0
Result: \mathfrak{s} fix-point of the collecting memories for each program point (result of the analysis)

```

1  $\mathfrak{s} \leftarrow \mathfrak{s}_0$ ;
2  $\mathfrak{s}' \leftarrow \emptyset$ ;
3 while  $\mathfrak{s} \neq \mathfrak{s}'$  do
4    $\mathfrak{s}' \leftarrow \mathfrak{s}$ ;
5   foreach  $\langle \ell_1, \text{st}, \ell_2 \rangle \in \text{Edges}_P$  do
6      $\mathfrak{s} \leftarrow \mathfrak{s}[\mathfrak{s}_{\ell_2} / \mathfrak{s}_{\ell_2} \nabla (\text{st } \mathfrak{s}_{\ell_1})]$ ;
7   end
8 end
9 return  $\mathfrak{s}$ ;

```

```

1  $x = 0$ ;
2 while (B) {
3    $x = x + 2^4$ 
4 }
5

```

Since the value of boolean guard B is statically unknown also the number of the `while`-loop iterations is unknown. Hence, since Ints is not ACC, also ITy is not ACC and the value of x diverges at the program point 2: $[0, 2], [0, 4], [0, 6], \dots$. As we have already mentioned before, we can use a widening operator in order to enforce termination, still guaranteeing soundness. Hence, we introduce the widening operator $\nabla_{\text{ITy}} : \text{ITy} \times \text{ITy} \rightarrow \text{ITy}$ defined as follows.

$$v_1 \nabla_{\text{ITy}} v_2 \triangleq \begin{cases} v_1 \nabla_{\text{Ints}} v_2 & \text{if } v_1 \in \text{Ints}, v_2 \in \text{Ints} \\ v_1 \sqcup_{\text{ITy}} v_2 & \text{otherwise} \end{cases}$$

The widening defined above applies the interval widening when both operands are intervals, while it behaves as its least upper bound in the other cases, since the source of divergence is the interval abstract domain, where a widening is needed. Given ∇_{ITy} , we can build upon it the widenings $\nabla_{\text{m}^{\text{ITy}}} : \mathbb{M}^{\text{ITy}} \times \mathbb{M}^{\text{ITy}} \rightarrow \mathbb{M}^{\text{ITy}}$ and $\nabla_{\mathfrak{s}^{\text{ITy}}} : \mathfrak{S}^{\text{ITy}} \times \mathfrak{S}^{\text{ITy}} \rightarrow \mathfrak{S}^{\text{ITy}}$, namely the widening on abstract memories and abstract stores, respectively.

$$\begin{aligned} \text{m}^{\text{ITy}} \nabla_{\text{m}^{\text{ITy}}} \text{m}^{\text{ITy}'} &\triangleq \lambda x \in \text{ID}. [x \mapsto \text{m}^{\text{ITy}}(x) \nabla_{\text{ITy}} \text{m}^{\text{ITy}'}(x)] \\ \mathfrak{s}^{\text{ITy}} \nabla_{\mathfrak{s}^{\text{ITy}}} \mathfrak{s}^{\text{ITy}'} &\triangleq \lambda \ell \in \text{Lab}. \mathfrak{s}^{\text{ITy}}(\ell) \nabla_{\text{m}^{\text{ITy}}} \mathfrak{s}^{\text{ITy}'}(\ell) \end{aligned}$$

At this point, we need to integrate the widening operator on abstract store on the already presented Algorithm 8, in order to obtain an algorithm able to guarantee convergence of any μ JS program, even in presence of abstract domains that are not ACC. This algorithm is reported in Algorithm 9 and it is defined independently from the abstraction: it is enough to substitute the abstraction-related operations (e.g., widening ∇) in Algorithm 9 in order to obtain the abstract interpreter using the desired abstraction. What changes w.r.t. Algorithm 8 is line 6: when an edge $\langle \ell_1, \text{st}, \ell_2 \rangle$ is processed and analyzed, the resulting memory should be put at ℓ_2 , lubbing it with the previous memory at ℓ_2 . Instead, we substitute least upper bound with widening, that is the previous memory holding at ℓ_2 is widened with the resulting memory of

# iteration \ Line	I	II	III	IV	V	VI	VII
1	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
2	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, +\infty]$	$[0, +\infty]$	$[0, +\infty]$	$[0, +\infty]$
3	\emptyset	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, +\infty]$	$[0, +\infty]$	$[0, +\infty]$
4	\emptyset	\emptyset	$[2, 2]$	$[2, 2]$	$[2, 2]$	$[2, +\infty]$	$[2, +\infty]$
5	\emptyset	$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, +\infty]$	$[0, +\infty]$	$[0, +\infty]$

TABLE 3.1: Example of application of Algorithm 9.

$(\text{st})_{\mathfrak{S}_{\ell_1}}$.

For example, let us consider the μ JS divergent program previously reported and let's analyze its corresponding control-flow graph with Algorithm 9. In Table 3.1 we report the iterations of Algorithm 9, showing what is computed at each iteration for each program point. In particular, the algorithm converges after seven iterations, since the fix-point it is reached.

Final remarks. In this chapter, we have introduced our core dynamic language, namely μ JS. As we have shown, the language is expressive enough to write programs with some important dynamic features, such as implicit type conversion and dynamic typing. After that, we have recalled and provided the basic background of static program analysis by abstract interpretation (in particular analyzing the control-flow graph of a program), showing static analysis process on two different abstract domains, a finite abstract domain and an infinite abstract domain.

This chapter places the ground for the next chapters of the thesis. In particular, in the next chapters, we will present the main contribution of this thesis, presenting an abstract interpreter for μ JS augmented by string-to-code statements, able to both perform a precise string analysis and answering questions about programs that transform strings into code, at run-time. Before doing that, we will first present state-of-art string abstractions integrated in real-world JavaScript static analyzers. In particular, we will discuss the string abstractions of TAJIS and SAFE analyzers in the context of backward completeness property introduced in Section 2.5. Given an operation, when backward completeness is guaranteed means that no loss of information arises during input abstract process of the operation of interest. We show that these abstractions are not backward complete w.r.t. some string operations of interest, and then we will show how to build a complete version of these abstractions. In the final discussion of the chapter, we will discuss the possibility to build complete domains also for more challenging string operations, such as `eval`. This will lead us to motivate the need of designing a novel string abstract domain for dynamic languages.

Chapter 4

Towards a string abstract domain for dynamic languages

In this chapter, we discuss the precision of existing string abstractions integrated into state-of-the-art static JavaScript analyzers based on abstract interpretation. As discussed in Section 2.5, when we talk about precision, we are talking about completeness.

Completeness in abstract interpretation is a well-known property, which ensures that the abstract framework does not lose information during the abstraction process, with respect to the property of interest. Completeness has never been taken into account for existing string abstract domains, due to the fact that it is difficult to prove it formally. However, the effort is fully justified when dealing with string analysis, which is a key issue to guarantee security properties in many software systems, in particular for JavaScript programs where poorly managed string manipulating code often leads to significant security flaws. In this chapter, we address backward completeness property¹ for the main JavaScript-specific string abstract domains, integrated into real JavaScript static analyzers, improving precision of them, w.r.t. some operations of interest. In particular, we will exploit the constructive methodologies presented in Section 2.5.5 providing suitable refinements of JavaScript-specific string abstract domains, and we discuss the benefits of guaranteeing completeness in the context of abstract interpretation-based string analysis of dynamic languages.

At the end of this chapter, we discuss and motivate the need of building a novel string abstract domain for the analysis of dynamic code, rather than the ones presented in this chapter.

4.1 An example of complete shell

As we have already mentioned in Section 2.5.5, the complete shell of an abstract domain A , w.r.t. an operation of interest $f : A \rightarrow B$, is a refinement of A that adds the *minimal* number of abstract points to A in order to make A complete w.r.t. the operation f . Before going into details of complete shells for JavaScript-specific string abstract domains, we give an informal introduction to complete shells by means of a simple but enough expressive example in the context of dynamic languages.

As mentioned more than once, a common feature of dynamic languages is to be not typed. Hence, in those languages, it is allowed to change the variable type through the program execution. For example, in PHP, it is completely legal to write fragments such as `$\$x=1;\$x=true;$` , where the type of the variable x changes from integer to boolean. One of the first attempt to statically reasoning about variable

¹In the rest of this chapter, when we talk about completeness, we mean backward completeness.

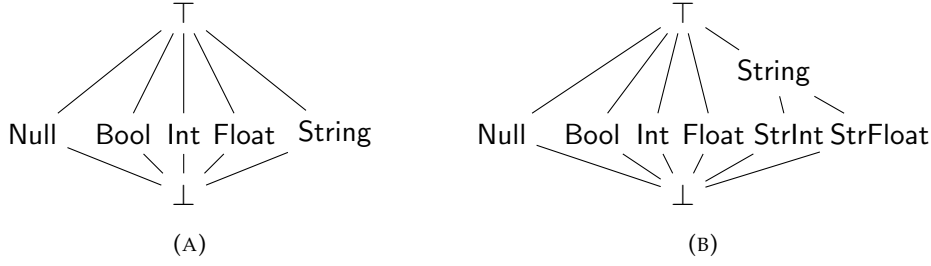


FIGURE 4.1: (a) Type abstract domain for PHP. (b) Complete shell of type abstract domain w.r.t. the sum operation.

types was to track the latter adopting the type abstract domain [Arceri and Maffei, 2017; Kneuss, Suter, and Kuncak, 2010] (similar to the one presented in Section 3.2), in order to detect whether a certain variable has constant type through the whole program execution. In Figure 4.1a, we report the type abstract domain for an intra-procedural version of PHP [Arceri and Maffei, 2017], that tracks null, boolean, integer, float and string types². Consider the formal semantics of the sum operation in PHP [Filaretti and Maffei, 2014]. When one of the operands is a string, since the sum operation is feasible only between numbers, *implicit type conversion* occurs and converts the string operand to a number. In particular, if the prefix of the string is a number, it is converted to the maximum prefix of the string corresponding to a number, otherwise it is converted to 0. For example, the expression $e = "2.4\text{hello}" + "4"$ returns 6.4. Let \oplus be the abstract sum operation on the type abstract domain. The type of the expression e is given by:

$$\alpha(\{"2.4\text{hello}"\}) \oplus \alpha(\{"4"\}) = \text{String} \oplus \text{String} = \top$$

The static type analysis based on the type abstract domain returns \top (i.e., any possible type), since the sum between two strings may return either an integer or a float value. Precisely, the domain is not complete w.r.t. the PHP sum operation, since for any string σ and σ' , it does not meet the completeness condition: $\alpha(\sigma \oplus \sigma') = \alpha(\sigma) \oplus \alpha(\sigma')$, e.g., $\alpha(\sigma + \sigma') = \text{Float} \neq \alpha(\sigma) \oplus \alpha(\sigma') = \top$. Intuitively, the type abstract domain is not complete w.r.t. the sum operation due to the loss of precision that occurs during the abstraction process of the inputs, since the domain is not precise enough to distinguish between strings that may be implicitly converted to integers or floats.

Figure 4.1b shows the complete shell of the type abstract domain w.r.t. the sum. The latter adds two abstract values to the original domain, namely StrFloat and StrInt, that correspond to the abstractions of the strings that may be implicitly converted to floats and to integers, respectively. Note that, the type analysis on the novel abstract domain is now complete w.r.t. the sum operation. Indeed, the completeness condition also holds for the expression e , as shown below.

$$\begin{aligned} \alpha(\{"2.4\text{hello}"\} + \{"4"\}) &= \text{Float} \\ &= \alpha(\{"2.4\text{hello}"\}) \oplus \alpha(\{"4"\}) \\ &= \text{StrFloat} \oplus \text{StrInt} \\ &= \text{Float} \end{aligned}$$

²Closing the type abstract domain by the powerset operation, a more precise abstract domain is obtained, called *union type* abstract domain [Kneuss, Suter, and Kuncak, 2010], that tracks the set of the possible types of a certain variable during program execution.

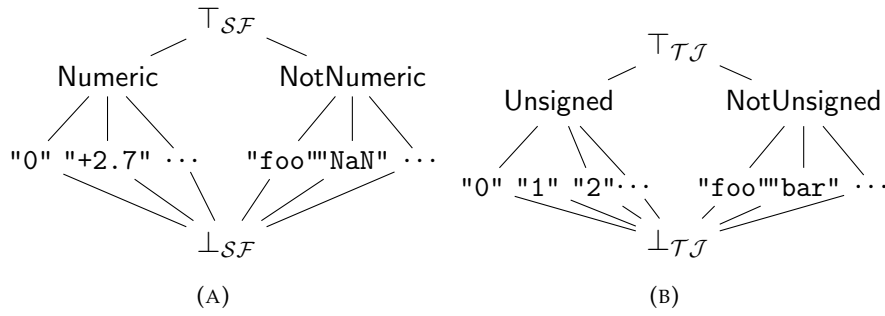


FIGURE 4.2: (a) SAFE, (b) TAJs string abstract domains recasted for μJS .

As pointed out above, guaranteeing completeness in abstract interpretation is a precious and desirable property that an abstract domain should aim to, since it ensures that no loss of precision occurs during the input abstraction process of the operation of interest. It is worth noting that *guessing* a complete abstract domain for a certain operation becomes particularly hard when the operation has a tricky semantics, such as in our example or, more in general, in dynamic languages operations. For this reason, complete shells become important since they are able to mathematically guarantee completeness for a certain operation, starting from an abstract domain of interest.

4.2 Making JavaScript string abstract domains complete

In this section, we study the completeness property of two string abstract domains integrated into two state-of-the-art JavaScript static analyzers based on abstract interpretation, that are SAFE [Lee et al., 2012] and TAJs [Jensen, Møller, and Thiemann, 2009]. Both the abstract domains track important information on JavaScript strings, e.g., SAFE tracks numeric strings, such as "2.5" or "+5", and TAJs is able to infer when a string corresponds to an unsigned integer, that may be used as array index.

For the sake of readability, we recast the original string abstract domains for μJS , following the notation adopted in [Amadini et al., 2017]. Figure 4.2 depicts them. Moreover, without loss of generality, the string "NaN", has no particular meaning here, and it is treated as a non-numerical string.

For each string abstract domain D , we denote by $\alpha_D : \wp(\Sigma^*) \rightarrow D$ its abstraction function, by $\gamma_D : D \rightarrow \wp(\Sigma^*)$ its concretization function, and by $\rho_D : \wp(\Sigma^*) \rightarrow \wp(\Sigma^*) \in \text{uco}(D)$ the associated upper closure operator.

4.2.1 Completing SAFE string abstract domain

Figure 4.2a depicts the string abstract domain $\mathcal{S}\mathcal{F}$, i.e., the recasted version of the domain integrated into the SAFE [Lee et al., 2012] static analyzer. It splits strings into the abstract values: Numeric (i.e., numerical strings) and NotNumeric (i.e., all the other strings). Before reaching these abstract values, $\mathcal{S}\mathcal{F}$ precisely tracks each single string value. For instance, $\alpha_{\mathcal{S}\mathcal{F}}(\{ "+9.6", "7" \}) = \text{Numeric}$, and $\alpha_{\mathcal{S}\mathcal{F}}(\{ "+9.6", "bar" \}) = \top_{\mathcal{S}\mathcal{F}}$.

We study the completeness of $\mathcal{S}\mathcal{F}$ w.r.t. concat operation. Figure 4.3 presents the abstract semantics of the concatenation operation for $\mathcal{S}\mathcal{F}$, that is:

$$\llbracket \text{concat}(\bullet, \bullet) \rrbracket^{\mathcal{S}\mathcal{F}} : \mathcal{S}\mathcal{F} \times \mathcal{S}\mathcal{F} \rightarrow \mathcal{S}\mathcal{F}$$

$\llbracket \text{concat}(s_1, s_2) \rrbracket^{\mathcal{SF}}$	$\perp_{\mathcal{SF}}$	$\sigma_2 \in \Sigma^*$	Numeric	NotNumeric	$\top_{\mathcal{SF}}$
$\perp_{\mathcal{SF}}$	$\perp_{\mathcal{SF}}$	$\perp_{\mathcal{SF}}$	$\perp_{\mathcal{SF}}$	$\perp_{\mathcal{SF}}$	$\perp_{\mathcal{SF}}$
$\sigma_1 \in \Sigma^*$	$\perp_{\mathcal{SF}}$	$\sigma_1 \cdot \sigma_2$	$\begin{cases} \text{Numeric} & \sigma_1 = "" \text{ or} \\ & \sigma_1 \in \Sigma_{\text{UInt}}^* \\ \text{NotNumeric} & \text{otherwise} \end{cases}$	NotNumeric	$\top_{\mathcal{SF}}$
Numeric	$\perp_{\mathcal{SF}}$	$\begin{cases} \text{Numeric} & \sigma_2 = "" \text{ or} \\ & \sigma_2 \in \Sigma_{\text{UInt}}^* \\ \text{NotNumeric} & \text{otherwise} \end{cases}$	$\top_{\mathcal{SF}}$	NotNumeric	$\top_{\mathcal{SF}}$
NotNumeric	$\perp_{\mathcal{SF}}$	NotNumeric	NotNumeric	NotNumeric	$\top_{\mathcal{SF}}$
$\top_{\mathcal{SF}}$	$\perp_{\mathcal{SF}}$	$\top_{\mathcal{SF}}$	$\top_{\mathcal{SF}}$	$\top_{\mathcal{SF}}$	$\top_{\mathcal{SF}}$

FIGURE 4.3: SAFE concat abstract semantics.

In particular, when both abstract values correspond to single strings, the standard string concatenation is applied (second row, second column). In the case in which one abstract value, involved in the concatenation, is a string and the other is Numeric (third row, second column and second row, third column) we distinguish two cases: if the string σ is empty or corresponds to an unsigned integer string (i.e., $\sigma \in \Sigma_{\text{UInt}}^*$) we can safely return Numeric, otherwise NotNumeric is returned. This happens because, when two float strings (hence numerical strings) are concatenated, a non-numerical string is returned (e.g., $\text{concat}("1.1", "2.2") = "1.12.2"$). For the same reason, when both input abstract values are Numeric, the result is not guaranteed to be numerical, indeed, $\llbracket \text{concat}(\text{Numeric}, \text{Numeric}) \rrbracket^{\mathcal{SF}} = \top_{\mathcal{SF}}$.

Lemma 4.1. \mathcal{SF} is not complete w.r.t. concat. In particular³, $\forall S_1, S_2 \in \wp(\Sigma^*)$ we have that:

$$\alpha_{\mathcal{SF}}(\llbracket \text{concat}(S_1, S_2) \rrbracket) \subsetneq \llbracket \text{concat}(\alpha_{\mathcal{SF}}(S_1), \alpha_{\mathcal{SF}}(S_2)) \rrbracket^{\mathcal{SF}}$$

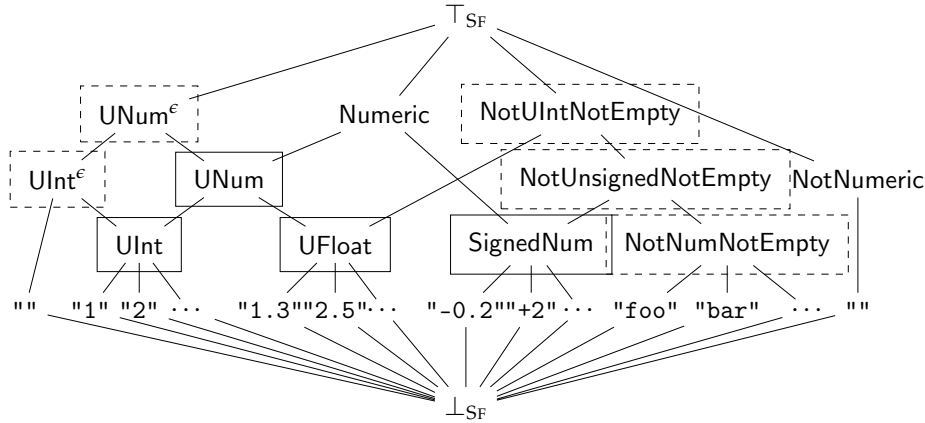
Consider $S_1 = \{"2.2", "2.3"\}$ and $S_2 = \{"2", "3"\}$. The completeness property does not hold:

$$\alpha_{\mathcal{SF}}(\llbracket \text{concat}(S_1, S_2) \rrbracket) = \text{Numeric} \neq \top_{\mathcal{SF}} = \llbracket \text{concat}(\alpha_{\mathcal{SF}}(S_1), \alpha_{\mathcal{SF}}(S_2)) \rrbracket^{\mathcal{SF}}$$

The \mathcal{SF} abstract domain loses too much information during the abstraction process; information that cannot be retrieved during the abstract concatenation. Intuitively, to gain completeness w.r.t. concat operation, \mathcal{SF} should improve the precision of the numerical strings abstraction, e.g., discriminating between float and integer strings. Following Theorem 2.40, we can formally construct the absolute complete shell of $\rho_{\mathcal{SF}}$ w.r.t. concat operation, that is $\overline{\mathcal{S}}_{\text{concat}}^{\rho_{\mathcal{SF}}}$. The abstract domain corresponding to this complete shell, that is complete for concat, is reported in Figure 4.4 and we denote it by SF (and hence its corresponding upper closure operator by ρ_{SF}).

In particular, the points inside dashed boxes are the abstract values added during the iterative computations of ρ_{SF} , the points inside standard boxes are instead obtained by the Moore closure of the other points of the domain, while the remaining abstract values were already in \mathcal{SF} . The meaning of abstract values in SF is intuitive. In order to satisfy the completeness property, SF splits the Numeric abstract value, already taken into account in \mathcal{SF} , into all the strings corresponding to unsigned integer (UInt), unsigned floats (UFloat), and signed numbers (SignedNum).

³We abuse notation denoting with $\llbracket \cdot \rrbracket$ the additive lift to set of basic values of the concrete semantics, i.e., the collecting semantics.

FIGURE 4.4: Absolute complete shell of ρ_{SF} w.r.t. concat .

Moreover, particular importance is given to the empty string, since the novel abstract domain specifies whether each abstract value contains "". Indeed, the UInt^ϵ abstract value represents the strings corresponding to unsigned integer or to the empty string, and the UNum^ϵ abstract value represents the strings corresponding to unsigned numbers or to the empty string. An unexpected abstract value considered in SF is $\text{NotUnsignedNotEmpty}$, such that:

$$\gamma_{SF}(\text{NotUnsignedNotEmpty}) = \{ \sigma \in \Sigma^* \mid \sigma \in \Sigma_{S\text{Num}}^* \cup (\Sigma_{\text{NotNum}}^* \setminus \{""\}) \}$$

where $\Sigma_{S\text{Num}}^*$ and Σ_{NotNum}^* correspond to the set of strings that are signed numbers and not numerical strings, respectively. Hence, the concretization of the above abstract point corresponds to the set of any non-numerical string, except the empty string, and any string corresponding to a signed number. This abstract point has been added to SF following the computation of the formula below:

$$\text{NotUnsignedNotEmpty} \in \max(\{ Z \in \wp(\Sigma^*) \mid \llbracket \text{concat}(\text{Numeric}, Z) \rrbracket \subseteq \gamma_{SF}(\text{NotNumeric}) \})$$

Informally speaking, we are wondering the following question: *which is the maximal set of strings s.t. concatenated to any possible numerical string will produce something that is dominated by any possible non-numerical string?* Indeed, in order to be sure to obtain non-numerical strings, the maximal set doing so is exactly the set of any non-numerical non-empty string, and any string corresponding to a signed number, that is $\text{NotUnsignedNotEmpty}$.

Theorem 4.2. ρ_{SF} is the absolute complete shell of ρ_{SF} w.r.t. concat operation and it is complete for it.

For example, consider again $S_1 = \{ "2.2", "2.3" \}$ and $S_2 = \{ "2", "3" \}$. Given SF , the completeness condition holds:

$$\begin{aligned} \alpha_{SF}(\llbracket \text{concat}(S_1, S_2) \rrbracket) &= \text{UFloat} \\ &= \llbracket \text{concat}(\alpha_{SF}(S_1), \alpha_{SF}(S_2)) \rrbracket^{SF} \\ &= \llbracket \text{concat}(\text{UFloat}, \text{UInt}) \rrbracket^{SF} \end{aligned}$$

4.2.2 Completing TAJs string abstract domain

Figure 4.2b depicts the string abstract domain \mathcal{TJ} , that is the string domain integrated into TAJs static analyzer [Jensen, Møller, and Thiemann, 2009]. Differently

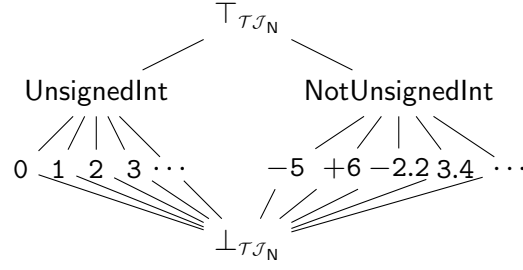


FIGURE 4.5: TAJN numerical abstract domain.

from \mathcal{SF} , it splits the strings into `Unsigned`, that denotes the strings corresponding to unsigned numbers, and `NotUnsigned`, any other string. Hence, for example, $\alpha_{\mathcal{TJ}}(\{"9", "+9"\}) = \top_{\mathcal{TJ}}$ and $\alpha_{\mathcal{TJ}}(\{"9.2", "\text{foo}"\}) = \text{NotUnsigned}$. As for \mathcal{SF} , before reaching these abstract values, \mathcal{TJ} precisely tracks single string values.

In this section, we focus on the `toNum` (i.e., string-to-number) operation. Since this operation clearly involves numbers, in Figure 4.5 we report the TAJN numerical abstract domain, denoted by \mathcal{TJ}_N . The latter domain behaves similarly to \mathcal{TJ} , distinguishing between unsigned and not unsigned integers. As far as TAJN is concerned, we consider the following string operation

$$\llbracket \text{toNum}(\sigma) \rrbracket = \begin{cases} \mathcal{I}(\sigma) & \sigma \in \Sigma_{\text{Num}}^* \\ 0 & \text{otherwise} \end{cases}$$

that takes a string as input and returns the number that it represents if the input string corresponds to a numerical string (i.e., $\sigma \in \Sigma_{\text{Num}}^*$), 0 otherwise. For example, $\text{toNum}("4.2") = 4.2$ and $\text{toNum}("asd") = 0$.

Below we define the abstract semantics of the string-to-number operation for \mathcal{TJ} . In particular, we define the function:

$$\llbracket \text{toNum}(\bullet) \rrbracket^{\mathcal{TJ}} : \mathcal{TJ} \rightarrow \mathcal{TJ}_N$$

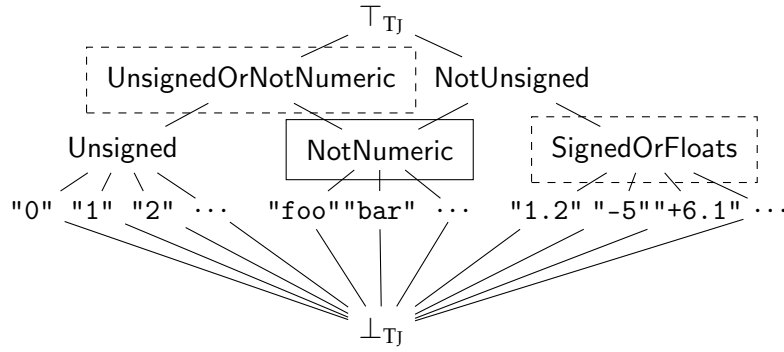
that takes as input a string abstract value in $\sigma \in \mathcal{TJ}$, and returns an integer abstract value in \mathcal{TJ}_N .

$$\llbracket \text{toNum}(\sigma) \rrbracket^{\mathcal{TJ}} = \begin{cases} \perp_{\mathcal{TJ}_N} & \sigma = \perp_{\mathcal{TJ}} \\ \llbracket \text{toNum}(\sigma) \rrbracket & \sigma \in \Sigma^* \\ \text{UnsignedInt} & \sigma = \text{Unsigned} \\ \top_{\mathcal{TJ}_N} & \sigma = \text{NotUnsigned} \vee \sigma = \top_{\mathcal{TJ}} \end{cases}$$

When the input evaluates to $\perp_{\mathcal{TJ}}$, bottom is propagated and $\perp_{\mathcal{TJ}_N}$ is returned (first row). If the input evaluates to a single string value, the abstract semantics relies on its concrete one (second row). When the input evaluates to the string abstract value `Unsigned` (third row), the integer abstract value `UnsignedInt` is returned. Finally, when the input evaluates to `NotUnsigned` or $\top_{\mathcal{TJ}}$, the top integer abstract value is returned (forth row).

Lemma 4.3. \mathcal{TJ} is not complete w.r.t. `toNum`. In particular, $\forall S \in \wp(\Sigma^*)$ we have that:

$$\alpha_{\mathcal{TJ}_N}(\llbracket \text{toNum}(S) \rrbracket) \subsetneq \llbracket \text{toNum}(\alpha_{\mathcal{TJ}}(S)) \rrbracket^{\mathcal{TJ}}$$

FIGURE 4.6: Complete shell of ρ_{TJ} relative to ρ_{TJ_N} w.r.t. toNum.

For example, consider $S = \{ "2.3", "3.4" \}$. The completeness property does not hold:

$$\alpha_{TJ_N}(\llbracket \text{toNum}(S) \rrbracket) = \text{NotUnsignedInt} \neq \top_{TJ_N} = \llbracket \text{toNum}(\alpha_{TJ}(S)) \rrbracket^{TJ}$$

Again, the completeness condition does not hold because the TJ string abstract domain loses too much information during the input abstraction process, and the latter information cannot be retrieved during the abstract toNum operation. In particular, when non-numeric strings and unsigned integer strings are converted to numbers by toNum, they are mapped to the same value, namely 0. Indeed, TJ does not differentiate between non-numeric and unsigned integer string values, and this is the principal cause of the TJ incompleteness w.r.t. toNum. Additionally, more precision can be obtained if we could differentiate numeric strings holding float numbers from those holding integer numbers. Thus, in order to make TJ complete w.r.t. toNum, we have to derive the complete shell of the TJ string abstract domain relative to the TJ_N numerical abstract domain, applying Theorem 2.39. In particular, let ρ_{TJ} and ρ_{TJ_N} be the upper closure operators related to TJ and TJ_N abstract domains, respectively. By applying Theorem 2.39, we obtain $S_{\text{toNum}}^{\rho_{TJ}}(\rho_{TJ_N})$, namely the complete shell of ρ_{TJ} relative to ρ_{TJ_N} w.r.t. toNum. For the sake of readability, we denote this upper closure operator by ρ_{TJ} and the corresponding abstract domain, denoted by TJ , is depicted in Figure 4.6.

In particular, the abstract points inside dashed boxes are the abstract values added during the iterative computations of ρ_{TJ} , the points inside the standard boxes are instead obtained by the Moore closure of the other points of the domain, while the remaining abstract values were already in TJ . A non-intuitive point added by TJ is SignedOrFloats, namely the abstract value s.t. its concretization contains any float string and the signed integer strings. This abstract point is added during the computation of TJ , following the formula below:

$$\text{SignedOrFloats} \in \max(\{ Z \in \wp(\Sigma^*) \mid \llbracket \text{toNum}(Z) \rrbracket \subseteq \gamma_{TJ_N}(\text{NotUnsignedInt}) \})$$

Informally speaking, we are wondering the following question: *which is the maximal set of strings Z s.t. toNum(Z) is dominated by NotUnsignedInt?* In order to obtain from toNum(Z) only values dominated by NotUnsignedInt, the maximal set doing so is exactly the set of the float strings and the signed strings. Other strings, such that: unsigned integer strings or not numerical strings are excluded, since they are both converted to unsigned integers, and they would violate the dominance relation.

Similarly, the abstract point UnsignedOrNotNumeric is added to the absolute complete shell TJ , when the following formula is computed:

$$\text{UnsignedOrNotNumeric} \in \max(\{ Z \in \wp(\Sigma^*) \mid \llbracket \text{toNum}(Z) \rrbracket \subseteq \gamma_{\mathcal{T}\mathcal{J}_N}(\text{UnsignedInt}) \})$$

In order to obtain from $\text{toNum}(Z)$ only values dominated by UnsignedInt , the maximal set doing so is exactly the set of unsigned integer strings and non-numerical strings, since the latter are converted to 0.

Theorem 4.4. ρ_{TJ} is the complete shell of $\rho_{\mathcal{T}\mathcal{J}}$ relative to $\rho_{\mathcal{T}\mathcal{J}_N}$ w.r.t. toNum operation and hence it is complete for it.

For example, consider again the string set $S = \{"2.3", "3.4"\}$. Given TJ , the completeness condition holds:

$$\begin{aligned} \alpha_{\mathcal{T}\mathcal{J}_N}(\llbracket \text{toNum}(S) \rrbracket) &= \text{NotUnsignedInt} \\ &= \llbracket \text{toNum}(\alpha_{\text{TJ}}(S)) \rrbracket^{\mathcal{T}\mathcal{J}} \\ &= \llbracket \text{toNum}(\text{SignedOrFloats}) \rrbracket^{\mathcal{T}\mathcal{J}} \end{aligned}$$

4.3 What we gain from using a complete abstract domain?

In this section, we discuss and evaluate the benefits of adopting the complete shells reported in Section 4.2 and, more in general, complete domains, w.r.t. a certain operation. In particular, we compare the string abstract domains adopted by SAFE and TAJs with their corresponding complete shells, we discuss the complexity of the complete shells, and finally we argue how adopting complete abstract domains can be useful into static analyzers.

Precision. In the previous section, we focused on the completeness of the string abstract domains integrated into SAFE and TAJs w.r.t. two string operations, namely concat and toNum , respectively. While string concatenation is common in any programming language, toNum assumes critical importance in the dynamic language context, mostly where implicit type conversion is provided. Since type conversion is often hidden from the developer, aim to completeness of the analysis increases the precision of such operations. For instance, let x be a variable, at a certain program execution point. x may have concrete value in the set $S = \{"foo", "bar"\}$. If S is abstracted into the starting TAJs string abstract domain, its abstraction will correspond to NotUnsigned , losing the information about the fact that the concrete value of x surely does not contain numerical values. Hence, when the abstract value of S is used as input of toNum , the result will return $\top_{\mathcal{T}\mathcal{J}_N}$, i.e., any possible concrete integer value. Conversely, abstracting S in TJ (the absolute complete shell of $\mathcal{T}\mathcal{J}$ relative to toNum discussed in Section 4.2.2) leads to a more precise abstraction, since TJ is able to differentiate between non-numerical and numerical strings. In particular, the abstract value of S in TJ is NotNumeric , hence $\text{toNum}(\text{NotNumeric})$ will precisely return 0.

Adopting a complete shell w.r.t. a certain operation does not compromise the precision of the others. For example, consider again the original string abstract domain into TAJs static analyzer and the following JavaScript fragment.

```
1 var obj = {
2   "foo" : 1,
3   "bar" : 2,
4   "1.2" : 3,
5   "2.2" : "hello"
6 }
7
8 y = obj[idx];
```

Let us suppose that the value of `idx` is the abstraction, in the starting TAJIS string abstract domain, of the string set $S = \{\text{"foo"}, \text{"bar"}\}$, namely the abstract value `NotUnsigned`. The variable `idx` is used to access a property of the object `obj` at line 8 and, in the abstract computation, to guarantee soundness, it accesses *all* the properties of `obj`, includes the fields `"1.2"` and `"2.2"`, introducing noise in the abstract computation, since `"1.2"` and `"2.2"` are false positives values introduced by the abstraction of the values of `idx`. If we analyze the same JavaScript fragment with the absolute complete shell (w.r.t. `toNum` operation) of the TAJIS string abstract domain defined in Section 4.2.2, we obtain more precise results. Indeed, in this case, the value of `idx` corresponds to the abstract value `NotNumeric`, and when it is used to access the object `obj` at line 8, only `"foo"` and `"bar"` are accessed, since they are the only non-numerical string properties of `obj`.

Complexity of the complete shells. We evaluate the complexity of the complete shells we have provided in the previous section. As usual in static analysis by abstract interpretation, there exists a trade-off between precision and efficiency: choose a more precise abstract domain may compromise the efficiency of the abstract computations. A representative example is reported in [Giacobazzi, Ranzato, and Scozari, 2000]: the complete shell of the sign abstract domain w.r.t. addition is the interval abstract domain. Hence, starting from a finite height abstract domain (signs) we obtain an infinite height abstract domain (intervals). In particular, fix-point computations on signs converge, while they may diverge on intervals, being interval abstract domain not-ACC. Indeed, after the completion, the interval abstract domain should be equipped also with a widening [Cousot and Cousot, 1977] in order to still guarantee termination. A worst-case scenario is when the complete shells w.r.t. a certain operation exactly corresponds to the collecting abstract domain, i.e., the concrete domain. Clearly, we cannot use the concrete domain due to undecidability reasons, but this suggests us to change the starting abstract domain, since it is not able to track any information related to the operation of interest. An example is the suffix abstract domain [Costantini, Ferrara, and Cortesi, 2015] with substring operation: since this abstract domain tracks only the common suffix of a strings set, it cannot track the information about the indexes of the common suffix, and the complete shell of the suffix abstract domain w.r.t. substring would lead to the concrete domain. Hence, if the focus of the abstract interpreter is to improve the precision of the substring operation, we should change the abstract domain with a more precise one for substring.

Consider now the complete shells reported in Section 4.2. The obtained complete shells still have finite height, hence termination is still guaranteed without the need to equip the complete shells with widening operators. Moreover, the complexity of the string operations of interest is preserved after completion. Indeed, in both TAJIS and SAFE abstract domains, `concat` and `toNum` operations have constant complexity, respectively, and the same complexity is preserved in the corresponding complete

shells. It is worth noting that also the complexity of the abstract domain-related operations, such as least upper bound, greatest lower bound and the ordering operator, is preserved in the complete shells. Hence, to conclude, as far as the complete shells we have reported for TAJs and SAFE are concerned, there is no worsening when we substitute the original string abstract domains with the corresponding complete shells, and this leads, as we have already mentioned before, to completeness during the input abstraction process w.r.t. the relative operations, namely `concat` for SAFE and `toNum` for TAJs.

False positives reduction. As we have already mentioned before, in static analysis a certain degree of abstraction must be added in order to obtain decidable procedures to infer invariants on a generic program. Clearly, using less precise abstract domains lead to an increase of *false positive* values of the computed invariants. In particular, after a program is analyzed, this burdens the phase of false positive detection: when a program is analyzed, the phase after consists of detecting which values of the invariants derived by the static analyzer are spurious values, namely values that are not certainly computed by the concrete execution of the program of interest. In particular, using imprecise (i.e., not complete) abstract domains clearly augment the number of false positives in the abstract computation of the static analyzer, burdening the next phase of detection of the spurious values. On the other hand, adopting (backward) complete abstract domains w.r.t. a certain operation reduces the numbers of false positives introduced during the abstract computations, at least in the input abstraction process. Clearly, in this way, the next phase of detection of false positives will be lighter since less noise has been introduced during the abstract computation of the invariants. Consider again the JavaScript fragment reported in the previous paragraph. As we have already discussed before, using the starting TAJs abstract domain to abstract the variable `idx` leads to a loss of precision, since the spurious value `"1.1"` and `"1.2"` are taken into account in its abstract value, namely `NotUnsigned`. Using the complete shell of TAJs w.r.t. `toNum` instead does not add noise when `idx` is used to access `obj`.

4.4 Can we use complete shells for dynamic code analysis?

We have addressed the problem of backward completeness in JavaScript-specific string abstract domains, and provided, in particular, the complete shells of TAJs and SAFE string abstract domains w.r.t. `concat` and `toNum` operations, respectively. Our results can be easily applied also to JSAI string abstract domain [Kashyap et al., 2014], as it can be seen as an extension of the SAFE domain. At the end, we have also discussed the importance, for strings abstractions, of guaranteeing completeness for strings operations.

Complete shells can help to provide more precise string analyses, enabling in turn more precision in abstracting string-to-code inputs. One can think to use the methodologies of complete shells also with respect to string-to-code statements, starting from the string abstract domains presented in this chapter. Unfortunately, doing so, will lead to the string concrete domain, producing no useful abstract domain for analyzing string-to-code statements. Informally speaking, string-to-code statements (e.g., `eval`) exactly implement the concrete interpreter of a language and it does make sense to get the concrete domain when you try to compute the complete shell for such operations. Hence, complete shells do not help in constructively

building a useful domain for analyzing dynamic code and we need to build a novel string abstract domain upon which we can build an analysis for dynamic code.

We aim at a strings abstraction collecting, as faithfully as possible, the set of possible values that a string variable may receive before string-to-code executes it. It surely has to approximate the set of possible string values, hence it has to be a language, it has also to keep enough information for allowing us to extract code from it, but it has also to keep enough information for analyzing properties of string variables that are never executed by a string-to-code statement during computation.

We think that a suitable string abstraction meeting all the requirements is regular languages, by means of their representation of finite state automata. In particular, in the next chapter we will formally present the finite state automata abstract domain, that prepares the ground for the dynamic code analysis.

Chapter 5

The finite state automata domain

In this chapter, we define the finite state automata abstract domain for string values, namely the domain of regular languages over Σ^* , inspired by similar string domains such as the ones reported in [Park, Im, and Ryu, 2016; Choi et al., 2006; Yu et al., 2008]. The domain is the core of the abstract interpreter for dynamically generated code we will describe later. In particular, we will first formally define the abstract domain, then we will define some novel important operations on finite state automata, characterizing several notions of substring languages and automata, useful to integrate the domain into an abstract interpreter for string manipulation programming languages.

5.1 $DFA_{/\equiv}$ abstract domain

The aim of this section is that of characterize automata as a domain for abstracting the computation of program semantics in the abstract interpretation framework. The exploited idea is that of approximating strings as regular languages represented by the minimum DFA [Davis, Sigal, and Weyuker, 1994] recognizing them. We denote the set of all (deterministic) finite state automata by DFA . In general, given a regular language, we can have more finite state automata that recognize that language. Let us consider the equivalence relation $\equiv_{\subseteq DFA \times DFA}$ defined as

$$\forall A_1, A_2 \in DFA. A_1 \equiv A_2 \Leftrightarrow \mathcal{L}(A_1) = \mathcal{L}(A_2)$$

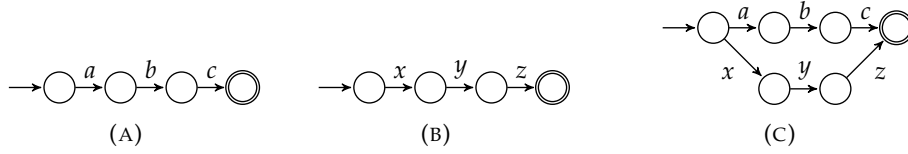
We consider the partition induced by \equiv of DFA , denoted by $DFA_{/\equiv} \triangleq \{ [A]_{\equiv} \mid A \in DFA \}$, namely the set of all the equivalence classes induced by \equiv , where any equivalence class $[A]_{\equiv}$ is composed by the DFA that recognize the same language. When we refer to an element of $DFA_{/\equiv}$, we abuse notation by representing equivalence classes in the domain $DFA_{/\equiv}$ w.r.t. \equiv by one of its automata (usually the minimum w.r.t. the number of states), that is when we write $A \in DFA_{/\equiv}$ we mean $[A]_{\equiv}$. We define the partial order \sqsubseteq_{DFA} induced by language inclusion, that is:

$$\forall A_1, A_2 \in DFA_{/\equiv}. A_1 \sqsubseteq_{DFA} A_2 \Leftrightarrow \mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$$

Lemma 5.1. $\langle DFA_{/\equiv}, \sqsubseteq_{DFA} \rangle$ is a poset.

Proof. For all $A_1, A_2, A_3 \in DFA_{/\equiv}$ the relation \sqsubseteq_{DFA} satisfies:

1. *Reflexivity:* $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_1) \Leftrightarrow A_1 \sqsubseteq_{DFA} A_1$

FIGURE 5.1: (a) A_1 (b) A_2 (c) $\text{Min}(A_1 \sqcup_{\text{DFA}} A_2)$ 2. *Anti-symmetry:*

$$\begin{aligned}
 & A_1 \sqsubseteq_{\text{DFA}} A_2 \wedge A_2 \sqsubseteq_{\text{DFA}} A_1 \\
 \Leftrightarrow & \mathcal{L}(A_1) \subseteq \mathcal{L}(A_2) \wedge \mathcal{L}(A_2) \subseteq \mathcal{L}(A_1) \\
 \Leftrightarrow & \mathcal{L}(A_1) = \mathcal{L}(A_2) \\
 \Leftrightarrow & A_1 = A_2
 \end{aligned}$$

3. *Transitivity:*

$$\begin{aligned}
 & A_1 \sqsubseteq_{\text{DFA}} A_2 \wedge A_2 \sqsubseteq_{\text{DFA}} A_3 \\
 \Leftrightarrow & \mathcal{L}(A_1) \subseteq \mathcal{L}(A_2) \wedge \mathcal{L}(A_2) \subseteq \mathcal{L}(A_3) \\
 \Rightarrow & \mathcal{L}(A_1) \subseteq \mathcal{L}(A_3) \\
 \Leftrightarrow & A_1 \sqsubseteq_{\text{DFA}} A_3
 \end{aligned}$$

□

$\text{DFA}_{/\equiv}$ has bottom element, denoted by $\text{Min}(\emptyset)$, that is the equivalence class represented by the automaton that recognizes the empty language, and top element, denoted by $\text{Min}(\Sigma^*)$, that is the equivalence class represented by the automaton that recognizes Σ^* .

The least upper bound (lub) $\sqcup_{\text{DFA}} : \text{DFA}_{/\equiv} \times \text{DFA}_{/\equiv} \rightarrow \text{DFA}_{/\equiv}$ on the domain $\text{DFA}_{/\equiv}$, corresponds to the standard union between automata:

$$\forall A_1, A_2 \in \text{DFA}_{/\equiv}. A_1 \sqcup_{\text{DFA}} A_2 \triangleq \text{Min}(\mathcal{L}(A_1) \cup \mathcal{L}(A_2))$$

It is the minimum automaton recognizing the union of the languages $\mathcal{L}(A_1)$ and $\mathcal{L}(A_2)$. This is a well-defined notion since regular languages are closed under union. As example, consider Figure 5.1, where the automaton in Figure 5.1c is the least upper bound of A_1 and A_2 given in Figure 5.1a and Figure 5.1b, respectively. Dually, the greatest lower bound (glb) $\sqcap_{\text{DFA}} : \text{DFA}_{/\equiv} \times \text{DFA}_{/\equiv} \rightarrow \text{DFA}_{/\equiv}$ corresponds to automata intersection, formally:

$$\forall A_1, A_2 \in \text{DFA}_{/\equiv}. A_1 \sqcap_{\text{DFA}} A_2 \triangleq \text{Min}(\mathcal{L}(A_1) \cap \mathcal{L}(A_2))$$

As for automata least upper bound, also the greatest lower bound is well-defined since regular languages are closed under intersection.

Lemma 5.2. $\langle \text{DFA}_{/\equiv}, \sqsubseteq_{\text{DFA}}, \sqcup_{\text{DFA}}, \sqcap_{\text{DFA}}, \text{Min}(\emptyset), \text{Min}(\Sigma^*) \rangle$ is a lattice.

Proof. The lemma holds since regular languages (hence, finite state automata) are closed under finite union and intersection, i.e., \sqcup_{DFA} and \sqcap_{DFA} . □

Theorem 5.3. $\langle \text{DFA}_{/\equiv}, \sqsubseteq_{\text{DFA}}, \sqcup_{\text{DFA}}, \sqcap_{\text{DFA}}, \text{Min}(\emptyset), \text{Min}(\Sigma^*) \rangle$ is not a complete lattice.

Proof. Consider the family of regular languages $\mathcal{L}_j = \Sigma^* \setminus \{a^j b^j\}$, with $j \in \mathbb{N}$. For each \mathcal{L}_j there exists a deterministic finite state automaton A_j such that $\mathcal{L}(A_j) = \mathcal{L}_j$. Suppose that $A \in \text{DFA}_{/\equiv}$ is the greatest lower bound of $\{A_j\}_{j \in \mathbb{N}}$ and consider the language $\mathcal{L}' = \{ \sigma \in \{a, b\}^* \mid |\sigma|_a \neq |\sigma|_b \}$. Since $\mathcal{L}(A)$ contains all the strings not in the form $a^i b^i$, $\mathcal{L}' \subseteq \mathcal{L}(A)$. In particular, $\mathcal{L}' \subset \mathcal{L}(A)$.

$$\begin{aligned} & \mathcal{L}' \subset \mathcal{L}(A) \\ \Rightarrow & \exists \sigma \in \mathcal{L}(A) \setminus \mathcal{L}' \\ \Rightarrow & \mathcal{L}' \subset \mathcal{L}(A) \setminus \{\sigma\} \subset \mathcal{L}(A) \end{aligned}$$

This leads to a contradiction, as $\mathcal{L}(A)$ does not contain only the strings not in the form $a^i b^i$ and, in turn, the automaton A is not the greatest lower bound of $\{A_j\}_{j \in \mathbb{N}}$. \square

In other words, there exists no Galois connections between $\text{DFA}_{/\equiv}$ and $\wp(\Sigma^*)$, i.e., there may exist no minimal automaton abstracting a language. Some works have studied automatic procedures to compute, given an input language \mathcal{L} , the *regular cover* of \mathcal{L} [Domaratzki, Shallit, and Yu, 2001], namely an automaton containing the language \mathcal{L} . In particular, [Câmpeanu, Paun, and Yu, 2002; Domaratzki, Shallit, and Yu, 2001] have studied regular covers guaranteeing that the automaton obtained is the best w.r.t. a *minimal relation* (but not minimum).

The best abstraction fails since the set of the possible abstractions of a language $\mathcal{L} \in \wp(\Sigma^*)$ may be a strictly decreasing chain w.r.t. \sqsubseteq_{DFA} . Let us consider the context-free language $\mathcal{L} = \{ a^n b^n \mid n \in \mathbb{N} \}$. The set of the possible finite state automata approximating the language is indeed reported in the following.

$$\begin{aligned} & \text{Min}(\{\epsilon\} \cup \{ a^n b^m \mid n + m \geq 0 \}) \\ & \sqsupseteq_{\text{DFA}} \text{Min}(\{\epsilon, ab\} \cup \{ a^n b^m \mid n + m \geq 1 \}) \\ & \sqsupseteq_{\text{DFA}} \text{Min}(\{\epsilon, ab, a^2 b^2\} \cup \{ a^n b^m \mid n + m \geq 2 \}) \\ & \sqsupseteq_{\text{DFA}} \text{Min}(\{\epsilon, ab, a^2 b^2, a^3 b^3\} \cup \{ a^n b^m \mid n + m \geq 3 \}) \\ & \dots \end{aligned}$$

However, this is not a concern. Indeed, as stated in [Cousot and Cousot, 1992a], one can abandon the idea of having the best abstraction, choosing an arbitrary abstract element among the possible abstractions of a concrete element. Other solutions can be performed since the relation between concrete semantics and abstract semantics can be weakened while still ensuring soundness [Cousot and Cousot, 1992b]. A well-known example of abstraction where the best abstraction does not exist is the convex polyhedra domain [Cousot and Halbwachs, 1978]. From here on, we denote by $\gamma_{\text{DFA}} : \text{DFA}_{/\equiv} \rightarrow \wp(\Sigma^*)$ the concretization function of $\text{DFA}_{/\equiv}$, that given an automaton returns the language recognized by the automaton, namely $\gamma_{\text{DFA}}(A) \triangleq \mathcal{L}(A)$.

Widening automata. It is easy to see that the abstract domain $\text{DFA}_{/\equiv}$ is infinite and it is not ACC, as stated by the following theorem.

Theorem 5.4. $\langle \text{DFA}_{/\equiv}, \sqsubseteq_{\text{DFA}} \rangle$ is not ACC.

Proof. As a counterexample, let us consider the family of regular languages $\mathcal{L}_j = \{ a^i b^i \mid 0 \leq i \leq j \}$. For each $j \in \mathbb{N}$ there exists the minimal automaton A_j that recognizes the language \mathcal{L}_j , since these languages are regular. It is trivial to prove

that $\forall i \in \mathbb{N}. A_i \subseteq A_{i+1}$. Let $\{A_j\}_{j \in \mathbb{N}}$ be an infinite increasing chain and suppose that stabilizes after a certain number of steps $k \in \mathbb{N}$.

$$\begin{aligned} & \exists k \in \mathbb{N}. \forall j \geq k \quad A_k = A_j \\ \Rightarrow & A_k = A_{k+1} \\ \Leftrightarrow & \mathcal{L}(A_k) = \mathcal{L}(A_{k+1}) \\ \Leftrightarrow & \{ a^i b^i \mid 0 \leq i \leq k \} = \{ a^i b^i \mid 0 \leq i \leq k+1 \} \end{aligned}$$

The two languages are not equal since the string $\sigma = a^{k+1}b^{k+1} \in \mathcal{L}(A_{k+1})$ but $\sigma \notin \mathcal{L}(A_k)$, leading to a contradiction. \square

The domain $\text{DFA}_{/\equiv}$ is an infinite domain, and it is not ACC, i.e., it contains infinite ascending chains. This clearly implies that any computation on $\text{DFA}_{/\equiv}$ may lose convergence [Cousot and Cousot, 1992b]. Most of the proposed abstract domains for strings [Costantini, Ferrara, and Cortesi, 2015; Kashyap et al., 2014; Jensen, Møller, and Thiemann, 2009; Lee et al., 2012] trivially satisfy ACC by being finite, but they may lose precision during the abstract computation [Cousot and Cousot, 1992b]. In these cases, domains must be equipped with a widening operator approximating the least upper bound in order to enforce convergence (by necessarily losing precision) for any increasing chain [Cousot and Cousot, 1992b]. As far as automata are concerned, existing widenings are defined in terms of a state equivalence relation merging states that recognize the same language, up to a fixed length n (set as parameter for tuning the widening precision) [D'Silva, 2006; Bartzis and Bultan, 2004]. We denote this parametric widening with $\nabla_{\text{DFA}_{/\equiv}}^n : \text{DFA}_{/\equiv} \times \text{DFA}_{/\equiv} \rightarrow \text{DFA}_{/\equiv}$, with $n \in \mathbb{N}$ [D'Silva, 2006] and it is defined in the following.

Let $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ and $A' = \langle Q', \Sigma, \delta', q'_0, F' \rangle$ be two finite state automata such that $\mathcal{L}(A) \subseteq \mathcal{L}(A')$: the widening between A and A' is formalized in terms of a relation $R \subseteq Q \times Q'$ between the sets of states of the two automata. The relation R is used to define an equivalence relation $\equiv_R \subseteq Q' \times Q'$ over the states of A' , such that $\equiv_R = R \circ R^{-1}$. The widening between A and A' is then given by the quotient automaton of A' w.r.t. the partition induced by \equiv_R : $A' \nabla_R A = A'_{\equiv_R}$ ¹. Thus, the widening operator merges the states of A' that are equivalent by the relation \equiv_R . By changing the relation R , we obtain different widening operators [D'Silva, 2006]. It has been proved that convergence is guaranteed when the relation $R_n \subseteq Q \times Q'$ is such that $(q, q') \in R_n$ iff q and q' recognize the same language of strings of length at most n [D'Silva, 2006]. Thus, the parameter n tunes the length of strings determining the equivalence of states used for merging them in the widening. It is worth noting that, the smaller is n , the more information will be lost by widening automata.

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

$$A \nabla_{\text{DFA}_{/\equiv}}^n A' \triangleq A \nabla_{R_n} (A \sqcup_{\text{DFA}} A')$$

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

Example 5.5. Consider the following μJS program.

¹Given $A \in \text{DFA}_{/\equiv}$ and a partition π over its states, we denote as $A_\pi = \langle Q', \delta', q'_0, F', \Sigma \rangle$ the *quotient automaton* [Davis, Sigal, and Weyuker, 1994].

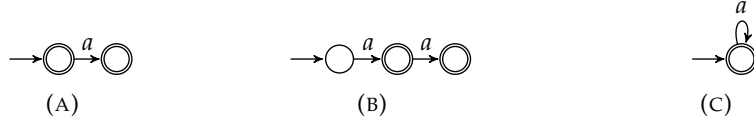


FIGURE 5.2: (a) A_1 , $\mathcal{L}(A_1) = \{\epsilon, a\}$ (b) A_2 , $\mathcal{L}(A_2) = \{a, aa\}$ (c)
 $A_1 \nabla_{\text{DFA}/\equiv}^1 A_2$

```

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

```

Since the value of the variable x is unknown, also the number of iterations of the while-loop is unknown. In these cases, in order to guarantee soundness and termination, we apply the widening operator. In Figure 5.2a, we report the abstract value of the variable str at the beginning of the second iteration of the loop, while in Figure 5.2b the abstract value of the variable str at the end of the second iteration is reported. Before starting a new iteration, in the example, we apply $\nabla_{\text{DFA}/\equiv}^1$ between two automata, namely we merge all the states having the same outgoing character. The minimization of the obtained automaton is reported in Figure 5.2c. The next iteration will reach the fix-point, guaranteeing termination. Moreover, in this case, the result that we obtain is the most precise we can hope for.

5.2 Characterization of substrings languages

In this section, we characterize the substring languages of a regular language. In Section 5.1, we have introduced the factors of a regular language, and the corresponding operator on finite state automata, containing any substring of a regular language. Here, we want to characterize sets of substrings between an initial and a final index (hence subsets of factors). After that, we aim at computing the automata recognizing these sets. Hence, we will introduce several novel and non-standard automata operations, needed to integrate the abstract domain DFA/\equiv into the abstract interpreter we will define in the next chapter.

5.2.1 Substring language between two fixed indexes

In this section, we are interested in studying the *substring* language, that is, given a regular language \mathcal{L} , we aim at computing another language \mathcal{L}' of substrings of \mathcal{L} between two indexes $i, j \in \mathbb{N}$, such that $i \leq j$. In particular, we identify two types of substrings: *proper* and *non proper substrings*.

For example, let us consider the string *hello* and the substring from 1 to 3, namely *el*. We say that *el* is a *proper substring*, since it is fully contained in *hello*. Hence, given a regular language, we can define the proper substring language of \mathcal{L} .

Definition 5.6 (Proper substring language). Let \mathcal{L} be a regular language and $i, j \in \mathbb{N}$ s.t. $i \leq j$. The proper substring language of \mathcal{L} from i to j is defined as follows.

$$\text{PSS}(\mathcal{L}, i, j) \triangleq \{ y \in \Sigma^* \mid \exists z \in \Sigma^*. yz \in \text{Su}(\mathcal{L}, i), z \in \text{Su}(\mathcal{L}, j) \}$$

For example, let us consider $\mathcal{L} = \{ a^n \mid n \in \mathbb{N} \}$. The proper substring language of \mathcal{L} from 1 to 3 is $\text{PSS}(\mathcal{L}, 1, 3) = \{aa\}$. It is trivial to see that any proper substring

has length $j - i$, that is $\text{PSS}(\mathcal{L}, i, j) \subseteq \Sigma^{j-i}$. Let us consider again the string *hello* and consider its substring from 3 to 7. We call such substring *non-proper substring*, since at least one of the indexes is out-of-bound of the length of the string. As we have done for proper substrings, we can define the non-proper substring language of a regular language.

Definition 5.7 (Non-proper substring language). Let \mathcal{L} be a regular language and $i, j \in \mathbb{N}$ s.t. $i \leq j$. The non-proper substring language of \mathcal{L} from i to j is defined as follows.

$$\text{NSS}(\mathcal{L}, i, j) \triangleq \{ y \in \Sigma^* \mid y \in \text{SU}(\mathcal{L}, i), y \in \Sigma^{<j-i} \} \cup \{ \epsilon \mid \mathcal{L} \cap \Sigma^{\leq i} \neq \emptyset \}$$

For example, let us consider $\mathcal{L} = \{ a^n \mid n \in \mathbb{N} \}$. The non-proper substring language from 1 to 3 of \mathcal{L} is $\text{PSS}(\mathcal{L}, 1, 3) = \{ \epsilon, a \}$, corresponding to the substring from 1 to 3 of ϵ and a , respectively. It is trivial to see that any non proper substring has length less than $j - i$, that is $\text{NSS}(\mathcal{L}, i, j) \subseteq \Sigma^{<j-i}$.

Hence, we can characterize the substring language of a regular language as the union of its proper and non-proper substrings.

Definition 5.8 (Substring language). Let \mathcal{L} be a regular language and $i, j \in \mathbb{N}$ s.t. $i \leq j$. The substring language of \mathcal{L} from i to j is defined as follows.

$$\text{Ss}(\mathcal{L}, i, j) \triangleq \text{PSS}(\mathcal{L}, i, j) \cup \text{NSS}(\mathcal{L}, i, j)$$

Let us consider again $\mathcal{L} = \{ a^n \mid n \in \mathbb{N} \}$, we have that its substring language from 1 to 3 is $\text{Ss}(\mathcal{L}, 1, 3) = \{ \epsilon, a, aa \}$.

At this point, our goal is to compute these particular sets by means of the automata that recognize these languages. Given $i, j \in \mathbb{N}$ ($i \leq j$) and a finite state automaton $A \in \text{DFA}_{/\equiv}$ recognizing the language \mathcal{L} , we aim at building the automata recognizing $\text{PSS}(\mathcal{L}, i, j)$ and $\text{NSS}(\mathcal{L}, i, j)$, and consequently $\text{Ss}(\mathcal{L}, i, j)$.

Definition 5.9 (Proper substring automaton). Let $A \in \text{DFA}_{/\equiv}$, and $i, j \in \mathbb{N}$ s.t. $i \leq j$. The proper substring automaton of A from i to j is defined as follows.

$$\text{PSS}(A, i, j) \triangleq \text{RQ}(\text{SU}(A, i), \text{SU}(A, j)) \sqcap_{\text{DFA}} \text{Min}(\Sigma^{j-i})$$

For example, let us consider the automaton A such that $\mathcal{L}(A) = \{ a^n \mid n \in \mathbb{N} \} \cup \{ \textit{hello}, bc \}$. The proper substring language from 1 to 3 of $\mathcal{L}(A)$ is $\{ aa, el \}$ and the automaton recognizing that language is correctly computed by Definition 5.9. In particular, the following theorem holds.

Theorem 5.10. Let $A \in \text{DFA}$ and $i, j \in \mathbb{N}$ s.t. $i \leq j$. Then, PSS is complete, namely

$$\text{PSS}(\mathcal{L}(A), i, j) = \mathcal{L}(\text{PSS}(A, i, j))$$

Proof.

$$\begin{aligned} \text{PSS}(\mathcal{L}(A), i, j) &= \\ &= \left\{ y \in \Sigma^* \mid \begin{array}{l} \exists x, z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(A), i) \\ z \in \text{SU}(\mathcal{L}(A), j) \end{array} \right\} && \text{\textcolor{green}\{Def. 5.6\}} \\ &= \left\{ y \in \Sigma^* \mid \begin{array}{l} \exists x, z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(A), i) \\ z \in \text{SU}(\mathcal{L}(A), j), y \in \Sigma^{j-i} \end{array} \right\} && \text{\textcolor{green}\{Def. proper substring\}} \end{aligned}$$

$$\begin{aligned}
&= \left\{ y \in \Sigma^* \mid \begin{array}{l} \exists x, z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(\mathbf{A}), i) \\ z \in \text{SU}(\mathcal{L}(\mathbf{A}), j) \end{array} \right\} \cap \Sigma^{j-i} && \text{[Def. } \Sigma^{j-i}\text{]} \\
&= \text{RQ}(\text{SU}(\mathcal{L}(\mathbf{A}), i), \text{SU}(\mathcal{L}(\mathbf{A}), j)) \cap \Sigma^{j-i} && \text{[Def. 2.44]} \\
&= \mathcal{L}(\text{RQ}(\text{SU}(\mathbf{A}, i), \text{SU}(\mathbf{A}, j)) \cap \text{Min}(\Sigma^{j-i})) && \text{[Thm. 2.50]} \\
&= \mathcal{L}(\text{PSS}(\mathbf{A}, i, j)) && \text{[Def. 5.9]}
\end{aligned}$$

□

Similarly, we can define the non-proper substring automaton, as follows.

Definition 5.11 (Non-proper substring automaton). Let $\mathbf{A} \in \text{DFA}_{/\equiv}$, and $i, j \in \mathbb{N}$ s.t. $i \leq j$. The non-proper substring automaton of \mathbf{A} from i to j is defined as follows.

$$\text{NSS}(\mathbf{A}, i, j) \triangleq (\text{SU}(\mathbf{A}, i) \sqcap_{\text{DFA}} \text{Min}(\Sigma^{<j-i})) \sqcup_{\text{DFA}} (\mathbf{A} \sqcap_{\text{DFA}} \text{Min}(\Sigma^{\leq i}) = \text{Min}(\emptyset) ? \text{Min}(\{\epsilon\}) : \text{Min}(\emptyset))$$

The left-hand side of the automata union computes the non-proper substrings, while the second one, following Definition 5.7, checks if \mathbf{A} recognizes strings of length less or equals to i . If so, also the automaton recognizing the empty string is added to the result. For example, let us consider again the automaton \mathbf{A} such that $\mathcal{L}(\mathbf{A}) = \{ a^n \mid n \in \mathbb{N} \} \cup \{ \text{hello}, bc \}$. The non-proper substring language from 1 to 3 of $\mathcal{L}(\mathbf{A})$ is $\{ \epsilon, a, c \}$ and the automaton recognizing that language is correctly computed by Definition 5.11. Note that the resulting automaton correctly recognizes also the empty string. In particular, the following theorem holds.

Theorem 5.12. Let $\mathbf{A} \in \text{DFA}_{/\equiv}$ and $i, j \in \mathbb{N}$ s.t. $i \leq j$. Then, NSS is complete, namely

$$\text{NSS}(\mathcal{L}(\mathbf{A}), i, j) = \mathcal{L}(\text{NSS}(\mathbf{A}, i, j))$$

Proof.

$$\begin{aligned}
&\text{NSS}(\mathcal{L}(\mathbf{A}), i, j) = \\
&= \{ y \in \Sigma^* \mid y \in \text{SU}(\mathcal{L}(\mathbf{A}), i), y \in \Sigma^{<j-i} \} \cup \{ \epsilon \mid \mathcal{L}(\mathbf{A}) \cap \Sigma^{\leq i} \neq \emptyset \} && \text{[Def. 5.7]} \\
&= (\text{SU}(\mathcal{L}(\mathbf{A}), i) \cap \Sigma^{<j-i}) \cup (\mathcal{L}(\mathbf{A}) \cap \Sigma^{\leq i} \neq \emptyset ? \epsilon : \emptyset) && \text{[Def. } \Sigma^{<j-i}\text{]} \\
&= \mathcal{L}((\text{SU}(\mathbf{A}, i) \\
&\sqcap_{\text{DFA}} \text{Min}(\Sigma^{<j-i})) \sqcup_{\text{DFA}} (\mathbf{A} \sqcap_{\text{DFA}} \text{Min}(\Sigma^{\leq i}) ? \text{Min}(\{\epsilon\}) : \text{Min}(\emptyset))) && \text{[Thm. 2.50]} \\
&= \mathcal{L}(\text{NSS}(\mathbf{A}, i, j)) && \text{[Def. 5.11]}
\end{aligned}$$

□

At this point, given an automaton, we can define the substring automaton between two indexes as union of its proper substring automaton and its non-proper substring automaton.

Definition 5.13 (Substring automaton). Let $\mathbf{A} \in \text{DFA}_{/\equiv}$, and $i, j \in \mathbb{N}$ s.t. $i \leq j$. The substring automaton of \mathbf{A} from i to j is defined as follows.

$$\text{SS}(\mathbf{A}, i, j) \triangleq \text{PSS}(\mathbf{A}, i, j) \sqcup_{\text{DFA}} \text{NSS}(\mathbf{A}, i, j)$$

For example, let us consider again the automaton \mathbf{A} such that $\mathcal{L}(\mathbf{A}) = \{ a^n \mid n \in \mathbb{N} \} \cup \{ \text{hello}, bc \}$. The substring language from 1 to 3 of $\mathcal{L}(\mathbf{A})$ is $\{ aa, el, \epsilon, a, c \}$ and the automaton recognizing that language is correctly computed by Definition 5.13. In particular, the following theorem trivially holds.

Theorem 5.14. Let $A \in \text{DFA}_{/\equiv}$ and $i, j \in \mathbb{N}$ s.t. $i \leq j$. Then, SS is complete, namely

$$\text{SS}(\mathcal{L}(A), i, j) = \mathcal{L}(\text{SS}(A, i, j))$$

Proof. The proof follows from Theorem 5.10 and Theorem 5.12. \square

5.2.2 Substring language after a fixed initial index

We have previously characterized the substring automaton between two indexes i and j of A , namely the automaton recognizing any substring from i to j of any string recognized by A . This operation will be crucial when we will define the abstract interpreter for μJS using finite state automata. In the following, we characterize another type of substring language. For example, let us consider the language $\mathcal{L} = \{\text{hello}\}$ and suppose we want to characterize the language of any substring starting after the index 2, that, in our example, are $\{\epsilon, l, ll, llo, o, lo\}$. In the following, we define the language of any substring starting after a fixed index.

Definition 5.15 (Substring language after a fixed index). Let \mathcal{L} be a regular language and $i \in \mathbb{N}$. The substring language of \mathcal{L} after i of \mathcal{L} is defined as follows.

$$\text{SS}^{\leftrightarrow}(\mathcal{L}, i) \triangleq \{ y \mid \exists z \in \Sigma^*. a \in \mathbb{N}, yz \in \text{SU}(\mathcal{L}, a), a \geq i \}$$

As we have already done before, we want to compute the automaton recognizing the language we have characterized in Definition 5.15.

Definition 5.16 (Substring automaton after a fixed index). Let $A \in \text{DFA}_{/\equiv}$, and $i \in \mathbb{N}$. The substring automaton of A after i is defined as follows.

$$\text{SS}^{\leftrightarrow}(A, i) \triangleq \text{FA}(\text{SU}(A, i))$$

Next theorem shows that $\text{SS}^{\leftrightarrow}$ exactly computes the substring language after a fixed index of an automaton.

Theorem 5.17. Let $A \in \text{DFA}_{/\equiv}$ and $i \in \mathbb{N}$. Then, $\text{SS}^{\leftrightarrow}$ is complete, namely

$$\text{SS}^{\leftrightarrow}(\mathcal{L}(A), i) = \mathcal{L}(\text{SS}^{\leftrightarrow}(A, i))$$

Proof.

$$\begin{aligned} & \text{SS}^{\leftrightarrow}(\mathcal{L}(A), i) \\ &= \{ y \mid \exists z \in \Sigma^*. a \in \mathbb{N}, yz \in \text{SU}(\mathcal{L}(A), a), a \geq i \} && \text{[Def. 5.15]} \\ &= \{ y \mid \exists z \in \Sigma^*. yz \in \text{SU}(\text{SU}(\mathcal{L}(A), i)) \} && \text{[Eq. 2.1]} \\ &= \text{PR}(\text{SU}(\text{SU}(\mathcal{L}(A), i))) && \text{[Def. 2.42]} \\ &= \text{FA}(\text{SU}(\mathcal{L}(A), i)) && \text{[Def. 2.46]} \\ &= \mathcal{L}(\text{FA}(\text{SU}(A, i))) && \text{[Thm. 2.50]} \\ &= \mathcal{L}(\text{SS}^{\leftrightarrow}(A, i)) && \text{[Def. 5.16]} \end{aligned}$$

\square

5.2.3 Substring language to an unbounded final index

The last type of substring language we introduce is the substring language between a fixed initial index and a final index starting after a certain point. For example, let us

consider the language $\mathcal{L} = \{hello\}$ and suppose we want to know the language of its substrings between 1 and any index greater than 2. This language, corresponds to $\{el, ell, ello\}$, that is the substrings between 1 and 3, 1 and 4, 1 and 5. The following definition characterizes such language.

Definition 5.18 (Substring language to undefined final index). Let \mathcal{L} be a regular language and $i, j \in \mathbb{N}$ s.t. $i \leq j$. The substring language from i to any index greater or equal to j of \mathcal{L} is defined as follows.

$$SS^{\rightarrow}(\mathcal{L}, i, j) \triangleq \{ y \mid \exists x, z \in \Sigma^* \mid |x| = i, |z| \geq i + j, xyz \in \mathcal{L} \}$$

Note that, we can also characterize $SS^{\rightarrow}(\mathcal{L}, i, j)$ as follows:

$$SS^{\rightarrow}(\mathcal{L}, i, j) = \bigcup_{k \geq j} SS(\mathcal{L}, i, k) \quad (5.1)$$

As before, our aim is to compute the automata recognizing the language we have characterized in Definition 5.18.

Definition 5.19 (Substring automaton to undefined final index). Let $A \in \text{DFA}_{/\equiv}$, and $i, j \in \mathbb{N}$ s.t. $i \leq j$. The substring automaton from i to any index greater or equal to j of A is defined as follows.

$$SS^{\rightarrow}(A, i, j) \triangleq \text{RQ}(\text{SU}(A, i), \text{SU}(\text{SU}(A, j)))$$

Next theorem shows that SS^{\rightarrow} exactly computes the substring language to undefined final index.

Theorem 5.20. Let $A \in \text{DFA}$ and $i, j \in \mathbb{N}$ s.t. $i \leq j$. Then, SS^{\rightarrow} is complete, namely

$$SS^{\rightarrow}(\mathcal{L}(A), i, j) = \mathcal{L}(SS^{\rightarrow}(A, i, j))$$

Proof.

$$\begin{aligned} SS^{\rightarrow}(\mathcal{L}(A), i, j) &= \\ &= \{ y \mid \exists x, z \in \Sigma^*. |x| = i, |z| \geq i + j, xyz \in \mathcal{L}(A) \} && \text{\{Def. 5.18\}} \\ &= \{ y \mid \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(A), i), |z| \geq i + j \} && \text{\{Def. 2.45\}} \\ &= \{ y \mid \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(A), i), z \in \text{SU}(\text{SU}(\mathcal{L}(A), i), i + j) \} && \text{\{Def. 2.42\}} \\ &= \{ y \mid \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(A), i), z \in \text{SU}(\text{SU}(\mathcal{L}(A), j)) \} && \text{\{Eq. 2.1\}} \\ &= \text{RQ}(\text{SU}(\mathcal{L}(A), i), \text{SU}(\text{SU}(\mathcal{L}(A), j))) && \text{\{Def. 2.44\}} \\ &= \mathcal{L}(\text{RQ}(\text{SU}(A, i), \text{SU}(\text{SU}(A, j)))) && \text{\{Thm. 2.50\}} \\ &= \mathcal{L}(SS^{\rightarrow}(A, i, j)) && \text{\{Def. 5.19\}} \end{aligned}$$

□

Theorem 5.21. Regular languages are closed under SS , SS^{\leftrightarrow} and SS^{\rightarrow} .

Proof. The proof follows by Theorem 5.14, Theorem 5.17 and Theorem 5.20. □

In this section, we have defined three classes of languages, that is the substring language, the substring language from a certain index and substring languages between a fixed initial index to an undefined final index. For each language, we have also defined the corresponding operator on finite state automata that computes the automaton recognizing each language. The automata operators SS , SS^{\leftrightarrow} and SS^{\rightarrow}

will play a crucial role when we will talk about the abstract semantics of `substr`, in our abstract interpreter that we will define in the next chapter.

Implementation and final remarks. The finite state automata abstract domain presented in this chapter has been implemented and it is publicly available at <https://github.com/SPY-Lab/java-fsm-library>. The library includes the implementation of all the algorithms reported in Section 2.6, such as suffix, prefix, right quotient, left quotient operations. Moreover, the library provides minimization and determinization algorithms discussed in Section 2.6 and the abstract-domain related operations discussed in this chapter, such as least upper bound \sqcup_{DFA} , greatest lower bound \sqcap_{DFA} and the parametric widening $\nabla_{\text{DFA}/\equiv}^n$, for tuning precision and forcing convergence in abstract computations. Moreover, it provides also all the substring automata operations reported in this chapter, namely SS , $\text{SS}^{\leftrightarrow}$ and SS^{\rightarrow} . The abstract semantics reported in this paper often relies on minimization of finite state automata, in order to keep the automata arising during abstract computations determinized and minimized. In general, minimization has exponential complexity but this is not a concern. Indeed, our library relies on the Brzozowski's algorithm, since in practice it is extremely fast on average and consistently outperforms other minimization algorithms (e.g., Hopcroft's algorithm [Hopcroft, Motwani, and Ullman, 2007], having average-case complexity $O(n \log n)$, where n is the number of states) as reported in [Holzer and Jakobi, 2013]. Moreover, the minimization is only applied when the input automaton is not-deterministic. The finite state automata abstract domain is the core of the abstract domain that we will present in the next chapter. In particular, we will show that the finite structure of finite state automata lends itself well to both perform a precise string static analysis and dynamically-generated code analysis, corresponding to the core of the contribution of this thesis.

Chapter 6

A sound abstract interpreter for dynamic code

In this chapter, we go into detail of the main contribution of this thesis, that is an abstract interpreter for string manipulation languages with dynamic code generation. As we have already mentioned in the introduction, dynamic languages employ string-to-code primitives to turn dynamically generated text into executable code at run-time. These features make standard static analysis (presented in Chapter 3) 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.

In this chapter, we augment the syntax of μJS with a string-to-code statement, namely `eval`, that takes as input a string and executes the code expressed by its input. It should be clear that an analysis of `eval` is strongly dependent on the choice of the performed string analysis. Hence, we first describe a precise string analysis based on the finite state automata defined in Chapter 5. After that, we propose a static analysis for `eval` exploiting the finite state automata abstract domain. Our goal is to use the automata domain allowing us to soundly over-approximate and analyze the code potentially executed by a string-to-code statement.

6.1 μJS with `eval`

In this section, we augment the syntax of μJS with a string-to-code statement, namely `eval`, and then we define the concrete semantics of `eval`. In particular, we add `eval` syntax to the statement syntax of μJS as

$$\text{st} \in \text{STMT} ::= \dots \mid \ell_1 \text{eval}(s) \ell_2$$

taking as input a string expression $s \in \text{SE}$. We also extend the big-step semantics of μJS extending the function $\llbracket \text{eval}(s) \rrbracket : \text{STATE} \rightarrow \text{STATE}$ as

$$\llbracket \text{eval}(s) \rrbracket \zeta = \begin{cases} \llbracket \text{st} \rrbracket \zeta & \text{if } \text{st} = \text{code}(\llbracket s \rrbracket \zeta) \\ \zeta & \text{otherwise} \end{cases}$$

where $\text{code} : \text{STR} \rightarrow \mu\text{JS}$ converts a string to the corresponding μJS program, if it is executable. For example, $\text{code}(a = 1;) = a = 1;$. Hence, the semantics of `eval` evaluates its input and then it interprets the string as code, if possible, and it execute that code. For example, $\llbracket \text{eval}("x=1") \rrbracket \zeta = \llbracket x=1; \rrbracket \zeta$ and $\llbracket \text{eval}("hello") \rrbracket \zeta = \zeta$. For the sake of simplicity, when the `eval` input string is non-executable, the semantics behaves as a no-op statement (i.e., `skip`), propagating the input state, unlike the JavaScript semantics that throws a syntax error.

As we have already discussed in Chapter 3, our aim is to reason about a μJS program analyzing its corresponding control-flow graph. Hence, we extend the function *Edges* in order to handle also *eval*.

$$\text{Edges}(\ell_1 \text{eval}(s) \ell_2) = \{\ell_1, \text{eval}(s), \ell_2\}$$

Moreover, from the control-flow graph construction, also the grammar of edges labels changes.

$$\mu\text{JS-CFG} \ni 1 ::= \dots \mid \text{eval}(s)$$

Consequently, we need to also extend the collecting semantics of statement *st* \in $\mu\text{JS-CFG}$, defined in Section 3.2 as the function $\llbracket \text{st} \rrbracket : \mathbb{M} \rightarrow \mathbb{M}$.

$$\begin{aligned} \llbracket \text{eval}(s) \rrbracket m &= \bigcup_{c \in C} \llbracket c \rrbracket m \\ \text{where } C &= \{ \text{code}(\sigma) \in \mu\text{JS} \mid \text{code}(\sigma) : \text{skip} \mid \sigma \in \llbracket e \rrbracket m \} \end{aligned}$$

We have defined and extended the collecting semantics of μJS . But, as we have already mentioned before, we need abstraction to guarantee convergence in analyzing μJS programs. Unfortunately, this is sufficient to avoid divergence when the code is static (namely without string-to-code statements), but when code is dynamic other aspects of computation, not controllable by data abstraction, may cause divergence. It is worth noting that the collection of potential executable strings reaching an *eval* argument may be infinite, and this implies that, precisely as it happens for data values, we need to abstract also code (by suitable finite representations of potential infinite programs) in order to be able to enforce convergence by losing precision.

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 code below.

```

1x = "eval(x)";
2eval(x);3

```

In this case, the second line activates 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 the string-to-code statement. In the following, in order to build a static analyzer for μJS , we tackle three problems separately, in particular:

- we build a suitable data abstraction, and the corresponding abstract semantics, preparing the field for analyzing *eval* (Section 6.2);
- we provide an over-approximation of the code executed by *eval*. Once we have such abstraction, we recursively call the abstract interpreter on the abstracted code (Section 6.5);
- finally, we provide a sound solution to control nested *eval* calls (Section 6.5.3).

6.2 Dyn: An abstract domain for μJS

In order to solve the first standard source of divergence, we have to consider a suitable abstraction of data. In particular, we have to combine an abstraction of numerical values, of boolean, of NaN and of strings. For the first three data types, the choice is not relevant in presence of string-to-code statements such as *eval*, except for tuning precision. We have decided to abstract integers value to the well-known interval

domain Ints for numerical values [Cousot and Cousot, 1977] and boolean values and NaN value to their identities, i.e., $\text{Bool} = \{\perp_{\text{Bool}}, \text{true}, \text{false}, \top_{\text{Bool}}\}$, with γ_{Bool} denoting its concretization function, and $\text{NaN} = \{\perp_{\text{NaN}}, \text{NaN}\}$, γ_{NaN} denoting its concretization function. As far as strings are concerned, we use the finite state automata abstract domain $\text{DFA}_{/\equiv}$ described in Chapter 5, since DFA are enough precise for analyzing string properties in general, and since their finite representation is suitable for building algorithms able to extract/approximate the executable sub-language of the string when necessary, namely in presence of string-to-code statements. We will present and discuss these algorithms in the next sections.

Further, we need to combine these abstract domains, in order to obtain a composed abstract domain abstracting values in the collecting domain, namely $\text{VAL} = \wp(\text{INT}) \cup \wp(\text{BOOL}) \cup \wp(\text{STR}) \cup \wp(\{\text{NaN}\}) \cup \wp(\{\uparrow\})$. There exist several ways to combine abstract domains, such as by applying Cartesian product, direct product or reduced product operators [Cortesi, Costantini, and Ferrara, 2013]. In our abstract interpreter, we combine our abstract domains by coalesced sum (also called smashed sum).

Definition 6.1 (Coalesced sum domain [Reynolds, 1998]). Let $\langle A, \leq_A, \sqcup_A, \sqcap_A, \perp_A, \top_A \rangle$ and $\langle B, \leq_B, \sqcup_B, \sqcap_B, \perp_B, \top_B \rangle$ be two domains. The coalesced sum domain $A \oplus B$ is defined as

$$A \oplus B \triangleq \{\perp_{A \oplus B}\} \cup \{A \setminus \{\perp_A\}\} \cup \{B \setminus \{\perp_B\}\} \cup \{\top_{A \oplus B}\}$$

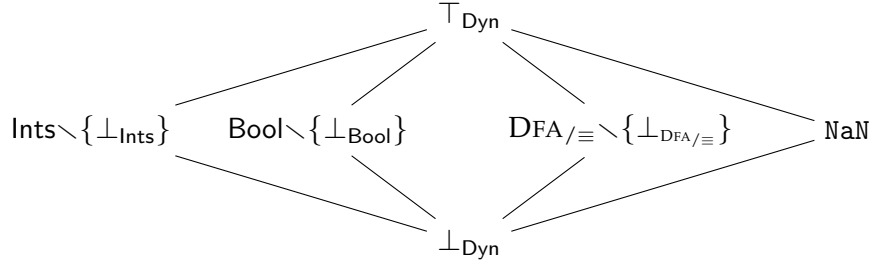
such that the partial order is defined as $x \leq_{A \oplus B} y \Leftrightarrow x \leq_A y \ (x, y \in A) \vee x \leq_B y \ (x, y \in B)$ and $\forall x \in A \oplus B. \perp_{A \oplus B} \leq_{A \oplus B} x \leq_{A \oplus B} \top_{A \oplus B}$, its least upper bound is defined as

$$x \sqcup_{A \oplus B} y \triangleq \begin{cases} x \sqcup_A y & \text{if } x, y \in A \\ x \sqcup_B y & \text{if } x, y \in B \\ \top_{A \oplus B} & \text{otherwise} \end{cases}$$

and its greatest lower bound as

$$x \sqcap_{A \oplus B} y \triangleq \begin{cases} x \sqcap_A y & \text{if } x, y \in A \wedge x \sqcap_A y \neq \perp_A \\ x \sqcap_B y & \text{if } x, y \in B \wedge x \sqcap_B y \neq \perp_B \\ \perp_{A \oplus B} & \text{otherwise} \end{cases}$$

The name of this domain composition comes from the fact that a new bottom element is added to the domain ($\perp_{A \oplus B}$) and it *smashes/coalesces* the bottom elements of A and B together. It is worth noting that coalesced sum domains are able to precisely track a variable value if it has constant type: for example, let us consider the following μJS program `if (y < 5){x = 42} else {x = true}`; and suppose to abstractly execute it with a coalesced sum abstract domain, where integers are abstracted into intervals. The value of the variable y is statically unknown, hence we must take into account both the branches. The `true`-branch body abstracts the value of x to the interval $[42, 42]$ and the `false`-branch body abstracts the value of x to `true`. Hence, in order to answer about the value of x at the end of the `if` statement, these values must be lubbed into the coalesced sum abstract domain, namely $[42, 42] \sqcup \text{true} = \top$. In presence of dynamic typing, such as in μJS and in any dynamic programming language, the coalesced sum abstract domain is not a good choice. Indeed, variables can change type during program execution and coalesced sum abstract domains easily lose any information about variable values.

FIGURE 6.1: Coalesced sum abstract domain for μJS

A better choice to master dynamic typing is Cartesian product abstract domain [Cortesi, Costantini, and Ferrara, 2013], integrated into several static analyses for dynamic languages [Jensen, Møller, and Thiemann, 2009; Fromherz, Ouadjaout, and Miné, 2018; Arceri and Maffei, 2017; Arceri and Mastroeni, 2019; Hauzar and Kofron, 2015a]. In order to catch union types, we need to complete [Giacobazzi, Ranzato, and Scozzari, 2000; Giacobazzi and Quintarelli, 2001; Giacobazzi and Mastroeni, 2016] Dyn domain in order to observe collections of values of different types. This combination is captured by Cartesian product composition.

Definition 6.2 (Cartesian product domain [Cortesi, Costantini, and Ferrara, 2013]). Let $\langle A, \leq_A, \sqcup_A, \sqcap_A, \perp_A, \top_A \rangle$ and $\langle B, \leq_B, \sqcup_B, \sqcap_B, \perp_B, \top_B \rangle$ be two domains. The Cartesian product domain $A \times B$ is defined as follows.

$$A \times B \triangleq \{ \langle a, b \rangle \mid a \in A, b \in B \}$$

such that the partial order is defined as $\langle a, b \rangle \leq_{A \times B} \langle a', b' \rangle \Leftrightarrow a \leq_A a' \wedge b \leq_B b'$, its least upper bound is defined as $\langle a, b \rangle \sqcup_{A \times B} \langle a', b' \rangle \triangleq \langle a \sqcup_A a', b \sqcup_B b' \rangle$ and its greatest lower bound is defined as $\langle a, b \rangle \sqcap_{A \times B} \langle a', b' \rangle \triangleq \langle a \sqcap_A a', b \sqcap_B b' \rangle$

Hence, as shown in Arceri and Maffei, 2017; Arceri and Mastroeni, 2019, the Cartesian product abstract domain is able to capture union types, being complete w.r.t. dynamic typing. Nevertheless, the choice of adapting a particular domain composition does not affect the eval analysis we will explain in the next sections. For this reason, in order to lighten the presentation of the novel approach, we compose the singleton value abstract domains of μJS by coalesced sum, abstracting VAL, as follows.

$$\text{Dyn} = \text{DFA}_{/≡} \oplus \text{Ints} \oplus \text{Bool} \oplus \text{NaN} \quad (6.1)$$

In Figure 6.1, Dyn is graphically reported and the concretization function $\gamma_{\text{Dyn}} : \text{Dyn} \rightarrow \text{VAL}$ is reported in the following.

$$\gamma_{\text{Dyn}}(a) \triangleq \begin{cases} \{\uparrow\} & \text{if } a = \perp_{\text{Dyn}} \\ \gamma_{\text{Ints}}(a) & \text{if } a \in \text{Ints} \\ \gamma_{\text{Bool}}(a) & \text{if } a \in \text{Bool} \\ \gamma_{\text{DFA}}(a) & \text{if } a \in \text{DFA}_{/≡} \\ \gamma_{\text{NaN}}(a) & \text{if } a \in \text{NaN} \\ \text{VAL} & \text{otherwise} \end{cases}$$

Moreover, we denote by $\sqcup_{\text{Dyn}}, \sqcap_{\text{Dyn}} : \text{Dyn} \times \text{Dyn} \rightarrow \text{Dyn}$ the least upper bound and the greatest lower bound, respectively, and by \sqsubseteq_{Dyn} the partial order on Dyn.

Since Ints and $\text{DFA}_{/\equiv}$ are not ACC, also Dyn is not ACC. As we have observed before, the static analysis of a μJS program control-flow graph with a non-ACC abstract domain not equipped with a widening, may diverge. Hence, we equip Dyn with the widening operator $\nabla_{\text{Dyn}}^n : \text{Dyn} \times \text{Dyn} \rightarrow \text{Dyn}$, with $n \in \mathbb{N}$, defined as

$$a \nabla_{\text{Dyn}}^n b \triangleq \begin{cases} a \nabla_{\text{Ints}} b & \text{if } a, b \in \text{Ints} \\ a \nabla_{\text{DFA}_{/\equiv}}^n b & \text{if } a, b \in \text{DFA}_{/\equiv} \\ a \sqcup_{\text{Bool}} b & \text{if } a, b \in \text{Bool} \\ \text{NaN} & \text{if } a, b \in \text{NaN} \\ \top_{\text{Dyn}} & \text{otherwise} \end{cases}$$

The widening operator ∇_{Dyn}^n is defined point-wise. Since boolean abstract domain is ACC, it does not need a widening for ensuring termination and it simply lubs the operands. Finally, the parameter $n \in \mathbb{N}$ is needed only by the $\text{DFA}_{/\equiv}$ widening for tuning the precision, as explained in Chapter 5.

In the following, we define the abstract semantics of μJS that used the coalesced sum abstract domain just defined. Since we perform static analysis of μJS programs by analyzing their control-flow graphs, it is enough to define the abstract semantics of the edge labels of a control-flow graph, namely of $\mu\text{JS}\text{-CFG}$, and the abstract semantics of expressions.

6.2.1 Abstract semantics of μJS

Following the concrete semantics defined in Section 3.2, we denote the set of abstract memories by $\mathbb{M}^\#$, ranged over $\mathfrak{m}^\#$, associating with variables values in the abstract domain Dyn . Similarly to the concrete counterpart, the update of an abstract memory $\mathfrak{m}^\#$ for $x \in \text{ID}$ with the abstract value $v \in \text{Dyn}$ is denoted by $\mathfrak{m}^\#[x/v]$, while lub and glb of memories are defined point-wise, namely $\mathfrak{m}^\#_1 \sqcup_{\mathfrak{m}^\#} \mathfrak{m}^\#_2(x) = \mathfrak{m}^\#_1(x) \sqcup_{\text{Dyn}} \mathfrak{m}^\#_2(x)$ and $\mathfrak{m}^\#_1 \sqcap_{\mathfrak{m}^\#} \mathfrak{m}^\#_2(x) = \mathfrak{m}^\#_1(x) \sqcap_{\text{Dyn}} \mathfrak{m}^\#_2(x)$. The abstract semantics of $\mu\text{JS}\text{-CFG}$ is captured by the function $\langle \text{st} \rangle^\# : \mathbb{M}^\# \rightarrow \mathbb{M}^\#$ and the one of assignments, skip and boolean expressions is standard and it is reported below.

$$\begin{aligned} \langle x = e \rangle^\# \mathfrak{m}^\# &= \mathfrak{m}^\#[x / \langle e \rangle^\# \mathfrak{m}] \\ \langle \text{skip} \rangle^\# \mathfrak{m}^\# &= \mathfrak{m}^\# \\ \langle b \rangle^\# \mathfrak{m}^\# &= \mathfrak{m}^\# \sqcap_{\mathfrak{m}^\#} \bigsqcup^\# \{ \mathfrak{m}^\#_t \mid \langle b \rangle^\# \mathfrak{m}^\#_t = \text{true} \} \end{aligned}$$

We abuse notation denoting by $\langle e \rangle^\# : \mathbb{M}^\# \rightarrow \text{Dyn}$ the abstract semantics of expressions. Similarly to the concrete counterpart, we define a collection of abstract stores for each program point, that is *flow-sensitive* abstract store, ranged over $\mathfrak{s}^\#$, elements of $\mathbb{S}^\# : \text{Lab}_P \rightarrow \mathbb{M}^\#$. Let us observe how Equation 3.1 is rewritten on the abstract semantics

$$\forall \mathfrak{m}^\# \in \mathbb{M}^\#. \exists \Pi \subseteq \text{Paths}(\text{G}_P). \langle P \rangle^\# \mathfrak{m}^\# \subseteq \bigcup_{\pi \in \Pi} \langle \pi \rangle^\# \mathfrak{m}^\# \quad (6.2)$$

At this point, in order to integrate the abstract domain Dyn into an abstract interpreter (that uses the static analysis algorithm reported in Algorithm 9), we need to define the abstract semantics of string operations. While the abstract semantics of the other operations, e.g., numerical and boolean operations, are quite standard, the abstract semantics of string operations require more attention. In particular, μJS

provides five string operations: `substring`, `charAt`, `indexOf`, `length` and `concat`. In the following, we provide the abstract semantics of these string operations. Finally, we will also define the abstraction of the implicit type conversion functions reported in Section 3.

Abstract semantics of `substring`. Here, we will define the abstract semantics of `substring`. In particular, following its concrete semantics in Section 3, we define the function $SS^\# : \text{DFA}_{/\equiv} \times \text{Ints} \times \text{Ints} \rightarrow \text{DFA}_{/\equiv}$, taking as input an automaton and two intervals, corresponding to the interval of initial indexes and the interval of final indexes, respectively. The results is an automaton recognizing the set of all substrings of the input automata language between the indexes in the two intervals. In Section 5.2, we have defined several automata substring operators, taking as input two fixed indexes. The idea is to lift those operators on set of integers, namely intervals. Unfortunately, this is not enough since we also need to face two main problems: intervals may contain infinite indexes (e.g., $[5, +\infty]$) and we need to take into account the swaps of indexes, following the μJS semantics, when the initial index is greater than the final one¹. For these reasons, several cases arise in defining the abstract semantics of `substring`.

Let $A \in \text{DFA}_{/\equiv}$ be the input automaton, $[i, j] \in \text{Ints}$ the interval of the initial indexes, and $[l, k] \in \text{Ints}$ the set of final indexes. We define the function $SS^\#$ recursively defined with four base cases, while the others are recursive call splitting and rewriting the input intervals in order to match or to get closer to base cases. Table 6.1 reports the abstract semantics of $SS^\#$ when $i, j \leq l$ (hence $i \leq k$). Without loss of generality, when a negative index is met ($i, j, l, k < 0$), we suppose that the index is implicitly treated as 0 (following the definition of `substring` given in Chapter 3).

1. If $i, j, l, k \in \mathbb{Z}$ (second row, second column of Table 6.1) we have to compute the language of all the substrings between an initial index in $[i, j]$ and a final index in $[l, k]$. For example, let $\mathcal{L} = \{a\}^* \cup \{\text{hello}, bc\}$, the set of its substrings from 1 to 3 is $SS(\mathcal{L}, [1, 1], [3, 3]) = \{\epsilon, a, aa, el, c\}$. In this case, since both intervals are bounded, we can rely on the substring operator defined in Definition 5.13, lifting it to intervals and taking into account swaps. Hence, the abstract semantics is defined as

$$SS^\#(A, [i, j], [l, k]) \triangleq \bigsqcup_{a \in [i, j], b \in [l, k]} SS(A, \min(\{a, b\}), \max(\{a, b\})) \quad (6.3)$$

In order to handle the `substring` swaps, for each element $a \in [i, j]$ and $b \in [l, k]$ we apply SS on A between the minimum and the maximum of a and b ;

2. Since any negative index is treated as 0, in the `substring` semantics, when both intervals correspond to $[-\infty, +\infty]$ (fifth row, fifth column), they can be seen as $[0, +\infty]$. Hence, the result is the automaton recognizing any possible substring of strings accepted by the input automaton, namely the result is $FA(A)$;
3. If $[i, j]$ is bounded and the interval of final indexes is unbounded, i.e., $[l, +\infty]$ (second row, fourth column), we have to compute the automaton recognizing the language of all the substrings between a finite interval of initial indexes and an unbounded final index. In this case, we can rely on the `substring` operation

¹We recall that, for example, `substring("hello", 3, 1) = substring("hello", 1, 3)`.

$\text{SS}^\#(\mathbf{A}, [i, j], [l, k])$ $i, j \leq l$ ($i \leq k$)	$l, k \in \mathbb{Z}$	$l = -\infty$ $k \in \mathbb{Z}$	$l \in \mathbb{Z}$ $k = +\infty$	$l = -\infty$ $k = +\infty$
$i, j \in \mathbb{Z}$	Eq. 6.3	$\text{SS}^\#(\mathbf{A}, [i, j], [0, k])$	$\bigsqcup_{k \in [i, j]} \text{SS}^\rightarrow(\mathbf{A}, k, l)$	$\text{SS}^\#(\mathbf{A}, [i, j], [0, +\infty])$
$i = -\infty$ $j \in \mathbb{Z}$	$\text{SS}^\#(\mathbf{A}, [0, j], [l, k])$	$\text{SS}^\#(\mathbf{A}, [0, j], [0, k])$	$\text{SS}^\#(\mathbf{A}, [0, j], [l, +\infty])$	$\text{SS}^\#(\mathbf{A}, [0, j], [0, +\infty])$
$i \in \mathbb{Z}$ $j = +\infty$	$\text{SS}^\#(\mathbf{A}, [l, k], [k, +\infty])$ $\sqcup \text{SS}^\#(\mathbf{A}, [i, k], [l, k])$	$\text{SS}^\#(\mathbf{A}, [i, +\infty], [0, k])$	$\text{SS}^\rightarrow(\mathbf{A}, [i, l], l) \sqcup \text{SS}^{\leftrightarrow}(\mathbf{A}, l)$	$\text{SS}(\mathbf{A}, [i, +\infty], [0, +\infty])$
$i = -\infty$ $j = +\infty$	$\text{SS}^\#(\mathbf{A}, [0, +\infty], [l, k])$	$\text{SS}^\#(\mathbf{A}, [0, +\infty], [0, k])$	$\text{SS}^\#(\mathbf{A}, [0, +\infty], [l, +\infty])$	$\text{FA}(\mathbf{A})$

TABLE 6.1: Definition of $\text{SS}^\#$ when $i, j \leq l$ (and thus $i \leq k$)

to an undefined final index, namely SS^\rightarrow defined in Section 5.2.

$$\text{SS}^\#(\mathbf{A}, [i, j], [l, +\infty]) \triangleq \bigsqcup_{k \in [i, j]} \text{SS}^\rightarrow(\mathbf{A}, k, l)$$

In this case, the abstract semantics returns the least upper bound of all the automata of substrings from k in $[i, j]$ to an unbounded index greater than or equal to l ;

4. When both intervals are unbounded ($[i, +\infty]$ and $[l, +\infty]$, forth row, forth column of Table 6.1), we split the language to accept. In particular, we compute the substrings between $[i, l]$ and $[l, +\infty]$ (and this has been considered in case 3), and the automaton recognizing the language of all substrings with both initial and final index greater than l , i.e., the language $\text{SS}^{\leftrightarrow}(\mathbf{A}, l)$, defined in Definition 5.16.

We have shown the abstract semantics of substring, captured by $\text{SS}^\#$, only for the case $i, j \leq l$ (and thus $i \leq k$). The remaining cases, namely when $i > l, i \leq k$ and $i > k \vee (i \leq l, j > l)$ are reported in Table 6.2 and Table 6.3, respectively. Even in these cases, the semantics splits and rewrites the input intervals reducing each recursive call to an already defined case.

Let $\text{Ss}(\mathcal{L}, [i, j], [l, k]) \triangleq \{ \text{Ss}(\mathcal{L}, \min(\{a, b\}), \max(\{a, b\})) \mid a \in [i, j], b \in [l, k] \}$ be the collecting semantics of substring (obtained lifting integers to intervals in Definition 5.8 and taking into account the indexes swaps).

Theorem 6.3 (Soundness and completeness of $\text{SS}^\#$). *$\text{SS}^\#$ is sound and complete, formally*

$$\forall \mathbf{A} \in \text{DFA}_{/\equiv}, [i, j], [l, k] \in \text{Ints}. \text{Ss}(\mathcal{L}(\mathbf{A}), [i, j], [l, k]) = \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, j], [l, k]))$$

$\text{SS}^\#(\mathbf{A}, [i, j], [l, k])$ $l < i \leq k$	$l, k \in \mathbb{Z}$	$l = -\infty$ $k \in \mathbb{Z}$	$l \in \mathbb{Z}$ $k = +\infty$	$l = -\infty$ $k = +\infty$
$i, j \in \mathbb{Z}$	Table 6.1	Table 6.1	$\text{SS}^\#(\mathbf{A}, [l, j], [i, j]) \sqcup_{\text{DFA}} \text{SS}^\#(\mathbf{A}, [i, j], [j, +\infty])$	Table 6.1
$i = -\infty$ $j \in \mathbb{Z}$	Table 6.1	Table 6.1	Table 6.1	Table 6.1
$i \in \mathbb{Z}$ $j = +\infty$	$\text{SS}^\#(\mathbf{A}, [l, k], [i, k]) \sqcup_{\text{DFA}} \text{SS}^\#(\mathbf{A}, [l, i], [i, +\infty]) \sqcup_{\text{DFA}} \text{SS}^\#(\mathbf{A}, [i, k], [k, +\infty])$	Table 6.1	$\text{SS}^\#(\mathbf{A}, [l, +\infty], [i, +\infty])$	Table 6.1
$i = -\infty$ $j = +\infty$	Table 6.1	Table 6.1	Table 6.1	Table 6.1

TABLE 6.2: Definition of $\text{SS}^\#$ when $i > l, i \leq k$.

$SS^\#(\mathbf{A}, [i, j], [l, k])$ $i > k \vee (i \leq l, j > l)$	$l, k \in \mathbb{Z}$	$l = -\infty$ $k \in \mathbb{Z}$	$l \in \mathbb{Z}$ $k = +\infty$	$l = -\infty$ $k = +\infty$
$i, j \in \mathbb{Z}$	Table 6.1	Table 6.1	$i \leq l, j > l$ $SS^\#(\mathbf{A}, [l, j], [l, j]) \sqcup_{\text{DFA}}$ $SS^\#(\mathbf{A}, [i, l], [l, +\infty]) \sqcup_{\text{DFA}}$ $SS^\#(\mathbf{A}, [l, j], [j, +\infty])$	Table 6.1
$i = -\infty$ $j \in \mathbb{Z}$	Table 6.1	Table 6.1	Table 6.1	Table 6.1
$i \in \mathbb{Z}$ $j = +\infty$	$i > k$ $SS^\#(\mathbf{A}, [l, k], [i, +\infty])$	Table 6.1	if $i \leq l$ Table 6.1; if $i > l$ Table 6.2	Table 6.1
$i = -\infty$ $j = +\infty$	Table 6.1	Table 6.1	Table 6.1	Table 6.1

TABLE 6.3: Definition of $SS^\#$ when $i > k \vee (i \leq l, j > l)$

Abstract semantics of charAt. The abstract semantics of charAt should return the automaton accepting the language of all the characters of strings accepted by an automaton \mathbf{A} , in a position inside a given interval $[i, j]$: This is computed by the function $CA^\# : \text{DFA}_{/\equiv} \times \text{Ints} \rightarrow \text{DFA}_{/\equiv}$, defined in the following.

$$CA^\#(\mathbf{A}, [i, j]) \triangleq \begin{cases} \bigsqcup_{k \in [i, j]} SS^\#(\mathbf{A}, [k, k], [k + 1, k + 1]) & i, j \in \mathbb{Z} \\ CA^\#(\mathbf{A}, [0, j]) \sqcup_{\text{DFA}} \text{Min}(\{\epsilon\}) & i = -\infty, j \in \mathbb{Z}, j \geq 0 \\ \text{Min}(\{\epsilon\}) & i = -\infty, j \in \mathbb{Z}, j < 0 \\ \text{FA}(\text{SU}(\mathbf{A}, i)) \sqcap_{\text{DFA}} \text{Min}(\Sigma^{\leq 1}) & i \in \mathbb{Z}, i \geq 0, j = +\infty \\ \text{FA}(\mathbf{A}) \sqcap_{\text{DFA}} \text{Min}(\Sigma_{\leq 1}) & i = -\infty \text{ or } i \in \mathbb{Z}, i < 0, j = +\infty \end{cases}$$

When the interval index $[i, j]$ is finite, we rely on the already defined function $SS^\#$. When $l < 0$ or $l = -\infty$ and $h \in \mathbb{Z}$ is positive, we restrict the interval recursively calling $CA^\#$ only on the positive values of the interval, adding $\text{Min}(\{\epsilon\})$ as the result of the negative cases. A particular case is when $[i, j] \subseteq [-\infty, -1]$, where no valid index is inside the interval. In this case, we simply return $\text{Min}(\{\epsilon\})$. When $i \in \mathbb{Z}, j = +\infty$, we return the automaton recognizing any character starting from i of \mathbf{A} . Note that also the empty string is recognized by the resulting automaton (since for any $\mathcal{L} \in \wp(\Sigma^*), \epsilon \in \text{FA}(\mathcal{L})$). In the last case, when $i = -\infty$ or negative and $j = +\infty$, we return the automaton recognizing any substring of \mathbf{A} (computed by FA) of length less or equal than 1. Let $CA(\mathcal{L}, [i, j]) \triangleq \{ \text{Ss}(\sigma, k, k + 1) \mid \sigma \in \mathcal{L}, k \in [i, j] \}$ be the collecting semantics of charAt.

Theorem 6.4. $CA^\#$ is sound and complete: $\forall \mathbf{A} \in \text{DFA}_{/\equiv}, [i, j] \in \text{Ints}$,

$$CA(\mathcal{L}(\mathbf{A}), [i, j]) = \mathcal{L}(CA^\#(\mathbf{A}, [i, j]))$$

Abstract semantics of length. The abstract semantics of length should return the interval of all the possible string lengths in an automaton, i.e., it is $LE^\# : \text{DFA}_{/\equiv} \rightarrow \text{Ints}$ computed by Algorithm 10, where $\text{minPath}, \text{maxPath} : \text{DFA}_{/\equiv} \times Q \times Q \rightarrow \wp(Q)$ return the minimum and the maximum paths between two states of the input automaton, respectively. $\text{len} : \wp(Q) \rightarrow \mathbb{N}$ returns the size of a path, and $\text{hasCycle} : \text{DFA}_{/\equiv} \rightarrow \{\text{true}, \text{false}\}$ checks whether the automaton contains cycles. The idea is to compute the minimum and the maximum path reaching each final state in the automaton (in Figure 6.2a, we obtain 3 and 5). Then, we abstract the set of lengths obtained so far into intervals (in the example, $[3, 5]$). Problems arise when the automaton contains cycles. In this case, we simply return the undefined interval starting from the minimum path, to a final state, to $+\infty$. For example, in the automaton

Algorithm 10: $\text{LE}^\# : \text{DFA}_{/\equiv} \rightarrow \text{Ints}$ algorithm

Data: $A = \langle Q, \Sigma, \delta, q_0, F \rangle$
Result: $\text{LE}^\#(A)$

```

1 P_len  $\leftarrow$  0; p_len  $\leftarrow$   $\infty$ 
2 if hasCycle(A) then
3   foreach  $q_f \in F$  do
4      $p \leftarrow$  minPath(A,  $q_0, q_f$ );
5     if len(p) < p_len then
6        $p\_len \leftarrow$  len(p)
7     end
8   end
9   return [p_len,  $+\infty$ ];
10 else
11   foreach  $q_f \in F$  do
12      $p \leftarrow$  minPath(A,  $q_0, q_f$ );
13      $P \leftarrow$  maxPath(A,  $q_0, q_f$ );
14     if len(p) < p_len then
15        $p\_len \leftarrow$  len(p)
16     end
17     if len(P) > P_len then
18        $P\_len \leftarrow$  len(P)
19     end
20   end
21   return [p_len, P_len];
22 end

```

in Figure 6.2b, the length interval is $[3, +\infty]$. Let $\text{LE}(\mathcal{L}) \triangleq \{ |\sigma| \mid \sigma \in \mathcal{L} \}$ be the collecting semantics of length.

Theorem 6.5. $\text{LE}^\#$ is sound but not complete. Formally,

$$\forall A \in \text{DFA}_{/\equiv}. \text{LE}(\mathcal{L}(A)) \subset \text{LE}^\#(A)$$

Abstract semantics of index0f. The abstract semantics of index0f is captured by the function $\text{IO}^\# : \text{DFA}_{/\equiv} \times \text{DFA}_{/\equiv} \rightarrow \text{Ints}$ and should return the interval of any possible positions of strings in a language inside strings of another language. Consider for instance the automaton A in Figure 6.3a and suppose to call $\text{IO}^\#(A, A')$ where $A' = \text{Min}(\{bc\})$. The idea is that of building, for each state q in A, the automaton A_q which is A where all the states are final and the initial state is q . Hence, we check whether $A_q \sqcap_{\text{DFA}} A'$ is non empty and we collect the size of the maximum path from q_0 to q in A. If there exists at least one state from which any string accepted by A' cannot be read, we collect -1. In the example, A_{q_0} adds $\{0\}$, A_{q_1} adds $\{1\}$, while all the other states add $\{-1\}$. Finally, we return the interval $[\min\{-1, 1, 0\}, \max\{-1, 1, 0\}] = [-1, 1]$. The full algorithm is reported in Algorithm 11.

Theorem 6.6. $\text{IO}^\#$ is sound but not complete. Formally,

$$\forall A, A' \in \text{DFA}_{/\equiv}. \text{IO}(\mathcal{L}(A), \mathcal{L}(A')) \subset \text{IO}(A, A')$$



FIGURE 6.2: (a) A_1 , $\mathcal{L}(A_1) = \{abc, hello\}$. (b) A_2 , $\mathcal{L}(A_2) = \{abc, hello\} \cup \{(abb)^n \mid n > 0\}$.

As a counterexample to completeness, consider the automaton A' in Figure 6.3b and the automaton $A'' = \text{Min}(\{b\})$: $\text{IO}^\#(A', A'') = [-1, 3] \not\subseteq \text{IO}(\mathcal{L}(A'), \mathcal{L}(A'')) = \{0, 3\}$. The interval $[-1, 3]$ contains also indexes where the string b is not recognized (e.g., 2), but it also contains the information (-1) meaning that there exists at least one accepted string without b as substring, which is not true.

Abstract semantics of concat. The abstract semantics of string concatenation is $\text{CC}^\# : \text{DFA}_{/\equiv} \times \text{DFA}_{/\equiv} \rightarrow \text{DFA}_{/\equiv}$ and returns the concatenation between two automata. Since regular languages are closed under concatenation, the property also holds on automata. In Figure 6.4, we report an example of concatenation between two automata. Hence, $\text{CC}^\#$ exactly implements the standard concatenation operation between automata. Let $\text{CC}(\mathcal{L}, \mathcal{L}') \triangleq \{\sigma\sigma' \mid \sigma \in \mathcal{L}, \sigma' \in \mathcal{L}'\}$ be the collecting semantics of concat. Given the closure property on automata w.r.t. concatenation, the following result holds.

Theorem 6.7. $\text{CC}^\#$ is sound and complete. Formally,

$$\forall A, A' \in \text{DFA}_{/\equiv}. \text{CC}(\mathcal{L}(A), \mathcal{L}(A')) = \text{CC}^\#(A, A')$$

Proof. Soundness and completeness follow from the fact that finite state automata and regular languages are closed under finite concatenation (see Theorem 2.50). \square

Abstract implicit type conversion. Here we discuss the abstraction of the implicit type conversion functions reported in Figure 3.2. We will focus only on the conversion of automata into intervals and booleans and viceversa, being the other conversions straightforward. In the definition of the semantics of implicit type conversion functions we follow the definition of implicit type conversion for basic types of JavaScript.



FIGURE 6.3: (a) A , $\mathcal{L}(A) = \{ddd, abc, bc\}$. (b) A' , $\mathcal{L}(A') = \{bcd, aab\}$

Algorithm 11: $\text{IO}^\# : \text{DFA}_{/\equiv} \times \text{DFA}_{/\equiv} \rightarrow \text{Ints}$ algorithm

Data: $A = \langle Q, \Sigma, \delta, q_0, F \rangle, A' = \langle Q', \Sigma, \delta', q'_0, F' \rangle$
Result: $\text{IO}^\#(A, A')$

```

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

```

String to boolean conversion. The function $\text{toBool}^\# : \text{Dyn} \rightarrow \text{Bool}$ is the function that handles the conversion from boolean values to other type values. Let this function be applied to $A \in \text{DFA}_{/\equiv}$. The abstract function $\text{toBool}^\#$ is defined as follows.

$$\text{toBool}^\#(A) = \begin{cases} \text{true} & \text{if } A \sqcap_{\text{DFA}} \text{Min}(\{\epsilon\}) = \text{Min}(\emptyset) \\ \text{false} & \text{if } A = \text{Min}(\{\epsilon\}) \\ \top_{\text{Bool}} & \text{otherwise} \end{cases}$$

String to integer conversion. Unlike boolean-to-string conversion, converting intervals to FA is more tricky. The function $\text{toInt}^\# : \text{Dyn} \rightarrow \text{Ints} \cup \{\text{NaN}\}$ handles conversion to intervals. Given an automaton A , we split the behavior of $\text{toInt}^\#$ in the following cases:

- $A \sqcap_{\text{DFA}} \text{Min}(\Sigma_{\mathbb{Z}}) = \emptyset$: in this case, A does not recognize any numerical string, hence the automaton is precisely converted to NaN, namely $\text{toInt}^\#(A) = \text{NaN}$;
- $A \sqsubseteq_{\text{DFA}} \text{Min}(\Sigma_{\mathbb{Z}})$: it means that A recognizes only numerical strings. For the sake of precision, we check whether A recognizes positive numeric strings (checking if the initial state reads only $+$ or number symbols), negative numeric strings

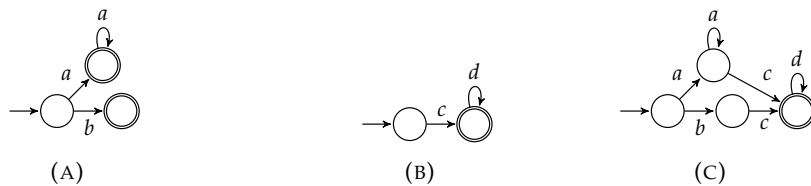


FIGURE 6.4: (a) $A, \mathcal{L}(A) = \{ a^n \mid n > 0 \} \cup \{ b \}$ (b) $A', \mathcal{L}(A') = \{ cd^n \mid n \in \mathbb{N} \}$ (c) $A'' = \text{CC}^\#(A, A')$

Input abstract value	Applied conversion	Result
$[0, +\infty]$	$\text{toString}^\#$	
$[-\infty, 0]$	$\text{toString}^\#$	
$\text{Min}(\{"12abc", "15"\})$	$\text{toInt}^\#$	$[0, +\infty]$
$\text{Min}(\{"12abc", "hello"\})$	$\text{toInt}^\#$	$[0, +\infty] \sqcup_{\text{Dyn}} \text{NaN}$
$\text{Min}(\{\epsilon, "hello"\})$	$\text{toBool}^\#$	$\{\text{false}, \text{true}\}$
$\text{Min}(\{"asd", "false"\})$	$\text{toBool}^\#$	true

TABLE 6.4: Examples of abstract implicit type conversions.

(checking if the initial state reads only $-$ or 0 symbols) or both. In the first case, the function $\text{toInt}^\#$ returns $[0, +\infty]$, in the second $[-\infty, 0]$ and in the last $[-\infty, +\infty]$.

- in the remaining cases, A can recognize both numerical and non-numerical strings, hence $\text{toInt}^\#$ should return an interval containing the values expressed by numerical strings recognized by A together with NaN . Hence, $\text{toInt}^\#$ returns \top_{Dyn} , being the most precise abstract value containing both the interval and NaN in Dyn .

Boolean and integer to string conversion. Implicit type conversion to DFA/\equiv is handled by the function $\text{toString}^\# : \text{Dyn} \rightarrow \text{DFA}/\equiv$. If the input is the boolean value true (false) it returns $\text{Min}(\{\text{true}\})$ ($\text{Min}(\{\text{false}\})$), otherwise it returns the automaton $\text{Min}(\{\text{true}, \text{false}\})$. As far as interval-to-automata conversion is concerned, several cases arises depending on the value of the input interval $[i, j]$. If $i, j \in \mathbb{Z}$, it means that the interval is finite and the conversion to automata of the interval $[i, j]$ is $\sqcup_{n \in [i, j]} \text{Min}(\{\mathcal{S}(n)\})$, recalling that \mathcal{S} converts a numerical string to the corresponding integer value. In this case, since $[i, j]$ is bounded, the function $\text{toString}^\#$ converts any integer in the interval to its corresponding automaton and afterward joins them all. The interval-to-automaton conversion for $[0, +\infty]$ and $[-\infty, 0]$ are the automata A^+ and A^- , respectively shown in Table 6.4 in the second and third rows. Other unbounded intervals, $[k, +\infty]$ and $[-k, +\infty]$ (with $k > 0$), are converted in $\text{toString}^\#(A^+) \setminus_{\text{DFA}/\equiv} \text{toString}^\#([0, k-1])$ and $\text{toString}^\#([-k, 1]) \sqcup \text{toString}^\#(A^+)$, respectively. Conversions of intervals $[-\infty, k]$ and $[-\infty, -k]$ (with $k > 0$) are similar and in particular are converted to $\text{toString}^\#(A^-) \sqcup_{\text{DFA}} \text{toString}^\#([1, k])$ and $\text{toString}^\#(A^-) \setminus_{\text{DFA}/\equiv} \text{toString}^\#([-k-1, 0])$. The last case is $\text{toString}^\#([- \infty, +\infty]) = \text{Min}(\Sigma_{\mathbb{Z}})$.

Theorem 6.8. *The abstract implicit type conversion methods $\text{toBool}^\#$, $\text{toInt}^\#$ and $\text{toString}^\#$ are sound.*

```

vd, ac, la = "";
v = "wZsZ"; m = "AYcYtYiYvYeYXY";
tt = "A0byaSZjectB";
l = "WYSYcYrYiYpYtY.YSYhYeYlYlY";

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

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

  ac += tt.substring(tt.indexOf("0"), 3);
  ac += tt.substring(tt.indexOf("j"), 11);

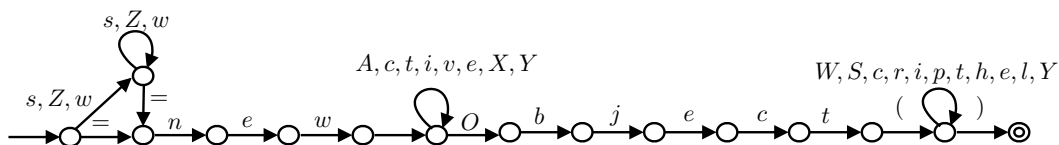
while (k+=2 < l.length)
  la = la + l.charAt(k);

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

```

FIGURE 6.5: A potentially malicious obfuscated JavaScript program.

Example: Obfuscated malware The abstract interpreter for the abstract semantics so far defined has been tested by means of the implementation of an automata library, available at <https://github.com/SPY-Lab/java-fsm-library>. The library is suitable and easily pluggable into existing static analyzers, such as [Jensen, Møller, and Thiemann, 2009; Lee et al., 2012; Kashyap et al., 2014]. It is worth noting that, as reported in Theorem 5.3, $\wp(\Sigma^*)$ (string concrete domain) and $\text{DFA}_{/\equiv}$ (string abstract domain) do not form a Galois connection but, nevertheless, this is not a concern. We have shown, for the core language we adopted, that the abstract semantics we have defined for string operations guarantee soundness hence, if the abstract interpreter starts from regular initial conditions (i.e., constraints expressible as finite state automata) it will always compute regular invariants. Indeed, it is sound to start from \top initial condition that, in our string abstract domain, is expressible by $\text{Min}(\wp(\Sigma^*))$, which is regular. Consider the fragment reported in Figure 6.5. By abstractly executing this code, we obtain that the abstract value of d , at the `eval` call, is the automaton A_d in Figure 6.6. The cycles are caused by the widening application in the while computation. From this automaton we are able to retrieve some important and non-trivial information. For example, we are able to answer to the following question: *May A_d contain a string corresponding to an assignment to an `ActiveXObject`?* We can simply answer by checking the predicate $A_d \sqcap_{\text{DFA}} \text{Min}(\text{ID} \cdot \{\text{new ActiveXObject}()\} \cdot \text{STR} \cdot \{\}) \neq \emptyset$, checking whether A_d recognizes strings that are concatenations of any

FIGURE 6.6: A_d abstract value of d before `eval` call of the program in Figure 6.5

identifier with the string *new ActiveXObject*, followed by any possible string. In the example, the predicate returns true. Another interesting information could be: *May A_d contain eval string?* We can answer by checking whether $A_d \sqcap_{\text{DFA}} \text{Min}(\{\text{eval}\}) \neq \emptyset$, that is false and guarantees that no explicit call to *eval* can occur.

We observe that such analysis may lose precision during fix-point computations, causing the cycles in the automaton in Figure 6.6, due to the widening application. Nevertheless, it is worth noting that this result is obtained without any precision improvement on fix-point computations, such as loop unrolling or widening with thresholds. We think these analyses will drastically decrease false positives of the proposed string analysis but we will address this topic in future work.

6.3 Towards an analysis for dynamic code

At this point, we are able to compute a static analysis of strings, where strings are abstracted to finite state automata. In the concrete semantics, *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

$$\langle \text{eval}(s) \rangle^{\#} m^{\#} = \bigsqcup_{c \in C} \langle c \rangle^{\#} m^{\#} \quad \text{where } C \triangleq \mathcal{L}(\langle s \rangle^{\#} m^{\#}) \cap \mu\text{JS}$$

Example 6.9. Consider the following μJS program. For the sake of readability, we omit the *else* empty branches.

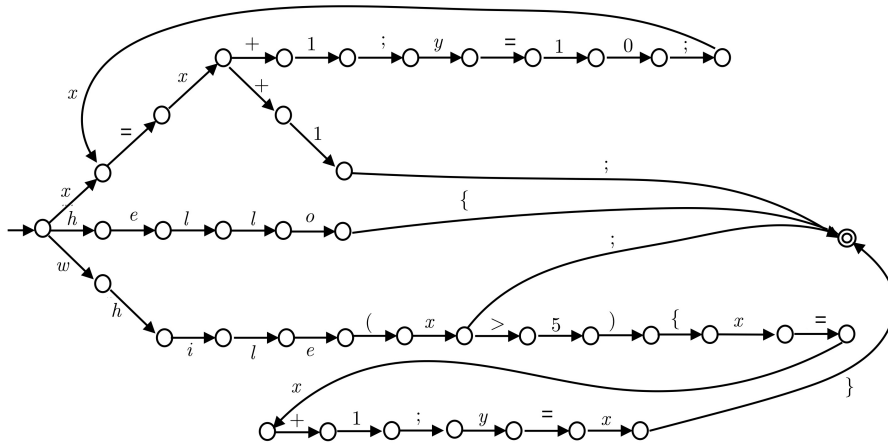
```

1while (x < 3) {
  2os = concat(os, "xA := Bx+1B; y := 1A0; x := Bx+1A; ");
  3x = x + 14
};5
if (x > 10) {
  6os = "whiA1eB(x>5A)A{x:A=x+1;y:=x};B"7
};8
if (x == 5) {
  9os = "hello{"10
};11
if (x == 8) {
  12os = "while(y;"13
};14
ds = deobf(os);15
eval(ds);16

```

The statement $ds = \text{deobf}(os)$ is syntactic sugar for the string transformer that removes the chars "A" and "B" from the string. In Figure 6.7 we depict the automaton A_{ds} , namely the abstract value of *ds* at the program point 15, computed analyzing the corresponding control-flow graph using the Dyn abstract domain, w.r.t. the widening ∇_{Dyn}^n , with $n = 3$.

Since A_{ds} is used as input of *eval*, we need to extrapolate a control-flow graph from the automaton. In particular, we first need 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 $\mathcal{L}(\langle ds \rangle^{\#} m^{\#})$ for a given memory $m^{\#}$) and the (context-free) μJS language (also denoted by μJS): $\mathcal{L}(\langle ds \rangle^{\#} m^{\#}) \cap \mu\text{JS}$. Note that, the intersection between a context-free language and a regular language (which is our case) is always a context-free language. This

FIGURE 6.7: FA A_{ds} abstract value of ds at line 15 of Example 6.9.

means that we could remove the non-executable strings accepted by the input automaton by performing an intersection such the one above, but unfortunately the computation of this intersection could be costly in practice due to the size of a real language grammar.

6.4 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 μJS 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 $\mathcal{L}(\langle \langle s \rangle \rangle^\# m^\#) \cap \mu JS$, by building a control-flow graph that soundly approximates the executable μJS programs in $\mathcal{L}(\langle \langle s \rangle \rangle^\# m^\#)$, 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 control-flow graph.

This original approach works by steps: Let $\langle \langle \text{eval}(s) \rangle \rangle^\# m^\#$ be the semantics the analyzer has to compute

1. First, we have to clean up the language $\mathcal{L}(\langle \langle s \rangle \rangle^\# m^\#)$ from all the strings that are surely not executable. This is obtained by visiting the FA $A_s = \langle \langle s \rangle \rangle^\# m^\#$ 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 $A_s^{\text{pStm}} \triangleq \text{StmSyn}(A_s)$;
2. From the regular expression corresponding to A_s^{pStm} , namely $\text{Regex}(A_s^{\text{pStm}})$, we build a control-flow graph, over-approximating the executable strings in the FA, i.e., $G_s \triangleq \text{CFGGen}(\text{Regex}(A_s^{\text{pStm}}))$. Then on this control-flow graph the analyzer can be recursively called.

The whole architecture is given in Figure 6.8, where the procedure $\text{Exe}^\#$ encapsulates (1) and (2) and their details are in the next sections. In particular, in the next sections,

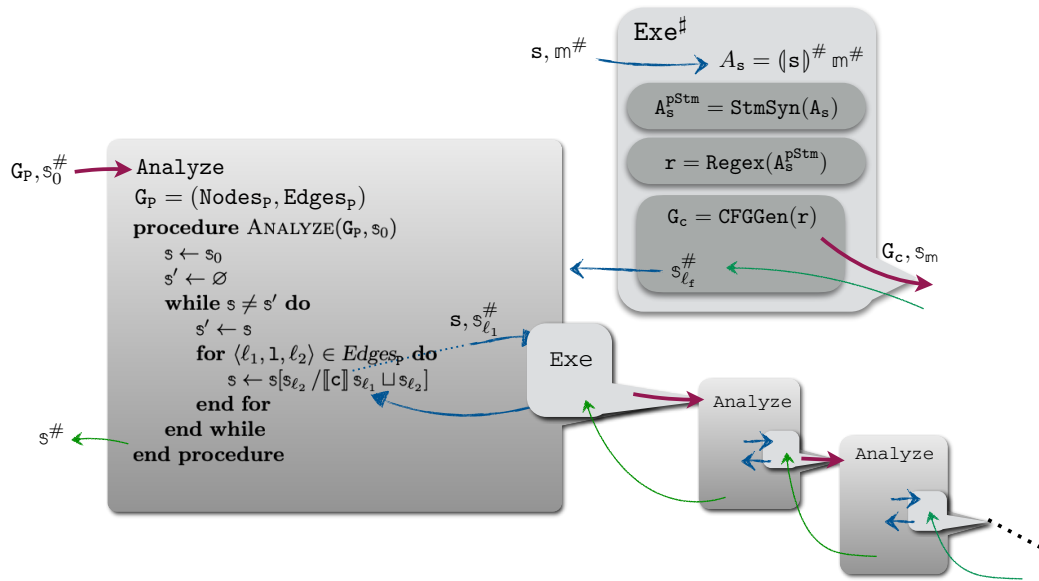


FIGURE 6.8: Analyzer architecture and call execution structure.

we suppose that the FA A_s is such that any cycle of A_s accepts only executable strings of μJS . We say that a FA A satisfying this property is *cycle-executable*. For example, the FA that accepts the language $\{ (x=x+1;)^n \mid n > 1 \}$ is cycle-executable, while the one accepting $\{ x=(1)^n; \mid n > 1 \}$ is not. In Section 6.6.1 we will discuss this constraint on the input automaton of `eval`.

6.5 Approximating `eval` executable code

In this section, we go into the details of how the synthesis of the control-flow graph executed by an `eval` works, i.e., how `Exe#` works. The abstract interpreter reported in Figure 6.8, when an `eval` is met, calls `Exe#` on the FA approximating the `eval` input. Let us consider Example 6.9. At line 15, we need to execute `Exe#` on A . In particular, `Exe#` goes through two steps: (1) extract from a FA the sub-language of executable strings (procedure `StmSyn`); (2) generate from the FA of the sub-language of executable strings a control-flow graph (procedure `CFGGen`). In the following, we describe these two sub-modules of `Exe#`.

6.5.1 `StmSyn`: Extracting the executable language

The first step consists in reducing the number of states of the FA, by (over) approximating every string recognized as a statement, or partial statement, in μJS . The idea is to derive, starting from the original FA A_s (generated by the string analysis), whose alphabet is the set of characters Σ , 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 $\text{Punct} \triangleq \{ ;, \{, \}, (,) \}$. 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 μJS grammar, which will be the alphabet of the resulting FA.

Algorithm 12: StmSyn function, building the FA.

Input: $A = \langle Q, \delta, q_0, F, \Sigma \rangle$
Output: $A' = \langle Q', \delta', q_0, F', \Sigma_{\text{pStm}} \rangle$

```

1 function StmSyn(A):
2    $Q' \leftarrow \{q_0\};$ 
3    $F' \leftarrow F \cap \{q_0\};$ 
4    $\delta' \leftarrow \emptyset;$ 
5   Visited  $\leftarrow \{q_0\};$ 
6   return StmSynTr( $q_0$ );
1 function StmSynTr( $q$ ):
2    $B \leftarrow \text{Build}(A, q);$ 
3   Visited  $\leftarrow \text{Visited} \cup \{q\};$ 
4    $Q' \leftarrow Q' \cup \{p \mid (\mathbf{a}, p) \in B\};$ 
5    $F' \leftarrow Q' \cap F;$ 
6    $\delta' \leftarrow \delta' \cup \{(q, \mathbf{a}, p) \mid (\mathbf{a}, p) \in B\};$ 
7    $W \leftarrow \{p \mid (\mathbf{a}, p) \in B\} \setminus \text{Visited};$ 
8   while  $W \neq \emptyset$  do
9     select  $p$  in  $W;$ 
10     $W \leftarrow W \setminus \{p\};$ 
11    StmSynTr( $p$ );
12  end

```

The *partial statements* $\Sigma_{\text{pStm}} \subseteq \Sigma^*$ are defined as follows

$$\Sigma_{\text{pStm}} \triangleq \text{Punct} \cup \left\{ x \in \Sigma^* \left| \begin{array}{l} x \text{ is a maximal substring of a } \mu\text{JS statement} \\ \text{between two punctuation symbols} \\ \text{(first punctuation symbol excluded)} \end{array} \right. \right\}$$

Lemma 6.10. Let \mathfrak{S} be the function mapping any sequence $(\Sigma^*)^*$ on its string counterpart on Σ^* (and, abusing notation, also its additive lift to sets of sequences), then $\mu\text{JS} \subseteq \mathfrak{S}((\Sigma_{\text{pStm}})^*)$, namely any statement $c \in \mu\text{JS}$ can be written as a sequence of partial statements in Σ_{pStm} . Formally,

$$\forall p \in \mu\text{JS}. \exists k \in \mathbb{N}. \{\sigma_i\}_{i \in [0, k]} \in \Sigma_{\text{pStm}}. c = \mathfrak{S}(\sigma_0 \sigma_1 \dots \sigma_k).$$

At this point, the idea is that of transforming the FA A_s on the alphabet Σ in the FA A_s^{pStm} on the alphabet Σ_{pStm} , removing any string recognized by A_s which will be surely not executable. The soundness constraint 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 (Algorithm 12) taking as input a FA on Σ (i.e., A_s for eval(s)) and returning the FA on a finite subset of Σ_{pStm} . In particular, the idea of Algorithm 12 is to perform, starting from the state q_0 , a visit of the states recursively identified by Algorithm 13, that is the states reached by q_0 reading partial statements, and to recursively replace the sequences of edges that recognize a symbol in Σ_{pStm} with a single edge labeled by the corresponding string. Algorithm 13 scans the edges of the original FA A_s and, when a punctuation symbol occurs or a final state is reached, it verifies whether the string read so far is in Σ_{pStm} , otherwise it is discarded: This *executability* check is performed

Algorithm 13: Statements recognized from a state q .

Input: $A = \langle Q, \delta, q_0, F, \Sigma \rangle$
Output: I_q set of all pairs (partial statement, reached state)

```

1 function Build(A):
2    $I_q \leftarrow \emptyset$ ;
3   BuildTr( $q, \varepsilon, \emptyset$ );
4   return  $I_q$ ;
1 function BuildTr( $q, word, Mark$ ):
2    $\Delta_q \leftarrow \{ (\sigma, p) \mid \delta(q, \sigma) = p \}$ ;
3   while  $\Delta_q \neq \emptyset$  do
4     select  $(\sigma, p)$  in  $\Delta_q$ ;
5      $\Delta_q \leftarrow \Delta_q \setminus \{ (\sigma, p) \}$ ;
6     if  $(q, p) \notin Mark$  then
7       if  $\sigma \notin Punct \wedge p \notin F$  then
8         BuildTr( $p, word.\sigma, Mark \cup \{(q, p)\}$ );
9       if  $\sigma \in Punct \wedge word.\sigma \in \Sigma_{pStm}$  then
10         $I_q \leftarrow I_q \cup \{ (word.\sigma, p) \}$ ;
11      if  $p \in F \wedge word.\sigma \in \Sigma_{pStm}$  then
12         $I_q \leftarrow I_q \cup \{ (word.\sigma, p) \}$ ;
13    end

```

at lines 9 and 11 and ensures, for any state q of the FA A_s , that I_q contains only (partial) statements of μJS . In particular, from the state 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 Figure 6.9 we have the FA $A_{ds}^{pStm} = StmSyn(A_{ds})$. Note that the string `hello{`, that it is recognized by A_{ds} , is not recognized by A_{ds}^{pStm} since it is discarded by Algorithm 12, because it does not belong to Σ_{pStm} . Instead, the string `while(y; 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).`

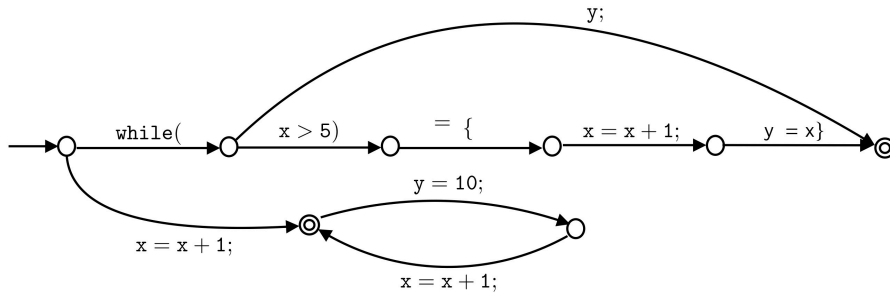
Lemma 6.11. Let A be a cycle-executable finite state automaton. Then, $\forall \sigma \in \Sigma_{pStm}^* \cdot \mathfrak{S}(\sigma) \in \mathcal{L}(A) \Rightarrow \sigma \in \mathcal{L}(A^{pStm})$.

Next theorem tells us that any executable string collected during computation is kept in the transformed FA, guaranteeing soundness.

Theorem 6.12. Let $s \in SE$, let A_s be the FA recognizing the strings associated with s , and $A_s^{pStm} \triangleq StmSyn(A_s)$, then $\forall \sigma \in \mathcal{L}(A_s) \cap \mu JS. \exists \delta \in \mathcal{L}(A_s^{pStm})$ s.t. $code(\mathfrak{S}(\delta)) = \sigma$.

Proof. Given $\sigma \in \mathcal{L}(A_s) \cap \mu JS$, from Lemma 6.10, $\exists \delta \in \Sigma_{pStm}^*$ such that $code(\mathfrak{S}(\delta)) = \sigma$ and from Lemma 6.11, $\delta \in \mathcal{L}(A^{pStm})$. \square

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 $O(|Q|^4)$.

FIGURE 6.9: FA $A_{ds}^{pStm} = \text{StmSyn}(A_{ds})$.

6.5.2 CFGGen: Control-flow graph generation

At this point, the idea is to use the so far obtained FA over Σ_{pStm} to generate a control-flow graph approximating the program executed in $\text{eval}(s)$. In particular, in this section we design the procedure `CFGGen`, that works by several steps. It is well known that a FA A can be equivalently rewritten as a regular expression r , s.t. $\mathcal{L}(A) = \mathcal{L}(r)$ [Brzowski, 1964]. Let RE be the domain of the regular expressions over Σ_{pStm} , and $\text{Regex} : \text{FA} \rightarrow RE$ be such an extractor. In the running example, $r_{ds} = \text{Regex}(A_{ds}^{pStm})$ is the following regular expression:

$$r_{ds} = x=x+1; \mid \mid \text{while}(y; \mid \mid \text{while}(x>5)\{x=x+1;y=x\}; \mid \mid x=x+1;(y=10;x=x+1;)^*$$

The analyzer implements the Brzowski algebraic method [Brzowski, 1964] to convert a FA to an equivalent regular expression. It is worth noting that several regular expression simplifications are applied (e.g., the ones reported in Section 2.6.1). One important rearrangement is when we meet a concatenation d_1d_2 : when d_1d_2 are ground terms (i.e., elements of Σ_{pStm}), concatenation is removed and a single ground term is created. Hence, any executable regular expression d_1d_2 cannot be such that d_1 and d_2 are not executable.²

The idea is to exploit the inductive structure of regular expressions to generate a μJS program that can be converted to a control-flow graph over-approximating the executable program contained in s . In particular, we introduce a special symbol to the boolean expressions of μJS :

$$b \in \text{BE} ::= \dots \mid \otimes$$

The special symbol \otimes is used in the following to generate, from a regular expression over Σ_{pStm} , a μJS program augmented with \otimes . We abuse notation denoting by μJS the language taking into account also \otimes . This symbol is only used to label the boolean guards of an `if` or a `while` statements, during the translation from regular expressions to our code intermediate representation. The special symbol \otimes must be intended only as a placeholder and it will be never executed neither in the concrete nor the abstract semantics. In particular, we augment the `Edges` function, namely the one that generates the edges of a control-flow graph, for `if` and `while`

²Moreover, since concatenation is distributive w.r.t. $\mid \mid$, the conversion algorithm always distributes, in this case. For instance, $x=(1; \mid \mid 2;)$ is converted to $(x=1;)\mid \mid (x=2;)$.

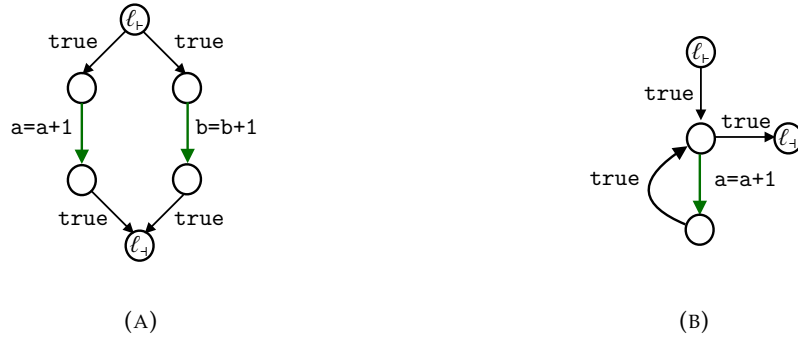


FIGURE 6.10: (a) $\text{CFG}_{\mu\text{JS}}(\lambda a:=a+1; \parallel b:=b+1; \})$, (b) $\text{CFG}_{\mu\text{JS}}(\lambda (a:=a+1; \}))$

statements when their boolean guard is \otimes .

$$\begin{aligned} \text{Edges}^{\ell_1}(\text{if}(\otimes)\{\ell_2\text{st}_1^{\ell_3}\}\text{else}\{\ell_4\text{st}_2^{\ell_5}\}\ell_6) &= \{\langle \ell_1, \text{true}, \ell_2 \rangle, \langle \ell_1, \text{true}, \ell_4 \rangle, \} \\ &\cup \{\langle \ell_3, \text{true}, \ell_6 \rangle, \langle \ell_5, \text{true}, \ell_6 \rangle\} \\ &\cup \text{Edges}^{\ell_2\text{st}_1^{\ell_3}} \cup \text{Edges}^{\ell_4\text{st}_2^{\ell_5}} \\ \text{Edges}^{\ell_1}(\text{while}(\otimes)\{\ell_2\text{st}^{\ell_3}\}\ell_4) &= \{\langle \ell_1, \text{true}, \ell_2 \rangle, \langle \ell_1, \text{true}, \ell_4 \rangle\} \\ &\cup \{\langle \ell_3, \text{true}, \ell_1 \rangle\} \cup \text{Edges}^{\ell_2\text{st}^{\ell_3}} \end{aligned}$$

The special guard \otimes is used as a non-deterministic boolean guard. For example, let us consider following program:

```
if( $\otimes$ ){ a=a+1 }else{ b=b+1 };
```

Its control-flow graph, generated with the augmented version of *Edges*, is reported in Figure 6.10a, where the true and false branches are both labeled with true. In this way, the static analysis algorithm (namely Algorithm 9) must take into account both branches, similarly to an if abstract execution where the boolean guard is statically unknown, namely it evaluates to $\{\text{true}, \text{false}\}$. Similarly, let us consider the while case:

```
while( $\otimes$ ){ a=a+1 };
```

The corresponding control-flow graph is reported in Figure 6.10b, where, both the true branch and the false branch (that exits from the body) are labeled with true. In this way, we emulate a while loop where the boolean guard is statically unknown, namely it evaluates to $\{\text{true}, \text{false}\}$. We abuse notation denoting by $\text{CFG}_{\mu\text{JS}}$ the control-flow graph generator that takes into account the novel special guard \otimes and the corresponding *Edges* definition.

At this point, we have all the ingredients to generate a μJS program from a regular expression. In particular, we define the function $\lambda \cdot \} : RE \rightarrow \mu\text{JS}$ that, given $r \in RE$, translates r to a μJS program augmented with the special boolean symbol \otimes . The function $\lambda \cdot \}$ is inductively defined on the structure of regular expressions, where $d \in \Sigma_{\text{pstm}}$.

$$\begin{aligned}
\lambda d \rfloor &= \begin{cases} \text{code}(\mathfrak{S}(d)) & \text{if } \text{code}(\mathfrak{S}(d)) \in \Sigma_{\text{pstm}} \\ \text{skip} & \text{otherwise} \end{cases} \\
\lambda r_1 r_2 \rfloor &= \lambda r_1 \rfloor \lambda r_2 \rfloor \\
\lambda r_1 \parallel r_2 \rfloor &= \text{if}(\otimes)\{\lambda r_1 \rfloor \in \mu\text{JS} ? \lambda r_1 \rfloor : \text{skip}\} \text{else}\{\lambda r_2 \rfloor \in \mu\text{JS} ? \lambda r_2 \rfloor : \text{skip}\} \\
\lambda (r)^* \rfloor &= \text{while}(\otimes)\{\lambda r \rfloor \in \mu\text{JS} ? \lambda r \rfloor : \text{skip}\}
\end{aligned}$$

In the base case (first line), we check if d is a partial statement, namely if $d \in \Sigma_{\text{pstm}}$. If so, it is returned as code (abusing notation of code), otherwise skip statement is returned. In the case of $\lambda r_1 r_2 \rfloor$, the function concatenates the two programs inductively generated.

In the case of $\lambda r_1 \parallel r_2 \rfloor$, we need to emulate the non-deterministic execution of both operands. Here comes to play the special symbol \otimes , previously introduced. In particular, we return an if statement where the if-true body is replaced with $\lambda r_1 \rfloor$ if it is executable, skip otherwise, and the if-false body is replaced with $\lambda r_2 \rfloor$, if it is executable, skip otherwise. The boolean guard of the if statement is \otimes . It is worth noting that we need to check the executability of $\lambda r_1 \rfloor$ and $\lambda r_2 \rfloor$, since the true and false bodies must be μJS executable. For example, let us consider the following regular expression:

$$\text{while}(\parallel a=a+1;$$

Clearly, $\text{while}(\parallel a=a+1;$ is not executable, hence, following the definition of $\lambda \cdot \rfloor$, it is replaced with skip (i.e., no-op) without affecting the abstract semantics of other potentially statements (e.g., $a=a+1;$). Indeed, the resulting program is

$$\text{if}(\otimes)\{\text{skip}\} \text{else}\{a=a+1\};$$

We treat in a similar way the case of $\lambda (r)^* \rfloor$: in order to guarantee soundness, the μJS program $\lambda r \rfloor$ must be executed an undefined number of times, hence, we build a while loop program, where the boolean guard is \otimes . Hence, the final function to transform a regular expression over partial statements into a μJS program, using the function $\lambda \cdot \rfloor$ is $\lambda r \rfloor^P \triangleq \lambda r \rfloor \in \mu\text{JS} ? \lambda r \rfloor : \text{skip}$, that uses the rules defined above and just explained. In the following, we abuse notation denoting $\lambda \cdot \rfloor^P$ as $\lambda \cdot \rfloor$.

In our running example, the code synthesis from the regular expression r_{ds} is the augmented μJS program reported in Figure 6.11.

Finally, we need to generate a control-flow graph on which we can recursively call our abstract interpreter. Hence, the last step corresponds to call $\text{CFG}_{\mu\text{JS}}$ on the synthesized code, namely $\text{CFGGen}(r) = \text{CFG}_{\mu\text{JS}}(\lambda r \rfloor)$. In our running example, the synthesis from the regular expression r_{ds} is the control-flow graph $G_{\text{ds}} = \text{CFGGen}(r_{\text{ds}})$ reported in Figure 6.12, where the labels true are omitted. From here on, for the sake of readability, consecutive edges labeled with trues are omitted. Note that the control-flow graph of $\text{while}(y; \cdot)$ corresponds to the control-flow graph of skip (right-most path in Figure 6.12).

Finally, we have to prove soundness, proving that the output control-flow graph contains the computation of all the executable strings that are in the starting FA. In particular, the next lemma shows that the control-flow graph generated by $\text{CFGGen}(r)$ contains all the computations of executable strings recognized by r .

Lemma 6.13. Given $r \in RE$, let $G_r \triangleq \text{CFGGen}(r)$, then $\forall \delta \in \mathcal{L}(r), \forall m^\# \in \mathbb{M}^\#. \exists \Pi \subseteq \text{Paths}(G_r)$ s.t. $(\text{code}(\mathfrak{S}(\delta)))^\# m^\# \subseteq \bigcup_{\pi \in \Pi} (\pi)^\# m^\#$


```

1  if (⊗) {
2    if (⊗) {
3      if (⊗) {
4        x = x + 1
5      } else {
6        skip
7      }
8    } else {
9      while (x > 5) {
10       x = x + 1;
11       y = x
12     }
13   }
14 } else {
15   x = x + 1;
16   while (⊗) {
17     y = 10;
18     x = x + 1
19   }
20 }

```

FIGURE 6.11: μ JS program of $\{r_{ds}\}$.

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

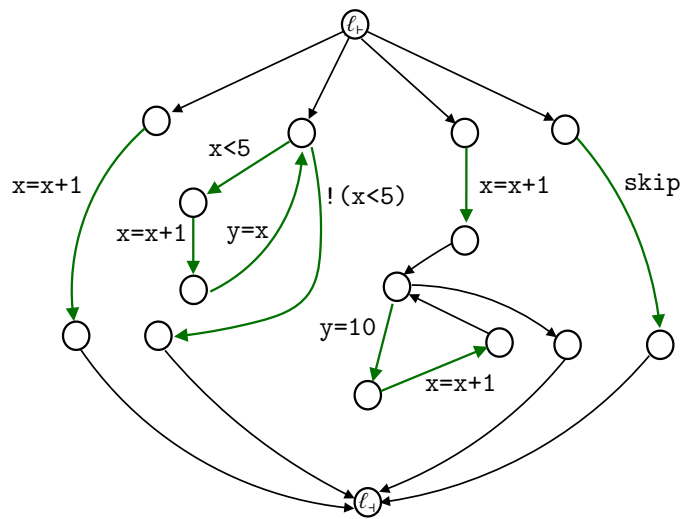
Theorem 6.14 (Soundness). *Let $s \in \text{SE}$, let A_s be the FA recognizing the strings associated with s , then $\forall \sigma \in \mathcal{L}(A_s) \cap \mu\text{JS}, \forall m^\# \in \mathbb{M}^\#$*

$$\exists \Pi \subseteq \text{Paths}(G_s). G_s \triangleq \text{CFGGen}(\text{Regex}(\text{StmSyn}(A_s))). \langle \sigma \rangle^\# m^\# \subseteq \bigcup_{\pi \in \Pi} \langle \pi \rangle^\# m^\#$$

Proof. By Theorem 6.12, we have that $\forall \sigma \in \mathcal{L}(A_s) \cap \mu\text{JS} \exists \delta \in \mathcal{L}(\text{StmSyn}(A_s))$ such that $\text{code}(\mathfrak{S}(\delta)) = \sigma$, hence any string collected in A_s corresponding to executable code, is kept in the transformed automaton $A_s^{\text{pStm}} = \text{StmSyn}(A_s)$. Then, $\mathcal{L}(\text{StmSyn}(A_s)) = \mathcal{L}(\text{Regex}(\text{StmSyn}(A_s)))$, hence $\delta \in \mathcal{L}(\text{Regex}(\text{StmSyn}(A_s)))$. Finally Lemma 6.13 we have that $\forall m^\# \in \mathbb{M}^\#. \langle \sigma \rangle^\# m^\# = \langle \text{code}(\mathfrak{S}(\delta)) \rangle^\# m^\# \subseteq \bigcup_{\pi \in \Pi} \langle \pi \rangle^\# m^\#$. \square

6.5.3 Abstracting sequences of eval nested calls

We have previously described the architecture of the analysis, which recursively calls the analysis on the synthesized control-flow graph 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. In particular, this kind of divergence comes directly from the meaning of dynamically generated code from strings and cannot be controlled by the semantics (namely with standard techniques such as in the case of loop computations or recursive function calls) once we execute the string-to-code statement. At the beginning of Section 6.1, 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

FIGURE 6.12: Control-flow graph G_{ds} generated by CFGGen module

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 in fixing a threshold of allowed height of the nested `eval` calls. Once we reach the threshold, the only way to keep soundness consists in 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.

6.6 Evaluating the analyzer

We have implemented the μ JS static analyzer (available at <https://github.com/SPY-Lab/mujs-analyzer>) described in this chapter, 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 [Jensen, Møller, and Thiemann, 2009]. 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:

Q1: Does the analyzer handle 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

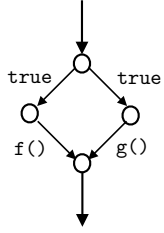
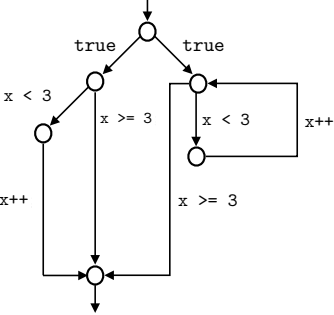
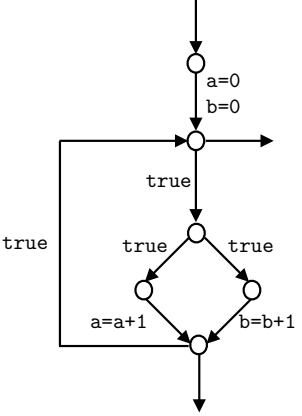
Code fragment	Exe# output
<pre> str = "x="; if (B) str = str + "f"; else str = str + "g"; eval(str + "()"); </pre>	
<pre> if (B) str = "if"; else str = "while"; str = str + "(x<3){x++;}" eval(str); </pre>	
<pre> str = "a=0;b=0;"; while (i++ < 100) { if (B) str = str + "a++;"; else str = str + "b++;"; } eval(str); </pre>	

TABLE 6.5

TAJS [Jensen, Møller, and Thiemann, 2009; Jensen, Jonsson, and Møller, 2012](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 Table 6.5, supposing that the boolean guard `B` is unknown; hence both 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 control-flow graph reported in the first row, second column, in Table 6.5 (Exe# output). Concerning precision, the synthesized control-flow graph is precise since it precisely contains the two possible executions.

Consider a more challenging example, provided in the second row of Table 6.5. 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 think (and we have found) that this is a 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 control-flow graph is precise since it contains the two possible programs to execute.

Finally, in the last row of Table 6.5, 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 value of `i` before the loop. Hence, we need to apply the widening operator ∇_{Dyn}^n , to ensure termination. In the example, we fix $n = 5$, using the DFA_{\equiv} widening operator $\nabla_{\text{DFA}_{\equiv}}^5$, allowing us to over-approximate the value of `str` by the regular expression `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 control-flow graph, over-approximating the code executed by `eval`, is shown in the last row. In this case, the control-flow graph generation process adds further imprecision due to both the widening (generating cycles in the FA) and the way a control-flow graph is generated starting from a Kleene-star regular expression.

Nested eval calls (Q2). As explained in Section 6.5.3, 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 than 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 Section 6.5.3. 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 Σ^* , making the analyzer able to trace string manipulations also of unknown (set to \top) variables. 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);
```



FIGURE 6.13: A_{str} s.t. $\mathcal{L}(A_{\text{str}}) = \{x = 5^n; | n > 0\}$

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.

6.6.1 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 control-flow graph that over-approximates the `eval` input string. Consider the fragment reported in Figure 6.13a, assuming that the value of `i` is unknown. Moreover, consider to apply $\nabla_{\text{DFA}/\equiv}^2$ in the `while`-loop. In Figure 6.13b, we report the FA abstracting the `eval` input, where the cycle in the FA is caused by the application of the widening $\nabla_{\text{DFA}/\equiv}^2$ to ensure termination. In this case, our analyzer cannot return a control-flow graph that over-approximates the code that may be concretely executed: the hypothetical control-flow graph 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 control-flow graph 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, namely when the automaton is not a cycle-executable automaton, 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. It is worth noting that these cases occur only when the FA contains cycles that do not repeat valid statements. Nevertheless, we are currently investigating how to handle even these cases, trying to propose a solution to integrate into our analyzer.

6.6.2 Comparison with TAJs

In [Jensen, Jonsson, and Møller, 2012], 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 [Richards et al., 2011] 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 expressions 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 string 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

P	Tajs result	Exe [#] (A _y) result
<pre>y="x=x+1;"; eval(y);</pre>	<code>x=x+1;</code>	
<pre>if (x > 0) y="a=a+1;"; else y="b=b+1;"; eval(y);</pre>	Analysis Limitation Exception	
<pre>y=""; while (x < 3) { y = y + "x=x+1;"; x=x+1; } eval(y);</pre>	Analysis Limitation Exception	

TABLE 6.6: Comparison with Tajs

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 approach. We report three significant examples in Table 6.6, 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 Table 6.6, 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 Table 6.6. In this case, since we do not have any information about `x`, we must consider both 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, 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_y recognizing the language expressed by the regular expression `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 Table 6.6, 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, 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 $\nabla_{\text{DFA}/=}^3$, we compute a FA A_y strictly containing the concrete set of possible string values, recognizing the regular expression `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 the parameter, the more precise and costly the analysis is. The control-flow graph extracted from A_y is reported in the third row.

Final remarks. The novel approach presented in this chapter 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 script-like programming language. The main contribution of this chapter is in proposing an innovative approach for designing sound static analyzer for dynamic code, i.e., for code that may change during execution. The main idea is to analyze strings by approximating them as regular languages, i.e., by using finite state automata. When a string-to-code instruction is met, the automata modeling the string-to-code instruction input is analyzed in order to approximate its *executable* sub-language, namely the sub-language of all the executable statements at that program point. This approximated sub-language is then used for building a control-flow graph whose semantics soundly approximates the semantics of what is concretely executed by the string-to-code statement. In this way we can recursively call the same abstract interpreter on the synthesized control-flow graph. 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.

We have implemented the analyzer for μ JS and it is available at <https://github.com/SPY-Lab/mujs-analyzer>. As far as the string analysis is concerned, in the thesis we have shown the abstract semantics of five popular string operations. Nevertheless, the static analyzer we have implemented provide several other string operations contained into the built-in JavaScript global object `String`, such as, for example `slice`, `startsWith` or `repeat`. We have decided to omit these other operations to do not burden the presentation, since their abstract semantics can be seen as corner cases of the abstract semantics reported in this chapter (e.g., `slice` for `substring`).

Moreover, we are currently integrating the finite state automata abstract domain and the dynamic code analysis reported in this chapter upon TAJs. The prototype is available at https://github.com/SPY-Lab/tajs_automata. The result, is still a

prototype and it cannot be considered a real tool for several reasons. For example, high-order functions and JavaScript prototype inheritance are, at the moment, partially supported, as well as some standard built-in global objects. Moreover, we have implemented only one string-to-code statement, i.e., `eval`. It is well known that there are other ways for dynamically executing code built out of strings, but it is clear that the same approach used for `eval` can be easily applied to any other string-to-code statement. We are currently improving the prototype to deal with full JavaScript.

Chapter 7

An abstract domain for objects in dynamic languages

In this last chapter, we present a preliminary approach that tries to face another problem in dynamic programming languages. As we have already mentioned more than once, dynamic languages such as JavaScript or PHP have gained a huge success in a very wide range of applications and this mainly happened due to the several features that such languages provide to developers. One of this features is the way strings may be used to interact with programs objects. Unlike strongly-typed object-oriented languages, where the structure of an object (i.e., its class) is known at compile-time and cannot change during program execution, usually in dynamic languages it is possible to create, manipulate, and delete object properties at run-time, interacting with them using strings. If, on the one hand, this may help developers to simplify coding and to build applications faster, on the other hand, this may lead to misunderstandings and bugs in the produced code. Furthermore, because of these dynamic features, reasoning about dynamic programs by means of static analysis is quite hard, producing very often imprecise results.

For instance, let us consider the simple yet expressive example reported in Figure 7.1, supposing that the value of the `if` guard is statically unknown. The value of `idx` is indeterminate after line 6 and it is updated at each iteration of the `while` loop (line 10). The `while` guard is also statically unknown and at each iteration we access `obj` with `idx`, incrementally saving the results in `n`. The goal is to statically retrieve the value of `idx` and `n` at the end of the program. It is worth noting that a crucial role here is played by the string abstraction used to approximate the value of `idx`, that is used to access `obj`. Indeed, adopting finite abstract domains, such as [Jensen, Møller, and Thiemann, 2009; Kashyap et al., 2014; Lee et al., 2012], will lead to infer that `idx` could be any possible string. Consequently, when `idx` is used to access `obj`, in order to guarantee soundness, we need to access all properties of `obj`. For instance, we also have to consider the property `ac`, which is never used to access `obj` during the execution of the program. This ends up in an imprecise approximation of `idx` and, in turn, of `n`.

In this chapter, we further augment the μ JS syntax and semantics in order to make it able to handle also object expressions. Since we do not model important object-related features such as encapsulation or inheritance, it would be more appropriate to refer to our model as associate arrays [Hauzar and Kofron, 2015a] nevertheless, in this chapter, for the sake of simplicity, we will refer to it as objects.

We exploit again the finite state automata abstract domain presented in Chapter 5. We still abstract strings values to the finite state automata abstract domain and we use it also to define a novel abstract domain for objects. The idea is to abstract the objects properties in the same domain used to abstract string values, namely the finite state automata abstract domain. We show that exploiting finite automata

```

1  if (?) {
2    idx = "a"
3  }
4  else {
5    idx = "b"
6  };
7  n = 0; obj = new {a:1, aa:2, ab:3, ac:"world"};
8  while (?) {
9    n = n + obj[idx];
10   idx = concat(idx, "a")
11  }
12  obj[idx] = n; // value of idx and n ?

```

FIGURE 7.1: Motivating example.

to abstract string values and objects properties produces precise results in abstract computations, in particular in object properties lookup and in object manipulations inside iterative constructs. We define the necessary semantic transformers for objects to be integrated in the static analysis algorithm on control-flow graph but, differently from what we have done until this chapter, for the sake of simplicity, all the examples of the abstract computations shown here are done directly on the μ JS program we aim to analyze and not on its corresponding control-flow graph.

7.1 Object concrete semantics

In this section, we extend the μ JS syntax with object expressions. An object is a comma-separated collection (potentially empty) of property-value associations.

$$o \in \text{OE} ::= \{ \} \mid \{ \sigma_0 : e_0, \sigma_1 : e_1, \dots, \sigma_n : e_n \}$$

For example, a μ JS object may be $\{a:1, b:2, c:3\}$. Afterwards, we also introduce object construction, object-property lookup and object-property update, as follows.

$$e \in \text{E} ::= \dots \mid x[s] \quad \text{st} \in \text{STMT} ::= \dots \mid x = \text{new } o \mid x[s] = e$$

For the sake of simplicity, it is worth noting that object properties cannot recursively contain other objects. Anyway, this choice is not restrictive and in [Arceri, Pasqua, and Mastroeni, 2019] we have taken into account also objects that can recursively contain other object properties.

The set of the possible values associated with a variable must take into account also objects. We recall that VAL is defined as $\text{VAL} \triangleq \text{INT} \cup \text{BOOL} \cup \text{STR} \cup \{\text{NaN}\} \cup \{\uparrow\}$. Hence, we define the set VAL^P as

$$\text{VAL}^P \triangleq \text{VAL} \cup \text{OBJ}$$

An object $o \in \text{OBJ}$ is represented as a map that associates strings to primitive values, namely $\text{OBJ} \triangleq \text{STR} \rightarrow \text{VAL}$. It is worth noting that there is no order relation between object properties, as it happens in standard programming languages. At this point, we are ready to extend the expression and statement big-step semantics defined in Chapter 3. The evaluation of an object takes each association string-expression and it recursively evaluates the expressions, returning as result a map

containing the string-value associations. Hence, we extend the expressions semantics $\llbracket e \rrbracket : \text{STATE} \rightarrow \text{VAL}^P$ as follows.

$$\llbracket \{s_0 : e_0, s_1 : e_1, \dots, s_n : e_n\} \rrbracket \zeta \triangleq [s_n \mapsto \llbracket e_n \rrbracket \zeta] \bullet \dots \bullet [s_1 \mapsto \llbracket e_1 \rrbracket \zeta] \bullet [s_0 \mapsto \llbracket e_0 \rrbracket \zeta]$$

where $f \bullet g(s) \triangleq g(s)$ if $g(s) \neq \uparrow \wedge f(s) = \uparrow$ and $f \bullet g(s) \triangleq f(s)$ otherwise. For example, the expression $\{a:1, b:\text{length}(\text{"foo"}), c:5+3\}$ evaluates to the object $[a \mapsto 1 \ b \mapsto 3 \ c \mapsto 8]$. Following the JavaScript semantics, it is worth noting that, for instance, $\{a:1, a:2\}$ evaluates to $[a \mapsto 2]$, saving only the last association with the same property a .

The semantics of objects properties lookup $x[s]$ checks whether the string resulting from the evaluation of s is a property of the object stored in x . If so, the associated value is returned. Hence, its definition is the following, supposing that $\llbracket s \rrbracket \zeta = \sigma \in \text{STR}$:

$$\llbracket x[s] \rrbracket \zeta \triangleq \begin{cases} \zeta(x)(\sigma) & \text{if } \zeta(x) \in \text{OBJ} \wedge \sigma \in \text{dom}(\zeta(x)) \\ \uparrow & \text{otherwise} \end{cases}$$

In our core language, we allow only to access already stored objects. Moreover, it is worth noting that when we try to access a property σ not present in the object contained in x , then \uparrow is returned.

Finally, we augment the statement semantics $\llbracket \text{st} \rrbracket : \text{STATE} \rightarrow \text{STATE}$, defining the semantics of object construction and object property update. The semantics of $x = \text{new } o$ stores into the states the evaluation of OBJ .

$$\llbracket x = \text{new } o \rrbracket \zeta \triangleq \zeta[x \leftarrow \llbracket o \rrbracket \zeta]$$

The semantics of $x[s] = e$ stores in the object stored by x , at the property expressed by the evaluation of s , the evaluation of e . In the following, we suppose that $o = \llbracket x \rrbracket \zeta$.

$$\llbracket x[s] = e \rrbracket \zeta \triangleq \zeta[x \leftarrow o \bullet \llbracket [s] \zeta \mapsto \llbracket e \rrbracket \zeta \rrbracket]$$

Note that, as it happens in dynamic languages, if the property is not contained in the object stored by x , a new association is created and added. As a final remark, we point out that in our extension of μJS we do not model features such as pointer arithmetic, objects comparisons and object-related implicit type conversion.

At this point, since we perform static analysis of the control-flow graph of a μJS program, we need to slightly change the collecting semantics defined in Section 3.2 taking into account also object values. First, the set of the possible collecting value associated with a variable contains also sets of object and hence it is extended as

$$\text{VAL}^P \triangleq \text{VAL} \cup \wp(\text{OBJ})$$

where VAL has been defined in Chapter 3 and it is $\text{VAL} = \wp(\text{INT}) \cup \wp(\text{BOOL}) \cup \wp(\text{STR}) \cup \wp(\{\text{NaN}\}) \cup \wp(\{\uparrow\})$. We recall that collecting memories $\mathbb{M} : \text{ID} \rightarrow \text{VAL}^P$, ranged over m , associate with each variable a set of values. Also the language of edge labels of a control-flow graph $\mu\text{JS-CFG}$ changes as follows.

$$\mu\text{JS-CFG} \ni 1 ::= \dots \mid x = \text{new } o \mid x[s] = e$$

since a control-flow graph edge can be also labeled with object constructor and object-property update statements. Next, we need to define how these new statements act on a collecting memory, extending the collecting semantics $(\llbracket \text{st} \rrbracket) : \mathbb{M} \rightarrow \mathbb{M}$

as follows.

$$\begin{aligned} \llbracket x = \text{new } o \rrbracket \mathfrak{m} &\triangleq \mathfrak{m}[x \leftarrow \llbracket o \rrbracket \mathfrak{m}] \\ \llbracket x[s] = e \rrbracket \mathfrak{m} &\triangleq \mathfrak{m}[x \leftarrow \{ o \bullet [s \mapsto v] \mid o \in \llbracket x \rrbracket \mathfrak{m}, s \in \llbracket s \rrbracket \mathfrak{m}, v \in \llbracket e \rrbracket \mathfrak{m} \}] \end{aligned}$$

We also augment the expression collecting semantics for the novel introduced expressions, as follows.

$$\begin{aligned} \llbracket \{s_0 : e_0, s_1 : e_1, \dots, s_n : e_n\} \rrbracket \mathfrak{m} &\triangleq \left\{ [s_n \mapsto v_n] \bullet \dots \bullet [s_1 \mapsto v_1] \bullet [s_0 \mapsto v_0] \mid \begin{array}{l} \forall i \in [0, n] \\ v_i \in \llbracket e_i \rrbracket \mathfrak{m} \end{array} \right\} \\ \llbracket x[s] \rrbracket \mathfrak{m} &\triangleq \bigcup \{ o[s] \mid o \in \llbracket x \rrbracket \mathfrak{m}, s \in \llbracket s \rrbracket \mathfrak{m}, s \in \text{props}(o) \} \end{aligned}$$

Above, we use the notation $\text{props}(o) \subseteq \Sigma^*$ to denote the set of the properties of an object o . In the following, we will present the abstract domain for objects $\text{OBJ}^\#$ and the abstract semantics of the object-related statements and expressions added to μJS . In particular, we will focus on augmenting the abstract function $\llbracket \cdot \rrbracket^\#$, that is the semantic transformer of edges labels of a control-flow graph.

7.2 An abstract domain for objects

Similarly to what we have already done for strings, integers and booleans, we need also to finitely represent an infinite set of concrete objects, namely we need to design an abstract domain for concrete objects. We start here with our representation of infinite sets of objects, namely we define an abstract domain approximating $\wp(\text{OBJ})$. In particular, we first have a non-relational abstraction between objects-properties and values, i.e., we abstract $\wp(\text{OBJ})$ in $\wp(\text{STR}) \rightarrow \text{VAL}$. Then, we abstract $\wp(\text{STR})$ to the finite state automata abstract domain $\text{DFA}_{/\equiv}$ and VAL in the coalesced abstract domain Dyn defined in Section 6.2 (see Equation 6.1). Finally, we can abstract values of VAL^P in the Dyn^P abstract domain, defined as follows.

$$\text{Dyn}^P \triangleq \text{Dyn} \oplus \text{OBJ}^\#$$

where the new object abstract domain is $\text{OBJ}^\# \triangleq \text{DFA}_{/\equiv} \rightarrow \text{Dyn}$. The partial order $\sqsubseteq_{\text{Obj}^\#}$ for $\text{OBJ}^\#$ is the point-wise ordering between functions, i.e., $o_1^\# \sqsubseteq_{\text{Obj}^\#} o_2^\# \triangleq (\forall A \in \text{DFA}_{/\equiv}. o_1^\#(A) \sqsubseteq_{\text{Dyn}} o_2^\#(A))$. This order is not optimal but it does not harm the analysis since, as we can see in Section 7.2.1, the order can be strengthened.

Similarly, the least upper bound for $\text{OBJ}^\#$ is defined as $\bigsqcup_{\text{Obj}^\#} X \triangleq \lambda A \in \text{DFA}. \bigsqcup_{\text{Dyn}} \{o^\#(A) \mid o^\# \in X\}$. We can similarly define the greatest lower bound for $\text{OBJ}^\#$. It is straightforward to see that $\langle \text{OBJ}^\#, \sqsubseteq_{\text{Obj}^\#} \rangle$ is a lattice, with minimum mapping every automaton to \perp_{Dyn} , and maximum mapping every automaton to \top_{Dyn} . The concretization $\gamma_{\text{O}} \in \text{OBJ}^\# \rightarrow \wp(\text{OBJ})$ is defined as:

$$\gamma_{\text{O}}(o^\#) \triangleq \left\{ o \in \text{OBJ} \mid \begin{array}{l} \forall \sigma \in \text{STR} \exists A \in \text{DFA}_{/\equiv}. \\ (\sigma \in \gamma_{\text{DFA}}(A) \wedge o(\sigma) \in \gamma_{\text{Dyn}}(o^\#(A)) \vee o(\sigma) = \uparrow) \end{array} \right\}$$

In order to show how our object abstract domain works, we consider a simple yet expressive μJS example (Figure 7.2, where we suppose that the boolean guards of `while` and `if` statements are statically unknown). The fragment declares the object `o` at line 1, and its abstract value at lines 1-9 is reported in Figure 7.3a. Then, it indefinitely iterates over the string variable `idx` at lines 3-6 appending either the

```

1  o = new {x:1, y:2, z:3};
2  idx = "x";
3  while (?) {
4    if (?) {
5      idx = concat(idx, "x")
6    } else {
7      idx = concat(idx, "y")
8    }
9  };
10 o[idx] = 7;

```

FIGURE 7.2: μ JS program example.

strings "x" or "y". Finally, `idx` is used to access the object `o` at line 7. Let us suppose to statically analyze the above program with the abstract domain previously presented. Since the number of iterations of the `while`-loop is statically unknown, the computation of the value of `idx`, abstracted as a finite state automaton, may diverge. In order to enforce termination, the automata widening $\nabla_{\text{DFA}/\equiv}^n$ is applied. Tuning $\nabla_{\text{DFA}/\equiv}^n$ with $n = 3$, the abstract value of `idx` at line 10, after the `while` computation, corresponds to the automaton expressed by the regular expression $x(x \parallel y)^*$. Since `idx` does not represent just a single string, when we analyze `o[idx]` we may have to overwrite an object property (e.g., `x`) and add new properties to `o` (e.g., `xy`). Since the abstract value of `idx` expresses an infinite number of object properties, we call this property *summary property*. The abstract value of `o` after line 10 is depicted in Figure 7.3b, where the summary property $x(x \parallel y)^*$ is added to the object reported in Figure 7.3a. Note that in the abstract object updated at line 10, the abstract properties `x` and $x(x \parallel y)^*$ share the common concrete property `x`. In particular, the value of `o["x"]` may be either 1 or 7. We aim at an objects representation where every property does not share any property with the others, namely when objects are in normal form.

7.2.1 Normalization

We now formally define the notion of abstract object normal form. Given an abstract object $o^\# \in \text{OBJ}^\#$, we abuse notation denoting by $\text{props}(o^\#) \subseteq \text{DFA}/\equiv$ the set of its abstract properties, namely the properties which are not undefined. Formally,

$$\begin{array}{ccc}
 \left[\begin{array}{c} x \mapsto [1,1] \\ y \mapsto [2,2] \\ z \mapsto [3,3] \\ \hline - \end{array} \right] & \left[\begin{array}{c} x \mapsto [1,1] \\ y \mapsto [2,2] \\ z \mapsto [3,3] \\ \hline x(x \parallel y)^* \mapsto [7,7] \end{array} \right] & \left[\begin{array}{c} x \mapsto [1,7] \\ y \mapsto [2,2] \\ z \mapsto [3,3] \\ \hline x(x \parallel y)^+ \mapsto [7,7] \end{array} \right] \\
 \text{(A)} & \text{(B)} & \text{(C)}
 \end{array}$$

FIGURE 7.3: (a) Abstract value of `o` after line 1 of the fragment reported in Figure 7.2 (b) Abstract value of `o` after line 10. (c) Normal form of `o` after line 10.

$\text{props}(o^\#) \triangleq \{p \in \text{DFA}_{/\equiv} \mid o^\#(p) \neq \uparrow\}$. Abstract properties represent sets of concrete properties. Since abstract properties are represented as finite state automata, we can refer to the set of the concrete properties captured by the abstract one. Hence, given an abstract property $p \in \text{props}(o^\#)$, we abuse notation denoting by $\mathcal{L}(p)$ the language of the concrete properties captured by p .

Definition 7.1 (Abstract object normal form). An abstract object $o^\# \in \text{OBJ}^\#$ is in normal form when:

$$\forall p \in \text{props}(o^\#). |\mathcal{L}(p)| \in \{1, \omega\} \wedge \forall p_1, p_2 \in \text{props}(o^\#). \mathcal{L}(p_1) \cap \mathcal{L}(p_2) = \emptyset$$

Informally speaking, we say that an abstract object is in normal form when each property p represents only a single string (i.e., $|\mathcal{L}(p)| = 1$) or an infinite language (i.e., $|\mathcal{L}(p)| = \omega$) and it does not share any concrete property with other abstract properties. Hence, a normal form abstract object has two kinds of properties: p is a *non-summary property*, if $|\mathcal{L}(p)| = 1$, and p is a *summary property*, if $|\mathcal{L}(p)| = \omega$. For instance, the abstract object in Figure 7.3a is in normal form, since any abstract property expresses concrete properties that are not expressed by other abstract properties and it only contains non-summary properties. Instead, the abstract object in Figure 7.3b is not in normal form, despite it has only summary and non-summary properties, since the string x is expressed by the non-summary property x and by the summary property $x(x \parallel y)^*$.

During abstract computations, it may happen that abstract objects are not in normal form, so we need to normalize them. We rely on the function $\text{Norm} : \text{OBJ}^\# \rightarrow \text{OBJ}^\#$ that normalizes an abstract object and its behavior is captured by the algorithm reported by Algorithm 14.

The first part of the algorithm, namely lines 1-6, checks if any property of $o^\#$ is summary or non-summary. If it finds a property p such that $\mathcal{L}(p) \notin \{1, \omega\}$ then the algorithm first removes that property from the object, and then looks at its language (that is finite) and adds any single property captured by p with its old corresponding value. For example, let us consider the object $[x \parallel y \mapsto [5, 5]]$, the algorithm returns as result the normal form abstract object $[x \mapsto [5, 5], y \mapsto [5, 5]]$. The idea of the second part of Algorithm 14 (lines 7-17) is to check, for any $p_1 \in \text{props}(o^\#)$, if it shares at least a concrete property with any other $p_2 \in \text{props}(o^\#)$ (lines 11-16). This boils down to check whether the intersection between p_1 and p_2 is not empty. If so, three new abstract properties are created in $o^\#$ (note that p_1 is removed at line 8 and p_2 will be removed at line 16). In particular:

- the property $p_1 \sqcap_{\text{DFA}} p_2$ points to the join of the previous values of p_1 and p_2 and the previous value (if present) of $p_1 \sqcap_{\text{DFA}} p_2$ in $o^\#$ (line 16);
- the property $p_1 \searrow_{\text{DFA}/\equiv} p_2$ points to the join of the previous value of p_1 and the previous value (if present) of $p_1 \searrow_{\text{DFA}/\equiv} p_2$ in $o^\#$ (line 17);
- the property $p_2 \searrow_{\text{DFA}/\equiv} p_1$ points to the join of the previous value of p_2 and the previous value (if present) of $p_2 \searrow_{\text{DFA}/\equiv} p_1$ in $o^\#$ (line 18);

Otherwise, if p_1 does not share any property with other abstract properties of $o^\#$, the association $[p_1 \mapsto o^\#(p_1)]$ is simply added to $o^\#$ (line 17). For example, let us consider again the abstract object reported in Figure 7.3b. The result obtained by applying Algorithm 14 is the abstract object reported in Figure 7.3c.

Proposition 7.2. Given $o^\# \in \text{OBJ}^\#$, the abstract object $\text{Norm}(o^\#)$, computed by Algorithm 14, is in normal form (Definition 7.1). Moreover, we have that $\gamma_{\text{O}}(o^\#) = \gamma_{\text{O}}(\text{Norm}(o^\#))$.

Algorithm 14: $\text{Norm} \in \text{OBJ}^\# \rightarrow \text{OBJ}^\#$ algorithm

Data: $o^\# \in \text{OBJ}^\#$
Result: $\text{Norm}(o^\#)$

```

1 foreach  $p \in \text{props}(o^\#)$  do
2    $v^\# \leftarrow o^\#(p)$ ;
3   if  $|\mathcal{L}(p)| \notin \{1, \omega\}$  then
4     remove  $p$  from  $o^\#$ ;
5     foreach  $s \in \mathcal{L}(p)$  do
6        $o^\# \leftarrow o^\# \bullet [s \mapsto v^\#]$ ;
7     end
8   end
9 end
10 foreach  $p_1 \in \text{props}(o^\#)$  do
11    $v_1^\# \leftarrow o^\#(p_1)$ ; remove  $p_1$  from  $o^\#$ ;  $\text{normalized} \leftarrow \text{false}$ ;
12   foreach  $p_2 \in \text{props}(o^\#)$  do
13      $v_2^\# \leftarrow o^\#(p_2)$ ;
14     if  $p_1 \sqcap_{\text{DFA}} p_2 \neq \text{Min}(\emptyset) \wedge p_1 \neq p_2$  then
15        $\text{normalized} \leftarrow \text{true}$ ;
16        $o^\# \leftarrow o^\# \bullet [p_1 \sqcap_{\text{DFA}} p_2 \mapsto o^\#(p_1 \sqcap_{\text{DFA}} p_2) \sqcup_{\text{Dyn}} v_1^\# \sqcup_{\text{Dyn}} v_2^\#]$ ;
17        $o^\# \leftarrow o^\# \bullet [p_1 \setminus_{\text{DFA}/\equiv} p_2 \mapsto o^\#(p_1 \setminus_{\text{DFA}/\equiv} p_2) \sqcup_{\text{Dyn}} v_1^\#]$ ;
18        $o^\# \leftarrow o^\# \bullet [p_2 \setminus_{\text{DFA}/\equiv} p_1 \mapsto o^\#(p_2 \setminus_{\text{DFA}/\equiv} p_1) \sqcup_{\text{Dyn}} v_2^\#]$ ;
19       remove  $p_2$  from  $o^\#$ ;
20     end
21   end
22   if  $\text{!normalized}$  then  $o^\# \leftarrow o^\# \bullet [p_1 \mapsto v_1^\#]$ ;
23 end
24 return  $o^\#$ ;

```

As we have previously mentioned in this section, normalization strengthens the abstract order between objects. For example, the objects $[a \mapsto [1, 1], b \mapsto [1, 1]]$ and $[a \parallel b \mapsto [1, 2]]$ are not comparable, but, if we normalize the second object (i.e., in $[a \mapsto [1, 2], b \mapsto [1, 2]]$), then we have $[a \mapsto [1, 1], b \mapsto [1, 1]] \sqsubseteq_{\text{Obj}} \text{Norm}([a \parallel b \mapsto [1, 2]])$.

7.2.2 Objects-related abstract semantics

In this section, we focus on defining the abstract semantics of the novel objects-related statements and expressions we have added to μJS . In particular, we define the abstract semantics of an object expression, an object-access property ($x[s]$), object constructor ($x = \text{new } o$) and object-property update ($x[s] = e$).

The object evaluation is straightforward and follows the concrete semantics reported in the previous section.

$$\langle \{ \sigma_0 : e_0, s_1 : e_1, \dots, s_n : e_n \} \rangle^\# m^\# \triangleq [\sigma_n \mapsto \langle e_n \rangle^\# m^\#] \bullet \dots \bullet [\sigma_1 \mapsto \langle e_1 \rangle^\# m^\#] \bullet [\sigma_0 \mapsto \langle e_0 \rangle^\# m^\#]$$

The abstract semantics of an object property $x[s]$ is reported in the following, supposing that $A = \langle s \rangle^\# m^\#$ and $o^\# = m^\#(x)$

$$\langle x[s] \rangle^\# m^\# \triangleq \bigsqcup_{\text{Dyn}} \{ o^\#(A') \mid A' \in \text{props}(o^\#), A' \sqcap_{\text{DFA}} A \neq \text{Min}(\emptyset) \}$$

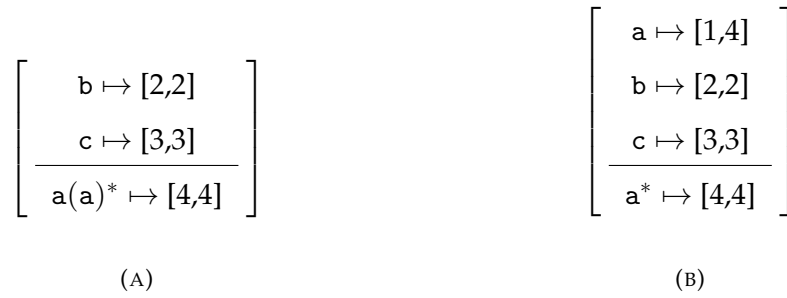


FIGURE 7.4: Example of materialization.

Since we abstract both strings and object properties to finite state automata, potentially recognizing more than a string (because of summary properties), the result of the least upper bound of any value corresponding to a property stored in x whose intersection with the automaton A is non-empty. For example, if we want to access the object $[a \leftarrow [1,1], b \leftarrow [2,2], c \leftarrow [3,3]]$ with the automaton recognizing the language $\{a, b\}$, the results would be $[1,2]$.

As far as the statement $x = \text{new } o$ is concerned, we need to store a novel abstract object into x and its semantics is straightforward and follows the concrete one. Formally:

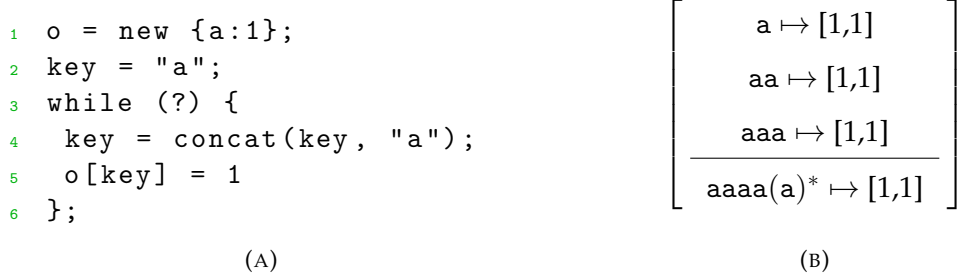
$$\llbracket x = \text{new } o \rrbracket^\# m^\# \triangleq m^\# [x \leftarrow \llbracket o \rrbracket^\# m^\#]$$

The case when we have the abstract semantics of object-property update, namely $x[s] = e$, is where *materialization* occurs. As we have already mentioned before, we allow to update only the objects that have been already stored into the memory. Suppose that $v^\# = \llbracket e \rrbracket^\# m^\#$, $A = \llbracket s \rrbracket^\# m^\#$ and $o^\# = \llbracket x \rrbracket^\# m^\#$:

$$\llbracket x[s] = e \rrbracket^\# m^\# \triangleq m^\# [x \leftarrow \text{Norm}(o^\# \bullet [A \leftarrow o^\#(A) \sqcup_{\text{Dyn}} v^\#])]$$

The abstract semantics of $x[s] = e$ needs to update properties of abstract objects stored into the variable x . In particular, the object stored into x must be updated at the property A , corresponding to the evaluation of the expression s , with the least upper bound between $v^\#$ (i.e., the abstract evaluation of the expression s) and the previous value stored at $o^\#(A)$. Before storing the updated abstract object in $m^\#$, the latter is normalized. It is worth noting that here we perform a *weak update* of the object contained in x [Balakrishnan and Reps, 2006]. We could improve the precision of the analysis performing a *must-may analysis* (as it has been done in [Fromherz, Ouadjaout, and Miné, 2018] for Python) in order to distinguishing between properties that certainly point to some value and properties that may point to others. This can be done improving the proposed analysis using standard techniques, such as the ones reported in [Balakrishnan and Reps, 2006; Nielson, Nielson, and Hankin, 1999; Wilhelm, Sagiv, and Reps, 2000].

For example, let us suppose that $m^\#(x)$ is the object reported in Figure 7.4a and we want to update the property a , with the interval $[1,1]$. Applying these values to the previously defined abstract semantics, we obtain the abstract object reported in Figure 7.4b, that it is stored into the variable x . We say that the property a has been *materialized*, since, before the update, it was part of a summary property, and after the update it is a non-summary property. We say that a (concrete) property is materialized when a string of an abstract object passes, during the update, from a

FIGURE 7.5: (a) μ JS fragment, (b) Value of o after `while`-loop.

summary property to a non-summary property. It is worth noting that normalization takes care of materialization. The abstract semantics is sound w.r.t. the concrete collecting semantics, i.e., it computes an over-approximation of state invariants at any statement.

Theorem 7.3 (Soundness). *The abstract semantics of object assignment, object update and object expressions is sound, namely $\forall m^\# \in \mathbb{M}^\#$*

$$(\text{st})\gamma_{m^\#}(m^\#) \subseteq \gamma_{m^\#}((\text{st})^\# m^\#) \quad (\text{e})\gamma_{m^\#}(m^\#) \subseteq \gamma_{\text{Dyn}}((\text{e})^\# m^\#)^1$$

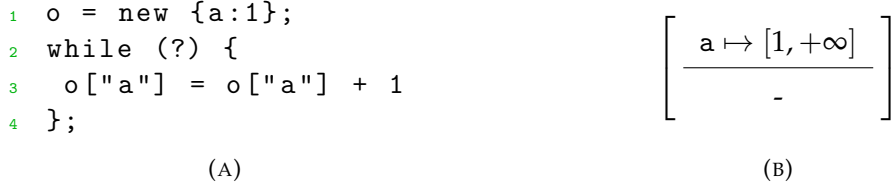
7.2.3 Widening

Note that, the augmented abstract domains Dyn^P defined in this section is not ACC, because of the intervals abstract domain, the automata abstract domain and the novel objects abstract domain. Hence, fix-point computations may diverge without introducing an extrapolation operator. We have already defined the widening operator for finite state automata and for intervals. Here, we define the widening operator also for the novel object abstract domain.

In particular, $\text{Obj}^\#$ is not ACC because we model objects properties with the finite state automata domain, which is not ACC. Anyway, a slight extension of the least upper bound \sqcup_{Obj} is enough to guarantee termination of computations, exploiting the widening of the finite state automata domain. Informally speaking, abstract string values, in `while`-loop computations, always converge since finite state automata domain is equipped with a widening. Consequently, since the object properties are finite state automata, properties also converge. For this reason, we cannot have infinite ascending chains depending on the objects properties, but only depending on the values that are associated with them.

Let us consider the μ JS fragment reported in Figure 7.5a and suppose that the boolean guard value is statically unknown. At each iteration on the `while`-loop, the string "a" is concatenated to the string value of `key` and then it is used to add a new property to the object `o`. If the $\text{DFA}_{/\equiv}$ were not equipped with a widening, the computation of the value of `key` would diverge. Since convergence of string computations is enforced by the widening $\nabla_{\text{DFA}_{/\equiv}}^n$ (with $n = 3$), the computations of object properties of `o` converge. Indeed, the `while`-loop converges and the abstract values of the variable `o` is the (normalized) object reported in Figure 7.5b. Clearly, the simple object join is enough for objects properties convergence but it is not for the associated value. For example, let us consider the μ JS fragment reported in Figure 7.6a. In this case, the number of properties of the object `o` does not increase in the `while`-loop but the value of the property `a` increases at each iteration. The

¹ $\gamma_{m^\#}$ is the concretization function on Dyn abstract memories defined as $\gamma_{m^\#}(m^\#) \triangleq \{ x \mapsto \gamma_{\text{Dyn}}(v) \mid x \mapsto v \in m^\# \}$

FIGURE 7.6: (a) μ JS fragment, (b) Value of o after while-loop.

idea behind the widening for objects is to apply the widening of values point-wisely between the properties of the two objects. Hence, we define the widening on $\text{Obj}^\#$ as: $o_1^\# \nabla_{\text{Obj}} o_2^\# \triangleq \lambda p. o_1^\#(p) \nabla_{\text{Dyn}}^n o_2^\#(p)$. Coming back to the example, applying the widening defined above, the abstract value of o after the while-loop is reported in Figure 7.6b. At this point, we can integrate this widening into the one for abstract memories and abstract stores.

Example. We now illustrate the so far defined analysis on the example reported in the introduction (Figure 7.1). It is worth noting that, in this example, objects widening does not occur. We have already commented it with the fragments reported in Figure 7.5 and Figure 7.6. The goal of the analysis is to reason about the value of idx (and, in turn, of n) at the end of the execution. At the beginning of the first iteration of the while loop, the value of n is $[0, 0]$ and the value of idx is the automaton expressed by the regular expression $a \parallel b$. The latter is used during the first iteration to access obj and then the result is stored in n (line 9). Since the property b is not present in obj , only the property a is accessed by idx , and the value of n is $[1, 1]$. Before starting the next iteration, idx is updated at line 10 and its value becomes the automaton recognizing $aa \parallel ba$.

Widening is applied before starting new iterations. Supposing to apply the widening $\nabla_{\text{DFA}/\equiv}^n$, with $n = 1$, and the widening for intervals, the values of the variables before the second iteration are: $n = [0, +\infty]$, $idx = a \parallel b \parallel aa \parallel ab$ since, in this case, the widening for automata coincides with the automata join. In the second iteration idx accesses the properties a and aa , hence n gets the value $[1, +\infty] = [0, +\infty] + ([1, 1] \sqcup_{\text{Ints}} [2, 2])$. Similarly to the previous iteration, idx becomes $aa \parallel ba \parallel aaa \parallel aba$. Before starting the new iteration we apply the widening, obtaining the values $n = [0, +\infty]$ and $idx = (a \parallel b) a^*$. The third iteration does not change the values of n and idx , hence the fixpoint is reached.

Finally, at line 12, the value of n is assigned to $obj[idx]$, updating the abstract object obj as follows: $[a, aa \mapsto [0, +\infty], ab \mapsto [3, 3], ac \mapsto \text{Min}(\{\text{"world"}\}), (a \parallel b) a^* \setminus \{a, aa\} \mapsto [0, +\infty]]$. The summary property $(a \parallel b) a^* \setminus \{a, aa\}$ is added and only the properties a and aa are modified. Properties already present in obj remain unaltered (e.g., ab and ac).

Final remarks. In this chapter we have presented a preliminary approach for analyzing objects in dynamic languages, exploiting again finite state automata to abstract objects properties. We have proposed an abstract domain suitable for the analysis of objects properties in dynamic programming languages. The novelty consists in exploiting finite state automata, in order to approximate objects properties. This leads to a better precision (less false positives), compared to state-of-the-art domains approximating strings (for instance, [Costantini, Ferrara, and Cortesi, 2015]). A key aspect of our abstract domain is the *normal form for objects* and we have presented a normalization algorithm: it transforms objects in their normal form. An object is

in normal form if and only if it has only two kind of properties: *summary* and *non-summary*. The idea behind summarization, and hence materialization, is not new in static analysis, and comes from the well-known shape analysis [Nielson, Nielson, and Hankin, 1999]. For example, this idea has been adopted in [Hauzar and Kofron, 2015a], where the authors present a static analyzer for PHP that also involve heap analysis, where the heap, in their abstraction, is made of summary heap identifiers and non-summary heap identifiers. In particular, in [Hauzar and Kofron, 2015a], a summary heap identifier summarizes all the elements of the heap that could be updated by statically unknown assignments. We have adopted the same idea with the difference that we may have more summary properties, expressed by automata recognizing infinite languages, rather than a single summary property that merges together heap elements updated by statically unknown assignments. The idea of summarization has been also taken into account in [Balakrishnan and Reps, 2006], where the authors propose the recency abstraction, which consists in representing each abstract allocation site with two memory regions, namely the *most recently allocated block* and the *not most recently allocated blocks*. The latter is basically a summary memory region, since more than one block may be allocated. Recency abstraction has been implemented also in TAJIS [Jensen, Møller, and Thiemann, 2009], showing that such abstraction outperforms other abstract allocation-based techniques. As future work, we aim to implement our objects abstract domain upon TAJIS. We believe that the combination of our abstract domain and the recency abstraction can produce good results, w.r.t. analysis precision, and it would be interesting to make a comparison with TAJIS and other JavaScript static analyzers, such as SAFE [Lee et al., 2012] and JSAI [Kashyap et al., 2014]. Finally, a similar work has been proposed in [Ko, Rival, and Ryu, 2019], where the authors propose a weakly sensitive analysis for objects manipulated into loops and fix-point computations, in the context of JavaScript. The authors build their analysis upon SAFE and focuses more on improving precision on loop computations and seem to have, in the object abstraction, a single summary property. As future work, it will be interesting to study the relation between the work proposed in this chapter and the work proposed in [Ko, Rival, and Ryu, 2019] and how to integrate our object abstraction.

Chapter 8

Conclusions

In this thesis, we made a little step towards a sound abstract interpreter for dynamic string manipulation languages. In particular, we first have tackled the problem of analyzing strings, showing a novel abstract domain for strings by means of finite state automata. On this domain, we have designed a sound abstract interpreter for string-to-code statements such as `eval`, treating the code as any other data type and treating the abstract interpreter as any other abstract function. We have shown our approach for a core dynamic language, namely μ JS. The language has been defined to precisely contain the string features we faced in the thesis. Hence, for example, the language does not contain several important features of JavaScript, such as functions, object prototypal inheritance or built-in objects. At the end of the thesis, we have exploited again finite state automata to abstract object properties. This contribution must be intended as a proof of concept, since the object extension we made for μ JS still misses important JavaScript object features (e.g., property deletion or inheritance).

In this chapter we conclude the thesis, reporting the most related work concerning string analysis, analysis of dynamic languages and dynamic code. Then, we will show the future directions of the work presented here.

8.1 Related works

Static analysis of strings. The issue of analyzing strings is a widely studied problem, and it has been tackled in the literature from different points of view. For this reason, the analysis of strings is nowadays a relatively common practice in program analysis due to the widespread use of dynamic scripting languages. Examples of analyses for string manipulation are in [Doh, Kim, and Schmidt, 2009; Christensen, Møller, and Schwartzbach, 2003; Yu, Alkhalaf, and Bultan, 2011; Thiemann, 2005; Minamide, 2005; Kim, Doh, and Schmidt, 2013; Loring, Mitchell, and Kinder, 2019]. The use of symbolic (grammar-based) objects in abstract domains is also not new (see [Cousot and Cousot, 1995; Heintze and Jaffar, 1994; Venet, 1999]) and some works explicitly use transducers for string analysis in script sanitisation ([Hooimeijer et al., 2011] and [Yu, Alkhalaf, and Bultan, 2011]), 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. In [Yu et al., 2008], the authors propose a symbolic string verifier for PHP based on finite state automata, represented by a particular form of binary decision diagrams, the MBDD. Even if it could be interesting to understand whether this representation of DFAs may be used also for improving our algorithms, their work only considers operations exclusively involving strings (not also integers such as `substring`) and therefore it provides a solution for different string manipulations.

In [Choi et al., 2006], the authors propose an abstract interpretation-based string analyzer approximating strings into a subset of regular languages, called *regular strings* and they define the abstract semantics for four string operations of interest together with a widening. As far as our string analysis based on finite state automata is concerned, this is the most related work, but our approach is strictly more general, since we do not introduce any restriction to regular languages and we abstract integers on intervals instead of on constants (meaning that our domain is strictly more precise). In [Park, Im, and Ryu, 2016], the authors propose a scalable static analysis for jQuery that relies on a novel abstract domain of regular expressions. The abstract domain in [Park, Im, and Ryu, 2016] contains the finite state automata one but pursues a different task and does not provide semantics for string manipulations. Surely it may be interesting to integrate our library for string manipulation operators into SAFE. The authors of [Midtgaard, Nielson, and Nielson, 2016] propose a lattice-based generalization of regular expression, formally illustrating a parametric abstract domain of regular expressions starting from a complete lattice of reference. However, this work does not tackle the problem of analyzing string manipulations, since it instantiates the parametric abstract domain in the network communication environment, analyzing the exchanged messages as regular expressions.

Finite state machines (transducer and automata) have found a critical application also in model checking both for enforcing string constraints and to model infinite transition systems [Lin and Barceló, 2016]. For example, the authors of [Abdulla et al., 2014] define a sound decision procedure for a regular language-based logic for verification of string properties. The authors of [Bouajjani, Habermehl, and Vojnar, 2004] propose an automata abstraction in the context of regular model checking to tackle the well-known problem of state space explosion. Moreover, other formal systems, similar to DFA, have been proposed in the context of string analysis [Bouajjani et al., 2008; Alur and Madhusudan, 2004; Holík et al., 2018]. As future work, it can be interesting to study the relation between standard DFA and the other existing formal models, such as logics or other forms of FA.

Static analysis of dynamic languages. During the years both the programming language research community and the industry have spent much effort in analyzing dynamic languages. For PHP language several tools and analyses have been proposed with the common purpose of detecting or mitigating vulnerabilities and security holes on web applications [Hauzar and Kofron, 2015b; Hauzar and Kofron, 2014; Wilhelm, Sagiv, and Reps, 2000; Jovanovic, Krügel, and Kirda, 2006; Dahse and Holz, 2014; Kneuss, Suter, and Kuncak, 2010; Saxena et al., 2010]. Nevertheless, none of them faced the problem of analyzing programs that dynamically generated code. Even for JavaScript, very few static analyzers tried to face the problem of statically analyzing `eval` and its variants. For example, some analyses are based on JavaScript subsets that do not include `eval`, forbid its usage [Anderson, Gianini, and Drossopoulou, 2005] or simply ignore its effects [Guarnieri et al., 2011], making the analysis drastically imprecise or even unsound. In the following, we briefly present some of the most important JavaScript static analyzers that have inspired this thesis, with the goal of improving the precision of JavaScript analyses, especially in the case of dynamic code generation.

We have already introduced TAJIS [Jensen, Jonsson, and Møller, 2012], but other static analyzers for JavaScript, based on abstract interpretation, have been developed, such as JSAI [Kashyap et al., 2014] and SAFE [Lee et al., 2012]. They aim

at a flexible, configurable and tunable tool focusing on context-sensitiveness, heap-sensitiveness [Kashyap et al., 2014] and loop-sensitiveness [Lee et al., 2012]. Nevertheless, they do not explicitly mention solutions to analyze dynamically generated code by `eval`. TamiFlex [Bodden et al., 2011] also synthesizes a program at every `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 with respect to the considered execution traces, producing a warning otherwise. Similarly, in [Loring, Mitchell, and Kinder, 2019], the authors relies on regular expressions proposing in dynamic symbolic execution for JavaScript, modeling the complete regular expression language of ECMAScript 6.

Static analysis for a static subset of PHP (i.e., ignoring `eval`-like primitives) has been developed in [Biggar and Gregg, 2009]. Static *taint analysis* keeping track of values derived from user inputs has been developed for self-modifying code by partial derivation of the control-flow graph [Wang et al., 2008]. The approach is limited to taint analysis, e.g., for limiting code-injection attacks. Staged information flow for JavaScript in [Chugh et al., 2009] with *holes* provides a conditional (a la abduction analysis in [Giacobazzi, 1998]) static analysis of dynamically evaluated code. Symbolic execution-based static analyses have been developed for scripting languages, e.g., PHP, including primitives for code reflection, still at the price of introducing false negatives [Xie and Aiken, 2006].

We are not aware of effective general purpose sound static analyses handling self-modifying code for high-level scripting languages. On the contrary, a huge effort was devoted to bring static type inference to object-oriented dynamic languages (e.g., see [An et al., 2011] for an account in Ruby) but with a different perspective: *Bring into dynamic languages the benefits of static ones – well-typed programs don't go wrong*. Our approach is different: *Bring into static analysis the possibility of handling dynamically mutating code*. A similar approach is in [Anckaert, Madou, and Bosschere, 2006]. The idea is that of 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 [Cai, Shao, and Vaynberg, 2007], the approach to self-modifying code consists in treating machine instructions as regular mutable data structures, and to incorporate a logic dealing with code mutation within a la Hoare logics for program verification.

8.2 Future directions

The approach of the thesis has been provided for a running core dynamic language, namely μ JS. Clearly, this is not a real-world programming language. Anyway, we are already integrating finite state automata and the `eval` analysis upon TAJs, in order to analyze real JavaScript programs.

As far as string analysis is concerned, we have presented the abstract semantics of five popular string operations, but the static analyzer implementation also contains other string operations taken from the built-in JavaScript global object `String`, as we have previously discussed. Nevertheless, we still miss some string operation, such as `replace`, and, more important, there is no support for JavaScript regular expressions. We are already working to fully cover the operations offered by the `String` global object, providing also support for JavaScript regular expressions. The

finite state automata domain is equipped with a widening in order to enforce convergence in fix-point computations. As we have mentioned in Section 2.5, abstract interpreters only equipped with widening may lead to a big loss of precision. For this reason, other investigations will be addressed in order to provide a narrowing for automata.

We are strongly confident the proposed approach for analyzing string-to-code statements places an important starting point for analyzing real-world programs that dynamically change their own code at run-time. We think that an important application is for analyzing JavaScript malware. As shown in the introduction, hiding the malicious intent into strings to be later converted to executable code seems to be a common practice in malware, also because of the lack of analyzers for such programs and the difficulty behind analyzing self-modifying code.

We have discussed the limitations of the analysis of `eval` in Section 6.6.1. In particular, problems arise when the cycles inside the automaton from which an executable program must be extracted do not read executable statements. Cycles in automaton occurs because of widening application in fix-points computations. We have already mentioned narrowing to mitigate this problem but also smarter techniques to improve precision of loop computations may be integrated in order to remove cycles in the automata constructed by fix-point computations. For instance, we think widenings with threshold and loop unrolling may decrease, in certain cases, the number of false positive values, obtaining automata without cycles. Also loop analysis techniques such as the one reported in [Park and Ryu, 2015] may help to obtain more precise string abstract values in loop computations. Further investigations will address our future works in order to improve the string analysis proposed here.

Rather than improving the underlying string analysis, we are currently investigating another (and more general) solution trying to answer about those `eval` patterns such that their inputs are abstracted to non-cycle executable automata. In this work-in-progress investigation, the idea is to formally relate semantic and syntactic abstractions. In the thesis we have focused only of semantic abstractions, acting on data abstraction (e.g., finite state automata) and standard control-flow graphs. We are currently studying how to involve syntactic abstraction, namely code abstraction, that does not interfere with the semantic one. Informally speaking, the aim is to abstract the syntax of the language of interest without changing the abstract interpreter we want to involve for answering about programs written in that language. The non-interference relation between syntactic and semantic abstractions is formalized in terms of forward completeness. In order to give the flavor of what it is our goal, let us consider the example reported in Section 6.6.1. As we have already mentioned before, our `eval` analysis fails to synthesize, to a control-flow graph, the automaton recognizing the language $\mathcal{L} = \{ x=(5)^n; \mid n > 0 \}$, since the automaton is non cycle-executable. Supposing to perform a sign analysis, it is worth noting that any executable string of \mathcal{L} is not distinguishable for the sign analysis, since the abstract execution of any executable string would assign the positive abstract value to `x`. Hence, the idea is to abstract the language syntax adding an abstract statement `x+=;` such that its concretization contains any statement that assigns positive values to `x`. Namely, the syntactic abstraction merges statements that cannot be distinguished by the semantic abstraction. In this sense, we say that the syntax abstraction does not interfere with the semantic one, ensuring to still preserving the property we want to analyze (e.g., signs in the above example).

Finally, in the thesis we have reported several μ JS examples showing that our approach is promising for analyzing more complex JavaScript programs but we are

aware of the fact that further comparisons with existing static analyzers must be taken into account in future works in order to validate our approach, also in more challenging real world JavaScript test cases. The absence of a more advanced comparison is due to the absence of a static analyzer for JavaScript, but only for a dynamic core language. Nevertheless, as we have already mentioned, we are integrating finite state automata abstract domain, the corresponding string analysis and the analysis of `eval` upon TAJs. This would permit us to better validate the approach by showing a comparison with the other static analyzers.

Appendix A

Proofs

In this appendix we report all the long proofs of the results presented in the thesis. The proofs are listed in order of appearance. Some other theorems and lemmas, which are needed by the proofs, are also presented here.

Theorem 4.2

Proof sketch of Theorem 4.2. Following Theorem 2.40, the result of the procedure to obtain the absolute complete shell of ρ_{SF} with respect to the concat operation is given below. In particular, we obtain the greatest fix-point after one step, computing:

$$\rho_{SF} = \mathcal{M}(\rho_{SF} \cup (\bigcup_{z \in \wp(\Sigma^*), y \in \rho_{SF}} \max(\{ x \in \wp(\Sigma^*) \mid \llbracket \text{concat}(z, x) \rrbracket \subseteq y \})))$$

since $\bigcup_{z \in \wp(\Sigma^*), y \in \rho_{SF}} \max(\{ x \in \wp(\Sigma^*) \mid \llbracket \text{concat}(x, z) \rrbracket \subseteq y \})$ does not add new abstract points to the complete shells.

Table A.1 shows how the abstract points of the complete shell are added (in particular, the ones reported in Figure 4.4 in the dashed boxes). The points in the standard boxes are added by Moore closure of the other points. In Table A.1, it is possible to check, for each cell, that no other string values are added to the ones reported inside the cell, since they would violate the dominance relation expressed by Theorem 2.40. □

Theorem 4.4

Proof sketch of Theorem 4.4. Following Theorem 2.39, the result of the procedure to obtain the complete shell of ρ_{TJ} relative to ρ_{TJ_N} with respect to the toNum operation is given below.

$$\rho_{TJ} = \mathcal{M}(\rho_{TJ} \cup (\bigcup_{y \in \rho_{TJ_N}} \max(\{ z \in \wp(\Sigma^*) \mid \llbracket \text{toNum}(z) \rrbracket \subseteq y \})))$$

Since \top_{TJ} and \perp_{TJ} trivially belongs to ρ_{TJ} , and the singleton strings do not add any novel abstract value in the complete shell, the only points we need to add to ρ_{TJ} are the ones reported in the following. Note that the concretization of the abstract value NotUnsignedInt in TJ are all the float strings and the non-numerical strings.

$$\begin{aligned} & \max(\{ z \in \wp(\Sigma^*) \mid \llbracket \text{toNum}(z) \rrbracket \subseteq \gamma_{TJ_N}(\text{UnsignedInt}) \}) \\ &= \max(\{ z \in \wp(\Sigma^*) \mid \llbracket \text{toNum}(z) \rrbracket \subseteq \{ n \mid n \in \mathbb{N} \} \}) \\ &= \{ \sigma \in \Sigma^* \mid \sigma \text{ is not numeric} \} \cup \{ \sigma \in \Sigma^* \mid \sigma \text{ is unsigned integer string} \} \end{aligned}$$

$y \setminus z$	T	Numeric	NotNumeric	"n" $\in \mathbb{Z}$	"f" $\in \mathbb{F}$	"s" $\in \text{NotNum}$	\perp
T	T	\perp	\perp	\perp	\perp	\perp	\perp
Numeric	T	$\{\text{"n"}\} \cup \text{U}Int$	$[\text{NotNum} \setminus \{\text{"n"}\}] \cup \text{NotU}Int \cup \text{NotU}Float$	\perp	\perp	\perp	\perp
NotNumeric	T	\perp	NotNumeric	\perp	\perp	\perp	\perp
"n", n is integer	T	$\{\text{"n"}\} \cup \text{U}Int \cup \text{U}Float$	$[\text{NotNum} \setminus \{\text{"n"}\}] \cup \text{NotU}Int \cup \text{NotU}Float$	$\perp \vee \sigma \in \Sigma^*$	$\perp \vee \sigma \in \Sigma^*$	$\perp \vee \sigma \in \Sigma^*$	\perp
"f", f is float	T	$\{\text{"n"}\} \cup \text{U}Int$	$[\text{NotNum} \setminus \{\text{"n"}\}] \cup \text{Float} \cup \text{NotU}Int$	\perp	$\perp \vee \sigma \in \Sigma^*$	$\perp \vee \sigma \in \Sigma^*$	\perp
"s" $\in \text{NotNum}$	T	\perp	NotNumeric	\perp	\perp	$\perp \vee \sigma \in \Sigma^*$	\perp
\perp	T	T	T	T	T	T	T

TABLE A.1: SAFE completion

= concretization of UnsignedOrNotNumeric

Any other string is not contained in the maximum set of strings whose image of toNum is dominated by elements of UnsignedInt , since toNum operation on signed number strings (i.e. "f" $\in \Sigma^*$ s.t. f is a float or signed number) return either signed numbers or float numbers, that are not dominated by elements of UnsignedInt .

$$\begin{aligned}
& \max(\{ z \in \wp(\Sigma^*) \mid \llbracket \text{toNum}(z) \rrbracket \subseteq \gamma_{\mathcal{T}, \mathcal{J}_N}(\text{NotUnsignedInt}) \}) \\
&= \max(\{ z \in \wp(\Sigma^*) \mid \llbracket \text{toNum}(z) \rrbracket \subseteq \{ \sigma \in \Sigma^* \mid \sigma \text{ is signed or float string} \} \}) \\
&= \{ \sigma \in \Sigma^* \mid \sigma \text{ is a float string or a signed number string} \} \\
&= \text{concretization of SignedOrFloat}
\end{aligned}$$

Any other string is not contained in the maximum set of strings whose image of toNum is dominated by elements of NotUnsignedInt , since toNum operation on unsigned strings (i.e. "n" s.t. n is a natural) or not-numerical strings return 0, that is

not dominated by elements of `NotUnsignedInt`

Hence, the complete shell of $\rho_{\mathcal{T}\mathcal{J}}$ relative to $\rho_{\mathcal{T}\mathcal{J}_N}$ w.r.t. `toNum` is obtained as

$$\rho_{\mathcal{T}\mathcal{J}} = \mathcal{M}(\rho_{\mathcal{T}\mathcal{J}} \cup \text{UnsignedOrNotNumeric} \cup \text{SignedOrFloat})$$

that is the abstract domain reported in Figure 4.6, where `NotNumeric` is obtained by Moore closure, namely by the intersection between the concretization of `NotUnsigned` and the concretization of `SignedOrFloats`. \square

Theorem 6.8

Proof of Theorem 6.8. Here we need to prove soundness of the abstract implicit type conversion methods `toBool#`, `toInt#` and `toString#`. As mentioned in Section 6.2.1, we focus on the implicit type conversion concerning strings, since the other cases are straightforward.

▷ `toBool#`

We abuse notation denoting by `toBool` : $\wp(\Sigma^*) \rightarrow \wp(\text{BOOL})$ the collecting semantics of the implicit boolean conversion function concerning strings, obtained lifting the definition of `toBool` on strings reported in Figure 3.2, namely

$$\text{toBool}(\mathcal{L}) \triangleq \{ \text{toBool}(\sigma) \mid \sigma \in \mathcal{L} \}$$

Here we show that `toBool#` is complete (and hence sound), formally $\forall A \in \text{DFA}_{/\equiv}$ holds

$$\text{toBool}(\gamma_{\text{Dyn}}(A)) = \gamma_{\text{Dyn}}(\text{toBool}^{\#}(A))$$

We have three cases:

- $A = \text{Min}(\{\epsilon\})$

$$\text{toBool}(\gamma_{\text{Dyn}}(A)) = \text{toBool}(\mathcal{L}(A)) = \text{toBool}(\{\epsilon\}) = \{\text{false}\} = \gamma_{\text{Dyn}}(\text{toBool}^{\#}(A))$$

- $A \sqcap_{\text{DFA}} \text{Min}(\{\epsilon\}) \neq \text{Min}(\emptyset)$. Since any non-empty string is converted to `true`, the thesis holds.

$$\text{toBool}(\gamma_{\text{Dyn}}(A)) = \text{toBool}(\mathcal{L}(A)) = \{\text{true}\} = \gamma_{\text{Dyn}}(\text{true}) = \gamma_{\text{Dyn}}(\text{toBool}^{\#}(A))$$

- In the remaining case, A recognizes both the empty string and at least a non empty string, hence `toBool`($\mathcal{L}(A)$) returns `{true, false}`

$$\begin{aligned} \text{toBool}(\gamma_{\text{Dyn}}(A)) &= \text{toBool}(\mathcal{L}(A)) = \{\text{false}, \text{true}\} \\ &= \gamma_{\text{Dyn}}(\top_{\text{Bool}}) = \gamma_{\text{Dyn}}(\text{toBool}^{\#}(A)) \end{aligned}$$

▷ `toInt#`

We abuse notation denoting by `toInt` : $\wp(\Sigma^*) \rightarrow \wp(\text{INT}) \cup \wp(\{\text{NaN}\})$ the collecting semantics of the implicit integer conversion function on strings obtained lifting the definition of `toInt` on strings reported in Figure 3.2, namely

$$\text{toInt}(\mathcal{L}) \triangleq \{ \text{toInt}(\sigma) \mid \sigma \in \mathcal{L} \}$$

Hence, in order to prove soundness we need to prove $\forall A \in \text{DFA}$ the following fact.

$$\text{toInt}(\mathcal{L}(A)) \sqsubseteq_{\text{Dyn}} \gamma_{\text{Dyn}}(\text{toInt}^{\#}(A))$$

We have three cases:

- $A \sqcap_{\text{DFA}} \text{Min}(\Sigma_{\mathbb{Z}})$. This means that A does not recognize any numerical string, consequently any string of $\mathcal{L}(A)$ is converted to NaN

$$\text{toInt}(\mathcal{L}(A)) = \{\text{NaN}\} = \gamma_{\text{Dyn}}(\text{toInt}^{\#}(A)) = \gamma_{\text{Dyn}}(\text{NaN})$$

- $A \sqsubseteq_{\text{DFA}} \text{Min}(\Sigma_{\mathbb{Z}})$. This means that A only recognizes numerical strings. In this case, the abstract function $\text{toInt}^{\#}$ checks if A either reads, from the initial state, only digits or $+$ symbol, or only the $-$ symbol. In the first case, it means that $\mathcal{L}(A)$ only contains numerical non-negative strings, that are converted, by toInt , only to non-negative numbers, hence

$$\text{toInt}(\mathcal{L}(A)) \subseteq \{n \mid n \geq 0\} \subseteq \gamma_{\text{Dyn}}(\text{toInt}^{\#}(A)) = \gamma_{\text{Dyn}}([0, +\infty))$$

In the second case, it means that $\mathcal{L}(A)$ only contains numerical non-positive strings, that are converted, by toInt , only to non-positive numbers, hence

$$\text{toInt}(\mathcal{L}(A)) \subseteq \{n \mid n \leq 0\} \subseteq \gamma_{\text{Dyn}}(\text{toInt}^{\#}(A)) = \gamma_{\text{Dyn}}([-\infty, 0])$$

If the automaton both reads, from the initial state, digits or $+$ symbol and $-$ symbol, it means that that $\mathcal{L}(A)$ contains numerical positive and negative strings, hence the thesis holds.

$$\text{toInt}(\mathcal{L}(A)) \subseteq \mathbb{Z} \subseteq \gamma_{\text{Dyn}}(\text{toInt}^{\#}(A)) = \gamma_{\text{Dyn}}([-\infty, +\infty])$$

- In the last case, A can recognize both numerical and non-numerical strings and $\text{toInt}^{\#}$ returns \top_{Dyn} , hence the thesis trivially holds.

▷ $\text{toString}^{\#}$

We abuse notation denoting by $\text{toString} : \text{VAL} \rightarrow \text{STR}$ the collecting semantics of the implicit string conversion function obtained lifting the definition of toString reported in Figure 3.2, namely

$$\text{toString}(S) \triangleq \{ \text{toString}(v) \mid v \in S \}$$

Hence, in order to prove soundness we need to prove $\forall v \in \text{Dyn}$ the following fact.

$$\text{toString}(\gamma_{\text{Dyn}}(v)) \subseteq \gamma_{\text{Dyn}}(\text{toString}^{\#}(v))$$

We split the proof in the following cases, depending on the type of v .

- $v \in \text{Bool}$

– if $v = \text{true}$:

$$\text{toString}(\{\text{true}\}) = \{\text{true}\} = \gamma_{\text{Dyn}}(\text{Min}(\{\text{true}\})) = \gamma_{\text{Dyn}}(\text{toString}^{\#}(v))$$

– if $v = \text{false}$:

$$\text{toString}(\{\text{false}\}) = \{\text{false}\} = \gamma_{\text{Dyn}}(\text{Min}(\{\text{false}\})) = \gamma_{\text{Dyn}}(\text{toString}^{\#}(v))$$

– if $v = \top_{\text{Bool}}$:

$$\text{toString}(\{\text{false}, \text{true}\}) = \{\text{false}, \text{true}\}$$

$$= \gamma_{\text{Dyn}}(\text{Min}(\{\text{false}, \text{true}\})) = \gamma_{\text{Dyn}}(\text{toString}^\#(v))$$

- $v \in \text{Ints}$: we split the proof in several cases, depending on the value of the input interval v .

- if $v = [i, j]$, with $i, j \in \mathbb{Z}$:

$$\begin{aligned} \text{toString}([i, j]) &= \{ \mathcal{S}(n) \mid n \in [i, j] \} = \bigcup_{n \in [i, j]} \mathcal{L}(\text{Min}(\{\mathcal{S}(n)\})) \\ &= \mathcal{L}(\bigcup_{n \in [i, j]} \text{Min}(\{\mathcal{S}(n)\})) = \gamma_{\text{Dyn}}(\bigcup_{n \in [i, j]} \text{Min}(\mathcal{S}(n))) = \gamma_{\text{Dyn}}(\text{toString}^\#([i, j])) \end{aligned}$$

- if $v = [0, +\infty]$ or $v = [-\infty, 0]$: in the first case, $\text{toString}(v) = \{ \mathcal{S}(i) \mid i \geq 0 \}$, containing only non-negative numerical strings. In this case, $\text{toString}^\#$ returns the automaton A^+ reported in Table 6.4 (second row), recognizing only non-negative numerical strings, hence $\text{toString}(v) = \gamma_{\text{Dyn}}(A^+) = \mathcal{L}(A^+)$. In the second case, $\text{toString}(v) = \{ \mathcal{S}(i) \mid i \leq 0 \}$, containing only non-positive numerical strings. The corresponding abstract semantics $\text{toString}^\#$ returns the automaton A^- reported in Table 6.4 (third row), recognizing only non-positive numerical strings, hence the thesis holds since $\text{toString}(v) = \gamma_{\text{Dyn}}(A^-) = \mathcal{L}(A^-)$.

- if $v = [k, +\infty]$, with $k > 0$:

$$\begin{aligned} \text{toString}(v) &= \{ \mathcal{S}(i) \mid i \geq k \} = \{ \mathcal{S}(i) \mid i \geq 0 \} \setminus \{ \mathcal{S}(0), \dots, \mathcal{S}(k-1) \} \\ &= \mathcal{L}(\text{Min}(A^+)) \setminus \mathcal{L}(\text{Min}(\{\mathcal{S}(0), \dots, \mathcal{S}(k-1)\})) \\ &= \mathcal{L}(\text{toString}^\#([0, +\infty])) \setminus \mathcal{L}(\text{toString}^\#([0, k-1])) \\ &= \mathcal{L}(\text{toString}^\#([0, +\infty]) \setminus_{\text{DFA}} \text{toString}^\#([0, k-1])) \\ &= \gamma_{\text{Dyn}}(\text{toString}^\#([k, +\infty])) \end{aligned}$$

- if $v = [-k, +\infty]$, with $k > 0$:

$$\begin{aligned} \text{toString}(v) &= \{ \mathcal{S}(i) \mid i \geq -k \} = \{ \mathcal{S}(-k), \dots, \mathcal{S}(1) \} \cup \{ \mathcal{S}(i) \mid i \geq 0 \} \\ &= \mathcal{L}(\text{Min}(\{\mathcal{S}(-k), \dots, \mathcal{S}(1)\})) \cup \mathcal{L}(\text{Min}(A^+)) \\ &= \mathcal{L}(\text{toString}^\#([-k, 1])) \cup \mathcal{L}(\text{toString}^\#([0, +\infty])) \\ &= \mathcal{L}(\text{toString}^\#([-k, 1]) \sqcup_{\text{DFA}} \text{toString}^\#([0, +\infty])) \\ &= \gamma_{\text{Dyn}}(\text{toString}^\#([-k, +\infty])) \end{aligned}$$

- $v = [-\infty, k]$ or $v = [-\infty, -k]$, with $k > 0$: the proof is analogous to the previous two cases.

- $v = [-\infty, +\infty]$:

$$\begin{aligned} \text{toString}(v) &= \{ \mathcal{S}(n) \mid n \in \mathbb{Z} \} = \{ n \mid n \leq 0 \} \cup \{ n \mid n \geq 0 \} \\ &= \mathcal{L}(\text{toString}^\#([-\infty, 0]) \sqcup_{\text{DFA}} \text{toString}^\#([0, +\infty])) \\ &= \mathcal{L}(\text{Min}(\Sigma_{\mathbb{Z}})) = \gamma_{\text{Dyn}}(\text{toString}^\#([-\infty, +\infty])) \end{aligned}$$

□

Lemma A.1. Let $A \in \text{DFA}_{/\equiv}$ and $l \in \mathbb{N}$. The following result holds

$$\text{SS}(\mathcal{L}(A), [l, +\infty], [l, +\infty]) = \mathcal{L}(\text{SS}^{\leftrightarrow}(A, l))$$

Proof. In the proof we will use the following result on suffixes. Given the regular language \mathcal{L} the following result holds:

$$\left\{ y \in \Sigma^* \left| \begin{array}{l} \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}) \\ z \in \text{SU}(\mathcal{L}) \end{array} \right. \right\} = \left\{ y \in \Sigma^* \mid \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}) \right\} \quad (\text{A.1})$$

$$\begin{aligned} & \text{SS}(\mathcal{L}(A), [l, +\infty], [l, +\infty]) = \\ & = \text{PSS}(\mathcal{L}(A), [l, +\infty], [l, +\infty]) \cup \text{NSS}(\mathcal{L}(A), [l, +\infty], [l, +\infty]) \quad \text{\textcolor{green}{[Def. 5.14]}} \\ & = \left\{ y \left| \begin{array}{l} \exists x, z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(A), a) \\ z \in \text{SU}(\mathcal{L}(A), b), a, b \in [l, +\infty] \end{array} \right. \right\} \\ & \cup \{ y \mid y \in \text{SU}(\mathcal{L}(A), a) \cap \Sigma^{\leq b-a}, a, b \in [l, +\infty] \} \cup \{\epsilon\} \quad \text{\textcolor{green}{[Defs. 5.6, 5.7]}} \\ & = \left\{ y \left| \begin{array}{l} \exists x, z \in \Sigma^*. yz \in \text{SU}(\text{SU}(\mathcal{L}(A), l)) \\ z \in \text{SU}(\text{SU}(\mathcal{L}(A), l)) \end{array} \right. \right\} \\ & \cup \{ y \mid y \in \text{SU}(\text{SU}(\mathcal{L}(A), l)) \} \cup \{\epsilon\} \quad \text{\textcolor{green}{[Eq. 2.1]}} \\ & = \{ y \mid \exists x, z \in \Sigma^*. yz \in \text{SU}(\text{SU}(\mathcal{L}(A), l)) \} \\ & \cup \{ y \mid y \in \text{SU}(\text{SU}(\mathcal{L}(A), l)) \} \cup \{\epsilon\} \quad \text{\textcolor{green}{[Eq. A.1]}} \\ & = \text{PR}(\text{SU}(\text{SU}(\mathcal{L}(A), l))) \cup \text{SU}(\text{SU}(\mathcal{L}(A), l)) \cup \{\epsilon\} \quad \text{\textcolor{green}{[\mathcal{L} \subseteq \text{PR}(\mathcal{L})]}} \\ & = \text{PR}(\text{SU}(\text{SU}(\mathcal{L}(A), l))) \cup \{\epsilon\} \quad \text{\textcolor{green}{[Def. 2.42]}} \\ & = \text{FA}(\text{SU}(\mathcal{L}(A), l)) \quad \text{\textcolor{green}{[\epsilon \in \text{FA}(\mathcal{L})]}} \\ & = \mathcal{L}(\text{FA}(\text{SU}(A, l))) \quad \text{\textcolor{green}{[Thm. 2.50]}} \\ & = \mathcal{L}(\text{SS}^{\leftrightarrow}(A, l)) \quad \text{\textcolor{green}{[Def. 5.16]}} \end{aligned}$$

□

Theorem 6.3

Proof of Theorem 6.3. Without loss of generality, we suppose that any integer value is positive. Indeed, when a negative value is met, it is treated as 0. For the same reason, when an interval that is fully contained in $[-\infty, -1]$ is met, it is rewritten as the interval $[0, 0]$, and when an interval $[-\infty, i]$, with $i \geq 0$, is met, it is rewritten as the interval $[0, i]$, following the SS semantics (any negative index is treated as 0)

▷ Table 6.1

Second row of Table. 6.1

- $i, j, l, k \in \mathbb{Z}$ (second row, second column): completeness follows from Theorem 5.14.
- $i, j \in \mathbb{Z}, j = -\infty, k \in \mathbb{Z}$ (second row, third column)

$$\text{SS}(\mathcal{L}(A), [i, j], [-\infty, k]) \quad \text{\textcolor{green}{[i \leq l]}}$$

$$\begin{aligned}
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [i, j], [-\infty, -1]) \\
&\cup \text{Ss}(\mathcal{L}(\mathbf{A}), [i, j], [0, k]) \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [i, j], [0, 0]) \cup \text{Ss}(\mathcal{L}(\mathbf{A}), [i, j], [0, k]) && \{\text{Def. of Ss}\} \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [i, j], [0, k]) && \{\cup \text{-left-most} \subseteq \cup \text{-right-most}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, j], [0, k])) && \{2^{\text{sd}}\text{-row}, 2^{\text{sd}}\text{col.}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, j], [-\infty, k])) && \{\text{Def. of SS}^\#\}
\end{aligned}$$

- $i, l \in \mathbb{Z}, j, k = +\infty$, with $i \leq l$: (second row, fourth column)

$$\begin{aligned}
&\text{Ss}(\mathcal{L}(\mathbf{A}), [i, +\infty], [l, +\infty]) \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [i, l], [l, l]) \cup \text{Ss}(\mathcal{L}(\mathbf{A}), [l, +\infty], [l, +\infty]) && \{i \leq l\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, l], [l, l])) && \{2^{\text{sd}}\text{-row}, 2^{\text{sd}}\text{col.}\} \\
&\sqcup_{\text{DFA}} \text{SS}^{\leftrightarrow}(\mathbf{A}, l) && \{\text{Lemma A.1}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, +\infty], [l, +\infty]))
\end{aligned}$$

- $i, j \in \mathbb{Z}, l = -\infty, k = -\infty$ (second row, fifth column)

$$\begin{aligned}
&\text{Ss}(\mathcal{L}(\mathbf{A}), [i, j], [-\infty, +\infty]) \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [i, j], [-\infty, -1]) \\
&\cup \text{Ss}(\mathcal{L}(\mathbf{A}), [i, j], [0, +\infty]) && \{\text{Def. of Ss}\} \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [i, j], [0, +\infty]) && \{\cup \text{-left-most} \subseteq \cup \text{-right-most}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, j], [0, +\infty])) && \{2^{\text{sd}}\text{-row}, 3^{\text{rd}}\text{col.}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, j], [-\infty, +\infty])) && \{\text{Def. of SS}^\#\}
\end{aligned}$$

Third row of Table. 6.1

- $i = -\infty, j \in \mathbb{Z}, l, k \in \mathbb{Z}$ (third row, second column)

$$\begin{aligned}
&\text{Ss}(\mathcal{L}(\mathbf{A}), [-\infty, j], [l, k]) \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [0, j], [l, k]) && \{\text{Def. of Ss}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [0, j], [l, k])) && \{2^{\text{sd}}\text{-row}, 2^{\text{nd}}\text{col.}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [-\infty, j], [l, k])) && \{\text{Def. of SS}^\#\}
\end{aligned}$$

- $i = -\infty, j \in \mathbb{Z}, l = -\infty, k \in \mathbb{Z}$ (third row, third column)

$$\begin{aligned}
&\text{Ss}(\mathcal{L}(\mathbf{A}), [-\infty, j], [-\infty, k]) \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [0, j], [0, k]) && \{\text{Def. of Ss}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [0, j], [0, k])) && \{2^{\text{sd}}\text{-row}, 2^{\text{nd}}\text{col.}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [-\infty, j], [-\infty, k])) && \{\text{Def. of SS}^\#\}
\end{aligned}$$

- $i = -\infty, j \in \mathbb{Z}, l \in \mathbb{Z}, k = +\infty$ (third row, fourth column)

$$\begin{aligned}
& \text{Ss}(\mathcal{L}(\mathbf{A}), [-\infty, j], [l, +\infty]) \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [0, j], [l, +\infty]) \quad \{\text{Def. of Ss}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [0, j], [0, k])) \quad \{2^{\text{sd}}\text{-row}, 3^{\text{rd}}\text{col.}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [-\infty, j], [l, +\infty])) \quad \{\text{Def. of SS}^\#\}
\end{aligned}$$

- $i = -\infty, j \in \mathbb{N}, l = -\infty, k = +\infty$ (third row, fifth column)

$$\begin{aligned}
& \text{Ss}(\mathcal{L}(\mathbf{A}), [-\infty, j], [-\infty, +\infty]) \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [0, j], [0, +\infty]) \quad \{\text{Def. of Ss}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [0, j], [0, +\infty])) \quad \{2^{\text{sd}}\text{-row}, 3^{\text{rd}}\text{col.}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [-\infty, j], [-\infty, +\infty])) \quad \{\text{Def. of SS}^\#\}
\end{aligned}$$

Forth row of Table. 6.1

- $i \in \mathbb{Z}, j = +\infty, l, k \in \mathbb{Z}$ (forth row, second column)

$$\begin{aligned}
& \text{Ss}(\mathcal{L}(\mathbf{A}), [i, +\infty], [l, k]) \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [i, l], [l, k]) \cup \text{Ss}(\mathcal{L}(\mathbf{A}), [l, +\infty], [l, k]) \quad \{i \leq l \leq k\} \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [i, l], [l, k]) \cup \text{Ss}(\mathcal{L}(\mathbf{A}), [l, k], [l, +\infty]) \quad \{\text{Def. of Ss (SWAPS)}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, l], [l, k]) \sqcup_{\text{DEFA}} \text{SS}^\#(\mathbf{A}, [l, k], [l, +\infty])) \quad \{2^{\text{sd}}\text{-row}, 2^{\text{nd}} \text{ and } 3^{\text{rd}}\text{cols.}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, +\infty], [l, k])) \quad \{\text{Def. of SS}^\#\}
\end{aligned}$$

- $i \in \mathbb{Z}, j = +\infty, l = -\infty, k \in \mathbb{Z}$ (forth row, third column)

$$\begin{aligned}
& \text{Ss}(\mathcal{L}(\mathbf{A}), [i, +\infty], [-\infty, k]) \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [i, +\infty], [0, k]) \quad \{\text{Def. of Ss}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, +\infty], [0, k])) \quad \{3^{\text{rd}}\text{-row}, 2^{\text{nd}}\text{-col.}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, +\infty], [-\infty, k])) \quad \{\text{Def. of SS}^\#\}
\end{aligned}$$

- $i \in \mathbb{Z}, j = +\infty, l \in \mathbb{Z}, k = +\infty$ (forth row, forth column)

$$\begin{aligned}
& \text{Ss}(\mathcal{L}(\mathbf{A}), [i, +\infty], [l, +\infty]) \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [i, l], [l, +\infty]) \cup \text{Ss}(\mathcal{L}(\mathbf{A}), [l, +\infty], [l, +\infty]) \quad \{\text{Def. of Ss}, i \leq l\} \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [i, l], [l, +\infty]) \cup \mathcal{L}(\text{SS}^{\leftrightarrow}(\mathcal{L}(\mathbf{A}), l)) \quad \{\text{Lemma A.1}\} \\
&= \text{Ss}^{\rightarrow}(\mathcal{L}(\mathbf{A}), [i, l], l) \cup \mathcal{L}(\text{SS}^{\leftrightarrow}(\mathcal{L}(\mathbf{A}), l)) \quad \{\text{Def. 5.18}\} \\
&= \mathcal{L}(\text{SS}^{\rightarrow}(\mathbf{A}, [i, l], l) \sqcup_{\text{DEFA}} \text{SS}^{\leftrightarrow}(\mathbf{A}, l)) \quad \{\text{Thm. 5.17, 5.20}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, +\infty], [l, +\infty])) \quad \{\text{Def. of SS}^\#\}
\end{aligned}$$

- $i \in \mathbb{Z}, j = +\infty, l = -\infty, k = +\infty$ (forth row, forth column)

$$\begin{aligned}
& \text{Ss}(\mathcal{L}(\mathbf{A}), [i, +\infty], [-\infty, +\infty]) \\
&= \text{Ss}(\mathcal{L}(\mathbf{A}), [i, +\infty], [0, +\infty]) \quad \{\text{Def. of Ss}\}
\end{aligned}$$

$$\begin{aligned}
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, +\infty], [0, +\infty])) && \{4^{\text{th}}\text{-row}, 4^{\text{th}}\text{-col.}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, +\infty], [-\infty, +\infty])) && \{\text{Def. of SS}^\#\}
\end{aligned}$$

Fifth row of Table 6.1

- $i = -\infty, j = +\infty, l, k \in \mathbb{Z}$ (fifth row, second column)

$$\begin{aligned}
&\text{SS}(\mathcal{L}(\mathbf{A}), [-\infty, +\infty], [l, k]) \\
&= \text{SS}(\mathcal{L}(\mathbf{A}), [0, +\infty], [l, k]) && \{\text{Def. of SS}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [0, +\infty], [l, k])) && \{4^{\text{th}}\text{-row}, 2^{\text{nd}}\text{col.}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [-\infty, +\infty], [l, k])) && \{\text{Def. of SS}^\#\}
\end{aligned}$$

- $i = -\infty, j = +\infty, l = -\infty, k \in \mathbb{Z}$ (fifth row, third column)

$$\begin{aligned}
&\text{SS}(\mathcal{L}(\mathbf{A}), [-\infty, +\infty], [-\infty, k]) \\
&= \text{SS}(\mathcal{L}(\mathbf{A}), [0, +\infty], [0, k]) && \{\text{Def. of SS}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [0, +\infty], [0, k])) && \{4^{\text{th}}\text{-row}, 2^{\text{nd}}\text{col.}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [-\infty, +\infty], [-\infty, k])) && \{\text{Def. of SS}^\#\}
\end{aligned}$$

- $i = -\infty, j = +\infty, l \in \mathbb{Z}, k = +\infty$ (fifth row, fourth column)

$$\begin{aligned}
&\text{SS}(\mathcal{L}(\mathbf{A}), [-\infty, +\infty], [l, +\infty]) \\
&= \text{SS}(\mathcal{L}(\mathbf{A}), [0, +\infty], [l, +\infty]) && \{\text{Def. of SS}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [0, +\infty], [0, k])) && \{4^{\text{th}}\text{-row}, 4^{\text{th}}\text{col.}\} \\
&= \mathcal{L}(\text{SS}^\#(\mathbf{A}, [-\infty, +\infty], [l, +\infty])) && \{\text{Def. of SS}^\#\}
\end{aligned}$$

- $i = -\infty, j = +\infty, l = -\infty, k = +\infty$ (fifth row, fifth column)

$$\begin{aligned}
&\text{SS}(\mathcal{L}(\mathbf{A}), [-\infty, +\infty], [-\infty, +\infty]) \\
&= \text{SS}(\mathcal{L}(\mathbf{A}), [0, +\infty], [0, +\infty]) && \{\text{Def. of SS}\} \\
&= \left\{ y \mid \begin{array}{l} \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(\mathbf{A}), i) \\ z \in \text{SU}(\mathcal{L}(\mathbf{A}), j), i, j \in \mathbb{N}, i \leq j \end{array} \right\} \\
&\cup \{ y \mid y \in \text{SU}(\mathcal{L}(\mathbf{A}), i) \cap \Sigma^{\leq j-i}, i, j \in \mathbb{N}, i \leq j \} \\
&\cup \{ \epsilon \mid \mathcal{L}(\mathbf{A}) \cap \Sigma^{\leq i} \neq \emptyset, i \in \mathbb{N} \} && \{i \in \mathbb{N}\} \\
&= \left\{ y \mid \begin{array}{l} \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(\mathbf{A}), i) \\ z \in \text{SU}(\mathcal{L}(\mathbf{A}), j), i, j \in \mathbb{N}, i \leq j \end{array} \right\} \\
&\cup \left\{ y \mid \begin{array}{l} y \in \text{SU}(\mathcal{L}(\mathbf{A}), i) \cap \Sigma^{\leq j-i} \\ i, j \in \mathbb{N}, i \leq j \end{array} \right\} \\
&\cup \{ \epsilon \} && \{ \epsilon \in \text{SU}(\mathcal{L}(\mathbf{A}), i) \cap \Sigma^{\leq j-i} \} \\
&= \left\{ y \mid \begin{array}{l} \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(\mathbf{A}), i) \\ z \in \text{SU}(\mathcal{L}(\mathbf{A}), j), i, j \in \mathbb{N}, i \leq j \end{array} \right\}
\end{aligned}$$

$$\begin{aligned}
& \cup \left\{ y \mid \begin{array}{l} y \in \text{SU}(\mathcal{L}(\mathbf{A}), i) \cap \Sigma^{\leq j-i} \\ i, j \in \mathbb{N}, i \leq j \end{array} \right\} && \{i, j \in \mathbb{N}\} \\
& = \left\{ y \mid \begin{array}{l} \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(\mathbf{A}), i) \\ z \in \text{SU}(\mathcal{L}(\mathbf{A}), j), i, j \in \mathbb{N}, i \leq j \end{array} \right\} \cup \text{SU}(\mathcal{L}(\mathbf{A})) \\
& = \left\{ y \mid \begin{array}{l} \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(\mathbf{A}), i) \\ z \in \text{SU}(\mathcal{L}(\mathbf{A}), j) \\ i, j \in \mathbb{N}, i \leq j \end{array} \right\} && \{\cup\text{-left} \supseteq \cup\text{-right}\} \\
& = \left\{ y \mid \begin{array}{l} \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(\mathbf{A})) \\ z \in \text{SU}(\mathcal{L}(\mathbf{A})) \end{array} \right\} && \{\text{Def. of SU}\} \\
& = \left\{ y \mid \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(\mathbf{A})) \right\} && \{\text{Eq. A.1}\} \\
& = \left\{ y \mid \exists x, z \in \Sigma^*. xyz \in \mathcal{L}(\mathbf{A}) \right\} \\
& = \text{FA}(\mathcal{L}(\mathbf{A})) && \{\text{Def. of FA}\} \\
& = \mathcal{L}(\text{FA}(\mathbf{A})) && \{\text{Thm. 2.50}\} \\
& = \mathcal{L}(\text{SS}^\#(\mathbf{A}, [-\infty, +\infty], [-\infty, +\infty])) && \{\text{Def. of SS}^\#\}
\end{aligned}$$

▷ Table 6.2

In Table 6.2, the proofs concerning the cases compatible with Table 6.1 (i.e., cells containing Table 6.1) are identical to the ones reported above. Here we proof the remaining cases.

Second row of Table. 6.2

- $i, j \in \mathbb{Z}, l = +\infty, k = +\infty$ (second row, forth column)

$$\begin{aligned}
& \text{SS}(\mathcal{L}(\mathbf{A}), [i, j], [l, +\infty]) \\
& = \text{SS}(\mathcal{L}(\mathbf{A}), [i, l], [i, j]) \\
& \cup \text{SS}(\mathcal{L}(\mathbf{A}), [i, j], [j, +\infty]) && \{l < i \leq k\} \\
& = \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, l], [i, j])) \\
& \sqcup_{\text{DFA}} \text{SS}^\#(\mathbf{A}, [i, j], [j, +\infty]) && \{\text{Tab.6.1}\} \\
& = \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, j], [l, +\infty])) && \{\text{Def. of SS}^\#\}
\end{aligned}$$

Forth row of Table. 6.2

- $i \in \mathbb{Z}, j = +\infty, l, k \in \mathbb{Z}$ (forth row, second column)

$$\begin{aligned}
& \text{SS}(\mathcal{L}(\mathbf{A}), [i, +\infty], [l, k]) \\
& = \text{SS}(\mathcal{L}(\mathbf{A}), [i, l], [i, k]) \\
& \cup \text{SS}(\mathcal{L}(\mathbf{A}), [l, i], [l, +\infty]) \\
& \cup \text{SS}(\mathcal{L}(\mathbf{A}), [i, k], [k, +\infty]) && \{l < i \leq k\} \\
& = \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, l], [i, k])) \\
& \sqcup_{\text{DFA}} \text{SS}^\#(\mathbf{A}, [l, i], [l, +\infty])
\end{aligned}$$

$$\begin{aligned} & \sqcup_{\text{DFA}} \text{SS}^\#(\mathbf{A}, [i, k], [k, +\infty]) && \{ \text{Compl. of Tab. 6.1} \} \\ & = \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, +\infty], [l, k])) && \{ \text{Def. of SS}^\# \} \end{aligned}$$

- $i \in \mathbb{N}, j = +\infty, l \in \text{ints}, k = +\infty$ (forth row, forth column)

$$\begin{aligned} & \text{Ss}(\mathcal{L}(\mathbf{A}), [i, +\infty], [l, +\infty]) \\ & = \text{Ss}(\mathcal{L}(\mathbf{A}), [l, i], [i, i]) \\ & \cup \text{Ss}(\mathcal{L}(\mathbf{A}), [i, +\infty], [i, +\infty]) && \{ i > l \} \\ & = \text{Ss}(\mathcal{L}(\mathbf{A}), [l, +\infty], [i, +\infty]) \\ & = \mathcal{L}(\text{SS}^\#(\mathbf{A}, [l, +\infty], [i, +\infty])) && \{ \text{Tab. 6.1} \} \\ & = \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, +\infty], [l, +\infty])) && \{ \text{Def. of SS}^\# \} \end{aligned}$$

▷ Table 6.3

In Table 6.3, the proofs concerning the cases compatible with Table 6.1 (i.e., cells containing Table 6.1) are identical to the ones reported above. Here we proof the remaining cases.

Second row of Table. 6.3

- $i, j \in \mathbb{Z}, l = +\infty, k = +\infty$ (second row, forth column)

$$\begin{aligned} & \text{Ss}(\mathcal{L}(\mathbf{A}), [i, j], [l, +\infty]) \\ & = \text{Ss}(\mathcal{L}(\mathbf{A}), [l, j], [l, j]) \\ & \cup \text{Ss}(\mathcal{L}(\mathbf{A}), [i, l], [l, +\infty]) \\ & \cup \text{Ss}(\mathcal{L}(\mathbf{A}), [l, j], [j, +\infty]) && \{ i \leq j \wedge j \geq l \} \\ & = \mathcal{L}(\text{SS}^\#(\mathbf{A}, [l, j], [l, j])) \\ & \sqcup_{\text{DFA}} \text{SS}^\#(\mathbf{A}, [i, l], [l, +\infty]) \\ & \sqcup_{\text{DFA}} \text{SS}^\#(\mathbf{A}, [l, j], [j, +\infty]) && \{ \text{Tab. 6.1} \} \\ & = \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, j], [l, +\infty])) && \{ \text{Def. of SS}^\# \} \end{aligned}$$

Forth row of Table. 6.3

- $i \in \mathbb{Z}, j = +\infty, l, k \in \mathbb{Z}$ (forth row, second column)

$$\begin{aligned} & \text{Ss}(\mathcal{L}(\mathbf{A}), [i, +\infty], [l, k]) \\ & = \text{Ss}(\mathcal{L}(\mathbf{A}), [l, k], [i, +\infty]) && \{ \text{Def. of Ss}, i > k \} \\ & = \mathcal{L}(\text{SS}^\#(\mathbf{A}, [l, k], [i, +\infty])) && \{ \text{Tab. 6.1} \} \\ & = \mathcal{L}(\text{SS}^\#(\mathbf{A}, [l, k], [i, +\infty])) && \{ \text{Def. of SS}^\# \} \end{aligned}$$

- $i \in \mathbb{N}, j = +\infty, l \in \text{ints}, k = +\infty$ (forth row, forth column)

$$\begin{aligned} & \text{Ss}(\mathcal{L}(\mathbf{A}), [i, +\infty], [l, +\infty]) \\ & = \text{Ss}(\mathcal{L}(\mathbf{A}), [l, i], [i, i]) \\ & \cup \text{Ss}(\mathcal{L}(\mathbf{A}), [i, +\infty], [i, +\infty]) && \{ i > l \} \\ & = \text{Ss}(\mathcal{L}(\mathbf{A}), [l, +\infty], [i, +\infty]) && \{ \text{Homom. of Ss} \} \\ & = \mathcal{L}(\text{SS}^\#(\mathbf{A}, [l, +\infty], [i, +\infty])) && \{ \text{Tab. 6.1} \} \\ & = \mathcal{L}(\text{SS}^\#(\mathbf{A}, [i, +\infty], [l, +\infty])) && \{ \text{Def. of SS}^\# \} \end{aligned}$$

□

Theorem 6.4

Proof of Theorem 6.4. We proof soundness and completeness of $\text{CA}^\#$ splitting the proof based on the values of the input interval $[i, j]$. We observe that if $[i, j] \sqsubseteq_{\text{Ints}} [-\infty, -1]$, then

$$\text{CA}(\mathcal{L}, [i, j]) = \{\epsilon\} \quad (\text{A.2})$$

Moreover, it easy to see that $\forall \mathcal{L} \in \wp(\Sigma), i, j \in \mathbb{Z}. \text{CA}(\mathcal{L}, [i, j]) \subseteq \Sigma^{\leq 1}$, that is we can obtain either strings of length 1 or the empty string.

- $i, j \in \mathbb{Z}$:

$$\begin{aligned} & \text{CA}(\mathcal{L}(\mathbf{A}), [i, j]) \\ &= \{ \text{SS}(\sigma, k, k+1) \mid \sigma \in \mathcal{L}(\mathbf{A}), k \in [i, j] \} && \text{\textit{[Def. of CA]}} \\ &= \bigcup_{k \in [i, j]} \{ \{ \text{SS}(\sigma, k, k+1) \} \mid \sigma \in \mathcal{L}(\mathbf{A}) \} \\ &= \bigcup_{k \in [i, j]} \mathcal{L}(\text{SS}^\#(\mathbf{A}, [k, k], [k+1, k+1])) && \text{\textit{[Thm. 6.3]}} \\ &= \mathcal{L}(\bigsqcup_{k \in [i, j]} \text{SS}(\mathbf{A}, [k, k], [k+1, k+1])) && \text{\textit{[Def. of } \sqcup_{\text{DFA}} \text{]}} \\ &= \mathcal{L}(\text{CA}^\#(\mathbf{A}, [i, j])) && \text{\textit{[Def. of CA}^\# \text{]}} \end{aligned}$$

- $i = -\infty, j \in \mathbb{Z}, j \geq 0$:

$$\begin{aligned} & \text{CA}(\mathcal{L}(\mathbf{A}), [-\infty, j]) \\ &= \text{CA}(\mathcal{L}(\mathbf{A}), [-\infty, -1]) \cup \text{CA}(\mathcal{L}(\mathbf{A}), [0, j]) \\ &= \{\epsilon\} \cup \text{CA}(\mathcal{L}(\mathbf{A}), [0, j]) && \text{\textit{[Eq. A.2]}} \\ &= \{\epsilon\} \cup \mathcal{L}(\text{CA}^\#(\mathbf{A}, [0, j])) && \text{\textit{[}i, j \in \mathbb{Z} \text{ case]}} \\ &= \mathcal{L}(\text{Min}(\{\epsilon\}) \sqcup_{\text{DFA}} \text{CA}^\#(\mathbf{A}, [0, j])) && \text{\textit{[Def. of } \sqcup_{\text{DFA}} \text{]}} \\ &= \mathcal{L}(\text{CA}^\#(\mathbf{A}, [-\infty, j])) && \text{\textit{[Def. of CA}^\# \text{]}} \end{aligned}$$

- $i = -\infty, j \in \mathbb{Z}, j < 0$:

$$\begin{aligned} & \text{CA}(\mathcal{L}(\mathbf{A}), [-\infty, j]) \\ &= \{\epsilon\} && \text{\textit{[Eq. A.2]}} \\ &= \mathcal{L}(\text{Min}(\{\epsilon\})) \\ &= \mathcal{L}(\text{CA}^\#(\mathbf{A}, [-\infty, j])) && \text{\textit{[Def. of CA}^\# \text{]}} \end{aligned}$$

- $i \in \mathbb{Z}, i \geq 0, j = +\infty$:

$$\begin{aligned} & \text{CA}(\mathcal{L}(\mathbf{A}), [i, +\infty]) = \\ &= \{\epsilon\} \cup \left\{ y \mid \begin{array}{l} \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(\mathbf{A}), k) \\ z \in \text{SU}(\mathcal{L}(\mathbf{A}), k+1), k \in [i, +\infty], |y| = 1 \end{array} \right\} && \text{\textit{[Def. of SS, } |y| \leq 1 \text{]}} \\ &= \{\epsilon\} \cup \left\{ y \mid \begin{array}{l} \exists z \in \Sigma^*. yz \in \text{SU}(\mathcal{L}(\mathbf{A}), k) \\ z \in \text{SU}(\mathcal{L}(\mathbf{A}), k+1), k \geq i, |y| = 1 \end{array} \right\} \end{aligned}$$

$$\begin{aligned}
&= \{\epsilon\} \cup \left\{ y \left| \begin{array}{l} \exists x, z \in \Sigma^*. \\ yz \in \text{SU}(\text{SU}(\mathcal{L}(\mathbf{A}), i)) \\ z \in \text{SU}(\text{SU}(\mathcal{L}(\mathbf{A}), i+1)), |y| = 1 \end{array} \right. \right\} && \text{\textcircled{Def. 2.45}} \\
&= \{\epsilon\} \cup \left\{ y \left| \begin{array}{l} \exists x, z \in \Sigma^*. \\ yz \in \text{SU}(\text{SU}(\mathcal{L}(\mathbf{A}), i)) \end{array} \right. \right\} \cap \Sigma^1 && \text{\textcircled{|y| = 1}} \\
&= \{\epsilon\} \cup \text{PR}(\text{SU}(\text{SU}(\mathcal{L}(\mathbf{A}), i))) \cap \Sigma^1 && \text{\textcircled{Def. 2.42}} \\
&= \text{PR}(\text{SU}(\text{SU}(\mathcal{L}(\mathbf{A}), i))) \cap \Sigma^{\leq 1} && \text{\textcircled{\epsilon \in PR}} \\
&= \mathcal{L}(\text{PR}(\text{SU}(\text{SU}(\mathbf{A}, i))) \sqcap_{\text{DFA}} \text{Min}(\Sigma^{\leq 1})) && \text{\textcircled{Thm. 2.50}} \\
&= \mathcal{L}(\text{FA}(\text{SU}(\mathbf{A}, i)) \sqcap_{\text{DFA}} \text{Min}(\Sigma^{\leq 1})) && \text{\textcircled{Def. of FA}} \\
&= \mathcal{L}(\text{CA}^\#(\mathbf{A}, [i, +\infty])) && \text{\textcircled{Def. of CA}^\#}
\end{aligned}$$

- $i = -\infty$ or $i \in \mathbb{Z}, i < 0, j = +\infty$: let consider the case when $i = -\infty$. The remaining case (i.e., $i < 0$) is identical.

$$\begin{aligned}
&\text{CA}(\mathcal{L}(\mathbf{A}), [-\infty, +\infty]) \\
&= \text{CA}(\mathcal{L}(\mathbf{A}), [-\infty, -1]) \cup \text{CA}(\mathcal{L}(\mathbf{A}), [0, +\infty]) \\
&= \{\epsilon\} \cup \text{CA}(\mathcal{L}(\mathbf{A}), [0, +\infty]) && \text{\textcircled{Eq. A.2}} \\
&= \{\epsilon\} \cup \left\{ y \left| \begin{array}{l} \exists z \in \Sigma^*. |y| = 1 \\ yz \in \text{SU}(\mathcal{L}(\mathbf{A}), k) \\ z \in \text{SU}(\mathcal{L}(\mathbf{A}), k+1) \\ k \in [0, +\infty] \end{array} \right. \right\} && \text{\textcircled{Def. of Ss}} \\
&= \{\epsilon\} \cup \left\{ y \left| \begin{array}{l} \exists z \in \Sigma^*. |y| = 1 \\ yz \in \text{SU}(\text{SU}(\mathcal{L}(\mathbf{A}), 0)) \\ z \in \text{SU}(\text{SU}(\mathcal{L}(\mathbf{A}), 1)) \end{array} \right. \right\} && \text{\textcircled{|y| = 1, Eq. 2.1}} \\
&= \{\epsilon\} \cup \left\{ y \left| \begin{array}{l} \exists z \in \Sigma^*. |y| = 1 \\ yz \in \text{SU}(\mathcal{L}(\mathbf{A})), z \in \text{SU}(\text{SU}(\mathcal{L}(\mathbf{A}), 1)) \end{array} \right. \right\} && \text{\textcircled{Eq. 2.1}} \\
&= \{\epsilon\} \cup \left\{ y \left| \begin{array}{l} \exists z \in \Sigma^*. |y| = 1 \\ yz \in \text{SU}(\mathcal{L}(\mathbf{A})) \end{array} \right. \right\} && \text{\textcircled{|y| = 1}} \\
&= \{\epsilon\} \cup \left\{ y \left| \begin{array}{l} \exists z \in \Sigma^*. \\ yz \in \text{SU}(\mathcal{L}(\mathbf{A})) \end{array} \right. \right\} \cap \Sigma^1 && \text{\textcircled{Def. of PR}} \\
&= \{\epsilon\} \cup \text{PR}(\text{SU}(\mathcal{L}(\mathbf{A}))) \cap \Sigma^1 \\
&= \{\epsilon\} \cup \text{FA}(\mathcal{L}(\mathbf{A})) \cap \Sigma^1 && \text{\textcircled{Def. of FA}} \\
&= \text{FA}(\mathcal{L}(\mathbf{A})) \cap \Sigma^{\leq 1} && \text{\textcircled{\epsilon \in FA(\mathcal{L})}} \\
&= \mathcal{L}(\text{FA}(\mathbf{A}) \sqcap_{\text{DFA}} \text{Min}(\Sigma^{\leq 1})) && \text{\textcircled{Thm. 2.50}} \\
&= \mathcal{L}(\text{CA}^\#(\mathbf{A}, [-\infty, +\infty])) && \text{\textcircled{Def. of CA}^\#}
\end{aligned}$$

□

Theorem 6.5

Proof of Theorem 6.5. $\text{LE}^\#$ is not complete, i.e. $\text{LE}^\#(\mathbf{A}) \not\subseteq \text{LE}(\mathcal{L}(\mathbf{A}))$. As a counterexample, consider the automaton \mathbf{A}_2 in Figure 6.2b.

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

As far as soundness is concerned, we argue $\forall \sigma \in \mathcal{L}(\mathbf{A}). |\sigma| \in \text{LE}^\#(\mathbf{A})$. Let consider the following cases:

A has cycle : let σ be the minimum string accepted by \mathbf{A} . Its length is computed by searching for the minimum path from the initial to final states (lines 4-8). Since \mathbf{A} has cycles, Algorithm 10 returns the intervals $[|\sigma|, +\infty]$ (line 9), hence any string length greater than $|\sigma|$ is contained in the resulting interval, since it is positive unbounded.

A has not cycle : let σ be the minimum string accepted by \mathbf{A} . Its length is contained in the resulting interval as explained in the previous case. Let σ' be the maximum string accepted by \mathbf{A} . Algorithm 10 searches for the maximum length path from the initial to final states (lines 11-20). The resulting interval is $[|\sigma|, |\sigma'|]$ hence the maximum length string is contained in it. Any other string length trivially belongs to the resulting interval, since the resulting interval goes from minimum string length and maximum string length.

□

Lemma 6.10

Proof of Lemma 6.10. The proof is done by structural induction on the structure of μJS (for the sake of readability we avoid program point labels, since they are not relevant in the proof). Note that, by definition, $\text{P} = c;$, and all the statements composing c are separated by $;$, hence by trivial induction on c we can prove that any statement generated by the grammar μJS is in c followed by a $;$. Hence, in the following in the base of the structural induction we consider also $c;$ (we need also c for the induction in the while and if body). We make the proof by structural induction on c .

Base cases:

- **skip and skip;**
Since both $\text{skip}, \text{skip}; \in \Sigma_{\text{pStm}}$, by definition and since \mathfrak{S} on sequences already in Σ^* is the identity, the thesis trivially hold.
- $x = e$ and $x = e$, with $x \in \text{ID}$ and $e \in \text{E}$.
Let us prove by cases on e
 - If e does not contain punctuation symbols in Punct , both $x = e, x = e; \in \Sigma_{\text{pStm}}$ and therefore, as in the previous case, we have the thesis.
 - If e contains at least one punctuation symbol in Punct .
Let $\mathfrak{S}(e) = \sigma^e$ the string counterpart of the expression e , such that $|\sigma^e| = n$. Let $k \in \mathbb{N} \setminus \{0\}$ the number of punctuation symbols that occurs in σ^e

and $\{p_i\}_{i \in [0, k-1]}$ the position in the string σ^e where a punctuation symbols occurs. A representation of the string in Σ_{pStm} .

$$x = \underbrace{\sigma_0^e \dots \sigma_{p_0}^e}_{\in \Sigma_{\text{pStm}}} \underbrace{\dots \sigma_{p_1}^e}_{\in \Sigma_{\text{pStm}}} \dots \underbrace{\dots \sigma_{p_k}^e}_{\in \Sigma_{\text{pStm}}} \underbrace{\dots \sigma_n^e}_{\in \Sigma_{\text{pStm}}}$$

Since p_0 is the first position where a punctuation symbol occurs in $\mathfrak{S}(x = e)$, the prefix up to $\sigma_{p_0}^e$ form a partial statement. The following sub-strings, i.e., all the sub-strings between p_i and $p_{i+1}, i > 0$, form partial statements, since between p_i and p_{i+1} , in σ^e , does not occur any punctuation symbol by construction. The substring of σ^e after $p_k + 1$ (i.e., the next character after the last punctuation symbol of σ^e if present) is a partial statement, since it does not contain other punctuation symbols always by construction.

- $\text{eval}(s)$ and $\text{eval}(s);$ with $s \in \text{SE}$.

Let us prove by cases on s

- s does not contain punctuation symbols. In this case, $\text{eval}(s) = \mathfrak{S}(\sigma_0 \sigma_1)$ where $\{\sigma_i\}_{i \in [0, 1]} \subseteq \Sigma_{\text{pStm}}$ are $\sigma_0 = \text{eval}()$ (and $\sigma_1 = s$). Instead, $\text{eval}(s); = \mathfrak{S}(\sigma_0 \sigma_1 \sigma_2)$ where $\{\sigma_i\}_{i \in [0, 2]} \subseteq \Sigma_{\text{pStm}}$ are $\sigma_0 = \text{eval}()$, $\sigma_1 = s$ and $\sigma_2 = ;$.
- s contains at least a punctuation symbol. In this case, s is split similarly to e in the proof of the assignment case.

Inductive step:

- $\text{st}_1; \text{st}_2$, with $\text{st}_1, \text{st}_2 \in \text{STMT}$ and st_1 not ending with ;

For inductive hypothesis, st_1 and st_2 can be written as concatenation of partial statements, i.e., $\exists k_1, k_2 \in \mathbb{N} \setminus \{0\}. \{\sigma_i\}_{i \in [0, k_1]} \subseteq \Sigma_{\text{pStm}}, \{\delta_i\}_{i \in [0, k_2]} \subseteq \Sigma_{\text{pStm}}. \text{st}_1 = \mathfrak{S}(\sigma_0 \dots \sigma_{k_1}), \text{st}_2 = \mathfrak{S}(\delta_0 \dots \delta_{k_2})$. Moreover, by definition $; \in \Sigma_{\text{pStm}}$ hence, we are able to rewrite the statement $\text{st}_1; \text{st}_2$ with the partial statements $\{\sigma_i\}_{i \in [0, k_1]}$ of st_1 , $;$ and $\{\delta_i\}_{i \in [0, k_2]}$ of st_2 .

$$\underbrace{\sigma_0 \dots \sigma'_{k_1}}_{\in \Sigma_{\text{pStm}}^*} \underbrace{\text{st}_1;}_{\in \Sigma_{\text{pStm}}} \underbrace{\delta_0 \dots \delta_{k_2}}_{\in \Sigma_{\text{pStm}}^*} \text{st}_2$$

where $\sigma'_k \triangleq \sigma_k$; which is a partial statement since, by hypothesis st_1 was not ending with a semi-colon.

- $\text{while}(e)\{\text{st}\}$ and $\text{while}(e)\{\text{st}\};$ with $e \in \text{E}$, $\text{st} \in \text{STMT}$ not ending with ;

Let us prove by cases on e :

- e does not contain punctuation symbols. By inductive hypothesis, the statement st can be rewritten as sequence of partial statements i.e., $\exists k \in \mathbb{N} \setminus \{0\}. \{\sigma_i\}_{i \in [0, k]} \subseteq \Sigma_{\text{pStm}}. \text{st} = \mathfrak{S}(\sigma_0 \dots \sigma_k)$. Hence, we can compose the while statement in the following way (consider the case ending with $;$, the other one is similar without the last partial statement).

$$\underbrace{\text{while}(e)}_{\in \Sigma_{\text{pStm}}} \underbrace{\{e\}}_{\in \Sigma_{\text{pStm}}} \underbrace{\{\sigma_0 \dots \sigma'_k\}}_{\in \Sigma_{\text{pStm}}^*} \text{st} \underbrace{\{;\}}_{\in \Sigma_{\text{pStm}}}$$

where $\sigma'_k \triangleq \sigma_k$ which is a partial statement since by hypothesis st does not end with a semi-colon.

- e contains at least a punctuation symbol. In this case, e is split similarly to the proof of the assignment.
- $\text{if}(e)\{\text{st}_1\}\text{else}\{\text{st}_2\}$ and $\text{if}(e)\{\text{st}_1\}\text{else}\{\text{st}_2\}$; $e \in E$, $\text{st}_1, \text{st}_2 \in \text{STMT}$ and st_1, st_2 not ending with a semicolon. The proof is similar to the `while` statement case.

□

Lemma 6.11

Proof of Lemma 6.11. We generalize the above lemma on the language recognized on a generic state $q \in Q$, namely

$$\forall \sigma \in \Sigma_{\text{pStm}}^*, \exists q \in Q. \mathfrak{S}(\sigma) \in \mathcal{L}_q(A) \Rightarrow \sigma \in \mathcal{L}_q(A^{\text{pStm}})$$

It is worth noting that the transformation from A to A^{pStm} is performed by the procedure `Build` (Algorithm 13), computing, given $q \in Q$, the set I_q of pairs (partial statement, reached state), namely the partial statements recognized from q and the corresponding reached state. The procedure `StmSyn` (Algorithm 12) does not affect the set I_q , since it simply builds the desired automaton adding the partial statements and the corresponding reached states to A^{pStm} (lines 6-8). In particular, the procedure `Build` is recursively called on these reached states. Once a generic couple (σ, q') is added to I_q , the corresponding transition (q, σ, q') is added to A^{pStm} . Hence, it is clear that $\exists q' \in Q. (\sigma, q') \in I_q \Leftrightarrow \sigma \in \mathcal{L}_q(A^{\text{pStm}})$, for some $q' \in Q^A$. For this reason, in order to prove the above lemma, we focus on showing, given $q \in Q$, that $\exists q' \in Q. \mathfrak{S}(\sigma) \in \mathcal{L}_q(A)$ then $(\mathfrak{S}(\sigma), q') \in I_q$. The proof is conducted by induction on the length of σ .

Base cases: $|\sigma| = 1$, meaning that $\sigma \in \Sigma_{\text{pStm}}$. Let suppose that $\exists q \in Q. \mathfrak{S}(\sigma) \in \mathcal{L}_q(A)$. We can split the base cases as follows.

- $\sigma \in \text{Punct}$: The first call to `Build` (line 6 of Algorithm 12) is `Build(A, q0)`, that calls `BuildTr(q0, ε, ∅)` (line 3 of Algorithm 13). By definition, any character of `Punct` is a single character, meaning that there exists a transition in A from q_0 to q labeled with $\mathfrak{S}(\sigma)$, namely $(q_0, \mathfrak{S}(\sigma), q) \in \delta^A$. The transition is taken into account at line 2 (i.e. the pair $(\mathfrak{S}(\sigma), q) \in \Delta_{q_0}$) and will be eventually selected at line 4. No states have been already marked, since it is the first call to `BuildTr`, hence the execution passes the test at line 6. The test at line 7 fails, since $\sigma \in \text{Punct}$ while the test at line 9 is successful, since $\mathfrak{S}(\sigma) \in \text{Punct}$ and $\text{word}(\mathfrak{S}(\sigma)) = \mathfrak{S}(\sigma) \in \text{Punct} \subseteq \Sigma_{\text{pStm}}$. Hence, the couple (σ, q) is added to I_q . Other transitions may be selected at line 4 of Algorithm 13, without removing the ones already added.
- $\sigma \notin \text{Punct}$: By definition of Σ_{pStm} , any single partial statement is a sequence of some non-punctuation symbols ending with a punctuation symbol (e.g., $x = 5;$). Hence, $\mathfrak{S}(\sigma)$ can be rewritten as $\mathfrak{S}(\sigma) = \mathfrak{S}(\sigma'p)$, $p \in \text{Punct}$, $\sigma' \in (\Sigma \setminus \text{Punct})^*$. Let $n = |\mathfrak{S}(\sigma')|$ be the length of $\mathfrak{S}(\sigma')$ and c_i be the i -th character of $\mathfrak{S}(\sigma')$. Since $\mathfrak{S}(\sigma) \in \mathcal{L}_q(A)$, there exists a path of A from q_0 to q that reads $\mathfrak{S}(\sigma) = \mathfrak{S}(\sigma'p)$, that is $\forall i \in [0, n-1]. (q_i, c_i, q_{i+1}) \in \delta^A \wedge (q_n, p, q) \in \delta^A$.

As in the previous case, the first call to `Build` is `Build(A, q0)`, calling then `BuildTr(q0, ε, ∅)` (line 3 of Algorithm 13). The transition (q_0, c_0, q_1) , i.e., the

transition that reads the first symbol of $\mathfrak{S}(\sigma)$, is taken into account at line 2 and will be eventually selected at line 4. Since no states have been marked yet, the test at line 6 is successful. c_0 is not a punctuation symbol, hence, the current call also passes the test at line 7, performing a recursive call to `BuildTr`, namely `BuildTr($q_1, c_0, \{q_0, q_1\}$)`, accumulating c_0 in the second parameter of the recursive call and searching for the first punctuation symbol. In particular, for any $i \in [0, |n - 1|]$, when the transition (q_i, c_i, q_{i+1}) is selected at line 4, any recursive call to `BUILDTR` on a state q_i will fall down in a recursive call to q_{i+1} , at lines 7-8 of Algorithm 13 and it accumulates the current character, namely c_i . Hence, when `BuildTr` is called on the state q_n , the parameter word is $\mathfrak{S}(\sigma')$, and this call corresponds to the last recursive call of the path we are considering, namely `BuildTr($q_n, \mathfrak{S}(\sigma'), \{q_0, q_1, \dots, q_n\}$)`. The transition (q_n, p, q) is added to Δ_{q_n} at line 2 and it will be eventually selected at line 4. Since $p \in \text{Punct}$ and $\sigma'p \in \Sigma_{\text{pStm}}$, the pair $(\sigma'p, q) = (\sigma, q)$ is added to I_{q_n} .

Inductive step: Let A be a cycle-executable FA and let consider $n \in \mathbb{N}, n > 1$. We suppose that $\forall \sigma \in \Sigma_{\text{pStm}}^*. |\sigma| < n, \exists q \in Q. \mathfrak{S}(\sigma) \in \mathcal{L}_q(A) \Rightarrow \sigma \in \mathcal{L}_q(A^{\text{pStm}})$. We prove that $\forall \delta \in \Sigma_{\text{pStm}}^*. |\delta| \geq n, \exists q' \in Q. \mathfrak{S}(\delta) \in \mathcal{L}_{q'}(A) \Rightarrow \delta \in \mathcal{L}_{q'}(A^{\text{pStm}})$.

Let consider $\delta = \sigma\sigma'$, where $\sigma' \in \Sigma_{\text{pStm}}$, and suppose that $\exists q' \in Q. \mathfrak{S}(\delta) \in \mathcal{L}_{q'}(A)$, meaning that there exists a path in A from q_0 to some state q' that reads δ . Since $\mathfrak{S}(\sigma)$ is a prefix of $\mathfrak{S}(\delta)$, it is clear that, starting from q_0 , it will reach some state q of A , namely $\exists q \in Q. \mathfrak{S}(\sigma) \in \mathcal{L}_q(A)$. Hence, for inductive hypothesis, $\sigma \in \mathcal{L}_q(A^{\text{pStm}})$. Let $\sigma = \sigma_0\sigma_1 \dots \sigma_{n-1}$, where $\sigma_i \in \Sigma_{\text{pStm}}$, for $i \in [0, n - 1]$. The state q is a reachable state in A^{pStm} and, in particular, it is reached by the last partial statement of σ , namely (σ_{n-1}) . For our hypothesis, from q it is possible to read $\mathfrak{S}(\sigma')$, hence, `Build(A, q)` will be called at line 6 of Algorithm 12. Since the lemma is independent from the state, we can apply the same cases showed in the base, starting from q and showing that also the pair (σ', q') will be added to the set $I_{q'}$, and consequently to A^{pStm} . Concluding, since $\sigma \in \mathcal{L}_q(A^{\text{pStm}}) \wedge (\sigma'q') \in I_{q'} \Rightarrow \delta = \sigma\sigma' \in \mathcal{L}_{q'}(A^{\text{pStm}})$.

Since the generalized lemma has been proved for a generic state $q \in Q$, clearly Lemma 6.11 also holds when $q \in F$. □

Proposition A.2. Let $r_1, r_2, r \in RE$. By construction of $\wr \cdot \wr$ and $\text{CFG}_{\mu\text{JS}}$ the following facts hold.

$$\begin{aligned} \text{Paths}(\text{CFG}_{\mu\text{JS}}(\wr r_1 \mid \mid r_2 \wr)) &= \left\{ \text{true}\pi\text{true} \left| \begin{array}{l} \pi \in \text{Paths}(\text{CFG}_{\mu\text{JS}}(\wr r_1 \wr)) \vee \\ \pi \in \text{Paths}(\text{CFG}_{\mu\text{JS}}(\wr r_2 \wr)) \end{array} \right. \right\} \\ \text{Paths}(\text{CFG}_{\mu\text{JS}}(\wr (r_1)^* \wr)) &= \{ \text{true}(\pi\text{true})^n \mid \pi \in \text{Paths}(\text{CFG}_{\mu\text{JS}}(\wr r_1 \wr)), n \in \mathbb{N} \} \end{aligned}$$

Proposition A.3. Given $r \in RE, \forall \delta \in \mathcal{L}(r)$, if $\delta \notin \mu\text{JS}$ then δ is replaced by skip (hence, discarded) by $\wr r \wr^P$.

Proposition A.3 follows from the construction of $\wr \cdot \wr^P$ and $\wr \cdot \wr$, and can be easily proved by induction of the structure of the regular expressions.

Lemma 6.13

Proof of Lemma 6.13. We recall that \mathfrak{S} converts a string of strings (in $(\Sigma^*)^*$) in a string of chars (in Σ^*), and code interprets a string of chars as a executable code, if possible. Let us prove by induction on the structure of r . Let us begin with the base case.

- let suppose that $r = d$ and $\text{code}(\mathfrak{S}(d)) \in \mu\text{JS}$, then $G_r = \text{CFG}_{\mu\text{JS}}(\text{code}(\mathfrak{S}(d)))$. By Equation 6.2 we have that $\forall m^\# \in \mathbb{M}^\#$. $\exists \Pi \subseteq \text{Paths}(G_r)$. $\langle \text{code}(\mathfrak{S}(d)) \rangle^\# m^\# \subseteq \bigcup_{\pi \in \Pi} \langle \pi \rangle^\# m^\#$.
- let suppose that $r = d$ and $\text{code}(\mathfrak{S}(d)) \notin \mu\text{JS}$, then $G_r = \text{CFG}_{\mu\text{JS}}(\text{skip})$. In this case, $\langle \text{code}(\mathfrak{S}(d)) \rangle^\# = \lambda m^\#. m^\#$ since it cannot modify memories not being a legal statement. By construction, $\forall \pi \in \text{Paths}(\text{CFG}(\text{skip}))$ we have that, for any $m^\# \in \mathbb{M}^\#$, $\langle \text{code}(\mathfrak{S}(\delta)) \rangle^\# = \langle \pi \rangle^\# m^\# = m^\#$, hence trivially we have the thesis.

Let us prove now the inductive steps.

- Let $r = r_1 \mid r_2$. Since a string $\delta \in \mathcal{L}(r_1 \mid r_2)$ can be either belong to $\mathcal{L}(r_1)$ or $\mathcal{L}(r_2)$, we can split the proof in two cases. We show the proof when $\delta \in \mathcal{L}(r_1)$, the case $\delta \in \mathcal{L}(r_2)$ is identical. Let $\delta \in \mathcal{L}(r_1)$. For inductive hypothesis, the thesis holds for r_1 and the following fact holds. Let $G_{r_1} \triangleq \text{CFGGen}(r_1)$, then $\forall \delta \in \mathcal{L}(r_1)$. $\forall m^\# \in \mathbb{M}^\#$

$$\exists \Pi_1 \subseteq \text{Paths}(G_{r_1}) \text{ s.t. } \langle \text{code}(\mathfrak{S}(\delta)) \rangle^\# m^\# \subseteq \bigcup_{\pi_1 \in \Pi_1} \langle \pi_1 \rangle^\# m^\#$$

Let $G_r \triangleq \text{CFGGen}(r)$ and $\Pi = \{ \text{true} \pi_1 \text{true} \mid \pi_1 \in \Pi_1 \}$. By Proposition A.2, $\Pi \subseteq \text{Paths}(G_r)$. Moreover, since $\langle \text{true} \rangle^\# m^\# = m^\#$, namely true semantics does not alter the input memory, the thesis holds.

$$\begin{aligned} & \langle \text{code}(\mathfrak{S}(\delta)) \rangle^\# m^\# \\ & \subseteq \bigcup_{\pi_1 \in \Pi_1} \langle \pi_1 \rangle^\# m^\# && \text{\{Inductive hp.\}} \\ & = \bigcup_{\pi_1 \in \Pi_1} \langle \text{true} \rangle^\# \circ \langle \pi_1 \rangle^\# \circ \langle \text{true} \rangle^\# m^\# && \text{\{true semantics\}} \\ & = \bigcup_{\pi \in \Pi} \langle \pi \rangle^\# m^\# && \text{\{Def. of \Pi\}} \end{aligned}$$

- Let $r = r_1 r_2$. For inductive hypothesis the following facts hold. Let $G_{r_1} \triangleq \text{CFGGen}(r_1)$ and $G_{r_2} \triangleq \text{CFGGen}(r_2)$. Then, $\forall m^\# \in \mathbb{M}^\#$

$$\begin{aligned} \forall \delta_1 \in \mathcal{L}(r_1). \exists \Pi_1 \subseteq \text{Paths}(G_{r_1}) \text{ s.t. } \langle \text{code}(\mathfrak{S}(\delta_1)) \rangle^\# m^\# & \subseteq \bigcup_{\pi_1 \in \Pi_1} \langle \pi_1 \rangle^\# m^\# \\ \forall \delta_2 \in \mathcal{L}(r_2). \exists \Pi_2 \subseteq \text{Paths}(G_{r_2}) \text{ s.t. } \langle \text{code}(\mathfrak{S}(\delta_2)) \rangle^\# m^\# & \subseteq \bigcup_{\pi_2 \in \Pi_2} \langle \pi_2 \rangle^\# m^\# \end{aligned}$$

Let $G_r \triangleq \text{CFGGen}(r) = \text{CFGGen}(r_1 r_2)$ and $\delta \in \mathcal{L}(r_1 r_2)$. Clearly, $\exists \delta_1 \in \mathcal{L}(r_1), \delta_2 \in \mathcal{L}(r_2)$. $\delta_1 \delta_2 = \delta$. Without loss of generality, let suppose that $\delta_1, \delta_2 \in \mu\text{JS}$, since, by Proposition A.3, any non-executable string would be discarded and replaced by skip, and the thesis would trivially holds (i.e., G_r would corresponds to the CFG of skip and $\langle \text{code}(\delta) \rangle^\# m^\# = \langle \text{skip} \rangle^\# m^\#$). Hence, the paths of G_r corresponds to the set $\Pi = \{ \pi_1 \pi_2 \mid \pi_1 \in \Pi_1, \pi_2 \in \Pi_2 \}$.

$$\begin{aligned} & \langle \text{code}(\mathfrak{S}(\delta)) \rangle^\# m^\# \\ & = \langle \text{code}(\mathfrak{S}(\delta_1 \delta_2)) \rangle^\# m^\# \\ & = \langle \text{code}(\mathfrak{S}(\delta_2)) \rangle^\# \circ \langle \text{code}(\mathfrak{S}(\delta_1)) \rangle^\# m^\# \end{aligned}$$

$$\begin{aligned}
&\subseteq \bigcup_{\pi_2 \in \Pi_2} (\downarrow \pi_2 \downarrow)^\# \left(\bigcup_{\pi_1 \in \Pi_1} (\downarrow \pi_1 \downarrow)^\# m^\# \right) && \text{[Inductive hp.]} \\
&= \bigcup_{\pi_1 \in \Pi_1, \pi_2 \in \Pi_2} (\downarrow \pi_2 \downarrow)^\# \circ (\downarrow \pi_1 \downarrow)^\# m^\# \\
&= \bigcup_{\pi \in \Pi} (\downarrow \pi \downarrow)^\# m^\# && \text{[Def. of } \Pi \text{]}
\end{aligned}$$

- Let $r = (r_1)^*$. For inductive hypothesis the following fact holds. Let $G_{r_1} \triangleq \text{CFGGen}(r_1)$. Then, $\forall \delta_1 \in \mathcal{L}(r_1). \forall m^\# \in \mathbb{M}^\#$

$$\exists \Pi_1 \subseteq \text{Paths}(G_{r_1}) \text{ s.t. } (\downarrow \text{code}(\mathfrak{S}(\delta_1)) \downarrow)^\# m^\# \subseteq \bigcup_{\pi_1 \in \Pi_1} (\downarrow \pi_1 \downarrow)^\# m^\#$$

Let $G_r \triangleq \text{CFGGen}(r_1)^*$ and $\Pi = \{ \text{true}(\pi_1 \text{true})^n \mid \pi_1 \in \Pi_1 \}$. By Proposition A.2, $\Pi \subseteq \text{Paths}(G_{(r_1)^*})$. Let $\delta_1 \in \mathcal{L}(r_1)$ and $\delta = (\delta_1)^n \in \mathcal{L}((r_1)^*)$, for some $n \in \mathbb{N}$. Since r is regular expression obtained from a cycle executable automaton, $\delta_1 \in \mu\text{JS}$. Then, $\forall m^\# \in \mathbb{M}^\#$

$$\begin{aligned}
&(\downarrow \text{code}(\mathfrak{S}(\delta)) \downarrow)^\# m^\# = (\downarrow \text{code}(\mathfrak{S}((\delta_1)^n)) \downarrow)^\# m^\# \\
&= ((\downarrow \text{code}(\mathfrak{S}(\delta_1)) \downarrow)^\#)^n m^\# && \text{[}\delta_1 \in \mu\text{JS]} \\
&\subseteq \left(\bigcup_{\pi_1 \in \Pi_1} (\downarrow \pi_1 \downarrow)^\# \right)^n m^\# && \text{[Inductive hp.]} \\
&= (\downarrow \text{true} \downarrow)^\# \circ \left(\bigcup_{\pi_1 \in \Pi_1} (\downarrow \pi_1 \downarrow)^\# \circ (\downarrow \text{true} \downarrow)^\# m^\# \right)^n && \text{[true semantics]} \\
&= \bigcup_{\pi \in \Pi} (\downarrow \pi \downarrow)^\# m^\# && \text{[Def. of } \Pi \text{]}
\end{aligned}$$

□

Theorem 7.3

Proof of Theorem 7.3. We need to prove the soundness of the expressions and the statements concerning objects. For any other expression and statement nor regarding objects, we suppose to have soundness. namely $\forall m^\# \in \mathbb{M}^\#$

$$(\downarrow \text{st} \downarrow) \gamma_{m^\#}(m^\#) \subseteq \gamma_{m^\#}((\downarrow \text{st} \downarrow)^\# m^\#) \quad (\downarrow e \downarrow) \gamma_{m^\#}(m^\#) \subseteq \gamma_{\text{Dyn}}((\downarrow e \downarrow)^\# m^\#) \quad (\text{A.3})$$

In order to do not clutter the notation, we denote $\gamma_{m^\#}$ as γ .

Case $\{s_0 : e_0, s_1 : e_1, \dots, s_n : e_n\}$

Given the following object expression

$$\{s_0 : e_0, s_1 : e_1, \dots, s_n : e_n\}$$

in the proof, let $o^\# = [\text{Min}(\{s_n\}) \mapsto (\downarrow e_n \downarrow)^\# m^\#] \bullet \dots \bullet [\text{Min}(\{s_0\}) \mapsto (\downarrow e_0 \downarrow)^\# m^\#]$.

$$\begin{aligned}
&(\downarrow \{s_0 : e_0, s_1 : e_1, \dots, s_n : e_n\} \downarrow) \gamma(m^\#) \\
&= \left\{ [s_n \mapsto v_n] \bullet \dots [s_1 \mapsto v_1] \bullet [s_0 \mapsto v_0] \left| \begin{array}{l} \forall i \in [0, n] \\ v_i \in (\downarrow e_i \downarrow) \gamma(m^\#) \end{array} \right. \right\} && \text{[Def. } (\downarrow \cdot \downarrow) \text{]}
\end{aligned}$$

$$\begin{aligned}
&\subseteq \left\{ [s_n \mapsto v_n] \bullet \dots [s_1 \mapsto v_1] \bullet [s_0 \mapsto v_0] \left| \begin{array}{l} \forall i \in [0, n] \\ v_i \in \gamma(\langle e_i \rangle^\# m^\#) \end{array} \right. \right\} && \text{[Eq. A.3]} \\
&= \left\{ [s_n \mapsto v_n] \bullet \dots [s_1 \mapsto v_1] \bullet [s_0 \mapsto v_0] \left| \begin{array}{l} \forall i \in [0, n] \\ v_i \in \gamma(o^\#(s_i)) \end{array} \right. \right\} && \text{[} o^\#(s_i) = \langle e_i \rangle^\# m^\# \text{]} \\
&= \left\{ [\sigma_n \mapsto v_n] \bullet \dots [\sigma_1 \mapsto v_1] \bullet [\sigma_0 \mapsto v_0] \left| \begin{array}{l} \forall i \in [0, n] \\ \forall \sigma \in \text{Min}(\{s_i\}) \\ v_i \in \gamma(o^\#(\sigma)) \end{array} \right. \right\} && \text{[} |\mathcal{L}(\text{Min}(\{s_i\}))| = 1 \text{]} \\
&= \gamma(\langle \text{Min}(\{s_n\}) \mapsto \langle e_n \rangle^\# m^\# \bullet \dots \bullet \langle \text{Min}(\{s_0\}) \mapsto \langle e_0 \rangle^\# m^\# \rangle) && \text{[Def. of } \gamma \text{]} \\
&= \gamma(\langle \{s_0 : e_0, s_1 : e_1, \dots, s_n : e_n\} \rangle^\# m^\#) && \text{[Def. of } \langle \cdot \rangle^\# \text{]}
\end{aligned}$$

Case $\langle x = \text{new } o \rangle$

$$\begin{aligned}
&\langle x = \text{new } o \rangle \gamma(m^\#) = \\
&= \gamma(m^\#)[x \leftarrow \langle o \rangle \gamma(m^\#)] && \text{[Def. } \langle \cdot \rangle \text{]} \\
&\subseteq \gamma(m^\#)[x \leftarrow \gamma(\langle o \rangle^\# m^\#)] && \text{[Eq. A.3]} \\
&= \lambda y. (\gamma(m^\#)[x \leftarrow \gamma(\langle o \rangle^\# m^\#)])(y) && \text{[Def. memory]} \\
&= \lambda y. (\gamma(m^\#[x \leftarrow \langle o \rangle^\# m^\#]))(y) && \text{[Def. memory update]} \\
&= \gamma(m^\#[x \leftarrow \langle o \rangle^\# m^\#]) && \text{[Def. memory update]} \\
&= \gamma(m^\#[x \leftarrow \text{Norm}(\langle o \rangle^\# m^\#)]) && \text{[Prop. 7.2]} \\
&= \gamma(\langle x = \text{new } o \rangle^\# m^\#) && \text{[Def. } \langle \cdot \rangle^\# \text{]}
\end{aligned}$$

Case $x[s]$

$$\begin{aligned}
&\langle x[s] \rangle \gamma(m^\#) = \\
&= \bigcup \{ o(s) \mid o \in \langle x \rangle \gamma(m^\#), s \in \langle s \rangle \gamma(m^\#), s \in \text{props}(o) \} && \text{[Def. } \langle \cdot \rangle \text{]} \\
&\subseteq \bigcup \{ o(s) \mid o \in \gamma(\langle x \rangle^\# m^\#), s \in \gamma(\langle s \rangle^\# m^\#), s \in \text{props}(o) \} && \text{[Eq. A.3]} \\
&= \bigcup \{ o(s) \mid o \in \gamma(m^\#(x)), s \in \gamma(\langle s \rangle^\# m^\#), s \in \text{props}(o) \} && \text{[Def. } \langle x \rangle^\# m^\# \text{]} \\
&= \bigcup \left\{ o(s) \left| \begin{array}{l} o \in \gamma(m^\#(x)), s \in \gamma(\langle s \rangle^\# m^\#) \\ s \in P, P \in \text{props}(\gamma(m^\#(x))) \end{array} \right. \right\} \\
&= \bigcup \left\{ o(s) \left| \begin{array}{l} o \in \gamma(m^\#(x)), s \in \gamma(\langle s \rangle^\# m^\#) \\ s \in \gamma(\mathbf{A}), \mathbf{A} \in \text{props}(m^\#(x)) \end{array} \right. \right\} && \text{[} \mathbf{A} = \mathcal{L}(P) = \gamma(\mathbf{A}) \text{]} \\
&= \bigcup \left\{ o(s) \left| \begin{array}{l} o \in \gamma(m^\#(x)), s \in \gamma(\langle s \rangle^\# m^\#), s \in \gamma(\mathbf{A}) \\ \mathbf{A} \in \text{props}(m^\#(x)), \gamma(\mathbf{A}) \cap \gamma(\langle s \rangle^\# m^\#) \neq \emptyset \end{array} \right. \right\} && \text{[Def. } \cap \text{]} \\
&= \bigcup \left\{ o(s) \left| \begin{array}{l} o \in \gamma(m^\#(x)), s \in \gamma(\langle s \rangle^\# m^\#), s \in \gamma(\mathbf{A}) \\ \mathbf{A} \in \text{props}(m^\#(x)), \mathbf{A} \sqcap_{\text{DFA}} \langle s \rangle^\# m^\# \neq \text{Min}(\emptyset) \end{array} \right. \right\} && \text{[Closure props. DFA/}\equiv \text{]} \\
&\subseteq \gamma(\bigsqcup \left\{ m^\#(x)(\mathbf{A}) \left| \begin{array}{l} \mathbf{A} \in \text{props}(m^\#(x)) \\ \mathbf{A} \sqcap_{\text{DFA}} \langle s \rangle^\# m^\# \neq \text{Min}(\emptyset) \end{array} \right. \right\}) && \text{[Def. } \gamma \text{]}
\end{aligned}$$

$$= \gamma(\llbracket x[s] \rrbracket^{\#} m^{\#}) \quad \{\text{Def. } \llbracket \cdot \rrbracket^{\#}\}$$

Case $x[s] = e$

$$\begin{aligned} & \llbracket x[s] = e \rrbracket^{\#} \gamma(m^{\#}) = \\ & = \gamma(m^{\#})[x \leftarrow \left\{ o \bullet [s \mapsto v] \left| \begin{array}{l} o \in \llbracket x \rrbracket^{\#} \gamma(m^{\#}) \\ s \in \llbracket s \rrbracket^{\#} \gamma(m^{\#}) \\ v \in \llbracket e \rrbracket^{\#} \gamma(m^{\#}) \end{array} \right. \right\}] \quad \{\text{Def. } \llbracket \cdot \rrbracket^{\#}\} \\ & \subseteq \gamma(m^{\#})[x \leftarrow \left\{ o \bullet [s \mapsto v] \left| \begin{array}{l} o \in \gamma(\llbracket x \rrbracket^{\#} m^{\#}) \\ s \in \gamma(\llbracket s \rrbracket^{\#} m^{\#}) \\ v \in \gamma(\llbracket e \rrbracket^{\#} m^{\#}) \end{array} \right. \right\}] \quad \{\text{Eq. A.3}\} \\ & = \gamma(m^{\#})[x \leftarrow \left\{ o \bullet [s \mapsto v] \left| \begin{array}{l} o \in \gamma(m^{\#}(x)) \\ s \in \gamma(\llbracket s \rrbracket^{\#} m^{\#}) \\ v \in \gamma(\llbracket e \rrbracket^{\#} m^{\#}) \end{array} \right. \right\}] \quad \{\text{Def. } \llbracket x \rrbracket^{\#} m^{\#}\} \\ & \subseteq \gamma(m^{\#})[x \leftarrow \left\{ o \bullet o' \left| \begin{array}{l} o \in \gamma(m^{\#}(x)) \\ o' \in \gamma(\llbracket s \rrbracket^{\#} m^{\#} \mapsto \llbracket e \rrbracket^{\#} m^{\#}) \end{array} \right. \right\}] \\ & = \gamma(m^{\#})[x \leftarrow \{ o \mid o \in \gamma(m^{\#}(x)) \bullet \gamma(\llbracket s \rrbracket^{\#} m^{\#} \mapsto \llbracket e \rrbracket^{\#} m^{\#}) \}] \\ & = \gamma(m^{\#})[x \leftarrow \{ o \mid o \in \gamma(m^{\#}(x)) \bullet \llbracket s \rrbracket^{\#} m^{\#} \mapsto \llbracket e \rrbracket^{\#} m^{\#} \}] \quad \{\text{Def. } \bullet\} \\ & = \gamma(m^{\#})[x \leftarrow \gamma(m^{\#}(x)) \bullet \llbracket s \rrbracket^{\#} m^{\#} \mapsto \llbracket e \rrbracket^{\#} m^{\#}] \quad \{\text{Def. } \gamma(m^{\#})\} \\ & = \gamma(m^{\#}[x \leftarrow m^{\#}(x) \bullet \llbracket s \rrbracket^{\#} m^{\#} \mapsto \llbracket e \rrbracket^{\#} m^{\#}]) \quad \{\text{Def. mem. update}\} \\ & \subseteq \gamma(m^{\#}[x \leftarrow m^{\#}(x) \bullet \llbracket s \rrbracket^{\#} m^{\#} \mapsto \llbracket e \rrbracket^{\#} m^{\#} \sqcup_{\text{Dyn}} m^{\#}(x)(\llbracket s \rrbracket^{\#} m^{\#})]) \quad \{\text{For } \sqcup_{\text{Dyn}}\} \\ & = \gamma(\llbracket x[s] = e \rrbracket^{\#} m^{\#}) \quad \{\text{Def. } \llbracket \cdot \rrbracket^{\#}, \text{Prop. 7.2}\} \end{aligned}$$

□

Bibliography

- Abdulla, Parosh Aziz et al. (2014). “String Constraints for Verification”. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pp. 150–166. DOI: [10.1007/978-3-319-08867-9_10](https://doi.org/10.1007/978-3-319-08867-9_10).
- Alur, Rajeev and P. Madhusudan (2004). “Visibly pushdown languages”. In: *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pp. 202–211. DOI: [10.1145/1007352.1007390](https://doi.org/10.1145/1007352.1007390).
- Amadini, Roberto et al. (2017). “Combining String Abstract Domains for JavaScript Analysis: An Evaluation”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pp. 41–57. DOI: [10.1007/978-3-662-54577-5_3](https://doi.org/10.1007/978-3-662-54577-5_3).
- An, Jong-hoon (David) et al. (2011). “Dynamic inference of static types for ruby”. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pp. 459–472. DOI: [10.1145/1926385.1926437](https://doi.org/10.1145/1926385.1926437).
- Anckaert, Bertrand, Matias Madou, and Koen De Bosschere (2006). “A Model for Self-Modifying Code”. In: *Information Hiding, 8th International Workshop, IH 2006, Alexandria, VA, USA, July 10-12, 2006. Revised Selected Papers*, pp. 232–248. DOI: [10.1007/978-3-540-74124-4_16](https://doi.org/10.1007/978-3-540-74124-4_16).
- Anderson, Christopher, Paola Giannini, and Sophia Drossopoulou (2005). “Towards Type Inference for JavaScript”. In: *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, pp. 428–452. DOI: [10.1007/11531142_19](https://doi.org/10.1007/11531142_19).
- Arceri, Vincenzo and Sergio Maffei (2017). “Abstract Domains for Type Juggling”. In: *Electr. Notes Theor. Comput. Sci.* 331, pp. 41–55. DOI: [10.1016/j.entcs.2017.02.003](https://doi.org/10.1016/j.entcs.2017.02.003). URL: <https://doi.org/10.1016/j.entcs.2017.02.003>.
- Arceri, Vincenzo and Isabella Mastroeni (2019). “Static Program Analysis for String Manipulation Languages”. In: *Proceedings Seventh International Workshop on Verification and Program Transformation, Genova, Italy, 2nd April 2019*. Ed. by Alexei Lisitsa and Andrei Nemytykh. Vol. 299. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 19–33. DOI: [10.4204/EPTCS.299.5](https://doi.org/10.4204/EPTCS.299.5).
- (2020). “A sound abstract interpreter for dynamic code”. In: *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*. Ed. by Chih-Cheng Hung et al. ACM, pp. 1979–1988. DOI: [10.1145/3341105.3373964](https://doi.org/10.1145/3341105.3373964).
- Arceri, Vincenzo, Michele Pasqua, and Isabella Mastroeni (2019). “An abstract domain for objects in dynamic programming languages”. In: *8th International Workshop on Numerical and Symbolic Abstract Domains - NSAD'19*.
- Arceri, Vincenzo et al. (2019). “Completeness of Abstract Domains for String Analysis of JavaScript Programs”. In: *Theoretical Aspects of Computing - ICTAC 2019 -*

- 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, *Proceedings*, pp. 255–272. DOI: [10.1007/978-3-030-32505-3_15](https://doi.org/10.1007/978-3-030-32505-3_15).
- Balakrishnan, Gogul and Thomas W. Reps (2006). “Recency-Abstraction for Heap-Allocated Storage”. In: *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, pp. 221–239. DOI: [10.1007/11823230_15](https://doi.org/10.1007/11823230_15).
- Bancerek, Grzegorz and Piotr Rudnicki (2002). “A Compendium of Continuous Lattices in MIZAR”. In: *J. Autom. Reasoning* 29.3-4, pp. 189–224. DOI: [10.1023/A:1021966832558](https://doi.org/10.1023/A:1021966832558). URL: <https://doi.org/10.1023/A:1021966832558>.
- Bartzis, Constantinos and Tevfik Bultan (2004). “Widening Arithmetic Automata”. In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, pp. 321–333. DOI: [10.1007/978-3-540-27813-9_25](https://doi.org/10.1007/978-3-540-27813-9_25). URL: https://doi.org/10.1007/978-3-540-27813-9_25.
- Bessey, Al et al. (2010). “A few billion lines of code later: using static analysis to find bugs in the real world”. In: *Commun. ACM* 53.2, pp. 66–75. DOI: [10.1145/1646353.1646374](https://doi.org/10.1145/1646353.1646374).
- Biggar, P. and D. Gregg (2009). *Static analysis of dynamic scripting languages*. Technical Report. Department of Computer Science, Trinity College Dublin.
- Bodden, Eric et al. (2011). “Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders”. In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011*, pp. 241–250. DOI: [10.1145/1985793.1985827](https://doi.org/10.1145/1985793.1985827).
- Bouajjani, Ahmed, Peter Habermehl, and Tomas Vojnar (2004). “Abstract Regular Model Checking”. In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, pp. 372–386. DOI: [10.1007/978-3-540-27813-9_29](https://doi.org/10.1007/978-3-540-27813-9_29).
- Bouajjani, Ahmed et al. (2008). “Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata”. In: *Implementation and Applications of Automata, 13th International Conference, CIAA 2008, San Francisco, California, USA, July 21-24, 2008, Proceedings*, pp. 57–67. DOI: [10.1007/978-3-540-70844-5_7](https://doi.org/10.1007/978-3-540-70844-5_7).
- Brzozowski, J.A. (1962). “Canonical regular expressions and minimal state graphs for definite events”. In: *Mathematical Theory of Automata* 12, pp. 529–561.
- Brzozowski, Janusz A. (1964). “Derivatives of Regular Expressions”. In: *J. ACM* 11.4. ISSN: 0004-5411.
- Cai, Hongxu, Zhong Shao, and Alexander Vaynberg (2007). “Certified self-modifying code”. In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pp. 66–77. DOI: [10.1145/1250734.1250743](https://doi.org/10.1145/1250734.1250743).
- Campeanu, Cezar, Andrei Paun, and Sheng Yu (2002). “An Efficient Algorithm for Constructing Minimal Cover Automata for Finite Languages”. In: *Int. J. Found. Comput. Sci.* 13.1, pp. 83–97. DOI: [10.1142/S0129054102000960](https://doi.org/10.1142/S0129054102000960).
- Chen, Liqian, Antoine Mine, and Patrick Cousot (2008). “A Sound Floating-Point Polyhedra Abstract Domain”. In: *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008, Proceedings*, pp. 3–18. DOI: [10.1007/978-3-540-89330-1_2](https://doi.org/10.1007/978-3-540-89330-1_2).
- Choi, Tae-Hyoung et al. (2006). “A Practical String Analyzer by the Widening Approach”. In: *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings*, pp. 374–388. DOI: [10.1007/11924661_23](https://doi.org/10.1007/11924661_23).

- Christensen, Aske Simon, Anders Møller, and Michael I. Schwartzbach (2003). “Precise Analysis of String Expressions”. In: *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, pp. 1–18. DOI: [10.1007/3-540-44898-5_1](https://doi.org/10.1007/3-540-44898-5_1).
- Chugh, Ravi et al. (2009). “Staged information flow for javascript”. In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pp. 50–62. DOI: [10.1145/1542476.1542483](https://doi.org/10.1145/1542476.1542483).
- Clarísó, Robert and Jordi Cortadella (2007). “The Octahedron Abstract Domain”. In: *Sci. Comput. Program.* 64.1, pp. 115–139.
- Cortesi, Agostino (2008). “Widening Operators for Abstract Interpretation”. In: *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*, pp. 31–40. DOI: [10.1109/SEFM.2008.20](https://doi.org/10.1109/SEFM.2008.20). URL: <https://doi.org/10.1109/SEFM.2008.20>.
- Cortesi, Agostino, Giulia Costantini, and Pietro Ferrara (2013). “A Survey on Product Operators in Abstract Interpretation”. In: *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*. Pp. 325–336. DOI: [10.4204/EPTCS.129.19](https://doi.org/10.4204/EPTCS.129.19).
- Cortesi, Agostino et al. (1997). “Complementation in Abstract Interpretation”. In: *ACM Trans. Program. Lang. Syst.* 19.1, pp. 7–47. DOI: [10.1145/239912.239914](https://doi.org/10.1145/239912.239914).
- Costantini, Giulia, Pietro Ferrara, and Agostino Cortesi (2015). “A suite of abstract domains for static analysis of string values”. In: *Softw., Pract. Exper.* 45.2, pp. 245–287. DOI: [10.1002/spe.2218](https://doi.org/10.1002/spe.2218). URL: <https://doi.org/10.1002/spe.2218>.
- Cousot, P. and R. Cousot (1995). “Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation”. In: *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*. ACM Press, New York, NY, pp. 170–181.
- Cousot, Patrick (1997). “Types as Abstract Interpretations”. In: *Conference Record of POPL’97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pp. 316–331. DOI: [10.1145/263699.263744](https://doi.org/10.1145/263699.263744). URL: <https://doi.org/10.1145/263699.263744>.
- Cousot, Patrick and Radhia Cousot (1976). “Static determination of dynamic properties of programs”. In: *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, pp. 106–130.
- (1977). “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pp. 238–252. DOI: [10.1145/512950.512973](https://doi.org/10.1145/512950.512973). URL: <https://doi.org/10.1145/512950.512973>.
- (1979). “Systematic Design of Program Analysis Frameworks”. In: *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pp. 269–282. DOI: [10.1145/567752.567778](https://doi.org/10.1145/567752.567778).
- (1992a). “Abstract Interpretation Frameworks”. In: *J. Log. Comput.* 2.4, pp. 511–547. DOI: [10.1093/logcom/2.4.511](https://doi.org/10.1093/logcom/2.4.511). URL: <https://doi.org/10.1093/logcom/2.4.511>.
- (1992b). “Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation”. In: *Programming Language Implementation and Logic*

- Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings*, pp. 269–295. DOI: [10.1007/3-540-55844-6_142](https://doi.org/10.1007/3-540-55844-6_142). URL: https://doi.org/10.1007/3-540-55844-6_142.
- Cousot, Patrick and Radhia Cousot (2012). “An abstract interpretation framework for termination”. In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pp. 245–258. DOI: [10.1145/2103656.2103687](https://doi.org/10.1145/2103656.2103687).
- Cousot, Patrick and Nicolas Halbwachs (1978). “Automatic Discovery of Linear Constraints Among Variables of a Program”. In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pp. 84–96. DOI: [10.1145/512760.512770](https://doi.org/10.1145/512760.512770). URL: <https://doi.org/10.1145/512760.512770>.
- Crockford, Douglas (2008). *JavaScript: The Good Parts*. O’Reilly Media, Inc. ISBN: 0596517742.
- Dahse, Johannes and Thorsten Holz (2014). “Simulation of Built-in PHP Features for Precise Static Code Analysis”. In: *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*.
- Davis, M. D., R. Sigal, and E. J. Weyuker (1994). *Computability, Complexity, and Languages: Fund. of Theor. CS*. Academic Press Professional, Inc. ISBN: 978-0-12-206380-0. DOI: [10.2307/2275691](https://doi.org/10.2307/2275691).
- Doh, Kyung-Goo, Hyunha Kim, and David A. Schmidt (2009). “Abstract Parsing: Static Analysis of Dynamically Generated String Output Using LR-Parsing Technology”. In: *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, pp. 256–272. DOI: [10.1007/978-3-642-03237-0_18](https://doi.org/10.1007/978-3-642-03237-0_18).
- Domaratzki, Michael, Jeffrey Shallit, and Sheng Yu (2001). “Minimal Covers of Formal Languages”. In: *Developments in Language Theory, 5th International Conference, DLT 2001, Vienna, Austria, July 16-21, 2001, Revised Papers*, pp. 319–329. DOI: [10.1007/3-540-46011-X_28](https://doi.org/10.1007/3-540-46011-X_28).
- D’Silva, V. (2006). *Widening for Automata*. Diploma Thesis, Institut Fur Informatik, UZH.
- Filaretti, Daniele and Sergio Maffei (2014). “An Executable Formal Semantics of PHP”. In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pp. 567–592. DOI: [10.1007/978-3-662-44202-9_23](https://doi.org/10.1007/978-3-662-44202-9_23). URL: https://doi.org/10.1007/978-3-662-44202-9_23.
- Fromherz, Aymeric, Abdelraouf Ouadjaout, and Antoine Miné (2018). “Static Value Analysis of Python Programs by Abstract Interpretation”. In: *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*, pp. 185–202. DOI: [10.1007/978-3-319-77935-5_14](https://doi.org/10.1007/978-3-319-77935-5_14).
- Giacobazzi, R. (1998). “Abductive analysis of modular logic programs”. In: *J. of Logic and Computation* 8.4, pp. 457–484.
- Giacobazzi, Roberto and Isabella Mastroeni (2016). “Making abstract models complete”. In: *Mathematical Structures in Computer Science* 26.4, pp. 658–701. DOI: [10.1017/S0960129514000358](https://doi.org/10.1017/S0960129514000358).
- (2018). “Abstract Non-Interference: A Unifying Framework for Weakening Information-flow”. In: *ACM Trans. Priv. Secur.* 21.2, 9:1–9:31. DOI: [10.1145/3175660](https://doi.org/10.1145/3175660).
- Giacobazzi, Roberto and Elisa Quintarelli (2001). “Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking”. In: *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings*, pp. 356–373. DOI: [10.1007/3-540-47764-0_20](https://doi.org/10.1007/3-540-47764-0_20).

- Giacobazzi, Roberto, Francesco Ranzato, and Francesca Scozzari (2000). "Making abstract interpretations complete". In: *J. ACM* 47.2, pp. 361–416. DOI: [10.1145/333979.333989](https://doi.org/10.1145/333979.333989).
- Granger, Philippe (Jan. 1989). "Static Analysis of Arithmetical Congruences". In: *International Journal of Computer Mathematics - IJCM* 30, pp. 165–190.
- Guarnieri, Salvatore et al. (2011). "Saving the world wide web from vulnerable JavaScript". In: *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pp. 177–187. DOI: [10.1145/2001420.2001442](https://doi.org/10.1145/2001420.2001442).
- Hauzar, David and Jan Kofron (2014). "WeVerca: Web Applications Verification for PHP". In: *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, pp. 296–301. DOI: [10.1007/978-3-319-10431-7_24](https://doi.org/10.1007/978-3-319-10431-7_24).
- (2015a). "Framework for Static Analysis of PHP Applications". In: *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pp. 689–711. DOI: [10.4230/LIPIcs.ECOOP.2015.689](https://doi.org/10.4230/LIPIcs.ECOOP.2015.689).
- (2015b). "Framework for Static Analysis of PHP Applications". In: *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pp. 689–711. DOI: [10.4230/LIPIcs.ECOOP.2015.689](https://doi.org/10.4230/LIPIcs.ECOOP.2015.689).
- Heintze, Nevin and Joxan Jaffar (1994). "Set Constraints and Set-Based Analysis". In: *Principles and Practice of Constraint Programming, Second International Workshop, PPCP'94, Rosario, Orcas Island, Washington, USA, May 2-4, 1994, Proceedings*, pp. 281–298. DOI: [10.1007/3-540-58601-6_107](https://doi.org/10.1007/3-540-58601-6_107).
- Holík, Lukás et al. (2018). "String constraints with concatenation and transducers solved efficiently". In: *PACMPL* 2.POPL, 4:1–4:32. DOI: [10.1145/3158092](https://doi.org/10.1145/3158092).
- Holzer, Markus and Sebastian Jakobi (2013). "Brzozowski's Minimization Algorithm - More Robust than Expected - (Extended Abstract)". In: *Implementation and Application of Automata - 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings*. Ed. by Stavros Konstantinidis. Vol. 7982. Lecture Notes in Computer Science. Springer, pp. 181–192. DOI: [10.1007/978-3-642-39274-0_17](https://doi.org/10.1007/978-3-642-39274-0_17).
- Hooimeijer, Pieter et al. (2011). "Fast and Precise Sanitizer Analysis with BEK". In: *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*.
- Hopcroft, John E. (1971). *An N Log N Algorithm for Minimizing States in a Finite Automaton*. Tech. rep. Stanford, CA, USA.
- Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman (2007). *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley. ISBN: 978-0-321-47617-3.
- Jensen, Simon Holm, Peter A. Jonsson, and Anders Møller (2012). "Remedying the eval that men do". In: *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pp. 34–44. DOI: [10.1145/2338965.2336758](https://doi.org/10.1145/2338965.2336758).
- Jensen, Simon Holm, Anders Møller, and Peter Thiemann (2009). "Type Analysis for JavaScript". In: *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, pp. 238–255. DOI: [10.1007/978-3-642-03237-0_17](https://doi.org/10.1007/978-3-642-03237-0_17). URL: https://doi.org/10.1007/978-3-642-03237-0_17.
- Jovanovic, Nenad, Christopher Krügel, and Engin Kirda (2006). "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)". In: *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*, pp. 258–263. DOI: [10.1109/SP.2006.29](https://doi.org/10.1109/SP.2006.29).

- Kashyap, Vineeth et al. (2014). "JSAI: a static analysis platform for JavaScript". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pp. 121–132. DOI: [10.1145/2635868.2635904](https://doi.org/10.1145/2635868.2635904). URL: <https://doi.org/10.1145/2635868.2635904>.
- Kim, Hyunha, Kyung-Goo Doh, and David A. Schmidt (2013). "Static Validation of Dynamically Generated HTML Documents Based on Abstract Parsing and Semantic Processing". In: *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pp. 194–214. DOI: [10.1007/978-3-642-38856-9_12](https://doi.org/10.1007/978-3-642-38856-9_12).
- Kneuss, Etienne, Philippe Suter, and Viktor Kuncak (2010). "Phantom: PHP analyzer for type mismatch". In: *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pp. 373–374. DOI: [10.1145/1882291.1882355](https://doi.org/10.1145/1882291.1882355).
- Ko, Yoonseok, Xavier Rival, and Sukyoung Ryu (2019). "Weakly sensitive analysis for JavaScript object-manipulating programs". In: *Softw. Pract. Exp.* 49.5, pp. 840–884. DOI: [10.1002/spe.2676](https://doi.org/10.1002/spe.2676). URL: <https://doi.org/10.1002/spe.2676>.
- Lee, H. et al. (2012). "SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript". In: *FOOL'12*.
- Lin, Anthony Widjaja and Pablo Barceló (2016). "String solving with word equations and transducers: towards a logic for analysing mutation XSS". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pp. 123–136. DOI: [10.1145/2837614.2837641](https://doi.org/10.1145/2837614.2837641).
- Logozzo, Francesco and Manuel Fähndrich (2010). "Pentagons: A weakly relational abstract domain for the efficient validation of array accesses". In: *Sci. Comput. Program.* 75.9, pp. 796–807. DOI: [10.1016/j.scico.2009.04.004](https://doi.org/10.1016/j.scico.2009.04.004).
- Loring, Blake, Duncan Mitchell, and Johannes Kinder (2019). "Sound regular expression semantics for dynamic symbolic execution of JavaScript". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, pp. 425–438. DOI: [10.1145/3314221.3314645](https://doi.org/10.1145/3314221.3314645).
- Maffeis, Sergio, John C. Mitchell, and Ankur Taly (2008). "An Operational Semantics for JavaScript". In: *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, pp. 307–325. DOI: [10.1007/978-3-540-89330-1_22](https://doi.org/10.1007/978-3-540-89330-1_22).
- McNaughton, Robert and Hisao Yamada (1960). "Regular Expressions and State Graphs for Automata". In: *IRE Trans. Electronic Computers* 9.1, pp. 39–47.
- Midtgaard, Jan, Flemming Nielson, and Hanne Riis Nielson (2016). "A Parametric Abstract Domain for Lattice-Valued Regular Expressions". In: *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pp. 338–360. DOI: [10.1007/978-3-662-53413-7_17](https://doi.org/10.1007/978-3-662-53413-7_17).
- Minamide, Yasuhiko (2005). "Static approximation of dynamically generated Web pages". In: *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, pp. 432–441. DOI: [10.1145/1060745.1060809](https://doi.org/10.1145/1060745.1060809).
- Miné, Antoine (2006). "The Octagon Abstract Domain". In: *Higher-Order and Symbolic Computation* 19.1, pp. 31–100.
- (2013). *Static analysis by abstract interpretation of concurrent programs. (Analyse statique par interprétation abstraite de programmes concurrents)*. URL: <https://tel.archives-ouvertes.fr/tel-00903447>.

- Nielson, Flemming, Hanne Riis Nielson, and Chris Hankin (1999). *Principles of program analysis*. Springer. ISBN: 978-3-540-65410-0. DOI: [10.1007/978-3-662-03811-6](https://doi.org/10.1007/978-3-662-03811-6). URL: <https://doi.org/10.1007/978-3-662-03811-6>.
- Park, Changhee, Hyeonseung Im, and Sukyoung Ryu (2016). "Precise and scalable static analysis of jQuery using a regular expression domain". In: *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*, pp. 25–36. DOI: [10.1145/2989225.2989228](https://doi.org/10.1145/2989225.2989228).
- Park, Changhee and Sukyoung Ryu (2015). "Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity". In: *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pp. 735–756. DOI: [10.4230/LIPIcs.ECOOP.2015.735](https://doi.org/10.4230/LIPIcs.ECOOP.2015.735).
- Petrak, Hynek (2018). <https://github.com/HynekPetrak/javascript-malware-collection>. Accessed: 2018-08-17.
- Pradel, Michael and Koushik Sen (2015). "The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript". In: *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pp. 519–541. DOI: [10.4230/LIPIcs.ECOOP.2015.519](https://doi.org/10.4230/LIPIcs.ECOOP.2015.519).
- Preda, Mila Dalla et al. (2008). "A semantics-based approach to malware detection". In: *ACM Trans. Program. Lang. Syst.* 30.5, 25:1–25:54. DOI: [10.1145/1387673.1387674](https://doi.org/10.1145/1387673.1387674).
- Rabin, Michael O. and Dana S. Scott (1959). "Finite Automata and Their Decision Problems". In: *IBM Journal of Research and Development* 3.2, pp. 114–125. DOI: [10.1147/rd.32.0114](https://doi.org/10.1147/rd.32.0114).
- Reynolds, John C. (1998). *Theories of programming languages*. Cambridge University Press. ISBN: 978-0-521-59414-1.
- Richards, Gregor et al. (2011). "The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications". In: *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, pp. 52–78. DOI: [10.1007/978-3-642-22655-7_4](https://doi.org/10.1007/978-3-642-22655-7_4).
- Saxena, Prateek et al. (2010). "A Symbolic Execution Framework for JavaScript". In: *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA*. IEEE Computer Society, pp. 513–528. DOI: [10.1109/SP.2010.38](https://doi.org/10.1109/SP.2010.38).
- Seidl, Helmut, Reinhard Wilhelm, and Sebastian Hack (2012). *Compiler Design - Analysis and Transformation*. Springer.
- Tarski, Alfred (1955). "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific J. Math.* 5.2, pp. 285–309. URL: <https://projecteuclid.org/443/euclid.pjm/1103044538>.
- Thiemann, Peter (2005). "Grammar-based analysis of string expressions". In: *Proceedings of TLDI'05: 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Long Beach, CA, USA, January 10, 2005*, pp. 59–70. DOI: [10.1145/1040294.1040300](https://doi.org/10.1145/1040294.1040300).
- Venet, Arnaud (1999). "Automatic Analysis of Pointer Aliasing for Untyped Programs". In: *Sci. Comput. Program.* 35.2, pp. 223–248.
- Wang, Xinran et al. (2008). "STILL: Exploit Code Detection via Static Taint and Initialization Analyses". In: *Twenty-Fourth Annual Computer Security Applications Conference, ACSAC 2008, Anaheim, California, USA, 8-12 December 2008*, pp. 289–298. DOI: [10.1109/ACSAC.2008.37](https://doi.org/10.1109/ACSAC.2008.37).
- Wilhelm, Reinhard, Shmuel Sagiv, and Thomas W. Reps (2000). "Shape Analysis". In: *Compiler Construction, 9th International Conference, CC 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin*,

- Germany, March 25 - April 2, 2000, *Proceedings*, pp. 1–17. DOI: [10.1007/3-540-46423-9_1](https://doi.org/10.1007/3-540-46423-9_1).
- Xie, Yichen and Alex Aiken (2006). “Static Detection of Security Vulnerabilities in Scripting Languages”. In: *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*. URL: <https://www.usenix.org/conference/15th-usenix-security-symposium/static-detection-security-vulnerabilities-scripting>.
- Xu, Wei, Fangfang Zhang, and Sencun Zhu (2012). “The power of obfuscation techniques in malicious JavaScript code: A measurement study”. In: *7th International Conference on Malicious and Unwanted Software, MALWARE 2012, Fajardo, PR, USA, October 16-18, 2012*, pp. 9–16. DOI: [10.1109/MALWARE.2012.6461002](https://doi.org/10.1109/MALWARE.2012.6461002).
- Yu, Fang, Muath Alkhalaf, and Tefvik Bultan (2011). “Patching vulnerabilities with sanitization synthesis”. In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011*, pp. 251–260. DOI: [10.1145/1985793.1985828](https://doi.org/10.1145/1985793.1985828).
- Yu, Fang et al. (2008). “Symbolic String Verification: An Automata-Based Approach”. In: *Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings*, pp. 306–324. DOI: [10.1007/978-3-540-85114-1_21](https://doi.org/10.1007/978-3-540-85114-1_21). URL: https://doi.org/10.1007/978-3-540-85114-1_21.