박 사 학 위 논 문
Ph.D. Dissertation

# 성장하는 프로그래밍 언어 명세를 위한 자바스크립트 정적 분석

JavaScript Static Analysis
for Evolving Language Specifications

2022

박 지 혁 (朴 智 爀 Park, Jihyeok)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

박 사 학 위 논 문

# 성장하는 프로그래밍 언어 명세를 위한 자바스크립트 정적 분석

2022

박 지 혁

한 국 과 학 기 술 원

전산학부

# 성장하는 프로그래밍 언어 명세를 위한 자바스크립트 정적 분석

박 지 혁

# JavaScript Static Analysis for Evolving Language Specifications

Jihyeok Park

Advisor: Sukyoung Ryu

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Daejeon, Korea
October 1, 2021

Approved by

Sukyoung Ryu
Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

DCS

박지혁. 성장하는 프로그래밍 언어 명세를 위한 자바스크립트 정적 분석. 전산학부 . 2022년. 108+v 쪽. 지도교수: 류석영. (영문 논문)
Jihyeok Park. JavaScript Static Analysis for Evolving Language Specifications. School of Computing . 2022. 108+v pages. Advisor: Sukyoung Ryu. (Text in English)

**초 록**

자바스크립트의 동적인 성질과 복잡한 의미론은 자바스크립트 프로그램들의 행동을 정확하게 이해하기 어렵게 만든다. 연구자들은 이를 해결하고자 다양한 자바스크립트 정적 분석기들을 개발해왔지만, 이들은 모두 수동으로 설계가 되었기에 노동집약적이고 오류에 취약하다. 또한, 2014년 말부터 자바스크립트의 언어 명세가 매년 갱신되기 시작하면서 이 문제는 더 심각해졌다. 본 학위 논문에서는 자바스크립트 언어 명세로부터 자동으로 자바스크립트 정적 분석기를 유도해내는 새로운 방식을 제안한다. 이는 1) 기계화 명세 추출, 2) 명세의 유효성 검사, 그리고 3) 정적 분석기 유도의 세 가지 단계로 구성된다. 본 논문에서는 기계화 명세 추출 기술을 제안하고, 이를 이용해 가장 최신 자바스크립트 명세로부터 기계화 명세를 추출한다. 또한, 자바스크립트 엔진을 이용한 $N$+1-버전 차분 테스팅과 기계화 명세의 타입 분석을 제안하고, 이를 통해 명세 및 엔진의 결함을 검출한다. 최종적으로, 메타 정적 분석 기술을 제안하고, 추출한 기계화 명세로부터 자동으로 자바스크립트 정적 분석기를 유도한다. 본 학위 논문에서 제시한 방식은 프로그래밍 언어를 위한 명세, 테스트, 그리고 도구들의 공진화를 위한 연구의 기틀을 마련할 것으로 기대한다.

**핵 심 낱 말** 자바스크립트, 기계화 명세 추출, $N$+1-버전 차분 테스팅, 명세 타입 분석, 메타 정적 분석

**Abstract**

The highly dynamic nature and complex semantics of JavaScript make it difficult to understand the behaviors of JavaScript programs correctly. To automatically reason about them, researchers have developed various JavaScript static analyzers that conform to ECMAScript, the standard specification of JavaScript. However, all the existing JavaScript static analyzers are manually designed; thus, the current approach is labor-intensive and error-prone. Moreover, since late 2014, this problem has become more critical because the JavaScript language itself rapidly evolves with a yearly release cadence and open development process. This thesis introduces a novel approach to derive JavaScript static analyzers from any version of ECMAScript automatically. Our approach consists of three steps: 1) mechanized specification extraction, 2) specification validity check, and 3) derivation of static analyzers. First, we present a tool JISET, which automatically extracts a mechanized specification from ECMAScript. We show that it successfully extracts a mechanized specification from the latest ECMAScript. Then, we present two different tools to detect bugs in JavaScript specifications; JEST performs $N$+1-version differential testing with JavaScript engines, and JSTAR performs a type analysis for the specification. Finally, we present JSAVER, which automatically derives JavaScript static analyzers from mechanized specifications using a meta-level static analysis. For evaluation, we derived a JavaScript static analyzer from the latest ECMAScript (ES12, 2021). The derived analyzer soundly analyzed all applicable 18,556 official conformance tests with 99.0% of precision in 1.59 seconds on average. We believe that the thesis would be the first step towards the co-evolution of specifications, tests, and tools for programming languages.

**Keywords** JavaScript, mechanized specification extraction, $N$+1-version differential testing, specification type analysis, meta-level static analysis.

# Contents

# List of Tables

# List of Figures

# Chapter 1. Introduction

JavaScript is one of the most widely used programming languages. In the beginning, JavaScript started as a small scripting language for client programming. Nowadays, it has become the de-facto programming language for web development. According to W3Techs[1], 97.7% of websites use JavaScript as their client-side programming language. Moreover, Node.js[2] introduced full-stack JavaScript by supporting server-side programming, and JavaScript has recently begun to be used even in embedded systems such as Espruino[3]. According to the official annual report of GitHub[4], JavaScript has consistently been the most popular programming language based on the number of contributors to GitHub projects.

Despite its popularity, understanding JavaScript program behaviors correctly is challenging due to the highly dynamic nature of the language. For example, we can generate a string value `"es"` by executing the following obfuscated JavaScript code:

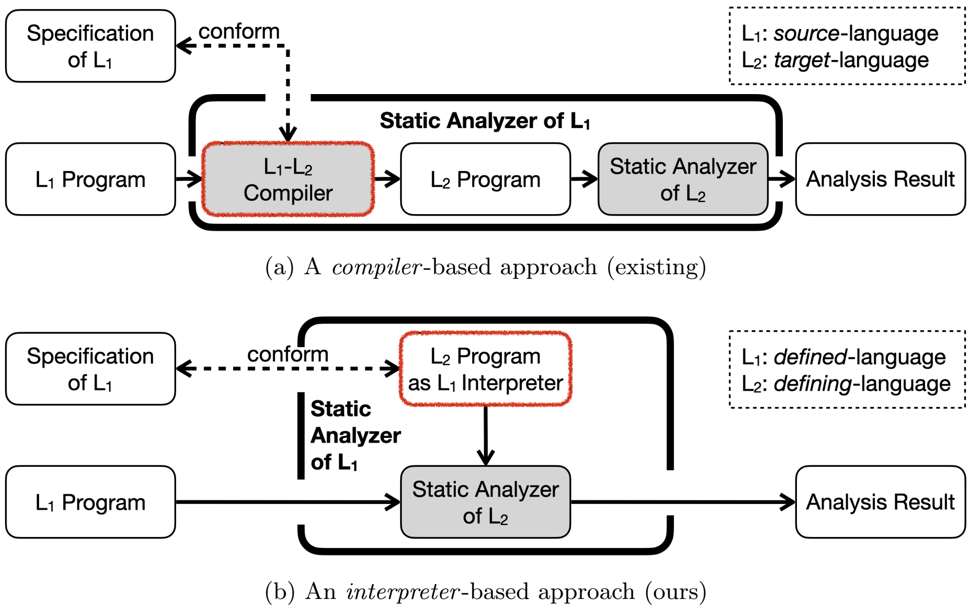
```
let x = (!![]+[])[!+[]+!+[]+!+[]]; // true[3]   == "e"
let y = (![]+[])[!+[]+!+[]+!+[]];  // false[3]  == "s"
let z = x + y;                     // "e" + "s" == "es"
```

In this example, `!![]` and `![]` evaluate to **`true`** and **`false`**, respectively, because of the implicit conversions to boolean values for the negation operator `!`. Then, `+[]` implicitly converts them to strings, and `!+[]+!+[]+!+[]` evaluates to 3. Thus, variables `x` and `y` have strings `"e"` and `"s"` that are the fourth characters of `"true"` and `"false"`, respectively. Finally, the variable `z` has the string `"es"` by concatenating them via the addition operator `+`. This example shows that implicitly converting types and using arbitrary expressions for property accesses make JavaScript behaviors complex.

To reason about their complex behaviors, researchers have developed various JavaScript static analyzers that conform to ECMAScript, the standard specification of JavaScript. ECMAScript describes the semantics of JavaScript language features using algorithms written in a natural language, English. Therefore, existing static analyzers, such as SAFE [56, 77], TAJS [45], WALA [92], and JSAI [48], have their own abstract semantics carefully designed to over-approximate the concrete semantics described in ECMAScript soundly. Various JavaScript static analysis techniques have been presented and implemented on these tools: loop sensitivity [68, 69], advanced string domains [13, 67], analysis based on property relations [53, 54, 66, 92], on-demand backward analysis [93], and combined analysis with dynamic analysis [74, 76, 78, 85, 100].

Existing JavaScript static analyzers take a *compiler*-based approach with intermediate representations (IRs). To reduce the burden of handling numerous language features, most analyzer developers design an IR with a compiler that translates a programming language to its IR to indirectly represent the language semantics [34, 95, 96]. For example, Figure 1.1(a) depicts a compiler-based approach for static analysis of a *source*-language $L_1$ using a static analyzer of a *target*-language $L_2$. It first compiles an $L_1$ program to an $L_2$ program using an $L_1$-$L_2$ compiler that conforms to the semantics described in the specification of $L_1$. Then, it analyzes the compiled $L_2$ program using a static analyzer of $L_2$. For a JavaScript static analyzer, JavaScript and its own IR are $L_1$ and $L_2$, respectively.

[1]https://w3techs.com/technologies/details/cp-javascript/all/all
[2]https://nodejs.org/
[3]https://www.espruino.com/
[4]https://octoverse.github.com/



(a) A *compiler*-based approach (existing)

(b) An *interpreter*-based approach (ours)

Figure 1.1: Two approaches of static analysis for a language $L_1$ using a static analyzer of a language $L_2$

However, static analyzers with the compiler-based approach are unable to keep up with fast-evolving JavaScript because they require *manual updates* for new language semantics. The JavaScript language itself is rapidly evolving nowadays. Since 2015, the Ecma Technical Committee 39 (TC39) has maintained the specification as an open-source GitHub project and released its official versions annually. The specification size has been getting bigger as well, and the latest version of ECMAScript (ES12, 2021) [2] is 879 pages. Because existing JavaScript static analyzers cannot update JavaScript-IR compilers automatically, they still focus on ES5.1 and only support a few ES6 features manually, even though six more versions from ES7 to ES12 have been released since 2015. Because many recent JavaScript programs frequently use new language features like `let` bindings, arrow functions, generators, and promises, the lack of new feature support grows increasingly problematic over time.

This thesis introduces a novel technique to automatically derive a JavaScript static analyzer from any version of ECMAScript. The main idea is to shift the paradigm from *compiler*-based approaches to *interpreter*-based approaches to utilize "the interpreter-based nature" of JavaScript. The history of JavaScript [101] testifies that the initial working group designing JavaScript in the 1990s defined the semantics using reference interpreters:

> Guy Steele would ask a question about some edge-case feature behavior. [...] they would each turn to their respective implementation and try a test case. If they got the same answer, that became the specified behavior.

The interpreter-based nature also affects the writing style of the language specifications. ECMAScript describes the language semantics with pseudocode algorithms consisting of sequentially numbered steps to represent program executions. To fully utilize this interpreter-based nature of JavaScript, our approach derives a static analyzer by:

1. Extracting a *mechanized specification* from ECMAScript as a *definitional interpreter*.

2. Checking the *validity* of the extracted mechanized specification.

3. Performing a *meta-level static analysis* with the extracted mechanized specification.

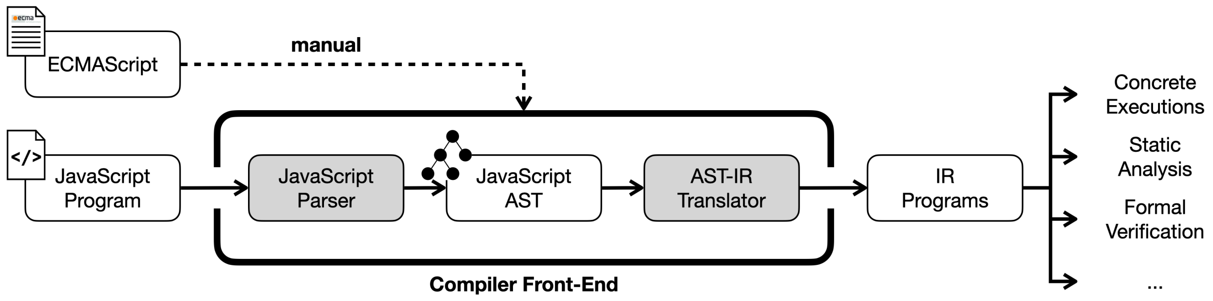


Figure 1.2: A compiler front-end from JavaScript to an IR with ECMAScript

## 1.1 Mechanized Specification Extraction

Researchers have defined various JavaScript formal semantics [16, 31, 36, 70] suitable for static analysis [45, 48, 56, 92] and formal verification [31] by referring to ECMAScript, the standard specification of JavaScript. ECMAScript describes the JavaScript syntax using a variant of the extended Backus–Naur form (EBNF) notation and its semantics using abstract algorithms written in English in a clear and structured manner. A traditional way to define the formal semantics of JavaScript is to build a *compiler front-end* that takes JavaScript programs and produces programs in Intermediate Representations (IRs) to represent the semantics of the given programs indirectly. As illustrated in Figure 1.2, the compiler front-end consists of 1) a parser that constructs Abstract Syntax Trees (ASTs) of given programs and 2) an AST-IR translator that converts ASTs to their own IRs. It helps researchers focus on IRs without worrying about the diverse and enormous features in JavaScript in developing new techniques for static analysis and formal verification.

However, such compiler-based approaches require a manual update of the compiler front-end when ECMAScript is updated. Although the manual update was reasonable until ES5.1, it is too tedious, labor-intensive, and error-prone to deal with large sizes of ES6 and later versions. ES6 introduced numerous new features such as lexical binding (**`let`**), the spread operator (...), classes (**`class`**), the **`for-of`** operator, the **`async`** functions, and generators. For example, consider KJS [70], one of the formal semantics of ES5.1 defined on top of $\mathbb{K}$ [84], which is a framework to define language semantics. According to an author of KJS, it took *four months* to implement an AST-IR translator for 1,370 steps out of 2,932 steps in 368 abstract algorithms[5]. However, ES12, the most recent version, has 2,640 abstract algorithms consisting of 13,544 steps. Thus, the manual approaches do not seem to be scalable enough to build an AST-IR translator for recent versions of ECMAScript, and indeed no formal semantics exists for ES6 to ES12. Moreover, roughly 1,000 to 3,000 steps of abstract algorithms have been modified or newly added in each annual update of ECMAScript. To handle these frequent and massive updates of ECMAScript, researchers should manually update parsers and AST-IR translators, which incurs tremendous efforts.

To alleviate this problem, we introduce JISET, a JavaScript IR-based Semantics Extraction Toolchain. Instead of the compiler-based approach, we introduce an interpreter-based approach to represent JavaScript semantics formally. A *definitional interpreter* provides a way to represent the language semantics of a *defined*-language using its interpreter written in a *defined*-language. JISET is the first tool that automatically extracts a mechanized specification from ECMAScript as a definitional interpreter of JavaScript. We introduce $\text{IR}_{\text{ES}}$, a specialized Intermediate Representation for ECMAScript, to utilize it as the defining-language and JavaScript as the defined-language of the extracted definitional interpreters.

[5]`https://daejunpark.github.io/2015-06-16-park-stefanescu-rosu-PLDI.pdf`

For a given version of ECMAScript, JISET automatically 1) generates parsers for syntax and 2) compiles abstract algorithms to functions of $\mathrm{IR}_{\mathrm{ES}}$ for semantics. Then, JISET produces a JavaScript definitional interpreter strictly conforming to the given version of ECMAScript based on the compiled $\mathrm{IR}_{\mathrm{ES}}$ functions. There are several technical challenges in generating parsers and compiling abstract algorithms. To represent JavaScript syntax, ECMAScript utilizes its own variant of EBNF with several new notations: parametric non-terminals, conditional alternatives, and even context-sensitive symbols. Thus, no existing parser generation technique for context-free grammars (CFGs) is directly applicable to this variant. JavaScript even supports automatic semicolon insertion with several complex rules in its parsing algorithm. However, the most critical problem is that ECMAScript describes JavaScript language semantics using abstract algorithms written in English. Besides, a general representation of abstract algorithms is necessary to support future versions of ECMAScript.

## 1.2 Specification Validity Check

In Peter O'Hearn's keynote speech in ICSE 2020, he quoted the following from Mark Zuckerberg's Letter to Investors [1]:

> The Hacker Way is an approach to building that involves continuous improvement and iteration. Hackers believe that something can always be better, and that nothing is ever complete.

Indeed, modern programming follows the continuous integration (CI) and continuous deployment (CD) approach [10] rather than the traditional waterfall model. Instead of a sequential model that divides software development into several phases, each of which takes time, CI/CD amounts to a cycle of quick software development, deployment, and back to development with feedback. Even the development of programming languages uses the CI/CD approach.

For example, various JavaScript engines provide diverse extensions to adapt to fast-changing user demands. At the same time, ECMAScript, the official specification that describes the syntax and semantics of JavaScript, is annually updated since ES6 to support new features in response to user demands. Such updates in both the specification and implementations in tandem make it challenging to sync them. Another example is Solidity [7], the standard smart contract programming language for the Ethereum blockchain. The Solidity language specification is continuously updated, and the Solidity compiler is also frequently released. According to Hwang and Ryu [43], the average number of days between consecutive releases from Solidity 0.1.2 to 0.5.7 is 27. In most cases, the Solidity compiler reflects updates in the specification. However, even the specification is revised according to the semantics implemented in the compiler. As in JavaScript, bidirectional effects in the specification and the implementation make it hard to guarantee their correspondence.

In this approach, both the specification and implementation may contain bugs, and it is challenging to guarantee their correctness. The conventional approach for building a programming language is unidirectional, from a language specification to its implementation. Language designers believe that the specification is correct and check the conformance of an implementation to the specification via dynamic testing. Unlike in the conventional approach, in the modern CI/CD approach, the specification may not be the oracle because both the specification and the implementation can co-evolve.

The correctness of ECMAScript is critical because an incorrect description in the specification can lead to wrong implementations of JavaScript engines in various fields. However, all the specification updates are currently manually reviewed by TC39 without any automated tools. This manual review process is inherently labor-intensive and error-prone, making ECMAScript vulnerable to specification bugs. Besides, the yearly release cadence and open development process of ECMAScript make this problem more challenging. In the official ECMAScript repository[6], 1,475 pull requests and 2,203 commits exist in the master branch. Therefore, manually checking all the frequent specification updates is a challenging task.

Unfortunately, no existing tools can automatically detect bugs in rapidly evolving JavaScript specifications written in English. Thus, the ECMAScript committee has pursued various manual annotations in abstract algorithms to reduce specification bugs. First, the committee has introduced two kinds of annotations: 1) *assertions* to denote assumptions at specific points of abstract algorithms and 2) two *prefixes* ? and ! to represent whether the execution of an abstract algorithm completes abruptly or not. For example, consider the following two algorithm steps:

1. Assert: Type($O$) is Object.

2. ? GetV($V$,$P$)

The first step denotes that the variable $O$ always has a JavaScript object value at the point of the assertion. The second one denotes that the execution of GetV($V$,$P$) may complete abruptly. Such annotations help readers understand specifications clearly, and they are also helpful for specification-based tools[7] such as JavaScript engines [4, 5, 8, 9], debuggers [19], static analyzers [45, 48, 56, 92], and verification tools [31, 32]. Second, the committee has started internal discussions on type annotations for variables, parameters, and return values of abstract algorithms[8]. However, manual annotations are labor-intensive and error-prone, and they do not provide any automatic mechanism to detect specification bugs.

In this thesis, we introduce two different ways to check the validity of ECMAScript by 1) checking the conformance with JavaScript engines via $N$+1-version differential testing and 2) performing type analysis for ECMAScript.

### 1.2.1 $N$+1-version Differential Testing

To check the conformance between ECMAScript and JavaScript engines, we propose a novel *$N$+1-version differential testing*, which enables testing of co-evolving specifications and their implementations. The differential testing [59] is a testing technique, which executes $N$ implementations of a specification concurrently for each input, and detects a problem when the outputs are in disagreement. In addition to $N$ implementations, our approach tests the specification as well using a mechanized specification. Recently, several approaches to extract syntax and semantics directly from language specifications have been presented [65, 75, 97]. We utilize them to bridge the gap between specifications and their implementations through conformance tests generated from mechanized specifications. The $N$+1-version differential testing consists of three steps: 1) to automatically synthesize programs guided by the syntax and semantics from a given language specification, 2) to generate conformance tests by injecting assertions to the synthesized programs to check their final program states, 3) to detect bugs in the specification and implementations via executing the conformance tests on multiple implementations, and 4) to localize bugs on the specification using statistical information.

[6]https://github.com/tc39/ecma262
[7]https://github.com/tc39/ecmarkup/issues/173
[8]https://github.com/tc39/ecma262/pull/545#issuecomment-559292107

Given a language specification and $N$ existing real-world implementations of the specification, we automatically generate a conformance test suite from the specification with assertions in each test code to make sure that the result of running the code conforms to the specification semantics. Then, we run the test suite for $N$ implementations of the specification. Because generated tests strictly comply with the specification, they reflect specification errors as well, if any. When one of the implementations fails in running a test, the implementation may have a bug, as in the differential testing. When most of the implementations fail in running a test, it is highly likely that the specification has a bug. By automatically generating a rich set of test code from the specification and running them with implementations of the specification, we can find and localize bugs either in the specification written in a natural language or in its implementations.

To show the practicality of the proposed approach, we present JEST, which is a JavaScript Engines and Specification Tester using $N+1$-version differential testing. We implement JEST by extending JISET to utilize the syntax and semantics automatically extracted from ECMAScript. Our tool automatically synthesizes initial seed programs based on the extracted syntax and expands the program pool by mutating specific target programs guided by semantics coverage. Then, the tool generates conformance tests by injecting assertions to synthesized programs. Finally, JEST detects and localizes bugs using execution results of the tests on $N$ JavaScript engines. We evaluate our tool with four JavaScript engines (Google V8 [9], GraalJS [4], QuickJS [8], and Moddable XS [5]) that support all core JavaScript language features in the latest ECMAScript, ES12.

### 1.2.2 Type Analysis for ECMAScript

To check the validity of ECMAScript, we also present a novel tool JSTAR, a JavaScript Specification Type Analyzer using Refinement. The main challenge of ECMAScript type analysis to statically detect type-related specification bugs automatically is that ECMAScript describes abstract algorithms in a natural language, English. While researchers [16, 31, 36, 70] have formally defined various JavaScript semantics for different versions of ECMAScript by hand, manual formalization is not suitable for automatically detecting bugs in rapidly evolving JavaScript specifications. Thus, recent approaches in diverse fields such as system architectures [65, 97], network protocols [49], and language specifications [86, 108] have utilized information directly extracted from specifications written in a natural language to lessen such burdens. Among them, JISET [75] compiles ECMAScript abstract algorithms written in a structured natural language to $\text{IR}_{\text{ES}}$ functions. Therefore, JSTAR leverages JISET to handle JavaScript specifications mechanically.

JSTAR takes mechanized JavaScript specifications from JISET and performs a type analysis of compiled functions using *specification types* defined in ECMAScript. ECMAScript contains not only JavaScript language types but also specification types such as abstract syntax trees (ASTs), internal list-like structures, and internal records including environments, completions, and property descriptors. We define their type hierarchies based on subtype relations. For records and AST types, we also define their fields. Using such type information, JSTAR performs a type analysis and detects specification bugs using a *bug detector* consisting of four checkers: 1) a reference checker, 2) an arity checker, 3) an assertion checker, and 4) an operand checker. JSTAR also uses a *condition-based refinement* for type analysis, which prunes out infeasible parts in abstract states by using conditions of assertions and branches to improve the precision of type analysis. We evaluated JSTAR with all 864 versions in the official ECMAScript repository for the recent three years from ES9 to ES12. The experiments showed that the refinement technique could reduce the number of false-positive bugs caused by imprecise type analysis.

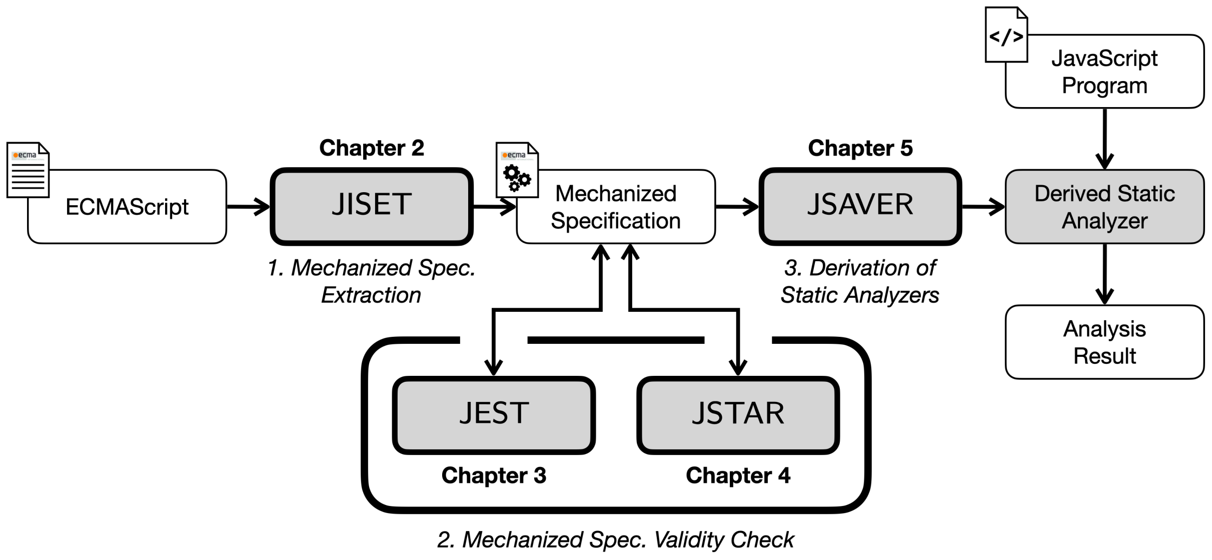


Figure 1.3: Overall structure to automatically derive a JavaScript static analyzer from ECMAScript

## 1.3 Derivation of Static Analyzers

To automatically derive a JavaScript static analyzer from any version of ECMAScript, we present a *meta-level static analysis* to analyze JavaScript programs indirectly using JavaScript definitional interpreters. A meta-level static analysis is an interpreter-based approach for static analysis of a *defined*-language $L_1$ using a static analyzer of a *defining*-language $L_2$ as depicted in Figure 1.1(b). Since an $L_1$ interpreter is an $L_2$ program, it indirectly analyzes an $L_1$ program by analyzing the interpreter using a static analyzer of $L_2$ with the $L_1$ program as the input. Thus, we develop JSAVER, a JavaScript Static Analyzer via ECMAScript Representations. It is a static analyzer of $\mathrm{IR}_{\mathrm{ES}}$ for a meta-level static analysis for JavaScript. For its expressiveness power, we also present ways to indirectly configure *abstract domains* and *analysis sensitivities* for JavaScript in the static analysis of $\mathrm{IR}_{\mathrm{ES}}$. First, we provide a method to configure abstract domains for JavaScript values and structures. Second, we present the *AST sensitivity* to express analysis sensitivities for JavaScript, such as flow-sensitivity and $k$-callsite-sensitivity.

Finally, we experimentally showed that JSAVER could effectively analyze JavaScript programs by indirectly analyzing the JavaScirpt definitional interpreter written in $\mathrm{IR}_{\mathrm{ES}}$ with the programs. We derived a static analyzer $\mathsf{JSA}_{\mathsf{ES12}}$ from the latest ECMAScript, ES12. The derived analyzer $\mathsf{JSA}_{\mathsf{ES12}}$ soundly analyzes all applicable 18,556 official conformance tests with 99.0% of precision in 1.59 seconds on average. Moreover, we demonstrate the configurability and adaptability of JSAVER with several case studies.

## 1.4 Overview

Figure 1.3 depicts the overall structure that automatically derives a JavaScript static analyzer from a given version of ECMAScript. In the remainder of this thesis, we explain how to extract a mechanized specification from ECMAScript (Chapter 2), how to check its validity using $N+1$-version differential testing (Chapter 3) and type analysis (Chapter 4), and how to derive a static analyzer from the mechanized specification (Chapter 5). After evaluating each tool, we discuss related work (Section 6) and conclude (Section 7).

# Chapter 2. JISET: Mechanized Specification Extraction

This chapter introduces JISET, a JavaScript IR-based Semantics Extraction Toolchain. The main contribution of the tool is as follows:

- **JISET is the *first tool that automatically extracts* mechanized specification from a language specification, ECMAScript, as a JavaScript definitional interpreter.** For syntax, we formally introduce $\text{BNF}_{\text{ES}}$ as a variant of EBNF for ECMAScript and propose a parser generation technique with lookahead parsing for $\text{BNF}_{\text{ES}}$, which supports automatic semicolon insertion. For semantics, we present $\text{IR}_{\text{ES}}$ to utilize it as a defining-language of the JavaScript definitional interpreters and propose a compilation from abstract algorithms to $\text{IR}_{\text{ES}}$ functions. The compilation is assisted by compile rules which describe how to compile each step of abstract algorithms into $\text{IR}_{\text{ES}}$ instructions. We evaluated JISET with the six most recent ECMAScript versions (ES7 to ES12). JISET successfully generated parsers for all versions and compiled 95.61% of the steps in abstract algorithms on average.

- **JISET bridges gaps between the specification written in a natural language and tests.** We manually completed missing parts in the compiled IRES functions for the ECMAScript 2019 (ES10) because we conducted this research in 2019. It was the *first mechanized specification of JavaScript* after the big update of ECMAScript in 2015. It failed for 1,709 tests because of specification errors in ES10. We found eight specification errors using the tests; three errors had not been reported before. They were all confirmed by TC39 and fixed in the next release, ES11. After fixing them, the mechanized semantics passed all 18,064 applicable tests for ES10.

- **JISET is also *adaptable* to new language features proposed for future ECMAScript specifications.** We evaluated the forward compatibility of JISET by applying it to proposals for new language features not yet included in ES10. When we conducted this research at the end of 2019, nine proposals were ready for inclusion in the next ECMAScript (ES11, 2020), and we applied JISET to all of them. It automatically synthesized parsers and compiled 560 out of 595 algorithm steps for all the proposals. After completing the missing parts, we found three specifications errors in the BigInt proposal by executing the corresponding tests in Test262. After fixing them, the extracted semantics passed all applicable ES10 tests and 303 new applicable tests. Moreover, we also applied to ES12 to check that JISET still works for the latest ECMAScript. After completing the missing parts, it passed all 18,556 applicable conformance tests for ES12.

## 2.1 Overview

In this section, we introduce the overall structure of JISET depicted in Figure 2.1. Compared to the compiler-based approach with a compiler-front end shown in Figure 1.2, our tool automatically generates JavaScript Parser and JavaScript Interpreter in $\text{IR}_{\text{ES}}$ directly from ECMAScript. The motivation of this work is twofold: 1) ECMAScript is written in a well-organized style, and 2) the writing style was converged since ES7 in 2016. Therefore, JISET first extracts the syntax and semantics extracted from ECMAScript via Spec Extractor in JSON format. Then, it utilizes them with the common patterns in the writing style to automatically generate JavaScript Parser and JavaScript Interpreter.

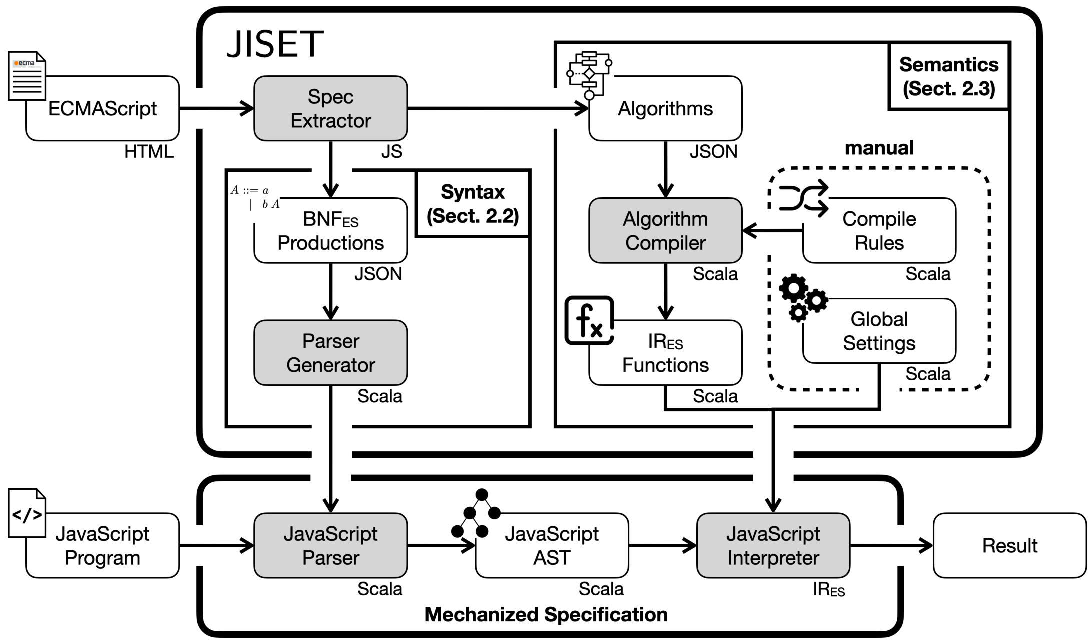


Figure 2.1: Overall structure of JISET: Automatic extraction of a mechanized specification consisting of JavaScript Parser and JavaScript Interpreter

**Syntax** ECMAScript provides JavaScript language syntax with lexical and syntactic productions of a variant of EBNF for ECMAScript in Appendix A[1]. We dub it $\text{BNF}_{\text{ES}}$ and formally define it in Section 2.2. Spec Extractor reads the $\text{BNF}_{\text{ES}}$ productions and converts them into JSON files. For example, Figure 2.2(a) shows the *ArrayLiteral* production in ES12. It takes two boolean parameters Yield and Await and has three alternatives. The first alternative consists of three symbols: two terminal symbols `[` and `]`, and one non-terminal symbol $\textit{Elision}_{\text{opt}}$. The opt subscript denotes that it is optional. In the second and third alternatives, $\textit{ElementList}_{\text{[?Yield, ?Await]}}$ denotes a parametric non-terminal symbol *ElementList* with the parameters Yield and Await of *ArrayLiteral* as its two arguments. The prefix ? of a symbol denotes that the symbol is passed as an argument.

To generate JavaScript Parser from given $\text{BNF}_{\text{ES}}$ productions, we construct Parser Generator in Scala using Scala parser combinators [63]. To parse $\text{BNF}_{\text{ES}}$ productions correctly and efficiently, we propose *lookahead parsers*, which keep track of *lookaheads*, sets of possible next tokens. With lookahead parsing, generated parsers have one-to-one mapping to their corresponding productions, improving readability. For example, Figure 2.2(b) shows the generated parser for the *ArrayLiteral* production in Figure 2.2(a). Each parser has the `List[Boolean] => LAParser[T]` type because each production in $\text{BNF}_{\text{ES}}$ is parametric with boolean values. The `memo` is a memoization function for pairs of boolean parameters and resulting parsers for performance optimization. The value `ArrayLiteral` corresponds to the *ArrayLiteral* production. In the parser, each string literal such as `"["` or `"]"` denotes a parser for a terminal symbol. The opt helper function creates optional parsers. The parametric non-terminal *ElementList* with arguments Yield and Await is represented as a function call `ElementList(Yield, Await)`. The `~` operator combines two parsers and the `^^` operator describes how to construct ASTs. When the left-hand side of `^^` is matched, its right-hand side shows a corresponding AST constructor, where the name of each constructor has a number denoting the order among alternatives. For example, the `ArrayLiteral0` constructor corresponds to the first alternative of the *ArrayLiteral* production.

[1]`https://262.ecma-international.org/#sec-grammar-summary`

*ArrayLiteral*`[Yield, Await]` **:**

  **[** *Elision*`opt` **]**

  **[** *ElementList*`[?Yield, ?Await]` **]**

  **[** *ElementList*`[?Yield, ?Await]` **,** *Elision*`opt` **]**

(a) *ArrayLiteral* production in ES12

```
val ArrayLiteral: List[Boolean] => LAParser[T] = memo {
  case List(Yield, Await) => {
    "[" ~ opt(Elision) ~ "]"                                ^^ ArrayLiteral0 |
    "[" ~ ElementList(Yield,Await) ~ "]"                    ^^ ArrayLiteral1 |
    "[" ~ ElementList(Yield,Await) ~ "," ~ opt(Elision)~ "]" ^^ ArrayLiteral2
  }
}
```

(b) The generated parser for the *ArrayLiteral* production

Figure 2.2: The *ArrayLiteral* production in ES12 and its parser

**Semantics** ECMAScript describes the language semantics as abstract algorithms in English. While they are written in a natural language, the writing style is well-organized with ordered steps and tagged tokens. Spec Extractor reads abstract algorithms with HTML tags and converts them into JSON files. For example, Figure 2.3(a) presents the Evaluation abstract algorithm of the third alternative of the *ArrayLiteral* production in ES12, and it has seven steps. Each non-terminal symbol (e.g. *ElementList*) or local variable (e.g. *array*) has the `<nt>` and `<var>` HTML tag, respectively.

To translate such abstract algorithms into a suitable form for manipulation, we define $\text{IR}_{\text{ES}}$, a specialized intermediate representation for ECMAScript. Then, we develop Algorithm Compiler in Scala using Scala parser combinators again to compile given abstract algorithms to $\text{IR}_{\text{ES}}$ functions. It also takes Compile Rules as another input, which has two parts: parsing rules and conversion rules. We manually establish the compile rules to cover English sentences used in abstract algorithms as much as possible. Algorithm Compiler utilizes the compile rules to compile each algorithm step to the corresponding $\text{IR}_{\text{ES}}$ instruction. For example, Figure 2.3(b) presents the generated $\text{IR}_{\text{ES}}$ function for the Evaluation abstract algorithm shown in Figure 2.3(a). Note that JavaScript ASTs are also $\text{IR}_{\text{ES}}$ values. The `ArrayLiteral[2].Evaluation` function takes JavaScript ASTs as arguments in two parameters `ElementList` and `Elision`$_{\text{opt}}$ that represent two non-terminal symbols *ElementList* and *Elision*, respectively. Moreover, $\text{IR}_{\text{ES}}$ handle not only JavaScript ASTs or values (e.g. objects, `undefined`, **`null`**, numbers, strings, and booleans) but also `absent`, completion records, constants, closures, and continuations only used in the specification. For example, the parameter `Elision` has a special value `absent` when the non-terminal symbol *Elision*$_{\text{opt}}$ is not present. Thus, Algorithm Compiler compiles the condition in step 4, "If *Elision* is present," into the negation of the equality check with `absent`: `if (= Elision absent)`.

Finally, JISET constructs JavaScript Interpreter with the compiled $\text{IR}_{\text{ES}}$ functions and manually specified Global Setting, which contains minor but necessary data to execute JavaScript programs as described in ECMAScript, including the structure of the standard built-in objects and ECMAScript data types. Putting them all together, we can parse and execute JavaScript programs. Even though JISET is not fully automatic because of Compile Rules and Global Setting, it could dramatically reduce the efforts to build parsers and interpreters from scratch.

**13.2.5.2 Runtime Semantics: Evaluation**

*ArrayLiteral* **:** **[** *ElementList* **,** *Elision*opt **]**

```
1. Let array be ! ArrayCreate(0).
2. Let nextIndex be the result of performing ArrayAccumulation
   for ElementList with arguments array and 0.
3. ReturnIfAbrupt(nextIndex).
4. If Elision is present, then
    a. Let len be the result of performing ArrayAccumulation
       for Elision with arguments array and nextIndex.
    b. ReturnIfAbrupt(len).
5. Return array.
```

(a) The Evaluation abstract algorithm for the third alternative

```
syntax def ArrayLiteral[2].Evaluation (ElementList, Elision) {
  let array = [! (ArrayCreate 0)]
  let nextIndex = (ElementList.ArrayAccumulation array 0)
  [? nextIndex]
  if (! (= Elision absent)) {
    let len = (ElementList.ArrayAccumulation array nextIndex)
    [? len]
  }
  return array
}
```

(b) The compiled $\text{IR}_{\text{ES}}$ function

Figure 2.3: The Evaluation abstract algorithm for the third alternative of *ArrayLiteral* in ES12 and its compiled $\text{IR}_{\text{ES}}$ function

In the remainder of this chapter, we explain the details of how to generate parsers (Section 2.2) and how to compile abstract algorithms to $\text{IR}_{\text{ES}}$ functions (Section 2.3). Then, we evaluate JISET to check its coverage, correctness, and adaptability (Section 2.4).

## 2.2 Parser Generator

In this section, we explain how to generate JavaScript parsers using a given ECMAScript.

### 2.2.1 $\text{BNF}_{\text{ES}}$: An Extended Backus-Naur Form (EBNF) for ECMAScript

ECMAScript describes the JavaScript syntax using a variant of the extended BNF. We dub it $\text{BNF}_{\text{ES}}$ and formally define its notation. It consists of a number of *productions* with the following form:

$$A(p_1, \cdots, p_k) ::= (c_1 \Rightarrow)^? \alpha_1 \mid \cdots \mid (c_n \Rightarrow)^? \alpha_n$$

The left-hand side of ::= represents a parametric non-terminal $A$ with multiple boolean parameters $p_1, \cdots, p_k$. If a non-terminal takes no parameter, parentheses are omitted for brevity. A production has multiple alternatives separated by | with optional conditions. A condition $c$ is either a boolean parameter $p$ or its negation $!p$. An alternative $\alpha$ is a sequence of symbols, where a symbol $s$ is one of the following:

- $\epsilon$: the empty sequence, which passes without any conditions
- $\texttt{a}$: a terminal, which is any token
- $A(a_1, \cdots, a_k)$: a non-terminal, which takes multiple arguments where each argument $a_i$ is either a boolean value `#t` or `#f`, or a parameter $p_i$
- $s?$: option, which is the same with $s \mid \epsilon$
- $+s$ $(-s)$ : positive (negative) lookahead, which checks whether $s$ succeeds (fails) and *never consumes any input*
- $s \setminus s'$: exclusion, which first checks whether $s$ succeeds and then checks whether the parsing result does not correspond to $s'$
- $\langle \neg \texttt{LT} \rangle$: no line-terminator, which is a special symbol that restricts the white spaces between two different symbols

For example, consider the following production:

$$A(p) ::= p \Rightarrow \texttt{a} \mid !p \Rightarrow \texttt{b} \mid \texttt{c}$$

Then, $A(\texttt{\#t})$ means $\texttt{a} \mid \texttt{c}$ and $A(\texttt{\#f})$ means $\texttt{b} \mid \texttt{c}$.

### 2.2.2 Lookahead Parsing

To support $\text{BNF}_{\text{ES}}$ correctly, we extend PEG-based parser generation techniques with lookahead parsing.

**Background: Parsing Expression Grammar**

Most parser generators target context-free languages with specific parsing algorithms for Context-Free Grammar (CFG): JavaCC with LL(k) [12], Bison with GLR [94], and ANTLR with ALL(*) [80]. However, they are not directly applicable for the ECMAScript syntax because ECMAScript lexical and syntactic grammars require context-sensitive lexers and parsers:

- **Context-sensitive tokens:** ECMAScript tokens are context-sensitive because of JavaScript regular expressions and template strings. For example, `/x/g` could be a single regular expression token or four tokens that represent division by variables `x` and `g` depending on enclosing contexts. Thus, lexers should be evaluated during parsing, not before parsing.
- **Context-sensitive $\text{BNF}_{\text{ES}}$ symbols:** $\text{BNF}_{\text{ES}}$ supports context-sensitive symbols, which are positive (negative) lookahead $+s$ $(-s)$, exclusion $s \setminus s'$, and no line-terminator $\langle \neg \texttt{LT} \rangle$. They are highly expressive and even represent the classic non-context-free language $\{a^n b^n c^n : n \geq 1\}$ with the following productions:

$$\begin{array}{ll} S ::= +(X\ \texttt{c})\ A\ Y & X ::= \texttt{a}\ X?\ \texttt{b} \\ A ::= \texttt{a}\ A? & Y ::= \texttt{b}\ Y?\ \texttt{c} \end{array}$$

However, it is not trivial to support such $\text{BNF}_{\text{ES}}$ symbols in CFG-based parser generators.

Unlike CFG-based parser generators, parser generators based on *Parsing Expression Grammar (PEG)* [30] can easily resolve these problems. PEGs are defined with a top-down (LL-style) recursive descent parser with *backtracking*. It visits each alternative of a production in order and backtracks to its previous production when parsing fails. PEG-based parser generators treat lexers as parsers; thus, we can use appropriate lexers depending on parsing contexts. Moreover, PEGs support *and-predicate* (&) and *not-predicate* (!) operators that denote the same meaning of the positive and negative lookahead symbols in $\text{BNF}_{\text{ES}}$, respectively. Therefore, we can easily support context-sensitive tokens and $\text{BNF}_{\text{ES}}$ symbols in PEG-based parser generators.

**Problem: Prioritized Choices.**

While PEG-based parser generators support the context-sensitivity, PEGs have one fundamental difference with $\text{BNF}_{\text{ES}}$: *prioritized choices*. PEGs use the prioritized choice operator '/' instead of the unordered pipe operator '|' in $\text{BNF}_{\text{ES}}$; even when multiple alternatives are applicable, PEGs always pick the first successful alternative. For example, consider the following $\text{BNF}_{\text{ES}}$:

$$\begin{aligned} S &::= E \texttt{ + } E \\ E &::= \texttt{x} \mid \texttt{x.p} \end{aligned} \tag{2.1}$$

As expected, this grammar accepts the string `x+x.p`. However, the following PEG:

$$\begin{aligned} S &::= E \texttt{ + } E \\ E &::= \texttt{x} \ / \ \texttt{x.p} \end{aligned} \tag{2.2}$$

does not accept the same string `x+x.p`. Because the first alternative `x` of $E$ is chosen whenever an input string starts with `x`, the second alternative `x.p` of $E$ is always unreachable. A simple solution to accept the string is just to change the order of alternatives of $E$ like $E ::= \texttt{x.p} \ / \ \texttt{x}$.

Unfortunately, simple reordering is not a general solution for all cases. Consider the following $\text{BNF}_{\text{ES}}$:

$$\begin{aligned} S &::= A \ \texttt{b} \\ A &::= \texttt{a} \mid \texttt{ab} \end{aligned} \tag{2.3}$$

It accepts both strings `ab` and `abb`. However, the following PEG:

$$\begin{aligned} S &::= A \ \texttt{b} \\ A &::= \texttt{a} \ / \ \texttt{ab} \end{aligned} \tag{2.4}$$

accepts only `ab`, and another PEG with reordered productions as follows:

$$\begin{aligned} S &::= A \ \texttt{b} \\ A &::= \texttt{ab} \ / \ \texttt{a} \end{aligned} \tag{2.5}$$

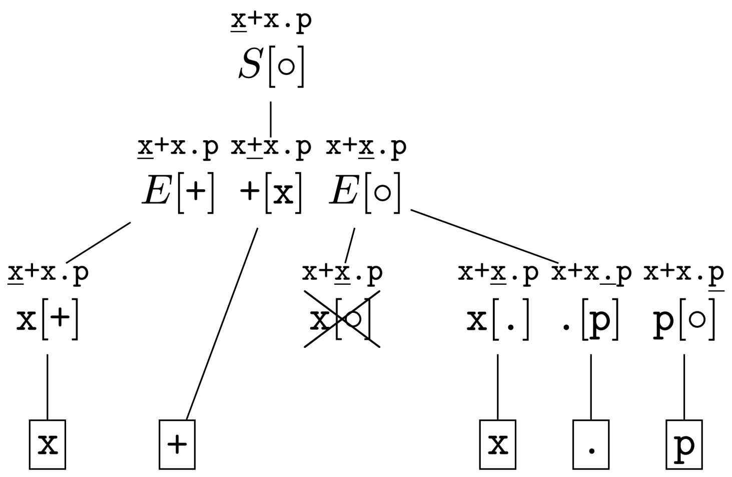
accepts only `abb`.

**Solution: Lookahead Tokens**

To alleviate the problem, we propose *lookahead parsing*, which is an extended parsing algorithm for PEGs with *lookahead tokens*. The key idea of lookahead parsing is to keep track of the next possible tokens by statically calculating a set of first tokens for each symbol using the algorithm in Figure 2.4. For example, the following steps explain how to utilize lookahead tokens during parsing of the string `x+x.p` with the PEG in Equation (2.2):

$$
\begin{aligned}
\textbf{first}_\alpha(s_1 \cdots s_n) &= \textbf{first}_s(s_1) :\!+\ \textbf{first}_\alpha(s_2 \cdots s_n) \\
&\quad \text{where } x :\!+\ y = \begin{cases} x \cup y & \text{if } \circ \in x \\ x & \text{otherwise} \end{cases} \\
\textbf{first}_s(\epsilon) &= \{\circ\} \\
\textbf{first}_s(\mathsf{a}) &= \{\mathsf{a}\} \\
\textbf{first}_s(A(a_1, \cdots, a_k)) &= \textbf{first}_\alpha(\alpha_1) \cup \cdots \cup \textbf{first}_\alpha(\alpha_n) \\
&\quad \text{where } A(a_1, \cdots, a_k) = \alpha_1 \mid \cdots \mid \alpha_n \\
\textbf{first}_s(s?) &= \textbf{first}_s(s) \cup \{\circ\} \\
\textbf{first}_s(+s) &= \textbf{first}_s(s) \\
\textbf{first}_s(-s) &= \{\circ\} \\
\textbf{first}_s(s \setminus s') &= \textbf{first}_s(s) \\
\textbf{first}_s(\langle\neg\mathsf{LT}\rangle) &= \{\circ\}
\end{aligned}
$$

Figure 2.4: Over-approximated first tokens of BNF$_{\text{ES}}$ symbols



Each node $s[L]$ denotes a symbol $s$ with a set of lookahead tokens $L$. The underlined character in the string of each node denotes the current position in the parsing process that follows a pre-order traversal. The parser starts from the starting non-terminal $S$ with the special lookahead $\circ$, which denotes the end of inputs. Then, it visits the first alternative $E$ `+` $E$ with the same lookahead $\circ$. Each symbol is visited with its corresponding lookahead, which is the first tokens of the right next symbol. For example, for the second symbol `+` in $E$ `+` $E$, the next symbol is $E$ and its first tokens are:

$$
\begin{aligned}
\textbf{first}_s(E) &= \textbf{first}_\alpha(\mathsf{x}) \cup \textbf{first}_\alpha(\mathsf{x.p}) \\
&= \textbf{first}_s(\mathsf{x}) \cup (\textbf{first}_s(\mathsf{x}) :\!+\ \textbf{first}_\alpha(\mathsf{.p})) = \{\mathsf{x}\}
\end{aligned}
$$

Thus, the parser visits `+` with the lookahead `x`. The most important point here is the difference between two visits of the non-terminal $E$ in $E$ `+` $E$. The first visit of $E$ has the lookahead `+` and the actual next character after matching `x` is also `+`. Thus, the first alternative `x` of $E$ is chosen for the first visit. However, in the second visit of $E$, the lookahead is the end of inputs $\circ$ but the next character after matching `x` is the dot character (`.`) instead of the end of inputs. Therefore, the second alternative `x.p` is chosen in the second visit and the parser now successfully parses the input `x+x.p`.

$$
\begin{array}{ll}
(s_1 \cdots s_n)[L] & = s_1[\mathbf{first}_s(s_2 \cdots s_n) :+ L]\ (s_1 \cdots s_n)[L] \\
\epsilon[L] & = +\mathbf{get}_s(L) \\
\mathsf{a}[L] & = \mathsf{a}\ +\mathbf{get}_s(L) \\
A(a_1, \cdots, a_k)[L] & = \alpha_1[L] \mid \cdots \mid \alpha_n[L] \\
 & \quad \text{where } A(a_1, \cdots, a_k) = \alpha_1 \mid \cdots \mid \alpha_n \\
s?[L] & = s[L] \mid \epsilon[L] \\
(\pm s)[L] & = \pm(s[L]) \\
(s \setminus s')[L] & = s[L] \setminus s' \\
\langle \neg \mathsf{LT} \rangle & = \langle \neg \mathsf{LT} \rangle\ +\mathbf{get}_s(L)
\end{array}
$$

Figure 2.5: Formal semantics of lookahead parsers

We formally define the semantics of lookahead parsers in Figure 2.5. The helper function $\mathbf{get}_s(L)$ generates a parser by combining all tokens in the lookahead $L$ using prioritized choices. In this case, the order does not change the semantics of lookahead parsers because $\mathbf{get}_s(L)$ just checks the existence of a given token.

### 2.2.3 Implementation

We implemented the lookahead parsing technique by extending the Scala parser combinators library, which is a Scala library for PEG-based parser generation. We developed Parser Generator to generate PEG-based parsers with lookahead parsing for $\text{BNF}_{\text{ES}}$.

#### AST Generation

from a given $\text{BNF}_{\text{ES}}$ grammar. Because the structure of lexical productions do not affect the ECMAScript semantics, we represent lexical non-terminals as string values. For each syntactic production $A(p_1, \cdots, p_k)\ ::=\ (c_1\ \Rightarrow)^? \alpha_1\ \mid\ \cdots\ \mid\ (c_n\ \Rightarrow)^? \alpha_n$ , the generator generates a trait `A` and its multiple subclasses $\mathtt{A}_i$ for $0 \leq i \leq n-1$ that represent its alternatives. Each class $\mathtt{A}_i$ has non-terminals in its corresponding alternative as its fields. For instance, the *ArrayLiteral* production in Figure 2.2 gets automatically translated to the following Scala classes:

```scala
trait ArrayLiteral extends AST
case class ArrayLiteral0(x1: Option[Elision]) extends ArrayLiteral
case class ArrayLiteral1(x1: ElementList) extends ArrayLiteral
case class ArrayLiteral2(x1: ElementList, x3: Option[Elision]) extends ArrayLiteral
```

#### Parser Generation

The next step is to generate the parser for each $\text{BNF}_{\text{ES}}$ production. We extended Scala parser combinators to support lookahead parsing and $\text{BNF}_{\text{ES}}$ notations. For example, the generated parser from the production *ArrayLiteral* in Figure 2.2(a) is shown in Figure 2.2(b). A naïve implementation of lookahead parsing would take exponential time because of backtracking. To reduce it to linear time, we applied the memoization technique introduced in Packrat parsing [29]. Moreover, we also implemented the *growing the seed* technique [99] to support direct and even indirect left recursive productions.

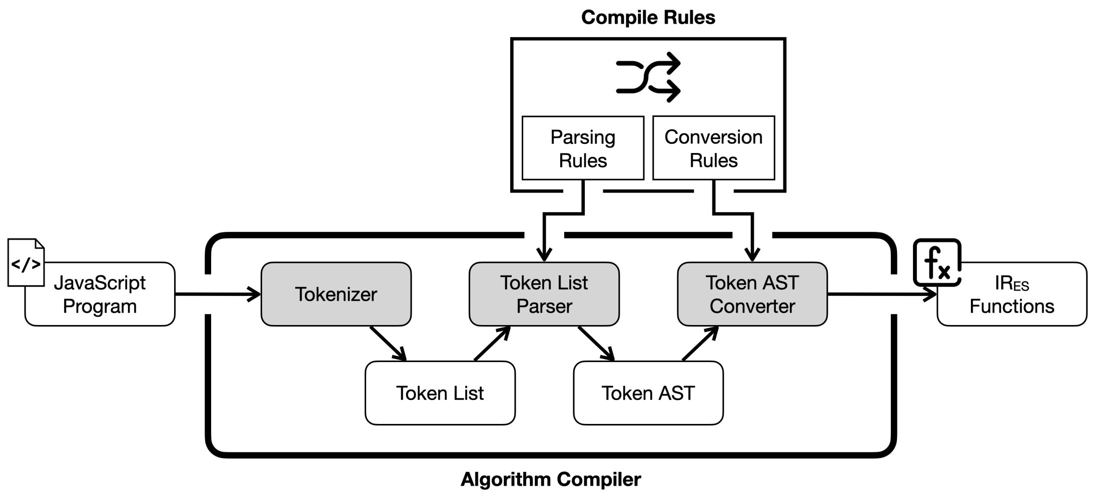


Figure 2.6: Overall structure of Algorithm Compiler

The generated parsers also support the automatic semicolon insertion algorithm, which is one of the most distinctive parsing features in ECMAScript. We extended our parsing algorithm to keep track of the right-most position that fails to be parsed in a given input. In ECMAScript, the token at that position is defined as an *offending token* and the automatic semicolon insertion algorithm is defined with such tokens. The algorithm is simple when we already have the positions of offending tokens. Thus, we just manually supported them by following the rules defined in Section 12.9[2] in ES12. The automatic semicolon insertion rules rarely change; since ES5.1 written in 2011, only one sub-rule was added.

## 2.3 Algorithm Compiler

In this section, we explain Algorithm Compiler that compiles abstract algorithms to $IR_{ES}$ functions as illustrated in Figure 2.6.

### 2.3.1 Tokenizer

Before compiling abstract algorithms, Tokenizer first tokenizes each abstract algorithm into a list of tagged tokens. An algorithm consists of ordered steps, and a step may contain sub-steps as well. For example, the Evaluation abstract algorithm in Figure 2.3(a) has five steps and its fourth step has two sub-steps. Moreover, the tokens of each step have their own HTML tags and each tag has a meaning. We keep such HTML tag information for each token to construct more precise Compile Rules. If an HTML element is just a text without any explicit tags, it is divided into multiple tokens and each token becomes a sequence of alphanumeric characters or a single non-alphanumeric character. For example, in the Evaluation algorithm, *array* is a single token with a HTML tag `<var>` and "! ArrayCreate(0)" is divided into five text tokens: !, `ArrayCreate`, `(`, `0`, and `)`.

Moreover, Tokenizer flattens a structured step to a single token list to handle multi-step statements easily. Some statements in abstract algorithms consist of multiple steps. For example, the fourth algorithm step of the Evaluation abstract algorithm in Figure 2.3(a) has two sub-steps 4.a and 4.b for the then-branch of the conditional statement. To treat them as a linear structure, we introduce three special tokens to break down structured algorithms: ↓ denotes the end of a single step, and ↘ and ↙ denote

[2]`https://262.ecma-international.org/12.0/#sec-automatic-semicolon-insertion`

```
/*----------- parsing rules ----------- ^^ ----------- conversion rules -----------*/
// statements
val Stmt =
  "Let" ~ varT ~ "be" ~ Expr ~ "."      ^^ { case _ ~ x ~ _ ~ e ~ _ => ILet(x, e) } |

// expressions
val Expr =
  // completion record unwrap
  "!" ~ Expr                            ^^ { case e => EUnwrapComp(e, false) }      |
  "?" ~ Expr                            ^^ { case e => EUnwrapComp(e, true) }       |
  // variables
  varT                                  ^^ { case x => EId(x) }                     |
  // numbers
  number                                ^^ { case n => ENum(n) }                    |
  // function calls
  word ~ "(" ~ repsep(Expr, ",") ~ ")"  ^^ { case f ~ _ ~ as ~ _ => ECall(f, as) }
```

Figure 2.7: A simplified version of compile rules for the first step of the Evaluation algorithm in Figure 2.3(a)

the start and the end of nested steps, respectively. For example, the following left abstract algorithm is tokenized to the right token list:

```
1. A
2. B
  a. C
```

$$\Longrightarrow \qquad \mathtt{A} \downarrow \mathtt{B} \searrow \mathtt{C} \downarrow \swarrow \downarrow$$

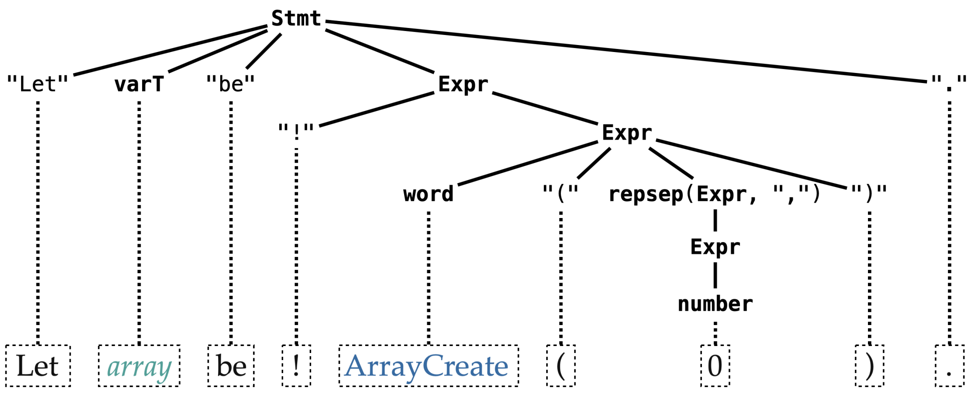
After tokenizing abstract algorithms, Algorithm Compiler compiles token lists into $\text{IR}_{\text{ES}}$ functions using Token List Parser and Token AST Converter. They depend on Compile Rules and each compile rule consists of a *parsing rule* and a *conversion rule*:

```
val CompileRule = ParsingRule ^^ ConversionRule
```

For each compile rule, its parsing rule describes how to parse a given token list into a structured token AST, and its conversion rule describes how to convert the given token AST structure into an $\text{IR}_{\text{ES}}$ component. For example, Figure 2.7 shows a simplified version of compile rules for the first step of the Evaluation algorithm in Figure 2.3(a): "Let *array* be ! ArrayCreate(0)". The `Stmt` compile rule is a single rule that describes how to compile statements, and the `Expr` compile rule consists of five rules that describes how to compile expressions. Now, we explain Token List Parser and Token AST Converter with parsing rules and conversion rules, respectively.

### 2.3.2 Token List Parser

Token List Parser is defined with *parsing rules*. A parsing rule is a basic parsing rule or a composition of multiple parsing rules. The composition `A | B` of two parsing rules `A` and `B` parses an input using both rules and collects the longest matched results. If both rules fail or match the same length of the input, the composition fails. We provide two kinds of basic parsing rules: *tag-based rules* and *content-based rules*. A tag-based rule just checks whether the next token has a given tag. For example, the tag-based parser `varT` checks whether the next token has the HTML tag `<var>`. A content-based parser checks whether the next token is a text token and its content passes a given condition. For example, the string literal `"ArrayCreate"` denotes a content-based parser that checks whether the next token is a text token with the content "ArrayCreate". We also define two content-based parsers `word` and `number` that check whether the content of the next token consists of only alphabets or numbers, respectively. In addition, we provide several helper functions such as the optional rule `A?` and the positive (negative) predicate `+A`(`-A`). For instance, the helper function `repsep(A, B)` generates a new parsing rule that denotes zero or more repetition of the parsing rule `A` using another parsing rule `B` as a separator. Therefore, Token List Parser with the parsing rules in the left side of Figure 2.7 parses the first step of the Evaluation algorithm to the following token AST:



## 2.3.3 Token AST Converter

*Conversion rules* describe how to convert token ASTs to the corresponding $\mathrm{IR}_{\mathrm{ES}}$ components, and Token AST Converter utilizes them. Each conversion rule is defined with its corresponding parsing rule. For basic parsing rules, their conversion rules always return the string values of the contents in parsed tokens. For example, the right side of Figure 2.7 describes the conversion rules. The conversion rule of the `Stmt` compile rule uses only the second and fourth sub-trees to construct a `let`-binding instruction. The constructor `ILet` for `let`-binding instructions takes two arguments an identifier name and an expression. For the fourth sub-tree, the conversion rule of the first `Expr` compile rule is applied to it to construct a completion record unwrap expression. The constructor `EUnwrapComp` for completion record unwrap expressions takes two arguments an expression and a boolean value that represents whether checking the abrupt completion. Each abstract algorithm in ECMAScript always returns a *completion record* to handle different kinds of JavaScript control flows. A completion record is *abrupt* if it contains values that represent abnormal control flows such as exceptions, **`return`**, or **`break`**; otherwise, it is *normal*. The prefix "?" checks whether a completion record is abrupt and returns immediately if so. Otherwise, it unwraps the completion record to its containing value. On the other hand, the prefix "!" unwraps a completion record to its containing value without checking the abrupt completion. Therefore, `EUnwrapComp(e, false)` denotes a completion record unwrap expression of another expression `e` without checking the abrupt completion. In this way, the first step of the algorithm is converted to the following $\mathrm{IR}_{\mathrm{ES}}$ instruction:

```
ILet("array", EUnwrapComp(ECall("ArrayCreate", 0), false))
```

and its beautified form is as follows:

```
let array = [! (ArrayCreate 0)]
```

We define $\text{IR}_{\text{ES}}$ to represent abstract algorithms as its functions with the following design choices:

- **Dynamic typing:** Because each variable in abstract algorithms is not statically typed, variables do not have their own static types while each value of $\text{IR}_{\text{ES}}$ has its dynamic type.
- **Imperative style:** $\text{IR}_{\text{ES}}$ represents algorithm steps as imperative instructions in the sense that each instruction changes the current state consisting of an environment and a heap.
- **Higher-order functions with restricted scopes:** In each function of $\text{IR}_{\text{ES}}$, only global variables, parameters, and its local variables are available, which means that a function closure does not capture its current environment. We use such restricted scopes because they are enough to represent abstract algorithms.
- **Primitive values:** $\text{IR}_{\text{ES}}$ supports ECMAScript primitive values except "symbols" because symbols can be represented as singleton objects. Also, $\text{IR}_{\text{ES}}$ provides the unique `absent` value to represent the absence of parameters. For example, when the optional second parameter *Elision* of Evaluation in Figure 2.3(a) is absent, the parameter has the `absent` value.
- **Abstract data types:** $\text{IR}_{\text{ES}}$ supports only three abstract data types: `Record` for mappings from values to values, `List` for sequential data, and `Symbol` for singleton data. For example, ECMAScript environment records are represented as `Record` from string values to addresses that represent the bindings of the string values.

We define the syntax of $\text{IR}_{\text{ES}}$ that has 15 kinds of instructions and 26 kinds of expressions with the notation $i$ and $e$, respectively. We also formally define its operational semantics $\sigma \vdash i \Rightarrow \sigma$ for instructions and $\sigma \vdash e \Rightarrow (v, \sigma)$ for expressions, where $\sigma$ denotes a state and $v$ denotes a value. For presentation brevity, we omit the formalization of $\text{IR}_{\text{ES}}$ in this thesis and include it in a companion report [14].

### 2.3.4 Implementation

We implemented Algorithm Compiler by extending the Packrat parsing [29] library in Scala parser combinators. We modified the meaning of the composition operator (|||) to collect all the longest matched results. If a parser detects a step that cannot be parsed or is parsed in multiple ways, it reports the step with parsing results.

#### Compile Rules

Algorithm Compiler requires compile rules to compile given abstract algorithms to $\text{IR}_{\text{ES}}$ functions. As already explained in Section 2.1, we found common patterns in the writing style of abstract algorithms. We manually defined general compile rules to represent such a writing style with six different kinds as summarized in Table 2.1. The compile rule for statements, `Stmt`, generates $\text{IR}_{\text{ES}}$ instructions. The `Expr`, `Cond`, and `Value` compile rules generate $\text{IR}_{\text{ES}}$ expressions, but they represent different contexts in ECMAScript; `Expr` represents a context where any expression can appear, `Cond` denotes a context where any boolean-valued expression can appear, and `Value` represents a context where a fully evaluated value can appear. The `Ty` compile rule denotes type names and generates string primitives used in object constructions. The `Ref` compile rule represents references such as identifier lookup and member accesses of objects, and it generates $\text{IR}_{\text{ES}}$ references.

Table 2.1: General compile rules for ECMAScript

| Name | `Stmt` | `Expr` | `Cond` | `Value` | `Ty` | `Ref` |
|---|---|---|---|---|---|---|
| # Rules | 21 | 27 | 16 | 11 | 34 | 9 |

**Global Setting** AST-$\text{IR}_{\text{ES}}$ Translator uses global settings consisting of *ECMAScript data types* and *built-in objects*. Unlike compile rules, global settings depend on ECMAScript versions. In this thesis, we construct global settings for the latest ECMAScript, ES12.

ECMAScript describes data types with some fields and methods. While the methods are like abstract algorithms, their semantics are slightly different from abstract algorithms. They implicitly get their receiver objects as arguments at callsites. To mimic such an implicit behavior, we added a special variable `this` as the first parameter of each method, and passed a receiver object at its callsite by modifying Algorithm Compiler. For example, an Environment Record type has the DeleteBinding($N$) method. Thus, its corresponding $\text{IR}_{\text{ES}}$ function has two parameters, the special parameter `this` and a normal parameter `N`, and the method call *DeclRec*.DeleteBinding($N$) in an abstract algorithm is compiled to the $\text{IR}_{\text{ES}}$ instruction: `(DclRec.DeleteBinding DclRec N)`.

In ECMAScript, built-in objects are pre-defined functions with several built-in functions. For example, `Array` is the constructor of array objects, and its prototype `Array.prototype` has built-in functions for array objects. For instance, `[1,2,3].flat()` calls the `Array.prototype.flat` built-in function with the array `[1,2,3]`. Because built-in functions are also abstract algorithms, each of them is automatically compiled to an $\text{IR}_{\text{ES}}$ function. However, the structures of built-in objects should be manually implemented. Thus, we implemented built-in objects in Scala and connected their properties with the compiled $\text{IR}_{\text{ES}}$ functions. Some built-in objects that are explicitly referenced in abstract algorithms are intrinsic objects, which have their own aliased names summarized in Table 8: Well-Known Intrinsic Objects in Section 6.1.7.4[3] of ES12. We extracted the alias into Global Setting to utilize it during evaluation.

## 2.4 Evaluation

We developed JISET as an open-source tool[4], and evaluated the tool based on the following research questions:

- RQ1. **Coverage:** How much percentage of the syntax and semantics does JISET automatically extract from ES7 to ES12?
- RQ2. **Correctness:** Does JISET correctly extract an IR-based formal semantics from ECMAScript compared to the official conformance tests?
- RQ3. **Adaptability:** Is JISET applicable to new language features ready for inclusion in the next version of ECMAScript?

We performed our experiments on a machine equipped with 4.2GHz Quad-Core Intel Core i7 and 64GB of RAM. On the machine, JISET took less than one minute to extract mechanized specifications from a given ECMAScript.

[3] `https://262.ecma-international.org/12.0/#sec-well-known-intrinsic-objects`
[4] `https://github.com/kaist-plrg/jiset`

Table 2.2: Syntax coverage - Number of productions in each specification and in each update between adjacent versions, from *all* of which JISET automatically generated parsers

(a) For each ECMAScript version from ES7 to ES12

| **Version** | **ES7** | **ES8** | **ES9** | **ES10** | **ES11** | **ES12** | **Average** |
|---|---|---|---|---|---|---|---|
| # Lexical productions | 124 | 124 | 140 | 140 | 143 | 146 | 136.17 |
| # Syntactic productions | 157 | 167 | 174 | 174 | 185 | 187 | 174.00 |

(b) For each update between adjacent versions

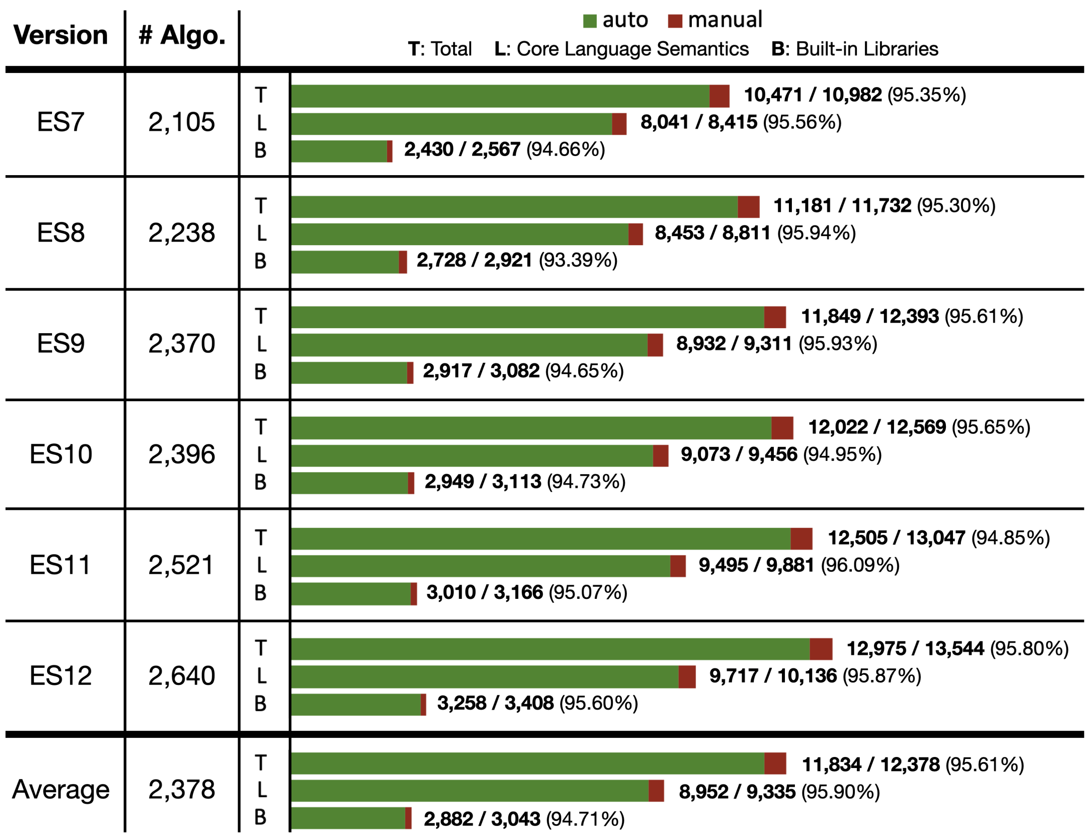
| **Old version** | **ES7** | **ES8** | **ES9** | **ES10** | **ES11** | **Average** |
|---|---|---|---|---|---|---|
| **New version** | **ES8** | **ES9** | **ES10** | **ES11** | **ES12** | |
| Δ # Lexical productions | 7 | 31 | 2 | 14 | 29 | 16.60 |
| Δ # Syntactic productions | 142 | 22 | 1 | 20 | 10 | 39.00 |

### 2.4.1 Coverage

We evaluated the coverage of JISET in two respects: syntax and semantics. First, we measured how many lexical and syntactic productions in specifications JISET generated parsers for syntax. Then, we measured how many abstract algorithm steps it compiled to $\mathrm{IR}_{\mathrm{ES}}$ functions for semantics. As discussed in Section 2.1, JISET utilizes common patterns in the converged writing style since ES7. Thus, we evaluated the coverage of JISET using the most recent six versions of ECMAScript, ES7 to ES12. We measured the numbers for each ECMAScript version and each update between adjacent versions. While we evaluated only four versions from ES7 to ES10 when we conducted this work in 2019, we evaluated six versions from ES7 to ES12 with the recent version of JISET in this thesis.

For syntax, JISET successfully generated parsers for the lexical and syntactic productions in all the versions of ECMAScript. Table 2.2 shows that the number of lexical and syntactic productions has consistently increased, and ES12 has 146 lexical and 187 syntactic productions to represent JavaScript syntax. All six versions of ECMAScript contain 136.17 and 174.00 lexical and syntactic productions on average. Each update between adjacent versions removes, modifies, or introduces 16.60 lexical and 39.00 syntactic productions on average. It showed that JISET successfully lessens the burden to manually design and update parsers for revised JavaScript syntax in ECMAScript.

For semantics, Figure 2.8(a) shows that JISET automatically compiled algorithm steps to corresponding $\mathrm{IR}_{\mathrm{ES}}$ instructions with the success rate of 95.61% on average for each ECMAScript version from ES7 to ES12. Figure 2.8(b) shows that it compiled 90.83% of modified or newly introduced algorithm steps on average for each update between adjacent versions. It showed that JISET could reduce efforts to develop JavaScript tools from scratch based on specifications and evolve existing tools for the specification update. ECMAScript abstract algorithms describe not only the core language semantics but also behaviors of built-in libraries with various helper functions. Note that built-in libraries are written in more diverse styles than core language semantics due to their own specific functionalities. For example, `String.prototype` or `Array.prototype` are built-in objects and have diverse library functions. Therefore, The English sentences used in their abstract algorithms contain diverse expressions related to manipulating strings or array objects. As a result, core language semantics have slightly higher success rates (95.61% for specifications and 91.53% for updates) than built-in libraries (94.71% for specifications and 89.06% for updates) on average.

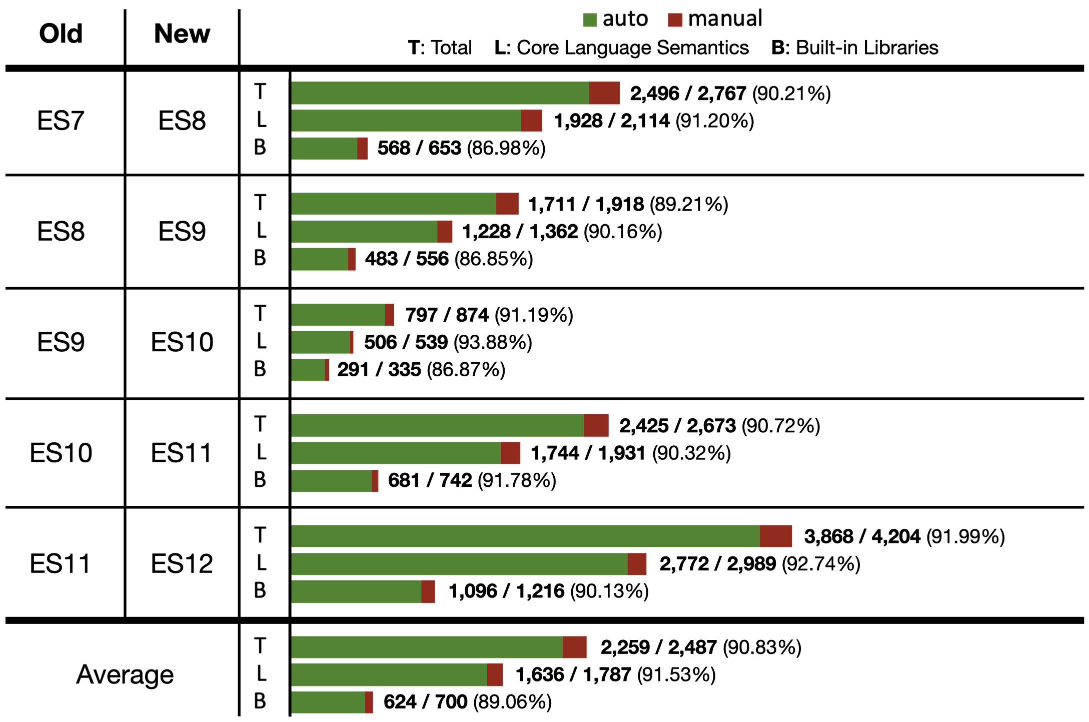


🟩 auto 🟥 manual

**T**: Total **L**: Core Language Semantics **B**: Built-in Libraries

| Version | # Algo. | | |
|---|---|---|---|
| ES7 | 2,105 | T | **10,471 / 10,982** (95.35%) |
| | | L | **8,041 / 8,415** (95.56%) |
| | | B | **2,430 / 2,567** (94.66%) |
| ES8 | 2,238 | T | **11,181 / 11,732** (95.30%) |
| | | L | **8,453 / 8,811** (95.94%) |
| | | B | **2,728 / 2,921** (93.39%) |
| ES9 | 2,370 | T | **11,849 / 12,393** (95.61%) |
| | | L | **8,932 / 9,311** (95.93%) |
| | | B | **2,917 / 3,082** (94.65%) |
| ES10 | 2,396 | T | **12,022 / 12,569** (95.65%) |
| | | L | **9,073 / 9,456** (94.95%) |
| | | B | **2,949 / 3,113** (94.73%) |
| ES11 | 2,521 | T | **12,505 / 13,047** (94.85%) |
| | | L | **9,495 / 9,881** (96.09%) |
| | | B | **3,010 / 3,166** (95.07%) |
| ES12 | 2,640 | T | **12,975 / 13,544** (95.80%) |
| | | L | **9,717 / 10,136** (95.87%) |
| | | B | **3,258 / 3,408** (95.60%) |
| Average | 2,378 | T | **11,834 / 12,378** (95.61%) |
| | | L | **8,952 / 9,335** (95.90%) |
| | | B | **2,882 / 3,043** (94.71%) |

(a) For each ECMAScript version from ES7 to ES12



🟩 auto 🟥 manual

**T**: Total **L**: Core Language Semantics **B**: Built-in Libraries

| Old | New | | |
|---|---|---|---|
| ES7 | ES8 | T | **2,496 / 2,767** (90.21%) |
| | | L | **1,928 / 2,114** (91.20%) |
| | | B | **568 / 653** (86.98%) |
| ES8 | ES9 | T | **1,711 / 1,918** (89.21%) |
| | | L | **1,228 / 1,362** (90.16%) |
| | | B | **483 / 556** (86.85%) |
| ES9 | ES10 | T | **797 / 874** (91.19%) |
| | | L | **506 / 539** (93.88%) |
| | | B | **291 / 335** (86.87%) |
| ES10 | ES11 | T | **2,425 / 2,673** (90.72%) |
| | | L | **1,744 / 1,931** (90.32%) |
| | | B | **681 / 742** (91.78%) |
| ES11 | ES12 | T | **3,868 / 4,204** (91.99%) |
| | | L | **2,772 / 2,989** (92.74%) |
| | | B | **1,096 / 1,216** (90.13%) |
| Average | | T | **2,259 / 2,487** (90.83%) |
| | | L | **1,636 / 1,787** (91.53%) |
| | | B | **624 / 700** (89.06%) |

(b) For each update between adjacent versions

Figure 2.8: Semantics coverage - Number of algorithm steps in specifications, from which JISET generated the semantics

Table 2.3: Test results with Test262 for ES10

| **All Test262 Tests** | **35,990** |
|---|---|
| Annexes | 1,060 |
| Internationalization | 640 |
| In-progress features | 5,338 |
| **ES10 Tests** | **28,952** |
| Non-strict mode | 1,150 |
| Modules | 918 |
| Early errors before actual execution | 2,288 |
| Inessential built-in objects | 6,532 |
| **Applicable Tests** | **18,064** |
| Passed tests | 16,355 |
| Failed tests | 1,709 |

### 2.4.2 Correctness

To evaluate the correctness of JISET, we tested the extracted semantics from the most recent version of ECMAScript (ES10) when we conducted this work in 2019. We utilized test programs in Test262 as of February 28, 2019, when ES10 was branched out from the main branch of the ECMAScript repository. To focus on the core language semantics of JavaScript, we completed only the necessary parts missing in the extracted mechanized specification. Figure 2.8(a) shows that Algorithm Compiler successfully compiled 12,022 out of 12,569 steps in ES10. It fully covered 2,034 out of 2,396 abstract algorithms, and 362 algorithms were partially covered. Among the remaining 547 algorithm steps, we manually completed 277 algorithm steps to cover essential parts for the language semantics. Based on this manual implementation, 146 more abstract algorithms are fully covered. We also manually implemented Global Setting as described in Section 2.3.4 for the core language features. Note that we do not support minor language features such as the non-strict mode, modules, early errors before actual execution, and inessential built-in objects. Among 35,990 tests in Test262, we filtered out 17,926 tests, as summarized in Table 2.3. To focus on ES10, we excluded 7,038 tests for annexes, internationalization, and in-progress features. We also filtered out 10,888 tests that use minor language features. Finally, the extracted semantics took about three hours to evaluate 18,064 applicable tests and failed 1,709 tests.

We investigated the failed tests and found that they failed due to specification errors in ES10. We discovered nine errors using the failed tests: ES10-1 to ES10-9 in Table 2.4. Among them, five errors (ES10-5 to ES10-9) were previously reported and fixed in the current draft of the next ECMAScript, and the remaining four errors (ES10-1 to ES10-4) were never reported before. All four errors were confirmed by TC39 and fixed in the next ECMAScript, ES11.

The specification error ES10-1 is due to a wrong assertion. While ES9 introduced the **`for await`** iteration statement with a new *iterationKind* tag, **`async`**`-iterate`, the ForIn/OfHeadEvaluation algorithm missed the **`async`**`-iterate` case in an assertion, which caused that 1,120 tests failed. We reported the error and proposed a specification fix to include the **`async`**`-iterate` case, and TC39 accepted it on March 25, 2020. Because the error was created on February 16, 2018, it existed for 768 days.

Table 2.4: Specification errors in ES10 and the BigInt proposal ready for inclusion in ES11

| Name | Feature | Description | Known | Created | Resolved | Existed | #Fails |
|---|---|---|---|---|---|---|---|
| ES10-1 | Iteration | Missing the `async-iterate` case in the assertion of ForIn/OfHeadEvaluation | X | 2018-02-16 | 2020-03-25 | 768 days | 1,116 |
| ES10-2 | Condition | Ambiguous grammar production for the dangling **`else`** problem in *IfStatement* | X | 2015-06-01 | 2020-10-01 | 1,949 days | 1 |
| ES10-3 | String | Wrong use of the `=` operator in StringGetOwnProperty | X | 2015-06-01 | 2020-05-07 | 1,802 days | 7 |
| ES10-4 | Completion | Unhandling abrupt completion in Abstract Equality Comparison | X | 2015-06-01 | 2020-04-28 | 1,793 days | 9 |
| ES10-5 | Completion | Unhandling abrupt completion in Evaluation of *EqualityExpression* | O | 2015-06-01 | 2019-05-02 | 1,431 days | 2 |
| ES10-6 | Await | Passing a value of wrong type to the second parameter of PromiseResolve | O | 2019-02-27 | 2019-04-13 | 45 days | 1,294 |
| ES10-7 | Function | No semantics of IsFunctionDefinition for `function(...){...}` | O | 2015-10-30 | 2020-01-18 | 1,541 days | 306 |
| ES10-8 | Function | No semantics of ExpectedArgumentCount for the base case of *FormalParameters* | O | 2016-11-02 | 2020-02-20 | 1,205 days | 81 |
| ES10-9 | Iteration | Two semantics of VarScopedDeclarations for `for await(var x of e){...}` | O | 2018-02-16 | 2019-10-11 | 602 days | 0 |
| BigInt-1 | Expression | Using the wrong variable `oldvalue` instead of `oldValue` in Evaluation of *UpdateExpression* | X | 2019-09-27 | 2020-04-23 | 209 days | 533 |
| BigInt-2 | Number | Using ToInt32 instead of ToUint32 in Number::unsignedRightShift | X | 2019-09-27 | 2020-04-23 | 209 days | 2 |
| BigInt-3 | Number | Unhandling BigInt values in the Number constructor | O | 2019-09-27 | 2019-11-19 | 53 days | 1 |

ES10-2 comes from the well-known dangling **`else`** problem introduced in ALGOL 60 [11]. ES10 describes how to parse it in prose: the **`else`** statement should be associated with the nearest **`if`** statement. Because it is written in prose rather than in the ES10 grammar productions, it caused one failed test. We proposed a fix to revise the ambiguous grammar production, and TC39 accepted it on October 1, 2020. Thus, ES10-2 existed for 1,949 days.

ES10-3 is due to a misuse of the `=` operator for numbers. In abstract algorithms, "$x = y$" denotes equality testing for double-precision 64-bit binary format IEEE 754-2008 values; thus, "`+0` = `-0`" evaluates to true. However, to check whether *index* is exactly the same with `-0`, StringGetOwnProperty used "*index* = `-0`", which is true even when *index* is `+0`. As a result, it caused seven failed tests. We proposed a fix accepted on May 7, 2020. Thus, ES10-3 existed for 1,802 days.

ES10-4 and ES10-5 happened because ES10 did not handle abrupt completion from function calls. Our proposed fix to ES10-4 was accepted on April 28, 2020, and ES10-5 was resolved on May 2, 2019 after existing for 1,431 days.

ES10-6 is due to incorrect uses of an abstract algorithm. While PromiseResolve($C$, $x$) expects a JavaScript object for its second argument, ES10 passed a list of values rather than an object in three invocations of PromiseResolve. The wrong invocations were introduced on February 27, 2019 and caused that 1,294 tests failed. They were fixed on April 13, 2019 after existing for 45 days.

Table 2.5: Proposals that will be included in ES11

| Proposal | Δ # Productions | | Δ # Steps | Δ # Tests | # Tests |
|---|---|---|---|---|---|
| | Lexical | Syntactic | | | |
| `matchAll` of `String` | 0 | 0 | 9/9 | 5/5 | 18,064/18,064 |
| `import()` | 0 | 2 | 38/38 | 0/0 | 18,064/18,064 |
| `BigInt` | 4 | 0 | 298/326 | 196/207 | 17,539/18,064 |
| `Promise.allSettled` | 0 | 0 | 79/85 | 50/50 | 18,064/18,064 |
| `globalThis` | 0 | 0 | 1/1 | 1/1 | 18,064/18,064 |
| `for-in` mechanics | 0 | 0 | 36/37 | 0/0 | 18,064/18,064 |
| Optional Chaining | 3 | 3 | 74/74 | 19/19 | 18,064/18,064 |
| Nullish Coalescing Operator | 1 | 4 | 10/10 | 21/21 | 18,064/18,064 |
| `import.meta` | 0 | 2 | 15/15 | 0/0 | 18,064/18,064 |
| **Total** | 8 | 11 | 560/595 | | |

ES10-7 and ES10-8 happened because ES10 missed semantics in some cases. They both existed for more than 1,200 days.

ES10-9 is due to multiple semantics. While no tests in Test262 fail with any of the semantics, we could detect this error via Spec Extractor even before executing the semantics. It is supplementary merit of the automation of mechanized specification extraction.

After resolving the nine specification errors in ES10, we extracted JavaScript semantics from the revised specification. The extracted semantics from the revised ES10 successfully parsed and executed all 18,064 applicable tests in Test262, which shows that JISET correctly extracted a mechanized specification from ES10. In addition, the evaluation witnesses that JISET can detect specification errors effectively. We could detect not only five previously-known errors but also four new errors. Therefore, we believe that JISET bridges gaps between ECMAScript written in a natural language and executable tests in Test262.

### 2.4.3 Adaptability

We evaluated whether JISET is adaptable to the proposals ready for inclusion in ECMAScript 2020 (ES11), the next version of ES10. Because ECMAScript is an open-source project, various proposals for new features are available with their own specification changes and tests. A separate repository [5] maintains them in six stages: Stage 0 to Stage 3, Finished, and Inactive. A proposal starts with Stage 0, and the TC39 committee examines proposals in Stage 3. If a proposal is confirmed, the committee changes its stage to Finished and integrates it into the next ECMAScript. Otherwise, its stage becomes Inactive.

As shown in Table 2.5, we applied JISET to all nine "Finished" proposals. Collectively, the proposals modified eight lexical and 11 syntactic productions, and JISET generated JavaScript parsers for them. As a result, the generated parsers successfully parsed all applicable tests for all nine proposals. Then, Algorithm Compiler with the same Compile Rules compiled their 560 algorithm steps out of 595 into corresponding $\text{IR}_{\text{ES}}$ instructions. Therefore, JISET has a success rate of 94.12% on average for forthcoming proposals.

[5] `https://github.com/tc39/proposals`

Table 2.6: Applicable conformance tests in Test262 for ES12

| **All Test262 Conformance Tests** | **41,415** |
|---|---|
| **Inapplicable Tests** | **22,859** |
| Web Browsers / Internationalization | 2,036 |
| In-Progress Features | 5,719 |
| Non-Strict / Module | 2,625 |
| Early Errors | 2,949 |
| Inessential Built-in Objects (e.g. `JSON`, `Atomics`) | 9,530 |
| **Applicable Tests** | **18,556** |

We checked the extracted semantics from the proposals by implementing missing parts of the mechanized specification for each proposal and checking the semantics with Test262. All of them passed all applicable tests except the semantics from the `BigInt` proposal. However, it failed for 11 out of 207 applicable tests for the proposal and 525 tests out of 18,064 applicable tests for ES10.

Using the failed tests, we discovered three errors in the BigInt proposal: two new errors (BigInt-1 and BigInt-2) and one known error (BigInt-3), as summarized in Table 2.4. All of them were confirmed by TC39 and will be fixed in ES11. The proposal added two new types: BigInt as a new type of primitives and Numeric as a unified type of the original Number type and the new BigInt type. Therefore, it not only added new algorithms for BigInt but also modified all existing algorithms for Number values. The error BigInt-1 is due to a misuse of the variable *oldValue* in Evaluation of *UpdateExpression*. BigInt-2 breaks the backward compatibility because of misusing ToInt32 instead of ToUint32 in unsigned right shift operators. BigInt-3 is due to missing BigInt primitives in the Number constructor. On average, three errors existed for 157 days in the proposal.

After fixing the errors in the proposal, we extracted a semantics from the revised specification. The extracted semantics passed all 207 applicable tests for the proposal and 18,064 applicable tests for ES10. Thus, JISET also correctly extracted an mechanized specification from future proposals, which implies that it is adaptable for new language semantics.

Moreover, we also extracted a mechanized specification from ES12 and checked its correctness to check the adaptability of JISET for the latest ECMAScript. As shown in Figure 2.8(a), Algorithm Compiler has a success rate of 95.80% by compiling 12,975 steps out of 13,544 for ES12. Among the remaining 569 steps, we manually completed 103 algorithm steps to define the mechanized specification. To check its correctness, we utilized Test262 again but its current version to cover new language semantics introduced after ES10. Since ES12 was released in June 2021, we used Test262 as of June 2021[6]. Among 41,415 tests, we filtered out 22,859 tests using minor language features, as summarized in Table 2.6. Therefore, we used 18,556 applicable Test262 tests to check the correctness of the extracted semantics from ES12. It successfully parsed and executed all 18,064 applicable tests in Test262, showing that JISET correctly extracted a mechanized specification from the current latest ECMAScript, ES12.

[6]`https://github.com/tc39/test262/tree/aaf4402b4ca9923012e6`

# Chapter 3. JEST: $N$+1-version Differential Testing

This chapter introduces JEST, a JavaScript Engines and Specification Tester using $N$+1-version differential testing. Since we conducted this work in 2020, all the explanations and evaluations for the tool are as of 2020. The main contributions of this work include the following:

- We present *$N$+1-version differential testing*, a novel solution to the new problem of co-evolving language specifications and their implementations. The main idea is to generate conformance tests from the language specification and detect errors in both specifications and implementations using the generated tests.
- We actualize $N$+1-version differential testing for JavaScript as a tool named JEST. It is the first tool that automatically generates conformance tests for JavaScript engines from ECMAScript. We developed the tool by extending JISET to fully utilize the mechanized specification extracted from any version of ECMAScript. JEST generates JavaScript conformance tests using the extracted mechanized specification and detects errors in both JavaScript engines and ECMAScript.
- We evaluate JEST with four modern JavaScript engines (V8, GraalJS, QuickJS, and Moddable XS) and ES11, the latest ECMAScript in 2020. Four JavaScript engines fully support ES11 and pass all Test262 tests for ES11. Our tool successfully generated 1,700 conformance tests from the specification. The semantics coverage of Test262 is 91.61% for statements and 82.91% for branches. The conformance tests fully automatically generated by JEST have similar semantics coverage: 87.70% for statements and 78.30% for branches. Finally, our tool successfully found and localized 44 engine bugs in four different engines and 27 specification bugs in ES11 using the tests.

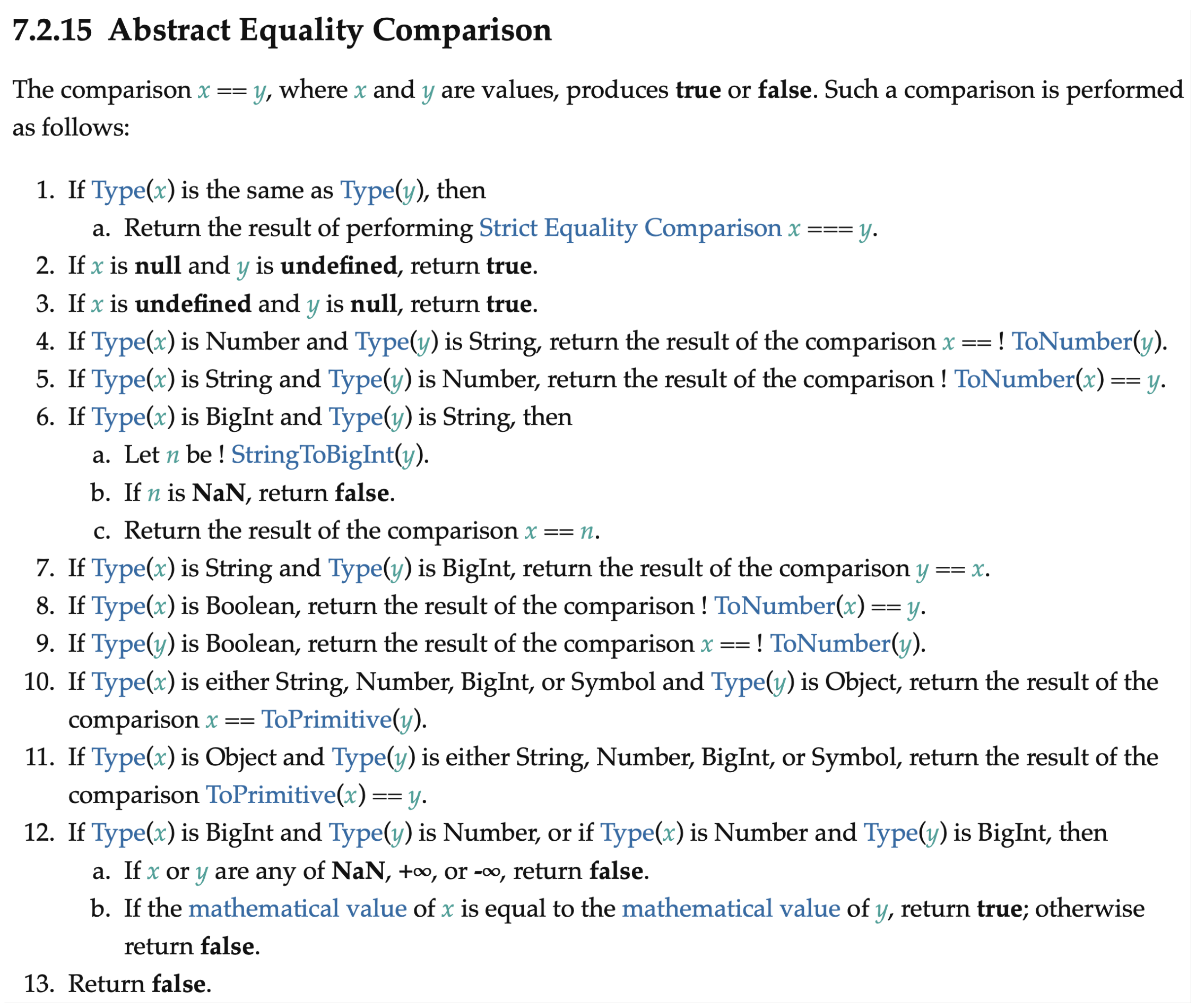
## 3.1 $N$+1-version Differential Testing

This section introduces the core concept of $N$+1-version differential testing with a simple running example. The overall structure consists of two phases: 1) a conformance test generation phase and 2) a bug detection and localization phase.

### 3.1.1 Main Idea

Differential testing utilizes the cross-referencing oracle, which is an assumption that any discrepancies between program behaviors on the same input could be bugs. It compares the execution results of a program with the same input on $N$ different implementations. When an implementation produces a different result from the one by the majority of the implementations, differential testing reports that the implementation may have a bug.

On the contrary, $N$+1-version differential testing utilizes not only the cross-referencing oracle using multiple implementations but also a mechanized specification. It first generates test code from a mechanized specification, and tests $N$ different implementations of the specification using the generated test code as in differential testing. In addition, it can detect possible bugs in the specification as well when most implementations fail for a test. In such cases, because a bug in the specification could be triggered by the test, it localizes the bug using statistical information as we explain later in this section.



**7.2.15 Abstract Equality Comparison**

The comparison $x$ == $y$, where $x$ and $y$ are values, produces **true** or **false**. Such a comparison is performed as follows:

1. If Type($x$) is the same as Type($y$), then
   a. Return the result of performing Strict Equality Comparison $x$ === $y$.
2. If $x$ is **null** and $y$ is **undefined**, return **true**.
3. If $x$ is **undefined** and $y$ is **null**, return **true**.
4. If Type($x$) is Number and Type($y$) is String, return the result of the comparison $x$ == ! ToNumber($y$).
5. If Type($x$) is String and Type($y$) is Number, return the result of the comparison ! ToNumber($x$) == $y$.
6. If Type($x$) is BigInt and Type($y$) is String, then
   a. Let $n$ be ! StringToBigInt($y$).
   b. If $n$ is **NaN**, return **false**.
   c. Return the result of the comparison $x$ == $n$.
7. If Type($x$) is String and Type($y$) is BigInt, return the result of the comparison $y$ == $x$.
8. If Type($x$) is Boolean, return the result of the comparison ! ToNumber($x$) == $y$.
9. If Type($y$) is Boolean, return the result of the comparison $x$ == ! ToNumber($y$).
10. If Type($x$) is either String, Number, BigInt, or Symbol and Type($y$) is Object, return the result of the comparison $x$ == ToPrimitive($y$).
11. If Type($x$) is Object and Type($y$) is either String, Number, BigInt, or Symbol, return the result of the comparison ToPrimitive($x$) == $y$.
12. If Type($x$) is BigInt and Type($y$) is Number, or if Type($x$) is Number and Type($y$) is BigInt, then
    a. If $x$ or $y$ are any of **NaN**, +∞, or -∞, return **false**.
    b. If the mathematical value of $x$ is equal to the mathematical value of $y$, return **true**; otherwise return **false**.
13. Return **false**.

(a) The Abstract Equality Comparison algorithm in ES11

```
// JavaScript engines: exception with "err"
// ECMAScript (ES11) : result === false
var obj = { valueOf: () => { throw "err"; } };
var result = 42 == obj;
```

(b) JavaScript code using abstract equality comparison

```
try {
  var obj = { valueOf: () => { throw "err"; } };
  var result = 42 == obj;
  assert(result === false);
} catch (e) {
  assert(false);
}
```

(c) JavaScript code with injected assertions

Figure 3.1: An abstract algorithm in ES11 and code example related to its language semantics

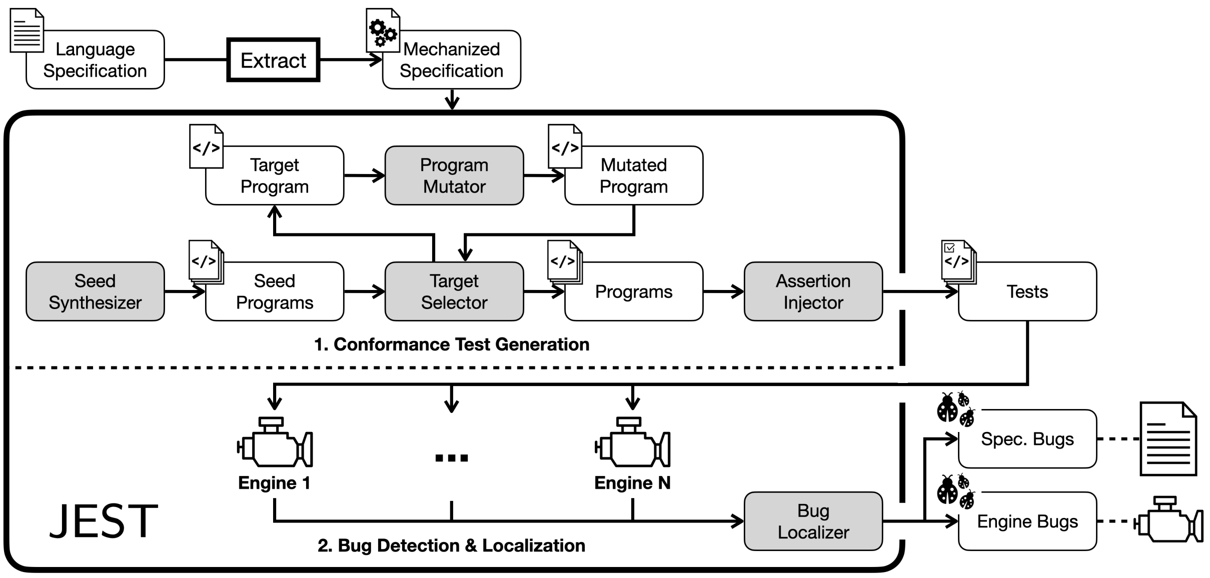


Figure 3.2: Overall structure of $N$+1-version differential testing for $N$ implementations (engines) and one language specification

### 3.1.2 Running Example

We explain how $N$+1-version differential testing works with a simple JavaScript example shown in Figure 3.1. Figure 3.1(a) is an excerpt from ECMAScript 2020 (ES11), which shows some part of the Abstract Equality Comparison abstract algorithm. It describes the semantics of non-strict equality comparison such as `==` and `!=`. For example, **`null`** `== undefined` is **`true`** because of the algorithm step 2. According to the steps 10 and 11, if the type of a value is String, Number, BigInt, or Symbol, and the type of the other value is Object, the algorithm calls ToPrimitive to convert the JavaScript object to a primitive value. Note that this is a specification bug caused by unhandled abrupt completions! To express control diverters such as exceptions, **`break`**, **`continue`**, **`return`**, and **`throw`** statements in addition to normal values, ECMAScript uses "abrupt completions." ECMAScript annotates the question mark prefix (?) to all function calls that may return abrupt completions to denote that they should be checked. However, even though ToPrimitive can produce an abrupt completion, the calls of ToPrimitive in steps 10 and 11 do not use the question mark, which is a bug.

Now, let's see how $N$+1-version differential testing can detect the bug in the specification. Consider the example JavaScript code in Figure 3.1(b), which triggers the above specification bug. In the Abstract Equality Comparison algorithm, variables $x$ and $y$ respectively denote `42` and an object with a property named `valueOf` whose value is a function throwing an error. Step 10 calls ToPrimitive with the object as its argument, and the call returns an abrupt completion because the call of `valueOf` throws an error. However, because the call of ToPrimitive in step 10 does not use the question mark, the specification semantics silently ignores the abrupt completion and returns **`false`** as the result of comparison. Using the specification semantics, we can inject assertions to check that the code does not throw any errors as shown in Figure 3.1(c). Then, by running the code with the injected assertions on $N$ JavaScript engines, which throw errors, we can find that the specification may have a bug. Moreover, we can localize the bug using statistical information: because most conformance tests that go through steps 10 and 11 of the algorithm would fail in most of JavaScript engines, we can use the information to localize the bug in the steps 10 and 11 of Abstract Equality Comparison with high probability.

### 3.1.3 Overall Structure

Figure 3.2 depicts the overall structure of $N$+1-version differential testing for $N$ different implementations (engines) and one language specification. It takes a mechanized specification extracted from a given language specification, it first performs the conformance test generation phase, which automatically generates conformance tests that reflect the language syntax and semantics described in the specification. Then, it performs the bug detection and localization phase, which detects and localizes bugs in the engines or the specification by comparing the results of the generated tests on $N$ engines.

The functionalities of each module in the overall structure are as follows:

#### Seed Synthesizer

The first module of the conformance test generation phase is Seed Synthesizer, which synthesizes an initial seed programs using the language syntax. Its main goal is to synthesize (1) a few number of (2) small-sized programs (3) that cover possible cases in the syntax rules as many as possible.

#### Target Selector

Starting from the seed programs generated by Seed Synthesizer as the initial *program pool*, Target Selector selects a target program in the program pool that potentially increases the coverage of the language semantics by the pool. From the selected target program, Program Mutator constructs a new mutated program and adds it to the program pool. When specific criteria, such as an iteration limit, are satisfied, Target Selector stops selecting target programs and returns the program pool as its result.

#### Program Mutator

The main goal of Program Mutator is to generate a new program by mutating a given target program in order to increase the coverage of the language semantics by the program pool. If it fails to generate a new program to increase the semantics coverage, Target Selector retries to select a new target program and repeats this process less than a pre-defined iteration limit.

#### Assertion Injector

Finally, the conformance test generation phase modifies the programs in the pool to generate conformance tests by injecting appropriate assertions reflecting the semantics described in the specification. More specifically, Assertion Injector executes each program in the pool on the mechanized specification and obtains the final state of its execution. It then automatically injects assertions to the program using the final state.

#### Bug Localizer

Then, the second phase executes the conformance tests on $N$ engines and collects their results. For each test, if a small number of engines fail, it reports potential bugs in the engines that fail the test. Otherwise, it reports potential bugs in the specification. In addition, its Bug Localizer module uses *Spectrum Based Fault Localization* (SBFL) [102], a localization technique utilizing the coverage and pass/fail results of test cases, to localize potential bugs.

## 3.2 $N$+1-version Differential Testing for JavaScript

We actualize $N$+1-version differential testing for the JavaScript programming language as JEST, which uses modern JavaScript engines and ECMAScript. This section explains the detail of five core modules of the tool: Seed Synthesizer, Target Selector, Program Mutator, Assertion Injector, and Bug Localizer.

### 3.2.1 Seed Synthesizer

The first module of JEST is Seed Synthesizer, which synthesizes seed programs using two synthesizers: a *non-recursive synthesizer* and a *built-in function synthesizer*. Because the synthesis algorithm is deterministic, the seed programs only depend on the JavaScript syntax and built-in libraries described in a given version of ECMAScript without any randomness.

#### Non-Recursive Synthesizer

The first synthesizer aims to cover as many syntax cases as possible in two steps: 1) to find the shortest string for each non-terminal and 2) to synthesize JavaScript programs using the shortest strings. For presentation brevity, we explain simple cases like terminals and non-terminals, but the implementation supports the extended grammar of ECMAScript such as parametric non-terminals, conditional alternatives, and special terminal symbols.

**Algorithm 1:** Worklist-based Shortest String

```
Input: $\mathbb{R}$ - syntax reduction rules
Output: $M$ - map from non-terminals to shortest strings derivable from them
Function shortestStrings($\mathbb{R}$):
    $M = \varnothing, W =$ a queue that contains $\mathbb{R}$
    while $W \neq \varnothing$ do
        pop $(A, \alpha) \leftarrow W$
        if update($A, \alpha$) then propagate($W, \mathbb{R}, A$)
Function update($A, \alpha$):
    $str$ = an empty string
    forall $s \in \alpha$ do
        if $s$ is a terminal $t$ then $str = str + t$
        else if $s$ is a non-terminal $A' \wedge A' \in M$ then
            $str = str + M[A']$
        else return false
    if $\exists M[A] \wedge \|str\| \geq \|M[A]\|$ then return false
    $M[A] = str$
    return true
Function propagate($W, \mathbb{R}, A$):
    forall $(A', \alpha') \in \mathbb{R}$ do
        if $A \in \alpha'$ then push $(A', \alpha') \rightarrow W$
```

The shortestStrings function in Algorithm 1 shows the first step. We modified McKenize's algorithm [60] that finds random strings to find the shorted string. It takes syntax reduction rules $\mathbb{R}$, a set of pairs of non-terminals and alternatives, and returns a map $M$ from non-terminals to shortest strings derivable from them. It utilizes a worklist $W$, a queue structure that includes syntax reduction rules affected by updated non-terminals. The function initializes the worklist $W$ with all the syntax reduction rules $\mathbb{R}$. Then, for a syntax reduction rule $(A, \alpha)$, it updates the map $M$ via the update function, and propagates updated information via the propagate function. The update function checks whether a given alternative $\alpha$ of a non-terminal $A$ can derive a string shorter than the current shortest one using the current map $M$. If possible, it stores the mapping from the non-terminal $A$ to the newly found shortest string in $M$ and invokes propagate. The propagate function finds all the syntax reduction rules whose alternatives contain the updated non-terminal $A$ and inserts them into $W$. The shortestStrings function repeats this process until the worklist $W$ becomes empty.

**Algorithm 2:** Non-Recursive Synthesize

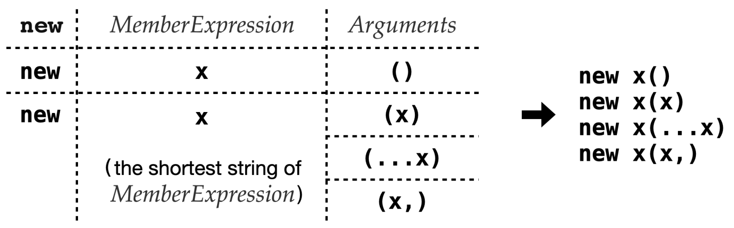
```
Input: $\mathbb{R}$ - syntax reduction rules, $S$ - start symbol
Output: $D$ - set of strings derivable from $S$
Function nonRecSynthesize($\mathbb{R}, S$):
    $V = \varnothing, M =$ shortestStrings($\mathbb{R}$)
    return getProd($M, V, \mathbb{R}, S$)
Function getProd($M, V, \mathbb{R}, A$):
    if $A \in V$ then return $\{M[A]\}$
    $D = \varnothing, V = V \cup \{A\}$
    forall $(A', \alpha) \in \mathbb{R}$ s.t. $A' = A$ do
        $D = D \cup$ getAlt($M, V, \mathbb{R}, A, \alpha$)
    return $D$
Function getAlt($M, V, \mathbb{R}, A, \alpha$):
    $L =$ an empty list
    forall $s \in \alpha$ do
        if $s$ is a terminal $t$ then
            append $(\{t\}, t)$ to $L$
        else if $s$ is a non-terminal $A'$ then
            append (getProd($M, V, \mathbb{R}, A'$), $M[A]$) to $L$
    $D =$ point-wise concatenation of first elements of pairs in $L$ using second elements as
     default ones.
    return $D$
```

Using shortest strings derivable from non-terminals, the nonRecSynthesize function in Algorithm 2 synthesize programs. It takes syntax reduction rules $\mathbb{R}$ and a start symbol $S$. For the first visit with a non-terminal $A$, the getProd function returns strings generated by getAlt with alternatives of the non-terminal $A$. For an already visited non-terminal $A$, it returns the single shortest string $M[A]$. The getAlt function takes a non-terminal $A$ with an alternative $\alpha$ and returns a set of strings derivable from $\alpha$ via point-wise concatenation of strings derived by symbols of $\alpha$. When the numbers of strings derived by symbols are different, it uses the shortest strings derived by symbols as default strings.

*MemberExpression* **:**
  *PrimaryExpression*
  *MemberExpression* **[** *Expression* **]**
  *MemberExpression* **.** *IdentifierName*
  **new** *MemberExpression* *Arguments*

Figure 3.3: The *MemberExpression* production in ES11

For example, Figure 3.3 shows a simplified *MemberExpression* production in ES11. For the first step, we find the shortest string for each non-terminal: `()` for *Arguments* and `x` for the other non-terminals. Note that we use pre-defined shortest strings for identifiers and literals such as `x` for identifiers and `0` for numerical literals. In the next step, we synthesize strings derivable from *MemberExpression*. The first alternative is a single non-terminal *PrimaryExpression*, which is never visited. Thus, it generates all cases of *PrimaryExpression*. The fourth alternative consists of one terminal **new** and two non-terminals *MemberExpression* and *Arguments*. Because *MemberExpression* is already visited, it generates a single shortest string `x`. For the first visit of *Arguments*, it generates all cases: `()`, `(x)`, `(...x)`, and `(x,)`. Note that the numbers of strings generated for symbols are different. In such cases, we use the shortest strings for symbols like `x` for *MemberExpression* as follows:



### Built-in Function Synthesizer

JavaScript supports diverse built-in functions for primitive values and built-in objects. To synthesize JavaScript programs that invoke built-in functions, we extract the information of each built-in function from the mechanized ECMAScript. We utilize the `Function.prototype.call` function to invoke built-in functions to easily handle the **`this`** object in Program Mutator; we use a corresponding object or **`null`** as the **`this`** object by default. In addition, we synthesize function calls with optional and variable number of arguments and built-in constructor calls with the **`new`** keyword.

Consider the following `Array.prototype.indexOf` function for JavaScript array objects that have a parameter *searchElement* and an optional parameter *fromIndex*:

**Array.prototype.indexOf (** *searchElement* **[ ,** *fromIndex* **] )**

the synthesizer generates the following calls with an array object or **`null`** as the **`this`** object as follows:

```
Array.prototype.indexOf.call(new Array(), 0);
Array.prototype.indexOf.call(new Array(), 0, 0);
Array.prototype.indexOf.call(null, 0);
Array.prototype.indexOf.call(null, 0, 0);
```

Moreover, `Array` is a built-in function and a built-in constructor with a variable number of arguments. Thus, we synthesize the following six programs for `Array`:

```
Array();         Array(0);         Array(0, 0);
new Array();     new Array(0);     new Array(0, 0);
```

### 3.2.2 Target Selector

From the synthesized programs, Target Selector selects a target program to mutate to increase the semantics coverage of the program pool. Consider the Abstract Equality Comparison algorithm in Figure 3.1(a) again where the first step has the condition "If Type($x$) is the same as Type($y$)." Assuming that the current pool has the following three programs:

```
1 + 2;              true == false;              0 == 1;
```

because later two programs that perform comparison have values of the same type, the pool covers only the true branch of the condition in the algorithm. To cover its false branch, Target Selector selects any program that covers the true branch like `true == false;` and Program Mutator mutates it to `42 == false;` for example. Then, since the mutated program covers the false branch, the pool is extended as follows:

```
1 + 2;              true == false;              0 == 1;               42 == false;
```

and this process repeats until the semantics coverage converges.

### 3.2.3 Program Mutator

JEST increases the semantics coverage of the program pool by mutating programs randomly using five mutation methods: 1) a *random mutation*, 2) a *nearest syntax tree mutation*, 3) a *string substitution*, 4) a *object substitution*, and 5) a *statement insertion*.

#### Random Mutation

The first naïve method is to randomly select a statement, a declaration, or an expression in a given program and to replace it with a randomly selected one from a set of syntax trees generated by the non-recursive synthesizer. For example, it may mutate a program `var x = 1 + 2;` by replacing its random expression `1` with a random expression `true` producing `var x = true + 2;`.

#### Nearest Syntax Tree Mutation

The second method targets uncovered branches in abstract algorithms. When only one branch is covered by a program, it finds the nearest syntax tree in the program that reaches the branch in the algorithm, and replaces the nearest syntax tree with a random syntax tree derivable from the same syntax production. For example, consider a JavaScript program: `var x = "" + (1 == 2);`. While it covers the false branch of the first step of Abstract Equality Comparison in Figure 3.1(a), assume that no program in the program pool can cover its true branch. Then, the mutator targets this branch, finds its nearest syntax tree `1 == 2` in the program, and replaces it with a random syntax tree.

#### String Substitution

We collect all string literals used in conditions of the algorithms in ES11 and use them for the random expression substitution. Because most string literals in the specification represent corner cases such as `-0`, `Infinity`, and `NaN`, they are necessary for mutation to increase the semantics coverage. For example, the semantics of the [[DefineOwnProperty]] internal method of array exotic objects depends on whether the value of its parameter `P` is `"length"` or not.

#### Object Substitution

We also collect string literals and symbols used as arguments of object property access algorithms in ES11, randomly generate objects using them, and replace random expressions with the generated objects. Because some abstract algorithms in the specification access object properties using HasProperty, GetMethod, Get, and OrdinaryGetOwnProperty, objects with such properties are necessary for mutation to achieve high coverage. Thus, the mutator mutates a randomly selected expression in a program with a randomly generated object that has properties whose keys are from collected string literals and symbols.

#### Statement Insertion

To synthesize more complex programs, the mutator inserts random statements at the end of randomly selected blocks like top-level code and function bodies. We generate random statements using the non-recursive synthesizer with pre-defined special statements. The special statements are control diverters, which have high chances of changing execution paths, such as function calls, `return`, `break`, and `throw` statements. The mutator selects special statements with a higher probability than the statements randomly synthesized by the non-recursive synthesizer.

### 3.2.4 Assertion Injector

After generating JavaScript programs, Assertion Injector injects assertions to them using their final states as specified in ECMAScript. It first obtains the final state of a given program from the mechanized specification and injects seven kinds of assertions in the beginning of the program. To check the final state after executing all asynchronous jobs, we enclose assertions with `setTimeout` to wait 100 ms when a program uses asynchronous features such as `Promise` and `async`:

```
... /* a given program */
setTimeout(() => { ... /* assertions */ }, 100)
```

#### Exceptions

JavaScript supports both internal exceptions like `SyntaxError` and `TypeError` and custom exceptions with the keyword `throw`. Note that catching such exceptions using the `try-catch` statement may change the program semantics. For example, the following does not throw any exception:

```
var x; function x() {}
```

but the following:

```
try { var x; function x() {} } catch (e) {}
```

throws `SyntaxError` because declarations of a variable and a function with the same name are not allowed in `try-catch`.

To resolve this problem, we exploit a comment in the first line of a program. If the program throws an internal exception, we tag its name in the comment. Otherwise, we tag `// Throw` for a custom exception and `// Normal` for normal termination. Using the tag in the comment, JEST checks the execution result of a program in each engine.

### Aborts

The mechanized semantics of ECMAScript can abort due to unspecified cases. For example, consider the following JavaScript program:

```
var x = 42; x++;
```

The postfix increment operator (`++`) increases the number value stored in the variable `x`. However, because of a typo in the Evaluation algorithm for such update expressions in ES11, the behavior of the program is not defined in ES11. To represent this situation in the conformance test, we tag `Abort` in the comment as follows:

```
// Abort
var x = 42; x++;
```

### Variable Values

We inject assertions that compare the values of variables with expected values. To focus on variables introduced by tests, we do not check the values of pre-defined variables like built-in objects. For numbers, we distinguish `-0` from `+0` using division by zero because `1/-0` and `1/+0` produce negative and positive infinity values, respectively. The following example checks whether the value of `x` is `3`:

```
var x = 1 + 2;
$assert.sameValue(x, 3);
```

### Object Values

To check the equality of object values, we keep a representative path for each object. If the injector meets an object for the first time, it keeps the current path of the object as its representative path and injects assertions for the properties of the object. Otherwise, the injector adds assertions to compare the values of the objects with the current path and the representative path. In the following example:

```
var x = {}, y = {}, z = { p: x, q: y };
$assert.sameValue(z.p, x);
$assert.sameValue(z.q, y);
```

because the injector meets two different new objects stored in `x` and `y`, it keeps the paths `x` and `y`. Then, the object stored in `z` is also a new object but its properties `z.p` and `z.q` store already visited objects values. Thus, the injector inserts two assertions that check whether `z.p` and `x` have the same object value and `z.q` and `y` as well. To handle built-in objects, we store all the paths of built-in objects in advance.

### Object Properties

Checking object properties involves checking four attributes for each property. We implement a helper `$verifyProperty` to check the attributes of each property for each object. For example, the following code checks the attributes of the property of `x.p`:

```
var x = { p: 42 };
$verifyProperty(x, "p", { value: 42.0,      writable: true,
                          enumerable: true, configurable: true });
```

### Property Keys

Since ECMAScript 2015 (ES6), the specification defines orders between property keys in objects. We check the order of property keys by `Reflect.ownKeys`, which takes an object and returns an array of the object's property keys. We implement a helper `$assert.compareArray` that takes two arrays and compares their lengths and contents. For example, the following program checks the property keys and their order of the object in `x`:

```
var x = {[Symbol.match]: 0, p: 0, 3: 0, q: 0, 1: 0}
$assert.compareArray(
    Reflect.ownKeys(x),
    ["1", "3", "p", "q", Symbol.match]
);
```

### Internal Methods and Slots

While internal methods and slots of JavaScript objects are generally inaccessible by users, the names in the following are accessible by indirect getters:

| Name | Indirect Getter |
|---|---|
| [[Prototype]] | `Object.getPrototypeOf(x)` |
| [[Extensible]] | `Object.isExtensible(x)` |
| [[Call]] | `typeof f === "function"` |
| [[Construct]] | `Reflect.construct(function(){},[],x)` |

The internal slot [[Prototype]] represents the prototype object of an object, which is available by a built-in function `Object.getPrototypeOf`. The internal slot [[Extensible]] is also available by a built-in function `Object.isExtensible`. The internal methods [[Call]] and [[Construct]] represent whether a given object is a function and a constructor, respectively. Because the methods are not JavaScript values, we simply check their existence using helpers `$assert.callable` and `$assert.constructable`. For [[Call]], we use the `typeof` operator because it returns `"function"` if and only if a given value is an object with the [[Call]] method. For [[Construct]] method, we use the `Reflect.construct` built-in function that checks the existence of the [[Construct]] methods and invokes it. To avoid invoking [[Construct]] unintentionally, we call `Reflect.construct` with a dummy function `function(){}` as its first argument and a given object as its third argument. For example, the following code shows how the injector injects assertions for internal methods and slots:

```
function f() {}
$assert.sameValue(Object.getPrototypeOf(f), Function.prototype);
$assert.sameValue(Object.isExtensible(x), true);
$assert.callable(f);
$assert.constructable(f);
```

## 3.2.5 Bug Localizer

The bug detection and localization phase uses the execution results of given conformance tests on multiple JavaScript engines. If a small number of engines fail in running a specific conformance test, the engines may have bugs causing the test failure. If most engines fail for a test, the test may be incorrect, which implies a bug in the specification.

When we have a set of failed test cases that may contain bugs of an engine or a specification, we classify the test cases using their failure messages and give ranks between possible buggy program elements to localize the bug. We use Spectrum Based Fault Localization (SBFL) [102], which is a ranking technique based on likelihood of being faulty for each program element. We use the following formula called $ER1_b$, which is one of the best SBFL formulae theoretically analyzed by Xie et al. [105]:

$$n_{\text{ef}} - \frac{n_{\text{ep}}}{n_{\text{ep}} + n_{\text{np}} + 1}$$

where $n_{\text{ef}}$, $n_{\text{ep}}$ , $n_{\text{nf}}$, and $n_{\text{np}}$ represent the number of test cases; subscripts $_\text{e}$ and $_\text{n}$ respectively denote whether a test case touches a relevant program element or not, and subscripts $_\text{f}$ and $_\text{p}$ respectively denote whether the test case is failed or passed.

We use abstract algorithms of ECMAScript as program elements used for SBFL. To improve the localization accuracy, we use method-level aggregation [91]. It first calculates SBFL scores for algorithm steps and aggregates them up to algorithm-level using the highest score among those from steps of each algorithm.

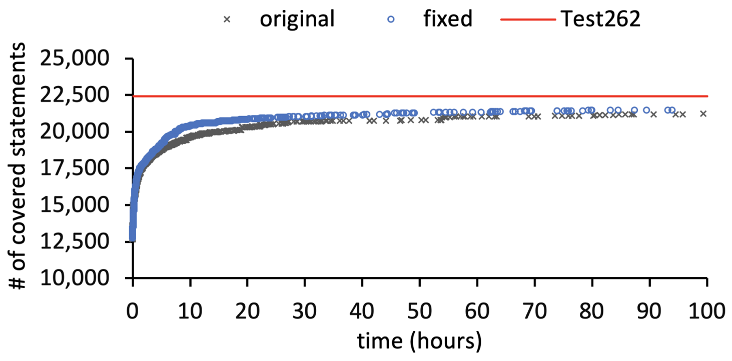
## 3.3 Evaluation

To evaluate JEST that performs $N$+1-version differential testing of JavaScript engines and its specification, we applied the tool to four JavaScript engines that fully support modern JavaScript features and the latest specification, ECMAScript 2020 (ES11, 2020). Our experiments use the following four JavaScript engines, all of which support ES11:

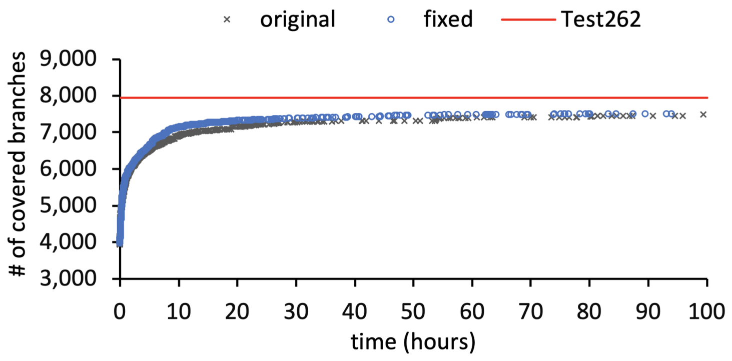
- **V8(v8.5)**[1]**:** An open-source high-performance engine for JavaScript and WebAssembly developed by Google
- **GraalJS(v20.1.0)**[2] A JavaScript implementation built on GraalVM, which is a Java Virtual Machine (JVM) based on HotSpot/OpenJDK developed by Oracle
- **QuickJS(2020-04-12)**[3]**:** A small and embedded JavaScript engine developed by Fabrice Bellard and Charlie Gordon
- **Moddable XS(v10.3.0)**[4]**:** A JavaScript engine at the center of the Moddable SDK, which is a combination of development tools and runtime software to create applications for micro-controllers

To extract a mechanized specification from ECMAScript, we utilize the tool JISET, which is a JavaScript IR-based semantics extraction toolchain, to automatically generate a JavaScript interpreter from ECMAScript. To focus on the core semantics of JavaScript, we consider only the semantics of strict mode JavaScript code that pass syntax checking including the EarlyError rules. To filter out JavaScript code that are not strict or fail syntax checking, we utilize the syntax checker of the most reliable JavaScript engine, V8. We performed our experiments on a machine equipped with 4.0GHz Intel(R) Core(TM) i7-6700k and 32GB of RAM (Samsung DDR4 2133MHz 8GB*4). We evaluated JEST with the following four research questions:

[1]https://v8.dev/
[2]https://github.com/graalvm/graaljs#current-status:
[3]https://bellard.org/quickjs/
[4]https://blog.moddable.com/blog/xs10/



(a) Statement coverage



(b) Branch coverage

Figure 3.4: The semantics coverage changes during the test generation phase

- **RQ1 (Coverage of Generated Tests)** Is the semantics coverage of the tests generated by JEST comparable to that of Test262, the official conformance test suite for ECMAScript, which is manually written?
- **RQ2 (Accuracy of Bug Localization)** Does JEST localize bug locations accurately?
- **RQ3 (Bug Detection in JavaScript Engines)** How many bugs of four JavaScript engines does JEST detect?
- **RQ4 (Bug Detection in ECMAScript)** How many bugs of ES11 does JEST detect?

## 3.3.1 Coverage of Generated Tests

JEST generates the seed programs via Seed Synthesizer, which synthesizes 1,125 JavaScript programs in about 10 seconds and covers 97.78% (397/406) of reachable alternatives in the syntax productions of ES11. Among them, we filtered out 602 programs that do not increase the semantics coverage and started the mutation iteration with 523 programs. Figure 3.4 shows the change of semantics coverage of the program pool during the iterative process in 100 hours. The left and right graphs present the statement and branch coverages, respectively, and the top red line denotes the coverage of Test262. We generated conformance tests two times before and after fixing bugs detected by JEST because the specification bugs affected the semantics coverage. In each graph, dark gray X marks and blue O marks denote the semantics coverage of generated tests before and after fixing bugs. The semantics that we target in ES11 consists of 1,550 algorithms with 24,495 statements and 9,596 branches. For the statement coverage,

Table 3.1: Number of generated programs and covered branches of mutation methods

| **Mutation Method** | **Program** | **Branch (Avg.)** |
|---|---|---|
| Nearest Syntax Tree Mutation | 459 | $1,230(2.68)$ |
| Random Mutation | 337 | $1,153(3.42)$ |
| Statement Insertion | 209 | 650(3.11) |
| Object Substitution | 169 | 491(2.91) |
| String Substitution | 3 | 3(1.00) |
| **Total** | $1,177$ | $3,527(3.00)$ |

Test262 covers 22,440 (91.61%) statements. The initial program pool covers 12,768 (52.12%) statements and the final program pool covers 21,230 (86.67%) and 21,482 (87.70%) statements before and after fixing bugs, respectively. For the branch coverage, Test262 covers 7,956 (82.91%) branches. The initial program pool covers 3,987 (41.55%) branches and the final program pool covers 7,480 (77.95%) and 7,514 (78.30%) branches before and after fixing bugs, respectively.

Table 3.1 shows the number of synthesized programs and covered branches for each mutation method during the test generation phase. JEST synthesized 1,177 new programs that cover 3,527 more branches than the initial program pool. Among five mutation methods, the nearest syntax tree mutation is the most contributed method (459 programs and 1,230 covered branches) and the least one is the string substitution (3 programs and 3 covered branches).

Finally, JEST generates 1,700 JavaScript programs and their average number of lines is 2.01. After injecting assertions, their average number of lines becomes 8.45. Compared to Test262, the number of generated tests are much smaller and their number of lines are also shorter than those of tests in Test262. Test262 provides 16,251 tests for the same range of semantics and their average number of lines is 49.67.

## 3.3.2 Accuracy of Bug Localization

To detect more bugs using more diverse programs, we repeated the conformance test generation phase for ten times. We executed the generated conformance tests on four JavaScript engines to find bugs in the engines and the specification. After inferring locations of the bugs in the engines or the specification based on the majority of the execution results, we manually checked whether the bugs are indeed in the engines or the specification. The following table shows that our method works well:

| # **Failed Engines** | **1** | **2** | **3** | **4** | **Total** | **Average** |
|---|---|---|---|---|---|---|
| Engine Bugs | 38 | 6 | 0 | 0 | 44 | 1.14 |
| Specification Bugs | 0 | 0 | 10 | 17 | 27 | 3.63 |

For engine bugs, the average number of engine failures is 1.14 while the average number of failed engines for specification bugs is 3.63. As we expected, when most engines fail for a test, the specification may have a bug.

Based on the results of conformance tests on four JavaScript engines, we localized the specification or engine bugs on the *semantics* of ES11. Among 71 bugs, we excluded 7 syntax bugs and localized only 64 semantics bugs. Figure 3.5 shows the ranks of algorithms that caused the semantics bugs. The average rank is 3.19, and 82.8% of the algorithms causing the bugs are ranked less than 5, 93.8% less than 10, and 98.4% less than 15. Note that the location of one bug is ranked 21 because of the limitation of SBFL; its localization accuracy becomes low for a small number of failed test cases.

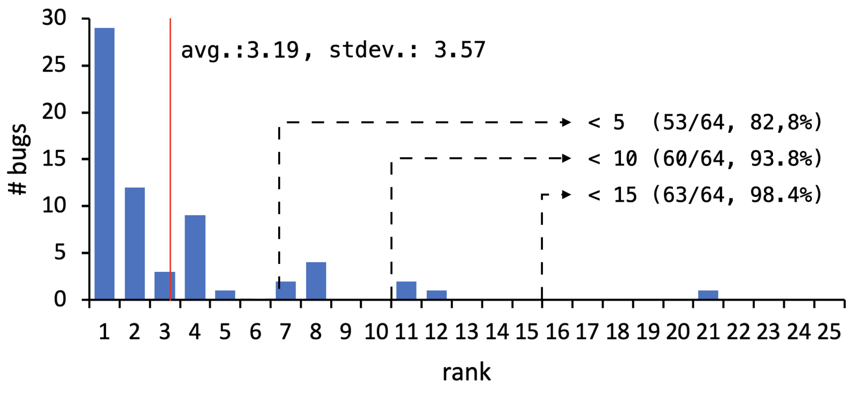


Figure 3.5: Ranks of algorithms that caused the bugs detected by JEST

Table 3.2: The number of engine bugs detected by JEST

| **Engines** | Exc | Abort | Var | Obj | Desc | Key | In | **Total** |
|---|---|---|---|---|---|---|---|---|
| V8 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 2 |
| GraalJS | 6 | 0 | 0 | 0 | 2 | 8 | 0 | 16 |
| QuickJS | 3 | 0 | 1 | 0 | 0 | 2 | 0 | 6 |
| Moddable XS | 12 | 0 | 0 | 0 | 3 | 5 | 0 | 20 |
| **Total** | 21 | 0 | 1 | 0 | 5 | 17 | 0 | 44 |

## 3.3.3 Bug Detection in JavaScript Engines

From four JavaScript engines, JEST detected 44 bugs: 2 from V8, 16 from GraalJS, 6 from QuickJS, and 20 from Moddable XS. Table 3.2 presents how many bugs for each assertion are detected for each engine. We injected seven kinds of assertions: exceptions (Exc), aborts (Abort), variable values (Var), object values (Obj), object properties (Desc), property keys (Key), and internal methods and slots (In). The effectiveness of bug finding is different for different assertions. The Exc and Key assertions detected engine bugs the most; out of 44 bugs, the former detected 21 bugs and the latter detected 17 bugs. Desc and Var detected 5 and 1 bugs, respectively, but the other assertions did not detect any engine bugs.

The most reliable JavaScript engine is V8 because JEST found only two bugs and the bugs are due to specification bugs in ES11. Because V8 strictly follows the semantics of functions described in ES11, it also implemented wrong semantics that led to ES11-1 and ES11-2 listed in Table 3.3. The V8 team confirmed the bugs and fixed them.

We detected 16 engine bugs in GraalJS and one of them caused an engine crash. When we apply the prefix increment operator for `undefined` as `++undefined`, GraalJS throws `java.lang.IllegalStateException`. Because it crashes the engine, developers even cannot catch the exception as follows:

```
try { ++undefined; } catch(e) { }
```

The GraalJS team has been fixing the bugs we reported and asked whether we plan to publish the conformance test suite, because the tests generated by JEST detected many semantics bugs that were not detected by other conformance tests: "Right now, we are running Test262 and the V8 and Nashorn unit test suites in our CI for every change, it might make sense to add your suite as well."

In QuickJS, JEST detected 6 engine bugs, most of which are due to corner cases of the function

Table 3.3: Specification bugs in ECMAScript 2020 (ES11) detected by JEST

| Name | Feature | Assert | # | Description | Known | Created | Resolved | Existed |
|---|---|---|---|---|---|---|---|---|
| ES11-1 | Function | Key | 12 | Wrong order between property keys for functions | O | 2019-02-07 | 2020-04-11 | 429 days |
| ES11-2 | Function | Key | 8 | Missing property `name` for anonymous functions | O | 2015-06-01 | 2020-04-11 | 1,776 days |
| ES11-3 | Loop | Exc | 1 | Returning iterator objects instead of iterator records in ForIn/OfHeadEvaluation for `for-in` loops | O | 2017-10-17 | 2020-04-30 | 926 days |
| ES11-4 | Expression | Abort | 4 | Using the wrong variable `oldvalue` instead of `oldValue` in Evaluation of *UpdateExpression* | O | 2019-09-27 | 2020-04-23 | 209 days |
| ES11-5 | Expression | Exc | 1 | Unhandling abrupt completion in Abstract Equality Comparison | O | 2015-06-01 | 2020-04-28 | 1,793 days |
| ES11-6 | Object | Exc | 1 | Unhandling abrupt completion in Evaluation of *PropertyDefinition* for object literals | X | 2019-02-07 | 2020-11-06 | 638 days |

semantics. For example, the following code should throw a `ReferenceError` exception:

```
function f (... { x = x }) { return x; } f()
```

because the variable `x` is not yet initialized when it tries to read the right-hand side of `x = x`. However, since QuickJS assumes that the initial value of `x` is `undefined`, the function call `f()` returns `undefined`. The QuickJS team confirmed our bug reports and it has been fixing the bugs.

JEST found the most bugs in Moddable XS; it detected 20 bugs for various language features such as optional chains, `Number.prototype.toString`, iterators of `Map` and `Set`, and complex assignment patterns. Among them, optional chains are newly introduced in ES11, which shows that our approach is applicable to finding bugs in new language features. We reported all the bugs found, and the Moddable XS team has been fixing them. They showed interests in using our test suite: "As you know, it is difficult to verify changes because the language specification is so big. Test262, as great a resource as it is, is not definitive."

### 3.3.4 Bug Detection in ECMAScript

JEST detected 27 specification bugs in ES11, the latest ECMAScript in 2020. Table 3.3 summarizes the bugs categorized by their root causes (ES11-1 to ES11-6) with the related JavaScript language features (**Feature**), the number of specification bugs (#), the kind of assertion used to detect them (**Assertion**), whether they are already known bugs (**Known**), when they are created (**Created**) and resolved (**Resolved**), and how long they lasted (**Existed**). Five out of six categories (ES11-1 to ES11-5) were already reported and fixed in the draft version of ES12 in August 2020 but ES11-6 was never reported before. Therefore, we reported it to TC39, and they confirmed and fixed it in ES12.

ES11-1 contains 12 bugs; it is due to a wrong order between property keys of all kinds of function values such as `async` and generator functions, arrow functions, and classes. For example, if we define a class declaration with a name `A` (`class A {}`), three properties are defined in the function stored in the variable `A`: `length` with a number value `0`, `prototype` with an object, and `name` with a string `"A"`. The problem is the different order of their keys because of the wrong order of their creation. From

ECMAScript 2015 (ES6), the order between property keys is no more implementation-dependent but it is related to the creation order of properties. While the order of property keys in the class `A` should be `[length, prototype, name]` according to the semantics of ES11, the order is `[length, name, prototype]` in three engines except V8. We found that it was already reported as a specification bug; we reported it to V8 and they fixed it. This bug was created on February 7, 2019 and TC39 fixed it on April 11, 2020; the bug lasted for 429 days.

ES11-2 contains 8 bugs that are due to the missing property `name` of anonymous functions. Until ES5.1, anonymous functions, such as an identity arrow function `x => x`, had their own property `name` with an empty string `""`. While ES6 removed the `name` property from anonymous functions, three engines except V8 still create the `name` property in anonymous functions. We also found that it was reported as a specification bug and reported it to V8, and it will be fixed in V8.

The bug in ES11-3 comes from the misunderstanding of the term "iterator object" and "iterator record". The algorithm ForIn/OfHeadEvaluation should return an iterator record, which is an implicit record containing only internal slots. However, In ES11, it returns an iterator object, which is a JavaScript object with some properties related to iteration. It causes a `TypeError` exception when executing the code `for(var x in {});` according to ES11 but all engines execute the code normally without any exceptions. This bug was resolved by TC39 on April 30, 2020.

ES11-4 contains four bugs caused by a typo for the variable in the semantics of four different update expressions: `x++`, `x--`, `++x`, and `--x`. In each Evaluation of four kinds of *UpdateExpression*, there exists a typo `oldvalue` in step 3 instead of `oldValue` declared in step 2. JEST could not execute the code `x++` using the semantics of ES11 because of the typo. For this case, we directly pass the code to Bug Localizer to test whether the code is executable in real-world engines and to localize the bug. Of course, four JavaScript engines executed the update expressions without any issues and this bug was resolved by TC39 on April 23, 2020.

Two bugs in ES11-5 and ES11-6 are caused by unhandling of abrupt completions in abstract equality comparison and property definitions of object literals, respectively. The bug in ES11-5 was confirmed by TC39 and was fixed on April 28, 2020. The bug in ES11-6 was a genuine one, and we reported it and received a confirmation from TC39 on November 6, 2020; the bug lasted for 638 days.

# Chapter 4. JSTAR: Type Analysis for ECMAScript

This chapter introduces JSTAR, a JavaScript Specification Type Analyzer using Refinement. In this thesis, we introduce two different tools to check the validity of ECMAScript. While JEST (Chapter 3) checks the validity using $N{+}1$-version differential testing with JavaScript engines, JSTAR performs a type analysis for the extracted mechanized specification to check it. The main contributions of this work include the following:

- We present JSTAR, the first tool that performs a *type analysis* on ECMAScript written in a natural language to check the correctness of JavaScript language specifications. JSTAR automatically detects type-related specification bugs such as unknown variables, duplicated variables, missing parameters, assertion failures, ill-typed operands, and unchecked abrupt completion bugs.
- We present a *condition-based refinement* for type analysis of ECMAScript to reduce the number of false-positive bugs by enhancing the analysis precision. We show that the refinement technique increases the analysis precision from 33.0% to 59.2% by removing 122 false bugs and detecting one more true bug.
- We demonstrate the practicality of JSTAR. It took 137.3 seconds on average to perform a type analysis for each version of ECMAScript and detected 157 type-related specification bugs with 59.2% precision; 93 out of 157 bugs are true bugs. Among them, JSTAR newly detected 14 bugs, and the ECMAScript committee confirmed them all.

## 4.1 Overview

In this section, we demonstrate the overall structure of JSTAR depicted in Figure 4.1. It consists of three phases: 1) specification extraction, 2) type analysis, and 3) bug detection.

### 4.1.1 Specification Extraction

As described in Chapter 2, JISET extracts a JavaScript mechanized specification from ECMAScript, including generated parsers for syntax and compiled $\text{IR}_{\text{ES}}$ functions for semantics. JSTAR utilizes it to extract JavaScript types and even specification types used in ECMAScript and perform type analysis on the compiled $\text{IR}_{\text{ES}}$ functions.

#### Syntax and Semantics

ECMAScript describes the JavaScript syntax in an EBNF notation and the semantics using abstract algorithms written in a structured natural language. From ECMAScript, JISET synthesizes AST structures for syntax and compiles the abstract algorithms to $\text{IR}_{\text{ES}}$ functions with parameters and local variables for semantics. For example, the algorithm step “Let *baseObj* be ! ToObject($V$.[[Base]])” is compiled to an $\text{IR}_{\text{ES}}$ instruction `let baseObj = [! (ToObject V.Base)]`. To make it suitable for type analysis, we modify $\text{IR}_{\text{ES}}$ as formally defined in Section 4.2.1.

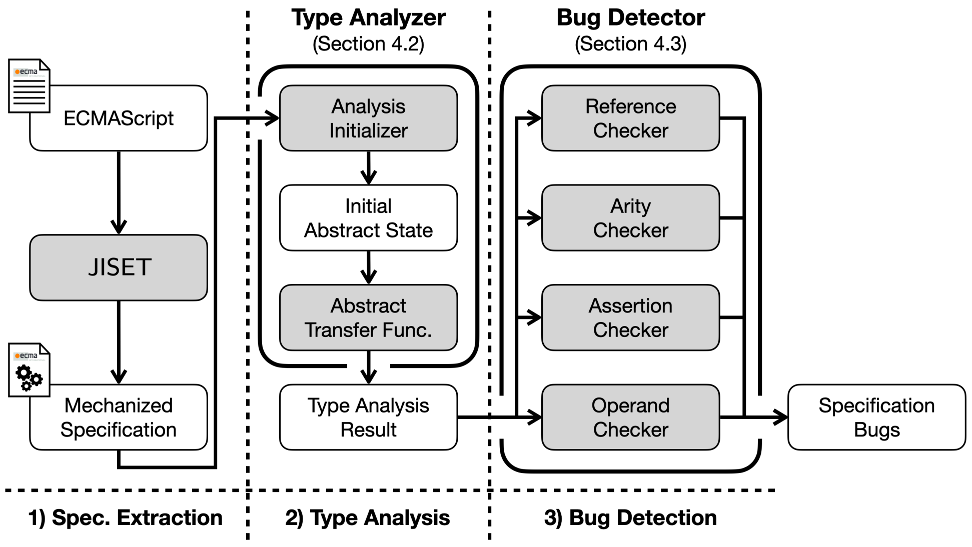


Figure 4.1: JSTAR: a type analyzer and a bug detector for mechanized specifications extracted from ECMAScript by JISET

### Types

In addition to JavaScript types, JSTAR represents three kinds of specification types. First, because ASTs are values in abstract algorithms, they can be stored in variables and passed as function arguments. For ASTs, we use their production names as their types and automatically link their corresponding syntax-directed algorithms to their fields. Second, ECMAScript supports various record types and fields whose possible values are defined in their corresponding tables. For example, "Table 9: Completion Record Fields" in the latest ECMAScript describes the fields of the completion records. Thus, we manually model the fields of record types based on the tables in the latest version and use them in a type analysis. Third, for list-like structures, we define types for empty list `[]` and parametric lists `[`$\tau$`]`.

## 4.1.2 Type Analysis

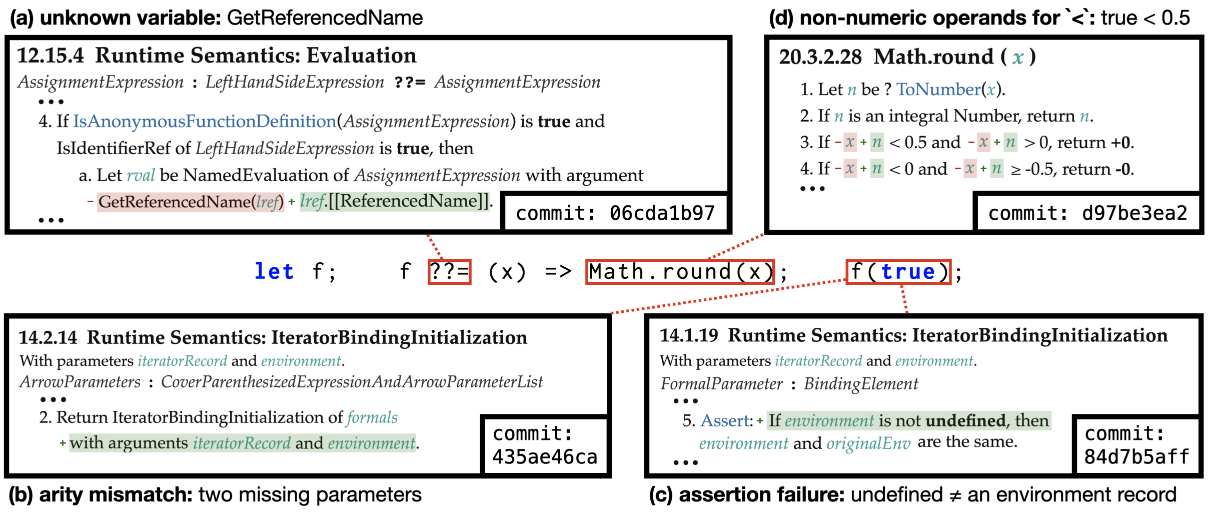
JSTAR performs a type analysis with flow-sensitivity and type-sensitivity for arguments. Each function is split into multiple flow- and type-sensitive views, and an abstract state stores mapping from views to corresponding abstract environments. To handle views separately, we use a worklist algorithm. The type analyzer consists of two sub-modules: an Analysis Initializer and an Abstract Transfer Function.

### Analysis Initializer

It defines the initial abstract state and the initial set of views for a worklist. ECMAScript provides three kinds of abstract algorithms: *normal*, *syntax-directed*, and *built-in*. As for entry points of type analysis, we use syntax-directed algorithms and built-in algorithms because they have their parameter types. For each entry point, the initializer defines its abstract environment with parameter types and adds the flow- and type-sensitive views of the entry point to the worklist.

### Abstract Transfer Function

For each iteration, the abstract transfer function gets a specific view from the worklist and updates the abstract environments of the next views based on the abstract semantics. It adds the next views to the



(a) unknown variable: GetReferencedName

```
12.15.4 Runtime Semantics: Evaluation
AssignmentExpression : LeftHandSideExpression ??= AssignmentExpression
  •••
  4. If IsAnonymousFunctionDefinition(AssignmentExpression) is true and
     IsIdentifierRef of LeftHandSideExpression is true, then
       a. Let rval be NamedEvaluation of AssignmentExpression with argument
          - GetReferencedName(lref) + lref.[[ReferencedName]].
  •••
commit: 06cda1b97
```

(d) non-numeric operands for `<`: true < 0.5

```
20.3.2.28 Math.round ( x )
  1. Let n be ? ToNumber(x).
  2. If n is an integral Number, return n.
  3. If - x + n < 0.5 and - x + n > 0, return +0.
  4. If - x + n < 0 and - x + n ≥ -0.5, return -0.
  •••
commit: d97be3ea2
```

```
let f;    f ??= (x) => Math.round(x);    f(true);
```

```
14.2.14 Runtime Semantics: IteratorBindingInitialization
With parameters iteratorRecord and environment.
ArrowParameters : CoverParenthesizedExpressionAndArrowParameterList
  •••
  2. Return IteratorBindingInitialization of formals
     + with arguments iteratorRecord and environment.
commit: 435ae46ca
```

(b) arity mismatch: two missing parameters

```
14.1.19 Runtime Semantics: IteratorBindingInitialization
With parameters iteratorRecord and environment.
FormalParameter : BindingElement
  •••
  5. Assert: + If environment is not undefined, then
     environment and originalEnv are the same.
  •••
commit: 84d7b5aff
```

(c) assertion failure: undefined ≠ an environment record

Figure 4.2: An example JavaScript program with related previous specification bugs and their bug fixes

worklist if it changes their abstract environments, and the iteration finishes when the worklist becomes empty. To increase the analysis precision, we perform a condition-based refinement for an abstract environment when the current control point is a branch or an assertion, as described in Section 4.2.3.

### 4.1.3 Bug Detection

To detect specification bugs utilizing the type analysis, we develop four checkers in a bug detector. We explain the targets of the checkers with an example JavaScript program that contains related previous specification bugs and their bug fixes, as shown in Figure 4.2.

#### Reference Checker

The example JavaScript program first defines a variable `f` without initialization, which has the value `undefined`. It then assigns an anonymous function to `f` using the operator `??=`. While the corresponding Evaluation algorithm in Figure 4.2(a) originally used the GetReferencedName algorithm to get a reference name on line 4.a, a contributor removed the GetReferencedName algorithm and replaced all its invocations with accesses of the field [[ReferencedName]] on October 28, 2020. However, the contributor missed several cases, including the semantics of `??=`, which was fixed by another contributor on November 3, 2020. Thus, the unknown variable bug for GetReferencedName lasted for seven days, which the reference checker can detect.

#### Arity Checker

The program finally calls `f` with an argument **`true`**. During the initialization of the function call, IteratorBindingInitialization in Figure 4.2(b) is executed with additional parameters *iteratorRecord* and *environment* to assign argument values to parameters. However, a contributor missed passing additional arguments to them on line 2 in IteratorBindingInitialization of *ArrowParameters* on September 6, 2018. As a result, it caused an arity mismatch bug, which lasted for 533 days until another contributor fixed it on February 20, 2020. The arity checker can detect such arity mismatches.

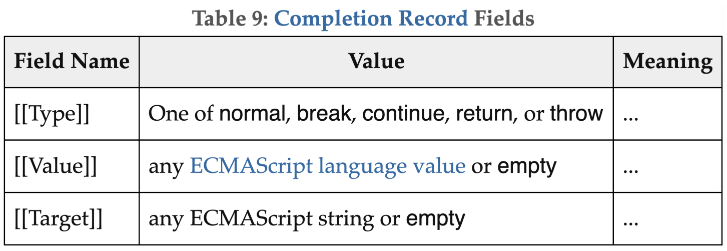


Table 9: Completion Record Fields

| Field Name | Value | Meaning |
|---|---|---|
| [[Type]] | One of normal, break, continue, return, or throw | ... |
| [[Value]] | any ECMAScript language value or empty | ... |
| [[Target]] | any ECMAScript string or empty | ... |

Figure 4.3: Fields of completion records in ECMAScript 2020

### Assertion Checker

During the initialization of the function call, IteratorBindingInitialization of *FormalParameter* in Figure 4.2(c) contains another bug. Even though the additional *environment* parameter may contain `undefined`, a contributor did not consider it on line 5 in the initial commit of the open development process on September 22, 2015. It caused an assertion failure bug, which lasted for 1,297 days until another contributor fixed it on April 10, 2019. The assertion checker can detect such assertion failures.

### Operand Checker

After the function call initialization, the parameter `x` has the value **`true`**, and `Math.round` in Figure 4.2(d) is invoked with the argument **`true`**. The `Math.round` built-in library first converts the given parameter $x$ to its corresponding number value $n$ using ToNumber, and performs the remaining steps using $n$. However, a contributor mistakenly used $x$ instead of $n$ on lines 3 and 4 on September 11, 2020. This bug caused the algorithm to compare the boolean value **`true`** with the numeric value 0.5 or 0 in the example code. This bug lived for two days until another contributor fixed it, and the operand checker can detect them.

In the remainder of this chapter, we explain how to perform type analysis for $\text{IR}_{\text{ES}}$ functions and increase the analysis precision using the condition-based refinement (Section 4.2). Then, we present how to detect type-related specification bugs (Section 4.3). Finally, we evaluate JSTAR to check its performance, precision, and effectiveness of refinement and detection of new bugs (Section 4.4).

## 4.2 Type Analyzer

This section formally defines a modified $\text{IR}_{\text{ES}}$ and its type analysis and presents a condition-based refinement of the type analysis to improve the analysis precision.

### 4.2.1 Intermediate Representation

$$
\begin{array}{llrl}
\text{Functions} & \mathcal{F} \ni & f ::= & \texttt{def}\ \texttt{x}(\texttt{x}^*, [\texttt{x}^*])\,\ell \\
\text{Instructions} & \mathcal{I} \ni & i ::= & \texttt{let}\ \texttt{x} = e \mid \texttt{x} = (e\ e^*) \mid \texttt{assert}\ e \mid \texttt{if}\ e\ \ell\ \ell \mid \texttt{return}\ e \mid r = e \\
\text{References} & & r ::= & \texttt{x} \mid r[e] \\
\text{Expressions} & & e ::= & t\ \{[\texttt{x} : e]^*\} \mid [e^*] \mid e : \tau \mid r? \mid e \oplus e \mid \ominus\ e \mid r \mid c \mid v^{\mathsf{p}} \\
\text{Primitives} & \mathbb{V}^{\mathsf{p}} \ni & v^{\mathsf{p}} ::= & \texttt{undefined} \mid \texttt{null} \mid b \mid n \mid i_{\mathsf{big}} \mid s \mid @s \\
\text{Types} & \mathbb{T} \ni & \tau ::= & t \mid [\,] \mid [\tau] \mid \texttt{js} \mid \texttt{prim} \mid \texttt{undefined} \mid \texttt{null} \\
 & & & \mid \texttt{bool} \mid \texttt{numeric} \mid \texttt{num} \mid \texttt{bigint} \mid \texttt{str} \mid \texttt{symbol}
\end{array}
$$

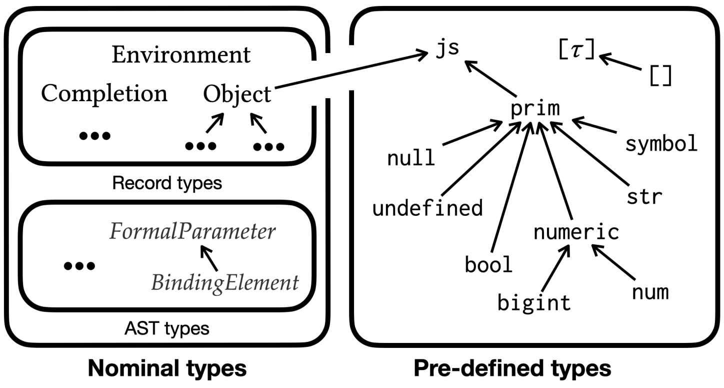


Figure 4.4: A graphical representation of the subtype relation <:

$\text{IR}_{\text{ES}}$ is an untyped intermediate representation for ECMAScript. We modify it as a label-based language to make it suitable for type analysis. A modified $\text{IR}_{\text{ES}}$ program $P = (\mathsf{func}, \mathsf{inst}, \mathsf{next})$ consists of three mappings; `func` : $\mathcal{L} \rightarrow \mathcal{F}$ maps labels to their functions, `inst` : $\mathcal{L} \rightarrow \mathcal{I}$ maps labels to their instructions, and `next` : $\mathcal{L} \rightarrow \mathcal{L}$ maps labels to their next labels, where a label $\ell \in \mathcal{L}$ denotes a program point. A function `def f(x*, [p*])`$\ell \in \mathcal{F}$ consists of its name `f`, normal parameters `x*`, optional parameters `p*`, and a body label $\ell$. For presentation brevity, we assume that no global variables exist in this chapter. An instruction $i$ is a variable declaration, a function call, an assertion, a branch, a return, or a reference update. An invocation of an abstract algorithm in ECMAScript is compiled to a function call instruction with a new temporary variable. We represent loops using branch instructions with cyclic pointing of labels in `next`. A reference $r$ is a variable `x` or a field access $r$`[`$e$`]`. We write $r$`.f` to briefly represent $r$`["f"]`. An expression $e$ is a record, a list, a type check, an existence check, a binary operation, a unary operation, a reference, a constant, or a primitive, which is either `undefined`, `null`, a `Boolean` $b$, a `Number` $n$, a `BigInt` $i_{\text{big}}$, a `String` $s$, or a `Symbol` $@s$.

A type $\tau \in \mathbb{T}$ is either a nominal type $t$, an empty list type `[]`, a parametric list type `[`$\tau$`]`, a JavaScript type `js`, a primitive type `prim`, a numeric type `numeric`, `num`, `bigint`, `str`, or `symbol`. A nominal type $t$ is either 1) an *AST type* with its corresponding syntax-directed algorithms as its fields, or 2) a *record type* with specific fields as described in ECMAScript. For example, Figure 4.3 shows an excerpt from ECMAScript 2020 (ES11) that describes the fields of completion records[1], which we model as follows:

```
Completion = {
    Type     : {c_normal, c_break, c_continue, c_return, c_throw},
    Value    : {js, c_empty},
    Target   : {str, c_empty}
}
```

The subtype relation $<:\subseteq \mathbb{T} \times \mathbb{T}$ between types is depicted in Figure 4.4; a directed edge from $\tau'$ to $\tau$ denotes $\tau' <: \tau$, and the relation is reflexive and transitive. The subtype relation depends on the nominal types defined in ECMAScript. We extract the subtype relation for AST types from the JavaScript syntax. For example, consider the syntax-directed abstract algorithm in Figure 4.2(c). Because the nonterminal *BindingElement* is the unique alternative of the *FormalParameter* production, we automatically extract the subtype relation: *BindingElement* <: *FormalParameter*. Using the subtype relation, the expression $e : \tau$ checks whether the evaluation result of $e$ has type $\tau'$ satisfying $\tau' <: \tau$.

[1]https://262.ecma-international.org/11.0/#table-8

We define a denotational semantics of the modified $\text{IR}_{\text{ES}}$ for instructions $[\![i]\!]_i : \mathbb{S} \to \mathbb{S}$, references $[\![r]\!]_r : \mathbb{S} \to \mathbb{S} \times \mathbb{V}$, and expressions $[\![e]\!]_e : \mathbb{S} \to \mathbb{S} \times \mathbb{V}$ where $\mathbb{S}$ and $\mathbb{V}$ denote states and values, respectively.

**States:** $\mathbb{S}$

We define states as follows:

| | | | |
|---|---|---|---|
| States | $\sigma \in$ | $\mathbb{S}$ | $= \mathcal{L} \times \mathbb{K}^* \times \mathbb{H} \times \mathbb{E}$ |
| Contexts | $\kappa \in$ | $\mathbb{K}$ | $= \mathcal{L} \times \mathbb{E} \times \mathcal{X}$ |
| Heaps | $h \in$ | $\mathbb{H}$ | $= \mathbb{A} \to \mathbb{O}$ |
| Addresses | $a \in$ | $\mathbb{A}$ | |
| Objects | $o \in$ | $\mathbb{O}$ | $= (\mathbb{T}_t \times (\mathbb{V}_{\mathsf{str}} \to \mathbb{V})) \uplus \mathbb{V}^*$ |
| Nominal Types | $t \in$ | $\mathbb{T}_t$ | |
| Environments | $\rho \in$ | $\mathbb{E}$ | $= \mathcal{X} \times \mathbb{V}$ |
| Values | $v \in$ | $\mathbb{V}$ | $= \mathcal{F} \uplus \mathbb{A} \uplus \mathbb{V}_{\mathsf{const}} \uplus \mathbb{V}^{\mathsf{p}}$ |
| Constants | $c \in$ | $\mathbb{V}_{\mathsf{const}}$ | |
| Strings | $s \in$ | $\mathbb{V}_{\mathsf{str}}$ | |

A state $\sigma \in \mathbb{S}$ consists of a label, a context stack, a heap, and an environment. A context $\kappa \in \mathbb{K}$ is a triple of a label, an environment, and a variable. A heap $h \in \mathbb{H}$ is a mapping from addresses to objects. For each address $a \in \mathbb{A}$, an object $o \in \mathbb{O}$ is a record from fields to values with its nominal type or a list of values. An environment $\rho \in \mathbb{E}$ is a mapping from variables to values. A value $v \in \mathbb{V}$ is a function, an address, a constant, or a primitive value.

**Instructions:** $[\![i]\!]_i : \mathbb{S} \to \mathbb{S}$

- Variable Declarations:

$$[\![\texttt{let x} = e]\!]_i(\sigma) = (\mathsf{next}(\ell), \overline{\kappa}, h, \rho[\mathsf{x} \mapsto v])$$

where

$$[\![e]\!]_e(\sigma) = ((\ell, \overline{\kappa}, h, \rho), v)$$

- Function Calls:

$$[\![\mathsf{x} = \texttt{(}e_0\ e_1 \cdots e_n\texttt{)}]\!]_i(\sigma) = (\ell_{\mathsf{f}}, \kappa :: \overline{\kappa}, h, \rho')$$

where

$$\begin{aligned}
&[\![e_0]\!]_e(\sigma) = (\sigma_0, \texttt{def f(}\mathsf{p}_1, \cdots, \mathsf{p}_m\texttt{)}\,\ell_{\mathsf{f}} \wedge \\
&[\![e_1]\!]_e(\sigma_0) = (\sigma_1, v_1) \wedge \cdots \wedge [\![e_n]\!]_e(\sigma_{n-1}) = (\sigma_n, v_n) \wedge \\
&\sigma_n = (\ell, \overline{\kappa}, h, \rho) \wedge k = \min(n, m) \wedge \\
&\rho' = [\mathsf{p}_1 \mapsto v_1, \cdots, \mathsf{p}_k \mapsto v_k] \wedge \kappa = (\mathsf{next}(\ell), \rho, \mathsf{x})
\end{aligned}$$

- Assertions:

$$[\![\texttt{assert } e]\!]_i(\sigma) = \sigma' \quad \text{if } [\![e]\!]_e(\sigma) = (\sigma', \texttt{\#t})$$

- Branches:

$$[\![\texttt{if } e\ \ell_{\mathsf{t}}\ \ell_{\mathsf{f}}]\!]_i(\sigma) = \begin{cases} (\ell_{\mathsf{t}}, \overline{\kappa}, h, \rho) & \text{if } v = \texttt{\#t} \\ (\ell_{\mathsf{f}}, \overline{\kappa}, h, \rho) & \text{if } v = \texttt{\#f} \end{cases}$$

where

$$[\![e]\!]_e(\sigma) = ((\ell_{\mathsf{t}}, \overline{\kappa}, h, \rho), v)$$

- Returns:

$$[\![\texttt{return}\ e]\!]_i(\sigma) = (\ell, \overline{\kappa}, h, \rho[\mathsf{x} \mapsto v])$$

  where

$$[\![e]\!]_e(\sigma) = ((\_, (\ell, \rho, \mathsf{x}) :: \overline{\kappa}, h, \_), v)$$

- Variable Updates:

$$[\![\mathsf{x} = e]\!]_i(\sigma) = (\mathsf{next}(\ell), \overline{\kappa}, h, \rho[\mathsf{x} \mapsto v])$$

  where

$$[\![e]\!]_e(\sigma) = ((\ell, \overline{\kappa}, h, \rho), v)$$

- Field Updates:

$$[\![r\texttt{[}e_0\texttt{]} = e_1]\!]_i(\sigma) = (\mathsf{next}(\ell), \overline{\kappa}, h[a \mapsto o'], \rho)$$

  where

$$\begin{aligned}
&[\![r]\!]_e(\sigma) = (\sigma', a) \wedge [\![e_0]\!]_e(\sigma') = (\sigma_0, v_0) \wedge \\
&[\![e_1]\!]_e(\sigma_0) = ((\ell, \overline{\kappa}, h, \rho), v_1) \wedge o = h(a) \wedge \\
&o' = \begin{cases} o_r & \text{if } o = (t, \mathsf{fs}) \wedge v_0 = s \\ o_l & \text{if } o = [v'_1, \cdots, v'_m] \wedge v_0 = n \end{cases} \wedge \\
&o_r = (t, \mathsf{fs}[s \mapsto v_1]) \wedge o_l = [\cdots, v'_{n-1}, v_1, v'_{n+1}, \cdots]
\end{aligned}$$

**References:** $[\![r]\!]_r : \mathbb{S} \to \mathbb{S} \times \mathbb{V}$

- Variable Lookups:

$$[\![\mathsf{x}]\!]_r(\sigma) = (\sigma, \rho(\mathsf{x}))$$

  where

$$\sigma = (\_, \_, \_, \rho)$$

- Field Lookups:

$$[\![r\texttt{[}e\texttt{]}]\!]_r(\sigma) = (\sigma'', v')$$

  where

$$\begin{aligned}
&[\![r]\!]_e(\sigma) = (\sigma', a) \wedge [\![e]\!]_e(\sigma') = (\sigma'', v) \wedge \\
&\sigma'' = (\ell, \overline{\kappa}, h, \rho) \wedge o = h(a) \wedge \\
&v' = \begin{cases} \mathsf{fs}(s) & \text{if } o = (t, \mathsf{fs}) \wedge v = s \\ v'_n & \text{if } o = [v'_1, \cdots, v'_m] \wedge v = n \\ n & \text{if } o = [v'_1, \cdots, v'_n] \wedge v = \texttt{"length"} \end{cases}
\end{aligned}$$

**Expressions:** $[\![e]\!]_e : \mathbb{S} \to \mathbb{S} \times \mathbb{V}$

- Records:

$$[\![t\ \texttt{\{}\mathsf{x}_1 : e_1, \cdots, \mathsf{x}_n : e_n\texttt{\}}]\!]_e(\sigma) = (\sigma', a)$$

  where

$$\begin{aligned}
&[\![e_1]\!]_e(\sigma) = (\sigma_1, v_1) \wedge \cdots \wedge [\![e_n]\!]_e(\sigma_{n-1}) = (\sigma_n, v_n) \wedge \\
&\sigma_n = (\ell, \overline{\kappa}, h, \rho) \wedge \mathsf{fs} = [\mathsf{x}_1 \mapsto v_1, \cdots, \mathsf{x}_n \mapsto v_n] \\
&a \notin \text{Domain}(h) \wedge \sigma' = (\ell, \overline{\kappa}, h[a \mapsto (t, \mathsf{fs})], \rho)
\end{aligned}$$

- Lists:

$$[\![\texttt{[}e_1, \cdots, e_n\texttt{]}]\!]_e(\sigma) = (\sigma', a)$$

where

$$\begin{aligned}&[\![e_1]\!]_e(\sigma) = (\sigma_1, v_1) \land \cdots \land [\![e_n]\!]_e(\sigma_{n-1}) = (\sigma_n, v_n) \land \\ &\sigma_n = (\ell, \overline{\kappa}, h, \rho) \land a \notin \text{Domain}(h) \land \\ &\sigma' = (\ell, \overline{\kappa}, h[a \mapsto [v_1, \cdots, v_n]], \rho)\end{aligned}$$

- Type Checks:

$$[\![e : \tau]\!]_e(\sigma) = (\sigma', b)$$

where

$$[\![e]\!]_e(\sigma) = (\sigma', v) \land b = \begin{cases} \texttt{\#t} & \text{if } v \text{ is a value of } \tau \\ \texttt{\#f} & \text{otherwise} \end{cases}$$

- Variable Existence Checks:

$$[\![\texttt{x?}]\!]_e(\sigma) = (\sigma, b)$$

where

$$\sigma = (\_, \_, \_, \rho) \land b = \begin{cases} \texttt{\#t} & \text{if } \texttt{x} \in \text{Domain}(\rho) \\ \texttt{\#f} & \text{otherwise} \end{cases}$$

- Field Existence Checks:

$$[\![r\texttt{[}e\texttt{]?}]\!]_e(\sigma) = (\sigma'', b)$$

where

$$\begin{aligned}&[\![r]\!]_e(\sigma) = (\sigma', a) \land [\![e]\!]_e(\sigma') = (\sigma'', v) \land \\ &\sigma'' = (\ell, \overline{\kappa}, h, \rho) \land o = h(a) \land \\ &b = \begin{cases} \texttt{\#t} & \text{if } o = (t, \texttt{fs}) \land v = s \land s \in \text{Domain}(\texttt{fs}) \\ \texttt{\#t} & \text{if } o = [v'_1, \cdots, v'_m] \land v = n \land 1 \leq n \leq m \\ \texttt{\#f} & \text{otherwise} \end{cases}\end{aligned}$$

- Binary Operations:

$$[\![e \oplus e]\!]_e(\sigma) = (\sigma'', v_0 \oplus v_1)$$

where

$$[\![e_0]\!]_e(\sigma) = (\sigma', v_0) \land [\![e_1]\!]_e(\sigma') = (\sigma'', v_1)$$

- Unary Operations:

$$[\![\ominus\ e]\!]_e(\sigma) = (\sigma', \ominus\ v)$$

where

$$[\![e]\!]_e(\sigma) = (\sigma', v)$$

- References:

$$[\![r]\!]_e(\sigma) = [\![r]\!]_r(\sigma)$$

- Constants:

$$[\![c]\!]_e(\sigma) = (\sigma, c)$$

- Primitives:

$$[\![v^{\mathsf{p}}]\!]_e(\sigma) = (\sigma, v^{\mathsf{p}})$$

### 4.2.2 Type Analysis

We design a type analysis for the modified $\text{IR}_{\text{ES}}$ based on the abstract interpretation framework [23, 25] with analysis sensitivity [52]. We first define abstract states $\widehat{\mathbb{S}}$ , and then define an abstract semantics of the modified $\text{IR}_{\text{ES}}$ for instructions $\widehat{[\![i]\!]}_i : (\mathcal{L} \times \mathbb{T}^*) \to \widehat{\mathbb{S}} \to \widehat{\mathbb{S}}$, references $\widehat{[\![r]\!]}_r : \widehat{\mathbb{E}} \to \widehat{\mathbb{T}}$, and expressions $\widehat{[\![e]\!]}_e : \widehat{\mathbb{E}} \to \widehat{\mathbb{T}}$.

**Abstract States: $\widehat{\mathbb{S}}$**

Before defining abstract states, we first extend types as follows:

$$\mathbb{T} \ni \tau ::= \cdots \mid f \mid c \mid \texttt{?} \mid b \mid s \mid \texttt{normal}(\tau) \mid \texttt{abrupt}$$

We add types for functions $f$ and constants $c$, `Boolean` values $b$ and `String` values $s$ to precisely handle the control flows of branches and field accesses, respectively, the absent type `?` to represent the absence of variables, and $\texttt{normal}(\tau)$ for normal completions whose `Value` fields have type $\tau$ and `abrupt` for abrupt completions to enhance the analysis precision.

Using the extended types, we define abstract states with flow-sensitivity and type-sensitivity for arguments:

| | | |
|---|---|---|
| Abstract States | $\widehat{\sigma} \in \widehat{\mathbb{S}}$ | $= \mathbb{M} \times \mathbb{R}$ |
| Result Maps | $m \in \mathbb{M}$ | $= \mathcal{L} \times \mathbb{T}^* \to \widehat{\mathbb{E}}$ |
| Return Point Maps | $r \in \mathbb{R}$ | $= \mathcal{F} \times \mathbb{T}^* \to \mathcal{P}(\mathcal{L} \times \mathbb{T}^* \times \mathcal{X})$ |
| Abstract Environments | $\widehat{\rho} \in \widehat{\mathbb{E}}$ | $= \mathcal{X} \to \widehat{\mathbb{T}}$ |
| Abstract Types | $\widehat{\tau} \in \widehat{\mathbb{T}}$ | $= \mathcal{P}(\mathbb{T})$ |

An abstract state $\widehat{\sigma} \in \widehat{\mathbb{S}}$ is a pair of a result map and a return point map. A result map $m \in \mathbb{M}$ represents an abstract environment for each flow- and type-sensitive view, and a return point map $r \in \mathbb{R}$ represents possible return points of each function with a type-sensitive context; each return point consists of a view for the caller function and a variable that represents the return value. An abstract environment $\widehat{\rho} \in \widehat{\mathbb{E}}$ represents possible types for variables, and $\widehat{\rho}(\texttt{x}) = \{\texttt{?}\}$ when `x` is not defined in $\widehat{\rho}$. An abstract type $\widehat{\tau} \in \widehat{\mathbb{T}}$ is a set of types. We define the join operator $\sqcup$, the meet operator $\sqcap$, and the partial order $\sqsubseteq$ for most of abstract domains in a point-wise manner, and define the operators for types with a normalization function `norm` because of their subtype relations:

$$\begin{aligned}
&\widehat{\tau}_0 \sqcup \widehat{\tau}_1 = \texttt{norm}(\widehat{\tau}_0 \cup \widehat{\tau}_1) \\
&\widehat{\tau}_0 \sqcap \widehat{\tau}_1 = \texttt{norm}(\{\tau_0 \in \widehat{\tau}_0 \mid \{\tau_0\} \sqsubseteq \widehat{\tau}_1\} \cup \{\tau_1 \in \widehat{\tau}_1 \mid \{\tau_1\} \sqsubseteq \widehat{\tau}_0\}) \\
&\widehat{\tau}_0 \sqsubseteq \widehat{\tau}_1 \Leftrightarrow \forall \tau_0 \in \widehat{\tau}_0.\ \exists \tau_1 \in \texttt{norm}(\widehat{\tau}_1).\ \text{s.t. } \tau_0 <: \tau_1
\end{aligned}$$

where $\texttt{norm}(\widehat{\tau}) = \{\tau \in \widehat{\tau} \mid \nexists \tau' \in \widehat{\tau} \setminus \{\tau\}.\ \text{s.t. } \tau <: \tau'\}$. Then, we define the abstract semantics $\widehat{[\![P]\!]}$ of a program $P$ as the least fixpoint of the abstract transfer $\widehat{F} : \widehat{\mathbb{S}} \to \widehat{\mathbb{S}}$:

$$\begin{aligned}
\widehat{[\![P]\!]} &= \lim_{n \to \infty} (\widehat{F})^n (\widehat{\sigma}^\iota) \\
\widehat{F}(\widehat{\sigma}) &= \widehat{\sigma} \sqcup \left( \bigsqcup_{(\ell, \overline{\tau}) \in \text{Domain}(m)} \widehat{[\![\texttt{inst}(\ell)]\!]}_i (\ell, \overline{\tau})(\widehat{\sigma}) \right)
\end{aligned}$$

where $\widehat{\sigma} = (m, \_)$ and $\widehat{\sigma}^\iota$ denotes the initial abstract state. As described in Section 4.1, $\widehat{\sigma}^\iota$ contains the entry points of all syntax-directed algorithms without additional parameters and built-in algorithms with appropriate abstract environments. For a syntax-directed algorithm, we construct its abstract environment containing the variable `this` with its production type and other variables for nonterminals.

For example, the syntax-directed algorithm in Figure 4.2(a) is initialized with the following abstract environment:

$$\begin{aligned}\texttt{this} &\mapsto \{\texttt{AssignmentExpression}\},\\ \texttt{LeftHandSideExpression} &\mapsto \{\texttt{LeftHandSideExpression}\},\\ \texttt{AssignmentExpression} &\mapsto \{\texttt{AssignmentExpression}\}\end{aligned}$$

For built-in algorithms, we assign pre-defined variables `this`, `args`, and `NewTarget` with their corresponding types and parameters with `js` types. For example, the following abstract environment is for the built-in algorithm `Math.round` in Figure 4.2(d):

$$\begin{aligned}\texttt{this} &\mapsto \{\texttt{js}\}, & \texttt{args} &\mapsto \{\texttt{[js]}\},\\ \texttt{NewTarget} &\mapsto \{\texttt{Object}, \texttt{undefined}\}, & \texttt{x} &\mapsto \{\texttt{js}\}\end{aligned}$$

**Instructions:** $\widehat{[\![i]\!]}_i : (\mathcal{L} \times \mathbb{T}^*) \to \widehat{\mathbb{S}} \to \widehat{\mathbb{S}}$

- Variable Declarations:

$$\widehat{[\![\texttt{let x =}\ e]\!]}_i(\ell, \overline{\tau})(\widehat{\sigma}) = (\{(\mathsf{next}(\ell), \overline{\tau}) \mapsto \widehat{\rho}_{\mathsf{x}}\}, \varnothing)$$

  where

$$\begin{aligned}&\widehat{\sigma} = (m, \_) \wedge \widehat{\rho} = m(\ell, \overline{\tau}) \wedge\\ &\widehat{\rho}_{\mathsf{x}} = \widehat{\rho}[\mathsf{x} \mapsto \widehat{[\![e]\!]}_e(\widehat{\rho})]\end{aligned}$$

- Function Calls:

$$\widehat{[\![\texttt{x = (}e\ e_1 \cdots e_n\texttt{)}]\!]}_i(\ell, \overline{\tau})(\widehat{\sigma}) = (m', r')$$

  where

$$\begin{aligned}&\widehat{\sigma} = (m, \_) \wedge \widehat{\rho} = m(\ell, \overline{\tau}) \wedge\\ &\widehat{\tau} = \widehat{[\![e]\!]}_e(\widehat{\rho}) \wedge\\ &\widehat{\tau}_1 = \widehat{[\![e_1]\!]}_e(\widehat{\rho}) \wedge \cdots \wedge \widehat{\tau}_n = \widehat{[\![e_n]\!]}_e(\widehat{\rho}) \wedge\\ &T' = \{\mathsf{up}([\tau_1, \cdots, \tau_n]) \mid \tau_1 \in \widehat{\tau}_1 \wedge \cdots \wedge \tau_n \in \widehat{\tau}_n\} \wedge\\ &f = \texttt{def f(}\mathsf{p}_1, \cdots, \texttt{[}\cdots, \mathsf{p}_{k_f}\texttt{])}\,\ell_f \wedge\\ &\widehat{\rho}_{f,\overline{\tau}'} = [\mathsf{p}_1 \mapsto \{\overline{\tau}'[1]\}, \cdots, \mathsf{p}_{k_f} \mapsto \{\overline{\tau}'[k_f]\}] \wedge\\ &m' = \{(\ell_f, \overline{\tau}') \mapsto \widehat{\rho}_{f,\overline{\tau}'} \mid f \in \widehat{\tau} \wedge \overline{\tau}' \in T'\} \wedge\\ &r' = \{(f, \overline{\tau}') \mapsto \{(\mathsf{next}(\ell), \overline{\tau}, \mathsf{x})\} \mid f \in \widehat{\tau} \wedge \overline{\tau}' \in T'\}\end{aligned}$$

- Returns:

$$\widehat{[\![\texttt{return}\ e]\!]}_i(\ell, \overline{\tau})(\widehat{\sigma}) = (m', \varnothing)$$

  where

$$\begin{aligned}&\widehat{\sigma} = (m, r) \wedge \widehat{\rho} = m(\ell, \overline{\tau}) \wedge\\ &R = r(\mathsf{func}(\ell), \overline{\tau}) \wedge\\ &m' = \{(\ell_r, \overline{\tau}_r) \mapsto \widehat{\rho}_r \mid (\ell_r, \overline{\tau}_r, \mathsf{x}) \in R \wedge\\ &\qquad \widehat{\rho}_r = m(\ell_r, \overline{\tau}_r)[\mathsf{x} \mapsto \widehat{[\![e]\!]}_e(\widehat{\rho})]\}\end{aligned}$$

- Assertions:

$$\widehat{[\![\texttt{assert}\ e]\!]}_i(\ell, \overline{\tau})(\widehat{\sigma}) = (m', \varnothing)$$

  where

$$\begin{aligned}&\widehat{\sigma} = (m, \_) \wedge \widehat{\rho} = m(\ell, \overline{\tau}) \wedge\\ &m' = \{(\mathsf{next}(\ell), \overline{\tau}) \mapsto \texttt{pass}(e, \texttt{\#t})(\widehat{\rho})\}\end{aligned}$$

- <u>Branches</u>:

$$[\![\widehat{\texttt{if}\ e\ \ell_{\texttt{t}}\ \ell_{\texttt{f}}}]\!]_i(\ell, \overline{\tau})(\widehat{\sigma}) = (m', \varnothing)$$

  where

$$\begin{aligned} &\widehat{\sigma} = (m, \_) \land \widehat{\rho} = m(\ell, \overline{\tau}) \land \\ &m' = \left\{ \begin{array}{l} (\ell_{\texttt{t}}, \overline{\tau}) \mapsto \texttt{pass}(e, \texttt{\#t})(\widehat{\rho}), \\ (\ell_{\texttt{f}}, \overline{\tau}) \mapsto \texttt{pass}(e, \texttt{\#f})(\widehat{\rho}) \end{array} \right\} \end{aligned}$$

- <u>Variable Updates</u>:

$$[\![\widehat{\texttt{x} = e}]\!]_i(\ell, \overline{\tau})(\widehat{\sigma}) = (\{(\mathsf{next}(\ell), \overline{\tau}) \mapsto \widehat{\sigma}_{\texttt{x}}\}, \varnothing)$$

  where

$$\begin{aligned} &\widehat{\sigma} = (m, \_) \land \widehat{\rho} = m(\ell, \overline{\tau}) \land \\ &\widehat{\sigma}_{\texttt{x}} = \widehat{\rho}[\texttt{x} \mapsto \widehat{[\![e]\!]}_e(\widehat{\rho})] \end{aligned}$$

- <u>Field Updates</u>:

$$[\![\widehat{r\texttt{[}e_0\texttt{]} = e_1}]\!]_i(\ell, \overline{\tau})(\widehat{\sigma}) = (\{(\mathsf{next}(\ell), \overline{\tau}) \mapsto \widehat{\rho}\}, \varnothing)$$

  where

$$\widehat{\sigma} = (m, \_) \land \widehat{\rho} = m(\ell, \overline{\tau})$$

To avoid the explosion of type-sensitive views, we upcast the argument type before function calls with the following function:

$$\texttt{up}(\tau) = \begin{cases} \texttt{normal}(\texttt{up}(\tau')) & \text{if } \tau = \texttt{normal}(\tau') \\ \texttt{[up}(\tau')\texttt{]} & \text{if } \tau = \texttt{[}\tau'\texttt{]} \\ \texttt{str} & \text{if } \tau = s \\ \texttt{bool} & \text{if } \tau = b \\ \tau & \text{otherwise} \end{cases}$$

and $\dot{\texttt{u}}\texttt{p}$ denotes a point-wise extension of $\texttt{up}$ for type sequences. For branches and assertions, we use the following $\texttt{pass}$ function to prevent infeasible control flows:

$$\texttt{pass}(e, b)(\widehat{\rho}) = \begin{cases} \texttt{refine}(e, b)(\widehat{\rho}) & \text{if } \{\texttt{\#t}\} \sqsubseteq \widehat{[\![e]\!]}_e(\widehat{\rho}) \\ \varnothing & \text{otherwise} \end{cases}$$

where $\texttt{refine}$ is a funcition that performs *condition-based refinement* of the type analysis for the modified $\mathrm{IR}_{\mathrm{ES}}$ to enhance the analysis precision. It prunes out infeasible parts in abstract environments using the conditions of branches and assertions. We formally define the $\texttt{refine}$ function as follows:

$$\begin{aligned} \texttt{refine}(\texttt{!}e, b)(\widehat{\rho}) &= \texttt{refine}(e, \neg b)(\widehat{\rho}) \\ \texttt{refine}(e_0 \texttt{ || } e_1, b)(\widehat{\rho}) &= \begin{cases} \widehat{\rho}_0 \sqcup \widehat{\rho}_1 & \text{if } b \\ \widehat{\rho}_0 \sqcap \widehat{\rho}_1 & \text{if } \neg b \end{cases} \\ \texttt{refine}(e_0 \texttt{ \&\& } e_1, b)(\widehat{\rho}) &= \begin{cases} \widehat{\rho}_0 \sqcap \widehat{\rho}_1 & \text{if } b \\ \widehat{\rho}_0 \sqcup \widehat{\rho}_1 & \text{if } \neg b \end{cases} \\ \texttt{refine}(\texttt{x.Type == } c_{\texttt{normal}}, \texttt{\#t})(\widehat{\rho}) &= \widehat{\rho}[\texttt{x} \mapsto \widehat{\tau}_{\texttt{x}} \cap \texttt{normal}(\mathbb{T})] \\ \texttt{refine}(\texttt{x.Type == } c_{\texttt{normal}}, \texttt{\#f})(\widehat{\rho}) &= \widehat{\rho}[\texttt{x} \mapsto \widehat{\tau}_{\texttt{x}} \cap \{\texttt{abrupt}\}] \\ \texttt{refine}(\texttt{x == } e, \texttt{\#t})(\widehat{\rho}) &= \widehat{\rho}[\texttt{x} \mapsto \widehat{\tau}_{\texttt{x}} \sqcap \widehat{\tau}_e] \\ \texttt{refine}(\texttt{x == } e, \texttt{\#f})(\widehat{\rho}) &= \widehat{\rho}[\texttt{x} \mapsto \widehat{\tau}_{\texttt{x}} \setminus \lfloor \widehat{\tau}_e \rfloor] \\ \texttt{refine}(\texttt{x} : \tau, \texttt{\#t})(\widehat{\rho}) &= \widehat{\rho}[\texttt{x} \mapsto \widehat{\tau}_{\texttt{x}} \sqcap \{\tau\}] \\ \texttt{refine}(\texttt{x} : \tau, \texttt{\#f})(\widehat{\rho}) &= \widehat{\rho}[\texttt{x} \mapsto \widehat{\tau}_{\texttt{x}} \setminus \{\tau' \mid \tau' <: \tau\}] \\ \texttt{refine}(e, b)(\widehat{\rho}) &= \widehat{\rho} \end{aligned}$$

where $\widehat{\rho}_j = \texttt{refine}(e_j, b)(\widehat{\rho})$ for $j = 0, 1$, $\widehat{\tau}_e = \widehat{[\![e]\!]}_e(\widehat{\rho})$, and $\lfloor \widehat{\tau} \rfloor$ returns $\{\tau\}$ if $\widehat{\tau}$ denotes a singleton type $\tau$, or returns $\varnothing$, otherwise.

**References:** $\widehat{[\![r]\!]}_r : \widehat{\mathbb{E}} \to \widehat{\mathbb{T}}$

- Variable Lookups:

$$\widehat{[\![\mathsf{x}]\!]}_r(\widehat{\rho}) = \widehat{\rho}(\mathsf{x})$$

- Field Lookups:

$$\widehat{[\![r\texttt{[}e\texttt{]}]\!]}_r(\widehat{\rho}) = \{\tau[v] \mid \tau \in \widehat{[\![r]\!]}_r(\widehat{\rho}) \wedge v \in \widehat{[\![e]\!]}_e(\widehat{\rho})\}$$

  where $\tau[v]$ denotes the access of field $v$ for a type $\tau$.

**Expressions:** $\widehat{[\![e]\!]}_e : \widehat{\mathbb{E}} \to \widehat{\mathbb{T}}$

- Completion Records:

$$\begin{aligned}&\widehat{[\![\texttt{Completion \{}\cdots, \texttt{Type}: e_0, \texttt{Value}: e_1, \cdots\texttt{\}}]\!]}_e(\widehat{\rho})\\ &= \begin{cases}\{\texttt{normal}(\tau) \mid \tau \in \widehat{[\![e_1]\!]}_e(\widehat{\rho})\} & \text{if } \widehat{[\![e_0]\!]}_e = c_{\texttt{normal}}\\ \{\texttt{abrupt}\} & \text{otherwise}\end{cases}\end{aligned}$$

- Records:

$$\widehat{[\![t\ \texttt{\{}\cdots\texttt{\}}]\!]}_e(\widehat{\rho}) = \{t\}$$

- Lists:

$$\begin{aligned}\widehat{[\![\texttt{[]}]\!]}_e(\widehat{\rho}) &= \texttt{[]}\\ \widehat{[\![\texttt{[}e_1, \cdots, e_n\texttt{]}]\!]}_e(\widehat{\rho}) &= \{\texttt{[}\tau\texttt{]} \mid \tau \in \textstyle\bigsqcup_{1 \le i \le n} \widehat{[\![e_i]\!]}_e(\widehat{\rho})\}\end{aligned}$$

- Type Checks:

$$\widehat{[\![e : \tau]\!]}_e(\widehat{\rho}) = \{\tau' <: \tau \mid \tau' \in \widehat{[\![e]\!]}_e(\widehat{\rho})\}$$

- Existence Checks:

$$\widehat{[\![r\texttt{?}]\!]}_e(\widehat{\rho}) = \{\tau \neq \texttt{?} \mid \tau \in \widehat{[\![e]\!]}_e(\widehat{\rho})\}$$

- Binary Operations:

$$\widehat{[\![e_0 \oplus e_1]\!]}_e(\widehat{\rho}) = \{\tau_0 \widehat{\oplus} \tau_1 \mid \tau_0 \in \widehat{\tau}_0 \wedge \tau_1 \in \widehat{\tau}_1\}$$

  where

$$\widehat{\tau}_0 = \widehat{[\![e_0]\!]}_e(\widehat{\rho}) \wedge \widehat{\tau}_1 = \widehat{[\![e_1]\!]}_e(\widehat{\rho})$$

- Unary Operations:

$$\widehat{[\![\ominus\ e]\!]}_e(\widehat{\rho}) = \{\widehat{\ominus}\tau \mid \tau \in \widehat{[\![e]\!]}_e(\widehat{\rho})\}$$

- References:

$$\widehat{[\![r]\!]}_e(\widehat{\rho}) = \widehat{[\![r]\!]}_r(\widehat{\rho}) \setminus \{\texttt{?}\}$$

- Constants:

$$\widehat{[\![c]\!]}_e(\widehat{\rho}) = c$$

- Primitives:

$$\widehat{[\![v^{\mathsf{p}}]\!]}_e(\widehat{\rho}) = \begin{cases}\texttt{num} & \text{if } v^{\mathsf{p}} = n\\ \texttt{bigint} & \text{if } v^{\mathsf{p}} = i_{\mathsf{big}}\\ \texttt{symbol} & \text{if } v^{\mathsf{p}} = \texttt{@}s\\ v^{\mathsf{p}} & \text{otherwise}\end{cases}$$

Table 4.1: Type-related specification bugs fixed by pull requests for the recent three years from 2018 to 2021

| **Category** | **Bug Kind** | **# Pull Requests** | **# Bug Fixes** |
|---|---|---|---|
| Reference | UnknownVar | 5 | 12 |
| | DuplicatedVar | 2 | 12 |
| Arity | MissingParam | 2 | 4 |
| Assertion | Assertion | 4 | 5 |
| Operand | NoNumber | 1 | 2 |
| | Abrupt | 5 | 6 |
| **Total** | | 19 | 41 |

### 4.2.3 Condition-based Refinement

We present a *condition-based refinement* of the type analysis for the modified $\mathrm{IR_{ES}}$ to enhance the analysis precision. It prunes out infeasible parts in abstract environments using the conditions of branches and assertions. We formally define the `refine` function as follows:

$$
\begin{aligned}
\texttt{refine}(\texttt{!}e, b)(\widehat{\rho}) &= \texttt{refine}(e, \neg b)(\widehat{\rho}) \\
\texttt{refine}(e_0 \texttt{ || } e_1, b)(\widehat{\rho}) &= \begin{cases} \widehat{\rho}_0 \sqcup \widehat{\rho}_1 & \text{if } b \\ \widehat{\rho}_0 \sqcap \widehat{\rho}_1 & \text{if } \neg b \end{cases} \\
\texttt{refine}(e_0 \texttt{ \&\& } e_1, b)(\widehat{\rho}) &= \begin{cases} \widehat{\rho}_0 \sqcap \widehat{\rho}_1 & \text{if } b \\ \widehat{\rho}_0 \sqcup \widehat{\rho}_1 & \text{if } \neg b \end{cases} \\
\texttt{refine}(\mathsf{x.Type} \texttt{ == } c_{\texttt{normal}}, \texttt{\#t})(\widehat{\rho}) &= \widehat{\rho}[\mathsf{x} \mapsto \widehat{\tau}_{\mathsf{x}} \sqcap \texttt{normal}(\mathbb{T})] \\
\texttt{refine}(\mathsf{x.Type} \texttt{ == } c_{\texttt{normal}}, \texttt{\#f})(\widehat{\rho}) &= \widehat{\rho}[\mathsf{x} \mapsto \widehat{\tau}_{\mathsf{x}} \sqcap \{\texttt{abrupt}\}] \\
\texttt{refine}(\mathsf{x} \texttt{ == } e, \texttt{\#t})(\widehat{\rho}) &= \widehat{\rho}[\mathsf{x} \mapsto \widehat{\tau}_{\mathsf{x}} \sqcap \widehat{\tau}_e] \\
\texttt{refine}(\mathsf{x} \texttt{ == } e, \texttt{\#f})(\widehat{\rho}) &= \widehat{\rho}[\mathsf{x} \mapsto \widehat{\tau}_{\mathsf{x}} \setminus \lfloor \widehat{\tau}_e \rfloor] \\
\texttt{refine}(\mathsf{x} : \tau, \texttt{\#t})(\widehat{\rho}) &= \widehat{\rho}[\mathsf{x} \mapsto \widehat{\tau}_{\mathsf{x}} \sqcap \{\tau\}] \\
\texttt{refine}(\mathsf{x} : \tau, \texttt{\#f})(\widehat{\rho}) &= \widehat{\rho}[\mathsf{x} \mapsto \widehat{\tau}_{\mathsf{x}} \setminus \{\tau' \mid \tau' <: \tau\}] \\
\texttt{refine}(e, b)(\widehat{\rho}) &= \widehat{\rho}
\end{aligned}
$$

where $\widehat{\rho}_j = \texttt{refine}(e_j, b)(\widehat{\rho})$ for $j = 0, 1$, $\widehat{\tau}_e = \widehat{[\![e]\!]}_e(\widehat{\rho})$, and $\lfloor \widehat{\tau} \rfloor$ returns $\{\tau\}$ if $\widehat{\tau}$ denotes a singleton type $\tau$, or returns $\varnothing$, otherwise.

## 4.3 Bug Detector

We develop a *bug detector* to statically detect type-related specification bugs in ECMAScript using an augmented abstract transfer function $\widehat{F}_a$ with additional checkers. Before implementing checkers, we manually investigated pull requests for the recent three years from 2018 to 2021 to identify important bugs to detect. As summarized in Table 4.1, we found 19 pull requests that fixed 41 type-related specification bugs and classified the bugs into four categories with six kinds. To detect them automatically, we implement four checkers: a *reference checker*, an *arity checker*, an *assertion checker*, and an *operand checker*.

### 4.3.1 Reference Checker

ECMAScript abstract algorithms dynamically introduce variables in any contexts. A *reference bug* occurs when trying to access variables not yet defined (UnknownVar) or to redefine variables already defined (DuplicatedVar). According to our manual investigation of the pull requests, the reference bug is the most prevalent type-related specification bugs; five pull requests fixed 12 unknown variable bugs and two pull requests fixed 12 duplicated variable declaration bugs. We implement a reference checker by adding additional checks to the abstract semantics of variable lookups and variable declarations as follows:

$$\begin{aligned}
\widehat{[\![\mathtt{x}]\!]}_e(\hat{\rho}) &= \begin{cases} \text{unknown variable } \mathtt{x} & \text{if } \widehat{[\![\mathtt{x}]\!]}_r(\hat{\rho}) = \{\mathtt{?}\} \\ \cdots & \text{otherwise} \end{cases} \\
\widehat{[\![\mathtt{let\ x} = e]\!]}_i(\ell, \overline{\tau})(\hat{\sigma}) &= \begin{cases} \text{already defined variable } \mathtt{x} & \text{if } \hat{\tau} = \{\mathtt{\#t}\} \\ \cdots & \text{otherwise} \end{cases} \\
&\quad \text{where } \hat{\tau} = \widehat{[\![\mathtt{x?}]\!]}_e(\hat{\sigma}(\ell, \overline{\tau}))
\end{aligned}$$

If the abstract semantics of a variable lookup for x is a singleton $\{\mathtt{?}\}$, x is always an unknown variable. For example, consider the syntax-directed algorithm in Figure 4.2(a). Since the GetReferencedName algorithm is removed, the variable `GetReferencedName` does not exist in abstract environments and its lookup returns $\{\mathtt{?}\}$. Thus, the reference checker reports the unknown variable bug for `GetReferencedName`. For duplicated variable declarations, the reference checker utilizes the abstract semantics of the existence check $\widehat{[\![\mathtt{x?}]\!]}_e$ to see whether the variable x of each variable declaration is already defined.

### 4.3.2 Arity Checker

The arity of a function $f = \mathtt{def\ f(p_1, \cdots, p_n, [\cdots, p_m])}\ell$ is an interval $[n, m]$ where $n$ and $m - n$ denote the numbers of normal and optional parameters, respectively. In function invocations, an *arity bug* occurs when the number of arguments does not match with the function arity (MissingParam). In the last three years, two pull requests fixed four missing parameter bugs. The arity checker detects them by adding an additional check to the abstract semantics of the function call instruction:

$$\begin{aligned}
&\widehat{[\![\mathtt{x} = (e\ e_1 \cdots e_k)]\!]}_i(\ell, \overline{\tau})(\hat{\sigma}) = \\
&\quad \begin{cases} \text{missing parameters } \mathtt{p}_{k+1}, \cdots, \mathtt{p}_{n_f} & \text{if } \exists f \in \hat{\tau}.\ \text{s.t. } k < n_f \\ \cdots & \text{otherwise} \end{cases} \\
&\quad \text{where } f = \mathtt{def\ f(p_1, \cdots, p_{n_f}, [\cdots, p_{m_f}])}\ell \ \wedge \\
&\qquad\qquad \hat{\tau} = \widehat{[\![e]\!]}_e(\hat{\sigma}(\ell, \overline{\tau})).
\end{aligned}$$

For each function $f$ in the abstract semantics of the function expression $e$, the arity checker compares the number of arguments with the arity of $f$ to detect missing parameters. For example, consider the syntax-directed algorithm in Figure 4.2(b). The algorithm invocation on line 2 is compiled to the following function call instruction:

```
x = (formals.IteratorBindingInitialization formals)
```

using a temporary variable x. Because it passes only a single argument `formals` even though the function arity is $[3, 3]$, the arity checker reports missing parameter bugs for two additional parameters `iteratorRecord` and `environment`.

### 4.3.3 Assertion Checker

An *assertion failure* (Assertion) is a specification bug that occurs when the condition of an assertion instruction is not true. We found four pull requests that fixed five assertion failures. The assertion checker detects them using an additional check in the abstract semantics of the assertion instruction:

$$\widehat{[\![\texttt{assert}\ e]\!]_i}(\ell, \overline{\tau})(\widehat{\sigma}) = \begin{cases} \text{assertion failure } e & \text{if } \{\texttt{\#t}\} \not\sqsubseteq \widehat{\tau} \\ \cdots & \text{otherwise} \end{cases}$$
$$\text{where } \widehat{\tau} = \widehat{[\![e]\!]_e}(\widehat{\sigma}(\ell, \overline{\tau}))$$

It checks whether the abstract semantics of the condition expression $e$ subsumes $\{\texttt{\#t}\}$. For example, consider the syntax-directed algorithm in Figure 4.2(c). The parameter `environment` of this algorithm has an environment record or `undefined`. Since type sensitivity divides the abstract types of arguments to upcasted single types, there are two different abstract environments whose variable `environment` points to $\{\texttt{Environment}\}$ or $\{\texttt{undefined}\}$. When `environment` is $\{\texttt{Environment}\}$, the abstract semantics of the assertion condition `environment = originalEnv` is $\{\texttt{bool}\}$. Even though we know that the type of `originalEnv` is also $\{\texttt{Environment}\}$, because `Environment` is not a singleton type, we cannot conclude that they are the exactly same environment. Thus, the assertion checker does not report any bug for this case. However, if `environment` is $\{\texttt{undefined}\}$, the abstract semantics of the condition `environment = originalEnv` is $\{\texttt{\#f}\}$ because an environment is never equal to `undefined`. Therefore, the assertion checker reports an assertion failure for the condition `environment = originalEnv`.

### 4.3.4 Operand Checker

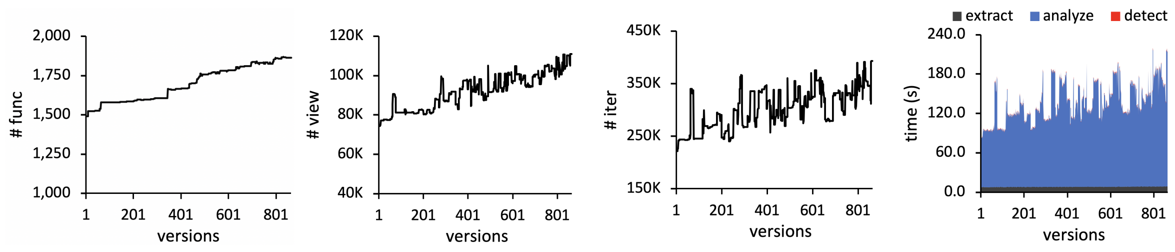
An *ill-typed operand bug* occurs when the type of an operand does not conform to its corresponding parameter type. The operand checker detects such ill-typed operand bugs by additional checks in the abstract semantics of operations:

$$\widehat{[\![e_0 \oplus e_1]\!]_e}(\widehat{\rho}) = \begin{cases} \text{ill-typed operand } e_0 & \text{if } \widehat{[\![e_0]\!]_r}(\widehat{\rho}) \not\sqsubseteq \widehat{\tau}_0 \\ \text{ill-typed operand } e_1 & \text{if } \widehat{[\![e_1]\!]_r}(\widehat{\rho}) \not\sqsubseteq \widehat{\tau}_1 \\ \cdots & \text{otherwise} \end{cases}$$
$$\widehat{[\![\ominus\ e]\!]_e}(\widehat{\rho}) = \begin{cases} \text{ill-typed operand } e & \text{if } \widehat{[\![e]\!]_r}(\widehat{\rho}) \not\sqsubseteq \widehat{\tau} \\ \cdots & \text{otherwise} \end{cases}$$

where $\widehat{\tau}_0$, $\widehat{\tau}_1$, and $\widehat{\tau}$ are expected abstract types of $e_0$, $e_1$, and $e$, respectively. The additional checks report when a given operand does not conform to its expected type. Our manual investigation found two non-numeric operand bugs (NoNumber) in one pull request and six unchecked abrupt completion bugs (Abrupt) in five pull requests.

For an example non-numeric operand bug, consider the built-in algorithm `Math.round` in Figure 4.2(d). The types of `x` and `n` are $\{\texttt{js}\}$ and $\{\texttt{num}\}$, respectively, because ToNumber always returns number values or abrupt completions, and the prefix ? removes the latter case. The built-in algorithm misuses `x` rather than `n` on lines 3 and 4, and because the expected abstract type $\{\texttt{num}, \texttt{bigint}\}$ does not subsume $\{\texttt{js}\}$, the operand checker reports non-numeric operand bugs.

An unchecked abrupt completion bug occurs when an actual value is necessary but it is an abrupt completion. ECMAScript has a special implicit conversion for normal completions when their actual values stored in the `Value` field are necessary. An actual value is necessary in various contexts such as conditions, values of field updates, and operands of operators. For example, if the variable `x` has a normal



(a) Functions (b) Views (c) Iterations (d) Analysis time

Figure 4.5: The statistics of the type analysis using JSTAR for 864 versions of ECMAScript

completion with `42` as its actual value, `x + 1` should be `43` because the normal completion gets implicitly converted into its actual value `42`. We define a unary operator $\downarrow$ to explicitly represent this conversion:

$$\downarrow v = \begin{cases} v & \text{if } v \text{ is not a completion} \\ v.\texttt{Value} & \text{if } v \text{ is normal} \\ \text{unchecked abrupt completion } v & \text{otherwise} \end{cases}$$

The operand checker detects unchecked abrupt completion bugs by assuming that the operator $\downarrow$ is used when the actual value is necessary.

## 4.4 Evaluation

We implemented JSTAR as an open-source tool[2] in Scala by extending JISET, a JavaScript IR-based semantics extraction toolchain [75], with a worklist-based fixpoint algorithm for type analysis. Thus, JSTAR reports type-related specification bugs detected in fully compiled abstract algorithms by JISET. For built-in libraries, JSTAR analyzes the abstract algorithms of the essential built-in objects: `Array`, `Object`, `Function`, `Math`, `Proxy`, and objects for JavaScript primitive types.

We evaluate JSTAR using the following research questions:

- RQ1. **(Performance)** How long does JSTAR take to perform type analysis for JavaScript specifications?
- RQ2. **(Precision)** How many type-related specification bugs detected by JSTAR are true bugs?
- RQ3. **(Effectiveness of Refinement)** Does the condition-based refinement improve the analysis precision with modest performance overhead?
- RQ4: **(Detection of New Bugs)** Does JSTAR detect new specification bugs in the latest version of ECMAScript?

Because the latest ECMAScript (ES12, 2021) is fixed on March 9, 2021, we analyzed all 864 versions in the official ECMAScript repository[3] for the last three years from January 1, 2018 to March 9, 2021. We performed our experiments on five Ubuntu machines equipped with 4.2GHz Quad-Core Intel Core i7 and 32GB of RAM.

[2] `https://github.com/kaist-plrg/jstar`
[3] `https://github.com/tc39/ecma262`

Table 4.2: The analysis precision of JSTAR without refinement (no-refine), with refinement (refine), and their difference (Δ)

| **Checker** | **Bug Kind** | **Precision = (# True Bugs) / (# Detected Bugs)** | | | | | |
|---|---|---|---|---|---|---|---|
| | | no-refine | | refine | | Δ | |
| Reference | UnknownVar | 62/106 ( 58.5%) | 17/60 ( 28.3%) | 63/78 ( 80.8%) | 17/31 ( 54.8%) | +1/-28 (+22.3%) | /-29 (+26.5%) |
| | DuplicatedVar | | 45/46 ( 97.8%) | | 46/47 ( 97.9%) | | +1/+1 ( +0.1%) |
| Arity | MissingParam | 4/ 4 (100.0%) | 4/ 4 (100.0%) | 4/ 4 (100.0%) | 4/ 4 (100.0%) | / ( %) | / ( %) |
| Assertion | Assertion | 4/ 56 ( 7.1%) | 4/56 ( 7.1%) | 4/31 ( 12.9%) | 4/31 ( 12.9%) | /-25 ( +5.8%) | /-25 ( +5.8%) |
| Operand | NoNumber | 22/113 ( 19.5%) | 2/65 ( 3.1%) | 22/44 ( 50.0%) | 2/ 6 ( 33.3%) | /-69 (+30.5%) | /-59 (+30.3%) |
| | Abrupt | | 20/48 ( 41.7%) | | 20/38 ( 52.6%) | | /-10 (+11.0%) |
| **Total** | | 92 / 279 (33.0%) | | 93 / 157 (59.2%) | | +1 / -122 (+26.3%) | |

### 4.4.1 Performance

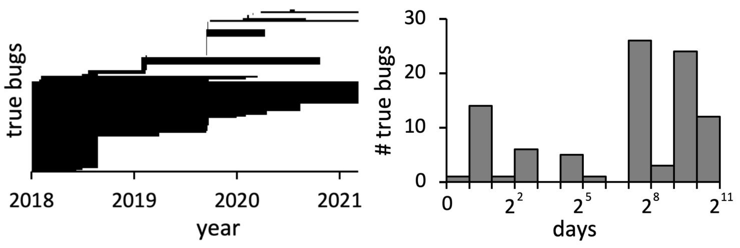
Figure 4.5 shows the statistics of the type analysis using JSTAR for 864 versions of ECMAScript: (a) the number of analyzed functions, (b) the number of flow- and type-sensitive views, (c) the number of worklist iterations, and (d) the analysis time. For each version, JSTAR analyzed 1,696.6 functions on average. Since ECMAScript has gradually evolved, it analyzed 1,491 functions for the first version in 2018 but analyzed 1,864 functions in the latest. JSTAR analyzes functions with flow- and type-sensitive views. On average, each version has 92.0K views, and each function has 54.1 views.

We measured the performance of JSTAR with the worklist iteration number and the analysis time. For each version of ECMAScript, JSTAR took 137.3 seconds with 301.6K worklist iterations on average. The average analysis time is 8.0 seconds for specification extraction (extract), 128.5 seconds for type analysis (analyze), and 0.8 seconds for bug detection (detect). The performance overhead is modest enough for JSTAR to be integrated in the open development process of ECMAScript.

### 4.4.2 Precision

We measured the analysis precision with the ratio of true bugs in the reported bugs by JSTAR. As summarized in the refine column of Table 4.2, the analysis precision is 59.2%; 93 out of 157 detected bugs are true bugs. The reference checker detected the most bugs with 80.8% precision; 17 unknown variables (UnknownVar) and 46 duplicated variable declarations (DuplicatedVar) are true bugs. We found four missing parameters (MissingParam) with 100.0% precision and four assertion failures (Assertion) with 12.9% precision. Finally, the operand checker detected two non-numeric operand bugs (NoNumber) with 33.3% precision and 20 unchecked abrupt completion bugs (Abrupt) with 52.6% precision.

To understand the impact of the detected true bugs, we extended JSTAR to automatically extract when they were created and resolved in the ECMAScript official repository. A bug is *created* when it exists in a specific version but does not exist in its previous version, and a bug is *resolved* vice versa. The *life span* of a bug denotes the number of days between the created date and the resolved date. Figure 4.6 illustrates the life spans of true bugs; Figure 4.6(a) depicts the life spans sorted by creation, and Figure 4.6(b) depicts the histogram of the life spans in a logarithmic scale. Among 93 true bugs, 49 bugs are *inherited*, which means that they were created before 2018. Moreover, 14 bugs still exist in the latest ECMAScript, which are newly detected by JSTAR. We discuss the details of 14 newly found bugs in Section 4.4.4. Even though we assume that 49 inherited bugs were created on January 1, 2018, the average life span is 422.8, and the maximum life span is 1,164. All the bugs with the maximum life span are inherited ones, and they are all newly detected.



(a) Life spans sorted by creation (b) The histogram of life spans

Figure 4.6: Life spans of true bugs

We manually investigated 64 false-positive bugs to understand why JSTAR detected them. Among them, 17 bugs are due to extraction failure of mechanized specifications caused by wrong writing styles. Because ECMAScript is written in HTML, JISET extracts abstract algorithms using the `emu-alg` HTML tag. Unfortunately, several abstract algorithms are defined with the opening tag `<emu-alg>` but without the closing tag `</emu-alg>`, which causes extraction failure of mechanized specifications leading to false-positive bugs. The remaining 47 bugs are due to imprecise analysis. We found that 28 bugs are due to imprecise analysis of the conditions of assertions and branches for specific function calls. For example, consider the following algorithm step for GetValue:

```
4. If IsPropertyReference(V) is true, then
    a. Let baseObj be ! ToObject(V.[[Base]]).
```

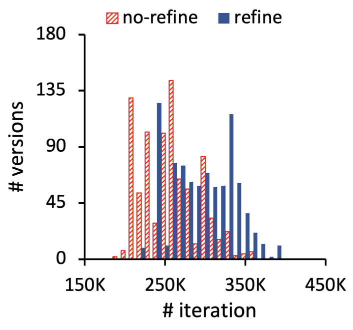
Since IsPropertyReference always returns `false` when the `Base` field of a given reference record is $c_{\texttt{unresolvable}}$, the field access $V$.[[Base]] cannot be $c_{\texttt{unresolvable}}$ on line 4.a. However, because the type analysis does not compute such information, $c_{\texttt{unresolvable}}$ is also passed as the argument of ToObject. We believe that an advanced refinement technique can resolve this problem by pruning out infeasible field types depending on specific contexts.

### 4.4.3 Effectiveness of Refinement

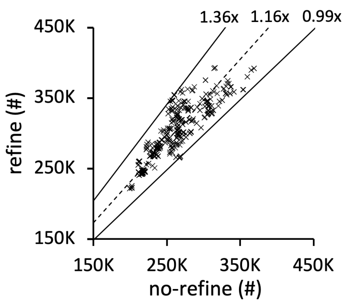
We measured the effectiveness of the condition-based refinement by comparing the performance and the analysis precision of JSTAR without (no-refine) and with refinement (refine).

For performance comparison, Figure 4.7 presents the iterations and analysis time without and with refinement. Figures 4.7(a) and 4.7(c) are histograms of the iterations and analysis time, respectively, and Figures 4.7(b) and 4.7(d) are scatter charts for their ratios. Without refinement, the type analysis took 91.9 seconds with 261.5K iterations on average. After applying refinement, the number of iterations is increased at least 0.99x, at most 1.36x, and 1.16x on average, and the analysis time is increased at least 1.05x, at most 1.99x, and 1.41x on average.

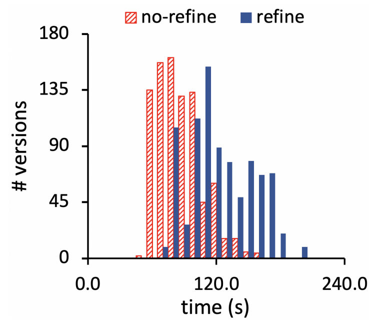
Table 4.2 shows the analysis precision without refinement, with refinement, and their difference. The refinement improved the analysis precision from 33.0% to 59.2% by removing 122 false-positive bugs and detecting one more true bug. Among six bug kinds, the most significant improvement is for non-numeric operand bugs (NoNumber) from 3.1% to 33.3% by removing 59 false-positive bugs. The refinement technique successfully prunes out non-numeric values for numeric types. The refinement significantly increased the analysis precision also for unknown variable bugs (UnknownVar) and assertion failures (Assertion) by removing 29 and 25 false-positive bugs, respectively. Because JSTAR can precisely analyze callees of function invocations without refinement, we found no improvement for missing parameter bugs (MissingParam).



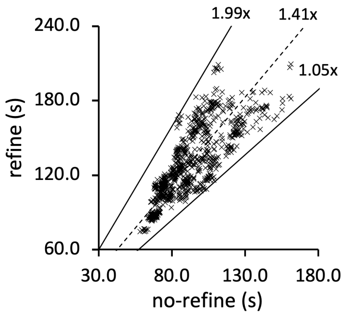
(a) The histogram of iterations



(b) The ratio of iterations



(c) The histogram of time



(d) The ratio of time

Figure 4.7: Comparison of iterations and analysis time without refinement (no-refine) and with refinement (refine)

### 4.4.4 Detection of New Bugs

Among 93 true bugs detected by JSTAR, 14 are newly detected and still exist in the latest version of ECMAScript. Table 4.3 summarizes the bugs, their related JavaScript language features, and their life spans. Except for two bugs in ES12-8, all bugs were introduced in the initial commit of the open development on September 22, 2015. Thus, 12 newly detected bugs last for 1,996 days until March 9, 2021. The two bugs in ES12-8 were created when a contributor introduced the prefixes ? and ! on December 16, 2015, and they last for 1,910 days. We reported the newly detected bugs to TC39, and all of them were confirmed by the committee and will be fixed in ECMAScript 2022 (ES13).

ES12-1 contains three bugs due to duplicated variable declarations in three syntax-directed algorithms for the **`case`** block of the **`switch`** statement: `hasDuplicates` in ContainsDuplicateLabels and `hasUndefinedLabels` in ContainsUndefinedBreakTarget and ContainsUndefinedContinueTarget. A **`case`** block optionally contains **`case`** clauses. In the beginning of three algorithms, `hasDuplicates` or `hasUndefinedLabels` is defined if the clauses exist. However, because the same variable is defined again after the conditional steps, three algorithms for **`case`** blocks with **`case`** clauses always have the duplicated variable declaration bugs for `hasDuplicates` or `hasUndefinedLabels`. Similarly, ES12-2 contains three bugs caused by the same reason in the same abstract algorithms for the **`try`** statement.

The bug in ES12-3 is a reference bug for a duplicated declaration of the variable `index` in the abstract algorithm CreateMappedArgumentsObject. For each function call in JavaScript programs, an `arguments` object is created by CreateMappedArgumentsObject. In the algorithm, the variable `index` is defined to handle the index of a given list of arguments. However, the variable is defined twice in steps 14 and 17 of the algorithm.

Table 4.3: Type-related specification bugs newly detected by JSTAR in the ECMAScript 2021 (ES12)

| Name | Feature | # | Description | Checker | Created | Life Span |
|---|---|---|---|---|---|---|
| ES12-1 | Switch | 3 | Variables `hasDuplicates` and `hasUndefinedLabels` are already defined in algorithms for **`case`** blocks of **`switch`** statements. | Reference | 2015-09-22 | 1,996 days |
| ES12-2 | Try | 3 | Variables `hasDuplicates` and `hasUndefinedLabels` are already defined in algorithms for **`try`** statements. | Reference | 2015-09-22 | 1,996 days |
| ES12-3 | Arguments | 1 | A variable `index` is already defined in CreateMappedArgumentsObject. | Reference | 2015-09-22 | 1,996 days |
| ES12-4 | Array | 2 | A variable `succeeded` is already defined in algorithms for `Array` objects. | Reference | 2015-09-22 | 1,996 days |
| ES12-5 | Async | 1 | A variable `value` is already defined in Evaluation for **`yield`** expressions. | Reference | 2015-09-22 | 1,996 days |
| ES12-6 | Class | 1 | A variable `ClassHeritage` is not defined in Contains for tails of **`class`** declarations. | Reference | 2015-09-22 | 1,996 days |
| ES12-7 | Branch | 1 | A variable `Statement` is not defined in EarlyErrors for **`if`** statement. | Reference | 2015-09-22 | 1,996 days |
| ES12-8 | Arguments | 2 | Abrupt completions are used in DefineOwnProperty and GetOwnProperty for `arguments` objects without any checks. | Operand | 2015-12-16 | 1,910 days |

ES12-4 contains two reference bugs for the already defined variable `succeeded` in DefineOwnProperty of `Array` objects and ArraySetLength. The `Array` objects are not ordinary objects and have special algorithms for specific behaviors. Two such algorithms are wrapper algorithms of OrdinaryDefineOwnProperty, which updates object properties. While they define the variable `succeeded` to representing the result of OrdinaryDefineOwnProperty, the variable is defined twice in a specific condition.

The bug in ES12-5 is a reference bug for the already defined variable `value` in Evaluation of the **`yield`** $*$ expression. In the evaluation of **`yield`** $*$ `e`, the variable `value` is defined twice to represent 1) the evaluation result of the given expression `e` in step 3, and 2) the iterator value in step 7.c.viii.1.

The bug in ES12-6 is a reference bug for the unknown variable `ClassHeritage` in Contains for the tails of **`class`** declarations. A tail of a **`class`** declaration consists of an optional class extension with the **`extends`** keyword and a class body. When the optional class extension does not exist, the variable `ClassHeritage` is not defined, but the Contains algorithm tries to access it without any check of its existence.

The bug in ES12-7 is a reference bug for the unknown variable `Statement` in EarlyErrors for the **`if`** statement. In syntax-directed algorithms, when a production produces multiple sub-ASTs, it uses ordinal numbers as prefixes of variables. Because the **`if`** statement contains two sub-ASTs produced by the *Statement* production, the ordinal number prefixes are necessary for the variable `Statement`. However, the EarlyErrors algorithm for the **`if`** statement uses the variable without any ordinal number prefixes.

ES12-8 contains two operand type bugs related to abrupt completions in DefineOwnProperty and GetOwnProperty for `arguments` objects. The two algorithms define or get own properties of `arguments` objects. They use the Get algorithm, which returns JavaScript values stored in object properties or abrupt completions. Thus, they should check whether the results of Get are abstract completions or not before using them, but they use the results without any checking of abrupt completion.

# Chapter 5. JSAVER: Derivation of Static Analyzers

In this thesis, we first introduce JISET (Chapter 2) to automatically extract a JavaScript mechanized specification from ECMAScript. Then, we present two different ways to check the validity of ECMAScript by 1) checking the conformance with JavaScript engines via $N+1$-version differential testing with JEST (Chapter 3) and 2) performing type analysis for ECMAScript (Chapter 4). Finally, This chapter introduces the last tool JSAVER, a JavaScript Static Analyzer via ECMAScript Representations. It is the first tool that automatically derives JavaScript static analyzers from any versions of ECMAScript. As explained in Chapter 1, the main idea of this work is to shift the paradigm from *compiler*-based approaches to *interpreter*-based approaches to utilize "the interpreter-based nature" of JavaScript. Our contributions to this work are as follows:

- We propose a novel *meta-level static analysis* technique. It indirectly analyzes a *defined*-language program by analyzing its *definitional interpreter* using a static analyzer of the *defining*-language with the program as the input.
- We present JSAVER, the first tool that automatically derives JavaScript static analyzers from any versions of ECMAScript by 1) extracting a definitional interpreter from ECMAScript and 2) performing a meta-level static analysis with the extracted interpreter.
- We derive a static analyzer $\text{JSA}_{\text{ES12}}$ from the latest ECMAScript, ES12, to evaluate JSAVER. The derived analyzer $\text{JSA}_{\text{ES12}}$ soundly analyzes all applicable 18,556 official conformance tests with 99.0% of precision in 1.59 seconds on average. Moreover, we demonstrate the configurability and adaptability of JSAVER with several case studies.

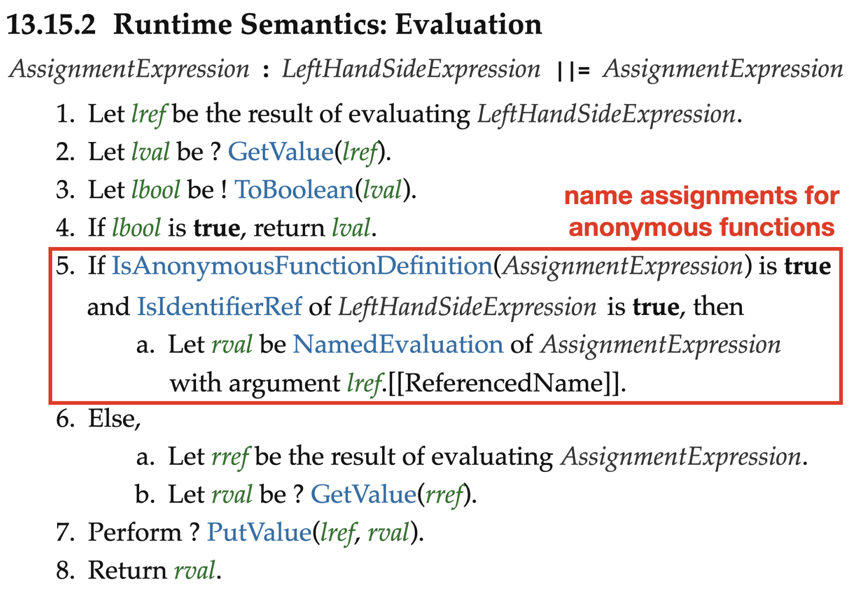
## 5.1 Background

In this section, we briefly remind ECMAScript and how JISET extracts a mechanized specification as a JavaScript definitional interpreter from ECMAScript. Since we perform meta-level static analysis for JavaScript using extracted definitional interpreters, it is essential to understand how ECMAScript describes the JavaScript semantics and how JISET extracts a definitional interpreter from it.

As a running example, we use the "logical OR assignment" introduced in ES12. Figure 5.1(a) shows its semantics described as an algorithm in English, Figure 5.1(b) shows an $\text{IR}_{\text{ES}}$ function extracted from the algorithm, and Figure 5.2 presents an example JavaScript program using a logical OR assignment.

### 5.1.1 JavaScript Semantics in ECMAScript

ECMAScript is the official specification of JavaScript, which describes its syntax in a variant of the extended Backus–Naur form (EBNF) and its semantics as algorithms in English. For example, consider the example code in Figure 5.2. It uses several language features not supported in ES5.1: the `let` statement, the arrow function, and the logical OR assignment. Among them, the logical OR assignment is newly introduced in ES12. Its syntax is defined by the eighth of nine alternatives of the syntactic production of *AssignmentExpression*, and their semantics is defined by the algorithm in Figure 5.1(a). It first evaluates *LeftHandSideExpression* to get a reference *lref* and its value *lval* in steps 1 and 2, respectively.


name assignments for anonymous functions

**13.15.2 Runtime Semantics: Evaluation**

*AssignmentExpression* **:** *LeftHandSideExpression* `||=` *AssignmentExpression*

```
1. Let lref be the result of evaluating LeftHandSideExpression.
2. Let lval be ? GetValue(lref).
3. Let lbool be ! ToBoolean(lval).
4. If lbool is true, return lval.
5. If IsAnonymousFunctionDefinition(AssignmentExpression) is true
   and IsIdentifierRef of LeftHandSideExpression is true, then
      a. Let rval be NamedEvaluation of AssignmentExpression
         with argument lref.[[ReferencedName]].
6. Else,
      a. Let rref be the result of evaluating AssignmentExpression.
      b. Let rval be ? GetValue(rref).
7. Perform ? PutValue(lref, rval).
8. Return rval.
```

(a) Evaluation algorithm for the logical OR assignment

```
syntax def AssignmentExpression[8].Evaluation(
  this, LeftHandSideExpression, AssignmentExpression
) { /* entry */
  let lref = (LeftHandSideExpression.Evaluation)
  let lval = [? (GetValue lref)]
  let lbool = [! (ToBoolean lval)] /* #1 */
  if (= lbool true) { /* #2 */ return lval } else {} /* #3 */
  if (&& (IsAnonymousFunctionDefinition AssignmentExpression)
     (LeftHandSideExpression.IsIdentifierRef)) { /* #4 */
    let rval = (AssignmentExpression.NamedEvaluation lref.ReferencedName)
  } else { /* #5 */
    let rref = (AssignmentExpression.Evaluation)
    let rval = [? (GetValue rref)]
  } /* #6 */
  [? (PutValue lref rval)]
  return rval
} /* exit */
```

(b) Extracted $\text{IR}_\text{ES}$ function for the logical OR assignment

Figure 5.1: Evaluation algorithm for the eighth alternative of *AssignmentExpression* in ES12 and its extracted $\text{IR}_\text{ES}$ function

```
let f = /* a random integer from 0 to 99 */;
f ||= x => x;   // f: {name: "f", ...} or [1, 99]
let y = f.name; // x: "f" or undefined
```

Figure 5.2: JavaScript code using the logical OR assignment

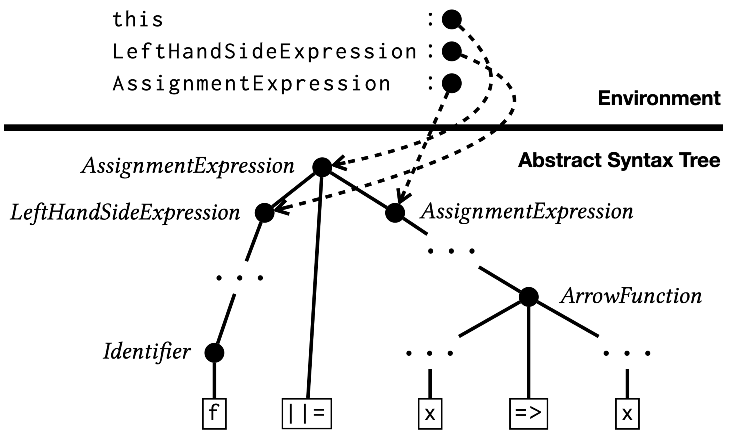


Figure 5.3: Result of `f ||= x => x` in a definitional interpreter

Then, it checks whether its boolean value *lbool* is `true` for short-circuiting in steps 3-4. In step 5, if the right-hand-side *AssignmentExpression* is an anonymous function and the *LeftHandSideExpression* is an identifier reference, it defines the name of the function as the identifier name in step 5-a. In step 6, otherwise, the algorithm evaluates the right-hand-side expression to the value *rval*. It then puts *rval* to the reference *lref* and returns *rval*. While the operator seems to be the same as combining the logical OR operator (`||`) with the assignment operator (`=`), they have different semantics when defining names of anonymous functions. Consider the example code. It first defines a variable `f` with a random integer from `0` to `99`. Then, it uses a logical OR assignment to update `f` with an arrow function whose name becomes `"f"` only if `f`'s value is `0` because `0` represents **`false`**, but the other integers represent **`true`**. Finally, it defines a variable `y` with `f.name`, whose value is `"f"` if `f`'s value is the arrow function, but `undefined`, otherwise. If the statement on line 2 is `f = f || (x => x);`, the value of `y` is `undefined` or `""` instead of `"f"`. Thus, to construct a sound static analyzer, one should consider such detailed semantics by referring to all the algorithms in ECMAScript.

ECMAScript uses two kinds of algorithms: *syntax-directed algorithms* and *normal algorithms*. A syntax-directed algorithm consists of 1) its corresponding alternative of a syntactic production, 2) its name, 3) parameters, and 4) body steps. For example, the algorithm in Figure 5.1(a) is a syntax-directed algorithm consisting of the eighth alternative of *AssignmentExpression*, Evaluation as its name, no parameters, and the body consisting of eight steps. Such syntax-directed algorithms are invoked with a verb "evaluate" for Evaluation algorithms or a preposition "of" for the other named algorithms. For example, step 1 of the algorithm invokes the Evaluation algorithm for *LeftHandSideExpression* without any arguments. On the other hand, step 5-a invokes the NamedEvaluation algorithm of *AssignmentExpression* with the argument *lref*.[[ReferencedName]]. Unlike syntax-directed algorithms, a normal algorithm is defined with only its name, parameters, and body steps. Their invocations are like function calls with parentheses: GetValue(*lref*) in step 2 and ToBoolean(*lval*) in step 3. Finally, each algorithm always returns a *completion record* to handle different kinds of JavaScript control flows. The prefix "?" checks whether a completion record is *abrupt* and returns immediately if so. Otherwise, it converts the completion record to its containing value. On the other hand, the prefix "!" converts a completion record to its containing value without checking for abrupt completion.

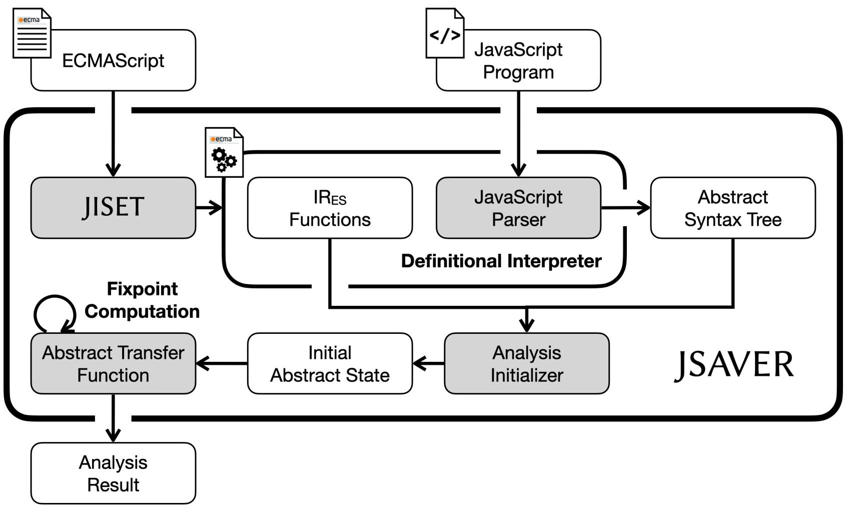


Figure 5.4: Overall structure of JSAVER

### 5.1.2 JavaScript Definitional Interpreter

Several researchers and developers have presented JavaScript *definitional interpreters* [3, 6, 16, 19, 39, 75] instead of the compiler-based approaches [31, 36, 45, 56, 70]. A definitional interpreter is written in a *defining*-language to describe the language semantics of a *defined*-language. Among them, we utilize JISET [75] to automatically extract a definitional interpreter from a given version of ECMAScript. The tool JISET 1) generates a parser for syntax and 2) transforms algorithms to corresponding $\mathrm{IR}_{\mathrm{ES}}$ functions for semantics. For example, when JISET takes ES12 as an input, it generates a parser that supports logical OR assignments according to the syntactic production of *AssignmentExpression*. It then transforms the syntax-directed algorithm in Figure 5.1(a) into the $\mathrm{IR}_{\mathrm{ES}}$ function in Figure 5.1(b). The defining-language of a definitional interpreter often treats ASTs of the defined-language as values. The defining-language $\mathrm{IR}_{\mathrm{ES}}$ also treats ASTs of the defined-language JavaScript as its values. For example, the parser generated from ES12 parses the second statement in Figure 5.2 and produces an AST shown at the bottom of Figure 5.3. Then, the extracted $\mathrm{IR}_{\mathrm{ES}}$ function in Figure 5.1(b) takes the AST and its left and right subtrees as its arguments and defines three local variables as shown at the top of Figure 5.3.

## 5.2 Overview

In this section, we explain the overall structure of JSAVER as depicted in Figure 5.4. It performs a *meta-level static analysis* with JavaScript as its *defined*-language and $\mathrm{IR}_{\mathrm{ES}}$ as its *defining*-language. Thus, JSAVER indirectly analyzes a JavaScript program by analyzing $\mathrm{IR}_{\mathrm{ES}}$ functions with the AST of the program as an argument. For a more detailed explanation, we describe how it performs a meta-level static analysis for the code in Figure 5.2 with ES12.

JSAVER first utilizes JISET to extract a definitional interpreter from ES12. As explained in Section 5.1, it generates a JavaScript parser supporting new language features, including the logical OR assignment, and extracts $\mathrm{IR}_{\mathrm{ES}}$ functions, including the function in Figure 5.1(b), by compiling algorithms. The generated parser parses the example code to produce an AST, which contains the AST shown at the bottom of Figure 5.3 as a subtree. Then, Analysis Initializer constructs an initial abstract state with the extracted $\mathrm{IR}_{\mathrm{ES}}$ functions and the produced AST. Finally, JSAVER computes the fixpoint of Abstract Transfer Function as the analysis result of the example code with the initial abstract state.

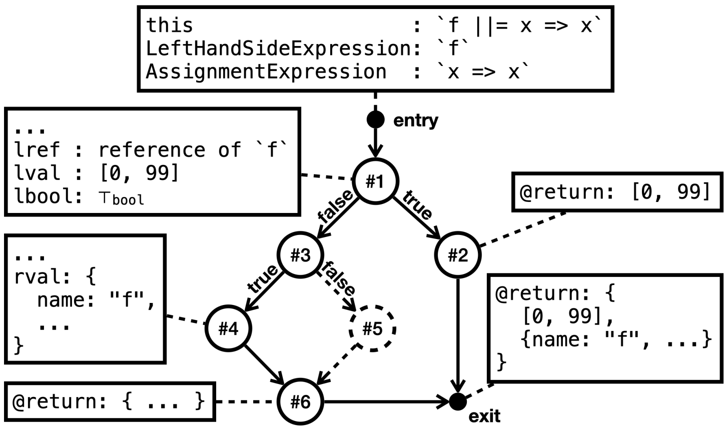
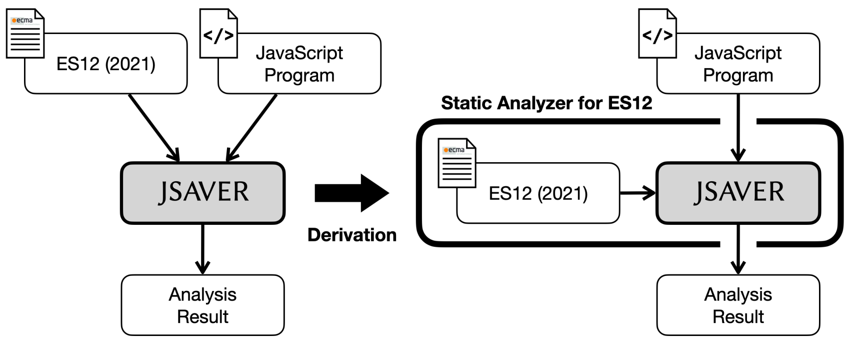


Figure 5.5: Control flow graph of the $\text{IR}_{\text{ES}}$ function in Figure 5.1(b) with its flow-sensitive analysis results

Now, let us explain what happens during the analysis of the $\text{IR}_{\text{ES}}$ function in Figure 5.1(b). We support *view-based analysis sensitivities* [52, 73] and utilize a worklist algorithm to perform view-wise updates of analysis results. In this example, we perform flow-sensitive analysis by splitting views based on program points annotated in comments of the $\text{IR}_{\text{ES}}$ function: `entry`, `exit`, and from `#1` to `#6`.

Figure 5.5 shows a control flow graph of the $\text{IR}_{\text{ES}}$ function with its flow-sensitive analysis results. In the graph, each node and arrow denotes a program point and a control flow, respectively. If nodes or arrows are dotted, they are unreachable. In addition, we use the interval domain [24] for integers in this example. At the entry point, three parameters point to three ASTs, respectively, as shown at the top of Figure 5.3. At point `#1`, three new local variables are defined: `lref`, `lval`, and `lbool`. Since the variable `LeftHandSideExpression` points to the AST of the JavaScript variable `f`, `lref` points to its reference and `lval` points to the interval `[0, 99]`. Moreover, `lbool` points to the top boolean value $\top_{\texttt{bool}}$ because `lval` contains `0` representing **`false`** and `[1, 99]` representing **`true`**. Therefore, both points `#2` and `#3` are reachable. At point `#2`, it returns `lval`; thus, the return value `@return` at the exit point becomes `[0, 99]`. At point `#3`, the condition is always true in this code; thus, only point `#4` is reachable, and it assigns a new variable `rval` with a JavaScript function object whose `name` property is a string `"f"`. At point `#6`, it updates the reference of the JavaScript variable `f` with `rval` and returns it. Therefore, the return value `@return` at the exit point is merged with the JavaScript function object stored in `rval`. Finally, the $\text{IR}_{\text{ES}}$ function returns the abstract value representing both `[0, 99]` and the JavaScript function object.

Finally, we can automatically derive a JavaScript static analyzer for a specific version of ECMAScript using JSAVER. For example, if we want to derive a JavaScript static analyzer for ES12, it is sufficient to fix the first argument of JSAVER as ES12 and passes a given JavaScript program as the second argument of the tool as follows:



In the remainder of this chapter, we formally define the meta-level static analysis for JavaScript with abstract domains and analysis sensitivities (Section 5.3). Then, we explain how to implement JSAVER with several optimization and analysis techniques (Section 5.4). Finally, we evaluate JSAVER (Section 5.5).

## 5.3 Meta-Level Static Analysis

In this section, we formalize a *meta-level static analysis* for JavaScript as a *defined*-language with $\text{IR}_{\text{ES}}$ as a *defining*-language. We first define a simplified $\text{IR}_{\text{ES}}$ for brevity and a JavaScript *definitional interpreter* as an $\text{IR}_{\text{ES}}$ program. Then, we define a meta-level static analysis for JavaScript with the abstract semantics of $\text{IR}_{\text{ES}}$ in the abstract interpretation framework [23, 25]. In addition, we explain how to indirectly express abstract domains and analysis sensitivities for JavaScript.

### 5.3.1 $\text{IR}_{\text{ES}}$: An IR for ECMAScript

We first define a simplified $\text{IR}_{\text{ES}}$ by excluding several minor language features for brevity. It is an intermediate representation for ECMAScript used as a defining-language for a JavaScript definitional interpreters extracted by JISET. Then, we defined its its collecting and restricted semantics.

| | | |
|---|---|---|
| Programs | $\mathfrak{P} \ni P$ | $::= f^*$ |
| Functions | $\mathcal{F} \ni f$ | $::= \texttt{syntax}^? \ \texttt{def} \ \texttt{x(x}^*\texttt{)} \ \texttt{\{}[\ell : i]^*\texttt{\}}$ |
| Variables | $\mathcal{X} \ni \texttt{x}$ | |
| Labels | $\mathcal{L} \ni \ell$ | |
| Instructions | $\mathcal{I} \ni i$ | $::= r := e \mid \texttt{x} := \texttt{\{\}} \mid \texttt{x} := e\texttt{(}e^*\texttt{)} \mid \texttt{if} \ e \ \ell \ \ell \mid \texttt{return} \ e$ |
| Expressions | $\mathcal{E} \ni e$ | $::= v^{\mathsf{p}} \mid \texttt{op(}e^*\texttt{)} \mid r$ |
| References | $\mathcal{R} \ni r$ | $::= \texttt{x} \mid e\texttt{[}e\texttt{]} \mid e\texttt{[}e\texttt{]}_{\mathsf{js}}$ |

**Syntax and Notations** An $\text{IR}_{\text{ES}}$ program $P$ is a sequence of functions. A function $f$ is defined with its name, parameters, and body instructions with labels. If it is defined with the prefix `syntax`, it is a syntax-directed function, otherwise, a normal function. An instruction $i$ is a reference update, an object allocation, a function call, a branch, or a return instruction. An expression $e$ is a primitive value, a primitive operation, or a reference expression. A reference is a variable, an internal field access, or an external field access. For a given program $P$, three helper functions $\mathsf{func} : \mathcal{L} \to \mathcal{F}$, $\mathsf{inst} : \mathcal{L} \to \mathcal{I}$, and $\mathsf{next} : \mathcal{L} \to \mathcal{L}$ return the function, instruction, and next label, respectively, of a given label.

| | | |
|---|---|---|
| States | $\sigma \in \mathbb{S}$ | $= \mathcal{L} \times \mathbb{E} \times \mathbb{K}^* \times \mathbb{H}$ |
| Environments | $\rho \in \mathbb{E}$ | $= \mathcal{X} \xrightarrow{\text{fin}} \mathbb{V}$ |
| Calling Contexts | $\kappa \in \mathbb{K}$ | $= \mathcal{L} \times \mathbb{E}$ |
| Heaps | $h \in \mathbb{H}$ | $= \mathbb{A} \xrightarrow{\text{fin}} \mathcal{L} \times \mathbb{M} \times \mathbb{M}_{\mathsf{js}}$ |
| Internal Field Maps | $m \in \mathbb{M}$ | $= \mathbb{V}_{\mathsf{str}} \xrightarrow{\text{fin}} \mathbb{V}$ |
| External Field Maps | $m_{\mathsf{js}} \in \mathbb{M}_{\mathsf{js}}$ | $= \mathbb{V}_{\mathsf{str}} \xrightarrow{\text{fin}} \mathbb{V}$ |
| Values | $v \in \mathbb{V}$ | $= \mathbb{A} \uplus \mathbb{V}^{\mathsf{p}} \uplus \Omega \uplus \mathcal{F}$ |
| Primitive Values | $v^{\mathsf{p}} \in \mathbb{V}^{\mathsf{p}}$ | $= \mathbb{V}_{\mathsf{bool}} \uplus \mathbb{V}_{\mathsf{int}} \uplus \mathbb{V}_{\mathsf{str}} \uplus \cdots$ |
| JS ASTs | $\omega \in \Omega$ | |

*CoalesceExpression* **:**

  *CoalesceExpressionHead* **??** *BitwiseORExpression*

*CoalesceExpressionHead* **:**

  *CoalesceExpression*

  *BitwiseORExpression*

Figure 5.6: A JavaScript syntactic production for coalesce expressions

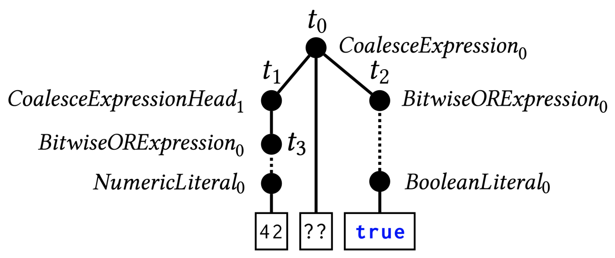
**Concrete States** An $\text{IR}_{\text{ES}}$ state $\sigma \in \mathbb{S}$ consists of a label, an environment, a stack of calling contexts, and a heap. An environment $\rho \in \mathbb{E}$ is a finite mapping from variables to values. A calling context $\kappa \in \mathbb{K}$ consists of a label and an environment of the caller. A heap $h \in \mathbb{H}$ is a finite mapping from addresses to labels for allocation sites and two finite mappings from strings to values. The former mapping represents internal fields accessible by $e$`[`$e$`]`, and the latter represents external fields accessible by $e$`[`$e$`]`$_{\texttt{js}}$. A value $v \in \mathbb{V}$ is an address, a primitive value (e.g., a boolean $b$, an integer $i$, and a string $s$), a JavaScript AST $\omega \in \Omega$, or a function $f \in \mathcal{F}$.

Since $\text{IR}_{\text{ES}}$ treats JavaScript ASTs as its values, we define them with tree nodes $\Phi$ as follows:

$$\begin{aligned}\Omega \ni \omega &::= \tau_k\langle \phi^* \rangle \\ \Phi \ni \phi &::= s \mid \omega\end{aligned}$$

A JavaScript AST $\tau_k\langle \phi_1, \cdots, \phi_n \rangle$ denotes $k$-th alternative in the syntactic production of nonterminal symbol $\tau$ with multiple tree nodes $\phi_1, \cdots, \phi_n$. A tree node is a string for a terminal symbol or another tree for a nonterminal symbol. We define several notations to easily deal with JavaScript ASTs. The notation $\tau_k$`.eval` denotes an Evaluation function of $k$-th alternative in the production $\tau$. Similarly, the notation $\omega$`.eval` denotes the Evaluation function of the AST $\omega$, and it is same with $\tau_k$`.eval` when $\omega = \tau_k\langle \cdots \rangle$. The Evaluation of each AST takes the AST itself and its tree nodes that are nonterminals as arguments. The notation $\mathsf{subs}(\omega)$ denotes tree nodes that are subtrees of $\omega$. For example, Figure 5.6 shows a syntactic production for coalesce expressions. Consider the JavaScript coalesce expression: `42 ?? true`. Then, the following AST is produced as its parsing result:

$$\begin{aligned}\omega_0 &= \mathit{CoalesceExpression}_0\langle \omega_1, \texttt{"??"}, \omega_2 \rangle \\ \omega_1 &= \mathit{CoalesceExpressionHead}_1\langle \omega_3 \rangle \\ \omega_2 &= \mathit{BitwiseORExpression}_0\langle \cdots \rangle \\ \omega_3 &= \mathit{BitwiseORExpression}_0\langle \cdots \rangle\end{aligned}$$


$t_0$
$\mathit{CoalesceExpression}_0$
$t_1$
$t_2$
$\mathit{CoalesceExpressionHead}_1$
$\mathit{BitwiseORExpression}_0$
$\mathit{BitwiseORExpression}_0$
$t_3$
$\mathit{NumericLiteral}_0$
$\mathit{BooleanLiteral}_0$
42
??
true

Its Evaluation function $\mathit{CoalesceExpression}_0$`.eval` takes three subtrees as arguments annotated by $\omega_0$, $\omega_1$, and $\omega_2$ in the figure. The ASTs $\omega_0$, $\omega_1$, $\omega_2$, and $\omega_3$ are subtrees of $\omega_0$ (i.e., $\omega_0 \lhd \omega_0, \cdots, \omega_3 \lhd \omega_0$), and $\mathsf{subs}(\omega_0) = [\omega_1, \omega_2] \wedge \mathsf{subs}(\omega_1) = [\omega_3]$.

**Collecting Semantics** We define denotational semantics of instructions $[\![i]\!] : \mathbb{S} \to \mathbb{S}$ and expressions $[\![e]\!] : \mathbb{S} \to \mathbb{V}$ in Section 5.3.1 and Section 5.3.1, respectively. Then, the *collecting semantics* $[\![P]\!]$ of an $\text{IR}_{\text{ES}}$ program $P$ is a set of reachable states $\mathcal{P}(\mathbb{S})$ from the initial states $\mathbb{S}^\iota \subseteq \mathbb{S}$. We can compute it using a fixpoint algorithm:

$$[\![P]\!] = \lim_{n\to\infty} F^n(\mathbb{S}^\iota)$$

with a *transfer function* $F : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$:

$$F(S) = S \cup \{\sigma' \in \mathbb{S} \mid \sigma \in S \wedge \sigma \rightsquigarrow_P \sigma'\}$$

where $\sigma \rightsquigarrow_P \sigma'$ denotes the one-step transition of a state $\sigma$ to another state $\sigma'$ in the program $P$:

$$\sigma \rightsquigarrow_P \sigma' \iff \sigma = (\ell, \_, \_, \_) \wedge [\![\mathsf{inst}(\ell)]\!](\sigma) = \sigma'$$

**Restricted Semantics** Moreover, the *restricted semantics* $[\![P]\!]^{\mathbf{R}} : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ is a set of reachable states from the initial states restricted by a given set of states:

$$[\![P]\!]^{\mathbf{R}}(S) = \lim_{n\to\infty} F^n(\mathbb{S}^\iota \cap S)$$

**Instructions:** $\boxed{[\![i]\!] : \mathbb{S} \to \mathbb{S}}$

- Variable Assignments:

$$[\![\mathsf{x} := e]\!](\sigma) = (\mathsf{next}(\ell), \rho[\mathsf{x} \mapsto v], \overline{\kappa}, h)$$

where $\sigma = (\ell, \rho, \overline{\kappa}, h)$ and $[\![e]\!] = v$

- Internal Field Assignments:

$$[\![e_0\,\texttt{[}e_1\texttt{]} := e_2]\!](\sigma) = (\mathsf{next}(\ell), \rho, \overline{\kappa}, h[a \mapsto (\ell', m', m_{\mathsf{js}})])$$

where

$$\begin{aligned} \sigma &= (\ell, \rho, \overline{\kappa}, h) \\ (a, s, v) &= ([\![e_0]\!](\sigma), [\![e_1]\!](\sigma), [\![e_2]\!](\sigma)) \\ (\ell', m, m_{\mathsf{js}}) &= h(a) \\ m' &= m[s \mapsto v] \end{aligned}$$

- External Field Assignments:

$$[\![e_0\,\texttt{[}e_1\texttt{]}_{\mathsf{js}} := e_2]\!](\sigma) = (\mathsf{next}(\ell), \rho, \overline{\kappa}, h[a \mapsto (\ell', m, m'_{\mathsf{js}})])$$

where

$$\begin{aligned} \sigma &= (\ell, \rho, \overline{\kappa}, h) \\ (a, s, v) &= ([\![e_0]\!](\sigma), [\![e_1]\!](\sigma), [\![e_2]\!](\sigma)) \\ (\ell', m, m_{\mathsf{js}}) &= h(a) \\ m'_{\mathsf{js}} &= m_{\mathsf{js}}[s \mapsto v] \end{aligned}$$

- Field Mapping Allocations:

$$[\![\mathsf{x} := \texttt{\{\}}]\!](\sigma) = (\mathsf{next}(\ell), \rho[\mathsf{x} \mapsto a], \overline{\kappa}, h[a \mapsto (\ell, \epsilon, \epsilon)])$$

where $\sigma = (\ell, \rho, \overline{\kappa}, h)$ and $a \notin \text{Domain}(h)$

- <u>Function Calls</u>:

$$[\![\mathsf{x} \coloneqq e\texttt{(}e_1 \cdots e_n\texttt{)}]\!](\sigma) = (\ell', \rho', \kappa :: \overline{\kappa}, h)$$

  where

$$\begin{aligned} \sigma &= (\ell, \rho, \overline{\kappa}, h) \\ [\![e]\!](\sigma) &= f = \cdots \texttt{(}\mathsf{x}_1, \cdots, \mathsf{x}_n\texttt{)}\ \texttt{\{}\ell' : \cdots\texttt{\}} \\ \rho' &= \bot[\mathsf{x}_1 \mapsto [\![e_1]\!](\sigma), \cdots, \mathsf{x}_n \mapsto [\![e_n]\!](\sigma)] \\ \kappa &= (\ell, \rho) \end{aligned}$$

- <u>Branches</u>:

$$[\![\texttt{if}\ e\ \ell_{\mathtt{t}}\ \ell_{\mathtt{f}}]\!](\sigma) = \begin{cases} (\ell_{\mathtt{t}}, \rho, \overline{\kappa}, h) & \text{if } [\![e]\!](\sigma) = \texttt{\#t} \\ (\ell_{\mathtt{f}}, \rho, \overline{\kappa}, h) & \text{if } [\![e]\!](\sigma) = \texttt{\#f} \end{cases}$$

- <u>Returns</u>:

$$[\![\texttt{return}\ e]\!](\sigma) = (\mathsf{next}(\ell), \rho[\mathsf{x} \mapsto v], \overline{\kappa}, h)$$

  where

$$\begin{aligned} \sigma &= (\_, \_, (\ell, \rho) :: \overline{\kappa}, h) \\ [\![e]\!](\sigma) &= v \\ \mathsf{inst}(\ell) &= \mathsf{x} \coloneqq \cdots \end{aligned}$$

**Expressions:** $\boxed{[\![e]\!] : \mathbb{S} \to \mathbb{V}}$

- <u>Primitive Values</u>:

$$[\![v^{\mathsf{p}}]\!](\sigma) = v^{\mathsf{p}}$$

- <u>Primitive Operations</u>:

$$[\![\mathsf{op}\texttt{(}e_1, \cdots, e_n\texttt{)}]\!](\sigma) = \mathsf{op}(v_1^{\mathsf{p}}, \cdots, v_n^{\mathsf{p}})$$

  where $\forall 1 \leq j \leq n.\ [\![e_k]\!](\sigma) = v_j^{\mathsf{p}}$

- <u>Variable Lookups</u>:

$$[\![\mathsf{x}]\!](\sigma) = \rho(\mathsf{x})$$

  where $\sigma = (\_, \rho, \_, \_)$

- <u>Internal Field Lookups</u>:

$$[\![e_0\texttt{[}e_1\texttt{]}]\!](\sigma) = v$$

  where

$$\begin{aligned} \sigma &= (\_, \_, \_, h) \\ v_0 &= [\![e_0]\!](\sigma) \\ v_1 &= [\![e_1]\!](\sigma) \\ v &= \begin{cases} m(s) & \text{if } (v_0, v_1) = (a, s) \wedge h(a) = (\_, m, \_) \\ \omega_j & \text{if } (v_0, v_1) = (\tau_k\langle \omega_1, \cdots, \omega_n \rangle, j) \\ \tau_k.\texttt{eval} & \text{if } (v_0, v_1) = (\tau_k\langle \omega_1, \cdots, \omega_n \rangle, \texttt{"eval"}) \end{cases} \end{aligned}$$

- <u>External Field Lookups</u>:

$$[\![e_0\texttt{[}e_1\texttt{]}_{\mathsf{js}}]\!](\sigma) = v$$

  where

$$\begin{aligned} \sigma &= (\_, \_, \_, h) \\ (a, s) &= ([\![e_0]\!](\sigma), [\![e_1]\!](\sigma)) \\ h(a) &= (\_, \_, m_{\mathsf{js}}) \\ v &= m_{\mathsf{js}}(s) \end{aligned}$$

### 5.3.2 JavaScript Definitional Interpreter

In a similar way to $[\![P]\!]$, the collecting semantics $[\![P_{\mathsf{js}}]\!]_{\mathsf{js}}$ of a JavaScript program $P_{\mathsf{js}}$ is a set of all reachable JavaScript states $\mathcal{P}(\mathbb{S}_{\mathsf{js}})$ from the initial JavaScript states $\mathbb{S}^{\iota}_{\mathsf{js}} \subseteq \mathbb{S}_{\mathsf{js}}$. Then, we define a *definitional interpreter* for JavaScript as an $\text{IR}_{\text{ES}}$ program:

**Definition 5.3.1** (JavaScript Definitional Interpreter)**.** An $\text{IR}_{\text{ES}}$ program $P$ is a JavaScript *definitional interpreter* if and only if the following condition holds for each JavaScript program $P_{\mathsf{js}} \in \mathfrak{P}_{\mathsf{js}}$:

$$[\![P_{\mathsf{js}}]\!]_{\mathsf{js}} = \mathsf{decode} \circ [\![P]\!]^{\mathbf{R}} \circ \mathsf{encode}(P_{\mathsf{js}})$$

where $\mathsf{encode} : \mathfrak{P}_{\mathsf{js}} \to \mathcal{P}(\mathbb{S})$ encodes a JavaScript program to $\text{IR}_{\text{ES}}$ states and $\mathsf{decode} : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S}_{\mathsf{js}})$ decodes $\text{IR}_{\text{ES}}$ states to JavaScript states.

Thus, a restricted semantics of the definitional interpreter with a JavaScript program $P_{\mathsf{js}}$ indirectly represents the collecting semantics $[\![P_{\mathsf{js}}]\!]_{\mathsf{js}}$ of the JavaScript program $P_{\mathsf{js}}$. We utilize JISET to automatically extract such JavaScript definitional interpreters from any versions of ECMAScript.

### 5.3.3 JavaScript Meta-Level Static Analysis

For a JavaScript *meta-level static analysis*, we define an abstract semantics of $\text{IR}_{\text{ES}}$ in the abstract interpretation framework with *view-based analysis sensitivities* [52, 73].

**Abstract Domains** We first define the abstract domain for each structure. We define an analysis sensitivity as a *view abstraction* $\delta : \Pi \to \mathcal{P}(\mathbb{S})$, a function from finite *views* to sets of states. Thus, a sensitive abstract state is defined as a function from pairs of labels and views to abstract states:

- Sensitive Abstract States: $\widehat{\mathbb{D}}_\delta = \mathcal{L} \times \Pi \to \widehat{\mathbb{S}}$

$$\begin{array}{lll}
\gamma & : & \widehat{\mathbb{D}}_\delta \to \mathcal{P}(\mathbb{S}) \\
\gamma(\widehat{d}_\delta) & = & \{\sigma \in \mathbb{S} \mid \forall n \geq 0.\ \mathsf{caller}^n(\sigma) = \sigma' \Rightarrow ( \\
& & \quad\quad \forall(\ell, \pi) \in \mathcal{L} \times \Pi. \\
& & \quad\quad \sigma' = (\ell, \_, \_, \_) \in \delta(\pi) \Rightarrow \sigma' \in \gamma \circ \widehat{d}_\delta(\pi) \\
& & )\} \\
\widehat{d}_\delta \sqsubseteq \widehat{d}'_\delta & \Leftrightarrow & \forall(\ell, \pi) \in \Pi.\ \widehat{d}_\delta(\ell, \pi) \sqsubseteq \widehat{d}'_\delta(\ell, \pi) \\
\widehat{d}_\delta \sqcup \widehat{d}'_\delta & = & \lambda(\ell, \pi) \in \Pi.\ \widehat{d}_\delta(\ell, \pi) \sqcup \widehat{d}'_\delta(\ell, \pi) \\
\widehat{d}_\delta \sqcap \widehat{d}'_\delta & = & \lambda(\ell, \pi) \in \Pi.\ \widehat{d}_\delta(\ell, \pi) \sqcap \widehat{d}'_\delta(\ell, \pi)
\end{array}$$

- Abstract States: $\widehat{\mathbb{S}} = \widehat{\mathbb{E}} \times \widehat{\mathbb{K}} \times \widehat{\mathbb{H}}$

$$\begin{array}{lll}
\gamma & : & \widehat{\mathbb{S}} \to \mathcal{P}(\mathbb{S}) \\
\gamma(\widehat{\sigma}) & = & \{\sigma \in \mathbb{S} \mid \rho \in \gamma(\widehat{\rho}) \wedge \sigma \in \gamma(\widehat{\kappa}) \wedge (h, \_) \in \gamma(\widehat{h})\} \\
& & \quad\quad \text{where } \widehat{\sigma} = (\widehat{\rho}, \widehat{\kappa}, \widehat{h}) \text{ and } \sigma = (\_, \rho, \_, h) \\
\widehat{\sigma} \sqsubseteq \widehat{\sigma}' & \Leftrightarrow & \widehat{\rho} \sqsubseteq \widehat{\rho}' \wedge \widehat{\kappa} \sqsubseteq \widehat{\kappa}' \wedge \widehat{h} \sqsubseteq \widehat{h}' \\
\widehat{\sigma} \sqcup \widehat{\sigma}' & = & (\widehat{\rho} \sqcup \widehat{\rho}', \widehat{\kappa} \sqcup \widehat{\kappa}', \widehat{h} \sqcup \widehat{h}') \\
\widehat{\sigma} \sqcap \widehat{\sigma}' & = & (\widehat{\rho} \sqcap \widehat{\rho}', \widehat{\kappa} \sqcap \widehat{\kappa}', \widehat{h} \sqcap \widehat{h}')
\end{array}$$

- Abstract Environments: $\widehat{\mathbb{E}} = \mathcal{X} \to \widehat{\mathbb{V}}$

$$\begin{array}{lcl} \gamma & : & \widehat{\mathbb{E}} \to \mathcal{P}(\mathbb{E}) \\ \gamma(\widehat{\rho}) & = & \{\rho \in \mathbb{E} \mid \forall \mathsf{x} \mapsto v \in \rho.\ v \in \gamma \circ \widehat{\rho}(\mathsf{x})\} \\ \widehat{\rho} \sqsubseteq \widehat{\rho}' & \Leftrightarrow & \forall \mathsf{x} \in \mathcal{X}.\ \widehat{\rho}(\mathsf{x}) \sqsubseteq \widehat{\rho}'(\mathsf{x}) \\ \widehat{\rho} \sqcup \widehat{\rho}' & \Leftrightarrow & \lambda \mathsf{x} \in \mathcal{X}.\ \widehat{\rho}(\mathsf{x}) \sqcup \widehat{\rho}'(\mathsf{x}) \\ \widehat{\rho} \sqcap \widehat{\rho}' & \Leftrightarrow & \lambda \mathsf{x} \in \mathcal{X}.\ \widehat{\rho}(\mathsf{x}) \sqcap \widehat{\rho}'(\mathsf{x}) \end{array}$$

- Abstract Contexts: $\widehat{\mathbb{K}} = \mathcal{P}(\mathcal{L} \times \Pi)$

$$\begin{array}{lcl} \gamma & : & \widehat{\mathbb{K}} \to \mathcal{P}(\mathbb{S}) \\ \gamma(\widehat{\kappa}) & = & \{\sigma \in \mathbb{S} \mid \mathsf{caller}(\sigma) = \sigma' = (\ell, \_, \_, \_) \Rightarrow \qquad \exists (\ell, \pi) \in \widehat{\kappa}.\ \sigma' \in \delta(\pi)\} \\ \widehat{\kappa} \sqsubseteq \widehat{\kappa}' & \Leftrightarrow & \widehat{\kappa} \subseteq \widehat{\kappa}' \\ \widehat{\kappa} \sqcup \widehat{\kappa}' & = & \widehat{\kappa} \cup \widehat{\kappa}' \\ \widehat{\kappa} \sqcap \widehat{\kappa}' & = & \widehat{\kappa} \cap \widehat{\kappa}' \end{array}$$

- Abstract Heaps: $\widehat{\mathbb{H}} = \widehat{\mathbb{A}} \to \widehat{\mathbb{M}} \times \widehat{\mathbb{M}_{\mathsf{js}}}$

$$\begin{array}{lcl} \gamma & : & \widehat{\mathbb{H}} \to \mathcal{P}(\mathbb{H}) \\ \gamma(\widehat{h}) & = & \{h \in \mathbb{H} \mid \forall a \mapsto (\ell, m, m_{\mathsf{js}}) \in h.\ \ell = \eta(a) \wedge (\widehat{m}, \widehat{m_{\mathsf{js}}}) = \widehat{h}(\ell) \wedge m \in \gamma(\widehat{m}) \wedge m_{\mathsf{js}} \in \gamma(\widehat{m_{\mathsf{js}}})\} \\ \widehat{h} \sqsubseteq \widehat{h}' & \Leftrightarrow & \forall \widehat{a} \in \widehat{\mathbb{A}}.\ \widehat{m} \sqsubseteq \widehat{m}' \wedge \widehat{m_{\mathsf{js}}} \sqsubseteq \widehat{m_{\mathsf{js}}}' \\ \widehat{h} \sqcup \widehat{h}' & = & \lambda \widehat{a} \in \widehat{\mathbb{A}}.\ (\widehat{m} \sqcup \widehat{m}', \widehat{m_{\mathsf{js}}} \sqcup \widehat{m_{\mathsf{js}}}') \\ \widehat{h} \sqcap \widehat{h}' & = & \lambda \widehat{a} \in \widehat{\mathbb{A}}.\ (\widehat{m} \sqcap \widehat{m}', \widehat{m_{\mathsf{js}}} \sqcap \widehat{m_{\mathsf{js}}}') \end{array}$$

$$\text{where } \widehat{h}(\widehat{a}) = (\widehat{m}, \widehat{m_{\mathsf{js}}}) \text{ and } \widehat{h}'(\widehat{a}) = (\widehat{m}', \widehat{m_{\mathsf{js}}}')$$

- Abstract Internal Field Maps: $\widehat{\mathbb{M}} = \mathbb{V}_{\mathsf{str}} \to \widehat{\mathbb{V}}$

$$\begin{array}{lcl} \gamma & : & \widehat{\mathbb{M}} \to \mathcal{P}(\mathbb{M}) \\ \gamma(\widehat{m}) & = & \{m \in \mathbb{M} \mid \forall s \mapsto v \in m.\ v \in \gamma \circ \widehat{m}(s)\} \\ \widehat{m} \sqsubseteq \widehat{m}' & \Leftrightarrow & \forall s \in \mathbb{V}_{\mathsf{str}}.\ \widehat{m}(s) \sqsubseteq \widehat{m}'(s) \\ \widehat{m} \sqcup \widehat{m}' & = & \lambda s \in \mathbb{V}_{\mathsf{str}}.\ \widehat{m}(s) \sqcup \widehat{m}'(s) \\ \widehat{m} \sqcap \widehat{m}' & = & \lambda s \in \mathbb{V}_{\mathsf{str}}.\ \widehat{m}(s) \sqcap \widehat{m}'(s) \end{array}$$

- Abstract External Field Maps: $\widehat{\mathbb{M}_{\mathsf{js}}} = \mathbb{V}_{\mathsf{str}} \to \widehat{\mathbb{V}}$

$$\begin{array}{lcl} \gamma & : & \widehat{\mathbb{M}_{\mathsf{js}}} \to \mathcal{P}(\mathbb{M}_{\mathsf{js}}) \\ \gamma(\widehat{m_{\mathsf{js}}}) & = & \{m_{\mathsf{js}} \in \mathbb{M}_{\mathsf{js}} \mid \forall s \mapsto v \in m_{\mathsf{js}}.\ v \in \gamma \circ \widehat{m_{\mathsf{js}}}(s)\} \\ \widehat{m_{\mathsf{js}}} \sqsubseteq \widehat{m_{\mathsf{js}}}' & \Leftrightarrow & \forall s \in \mathbb{V}_{\mathsf{str}}.\ \widehat{m_{\mathsf{js}}}(s) \sqsubseteq \widehat{m_{\mathsf{js}}}'(s) \\ \widehat{m_{\mathsf{js}}} \sqcup \widehat{m_{\mathsf{js}}}' & = & \lambda s \in \mathbb{V}_{\mathsf{str}}.\ \widehat{m_{\mathsf{js}}}(s) \sqcup \widehat{m_{\mathsf{js}}}'(s) \\ \widehat{m_{\mathsf{js}}} \sqcap \widehat{m_{\mathsf{js}}}' & = & \lambda s \in \mathbb{V}_{\mathsf{str}}.\ \widehat{m_{\mathsf{js}}}(s) \sqcap \widehat{m_{\mathsf{js}}}'(s) \end{array}$$

- Abstract Values: $\widehat{\mathbb{V}} = \mathcal{P}(\widehat{\mathbb{A}} \uplus \mathbb{V}^{\mathsf{p}} \uplus \Omega \uplus \mathcal{F})$

$$\begin{array}{lcl} \gamma & : & \widehat{\mathbb{V}} \to \mathcal{P}(\mathbb{V}) \\ \gamma(\widehat{v}) & = & (\widehat{v} \setminus \widehat{\mathbb{A}}) \uplus \{a \in \mathbb{A} \mid \eta(a) \in \widehat{v}\} \\ \widehat{v} \sqsubseteq \widehat{v}' & \Leftrightarrow & \widehat{v} \subseteq \widehat{v}' \\ \widehat{v} \sqcup \widehat{v}' & = & \widehat{v} \cup \widehat{v}' \\ \widehat{v} \sqcap \widehat{v}' & = & \widehat{v} \cap \widehat{v}' \end{array}$$

An abstract state $\widehat{\sigma} \in \widehat{\mathbb{S}}$ consists of an abstract environment, an abstract context, and an abstract heap. An abstract environment $\widehat{\rho} \in \widehat{\mathbb{E}}$ maps variables to abstract values. An abstract context $\widehat{\kappa} \in \widehat{\mathbb{K}}$ is a set of pairs of labels and views for callers. An abstract heap $\widehat{h} \in \widehat{\mathbb{H}}$ is a function from abstract addresses to pairs of abstract internal and external field maps. An abstract field map is a function from strings to abstract values. An abstract address $\widehat{a} \in \widehat{\mathbb{A}}$ is defined with the *allocation-site abstraction* [20], which partitions concrete addresses $\mathbb{A}$ based on their allocation sites $\mathcal{L}$. An abstract value $\widehat{v} \in \widehat{\mathbb{V}}$ is a set of abstract addresses and non-address values. While we use concrete strings in abstract field maps and sets of primitive values in abstract values in this formalization for brevity, we abstract them to bound the height of their lattices as finite in the implementation. We define a partial order $\sqsubseteq$, a join operator $\sqcup$, and a meet operator $\sqcap$. Then, we define the concretization function $\gamma$ for each abstract domain with the following a helper function $\mathsf{caller} : \mathbb{S} \rightarrowtail \mathbb{S}$ to get callers' states:

$$\sigma = (\_, \_, (\ell, \rho) :: \overline{\kappa}, h) \Rightarrow \mathsf{caller}(\sigma) = (\ell, \rho, \overline{\kappa}, h)$$

and a *valuation* [26] $\eta : \mathbb{A} \to \widehat{\mathbb{A}}$ to correctly concretize abstract addresses.

**Abstract Semantics** Using abstract domains, we define the *abstract semantics* $\widehat{[\![P]\!]}$ of an $\mathrm{IR}_{\mathrm{ES}}$ program $P$:

$$\widehat{[\![P]\!]} = \lim_{n \to \infty} \widehat{F}^n(\widehat{d}^\iota_\delta)$$

with an *initial sensitive abstract state* $\widehat{d}^\iota_\delta$ (i.e., $\mathbb{S}^\iota \subseteq \gamma(\widehat{d}^\iota_\delta)$) and an *abstract transfer function* $\widehat{F} : \widehat{\mathbb{D}}_\delta \to \widehat{\mathbb{D}}_\delta$:

$$\widehat{F}(\widehat{d}_\delta) = \widehat{d}_\delta \sqcup \bigsqcup_{(\ell,\pi) \in \mathcal{L} \times \Pi} \delta\widehat{[\![\mathsf{inst}(\ell)]\!]}(\ell, \pi, \widehat{d}_\delta(\ell, \pi))$$

where $\delta\widehat{[\![i]\!]} : \mathcal{L} \times \Pi \times \widehat{\mathbb{S}} \to \widehat{\mathbb{D}}_\delta$ is an abstract semantics of a view abstraction $\delta : \Pi \to \mathcal{P}(\mathbb{S})$.

**Restricted Abstract Semantics** Then, we also define the *restricted abstract semantics* $\widehat{[\![P]\!]^{\mathbf{R}}} : \widehat{\mathbb{D}}_\delta \to \widehat{\mathbb{D}}_\delta$ of an $\mathrm{IR}_{\mathrm{ES}}$ program $P$ with a given sensitive abstract state:

$$\widehat{[\![P]\!]^{\mathbf{R}}}(\widehat{d}_\delta) = \lim_{n \to \infty} \widehat{F}^n(\widehat{d}^\iota_\delta \sqcap \widehat{d}_\delta)$$

**Meta-Level Static Analysis** Finally, we define a JavaScript meta-level static analysis using the restricted abstract semantics $\widehat{[\![P]\!]^{\mathbf{R}}}$ of a JavaScript definitional interpreter $P$:

**Definition 5.3.2** (JavaScript Meta-Level Static Analysis)**.** A JavaScript *meta-level static analysis* is a way to indirectly analyze a JavaScript program $P_{\mathsf{js}}$ using a restricted abstract semantics $\widehat{[\![P]\!]^{\mathbf{R}}}$ of a JavaScript definitional interpreter $P$:

$$[\![P_{\mathsf{js}}]\!]_{\mathsf{js}} \subseteq \widehat{\mathsf{decode}} \circ \widehat{[\![P]\!]^{\mathbf{R}}} \circ \widehat{\mathsf{encode}}(P_{\mathsf{js}})$$

where $\widehat{\mathsf{encode}} : \mathfrak{P}_{\mathsf{js}} \to \widehat{\mathbb{D}}_\delta$ encodes a JavaScript program to a sensitive abstract state and $\widehat{\mathsf{decode}} : \widehat{\mathbb{D}}_\delta \to \mathcal{P}(\mathbb{S}_{\mathsf{js}})$ decodes a sensitive abstract state to JavaScript states.

### Flow-Sensitivity for $\mathrm{IR}_{\mathrm{ES}}$

We define the flow-sensitivity for $\mathrm{IR}_{\mathrm{ES}}$ with a view abstraction $\delta^{\mathsf{flow}} : \{\pi\} \to \mathcal{P}(\mathbb{S})$:

$$\delta^{\mathsf{flow}}(\pi) = \mathbb{S}$$

Because we use flow-sensitive abstract states, the view abstraction for flow-sensitivity consists of a single view $\pi$. Then, we define the abstract semantics $\delta^{\mathsf{flow}}\widehat{[\![i]\!]} : \mathcal{L} \times \{\pi\} \times \widehat{\mathbb{S}} \to \widehat{\mathbb{D}}_{\delta^{\mathsf{flow}}}$ of Instructions and the abstract semantics $\widehat{[\![e]\!]} : \widehat{\mathbb{S}} \to \widehat{\mathbb{V}}$ of expressions in the flow-sensitivity for IR$_{\text{ES}}$:

- **Instructions:** $\boxed{\delta^{\mathsf{flow}}\widehat{[\![i]\!]} : \mathcal{L} \times \{\pi\} \times \widehat{\mathbb{S}} \to \widehat{\mathbb{D}}_{\delta^{\mathsf{flow}}}}$
  - Variable Assignments:

$$\delta^{\mathsf{flow}}\widehat{[\![\mathsf{x} := e]\!]}(\ell, \pi, \widehat{\sigma}) = \bot[(\ell', \pi) \mapsto \widehat{\sigma}']$$

where

$$\begin{aligned} \ell' &= \mathsf{next}(\ell) \\ \widehat{\sigma} &= (\widehat{\rho}, \widehat{\kappa}, \widehat{h}) \\ \widehat{v} &= \widehat{[\![e]\!]}(\widehat{\sigma}) \\ \widehat{\sigma}' &= (\widehat{\rho}[\mathsf{x} \mapsto \widehat{v}], \widehat{\kappa}, \widehat{h}) \end{aligned}$$

  - Internal Field Assignments:

$$\delta^{\mathsf{flow}}\widehat{[\![e_0\,\texttt{[}e_1\texttt{]} := e_2]\!]}(\ell, \pi, \widehat{\sigma}) = \bot[(\ell', \pi) \mapsto \widehat{\sigma}']$$

where

$$\begin{aligned} \ell' &= \mathsf{next}(\ell) \\ \widehat{\sigma} &= (\widehat{\rho}, \widehat{\kappa}, \widehat{h}) \\ (\widehat{v}_0, \widehat{v}_1, \widehat{v}_2) &= (\widehat{[\![e_0]\!]}(\widehat{\sigma}), \widehat{[\![e_1]\!]}(\widehat{\sigma}), \widehat{[\![e_2]\!]}(\widehat{\sigma})) \\ \widehat{v}_0 \cap \widehat{\mathbb{A}} &= \{\widehat{a}_1, \cdots, \widehat{a}_n\} \\ \widehat{h}' &= \widehat{h}[\widehat{a}_1 \mapsto (\widehat{m}'_1, \widehat{m_{\mathsf{js}}}_1), \cdots, \widehat{a}_n \mapsto (\widehat{m}'_n, \widehat{m_{\mathsf{js}}}_n)] \\ \widehat{v}_1 \cap \mathbb{V}_{\mathsf{str}} &= \{s_1, \cdots, s_m\} \\ \forall 1 \leq j \leq n. & \\ (\widehat{m}_j, \widehat{m_{\mathsf{js}}}_j) &= \widehat{h}(\widehat{a}_j) \\ \widehat{m}'_j &= \widehat{m}_j \sqcup \bot[s_1 \mapsto \widehat{v}_2, \cdots, s_m \mapsto \widehat{v}_2] \\ \widehat{\sigma}' &= (\widehat{\rho}, \widehat{\kappa}, \widehat{h}') \end{aligned}$$

  - External Field Assignments:

$$\delta^{\mathsf{flow}}\widehat{[\![e_0\,\texttt{[}e_1\texttt{]} := e_2]\!]}(\ell, \pi, \widehat{\sigma}) = \bot[(\ell', \pi) \mapsto \widehat{\sigma}']$$

where

$$\begin{aligned} \ell' &= \mathsf{next}(\ell) \\ \widehat{\sigma} &= (\widehat{\rho}, \widehat{\kappa}, \widehat{h}) \\ (\widehat{v}_0, \widehat{v}_1, \widehat{v}_2) &= (\widehat{[\![e_0]\!]}(\widehat{\sigma}), \widehat{[\![e_1]\!]}(\widehat{\sigma}), \widehat{[\![e_2]\!]}(\widehat{\sigma})) \\ \widehat{v}_0 \cap \widehat{\mathbb{A}} &= \{\widehat{a}_1, \cdots, \widehat{a}_n\} \\ \widehat{h}' &= \widehat{h}[\widehat{a}_1 \mapsto (\widehat{m}_1, \widehat{m_{\mathsf{js}}}'_1), \cdots, \widehat{a}_n \mapsto (\widehat{m}_n, \widehat{m_{\mathsf{js}}}'_n)] \\ \widehat{v}_1 \cap \mathbb{V}_{\mathsf{str}} &= \{s_1, \cdots, s_m\} \\ \forall 1 \leq j \leq n. & \\ (\widehat{m}_j, \widehat{m_{\mathsf{js}}}_j) &= \widehat{h}(\widehat{a}_j) \\ \widehat{m_{\mathsf{js}}}'_j &= \widehat{m_{\mathsf{js}}}_j \sqcup \bot[s_1 \mapsto \widehat{v}_2, \cdots, s_m \mapsto \widehat{v}_2] \\ \widehat{\sigma}' &= (\widehat{\rho}, \widehat{\kappa}, \widehat{h}') \end{aligned}$$

  - Object Allocations:

$$\delta^{\mathsf{flow}}\widehat{[\![\mathsf{x} := \texttt{\{\}}]\!]}(\ell, \pi, \widehat{\sigma}) = \bot[(\ell', \pi) \mapsto \widehat{\sigma}']$$

where

$$\begin{aligned}
\ell' &= \mathsf{next}(\ell) \\
\widehat{\sigma} &= (\widehat{\rho}, \widehat{\kappa}, \widehat{h}) \\
\widehat{a} &= \ell \\
\widehat{\rho}' &= \widehat{\rho}[\mathsf{x} \mapsto \widehat{a}] \\
\widehat{h}' &= \widehat{h}[\widehat{a} \mapsto (\bot, \bot)] \\
\widehat{\sigma}' &= (\widehat{\rho}', \widehat{\kappa}, \widehat{h}')
\end{aligned}$$

– Function Calls:

$$\delta^{\mathsf{flow}}[\![\widehat{\mathsf{x} := e\,\mathtt{(}e_1 \cdots e_n\mathtt{)}}]\!](\ell, \pi, \widehat{\sigma}) = \widehat{d}'_{\delta^{\mathsf{flow}}}$$

where

$$\begin{aligned}
\widehat{\sigma} &= (\widehat{\rho}, \widehat{\kappa}, \widehat{h}) \\
\widehat{v} &= \widehat{[\![e]\!]}(\widehat{\sigma}) \\
\widehat{v}_j &= \widehat{[\![e_j]\!]}(\widehat{\sigma}) \ [\forall 1 \leq j \leq n] \\
F &= \widehat{v} \cap \mathcal{F} \\
\widehat{d}_{\delta^{\mathsf{flow}}} &= \lambda(\ell', \cdot) \in \mathcal{L} \times \{\pi\}. \\
&\quad \begin{cases} \widehat{\sigma}' & \text{if } \exists f \in F.\ f = \cdots\,\mathtt{(}\mathsf{x}_1, \cdots, \mathsf{x}_n\mathtt{)}\ \mathtt{\{}\ell' : \cdots\mathtt{\}} \\ \bot & \text{otherwise} \end{cases} \\
\widehat{\rho}' &= \bot[\mathsf{x}_1 \mapsto \widehat{v}_1, \cdots, \mathsf{x}_n \mapsto \widehat{v}_n] \\
\widehat{\sigma}' &= (\widehat{\rho}', \{(\ell, \pi)\}, \widehat{h}) \\
\ell'' &= \mathsf{next}(\ell) \\
\widehat{\sigma}'' &= (\widehat{\rho}, \bot, \bot) \\
\widehat{d}'_{\delta^{\mathsf{flow}}} &= \widehat{d}_{\delta^{\mathsf{flow}}}[(\ell'', \cdot) \mapsto \widehat{\sigma}'']
\end{aligned}$$

– Branches:

$$\delta^{\mathsf{flow}}[\![\widehat{\mathtt{if}\ e\ \ell_{\mathsf{t}}\ \ell_{\mathsf{f}}}]\!](\ell, \pi, \widehat{\sigma}) = \widehat{d}'_{\delta^{\mathsf{flow}}}$$

where

$$\begin{aligned}
\widehat{\sigma} &= (\widehat{\rho}, \widehat{\kappa}, \widehat{h}) \\
\widehat{v} &= \widehat{[\![e]\!]}(\widehat{\sigma}) \\
\widehat{d}_{\delta^{\mathsf{flow}}} &= \begin{cases} \bot[\ell_{\mathsf{t}} \mapsto \widehat{\sigma}] & \text{if } \mathtt{\#t} \in \widehat{v} \\ \bot & \text{otherwise} \end{cases} \\
\widehat{d}'_{\delta^{\mathsf{flow}}} &= \begin{cases} \widehat{d}_{\delta}[\ell_{\mathsf{f}} \mapsto \widehat{\sigma}] & \text{if } \mathtt{\#f} \in \widehat{v} \\ \widehat{d}_{\delta} & \text{otherwise} \end{cases}
\end{aligned}$$

– Returns:

$$\delta^{\mathsf{flow}}[\![\widehat{\mathtt{return}\ e}]\!](\ell, \pi, \widehat{\sigma}) = \widehat{d}_{\delta^{\mathsf{flow}}}$$

where

$$\begin{aligned}
\widehat{\sigma} &= (\widehat{\rho}, \widehat{\kappa}, \widehat{h}) \\
\widehat{v} &= \widehat{[\![e]\!]}(\widehat{\sigma}) \\
\widehat{d}_{\delta^{\mathsf{flow}}} &= \lambda(\ell', \cdot) \in \mathcal{L} \times \{\pi\}. \\
&\quad \begin{cases} \bot[\mathsf{x} \mapsto \widehat{v}] & \text{if } \exists(\ell', \cdot) \in \widehat{\kappa} \land \mathsf{inst}(\ell') = \mathsf{x} := \cdots \\ \bot & \text{otherwise} \end{cases}
\end{aligned}$$

• **Expressions:** $\boxed{\widehat{[\![e]\!]} : \widehat{\mathbb{S}} \rightarrow \widehat{\mathbb{V}}}$

– Primitive Values:

$$\widehat{[\![v^{\mathsf{p}}]\!]}(\widehat{\sigma}) = \{v^{\mathsf{p}}\}$$

– Primitive Operations:

$$\widehat{[\![\mathsf{op}(e_1, \cdots, e_n)]\!]}(\widehat{\sigma}) = \widehat{v}$$

where

$$\begin{array}{ll} \widehat{[\![e_j]\!]}(\widehat{\sigma}) & = \widehat{v}_j \ [\forall 1 \leq j \leq n] \\ \widehat{v} & = \dot{\mathsf{op}}(\widehat{v}_1 \cap \mathbb{V}^{\mathsf{p}}, \cdots, \widehat{v}_n \cap \mathbb{V}^{\mathsf{p}}) \end{array}$$

– Variable Lookups:

$$\widehat{[\![\mathsf{x}]\!]}(\widehat{\sigma}) = \widehat{\rho}(\mathsf{x})$$

where $\widehat{\sigma} = (\widehat{\rho}, \_, \_)$

– Internal Field Lookups:

$$\widehat{[\![e_0 \, [\![ e_1 ]\!]]\!]}(\widehat{\rho}) = \widehat{v}$$

where

$$\begin{array}{lcl} \widehat{[\![e_0]\!]}(\widehat{\rho}) & = & \widehat{v}_0 \\ \widehat{[\![e_1]\!]}(\widehat{\rho}) & = & \widehat{v}_1 \\ \widehat{v} & = & (\text{a point-wise internal field lookup definition with } \widehat{v}_0 \text{ and } \widehat{v}_1) \end{array}$$

– External Field Lookups:

$$\widehat{[\![e_0 \, [ e_1 ]]\!]}(\widehat{\rho}) = \widehat{v}$$

where

$$\begin{array}{lcl} \widehat{[\![e_0]\!]}(\widehat{\rho}) & = & \widehat{v}_0 \\ \widehat{[\![e_1]\!]}(\widehat{\rho}) & = & \widehat{v}_1 \\ \widehat{v} & = & (\text{a point-wise external field lookup definition with } \widehat{v}_0 \text{ and } \widehat{v}_1) \end{array}$$

**Callsite-Sensitivity for IR$_{\text{ES}}$**

We define the *callsite-sensitivity* [88, 89] for IR$_{\text{ES}}$ with a view abstraction $\delta^{k\text{-cfa}} : \mathcal{L}^{\leq k} \to \mathcal{P}(\mathbb{S})$:

$$\begin{array}{c} \delta^{k\text{-cfa}}([\ell_1, \cdots, \ell_n]) = \{\sigma = (\_, \_, [\kappa_1, \cdots, \kappa_m], \_) \in \mathbb{S} \mid \\ (n = k \leq m \vee n = m) \wedge \forall 1 \leq i \leq n.\ \kappa_i = (\ell_i, \_)\} \end{array}$$

We define the abstract semantics of the callsite-sensitivity for IR$_{\text{ES}}$ by modifying that of the flow-sensitivity for IR$_{\text{ES}}$ as follows:

$$\boxed{\delta^{k\text{-cfa}}\widehat{[\![i]\!]} : \mathcal{L} \times \mathcal{L}^{\leq k} \times \widehat{\mathbb{S}} \to \widehat{\mathbb{D}}_{\delta^{k\text{-cfa}}}}$$

- Function Calls:

$$\delta^{k\text{-cfa}}\widehat{[\![\mathsf{x} := e(e_1 \cdots e_n)]\!]}(\ell, [\ell_1, \cdots, \ell_n], \widehat{\sigma}) = \widehat{d}'_{\delta^{k\text{-cfa}}}$$

where

$$\begin{array}{l} \cdots \\ \widehat{d}_{\delta^{k\text{-cfa}}} = \lambda(\ell', [\ell'_1, \cdots, \ell'_m]) \in \mathcal{L} \times \mathcal{L}^{\leq k}. \\ \qquad \begin{cases} \widehat{\sigma}' & \text{if } \exists f \in F.\ f = \cdots (\mathsf{x}_1, \cdots, \mathsf{x}_n) \ \{\ell' : \cdots\} \\ & \qquad \left( \begin{array}{l} n = k = m \wedge \\ {[\ell', \ell_1, \cdots, \ell_n] = [\ell'_1, \cdots, \ell'_m, \ell_n]} \end{array} \right) \vee \\ & \qquad \left( \begin{array}{l} m = n + 1 \wedge \\ {[\ell', \ell_1, \cdots, \ell_n] = [\ell'_1, \cdots, \ell'_m]} \end{array} \right) \\ \bot & \text{otherwise} \end{cases} \\ \cdots \end{array}$$

**13.1.3 Runtime Semantics: Evaluation**

*IdentifierReference* **:** *Identifier*

1. Return ? ResolveBinding(StringValue of *Identifier*).

(a) Evaluation algorithm for identifier references

```
syntax def IdentifierReference[0].Evaluation(
  this, Identifier
) {
  return [? (ResolveBinding (Identifier.StringValue))]
}
```

(b) Extracted $\text{IR}_{\text{ES}}$ function for identifier references

Figure 5.7: The Evaluation algorithm and its compiled $\text{IR}_{\text{ES}}$ function for identifier references

### 5.3.4 Abstract Domains for JavaScript

Since the configuration of abstract domains in static analyzers allows fine-tuning the quality of analysis results, we provide a way to indirectly configure abstract domains for JavaScript *values* and *data structures* in a JavaScript meta-level static analysis.

**Values** Since a JavaScript value is also an $\text{IR}_{\text{ES}}$ value $v \in \mathbb{V}$, we can configure $\widehat{\mathbb{V}}$ for JavaScript values. For example, recall that Figure 5.5 shows the flow-sensitive analysis results of the code in Figure 5.2 using the interval domain. Assume that we desire to use the flat domain whose elements are concrete integer values, the bottom value $\perp_{\mathsf{int}}$ for nothing, and the top value $\top_{\mathsf{int}}$ for JavaScript integers. Then, it is sufficient to use the flat domain for integers in the $\text{IR}_{\text{ES}}$ abstract values $\widehat{\mathbb{V}}$. In this setting, the $\text{IR}_{\text{ES}}$ local variable `lval` points to $\top_{\mathsf{int}}$ at point `#1`. At the exit point, the $\text{IR}_{\text{ES}}$ function returns $\top_{\mathsf{int}}$ and the function object whose `name` property is a string `"f"`.

**Data Structures** In JavaScript, data structures including environment records and objects have *external* fields directly accessible by JavaScript syntax. For example, an environment record has variables as external fields, accessible by identifier references. Similarly, an object has properties as external fields accessible by property read expressions. However, they also have *internal* fields, which are not directly accessible by JavaScript syntax, and one should update them only indirectly. For example, an environment record has specific algorithms as internal methods such as `[[HasBinding]]` and `[[SetMutableBinding]]`. Similarly, an object has such internal methods, including `[[Get]]` and `[[Set]]`, and it also has internal fields such as `[[Prototype]]`. Such internal fields are pre-defined and the number of possible internal fields is finite. However, because one can dynamically create external fields using the `with` statement for environment records and property assignment expressions for objects, the number of external fields could be infinite. Since internal and external fields are quite different in this regard, we provide a way to configure them differently. In Section 5.3.3, we define an abstract heap $h \in \mathbb{H}$ as a finite mapping from abstract addresses $\widehat{\mathbb{A}}$ to pairs of abstract internal field maps $\widehat{\mathbb{M}}$ for internal fields and abstract external field maps $\widehat{\mathbb{M}_{\mathsf{js}}}$ for external fields. For example, one way to design them with different abstract domains is to define $\widehat{\mathbb{M}} : \mathbb{V}_{\mathsf{str}} \rightarrow \widehat{\mathbb{V}}$ as a mapping from internal fields to abstract values and $\widehat{\mathbb{M}_{\mathsf{js}}} : \widehat{\mathbb{V}_{\mathsf{str}}} \times \widehat{\mathbb{V}}$ as a single pair of merged external fields and merged values.

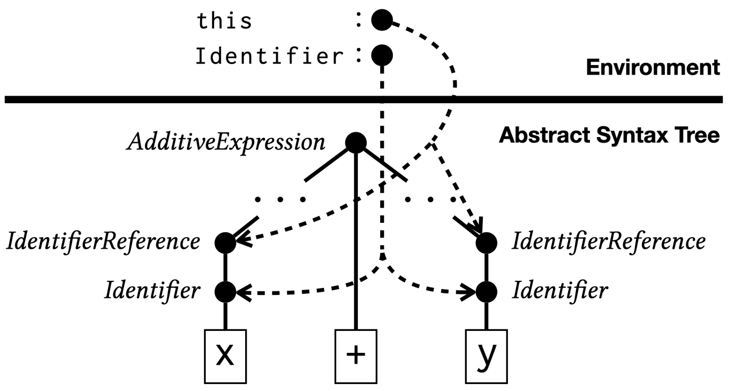


Figure 5.8: A JavaScript meta-level static analysis with the flow-sensitivity for $\text{IR}_{\text{ES}}$

## 5.3.5 Analysis Sensitivities for JavaScript

In a JavaScript meta-level static analysis, analysis sensitivities for JavaScript are different from those for $\text{IR}_{\text{ES}}$. For example, let us explain the analysis of the following JavaScript code with the flow-sensitivity for $\text{IR}_{\text{ES}}$:

```
let x = 1, y = 2;      x + y; // 3
```

Figure 5.7 shows (a) the Evaluation algorithm for identifier references and (b) its extracted $\text{IR}_{\text{ES}}$ function. Then, Figure 5.8 shows the parsing result of `x + y` and the initial local environment of the $\text{IR}_{\text{ES}}$ function. Since the flow-sensitivity merges states on the same labels, contexts for the evaluation of both identifier references `x` and `y` are merged. Thus, the $\text{IR}_{\text{ES}}$ variable `Identifier` points to their ASTs as illustrated at the bottom of Figure 5.7(c). Due to the imprecise merge of contexts, StringValue of `Identifier` returns `"x"` and `"y"`, and ResolveBinding with them returns both `1` and `2`. Finally, the analysis result of `x + y` becomes `{ 2, 3, 4 }`.

### Flow-Sensitivity for JavaScript

To resolve this problem, we present an *AST sensitivity* for $\text{IR}_{\text{ES}}$ as a variant of *object sensitivity* [62, 90] to represent flow-sensitivity for JavaScript. The object sensitivity uses abstract addresses $\widehat{\mathbb{A}}$ of receiver objects as views. However, the AST sensitivity utilizes JavaScript ASTs $\Omega$ stored in `this` parameter for syntax-directed functions as views with a view abstraction $\delta^{\text{js-flow}} : \Omega \uplus \{\bot\} \to \mathcal{P}(\mathbb{S})$:

$$\delta^{\text{js-flow}}(\omega_\bot) = \{\sigma = (\_, \_, \overline{\kappa}, \_) \in \mathbb{S} \mid \mathsf{ast}(\overline{\kappa}) = \omega_\bot\}$$

where $\mathsf{ast} : \mathbb{K}^* \to \Omega \uplus \{\bot\}$ denotes the JavaScript AST stored in `this` parameter of the top-most syntax-directed function for a given calling context stack:

$$\mathsf{ast}(\overline{\kappa}) = \begin{cases} \omega \text{ if } \exists\kappa.\ \overline{\kappa} = \kappa_1 :: \cdots :: \kappa_n :: \kappa :: \cdots \land \kappa = (\ell, \rho) \land \\ \qquad \mathsf{func}(\ell) = \mathtt{syntax\ def} \cdots \land \rho(\mathtt{this}) = \omega \land \\ \qquad \forall 1 \le j \le n.\ \kappa_j = (\ell_j, \_) \land \mathsf{func}(\ell_j) = \mathtt{def} \cdots \\ \bot \text{ otherwise} \end{cases}$$

Note that the number of views for the AST sensitivity is finite as well because JavaScript ASTs are finite in a JavaScript program. We define the flow-sensitivity for JavaScript using the AST sensitivity for $\text{IR}_{\text{ES}}$. It successfully divides contexts for the evaluation of JavaScript identifiers `x` and `y` in the example even though their labels in $\text{IR}_{\text{ES}}$ are the same.

We define the abstract semantics of the flow-sensitivity for JavaScript as follows:

$$\boxed{\delta^{\textsf{js-flow}}\widehat{[\![i]\!]} : \mathcal{L} \times (\Omega \uplus \{\bot\}) \times \widehat{\mathbb{S}} \to \widehat{\mathbb{D}}_{\delta_{\textsf{js-flow}}}}$$

- Function Calls:

$$\delta^{\textsf{js-flow}}[\![\texttt{x} := \widehat{e\texttt{(}e_1 \cdots e_n\texttt{)}}]\!](\ell, \omega_\bot, \widehat{\sigma}) = \widehat{d}'_{\delta_{\textsf{js-flow}}}$$

  where

$$\begin{aligned}
&\cdots\\
\widehat{d}_{\delta_{\textsf{js-flow}}} &= \lambda(\ell', \omega'_\bot) \in \mathcal{L} \times (\Omega \uplus \{\bot\}).\\
&\begin{cases}
\widehat{\sigma}' & \text{if } \exists f \in F.\ f = \cdots \texttt{(x}_1\texttt{,} \cdots \texttt{, x}_n\texttt{) \{}\ell' : \cdots\texttt{\}}\\
& \qquad \left(\begin{array}{l} f = \texttt{syntax def} \cdots \wedge \\ \omega'_\bot \in \widehat{v}_1 \end{array}\right) \vee \\
& \qquad \left(\begin{array}{l} f = \texttt{def} \cdots \wedge \\ \omega_\bot = \omega'_\bot \end{array}\right)\\
\bot & \text{otherwise}
\end{cases}\\
&\cdots
\end{aligned}$$

### Callsite-Sensitivity for JavaScript

We also formally define the *callsite-sensitivity* [88, 89] for JavaScript by extending the AST sensitivity for specific normal $\text{IR}_{\text{ES}}$ functions. In ECMAScript, all explicit and even implicit JavaScript function calls invoke normal $\text{IR}_{\text{ES}}$ functions Call and Construct. Thus, we define the callsite-sensitivity for JavaScript by extending the AST sensitivity with two normal $\text{IR}_{\text{ES}}$ functions with a view abstraction $\delta^{\textsf{js-}k\textsf{-cfa}} : \Omega^{\le k} \to \mathcal{P}(\mathbb{S})$:

$$\begin{aligned}
\delta^{\textsf{js-}k\textsf{-cfa}}([\omega_1, \cdots, \omega_n]) = \{\sigma = (\_, \_, \overline{\kappa}, \_) \in \mathbb{S} \mid\\
n \le k \wedge (n = k \vee \textsf{js-ctxt}^{n+1}(\overline{\kappa}) = \bot) \wedge\\
\forall 1 \le i \le n.\ \textsf{ast} \circ \textsf{js-ctxt}^i(\overline{\kappa}) = \omega_i\}
\end{aligned}$$

where $\textsf{js-ctxt} : \mathbb{K}^* \to \mathbb{K}^* \uplus \{\bot\}$ pops out calling contexts until the function of the top-most context is Call or Construct:

$$\textsf{js-ctxt}(\overline{\kappa}) = \begin{cases}
\overline{\kappa} & \text{if } \overline{\kappa} = (\ell, \rho) :: \_ \wedge\\
& \quad (\textsf{func}(\ell) = \texttt{def } \text{Call} \cdots \vee\\
& \quad\ \textsf{func}(\ell) = \texttt{def } \text{Construct} \cdots)\\
\textsf{js-ctxt}(\overline{\kappa}') & \text{if } \overline{\kappa} = \_ :: \overline{\kappa}'\\
\bot & \text{otherwise}
\end{cases}$$

Using this callsite-sensitivity for JavaScript, the meta-level static analyzer can discriminate not only explicit JavaScript function calls (e.g. `f()`) but also implicit JavaScript function calls, including getters/setters, user-defined implicit conversions, and implicit function calls in built-in libraries.

We define the abstract semantics of the callsite-sensitivity for JavaScript as follows:

$$\boxed{\delta^{\textsf{js-}k\textsf{-cfa}}\widehat{[\![i]\!]} : \mathcal{L} \times \Omega^{\le k} \times \widehat{\mathbb{S}} \to \widehat{\mathbb{D}}_{\delta_{\textsf{js-}k\textsf{-cfa}}}}$$

- Function Calls:

$$\delta^{\textsf{js-}k\textsf{-cfa}}[\![\texttt{x} := \widehat{e\texttt{(}e_1 \cdots e_n\texttt{)}}]\!](\ell, [\omega_1, \cdots, \omega_n], \widehat{\sigma}) = \widehat{d}'_{\delta_{\textsf{js-}k\textsf{-cfa}}}$$

where

$$
\begin{aligned}
&\cdots\\
&\widehat{d}_{\delta_{\text{js-}k\text{-cfa}}} = \lambda(\ell', [\omega'_1, \cdots, \omega'_m]) \in \mathcal{L} \times \Omega^{\leq k}.\\
&\quad\begin{cases}
\widehat{\sigma}' & \text{if } \exists f \in F.\ f = \cdots \texttt{(}\texttt{x}_1, \cdots, \texttt{x}_n\texttt{)}\ \texttt{\{}\ell' : \cdots\texttt{\}}\\
& \quad \omega' = (\text{an AST of the flow-sensitivity}\\
& \qquad \text{for JavaScript})\\
& \quad \begin{pmatrix} (f = \texttt{def}\ \text{Call} \cdots \vee \\ \ f = \texttt{def}\ \text{Construct} \cdots) \wedge \\ n = k = m \wedge \\ [\omega', \omega_1, \cdots, \omega_n] = [\omega'_1, \cdots, \omega'_m, \omega_n] \end{pmatrix} \vee \\
& \quad \begin{pmatrix} (f = \texttt{def}\ \text{Call} \cdots \vee \\ \ f = \texttt{def}\ \text{Construct} \cdots) \wedge \\ m = n + 1 \wedge \\ [\omega', \omega_1, \cdots, \omega_n] = [\omega'_1, \cdots, \omega'_m] \end{pmatrix} \vee \\
& \quad \begin{pmatrix} \neg(f = \texttt{def}\ \text{Call} \cdots \vee \\ \quad f = \texttt{def}\ \text{Construct} \cdots) \wedge \\ m = n \wedge \\ [\omega_1, \cdots, \omega_n] = [\omega'_1, \cdots, \omega'_m] \end{pmatrix} \\
\bot & \text{otherwise}
\end{cases}\\
&\cdots
\end{aligned}
$$

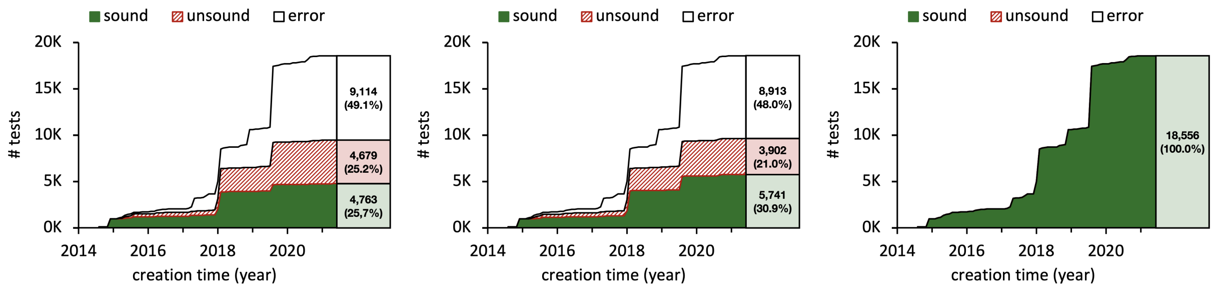
## 5.4 Implementation

We developed JSAVER as an open-source project[1] by extending JISET. In this section, we describe the challenges in implementing a meta-level static analyzer and present our solutions for them.

**Layered Abstract States** Unlike traditional JavaScript static analyses, a meta-level static analysis for JavaScript should track analysis results not only for JavaScript but also for $\text{IR}_{\text{ES}}$. Thus, the sizes of abstract states are much larger than those of the other JavaScript static analyzers. We implement *layered abstract states* to maintain only updated analysis results compared to the initial abstract state. It can reduce the time to perform the join $\sqcup$, meet $\sqcap$, and partial order $\sqsubseteq$ operations by considering only the updated parts in abstract states.

**Heap Cloning and Abstract Counting** Object properties in JavaScript could be dynamically added, modified, or deleted and even accessible by first-class property names. Thus, in JavaScript static analysis, performing *strong updates* rather than *weak updates* for object properties as many as possible is critical for precise analysis results. It becomes more important in a JavaScript meta-level static analysis than a traditional JavaScript static analysis because it should track even internal fields for $\text{IR}_{\text{ES}}$. Therefore, we implement *heap cloning* [55] and *abstract counting* [61] to increase the chances of performing strong updates for internal and external fields. Heap cloning refines allocation-sites using calling contexts for abstract addresses. Abstract counting checks how many times objects in the same allocation site have been allocated and performs strong updates only for singleton objects with singleton field values.

[1] `https://github.com/kaist-plrg/jsaver`



(a) Analysis results of TAJS (b) Analysis results of SAFE (c) Analysis results of JSA$_{\text{ES12}}$

Figure 5.9: Analysis results of TAJS, SAFE, and the derived ES12 static analyzer using JSAVER (JSA$_{\text{ES12}}$) for 18,556 applicable tests

**Loop Sensitivity** Similar to function calls, loops are also typical merging points in static analysis. Especially in the JavaScript static analysis, merged loop contexts often cause imprecise relations between object properties. Therefore, researchers presented diverse techniques to resolve this problem [54, 66, 92, 93]. Among them, a *loop sensitivity* [68, 69] is one of the representative techniques to increase the analysis precision by discriminating loop contexts. It is defined with two parameters $i$ and $j$ to discriminate loop contexts based on the maximum depth $i$ and the loop iteration $j$. We define it under the view-based analysis sensitivity and implement it in JSAVER to increase the precision of the IR$_{\text{ES}}$ static analysis. Moreover, we also define and implement the loop sensitivity for JavaScript using the loop sensitivity for IR$_{\text{ES}}$. Therefore, JSAVER discriminates contexts for explicit loops such as `for-in` and `for-of` and even implicit loops such as the assignment of arguments or the `length` property of arrays.

**Closures and Continuations** The defining-language IR$_{\text{ES}}$ contains more complex language features than its simplified version presented in Section 5.3.1, such as symbols, lists, and list operations. Among them, the two most complex features are *closures* with captured variables and first-class *continuations* because they introduce new kinds of control flows. ECMAScript uses closures to define implicit and explicit JavaScript iterators and uses continuations for iterators, generators, and asynchronous features such as `async`, `await`, and the `Proxy` object. Therefore, we define and implement abstract closures and abstract continuations; an abstract closure is a pair of a function and a mapping from captured variables to their abstract values, and an abstract continuation consists of a program point, a view for analysis sensitivity, parameters, and an abstract context.

## 5.5 Evaluation

We evaluate JSAVER using JSA$_{\text{ES12}}$, the JavaScript static analyzer derived from the latest ECMAScript (ES12, 2021) via JSAVER, with the following research questions:

- **RQ1: Soundness.** Can JSA$_{\text{ES12}}$ soundly analyze JavaScript programs using new language features?
- **RQ2: Precision.** Can JSA$_{\text{ES12}}$ precisely analyze JavaScript programs compared to the existing static analyzers?
- **RQ3: Configurability.** Can we configure abstract domains and analysis sensitivities for JavaScript in JSA$_{\text{ES12}}$?
- **RQ4: Adaptability.** Can JSAVER adapt to new language features not yet introduced in ES12?

Table 5.1: Applicable conformance tests in Test262

| **All Test262 Conformance Tests** | **41,415** |
|---|---|
| **Inapplicable Tests** | **22,859** |
| Web Browsers / Internationalization | 2,036 |
| In-Progress Features | 5,719 |
| Non-Strict / Module | 2,625 |
| Early Errors | 2,949 |
| Inessential Built-in Objects (e.g. `JSON`, `Atomics`) | 9,530 |
| **Applicable Tests** | **18,556** |

We performed experiments on an Ubuntu machine equipped with 4.2GHz Quad-Core Intel Core i7 and 32GB of RAM.

## 5.5.1 Soundness

To evaluate the soundness of $\textsf{JSA}_{\textsf{ES12}}$, we used Test262, the official conformance test suite. Since ES12 is officially released in June 2021, we used Test262 as of June 2021[2]. While it consists of 41,415 tests, it even contains tests using additional features for web browsers, in-progress features, modules, or early errors for the parsing process. To focus on the core language semantics of JavaScript in ES12, we excluded 22,859 tests for such features as summarized in Table 5.1 using JISET and analyzed 18,556 applicable Test262 tests, each of which is 211.8 lines on average. Moreover, we compared the soundness of $\textsf{JSA}_{\textsf{ES12}}$ with that of the existing JavaScript static analyzers, TAJS and SAFE. We used their default context sensitivities: the object sensitivity for TAJS and 20-callsite-sensitivity for SAFE. For a fair comparison, we used 20-callsite-sensitivity for $\textsf{JSA}_{\textsf{ES12}}$ as well.

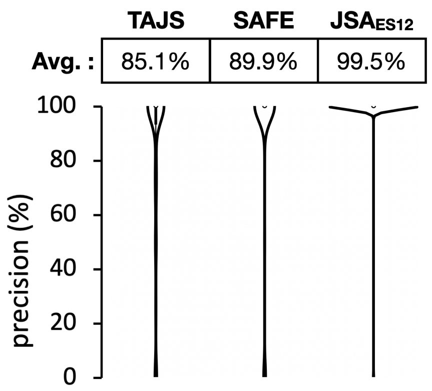
For each test program, we evaluated the soundness of an analyzer by comparing its analysis result with the final state of the program in concrete execution. The *comparison targets* are 1) the reachability of the exit and the exceptional exit points and 2) primitive values stored in variables and object properties at the exit point. We checked whether the analyzer over-approximates the expected values of comparison targets. For example, in the JavaScript program, `let x = 42; x++;`, only the exit point is reachable, and the variable `x` points to `43`. Thus, the analysis result should cover the reachability of the exit point and `43` in `x` for a sound result.

Figure 5.9 shows the analysis results of TAJS, SAFE, and $\textsf{JSA}_{\textsf{ES12}}$ for 18,556 applicable tests. In each chart, The $x$-axis denotes when tests are created, and the $y$-axis denotes the number of tests created before the time. The mark sound (green, filled) denotes a sound analysis, unsound (red, stripe) an unsound analysis, and error (white, blank) an unexpected error. The charts show that TAJS and SAFE analyzed most tests created before 2015 in a sound way. However, the number of tests that they cannot soundly analyze is consistently increased from 2015. TAJS and SAFE can soundly analyze only 4,763 (25.7%) and 5,741 (30.9%), respectively. On the other hand, $\textsf{JSA}_{\textsf{ES12}}$ successfully analyzes all 18,556 applicable test programs in a sound way.

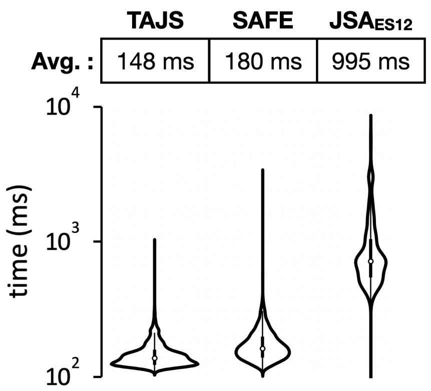
[2]`https://github.com/tc39/test262/tree/aaf4402b4ca9923012e6`

Table 5.2: Soundly analyzed tests for the top ten feature tags

| Feature Tag | Ver. | TAJS | SAFE | JSA$_{\text{ES12}}$ | # Tests |
|---|---|---|---|---|---|
| `destructuring-binding` | ES6 | 0 | 6 | 4,963 | 4,963 |
| `async-iteration` | ES9 | 0 | 0 | 2,835 | 2,835 |
| `generators` | ES6 | 19 | 0 | 2,450 | 2,450 |
| `default-parameters` | ES6 | 0 | 0 | 1,527 | 1,527 |
| `Symbol.iterator` | ES6 | 1 | 0 | 1,084 | 1,084 |
| `Symbol` | ES6 | 98 | 3 | 316 | 316 |
| `class` | ES6 | 0 | 0 | 205 | 205 |
| `BigInt` | ES11 | 0 | 0 | 199 | 199 |
| `object-rest` | ES9 | 0 | 0 | 190 | 190 |
| `async-functions` | ES8 | 0 | 0 | 142 | 142 |



(a) The analysis precision



(b) The analysis performance

Figure 5.10: The analysis precision and performance for 3,903 tests soundly analyzable by all of TAJS, SAFE, and JSA$_{\text{ES12}}$

In addition, Table 5.2 shows that JSA$_{\text{ES12}}$ can soundly analyze important new language features that TAJS and SAFE cannot. Since 2015 when ECMAScript began to be maintained in an open development process, each newly added Test262 test has been marked with tags of its related new language features. We counted how many applicable tests are related to each feature tag and how many of them analyzers can soundly analyze. The table shows the top ten language feature tags (**Feature Tag**) with the versions of ECMAScript in which they are introduced (**Ver.**), numbers of tests soundly analyzed by each analyzer, and the number of tests tagged with them (**# Tests**). This table shows that SAFE does not support most of the important new language features, and TAJS supports only a small part of the ES6 language features like `Symbol` and generators. However, JSA$_{\text{ES12}}$ supports all the language features introduced.

## 5.5.2 Precision

We measured the analysis precision by counting how many *comparison targets* were precisely analyzed. For all applicable 18,556 Test262 test programs, JSA$_{\text{ES12}}$ analyzed them with a high analysis precision of 99.0% in 1,591 ms on average. Then, we compared its analysis precision with that of TAJS

Table 5.3: Definitions of three string abstract domains: *String Set* ($\mathsf{SS}_k$), *Character Inclusion* ($\mathsf{CI}$), and *Prefix-Suffix* ($\mathsf{PS}$)

| **Domain** | **Definition** | | |
|---|---|---|---|
| $\mathsf{SS}_k$ | $\mathsf{SS}_k$ | $=$ | $\{\top\} \cup \{S \subseteq \Sigma^* \mid \|S\| \leq k\}$ |
| | $\gamma(S)$ | $=$ | $S$ |
| | $S \sqsubseteq S'$ | $\Leftrightarrow$ | $S \subseteq S'$ |
| | $S \sqcup S'$ | $=$ | $S \cup S'$ |
| | $S \sqcap S'$ | $=$ | $S \cap S'$ |
| | $S \cdot S'$ | $=$ | $\{s \cdot s' \mid s \in S \wedge s' \in S'\}$ |
| $\mathsf{CI}$ | $\mathsf{CI}$ | $=$ | $\{\bot\} \cup \{[L, U] \mid L, U \subseteq \Sigma \wedge L \subseteq U\}$ |
| | $\gamma([L, U])$ | $=$ | $\{w \in \Sigma^* \mid L \subseteq \mathsf{chars}(w) \subseteq U\}$ |
| | $[L, U] \sqsubseteq [L', U']$ | $\Leftrightarrow$ | $L' \subseteq L \wedge U \subseteq U'$ |
| | $[L, U] \sqcup [L', U']$ | $=$ | $[L \cap L', U \cup U']$ |
| | $[L, U] \sqcap [L', U']$ | $=$ | $[L \cup L', U \cap U']$ |
| | $[L, U] \cdot [L', U']$ | $=$ | $[L \cup L', U \cup U']$ |
| $\mathsf{PS}$ | $\mathsf{PS}$ | $=$ | $\{\bot\} \cup (\Sigma^* \times \Sigma^*)$ |
| | $\gamma(\langle p, s\rangle)$ | $=$ | $\{p \cdot w \mid w \in \Sigma^*\} \cap \{w \cdot s \mid w \in \Sigma^*\}$ |
| | $\langle p, s\rangle \sqsubseteq \langle p', s'\rangle$ | $\Leftrightarrow$ | $\mathsf{lcp}(\{p, p'\}) = p' \wedge \mathsf{lcs}(\{s, s'\}) = s'$ |
| | $\langle p, s\rangle \sqcup \langle p', s'\rangle$ | $=$ | $\langle \mathsf{lcp}(\{p, p'\}), \mathsf{lcs}(\{s, s'\})\rangle$ |
| | $\langle p, s\rangle \sqcap \langle p', s'\rangle$ | $=$ | (naturally induced by $\sqsubseteq$) |
| | $\langle p, s\rangle \cdot \langle p', s'\rangle$ | $=$ | $\langle p, s'\rangle$ |

and SAFE. For a fair comparison, we measured the analysis precision for 3,903 test programs soundly analyzable by all of TAJS, SAFE, and $\mathsf{JSA}_{\mathsf{ES12}}$. Figure 5.10(a) depicts the average and distribution of the analysis precision in violin plots [41]. TAJS and SAFE analyzed 3,903 test programs with 85.1% and 89.9% precision on average, respectively. $\mathsf{JSA}_{\mathsf{ES12}}$, on the contrary, has the highest analysis precision of 99.5%. However, the analysis speed of $\mathsf{JSA}_{\mathsf{ES12}}$ is slower than that of TAJS and SAFE, and Figure 5.10(b) depicts them in violin plots in a logarithmic scale. While TAJS and SAFE took 148 ms and 180 ms, respectively, to analyze 3,903 test programs on average, $\mathsf{JSA}_{\mathsf{ES12}}$ took 995 ms to analyze them because JSAVER derives precise abstract semantics for all language features. On the contrary, TAJS and SAFE developers often *imprecisely* or even *unsoundly* model the abstract semantics of specific language features to increase the analysis speed. For example, TAJS does not discriminate positive/negative infinity values or positive/negative zeros to reduce the number of possible cases in abstract values. Similarly, SAFE ignores the semantics of getters and setters to quickly analyze object property reads.

### 5.5.3 Configurability

We demonstrate the configurability of JSAVER with several case studies for abstract domains and analysis sensitivities. We discuss how different abstract domains or analysis sensitivities affect analysis results of $\mathsf{JSA}_{\mathsf{ES12}}$ with examples.

```
1 let x = /* "a" or "b" */;
2 let y = ‘c${x}d‘;    // "cad" or "cbd"
3 let z = ‘${x}e${x}‘; // "aea" or "beb"
```

Figure 5.11: A JavaScript program using template literals

**Abstract Domains**

As explained in Section 5.3.4, we can configure abstract domains for JavaScript values by configuring those for $\text{IR}_{\text{ES}}$ values. In JavaScript static analysis, researchers have presented diverse string domains to precisely analyze object property names. Among them, we implemented three representative string abstract domains [13]: the *String Set* ($\mathsf{SS}_k$) domain, the *Character Inclusion* ($\mathsf{CI}$) domain, and the *Prefix-Suffix* ($\mathsf{PS}$) domain. Table 5.3 summarizes formal definitions of their elements, concretization functions, and concatenation operations. In the table, $\Sigma$ denotes a set of characters, and the set of strings is $\mathbb{V}_{\mathsf{str}} = \Sigma^*$. We analyzed a JavaScript program in Figure 5.11 using $\mathsf{JSA}_{\mathsf{ES12}}$ with different string abstract domains. The program uses a new language feature introduced in ES6 called a *template literal*, which is a literal delimited with backticks (\`), allowing embedded expressions called *substitutions*. For example, the template literal `‘c${x}d‘` on line 2 concatenates a string `"c"`, the value in the variable `x`, and a string `"d"`. Since `x` points to `"a"` or `"b"` on line 1, the variable `y` points to `"cad"` or `"cbd"`. Similarly, `z` points to `"aea"` or `"beb"` by concatenating `x`, `"e"`, and `x`.

First, the *String Set* ($\mathsf{SS}_k$) domain represents a set of strings whose size is bounded by $k$ as an abstract string. Therefore, $\mathsf{JSA}_{\mathsf{ES12}}$ with $\mathsf{SS}_5$ produced the following analysis results:

$$\begin{aligned}
\mathsf{x} &\mapsto \{\texttt{"a"}, \texttt{"b"}\} \\
\mathsf{y} &\mapsto \{\texttt{"c"}\} \cdot \{\texttt{"a"}, \texttt{"b"}\} \cdot \{\texttt{"d"}\} = \{\texttt{"cad"}, \texttt{"cbd"}\} \\
\mathsf{z} &\mapsto \{\texttt{"a"}, \texttt{"b"}\} \cdot \{\texttt{"e"}\} \cdot \{\texttt{"a"}, \texttt{"b"}\} = \{\texttt{"aea"}, \texttt{"aeb"}, \texttt{"bea"}, \texttt{"beb"}\}
\end{aligned}$$

It produced precise analysis results for `x` and `y`. However, the result for `z` has spurious values `"aeb"` and `"bea"` because it does not keep the information that the left and right strings of `"e"` are the same.

The *Character Inclusion* ($\mathsf{CI}$) domain tracks the lower and upper bounds of characters occurring in strings. The analysis with this domain produced the following analysis results:

$$\begin{aligned}
\mathsf{x} &\mapsto [\varnothing, \{\mathsf{a}, \mathsf{b}\}] \\
\mathsf{y} &\mapsto [\{\mathsf{c}\}, \{\mathsf{c}\}] \cdot [\varnothing, \{\mathsf{a}, \mathsf{b}\}] \cdot [\{\mathsf{d}\}, \{\mathsf{d}\}] = [\{\mathsf{c}, \mathsf{d}\}, \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}] \\
\mathsf{z} &\mapsto [\varnothing, \{\mathsf{a}, \mathsf{b}\}] \cdot [\{\mathsf{e}\}, \{\mathsf{e}\}] \cdot [\varnothing, \{\mathsf{a}, \mathsf{b}\}] = [\{\mathsf{e}\}, \{\mathsf{a}, \mathsf{b}, \mathsf{e}\}]
\end{aligned}$$

This domain ignores structures of strings, but it is a cheap abstract domain to check only the inclusion of characters in strings. For example, it can say that the string in `y` always includes `c` and `d`, and the string in `z` always includes `e`.

The last one is the *Prefix-Suffix* ($\mathsf{PS}$) domain, which keeps prefixes and suffixes of strings. $\mathsf{JSA}_{\mathsf{ES12}}$ produced the following analysis results with $\mathsf{PS}$:

$$\begin{aligned}
\mathsf{x} &\mapsto \langle \texttt{""}, \texttt{""} \rangle \\
\mathsf{y} &\mapsto \langle \texttt{"c"}, \texttt{"c"} \rangle \cdot \langle \texttt{""}, \texttt{""} \rangle \cdot \langle \texttt{"d"}, \texttt{"d"} \rangle = \langle \texttt{"c"}, \texttt{"d"} \rangle \\
\mathsf{z} &\mapsto \langle \texttt{""}, \texttt{""} \rangle \cdot \langle \texttt{"e"}, \texttt{"e"} \rangle \cdot \langle \texttt{""}, \texttt{""} \rangle = \langle \texttt{""}, \texttt{""} \rangle
\end{aligned}$$

This domain is also cheap but focuses on prefixes and suffixes. Therefore, the analysis results cannot say anything about `x` or `z`, but it describes that the string in `y` starts with `"c"` and ends with `"d"`. Therefore, we showed that one can freely configure string abstract domains for JavaScript in $\mathsf{JSA}_{\mathsf{ES12}}$.

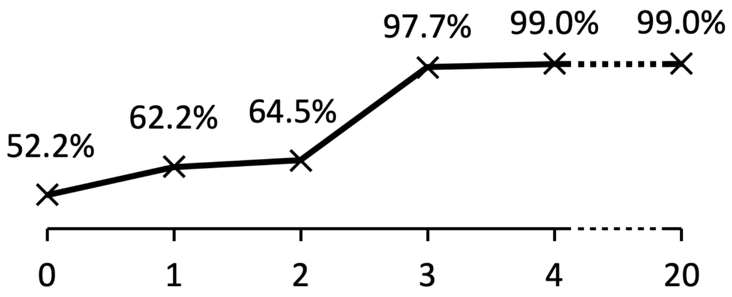


Figure 5.12: The analysis precision of $\mathsf{JSA}_{\mathsf{ES12}}$ with different $k$-callsite-sensitivities for all 18,556 applicable test programs

**Analysis Sensitivities**

As explained in Section 5.3.5, we formally define the flow- and $k$-callsite-sensitivity for JavaScript using the AST-sensitivity for $\mathrm{IR}_{\mathrm{ES}}$. In $\mathsf{JSA}_{\mathsf{ES12}}$, we can freely configure the value $k$ of the $k$-callsite-sensitivity. In Section 5.5.2, we showed that $\mathsf{JSA}_{\mathsf{ES12}}$ with the 20-callsite-sensitivity can precisely analyze 18,556 applicable test programs in Test262 with a high analysis precision of 99.0%. Now, we analyze the test programs with different $k$-callsite-sensitivities to understand how different $k$ values affect the analysis results of $\mathsf{JSA}_{\mathsf{ES12}}$. We started from the context-insensitive analysis ($k = 0$) and increased $k$ of the $k$-callsite-sensitivity until their analysis precision is similar to that of the 20-callsite-sensitivity as depicted in Figure 5.12. As expected, the context-insensitive analysis has the lowest analysis precision of 52.2%. Then, the analysis precision consistently increases with a higher $k$ value, and it reaches 99.0% when $k = 4$.

Therefore, we showed that one can configure the analysis precision of $\mathsf{JSA}_{\mathsf{ES12}}$ by using different $k$-callsite-sensitivities for JavaScript.

## 5.5.4 Adaptability

We evaluated the adaptability of JSAVER using two case studies with new language features. TC39 maintains proposals for future language features in GitHub repositories. In the order of the most GitHub stars, the top three features are the pipeline operator `|>`[3] with 5.8K stars, the pattern matching[4] with 4.0K stars, and the `Observable` library[5] with 2.8K stars. Because the pattern matching proposal is in an early stage with only basic concepts without any detailed semantics, we evaluated the adaptability of JSAVER with two proposals for the pipeline operator `|>` and the `Observable` library.

**Pipeline Operator (`|>`)**

The *pipeline operator* is a language feature typically supported in functional programming languages, such as F# and OCaml. Its behavior is almost the same with a syntactic sugar of a function call with a single argument. To support this operator, we first applied its proposal, which contains the syntactic production in Figure 5.13(a) and algorithms, to ES12. Then, we derived a JavaScript static analyzer from the updated ES12 via JSAVER. Finally, we analyzed the example JavaScript program in Figure 5.13(b) with the interval domain for integers using the derived analyzer.

[3]`https://github.com/tc39/proposal-pipeline-operator`
[4]`https://github.com/tc39/proposal-pattern-matching`
[5]`https://github.com/tc39/proposal-observable`

*PipelineExpression* **:** *PipelineExpression* **|>** *LogicalORExpression*

(a) Syntactic production for the pipeline operator

```
1 let add    = y => x => x + y;
2 let double = z => z * 2;
3 let n = /* any integer from 0 to 99 */;
4 let a = n |> add(1)   // [1, 100]
5           |> double;  // [2, 200]
6 let b = n |> add(1n)  // TypeError for '+'
7           |> unknown; // unreachable
```

(b) A JavaScript program using the pipeline operator

Figure 5.13: Syntax and use of the pipeline operator `|>`

First, the derived analyzer successfully analyzes the stored value in the variable `a`. The program defines two functions: `add` receives a value in `y` and adds it to the second argument in `x`, and `double` multiplies the argument `z` by `2`. The analyzer first analyzes that the variable `n` points to the interval `[0, 99]` on line 3. Then, the abstract value is updated to `[1, 100]` and `[2, 200]` by analyzing `|> add(1)` on line 4 and `|> double` on line 5, respectively. Therefore, the derived analyzer successfully analyzes that the variable `a` stores the interval `[2, 200]`.

Moreover, the derived analyzer correctly analyzes the execution order of the pipeline operator on lines 6–7. Assume that we treat the pipeline operator as just a syntactic sugar of a function call and replace the right-hand-side expression of the assignment on lines 6–7 with `unknown(add(1n)(n))`. Then, the identifier `unknown` is executed first because function parts are executed earlier than argument parts in function call expressions. Thus, it throws a `ReferenceError` exception by trying to read the variable `unknown`. However, because the pipeline operator first executes the argument part rather than the function part, the original program throws a `TypeError` exception on line 6 because the addition of the BigInt value `1n` with another numeric value is ill-typed. The derived analyzer successfully analyzes that the program terminates on line 6 with a `TypeError` exception by correctly considering the execution order of the pipeline operator.

### `Observable` Library

JSAVER can support not only a new syntactic feature but also a new built-in library. Using the `Observable` library, we can model push-based data sources, such as DOM events, timer intervals, and sockets. Consider an example program in Figure 5.14. On lines 1–2, the program first randomly defines variables `x` with `1` or `2` and `y` with a random string. Then, it *registers* an arrow function `subscriber => { ... }` to a new `Observable` object and assigns it to the variable `o` on lines 3–7. On line 8, it *subscribes* `k => x *= k` via `subscribe` to invoke the registered arrow function. Then, the arrow function `k => x *= k` is synchronously invoked three times with multiple values `1`, `2`, and `3`. Therefore, the variable `x` points to `6` or `12` because the initial value of `x` is `1` or `2`, and it is multiplied by `1`, `2`, and `3`. Similarly, the variable `y` points to any string ending with `"123"` because its initial value is a random string, and it is updated by concatenating string values of `1`, `2`, and `3`, on line 9.

To analyze the example program, we applied the proposal of the `Observable` library to ES12 and derived a JavaScript static analyzer from it. We used the interval domain for integers and the *Prefix-Suffix* (PS) domain explained in Section 5.5.3 for strings. On lines 1–2, the derived analyzer first

```
1 let x = /* 1 or 2 */;
2 let y = /* any string */;
3 let o = new Observable(subscriber => {
4   subscriber.next(1);
5   subscriber.next(2);
6   subscriber.next(3);
7 });
8 o.subscribe(k => x *= k); // x: 6 or 12
9 o.subscribe(k => y += k); // y: any string + "123"
```

Figure 5.14: An example of the `Observable` built-in library

assigns `[1, 2]` and $\langle$`""`, `""`$\rangle$ to the variables `x` and `y`, respectively. Then, it assigns the new abstract `Observable` object with the arrow function `subscriber => { ... }` to `o` by analyzing the invocation of the constructor of `Observable` on lines 3–7. On line 8, the analyzer analyzes that an arrow function `k => x *= k` is subscribed, and the variable `x` is updated to the interval `[6, 12]`. Similarly, it analyzes that another arrow function `k => y += k` is subscribed on line 9, and the variable `y` is updated to the abstract value $\langle$`""`, `"123"`$\rangle$. Thus, the derived analyzer successfully analyzes the example program and precisely represents the possible values of `x` and `y` at the end of the program.

# Chapter 6. Related Work

In this chapter, we explain related work of our techniques to automatically derive JavaScript static analyzers from given language specifications.

## 6.1 Mechanized Specification Extraction

**JavaScript Mechanized Specification** For JavaScript, researchers have proposed various mechanized specifications. In 2010, Guha et al. [36] presented an extended lambda calculus $\lambda_{\text{JS}}$, the first mechanized specification for a core calculus of the ES3 semantics by desugaring most of the syntax into $\lambda_{\text{JS}}$. The $\lambda_{\text{JS}}$ semantics looks quite different from the original algorithm steps. In the mid-2010s, researchers tried to define mechanized specifications for the ES5 and 5.1 semantics using similarly looking semantics with the original algorithm steps. Bodin et al. [16] defined JSCert, semantics of a small subset of ES5, using Coq and extracted a reference interpreter JSRef. Park et al. [70] defined an entire semantics of ES5, KJS, using the K [84] framework. Fragoso Santos et al. [31] presented JaVerT, a JavaScript verification toolchain, including JavaScript semantics defined in their own intermediate language, JSIL. However, all of them are manually defined; thus, they require much human effort for the annually evolving specification.

**Mechanized Specification Extraction** To resolve the fundamental problem of manual approaches, researchers in diverse fields presented approaches to automatically extract mechanized specifications from specifications written in natural languages. For system architectures, researchers utilize complex Natural Language Processing (NLP) and Machine Learning (ML) to extract mechanized specifications of small-sized low-level assembly languages, x86 [65] and ARM [97]. For Java API functions, Zhai et al. [108] presented a technique to automatically generate their models by synthesizing Java code from Javadoc comments for API functions. Using NLP techniques and heuristic methods, it produces candidate code and removes wrong ones by testing them with actual implementation. Unlike their approach, we introduce JISET, which extracts JavaScript mechanized specifications from ECMAScript using general compile rules that represent common writing patterns in specifications without using complex NLP or ML techniques. JISET is the first tool that automatically extracts mechanized specifications for a real-world high-level programming language, JavaScript. Moreover, the mechanized specifications extracted via JISET are also executable, which bridge gaps between the specification written in a natural language and executable tests.

**Parser Generation** Moreover, to the best of our knowledge, JISET is even the first tool to automatically generate JavaScript parsers from given versions of ECMAScript. We utilize *Parsing Expression Grammar (PEG)* [30] to generate parsers from given JavaScript syntax written in $\text{BNF}_{\text{ES}}$. From Packrat parsing [29] with PEG, recursive-descent parsers with backtracking support linear-time parsing. However, it has the fundamental problem of ordered choices: $ab$ is silently unmatched with $a$ / $ab$. While Generalized LL (GLL) parsing [87] is basically recursive-descent with backtracking that can support general context-free grammars even in the presence of ambiguous grammars, its worst-time complexity is $O(n^3)$ for the input size $n$, and it does not support context-sensitive features. Unlike GLL parsing, our lookahead parsing is applicable for JavaScript parsers with context-sensitive features such as positive/negative lookaheads. Moreover, the complexity of lookahead parsing is $O(k \cdot n)$ for the constant number of tokens $k$. We experimentally showed that it could generate parsers for the most recent four versions of ECMAScript.

## 6.2 Specification Validity Check

**Specification-based Testing** Recently, researchers have utilized specifications to test their implementations. For network protocols, Kim et al. [49] proposed a novel approach named BaseSpec, which extracts message structures from tables in cellular specifications for L3 protocols to perform a comparative analysis of baseband software. Schumi and Sun [86] presented SpecTest, which utilized an executable language semantics to perform fuzzing for Java and Solidity compilers. For JavaScript, Ye et al. [107] presented Comfort, a compiler fuzzing framework to detect JavaScript engine bugs using ECMAScript with deep learning-based language models. We presented JEST, which performs $N+1$-version differential testing with $N$ different JavaScript engines and a reference interpreter extracted from ECMAScript. JEST detects not only engine bugs but also specification bugs in ECMAScript using the cross-referencing oracle. Moreover, to lessen the burden of synthesizing conformance tests using multiple JavaScript engines in dozens of hours, we presented JSTAR to detect specification bugs without JavaScript engines in several minutes. Because JSTAR uses abstract semantics while JEST uses concrete semantics, JSTAR can quickly analyze more scope of semantics than JEST.

**Differential Testing** Differential testing [59] utilizes multiple implementations as cross-referencing oracles to find semantics bugs. Researchers applied this technique to various applications domain such as Java Virtual Machine (JVM) implementations [21], SSL/TLS certification validation logic [22, 33, 81], web applications [18], and binary lifters [50]. Moreover, Nezha [81] introduces a guided differential testing tool with the concept of $\delta$-diversity to efficiently find semantics bugs. However, they have a fundamental limitation that they cannot test specifications; they use only cross-referencing oracles and target potential bugs in implementations. Our $N+1$-version differential testing extends the idea of differential testing with not only $N$ different implementations but also a mechanized specification to test both of them. In addition, our approach automatically generates conformance tests directly from the specification.

**Fuzzing** Fuzzing is a software testing technique for detecting security vulnerabilities by generating [37, 42, 106] or mutating [17, 82, 104] test inputs. For JavaScript [101] engines, Godefroid et al. [35] presented white-box fuzzing using the JavaScript grammar, Han et al. [38] presented CodeAlchemist that generates JavaScript code snippets based on semantics-aware assembly, Wang et al. [98] presented Superion using Grammar-aware grey-box fuzzing, Park et al. [79] presented Die using aspect-preserving mutation, and Lee et al. [57] presented Montage using neural network language models (NNLMs). While they focus on finding security vulnerabilities rather than semantics bugs, our $N+1$-version differential testing focuses on finding semantics bugs by comparing multiple implementations with the mechanized specification, which was automatically extracted from ECMAScript by JISET. Note that JEST can also localize not only specification bugs in ECMAScript but also bugs in JavaScript engines indirectly using the bug locations in ECMAScript.

**Fault Localization** To localize detected bugs in ECMAScript, we used Spectrum Based Fault Localization (SBFL) [102], which is a ranking technique based on the likelihood of being faulty for each program element. Tarantula [46, 47] was the first tool that supports SBFL with a simple formula, and researchers have developed many formulae [27, 44, 64, 103] to increase the accuracy of bug localization. Sohn and Yoo [91] introduced a novel approach for fault localization using code and change metrics via learning of SBFL formulae. While we utilize a specific formula $ER1_b$ introduced by Xie et al. [105], we believe that it is possible to improve the accuracy of bug localization by using more advanced SBFL techniques.

## 6.3 Derivation of Static Analyzers

**JavaScript Analysis Tools** ECMAScript is the standard language specification for JavaScript maintained by TC39. In late 2014, the committee announced its plan to release ECMAScript annually and adopt the open development process to quickly adapt to evolving development environments. Various JavaScript engines such as Google V8, GraalJS, QuickJS, and Moddable XS should conform to the syntax and semantics described in annually updated ECMAScript. Beyond JavaScript engines, diverse research projects use JavaScript specifications. The main research direction has been static analyzers such as SAFE [56, 77], TAJS [45], WALA [92], and JSAI [48], based on the abstract interpretation framework [23, 25] with their own analysis techniques. They defined abstract semantics of the JavaScript semantics described in ECMAScript to statically analyze JavaScript programs in a finite time. Charguéraud et al. [19] presented JSExplain, a debugger for JavaScript, by implementing a reference interpreter in OCaml following the algorithm steps in ECMAScript closely. For a given JavaScript program, the debugger interactively produces execution traces investigated in a browser, with an interface that displays the JavaScript code and the interpreter's state. Fragoso Santos et al. [31] introduced JaVerT, a JavaScript verification toolchain, based on the separation logic with an intermediate goto language JSIL. JaVerT 2.0 [32] extends it to support compositional symbolic execution for JavaScript based on bi-abduction. However, because all of them manually handle ECMAScript with their own intermediate representations, most of them still target ES5.1 released in 2011 instead of the latest one.

**JavaScript Static Analysis Techniques** Various JavaScript static analysis techniques have been presented and implemented. Since string values of arbitrary expressions could be used in property accesses, a precise string analysis is much more important in JavaScript than in static analysis for other programming languages. Thus, several advanced string abstract domains are presented for JavaScript using hash values [58], state automata representation [51], and regular expressions [67]. Moreover, Amadini et al. [13] presented a framework to combine such advanced string abstract domains freely. Another challenging problem in JavaScript static analysis is imprecise relations between object properties. Researchers presented diverse techniques to resolve this problem with a correlation tracking for read/write pairs [92], a loop sensitivity [68, 69], a syntactic pattern-based trace partitioning [53, 54], a demand-driven value refinement [93], and a value partitioning [66]. Due to the highly dynamic nature of JavaScript, static analyzers suffer not only from imprecise analysis results but also from heavy computations. Thus, combined analyses [74, 76, 78, 85, 100] with dynamic analyses have been proposed to enhance the performance of analysis by leveraging highly optimized commercial JavaScript engines. JSAVER allows configuring abstract domains and analysis sensitivities freely for JavaScript. We believe that JSAVER will help researchers focus on designing novel analysis techniques without concerning about defining and implementing abstract semantics for the newly introduced language features.

**Abstracting Definitional Interpreter** Reynolds [83] first introduced the concept of definitional interpreters to describe the semantics of defined-languages using their interpreters written in defining-languages. Researchers have used them to describe the semantics of higher-order programming languages in which functions or labels are values. Darais et al. [28] extended them to a definitional abstract interpreter, representing the abstract semantics of a defined-language using its abstract interpreter written in a defining-language. However, unlike a meta-level static analysis, it directly describes the abstract semantics of the defined-language without using a static analyzer of the defining-language. Therefore, it still requires manual updates when the defined-language evolves. For the JavaScript programming language, Herman and Flanagan [39] proposed the first definitional interpreter written in ML to represent the JavaScript semantics. Then, Bodin et al. [16] manually defined the JavaScript semantics in JSCert using the Coq proof assistant and extracted a definitional interpreter from Coq to OCaml. It even inspired developers to design various definitional interpreters, such as Narcissus [6] and engine262 [3]. However, they require manual updates when JavaScript evolves. On the other hand, JISET automatically extracts a JavaScript definitional interpreter from ECMAScript. Because JISET provides a way to deal with the JavaScript semantics mechanically, we developed JSAVER by extending JISET to automatically derive a static analyzer via a meta-level static analysis for JavaScript.

**Automatic Modeling for JavaScript Built-in Libraries** For JavaScript static analysis, modeling behaviors of functions in built-in libraries is essential because they are implemented in different programming languages instead of JavaScript, such as C++ in the V8 JavaScript engine. Therefore, researchers have presented techniques to automatically model their behaviors using types in documentation [15, 71], program syntheses [40], and concrete executions [72]. However, because all of them are unsound approaches, they cannot cover all the possible behaviors of functions in built-in libraries. Moreover, their qualities highly rely on given documentation or JavaScript engines. On the other hand, JSAVER does not need to model their behaviors because they are described in the algorithms in ECMAScript and compiled directly to $\text{IR}_{\text{ES}}$ functions via JISET. Therefore, JSAVER can analyze the functions in built-in libraries without any modeling, and we also showed that it can analyze not yet introduced built-in libraries.

# Chapter 7. Conclusion

The fast evolution and massive size of JavaScript specifications make it difficult to develop and update JavaScript static analyzers manually. To alleviate this problem, we have presented a novel approach to automatically derive JavaScript static analyzers from language specifications with three steps: 1) mechanized specification extraction using JISET, 2) specification validity check using JEST and JSTAR, and 3) derivation of static analyzers using JSAVER.

JISET is the first tool that *automatically* extracts a mechanized specification from ECMAScript, a standard JavaScript specification written in English. The extracted mechanized specification consists of a JavaScript parser for syntax and functions of $\text{IR}_{\text{ES}}$, a specialized intermediate representation we defined for ECMAScript, for semantics. Moreover, it supports extracting IR-based semantics from ECMAScript by synthesizing AST-IR translators based on the compiled $\text{IR}_{\text{ES}}$ functions. The tool automatically extracts all the syntax and 95.03% of the semantics for the most recent four versions of ECMAScript (ES7 to ES10). We evaluated the correctness of the tool by testing the extracted semantics from ES10 with Test262, the official conformance suite. Using 1,709 failed tests, we found nine specification errors, four of which are newly discovered, confirmed by TC39, and planned to be integrated into ES11. After fixing the errors, the extracted semantics passed all 18,064 applicable tests in Test262. We also showed that JISET is adaptable to nine proposals for new language features to be included in ES11, which let us find three errors in the BigInt proposal. We believe that JISET can dramatically reduce human efforts in building various JavaScript tools correctly.

JEST is a tool that performs *$N$+1-version differential testing* for JavaScript using a specific version of ECMAScript and four modern JavaScript engines and to check the validity of both ECMAScript and engines. The development of modern programming languages follows the continuous integration (CI) and continuous deployment (CD) approach to instantly support fast-changing user demands. Such continuous development makes it difficult to find semantics bugs in both the language specification and its various implementations. To alleviate this problem, we present $N$+1-version differential testing, which is the first technique to test both implementations and its specification in tandem. We actualized our approach for the JavaScript programming language via JEST. It automatically generated 1,700 JavaScript programs with 97.78% of syntax coverage and 87.70% of semantics coverage on ES11. JEST injected assertions to the generated JavaScript programs to convert them as conformance tests. We executed generated conformance tests on four engines that support ES11: V8, GraalJS, QuickJS, and Moddable XS. Using the execution results, we found 44 engine bugs (16 for GraalJS, 6 for QuickJS, 20 for Moddable XS, and 2 for V8) and 27 specification bugs. All the bugs were confirmed by TC39, the committee of ECMAScript, and the corresponding engine teams, and they will be fixed in the specification and the engines. We believe that JEST takes the first step towards the co-evolution of software specifications, tests, and their implementations for CI/CD.

JSTAR is another tool to check the validity of JavaScript specifications by performing type analysis for them without leveraging JavaScript engines. Checking the correctness of ECMAScript is essential because an incorrect description in ECMAScript can lead to wrong implementations of JavaScript engines. However, since ECMAScript is annually released and developed in an open process, checking its correctness becomes more labor-intensive and error-prone. To alleviate the problem, we propose JSTAR that performs *type analysis* on JavaScript specifications and detects specification bugs using a *bug de-*

*tector*. The main challenge of ECMAScript type analysis to statically detect type-related specification bugs automatically is that ECMAScript describes abstract algorithms in a natural language, English. We first compile abstract algorithms to $\text{IR}_{\text{ES}}$ functions via JISET and define abstract semantics with specification types on top of them. We also present *condition-based refinement* for type analysis, which prunes out infeasible abstract states using conditions of assertions and branches to improve the analysis precision. We evaluated JSTAR with all 864 versions in the official ECMAScript repository for the last three years from 2018 to 2021. It took 137.3 seconds on average to perform type analysis for each version and detected 157 type-related specification bugs with 59.2% precision; 93 out of 157 reported bugs are true bugs. Among them, 14 bugs are newly detected by JSTAR, and the committee confirmed them all.

JSAVER is the first tool that automatically derives JavaScript static analyzers from any versions of ECMAScript. The main idea of JSAVER is to shift the paradigm from *compiler*-based approaches to *interpreter*-based approaches to fully utilize "the interpreter-based nature" of JavaScript. It performs a *meta-level static analysis* to indirectly analyze JavaScript programs using a definitional interpreter extracted from ECMAScript. We also present how to configure *abstract domains* and *analysis sensitivities* for JavaScript indirectly in the meta-level static analysis. We evaluated JSAVER by using a derived static analyzer $\text{JSA}_{\text{ES12}}$ from the latest ECMAScript, ES12. It soundly analyzes all applicable 18,556 official conformance tests with 99.0% of precision in 1.59 seconds on average. We also demonstrated the configurability and adaptability of JSAVER with several case studies. We believe that JSAVER can reduce the burden of defining the abstract semantics of numerous language features for static analysis of fast-evolving JavaScript.

# Bibliography

[1] "Mark Zuckerberg's Letter to Investors: 'The Hacker Way'." 2012. `https://www.wired.com/2012/02/zuck-letter/`.

[2] "ECMAScript 12th edition." 2021. `https://262.ecma-international.org/12.0/`.

[3] "engine262: An Implementation of ECMAScript in JavaScript." 2021. `https://github.com/engine262/engine262`.

[4] "GraalJS: A JavaScript Engine Built on GraalVM." 2021. `https://github.com/graalvm/graaljs`.

[5] "Moddable XS: A JavaScript Engine in Moddable SDK." 2021. `https://github.com/Moddable-OpenSource/moddable`.

[6] "Narcissus: A JavaScript Interpreter written in pure JavaScript developed by Mozilla." 2021. `https://github.com/mozilla/narcissus`.

[7] "Official Solidity Documentation." 2021. `https://docs.soliditylang.org/en/v0.8.7/`.

[8] "QuickJS: A Javascript Engine by Fabrice Bellard." 2021. `https://bellard.org/quickjs/`.

[9] "V8: A JavaScript and WebAssembly Engine by Google." 2021. `https://v8.dev/`.

[10] "What is CI/CD? Continuous Integration and Continuous Delivery Explained." 2021. `https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html`.

[11] P. W. Abrahams. "A final solution to the dangling else of ALGOL 60 and related languages." *Communications of the ACM (CACM)*, 9(9):679–682, 1966. `https://doi.org/10.1145/365813.365821`.

[12] A. V. Aho and J. D. Ullman. "The Theory of Parsing." *Translation and Compiling*, 1(1972), 1973.

[13] R. Amadini, A. Jordan, G. Gange, F. Gauthier, P. Schachte, H. Søndergaard, P. J. Stuckey, and C. Zhang. "Combining String Abstract Domains for JavaScript Analysis: An Evaluation." In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2017. `https://doi.org/10.1007/978-3-662-54577-5_3`.

[14] S. An, J. Park, and S. Ryu. "IR$_{\text{ES}}$: Intermediate Representation for ECMAScript Specifications." Technical report, 2020. `https://bit.ly/3CtLQh9`.

[15] S. Bae, H. Cho, I. Lim, and S. Ryu. "SAFEWAPI: Web API Misuse Detector for Web Applications." In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 2014. `https://doi.org/10.1145/2635868.2635916`.

[16] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. "A Trusted Mechanised JavaScript Specification." *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming (POPL)*, 49(1):87–100, 2014. `https://doi.org/10.1145/2535838.2535876`.

[17] S. K. Cha, M. Woo, and D. Brumley. "Program-Adaptive Mutational Fuzzing." In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 725–741. IEEE, 2015.

[18] P. Chapman and D. Evans. "Automated Black-box Detection of Side-Channel Vulnerabilities in Web Applications." In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pages 263–274. 2011. https://doi.org/10.1145/2046707.2046737.

[19] A. Charguéraud, A. Schmitt, and T. Wood. "JSExplain: A Double Debugger for JavaScript." In *Companion Proceedings of the The Web Conference (WWW)*. 2018. https://doi.org/10.1145/3184558.3185969.

[20] D. R. Chase, M. Wegman, and F. K. Zadeck. "Analysis of Pointers and Structures." In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI)*, PLDI '90, page 296–310. Association for Computing Machinery, New York, NY, USA, 1990. ISBN 0897913647. https://doi.org/10.1145/93542.93585.

[21] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. "Coverage-directed differential testing of JVM implementations." In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 85–99. 2016. https://doi.org/10.1145/2908080.2908095.

[22] Y. Chen and Z. Su. "Guided Differential Testing of Certificate Validation in SSL/TLS Implementations." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 793–804. 2015. https://doi.org/10.1145/2786805.2786835.

[23] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 238–252. 1977. https://doi.org/10.1145/512950.512973.

[24] P. Cousot and R. Cousot. "Static Determination of Dynamic Properties of Generalized Type Unions." In *Proceedings of an ACM Conference on Language Design for Reliable Software*, page 77–94. Association for Computing Machinery, New York, NY, USA, 1977. ISBN 9781450373807. https://doi.org/10.1145/800022.808314.

[25] P. Cousot and R. Cousot. "Abstract Interpretation Frameworks." *Journal of Logic and Computation*, 2(4):511–547, 1992. https://doi.org/10.1093/logcom/2.4.511.

[26] A. Cox, B.-Y. E. Chang, and X. Rival. "Automatic Analysis of Open Objects in Dynamic Language Programs." In *Proceedings of the 21st International Symposium on Static Analysis (SAS)*. 2014. https://doi.org/10.1007/978-3-319-10936-7_9.

[27] V. Dallmeier, C. Lindig, and A. Zeller. "Lightweight Bug Localization with AMPLE." In *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging (AADEBUG)*, pages 99–104. 2005. https://doi.org/10.1145/1085130.1085143.

[28] D. Darais, N. Labich, P. C. Nguyen, and D. Van Horn. "Abstracting Definitional Interpreters (Functional Pearl)." In *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2017. https://doi.org/10.1145/3110256.

[29] B. Ford. "Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl." In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 2002. `https://doi.org/10.1145/581478.581483`.

[30] B. Ford. "Parsing Expression Grammars: A Recognition-based Syntactic Foundation." In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2004. `https://doi.org/10.1145/964001.964011`.

[31] J. Fragoso Santos, P. Maksimović, D. Naudžiūnienė, T. Wood, and P. Gardner. "JaVerT: JavaScript Verification Toolchain." In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2017. `https://doi.org/10.1145/3158138`.

[32] J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner. "JaVerT 2.0: Compositional Symbolic Execution for JavaScript." In *Proceedings of the 46th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2019. `https://doi.org/10.1145/3290379`.

[33] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. "The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software." In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, pages 38–49. 2012. `https://doi.org/10.1145/2382196.2382204`.

[34] D. Glaze and D. Van Horn. "Abstracting Abstract Control." In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS)*. 2014. `https://doi.org/10.1145/2661088.2661098`.

[35] P. Godefroid, A. Kiezun, and M. Y. Levin. "Grammar-based Whitebox Fuzzing." In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 206–215. 2008. `https://doi.org/10.1145/1375581.1375607`.

[36] A. Guha, C. Saftoiu, and S. Krishnamurthi. "The essence of JavaScript." In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP)*. 2010. `https://doi.org/10.1007/978-3-642-14107-2_7`.

[37] H. Han and S. K. Cha. "IMF: Inferred Model-based Fuzzer." In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2345–2358. 2017. `https://doi.org/10.1145/3133956.3134103`.

[38] H. Han, D. Oh, and S. K. Cha. "CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines." In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2019.

[39] D. Herman and C. Flanagan. "Status Report: Specifying Javascript with ML." In *Proceedings of the 2007 Workshop on Workshop on ML*. 2007. `https://doi.org/10.1145/1292535.1292543`.

[40] S. Heule, M. Sridharan, and S. Chandra. "Mimic: Computing Models for Opaque Code." In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2015. `https://doi.org/10.1145/2786805.2786875`.

[41] J. L. Hintze and R. D. Nelson. "Violin Plots: A Box Plot-Density Trace Synergism." *The American Statistician*, 52(2):181–184, 1998.

[42] C. Holler, K. Herzig, and A. Zeller. “Fuzzing with Code Fragments.” In *Presented of the 21st USENIX Conference on Security Symposium (Security)*. ????

[43] S. Hwang and S. Ryu. “Gap between Theory and Practice: An Empirical Study of Security Patches in Solidity.” In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. 2020.

[44] T. Janssen, R. Abreu, and A. J. van Gemund. “Zoltar: A Toolset for Automatic Fault Localization.” In *Proceedings of 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 662–664. IEEE, 2009.

[45] S. H. Jensen, A. Møller, and P. Thiemann. “Type Analysis for JavaScript.” In *Proceedings of the 16th International Symposium on Static Analysis (SAS)*. 2009. `https://doi.org/10.1007/978-3-642-03237-0_17`.

[46] J. A. Jones, M. J. Harrold, and J. Stasko. “Visualization of Test Information to Assist Fault Localization.” In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 467–477. IEEE, 2002. `https://doi.org/10.1145/581339.581397`.

[47] J. A. Jones, M. J. Harrold, and J. T. Stasko. “Visualization for fault localization.” In *Proceedings of ICSE Workshop on Software Visualization*. Citeseer, 2001.

[48] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. “JSAI: A Static Analysis Platform for JavaScript.” In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 2014. `https://doi.org/10.1145/2635868.2635904`.

[49] E. Kim, D. Kim, C. Park, I. Yun, and Y. Kim. “BASESPEC: Comparative Analysis of Baseband Software and Cellular Specifications for L3 Protocols.” In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*. 2021. `https://doi.org/10.14722/ndss.2021.24365`.

[50] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha. “Testing Intermediate Representations for Binary Analysis.” In *Proceedings of ACM International Conference on Automated Software Engineering (ASE)*, pages 353–364. 2017. `https://doi.org/10.1109/ASE.2017.8115648`.

[51] S.-W. Kim, W. Chin, J. Park, J. Kim, and S. Ryu. “Inferring Grammatical Summaries of String Values.” In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 372–391. Springer, 2014. `https://doi.org/10.1007/978-3-319-12736-1_20`.

[52] S.-W. Kim, X. Rival, and S. Ryu. “A Theoretical Foundation of Sensitivity in an Abstract Interpretation Framework.” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(3), 2018. ISSN 0164-0925. `https://doi.org/10.1145/3230624`.

[53] Y. Ko, X. Rival, and S. Ryu. “Weakly Sensitive Analysis for Unbounded Iteration over JavaScript Objects.” In *Proceedings of the 15th Asian Symposium on Programming Languages and Systems (APLAS)*. 2017. `https://doi.org/10.1007/978-3-319-71237-6_8`.

[54] Y. Ko, X. Rival, and S. Ryu. "Weakly Sensitive Analysis for JavaScript Object-Manipulating Programs." *Software: Practice and Experience (SPE)*, 49(5):840–884, 2019. `https://doi.org/10.1002/spe.2676`.

[55] C. Lattner, A. Lenharth, and V. Adve. "Making Context-Sensitive Points-to Analysis with Heap Cloning Practical for the Real World." In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 278–289. 2007. `https://doi.org/10.1145/1250734.1250766`.

[56] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu. "SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript." In *Proceedings of 19th International Workshop on Foundations of Object-Oriented Languages (FOOL)*. 2012.

[57] S. Lee, H. Han, S. K. Cha, and S. Son. "Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer." 2020.

[58] M. Madsen and E. Andreasen. "String Analysis for Dynamic Field Access." In *Proceedings of the 23rd International Conference on Compiler Construction (CC)*. 2014. `https://doi.org/10.1007/978-3-642-54807-9_12`.

[59] W. M. McKeeman. "Differential Testing for Software." *Digital Technical Journal*, 10(1):100–107, 1998.

[60] B. McKenzie. "Generating Strings at Random from a Context Free Grammar." Technical Report TR-COSC 10/97, Department of Computer Science, University of Canterbury, 1997.

[61] M. Might and O. Shivers. "Improving Flow Analyses via ΓCFA: Abstract Garbage Collection and Counting." In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 13–25. 2006. `https://doi.org/10.1145/1159803.1159807`.

[62] A. Milanova, A. Rountev, and B. G. Ryder. "Parameterized Object Sensitivity for Points-to Analysis for Java." *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005. ISSN 1049-331X. `https://doi.org/10.1145/1044834.1044835`.

[63] A. Moors, F. Piessens, and M. Odersky. "Parser combinators in Scala." *CW Reports*, 54, 2008.

[64] L. Naish, H. J. Lee, and K. Ramamohanarao. "A Model for Spectra-based Software Diagnosis." *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):1–32, 2011. `https://doi.org/10.1145/2000791.2000795`.

[65] H. Nguyen. "Automatic Extraction of x86 Formal Semantics from its Natural Language Description." *Information Science*, 2018.

[66] B. B. Nielsen and A. Møller. "Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript." In *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP)*. 2020. `https://doi.org/10.4230/LIPIcs.ECOOP.2020.16`.

[67] C. Park, H. Im, and S. Ryu. "Precise and Scalable Static Analysis of jQuery using a Regular Expression Domain." In *Proceedings of the 12th Symposium on Dynamic Languages (DLS)*. 2016. `https://doi.org/10.1145/2989225.2989228`.

[68] C. Park, H. Lee, and S. Ryu. "Static Analysis of JavaScript Libraries in a Scalable and Precise Way using Loop Sensitivity." *Software: Practice and Experience (SPE)*, 48(4):911–944, 2018. https://doi.org/10.1002/spe.2676.

[69] C. Park and S. Ryu. "Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity." In *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP)*. 2015. https://doi.org/10.4230/LIPIcs.ECOOP.2015.735.

[70] D. Park, A. Stefănescu, and G. Roşu. "KJS: A Complete Formal Semantics of JavaScript." In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2015. https://doi.org/10.1145/2737924.2737991.

[71] J. Park. "JavaScript API misuse detection by using typescript." In *Proceedings of the companion publication of the 13th international conference on Modularity*. 2014. https://doi.org/10.1145/2584469.2584472.

[72] J. Park, A. Jordan, and S. Ryu. "Automatic Modeling of Opaque Code for JavaScript Static Analysis." In *Proceedings of the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE)*. 2019. https://doi.org/10.1007/978-3-030-16722-6_3.

[73] J. Park, H. Lee, and S. Ryu. "A Survey of Parametric Static Analysis." *ACM Computing Surveys (CSUR)*, 54(7):1–37, 2021. https://doi.org/10.1145/3464457.

[74] J. Park, I. Lim, and S. Ryu. "Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild." In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*. 2016. https://doi.org/10.1145/2889160.2889227.

[75] J. Park, J. Park, S. An, and S. Ryu. "JISET: Javascript IR-based Semantics Extraction Toolchain." In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2020. https://doi.org/10.1145/3324884.3416632.

[76] J. Park, J. Park, D. Youn, and S. Ryu. "Accelerating JavaScript Static Analysis via Dynamic Shortcuts." In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2021. https://doi.org/10.1145/3468264.3468556.

[77] J. Park, Y. Ryou, J. Park, and S. Ryu. "Analysis of JavaScript Web Applications Using SAFE 2.0." In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*. 2017. https://doi.org/10.1109/ICSE-C.2017.4.

[78] J. Park, K. Sun, and S. Ryu. "EventHandler-Based Analysis Framework for Web Apps Using Dynamically Collected States." In *Proceedings of the 21st International Conference on Fundamental Approaches to Software Engineering (FASE)*. 2018. https://doi.org/10.1007/978-3-319-89363-1_8.

[79] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim. "Fuzzing JavaScript Engines with Aspect-preserving Mutation." In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642. IEEE, 2020. https://doi.org/10.1109/SP40000.2020.00067.

[80] T. Parr, S. Harwell, and K. Fisher. “Adaptive LL(*) Parsing: The Power of Dynamic Analysis.” In *Proceedings of the 29th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* 2014. https://doi.org/10.1145/2660193.2660202.

[81] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. “NEZHA: Efficient Domain-Independent Differential Testing.” In *Proceedings of IEEE Symposium on Security and Privacy (SP)*, pages 615–632. 2017. https://doi.org/10.1109/SP.2017.27.

[82] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. “Optimizing Seed Selection for Fuzzing.” In *Proceedings of the 23rd USENIX Conference on Security Symposium (Security)*, pages 861–875. 2014. https://doi.org/10.5555/2671225.2671280.

[83] J. C. Reynolds. “Definitional Interpreters for Higher-Order Programming Languages.” In *Proceedings of the ACM Annual Conference - Volume 2.* 1972. https://doi.org/10.1145/800194.805852.

[84] G. Roșu and T. F. Șerbănută. “An Overview of the K Semantic Framework.” *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. https://doi.org/10.1016/j.jlap.2010.03.012.

[85] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. “Dynamic Determinacy Analysis.” In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI).* 2013. https://doi.org/10.1145/2499370.2462168.

[86] R. Schumi and J. Sun. “SpecTest: Specification-Based Compiler Testing.” In *Proceedings of the 24th International Conference on Fundamental Approaches to Software Engineering (FASE).* 2021. https://doi.org/10.1007/978-3-030-71500-7_14.

[87] E. Scott and A. Johnstone. “GLL Parsing.” In *Electronic Notes in Theoretical Computer Science.* 2010. https://doi.org/10.1016/j.entcs.2010.08.041.

[88] M. Sharir and A. Pnueli. *Two Approaches to Interprocedural Data Flow Analysis.* 1981.

[89] O. G. Shivers. *Control-Flow Analysis of Higher-Order Languages.* Carnegie Mellon University, 1991.

[90] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. “Pick Your Contexts Well: Understanding Object-Sensitivity.” In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, page 17–30. Association for Computing Machinery, New York, NY, USA, 2011. ISBN 9781450304900. https://doi.org/10.1145/1926385.1926390.

[91] J. Sohn and S. Yoo. “Fluccs: Using Code and Change Metrics to Improve Fault Localization.” In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 273–283. ACM, 2017. https://doi.org/10.1145/3092703.3092717.

[92] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. “Correlation Tracking for Points-To Analysis of JavaScript.” In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP).* 2012. https://doi.org/10.1007/978-3-642-31057-7_20.

[93] B. Stein, B. B. Nielsen, B.-Y. E. Chang, and A. Møller. "Static Analysis with Demand-Driven Value Refinement." In *Proceedings of the 34th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2019. `https://doi.org/10.1145/3360566`.

[94] M. Tomita. "An Efficient Context-Free Parsing Algorithm for Natural Languages." In *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, pages 756–764. 1985. `https://doi.org/10.5555/1623611.1623625`.

[95] D. Van Horn and M. Might. "Abstracting Abstract Machines." In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (POPL)*. 2010. `https://doi.org/10.1145/1863543.1863553`.

[96] D. Van Horn and M. Might. "Abstracting Abstract Machines: A Systematic Approach to Higher-Order Program Analysis." *Communications of the ACM (CACM)t*, 54(9):101–109, 2011. `https://doi.org/10.1145/1995376.1995400`.

[97] A. V. Vu and M. Ogawa. "Formal Semantics Extraction from Natural Language Specifications for ARM." In *International Symposium on Formal Methods (FM)*. 2019. `https://doi.org/10.1007/978-3-030-30942-8_28`.

[98] J. Wang, B. Chen, L. Wei, and Y. Liu. "Superion: Grammar-Aware Greybox Fuzzing." In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019. `https://doi.org/10.1109/ICSE.2019.00081`.

[99] A. Warth, J. R. Douglass, and T. Millstein. "Packrat Parsers Can Support Left Recursion." In *Proceedings of ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation (PEPM)*. 2008. `https://doi.org/10.1145/1328408.1328424`.

[100] S. Wei and B. G. Ryder. "Practical Blended Taint Analysis for JavaScript." In *Proceedings of the 22th International Symposium on Software Testing and Analysis (ISSTA)*. 2013. `https://doi.org/10.1145/2483760.2483788`.

[101] A. Wirfs-Brock and B. Eich. "JavaScript: the first 20 years." In *Proceedings of the ACM on Programming Languages*, volume 4, pages 1–189. 2020. `https://doi.org/10.1145/3386327`.

[102] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. "A Survey on Software Fault Localization." *IEEE Transactions on Software Engineering (TSE)*, 42(8):707–740, 2016. `https://doi.org/10.1109/TSE.2016.2521368`.

[103] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai. "Effective Fault Localization using Code Coverage." In *Proceedings of 31st Annual International Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 449–456. IEEE, 2007. `https://doi.org/10.1109/COMPSAC.2007.109`.

[104] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. "Scheduling Black-Box Mutational Fuzzing." In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 511–522. 2013. `https://doi.org/10.1145/2508859.2516736`.

[105] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu. "A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization." *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):1–40, 2013. `https://doi.org/10.1145/2522920.2522924`.

[106] X. Yang, Y. Chen, E. Eide, and J. Regehr. "Finding and Understanding Bugs in C Compilers." In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294. 2011. `https://doi.org/10.1145/1993498.1993532`.

[107] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang. "Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing." In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2021. `https://doi.org/10.1145/3453483.3454054`.

[108] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin. "Automatic Model Generation from Documentation for Java API Functions." In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. 2016. `https://doi.org/10.1145/2884781.2884881`.

# Acknowledgments in Korean

2016년부터 6년 동안의 제 20대 인생의 희로애락을 같이 했던 박사과정이 이렇게 끝이 나고, 새로운 세상을 향해 나아갈 수 있게 되었습니다. 박사과정동안 방황도 많이 했지만 길다면 길고 짧다면 짧은 기간 동안 수많은 사람들의 도움이 받아 성공적으로 이 여정을 마칠 수 있었습니다.

이 여정에 있어 당연하게도 가장 큰 도움을 주셨고, 언제나 저의 든든한 동료이자 인생의 선배로서 이끌어주신 류석영 교수님께 감사하다는 말씀을 전하고 싶습니다. 교수님께서 하시는 "프로그래밍 언어 이론" 수업을 학부 시절 수강 후, 흥미를 느껴 연구실에 들어오면서 시작하게 된 교수님과의 인연은 이렇게 박사과정까지 이어졌습니다. 연구를 하면서 겪었던 시행착오들과 실패들을 교수님께서는 항상 곁에서 지지를 해주셨고, 이상한 생각이라고 치부해버릴 수 있었던 저의 아이디어들을 흥미롭게 들어주시며 같이 발전시켜주신 덕분에 이렇게 긴 여정이 결실을 맺을 수 있었던 것 같습니다. 교수님의 긍정적인 생각과 학생들을 생각해주시는 마음과 더불어 날카로운 시선과 생각하게 만드는 질문들로부터 어떻게 연구를 해야 하는가를 배울 수 있었습니다. 저를 믿어주시고 항상 응원해주셔서 감사합니다, 류석영 교수님.

연구실에 머문 기간 동안, 수많은 연구실 동료들은 저에게 좋은 토론 상대이자, 친구이자, 인생의 선배였습니다. 박창희, 이성호, 황성재 박사님은 제가 아직 서투르고 좋은 연구자란 무엇일까에 대한 고민을 하고 있을 때, 앞서 연구자의 길을 걷고 있는 연구실 선배로서 수많은 조언을 해주셨고, 그 덕에 올바른 방향으로 나아갈 수 있었던 것 같습니다. 박준영 박사님은 같이 박사과정을 밟는 동안 가장 밀접한 연구를 진행했던 동료로서, 서로의 연구에 대해서 심도 깊은 토론을 하면서 연구의 부족한 점을 서로 보완하기도 하고, 서로의 연구를 지지해주었습니다. 안승민 님은 수많은 연구를 같이 진행했던 공동저자로서 수준 높은 구현을 통해 연구를 탄탄하게 만들어주고, 다양한 연구의 초석을 다지는 역할을 해주었습니다. 그 외에도 톡톡 튀는 아이디어를 가지고 토론을 해주던 박지희 님, 비판적인 시각과 날카로운 시선으로 연구에 대한 조언을 해준 홍재민 님 등에게 감사의 말씀을 드립니다.

이 외에도 외부에서 저에게 도움을 준 수많은 분들이 계십니다. Xavier Rival 교수님은 저에게 수식을 통해 사람들과 소통하는 방법을 알려주셨습니다. 프로그램 분석에 있어 다양한 분석 기술들을 수식적으로 표현하고, 머릿속에 들어있는 아이디어를 수식을 통해 표현할 수 있는지를 알려주었습니다. 이는 저의 연구에 있어 중요한 하나의 초석이 되었고, 연구자로서 성장할 수 있는 하나의 밑거름이 되었습니다. 유신 교수님은 소프트웨어 공학 분야에서 저의 아이디어가 어떻게 받아들여질지에 대한 조언을 다양하게 해주셨고, 이는 제 연구가 결실을 맺는데 중요한 역할을 해주었습니다. 또한, 연구 외적으로 힘든 점들과 고민들을 들어주었던 저의 정신적 지주인 친구 성재호 님에게도 감사하다고 전하고 싶습니다.

마지막으로, 박사 과정 도중 힘들 때마다 저에게 가장 큰 도움이 되었던 가족들에게 감사드린다는 말씀을 전하고 싶습니다. 어머니께서는 항상 제가 아프지는 않은지 고되지는 않은지 걱정하시며 멀리서나마 저를 응원해주셨습니다. 아버지께서는 다 잘 될 것이라는 희망적인 격려를 통해 긍정적인 생각을 가질 수 있도록 도와주셨습니다. 형은 잘 표현하지는 않더라도 항상 뒤에서 묵묵히 저를 응원해주었습니다. 이러한 가족들의 따뜻한 격려와 지지가 없었더라면 박사과정을 마치기 어려웠을 것입니다. 이 박사과정학위를 수여하는 영광을 가족들에게 돌리고 싶습니다.

## Curriculum Vitae in Korean

이　　　름: 박 지 혁

생 년 월 일: 1993년 02월 23일

### 학　　력

| | |
|---|---|
| 2012. 3. – 2016. 2. | 한국과학기술원 전산학부 (학사) |
| 2016. 3. – 2021. 2. | 한국과학기술원 전산학부 (M.S. & Ph.D.) |

### 경　　력

| | |
|---|---|
| 2016. 3. – 2016. 6. | 한국과학기술원 전산학부 프로그래밍 실습 수업 조교 |
| 2016. 9. – 2019.12. | 한국과학기술원 전산학부 프로그래밍 언어 수업 조교 |
| 2017. 9. – 2017.12. | 한국과학기술원 전산학부 전산학특강<검색 기반 소프트웨어 공학> 조교 |
| 2018. 9. – 2018.12. | 한국과학기술원 전산학부 전산학 프로젝트 수업 조교 |

### 연 구 업 적

1. **Jihyeok Park**, Seungmin An, Wonho Shin, Yusung Sim, and Sukyoung Ryu, "JSTAR: JavaScript Specification Type Analyzer using Refinement," In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, November 2021.
2. Joonyoung Park*, **Jihyeok Park***, Dongjun Youn, and Sukyoung Ryu (*equally contributed), "Accelerating JavaScript Static Analysis via Dynamic Shortcuts," In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, August 2021. `https://dl.acm.org/doi/10.1145/3468264.3468556`
3. **Jihyeok Park**, "JavaScript Static Analysis with Evolving Engines and Specification," In *Proceedings of the 35th European Conference on Object-Oriented Programming and Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis Doctoral Symposium track*, July 2021.
4. **Jihyeok Park***, Hongki Lee*, and Sukyoung Ryu (*equally contributed), "A Survey of Parametric Static Analysis," In *ACM Computing Surveys*, Volume 54, Issue 7, September 2022. `https://doi.org/10.1145/3464457`
5. **Jihyeok Park**, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu, "JEST: N+1-Version Differential Testing of Both JavaScript Engines and Specification," In *Proceedings of 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021. `https://dl.acm.org/doi/10.1109/ICSE43902.2021.00015`

6. **Jihyeok Park**, Jihee Park, Seungmin An, and Sukyoung Ryu, "JISET: JavaScript IR-based Semantics Extraction Toolchain," In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, December 2020. `https://dl.acm.org/doi/10.1145/3324884.3416632`
7. Sukyoung Ryu, **Jihyeok Park**, and Joonyoung Park, "Toward Analysis and Bug Finding in JavaScript Web Applications in the Wild," In *IEEE Software*, Volume: 36, Issue: 3, May-June 2019. `https://doi.org/10.1109/MS.2018.110113408`.
8. Jaemin Hong, **Jihyeok Park**, and Sukyoung Ryu, "Path Dependent Types with Path-Equality," In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, September 2018. `https://doi.org/10.1145/3241653.3241657`.
9. **Jihyeok Park**, Joonyoung Park, Yoonkyong Lee, Chul-Joo Kim, Byoungoh Kim, and Sukyoung Ryu, "A Framework for Dynamic Inter-Device Task Dispatch with Eventual Consistency," In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming)*, April 2018. `https://doi.org/10.1145/3191697.3191732`.
10. Keunhong Lee, Shinae Woo, Sanghyeon Seo, **Jihyeok Park**, Sukyoung Ryu, and Sue Moon, "Toward Building Memory-safe Network Functions with Modest Performance Overhead," In *Proceedings of the 3rd SIGCOMM Workshop on Networking and Programming Languages (NetPL)*, August 2017.
11. **Jihyeok Park**, Xavier Rival and Sukyoung Ryu, "Revisiting Recency Abstraction for JavaScript: Towards an Intuitive, Compositional, and Efficient Heap Abstraction," In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program (SOAP)*, June 2017. **Best Paper Award**. `https://doi.org/10.1145/3088515.3088516`.
12. **Jihyeok Park**, Yeonhee Ryou, Joonyoung Park, and Sukyoung Ryu, "Analysis of JavaScript Web Applications Using SAFE 2.0," In *Proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017. `https://doi.org/10.1109/ICSE-C.2017.4`
13. **Jihyeok Park**, "JavaScript API Misuse Detection by Using TypeScript," In *Proceedings of the companion publication of the 13th international conference on Modularity*, April 2014. **ACM Student Research Competition 3rd**. `https://doi.org/10.1145/2584469.2584472`.