# JSTAR: JavaScript Specification Type Analyzer using Refinement

Jihyeok Park
*School of Computing*
*KAIST*
Daejeon, South Korea
jhpark0223@kaist.ac.kr

Seungmin An
*School of Computing*
*KAIST*
Daejeon, South Korea
h2oche@kaist.ac.kr

Wonho Shin
*School of Computing*
*KAIST*
Daejeon, South Korea
new170527@kaist.ac.kr

Yusung Sim
*School of Computing*
*KAIST*
Daejeon, South Korea
yusungsim@kaist.ac.kr

Sukyoung Ryu
*School of Computing*
*KAIST*
Daejeon, South Korea
sryu.cs@kaist.ac.kr

***Abstract*—JavaScript is one of the mainstream programming languages for client-side programming, server-side programming, and even embedded systems. Various JavaScript engines developed and maintained in diverse fields must conform to the syntax and semantics described in ECMAScript, the standard specification of JavaScript. Since an incorrect description in ECMAScript can lead to wrong JavaScript engine implementations, checking the correctness of ECMAScript is critical and essential. However, all the specification updates are currently manually reviewed by the Ecma Technical Committee 39 (TC39) without any automated tools. Moreover, in late 2014, the committee announced the yearly release cadence and open development process of ECMAScript to quickly adapt to evolving development environments. Because of such frequent updates, checking the correctness of ECMAScript becomes more labor-intensive and error-prone.**

**To alleviate the problem, we propose JSTAR, a JavaScript Specification Type Analyzer using Refinement. It is the first tool that performs *type analysis* on JavaScript specifications and detects specification bugs using a *bug detector*. For a given specification, JSTAR first compiles each abstract algorithm written in a structured natural language to a corresponding function in $\text{IR}_{\text{ES}}$, an untyped intermediate representation for ECMAScript. Then, it performs type analysis for compiled functions with specification types defined in ECMAScript. Based on the result of type analysis, JSTAR detects specification bugs using a bug detector consisting of four checkers. To increase the precision of the type analysis, we present *condition-based refinement* for type analysis, which prunes out infeasible abstract states using conditions of assertions and branches. We evaluated JSTAR with all 864 versions in the official ECMAScript repository for the recent three years from 2018 to 2021. JSTAR 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 bugs are true bugs. Among them, 14 bugs are newly detected by JSTAR, and the committee confirmed them all.**

***Index Terms*—JavaScript, mechanized specification, type analysis, refinement, bug detection**

## I. Introduction

JavaScript is one of the most popular programming languages. According to the 2020 State of the Octoverse[1], the annual report of GitHub, the most dominating programming language in GitHub repositories was JavaScript since 2014 to 2020. While JavaScript was initially designed for client-side programming in web browsers, it is now widely used in server-side programming [1] and even in embedded systems [2]–[4]. Developers in diverse fields build and maintain JavaScript engines conforming to ECMAScript, the JavaScript standard specification, which describes the syntax and semantics of JavaScript in a natural language.

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 the Ecma Technical Committee 39 (TC39) without any automated tools. Such a manual review process is inherently labor-intensive and error-prone, making ECMAScript vulnerable to specification bugs. Besides, in late 2014, the committee announced the yearly release cadence and open development process of ECMAScript to quickly adapt to evolving development environments. According to Park et al. [5], the average number of updated steps of abstract algorithms between consecutive releases from ECMAScript 2016 (ES7) to 2019 (ES10) is 9645.5. In the official ECMAScript repository, 1,355 pull requests and 2,005 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, “Assert: Type($O$) is Object” denotes that the variable $O$ always has an `Object` value at the point of the assertion, and “**? GetV**($V$, $P$)” 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[2] such as JavaScript engines [4], [6]–[8], debuggers [9], static analyzers [10]–[13], and verification tools [14], [15]. Second, the committee has started internal discussions on type annotations for variables, parameters, and return values of abstract

[1] https://octoverse.github.com/

[2] https://github.com/tc39/ecmarkup/issues/173

algorithms[3]. However, any kinds of manual annotations are labor-intensive and error-prone, and they do not provide any automatic mechanism to detect specification bugs.

To alleviate this problem, we propose a novel tool JSTAR, a **J**avaScript **S**pecification **T**ype **A**nalyzer using **R**efinement. 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]–[18] 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 [19], [20], network protocols [21], and language specifications [22], [23] have utilized information directly extracted from specifications written in a natural language to lessen such burdens. Among them, JISET [5] compiles ECMAScript abstract algorithms written in a structured natural language to functions in $\text{IR}_{\text{ES}}$, an untyped intermediate representation for ECMAScript. Therefore, JSTAR leverages JISET to mechanically handle JavaScript specifications.

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 2018 to 2021. The experiments showed that the refinement technique could reduce the number of false-positive bugs caused by spurious types inferred by imprecise type analysis.

The main contributions of this paper are as follows:

- 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 takes 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.

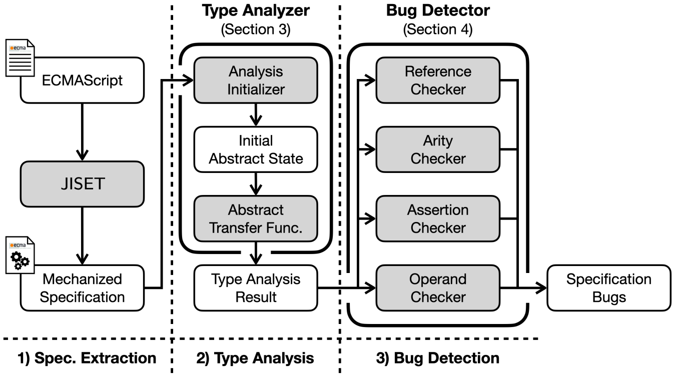


Fig. 1: JSTAR: a type analyzer and a bug detector for mechanized specifications extracted from ECMAScript by JISET

## II. Overview

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

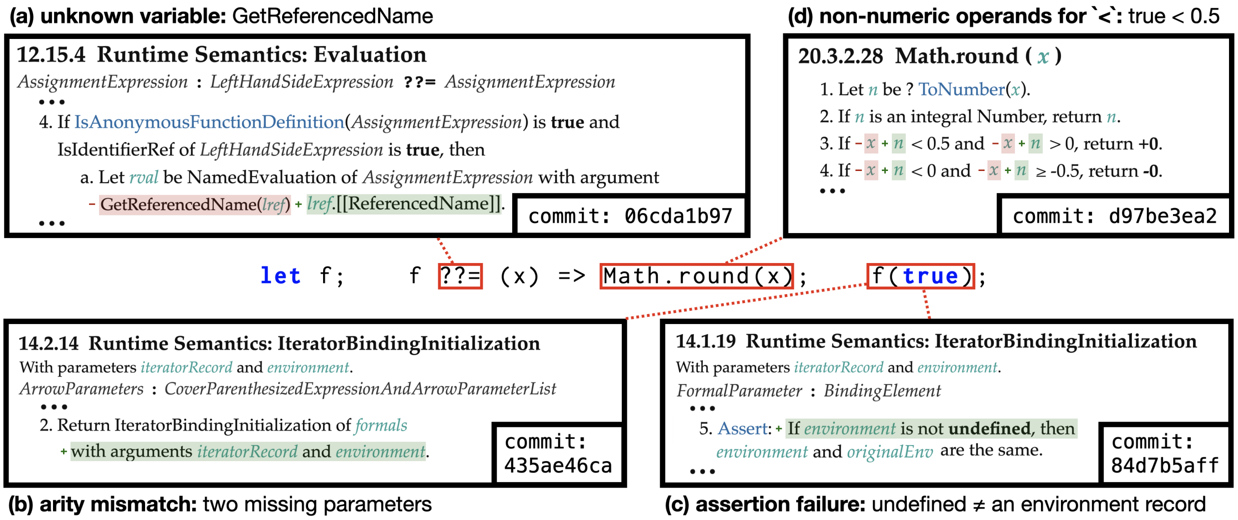
### A. Specification Extraction

JSTAR extracts the JavaScript syntax and semantics using JISET and extracts specification types from ECMAScript.

*a) 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 III-A.

*b) 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 the empty list type `[]` and parametric list types `[`$\tau$`]`.

[3] https://github.com/tc39/ecma262/pull/545#issuecomment-559292107



**(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

Fig. 2: An example JavaScript program with related previous specification bugs and their bug fixes

### B. 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*.

*a) 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.

*b) 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 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 III-C.

### C. 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 2.

*a) 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 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 7 days, which the reference checker can detect.

*b) Arity Checker:* The program finally calls `f` with an argument **`true`**. During the initialization of the function call, **IteratorBindingInitialization** in Figure 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. 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.

*c) Assertion Checker:* During the initialization of the function call, **IteratorBindingInitialization** of *FormalParameter* in Figure 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.

*d) Operand Checker:* After the function call initialization, the parameter `x` has the value **`true`**, and `Math.round` in Figure 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

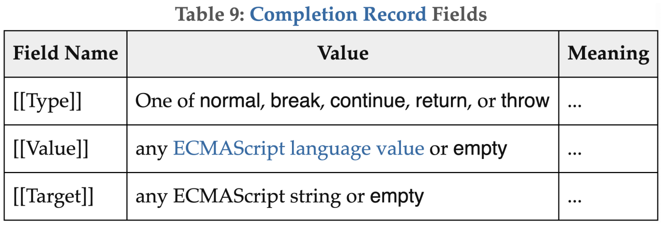


Table 9: Completion Record Fields

| 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 | ... |

Fig. 3: Fields of completion records in ECMAScript 2020

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 paper, we explain the details of how to perform type analysis for $\text{IR}_{\text{ES}}$ functions and how to increase the analysis precision using the condition-based refinement (Section III) and how to detect type-related specification bugs (Section IV). After we evaluate JSTAR (Section V), we discuss related work (Section VI) and conclude (Section VII).

## III. 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.

### A. Intermediate Representation

$\text{IR}_{\text{ES}}$ is an untyped intermediate representation for ECMAScript [5]. We modify it as a label-based language to make it suitable for type analysis as follows:

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

A modified $\text{IR}_{\text{ES}}$ program $P = (\texttt{func}, \texttt{inst}, \texttt{next})$ consists of three mappings; $\texttt{func} : \mathbb{L} \rightarrow \mathbb{F}$ maps labels to their functions, $\texttt{inst} : \mathbb{L} \rightarrow \mathbb{I}$ maps labels to their instructions, and $\texttt{next} : \mathbb{L} \rightarrow \mathbb{L}$ maps labels to their next labels, where a label $\ell \in \mathbb{L}$ denotes a program point. A function $\texttt{def f(x}^*\texttt{, [y}^*\texttt{])}\,\ell \in \mathbb{F}$ consists of its name $\texttt{f}$, normal parameters $\texttt{x}^*$, optional parameters $\texttt{y}^*$, and a body label $\ell$. For presentation brevity, we assume that no global variables exist in this paper. 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 $\texttt{next}$. A reference $r$ is a variable $\texttt{x}$ or a field access $r\texttt{[}e\texttt{]}$. We write $r\texttt{.f}$ to briefly represent $r\texttt{["f"]}$. 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 $\texttt{undefined}$, $\texttt{null}$, a $\texttt{Boolean}$ $b$, a $\texttt{Number}$ $n$, a $\texttt{BigInt}$ $j$, a $\texttt{String}$ $s$, or a $\texttt{Symbol}$ $\texttt{@}s$.

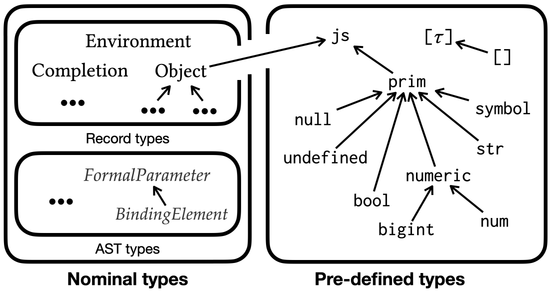


Fig. 4: A graphical representation of the subtype relation $<:$

A type $\tau \in \mathbb{T}$ is either a nominal type $t$, an empty list type $\texttt{[]}$, a parametric list type $\texttt{[}\tau\texttt{]}$, a JavaScript type $\texttt{js}$, a primitive type $\texttt{prim}$, a numeric type $\texttt{numeric}$, $\texttt{num}$, $\texttt{bigint}$, $\texttt{str}$, or $\texttt{symbol}$. 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 3 shows an excerpt from ECMAScript 2020 (ES11) that describes the fields of completion records[4], which we model as follows:

$$
\begin{array}{l}
\texttt{Completion} = \{ \\
\quad \texttt{Type} : \{c_{\texttt{normal}}, c_{\texttt{break}}, c_{\texttt{continue}}, c_{\texttt{return}}, c_{\texttt{throw}}\}, \\
\quad \texttt{Value} : \{\texttt{js}, c_{\texttt{empty}}\}, \quad \texttt{Target} : \{\texttt{str}, c_{\texttt{empty}}\} \\
\}
\end{array}
$$

The subtype relation $<:\subseteq \mathbb{T} \times \mathbb{T}$ between types is depicted in Figure 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 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 $\tau' <: \tau$.

We define a denotational semantics of the modified $\text{IR}_{\text{ES}}$ for instructions $[\![i]\!]_i : \mathbb{S} \rightarrow \mathbb{S}$, references $[\![r]\!]_r : \mathbb{S} \rightarrow \mathbb{S} \times \mathbb{V}$, and expressions $[\![e]\!]_e : \mathbb{S} \rightarrow \mathbb{S} \times \mathbb{V}$ where $\mathbb{S}$ and $\mathbb{V}$ denote states and values, respectively. For brevity, we omit it in this paper and refer the interested readers to a companion report [24].

### B. Type Analysis

We design a type analysis for the modified $\text{IR}_{\text{ES}}$ based on the abstract interpretation framework [25], [26] with analysis sensitivity [27]. We extend types as follows:

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

with function types $f$, constant types $c$, and the absent type $\texttt{?}$ representing the absence of variables. $\texttt{Boolean}$ values $b$, $\texttt{String}$ values $s$, $\texttt{normal}(\tau)$ denoting normal completions

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

$$
\begin{array}{l}
\llbracket \texttt{let x} = e \rrbracket_i^\sharp(\ell, \overline{\tau})(d^\sharp) \quad = (\{(\texttt{next}(\ell), \overline{\tau}) \mapsto \sigma^\sharp[\texttt{x} \mapsto \llbracket e \rrbracket_e^\sharp(\sigma^\sharp)]\}, \varnothing) \\
\llbracket \texttt{x} = \texttt{(}e\ e_1 \cdots e_n\texttt{)} \rrbracket_i^\sharp(\ell, \overline{\tau})(d^\sharp) = (m', r') \\
\text{where } \left\{ \begin{array}{l}
\tau^\sharp = \llbracket e \rrbracket_e^\sharp(\sigma^\sharp) \ \wedge\ \tau_1^\sharp = \llbracket e_1 \rrbracket_e^\sharp(\sigma^\sharp) \ \wedge\ \cdots \ \wedge\ \tau_n^\sharp = \llbracket e_n \rrbracket_e^\sharp(\sigma^\sharp) \ \wedge \\
T' = \{\dot{\texttt{up}}([\tau_1, \cdots, \tau_n]) \mid \tau_1 \in \tau_1^\sharp \ \wedge\ \cdots \ \wedge\ \tau_n \in \tau_n^\sharp\} \ \wedge \\
f = \texttt{def f(}p_1, \cdots, \texttt{[}\cdots, \texttt{p}_{k_f}\texttt{])}\, \ell_f \ \wedge \\
\sigma_{f,\overline{\tau}'}^\sharp = [\texttt{p}_1 \mapsto \{\overline{\tau}'[1]\}, \cdots, \texttt{p}_{k_f} \mapsto \{\overline{\tau}'[k_f]\}] \ \wedge \\
m' = \{(\ell_f, \overline{\tau}') \mapsto \sigma_{f,\overline{\tau}'}^\sharp \mid f \in \tau^\sharp \ \wedge\ \overline{\tau}' \in T'\} \ \wedge \\
r' = \{(f, \overline{\tau}') \mapsto \{(\texttt{next}(\ell), \overline{\tau}, \texttt{x})\} \mid f \in \tau^\sharp \ \wedge\ \overline{\tau}' \in T'\}
\end{array} \right.
\end{array}
$$

$$
\begin{array}{ll}
\llbracket \texttt{return}\ e \rrbracket_i^\sharp(\ell, \overline{\tau})(d^\sharp) & = (\{(\ell_r, \overline{\tau}_r) \mapsto \sigma_r^\sharp \mid (\ell_r, \overline{\tau}_r, \texttt{x}) \in R\}, \varnothing) \\
\multicolumn{2}{l}{\text{where } R = r(\texttt{func}(\ell), \overline{\tau}) \ \wedge\ \sigma_r^\sharp = m(\ell_r, \overline{\tau}_r)[\texttt{x} \mapsto \llbracket e \rrbracket_e^\sharp(\sigma^\sharp)]} \\
\llbracket \texttt{assert}\ e \rrbracket_i^\sharp(\ell, \overline{\tau})(d^\sharp) & = (\{(\texttt{next}(\ell), \overline{\tau}) \mapsto \texttt{pass}(e, \texttt{\#t})(\sigma^\sharp)\}, \varnothing) \\
\llbracket \texttt{if}\ e\ \ell_{\texttt{t}}\ \ell_{\texttt{f}} \rrbracket_i^\sharp(\ell, \overline{\tau})(d^\sharp) & = (\{(\ell_{\texttt{t}}, \overline{\tau}) \mapsto \texttt{pass}(e, \texttt{\#t})(\sigma^\sharp), \\
& \qquad (\ell_{\texttt{f}}, \overline{\tau}) \mapsto \texttt{pass}(e, \texttt{\#f})(\sigma^\sharp)\}, \varnothing) \\
\llbracket \texttt{x} = e \rrbracket_i^\sharp(\ell, \overline{\tau})(d^\sharp) & = (\{(\texttt{next}(\ell), \overline{\tau}) \mapsto \sigma^\sharp[\texttt{x} \mapsto \llbracket e \rrbracket_e^\sharp(\sigma^\sharp)]\}, \varnothing) \\
\llbracket r\texttt{[}e_0\texttt{]} = e_1 \rrbracket_i^\sharp(\ell, \overline{\tau})(d^\sharp) & = (\{(\texttt{next}(\ell), \overline{\tau}) \mapsto \sigma^\sharp\}, \varnothing) \\
\multicolumn{2}{l}{\text{where } d^\sharp = (m, r) \ \wedge\ \sigma^\sharp = m(\ell, \overline{\tau})}
\end{array}
$$

Fig. 5: Abstract semantics of instructions for a program $P = (\texttt{func}, \texttt{inst}, \texttt{next})$, $\llbracket i \rrbracket_i^\sharp : (\mathbb{L} \times \mathbb{T}^*) \to \mathbb{S}^\sharp \to \mathbb{S}^\sharp$

with `Value` fields of type $\tau$, and `abrupt` denoting abrupt completions serve to improve the analysis precision.

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

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

An abstract state $d^\sharp \in \mathbb{S}^\sharp$ 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 $\sigma^\sharp \in \mathbb{E}^\sharp$ represents possible types for variables, and $\sigma^\sharp(\texttt{x}) = \{\texttt{?}\}$ when `x` is not defined in $\sigma^\sharp$. An abstract type $\tau^\sharp \in \mathbb{T}^\sharp$ 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:

$$
\begin{array}{l}
\tau_0^\sharp \sqcup \tau_1^\sharp = \texttt{norm}(\tau_0^\sharp \cup \tau_1^\sharp) \\
\tau_0^\sharp \sqcap \tau_1^\sharp = \texttt{norm}(\{\tau_0 \in \tau_0^\sharp \mid \{\tau_0\} \sqsubseteq \tau_1^\sharp\} \cup \{\tau_1 \in \tau_1^\sharp \mid \{\tau_1\} \sqsubseteq \tau_0^\sharp\}) \\
\tau_0^\sharp \sqsubseteq \tau_1^\sharp \Leftrightarrow \forall \tau_0 \in \tau_0^\sharp.\ \exists \tau_1 \in \texttt{norm}(\tau_1^\sharp).\ \text{s.t. } \tau_0 <: \tau_1
\end{array}
$$

where $\texttt{norm}(\tau^\sharp) = \{\tau \in \tau^\sharp \mid \nexists \tau' \in \tau^\sharp \setminus \{\tau\}.\ \text{s.t. } \tau <: \tau'\}$.

We now define the abstract semantics of instructions $\llbracket i \rrbracket_i^\sharp : (\mathbb{L} \times \mathbb{T}^*) \to \mathbb{S}^\sharp \to \mathbb{S}^\sharp$ in Figure 5 and the abstract semantics of references $\llbracket r \rrbracket_r^\sharp : \mathbb{E}^\sharp \to \mathbb{T}^\sharp$ and expressions $\llbracket e \rrbracket_e^\sharp : \mathbb{E}^\sharp \to \mathbb{T}^\sharp$ in a companion report [24]. To avoid the explosion of type-sensitive views, we upcast the argument type before function calls with the following function:

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

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

$$
\texttt{pass}(e, b)(\sigma^\sharp) = \left\{ \begin{array}{ll}
\texttt{refine}(e, b)(\sigma^\sharp) & \text{if } b \sqsubseteq \llbracket e \rrbracket_e^\sharp(\sigma^\sharp) \\
\varnothing & \text{otherwise}
\end{array} \right.
$$

where `refine` prunes out infeasible parts in abstract environments using their conditions as we describe in Section III-C. Then, we define the abstract semantics $\llbracket P \rrbracket^\sharp$ of a program $P$ as the least fixpoint of the abstract transfer function $F^\sharp : \mathbb{S}^\sharp \to \mathbb{S}^\sharp$:

$$
\begin{array}{l}
\llbracket P \rrbracket^\sharp = \lim_{n \to \infty} (F^\sharp)^n (d_\iota^\sharp) \\
F^\sharp(d^\sharp) = d^\sharp \sqcup \left( \bigsqcup_{(\ell, \overline{\tau}) \in \text{Domain}(m)} \llbracket \texttt{inst}(\ell) \rrbracket_i^\sharp(\ell, \overline{\tau})(d^\sharp) \right)
\end{array}
$$

where $d^\sharp = (m, \_)$ and $d_\iota^\sharp$ denotes the initial abstract state. As described in Section II, $d_\iota^\sharp$ 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 2(a) is initialized with the following abstract environment:

$$
\begin{array}{r}
\texttt{this} \mapsto \{\texttt{AssignmentExpression}\}, \\
\texttt{LeftHandSideExpression} \mapsto \{\texttt{LeftHandSideExpression}\}, \\
\texttt{AssignmentExpression} \mapsto \{\texttt{AssignmentExpression}\}
\end{array}
$$

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 2(d):

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

### *C. Condition-based Refinement*

We present a *condition-based refinement* of the type analysis for the modified IR$_{\text{ES}}$ 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:

TABLE I: 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 |

$$
\begin{aligned}
\texttt{refine}(!e, b)(\sigma^\sharp) &= \texttt{refine}(e, \neg b)(\sigma^\sharp) \\
\texttt{refine}(e_0 \texttt{ || } e_1, b)(\sigma^\sharp) &= \begin{cases} \sigma_0^\sharp \sqcup \sigma_1^\sharp & \text{if } b \\ \sigma_0^\sharp \sqcap \sigma_1^\sharp & \text{if } \neg b \end{cases} \\
\texttt{refine}(e_0 \texttt{ \&\& } e_1, b)(\sigma^\sharp) &= \begin{cases} \sigma_0^\sharp \sqcap \sigma_1^\sharp & \text{if } b \\ \sigma_0^\sharp \sqcup \sigma_1^\sharp & \text{if } \neg b \end{cases} \\
\texttt{refine}(\texttt{x.Type == } c_{\texttt{normal}}, \texttt{\#t})(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \mapsto \tau_{\texttt{x}}^\sharp \sqcap \texttt{normal}(\mathbb{T})] \\
\texttt{refine}(\texttt{x.Type == } c_{\texttt{normal}}, \texttt{\#f})(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \mapsto \tau_{\texttt{x}}^\sharp \sqcap \{\texttt{abrupt}\}] \\
\texttt{refine}(\texttt{x == } e, \texttt{\#t})(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \mapsto \tau_{\texttt{x}}^\sharp \sqcap \tau_e^\sharp] \\
\texttt{refine}(\texttt{x == } e, \texttt{\#f})(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \mapsto \tau_{\texttt{x}}^\sharp \setminus \lfloor \tau_e^\sharp \rfloor] \\
\texttt{refine}(\texttt{x} : \tau, \texttt{\#t})(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \mapsto \tau_{\texttt{x}}^\sharp \sqcap \{\tau\}] \\
\texttt{refine}(\texttt{x} : \tau, \texttt{\#f})(\sigma^\sharp) &= \sigma^\sharp[\texttt{x} \mapsto \tau_{\texttt{x}}^\sharp \setminus \{\tau' \mid \tau' <: \tau\}] \\
\texttt{refine}(e, b)(\sigma^\sharp) &= \sigma^\sharp
\end{aligned}
$$

where $\sigma_j^\sharp = \texttt{refine}(e_j, b)(\sigma^\sharp)$ for $j = 0, 1$, $\tau_e^\sharp = \llbracket e \rrbracket_e^\sharp(\sigma^\sharp)$, and $\lfloor \tau^\sharp \rfloor$ returns $\{\tau\}$ if $\tau^\sharp$ denotes a singleton type $\tau$, or returns $\varnothing$, otherwise.

## IV. Bug Detector

We develop a *bug detector* to statically detect type-related specification bugs in ECMAScript using an augmented abstract transfer function $F_a^\sharp$ 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 I, 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*.

### A. 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:

$$
\begin{aligned}
\llbracket \texttt{x} \rrbracket_e^\sharp(\sigma^\sharp) &= \begin{cases} \text{unknown variable } \texttt{x} & \text{if } \llbracket \texttt{x} \rrbracket_r^\sharp(\sigma^\sharp) = \{\texttt{?}\} \\ \cdots & \text{otherwise} \end{cases} \\
\llbracket \texttt{let x} = e \rrbracket_i^\sharp(\ell, \overline{\tau})(d^\sharp) &= \begin{cases} \text{already defined variable } \texttt{x} & \text{if } \tau^\sharp = \{\texttt{\#t}\} \\ \cdots & \text{otherwise} \end{cases} \\
&\quad \text{where } \tau^\sharp = \llbracket \texttt{x?} \rrbracket_e^\sharp(d^\sharp(\ell, \overline{\tau}))
\end{aligned}
$$

If the abstract semantics of a variable lookup for $\texttt{x}$ is a singleton $\{\texttt{?}\}$, $\texttt{x}$ is always an unknown variable. For example, consider the syntax-directed algorithm in Figure 2(a). Since the **GetReferencedName** algorithm is removed, the variable `GetReferencedName` does not exist in abstract environments and its lookup returns $\{\texttt{?}\}$. 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 $\llbracket \texttt{x?} \rrbracket_e^\sharp$ to see whether the variable $\texttt{x}$ of each variable declaration is already defined.

### B. Arity Checker

The arity of a function $f = \texttt{def f(p}_1, \cdots, \texttt{p}_n, \texttt{[}\cdots, p_m\texttt{])}\,\ell$ 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:

$$
\begin{aligned}
&\llbracket \texttt{x} = \texttt{(}e\ e_1 \cdots e_k\texttt{)} \rrbracket_i^\sharp(\ell, \overline{\tau})(d^\sharp) = \\
&\qquad \begin{cases} \text{missing parameters } \texttt{p}_{k+1}, \cdots, \texttt{p}_{n_f} & \text{if } \exists f \in \tau^\sharp.\ \text{s.t. } k < n_f \\ \cdots & \text{otherwise} \end{cases} \\
&\qquad \text{where } f = \texttt{def f(p}_1, \cdots, \texttt{p}_{n_f}, \texttt{[}\cdots, p_{m_f}\texttt{])}\,\ell \ \land \\
&\qquad\qquad \tau^\sharp = \llbracket e \rrbracket_e^\sharp(d^\sharp(\ell, \overline{\tau})).
\end{aligned}
$$

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 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`.

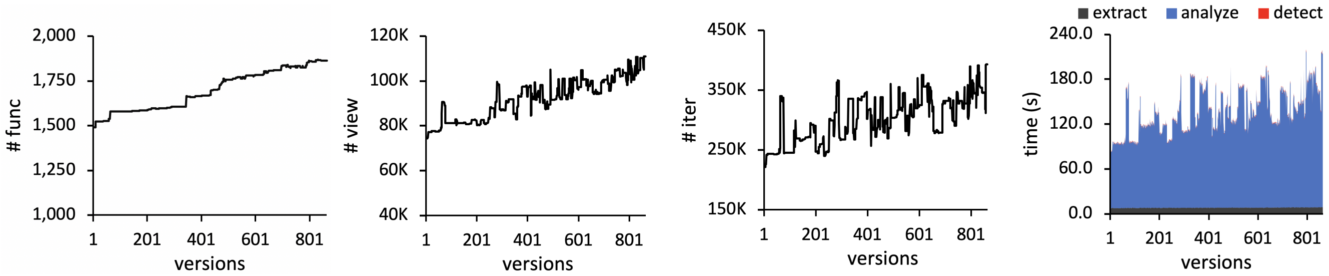
### C. Assertion Checker

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:

$$
\llbracket \texttt{assert } e \rrbracket_i^\sharp(\ell, \overline{\tau})(d^\sharp) = \begin{cases} \text{assertion failure } e & \text{if } \{\texttt{\#t}\} \not\sqsubseteq \tau^\sharp \\ \cdots & \text{otherwise} \end{cases}
$$

$$
\text{where } \tau^\sharp = \llbracket e \rrbracket_e^\sharp(d^\sharp(\ell, \overline{\tau}))
$$

It checks whether the abstract semantics of the condition expression $e$ subsumes $\{\texttt{\#t}\}$. For example, consider the syntax-directed algorithm in Figure 2(c). The parameter `environment` of this algorithm has an environment record or `undefined`. Since type sensitivity divides the abstract types



(a) The number of functions (b) The number of views (c) The number of iterations (d) The analysis time

Fig. 6: The statistics of the type analysis using JSTAR for 864 versions of ECMAScript

of arguments to upcasted single types, there are two different abstract environments whose variable `environment` points to {`Environment`} or {`undefined`}. When `environment` is {`Environment`}, the abstract semantics of the assertion condition `environment = originalEnv` is {`bool`}. Even though we know that the type of `originalEnv` is also {`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 {`undefined`}, the abstract semantics of the condition `environment = originalEnv` is {`#f`} because an environment is never equal to `undefined`. Therefore, the assertion checker reports an assertion failure for the condition `environment = originalEnv`.

### D. 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:

$$
\begin{aligned}
[\![e_0 \oplus e_1]\!]^\sharp_e(\sigma^\sharp) &= \begin{cases} \text{ill-typed operand } e_0 & \text{if } [\![e_0]\!]^\sharp_r(\sigma^\sharp) \not\sqsubseteq \tau^\sharp_0 \\ \text{ill-typed operand } e_1 & \text{if } [\![e_1]\!]^\sharp_r(\sigma^\sharp) \not\sqsubseteq \tau^\sharp_1 \\ \cdots & \text{otherwise} \end{cases} \\
[\![\ominus\, e]\!]^\sharp_e(\sigma^\sharp) &= \begin{cases} \text{ill-typed operand } e & \text{if } [\![e]\!]^\sharp_r(\sigma^\sharp) \not\sqsubseteq \tau^\sharp \\ \cdots & \text{otherwise} \end{cases}
\end{aligned}
$$

where $\tau^\sharp_0$, $\tau^\sharp_1$, and $\tau^\sharp$ 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 2(d). The types of `x` and `n` are {`js`} and {`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 {`num,bigint`} does not subsume {`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 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 ↓ to explicitly represent this conversion:

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

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

## V. Evaluation

We implemented JSTAR as an open-source tool[5] in Scala by extending JISET, a JavaScript IR-based semantics extraction toolchain [5], 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:

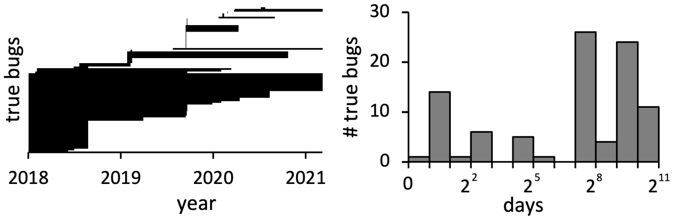
- RQ1. **(Performance)** How long does JSTAR take to perform type analysis for JavaScript specifications?
- RQ2. **(Precision)** How many type-related specification bugs detected by JSTAR are true bugs?
- RQ3. **(Effectiveness of Refinement)** Does the condition-based refinement improve the analysis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

Because the draft of the next ECMAScript (ES12, 2021) is fixed on March 9, 2021, we analyzed all 864 versions in the official ECMAScript repository[6] 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.

[5] The link is anonymized because of a double-blind review process.
[6] https://github.com/tc39/ecma262

TABLE II: 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%) | |



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

Fig. 7: Life spans of true bugs

### A. Performance

Figure 6 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.

### B. 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 II, 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 are 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 7 illustrates the life spans of true bugs; Figure 7(a) depicts the life spans sorted by creation and Figure 7(b) depicts the histogram of the life spans in a logarithmic scale. Among 93 true bugs, 48 bugs are *inherited*, which means that they are created before 2018, and 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 V-D. Even though we assume that 48 inherited bugs are created on January 1, 2018, the average life span is 414.5 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.

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.

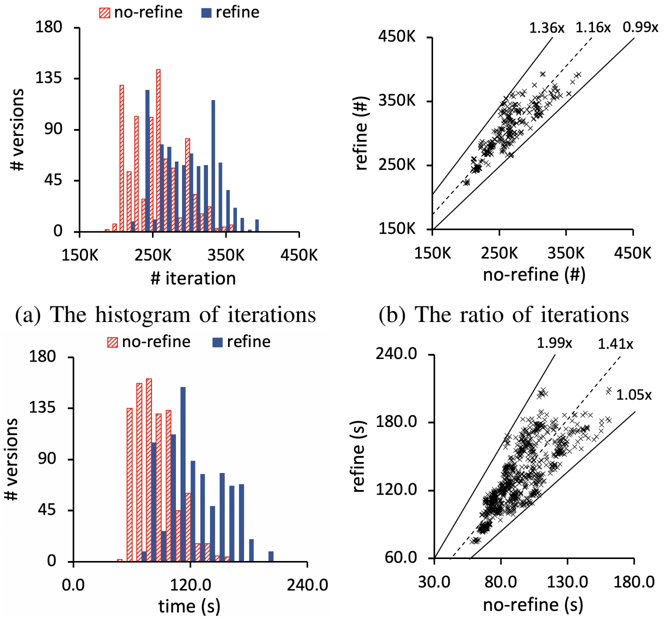
### C. Effectiveness of Refinement

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 8 presents the iterations and analysis time without and with refinement. Figures 8(a) and 8(c) are histograms of the iterations and analysis

TABLE III: Type-related specification bugs newly detected by JSTAR in the official draft of 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 |



(a) The histogram of iterations

(b) The ratio of iterations

(c) The histogram of time

(d) The ratio of time

Fig. 8: Comparison of iterations and analysis time without refinement (no-refine) and with refinement (refine)

time, respectively, and Figures 8(b) and 8(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 II 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).

## *D. 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 III 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 step 14 and 17 of the algorithm.

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 represent 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.

## VI. Related Work

Type analysis of JavaScript specifications has three related topics: JavaScript tools, mechanized specification extraction, and specification-based testing.

*a) JavaScript 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 [6], GraalJS [7], QuickJS [8], and Moddable XS [4] 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 JSAI [12], SAFE [10], TAJS [11], and WALA [13] based on the abstract interpretation framework [25], [26] 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. [9] 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. [14] introduced JaVerT, a JavaScript verification toolchain, based on the separation logic with an intermediate goto language JSIL. JaVerT 2.0 [15] 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.

*b) Mechanized Specification Extraction:* Researchers in various application domains have extracted mechanized specifications from specifications written in natural languages to handle the contents in the specifications automatically. For system architectures, researchers utilized Natural Language Processing (NLP) and Machine Learning techniques to extract formal semantics of small-sized low-level assembly languages for x86 [19] and ARM [20]. For Java API functions, Zhai et al. [23] presented a technique to automatically extract models from their documentation using NLP techniques. For the JavaScript programming language, Park et al. [5] presented JISET, a tool that extracts a mechanized specification from ECMAScript. While all the previous JavaScript formal semantics [16]–[18] were manually defined, JISET automatically extracts formal semantics directly from ECMAScript. We utilized JISET to analyze 864 ECMAScript versions via JSTAR.

*c) Specification-based Testing:* Recently, researchers have utilized specifications to test their implementations. For network protocols, Kim et al. [21] proposed a novel approach named BaseSpec, which extracts message structures from tables in cellular specifications for L3 protocols to perform comparative analysis of baseband software. Schumi and Sun [22] presented SpecTest, which utilized an executable language semantics to perform fuzzing for Java and Solidity compilers. For JavaScript, Ye et al. [28] presented Comfort, a compiler fuzzing framework to detect JavaScript engine bugs using ECMAScript with deep learning-based language models. Park et al. [29] extended JISET to 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. However, it requires multiple JavaScript engines and takes dozens of hours to test a version of ECMAScript. Instead, JSTAR can detect specification bugs without JavaScript engines in two minutes. Because JSTAR uses abstract semantics while JEST uses concrete semantics, JSTAR can quickly analyze more scope of semantics than JEST.

## VII. Conclusion

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, a JavaScript Specification Type Analyzer using Refinement. It is the first tool that performs *type analysis* on JavaScript specifications and detects specification bugs using a *bug detector*. 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.

## References

[1] (2021) Node.js, a JavaScript runtime built on Chrome's V8 JavaScript engine. [Online]. Available: https://nodejs.org/

[2] (2021) Espruino, JavaScript for Microcontrollers. [Online]. Available: https://www.espruino.com/

[3] (2021) Tessel 2, a robust IoT and robotics development platform. [Online]. Available: https://tessel.io/

[4] (2021) Moddable, a tool to create open IoT products using standard JavaScript on low cast microcontrollers. [Online]. Available: https://www.moddable.com/

[5] J. Park, J. Park, S. An, and S. Ryu, "JISET: JavaScript IR-based Semantics Extraction Toolchain," in *Proceedings of 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 647–658.

[6] (2021) Google's open source high-performance JavaScript and WebAssembly engine, written in C++. [Online]. Available: https://v8.dev/

[7] (2021) A high performance implementation of the JavaScript programming language. Built on the GraalVM by Oracle Labs. [Online]. Available: https://github.com/graalvm/graaljs

[8] (2021) A small and embeddable Javascript engine by Fabrice Bellard and Charlie Gordon. [Online]. Available: https://bellard.org/quickjs/

[9] A. Charguéraud, A. Schmitt, and T. Wood, "JSExplain: A Double Debugger for JavaScript," in *Companion Proceedings of the The Web Conference 2018*, 2018, pp. 691–699.

[10] C. Park and S. Ryu, "Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity," in *Proceedings of 29th European Conference on Object-Oriented Programming*, 2015, pp. 735–756.

[11] S. H. Jensen, A. Møller, and P. Thiemann, "Type Analysis for JavaScript," in *Proceedings of the 16th International Symposium on Static Analysis*, 2009, pp. 238–255.

[12] 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*, 2014, pp. 121–132.

[13] 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*, 2012, pp. 465–483.

[14] J. Fragoso Santos, P. Maksimović, D. Naudžiūnienundefined, T. Wood, and P. Gardner, "JaVerT: JavaScript Verification Toolchain," in *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2018, pp. 1–33.

[15] 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 Symposium on Principles of Programming Languages*, 2019, pp. 1–31.

[16] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The Essence of Javascript," in *Proceedings of the 24th European Conference on Object-oriented Programming*, 2010, pp. 126–150.

[17] M. Bodin, A. Chargueraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith, "A Trusted Mechanised JavaScript Specification," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014, pp. 87–100.

[18] 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*, 2015, p. 346–356.

[19] H. Nguyen, "Automatic extraction of x86 formal semantics from its natural language description," *Information Science*, 2018.

[20] A. V. Vu and M. Ogawa, "Formal semantics extraction from natural language specifications for ARM," in *International Symposium on Formal Methods*, 2019, pp. 465–483.

[21] 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*, 2021.

[22] R. Schumi and J. Sun, "SpecTest: Specification-Based Compiler Testing," in *Proceedings of the 24th International Conference on Fundamental Approaches to Software Engineering*, 2021, pp. 269–291.

[23] 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*, 2016, pp. 380–391.

[24] "Type Analysis for A Modified $\text{IR}_{\text{ES}}$," Tech. Rep., 2021. [Online]. Available: https://drive.google.com/uc?export=download&id=1uF4vXtMgZ7oLP6Cr0DzMLSlFovHOjewm

[25] 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*, 1977, pp. 238–252.

[26] ——, "Abstract interpretation frameworks," *Journal of Logic and Computation*, vol. 2, no. 4, pp. 511–547, 1992.

[27] 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*, vol. 40, no. 3, pp. 1–44, 2018.

[28] 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*, 2021.

[29] J. Park, S. An, D. Youn, G. Kim, and S. Ryu, "JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification," in *Proceedings of the 43rd International Conference on Software Engineering*, 2021.