Automatic Derivation of Static Analysis

Sukyoung Ryu
with PLRG@KAIST and friends

July 5, 2023
OPLSS: Static Analysis

(1) Concepts in Static Analysis
(2) Operational / Denotational Semantics
(3) Abstract Interpretation
(4) **Automatic Derivation of Static Analysis**
How to design and implement programming languages

Sukyoung Ryu

with PLRG@KAIST and friends

July 5, 2023
KAIST: Exception Analyzer

Links to other SML resources

Please feel free to send additional URLs that should be on this list to smlnj-dev-list@mailman.cs.uchicago.edu

SML Programming Resources

- Concurrent ML
- sml_tk, a library for using the TK graphical interface
- Martin Erwig's Functional Graph Library
- Yi and Ryu's SML/NJ exception analyzer

The Ariane-5 Rocket (1996)
Integer Overflow
$100M
Harvard: Debugging Everywhere

Debugging Everywhere

The goal of the Debugging Everywhere project is to make debugging a cheap, ubiquitous service. We intend to begin by getting compilers to emit Active Debugging Information, which we expect will support multi-language, multi-platform debugging much more readily than older approaches like Dwarf or dbx `"stabs."

```c
ldb Fib (stopped) > t
  0 <_print:2> (Mips/mjr.c:25,2) void _print(char *s = (0x10000000c) " ")
  1 <fib:51+0x24> (Fib.java:23,32)
    void fib(Fib this = {int buffer = 10,
      int[] a = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
      }, int n = 10)
  2 <main:3+0x18> (Fib.java:5,23) void main(String[] argv = {})
  3 <main:4+0x1c> (mininub.c:7,9)
    int main(int argc = 2, char **argv = 0x7fff7b24,
    char **envp = 0x7fff7b34)
```
Sun Microsystems: Fortress

Fortress is a discontinued experimental programming language for high-performance computing, created by Sun Microsystems with funding from DARPA's High Productivity Computing Systems project. One of the language designers was Guy L. Steele Jr., whose previous work includes Scheme, Common Lisp, and Java.

\[
z \text{: Vec := 0}
\]
\[
r \text{: Vec := } x
\]
\[
p \text{: Vec := } r
\]
\[
\rho \text{: Elt := } r^T r
\]
\[
\text{for } j \leftarrow \text{seq}(1: \text{cgit}_{\text{max}}) \text{ do}
\]
\[
q = Ap
\]
\[
\alpha = \frac{\rho}{p^T q}
\]
\[
z := z + \alpha p
\]
\[
r := r - \alpha q
\]
\[
\rho_0 = \rho
\]
\[
\rho := r^T r
\]
\[
\beta = \frac{\rho}{\rho_0}
\]
\[
p := r + \beta p
\]
\[
\text{end}
\]
\[
(z, \| x - Az \|)
\]
KAIST: JavaScript Analyzer

```
module M {s···}  module definition
module M = M ··· M;          module alias
import M ··· x;             qualified import
import M ··· x: x;           aliased import
import M ··· *;             import all
export var x [= e];          exported variable
export function x(x···){s···} exported function
export module M {s···}       exported module
export module M = M ··· M; exported module alias
export x;                   exported local
export x: x;                exported local alias
export x: M ··· x;           exported qualified alias

\[\begin{align*}
\phi & ::= x \cdots \\
\varphi_i & ::= \phi.(x) \\
\varphi_e & ::= \phi.x \\
\varphi & ::= \varphi_i \\
\varphi_e & \mid \varphi_e \\
\tilde{\varphi} & ::= \varphi.* \\
\tau & ::= \text{var} \\
\mid & \text{module} \\
\varsigma & ::= \epsilon \\
\mid & \text{local} \\
\mid & \text{export } \varphi_e \\
\rho & ::= \perp \\
\mid & \tau \varphi \\
\tilde{\rho} & ::= \perp \\
\mid & \top \\
\Sigma & ::= \{ (\varphi, \rho\varsigma) \cdots \} \\
\mid & \{ (\tilde{\varphi}, \tilde{\rho}) \cdots \} \\
\Sigma^* & ::= \epsilon \\
\mid & \Sigma^* \Sigma \\
\mid & \Sigma^* x
\end{align*}\]

\[\text{Figure 9. Extended syntax for JavaScript modules}\]

```
v ::= ··· | \alpha       value
\alpha ::= \langle v_g \rangle accessor
v_g ::= \text{func() \{ return e \}}  getter

\[\text{Figure 10. Extended syntax for } \lambda_{JS} \text{ to allow accessors}\]

\[\text{Figure 12. Desugaring environment for the modified } \lambda_{JS}\]
JavaScript Complex Semantics

```javascript
function f(x) {
    return x == !x;
}
```

Always return false?
JavaScript Complex Semantics

```javascript
function f(x) { return x == !x; }
```

Always return `false`?

**NO!!**

```javascript
f([]) -> [] == ![[]
-> [] == false
-> +[] == +false
-> 0 == 0
-> true
```
The production of `ArrayLiteral` in ES12
The production of `ArrayLiteral` in ES12

### 13.2.5.2 Runtime Semantics: Evaluation

Let `arrayLiteral : [ ElementList?, ElisionOpt ]`

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`.

The Evaluation algorithm for the third alternative of `ArrayLiteral` in ES12
Problem: Hand-Written JavaScript Static Analyzer

ECMA-262

define

Formal Semantics

abstract

λJS
KJS
JSIL
JSCert

[ECOOP’10]
[POPL’14]
[PLDI’15]
[POPL’17]

/

JavaScript
Programs

SAFE
TAJS
WALA
JSAI

... [FOOL’12]
[ECOOP’12]
[SAS’09]
[FSE’14]

Analysis
Result

manual

manual

manual

ECMA-262

Reference number

ECMA-

ECMAScript

Language

Specification

© Ecma International 2020
SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript

\[(H, A, tb) \in Heap \times Env \times ThisBinding\]

\[H \in Heap = \text{Loc} \xrightarrow{\text{fn}} \text{Object}\]

\[A \in Env ::= #\text{Global}\]

\[er :: A \rightarrow \text{DeclEnvRec} \cup \text{ObjEnvRec}\]

\[\sigma \in \text{DeclEnvRec} = \text{Var} \xrightarrow{\text{fn}} \text{StoreValue}\]

\[l \in \text{ObjEnvRec} = \text{Loc}\]

\[tb \in \text{ThisBinding} = \text{Loc}\]

---

**Figure 5.** Execution contexts and other domains

\[ct \in \text{Completion} ::= nc\]

\[nc \in \text{NormalCompletion} ::= \text{Normal}(vt)\]

\[ac \in \text{AbruptCompletion} ::= \text{Break}(vt, x)\]

\[\text{Return}(v)\]

\[\text{Throw}(ve)\]

\[vt \in \text{Val} \cup \{\text{empty}\}\]

\[ve \in \text{ValError} = \text{Val} \cup \text{Error}\]

---

**Figure 6.** Completion specification type
Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling

different levels of representations. Its default static analyzer on CFGs supports flow-sensitive and context-sensitive analyses of JavaScript programs by faithfully modeling the semantics of ECMAScript 5 [18]. A SAFE analysis computes the following summary map for a program:

\[ \hat{s} \in \mathbb{S} = \text{Node} \times \text{Context} \rightarrow \text{Heap} \]

which maps a program point represented by a pair of a CFG node and a context to an over-approximate abstract heap information. A variable or an object property in an abstract heap, \( \hat{h} \in \text{Heap} \), maps to an abstract value \( \hat{v} \) represented by a 6-tuple of lattice elements as follows:

\[ \hat{v} \in \mathbb{V} = \text{Undefined} \times \text{Null} \times \text{Boolean} \times \text{Number} \times \text{String} \times \varphi(\text{Loc}) \]

where \( \varphi(\text{Loc}) \) is a finite set of abstract locations that map to abstract objects in abstract heaps and the others are simple abstract domains for primitive types, Undefined, Null, Boolean, Number, and String. Their definitions are available in the SAFE manual [17]. For example, an abstract value \( \hat{v} \) that may be true or null is represented as follows:

\[ \hat{v} = (\bot_{\text{Undefined}}, \bot_{\text{Null}}, \text{true}, \bot_{\text{Number}}, \bot_{\text{String}}, \varnothing) \].

With the domains, the default static analyzer performs sound and elaborate analyses on CFGs of JavaScript programs with the transfer function \( \hat{F} \in \mathbb{S} \rightarrow \mathbb{S} \) to compute a final summary map \( \hat{s}_{\text{final}} \) from the following least fixpoint computation:

\[ \hat{s}_{\text{final}} = \text{leastFix} \lambda \hat{s}. (\hat{s}_{\text{init}} \cup \hat{s} \ast \hat{F}(\hat{s})) \]

Fig. 3: Overall structure of the SAFE\(_{\text{WApp}}\) framework
Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications

**Collecting Semantics.** We denote a program \( P \) as a directed graph \( (C, E) \) where \( C \) and \( E \) represent a set of nodes and a set of edges, respectively. A node \( c \in C \) denotes a control point in the program and an edge \( (c, c') \in E \subseteq C \times C \) denotes a control flow from \( c \) to \( c' \).

Given a program \( P \), we define its collecting semantics using \( (C, \varphi(S), f, \rightarrow_{\phi}) \) where:

- \( C \): a finite set of control points;
- \( \sigma \in \varphi(S) \): a powerset of concrete states;
- \( f(c) \in \varphi(S) \rightarrow \varphi(S) \): a set of local semantic functions at a given control point \( c \);
- \( \phi \in C \rightarrow \varphi(S) \): a set of program states mapping control points to sets of concrete states; and
- \( \rightarrow_{\phi} \): a set of control flows for a given program state

**C. Tuned Static Analysis**

Finally, using the baseline analysis \( \hat{F} \) and the contour \( \hat{C} \), our framework derives a tuned static analysis which over-approximates the selected set of program executions approximated by \( \hat{C} \). Using the sound abstract semantic function \( \hat{f} \) from the baseline analysis and the interpretation of the contour \( \mathcal{I}(\hat{C}) \), we can define a tuned analysis which over-approximates selected executions as the least fixpoint of the following abstract semantic function:

\[
\hat{F}_{s}(\hat{\phi}) = \lambda c \in C. \ \hat{f}(c) \left( \bigsqcup_{c' \in \hat{\Delta}_{c}} \hat{\phi}(c') \right) \cap \mathcal{I}(\hat{C}),
\]

**Theorem I (Correctness of \( \hat{F}_{s} \)):** \( \alpha_{m}(\text{lfp} F_{s}) \subseteq \text{lfp} \hat{F}_{s} \)

Also, the interpretation of the contour \( \mathcal{I}(\hat{C}) \) is an over-approximation of the derived tuned analysis of the selected executions:

**Theorem 2:** \( \text{lfp} \hat{F}_{s} \subseteq \mathcal{I}(\hat{C}) \)
SAFE\textsubscript{WAPI}: Web API Misuse Detector for Web Applications

**Figure 4: Architecture of SAFE\textsubscript{WAPI}**

```java
[Callback=FunctionOnly, NoInterfaceObject]
interface CalendarArraySuccessCallback {
    void onsuccess(Calendar[] calendars);
};

[NoInterfaceObject] interface Calendar {
    readonly attribute CalendarId id;
    readonly attribute DOMString name;
    CalendarItem get(CalendarItemID id);
    void add(CalendarItem item);
};

1. Accesses to absent properties of platform objects (AbsProp)
2. Wrong number of arguments to API function calls (ArgNum)
3. Missing error callback functions (ErrorCB)
4. Unhandled API calls that may throw exceptions (ExnHnd)
5. Wrong types of arguments to API function calls (ArgTyp)
6. Accesses to absent attributes of dictionary objects (AbsAttr)

**Figure 1: Web API specification written in Web IDL**
Automatic Modeling of Opaque Code for JavaScript Static Analysis

We write the SRA model as $\downarrow_{SRA}: C \times \hat{S} \rightarrow \hat{S}$ and define it as follows:

$$\downarrow_{SRA}(c, \hat{s}) = \text{Abstract}(\{\text{Run}(c, s) \mid s \in \text{Sample}(\hat{s})\})$$
$$= \text{Broaden}(\bigcup\{\alpha(\{\text{Run}(c, s)\}) \mid s \in \text{Sample}(\hat{s})\})$$

We now describe how $\downarrow_{SRA}$ works using an example abstract domain for even and odd integers as shown in Fig. 1. Let us consider the code snippet $x := \text{abs}(x)$ at a program point $c$ where the library function $\text{abs}$ is opaque. We use maps from variables to their concrete values for concrete states, maps from variables to their abstract values for abstract states, and the identity function for $\text{Broaden}$ in this example.

Case $\hat{s}_1 \equiv [x : n]$ where $n$ is a constant integer:

$$\downarrow_{SRA}(c, \hat{s}_1) = \bigcup\{\alpha(\{\text{Run}(c, s)\}) \mid s \in \text{Sample}(\hat{s}_1)\}$$
$$= \bigcup\{\alpha(\{\text{Run}(c, s)\}) \mid s \in \{[x : n]\}\}$$
$$= \bigcup\{\alpha(\{[x : n]\})\}$$
$$= [x : [n]]$$
Problem: Hand-Written JavaScript Static Analyzer

ECMA-262

\( \lambda_{JS} \) [ECOOP'10]
KJS [POPL'14]
JSIL [PLDI'15]
JSCert [POPL'17]

Formal Semantics

abstract

JavaScript Programs

SAFE
TAJS
WALA
JSAI

... 

JS Static Analyzer

Analysis Result

manual

analyzer
developer

manual

Reference number

ECMA-123:2009

© Ecma International 2009

ECMA-262 1

11th Edition / June 2020

ECMAScript ®

Language Specification
Problem: Fast Evolving JavaScript

1996 - ES1: First edition

1998 - ES2: Editorial changes

1999 - ES3: RegEx, String, Try/catch, etc

2000
2002
2004
2008
2010
2012
2014

2009 - ES5: getters/setters, strict mode, exceptions, etc

2011 - ES5.1: Editorial Changes

KJS, SAFE, TAJS, JSIL, WALA, JSAI
Problem: Fast Evolving JavaScript

- **1996**: First edition
- **1997 - ES1**: First edition
- **1998 - ES2**: Editorial changes
- **1999 - ES3**: RegEx, String, Try/catch, etc
- **2000**:
- **2004**:
- **2008**:
- **2010**:
- **2012**:
- **2014**:
- **2015 - ES6**: classes, modules, etc
- **2016**:
- **2017**:
- **2018**:
- **2019 - ES10**:
- **2020 - ES11**:
- **2020 - ES12**:
- **2021 - ES12**:
- **2022**:

Annual Releases

**ES.Next**

ECMAScript 2021 (ES12) - 879 pages
Main Idea: Deriving Static Analyzer from Spec.
Overall Structure
JISET: JavaScript IR-based Semantics Extraction Toolchain
Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu

1. Mechanized Spec. Extraction
2. Specification Validity Check
3. Derivation of Static Analyzers

Conformance Test Synthesis

JEST
[JICSE'21]

JEST_{fs}
[ICSE'21]

JSAVER

JSTAR
[ASE'21]

Type Analysis for Specification

Derived Static Analyzer

Analysis Result

JavaScript Programs

ECMA-262

[ASE'20]

Mechanized Specification

[ICSE'21]
Motivation: Patterns in Writing Style of ECMA-262

13.2.5.2 Runtime Semantics: Evaluation

\[ \text{ArrayLiteral} : [ \text{ElementList}, \text{Elision} \text{opt} ] \]

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

The Evaluation algorithm for the third alternative of ArrayLiteral in ES12
### Key Idea: Metalanguage for ECMA-262

- **IR\textsubscript{ES} - Intermediate Representation for ECMAScript**

<table>
<thead>
<tr>
<th>Programs</th>
<th>$\Psi \ni P := f^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>$\mathcal{F} \ni f := \text{syntax?} ; \text{def} ; x(x^<em>) ; {[l : i]^</em>}$</td>
</tr>
<tr>
<td>Variables</td>
<td>$\mathcal{X} \ni x$</td>
</tr>
<tr>
<td>Labels</td>
<td>$\mathcal{L} \ni \ell$</td>
</tr>
<tr>
<td>Instructions</td>
<td>$\mathcal{I} \ni i := r := e \mid x := {} \mid x := e(e^*) \mid \text{if} ; e ; \ell ; \ell \mid \text{return} ; e$</td>
</tr>
<tr>
<td>Expressions</td>
<td>$\mathcal{E} \ni e := v^p \mid \text{op}(e^*) \mid r$</td>
</tr>
<tr>
<td>References</td>
<td>$\mathcal{R} \ni r := x \mid e[e] \mid e[e]{_{\text{js}}}$</td>
</tr>
</tbody>
</table>

| States            | $\sigma \in \mathcal{S} = \mathcal{L} \times \mathcal{E} \times \mathcal{C}^* \times \mathcal{H}$ |
| Environments      | $\rho \in \mathcal{E} = \mathcal{X} \xrightarrow{\text{fin}} \mathcal{V}$ |
| Calling Contexts  | $c \in \mathcal{C} = \mathcal{L} \times \mathcal{E}$ |
| Heaps             | $h \in \mathcal{H} = \mathcal{A} \xrightarrow{\text{fin}} \mathcal{L} \times \mathcal{M} \times \mathcal{M}_{\text{js}}$ |
| Internal Field Maps| $m \in \mathcal{M} = \mathcal{V}_{\text{str}} \xrightarrow{\text{fin}} \mathcal{V}$ |
| External Field Maps| $m_{\text{js}} \in \mathcal{M}_{\text{js}} = \mathcal{V}_{\text{str}} \xrightarrow{\text{fin}} \mathcal{V}$ |
| Values            | $v \in \mathcal{V} = \mathcal{A} \cup \mathcal{V}^p \cup \mathcal{T} \cup \mathcal{F}$ |
| Primitive Values  | $v^p \in \mathcal{V}^p = \mathcal{V}_{\text{bool}} \cup \mathcal{V}_{\text{int}} \cup \mathcal{V}_{\text{str}} \cup \cdots$ |
| JS ASTs           | $t \in \mathcal{T}$ |

---

**Concrete States.**

Since $\mathcal{J} \ni \text{astid} \ni \text{op} \ni \mathcal{F}$ denotes the concrete state, the notation for ASTs. The notation $\mathcal{J} \ni \text{op} \ni \mathcal{F}$ yields several notations to easily deal with JavaScript programs.

**Evaluation**

In the syntactic production of nonterminal symbol $\text{Program}$, it is a syntax-directed function, otherwise, $\text{Program}$ is a terminal symbol or another tree for a nonterminal symbol.

**Syntax and Notations.**

Since $\mathcal{J} \ni \text{op} \ni \mathcal{F}$ denotes the concrete state, the notation for ASTs. The notation $\mathcal{J} \ni \text{op} \ni \mathcal{F}$ yields several notations to easily deal with JavaScript programs.

**References**

Since $\mathcal{J} \ni \text{op} \ni \mathcal{F}$ denotes the concrete state, the notation for ASTs. The notation $\mathcal{J} \ni \text{op} \ni \mathcal{F}$ yields several notations to easily deal with JavaScript programs.

**Programs**

We denote the denotational semantics of functions. A function is a variable or an expression. A function is a variable or an expression. A function is a variable or an expression.

---

**Environments**

Since $\mathcal{J} \ni \text{op} \ni \mathcal{F}$ denotes the concrete state, the notation for ASTs. The notation $\mathcal{J} \ni \text{op} \ni \mathcal{F}$ yields several notations to easily deal with JavaScript programs.

**Labels**

Since $\mathcal{J} \ni \text{op} \ni \mathcal{F}$ denotes the concrete state, the notation for ASTs. The notation $\mathcal{J} \ni \text{op} \ni \mathcal{F}$ yields several notations to easily deal with JavaScript programs.
Key Idea: Metalanguage for ECMA-262

13.2.5.2 Runtime Semantics: Evaluation

ArrayLiteral : [ ElementList , Elisionopt ]

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.

```javascript
syntax def ArrayLiteral[2].Evaluation(
   this, ElementList, Elision
) {
   let array = [! (ArrayCreate 0)]
   let nextIndex = (ElementList.ArrayAccumulation array 0) ![? nextIndex]
   if (! (= Elision absent)) {
      let len = (Elision.ArrayAccumulation array nextIndex) ![? len]
   }
   return array
}
```
In this thesis, we introduce a novel approach to automatically derive JavaScript static analyzers. We show that it successfully extracts a mechanized specification from the latest ECMAScript. Then, we present the extraction, mechanized specification validity check, and derivation of static analyzers. First, we perform a type analysis for the mechanized specifications using a meta-level static analysis. For evaluation, we derive a JavaScript static analyzer from the latest ECMAScript and show that it successfully analyzes all applicable o
In this thesis, we introduce a novel approach to automatically derive JavaScript static analyzers from any version of ECMAScript. Our approach consists of three steps: 1) mechanized specification extraction, 2) mechanized specification validity check, and 3) derivation of static analyzers. First, we derive a JavaScript specification. Finally, we present two differential testing with JavaScript engines and toolchain that it successfully extracts a mechanized specification from the latest ECMAScript. Then, we present 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 difficult to correctly understand. This is why we introduce JISET, which automatically derives JavaScript static analyzers from ECMAScript. We show that it successfully extracts a mechanized specification from ECMAScript. We also show that JISET automatically derives JavaScript static analyzers from any version of ECMAScript.
# JISET - Evaluation

![Approximately 96% Compiled]

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

<table>
<thead>
<tr>
<th>Version</th>
<th># Algo.</th>
<th>Test262 (Official Conformance Tests)</th>
<th>Parsing tests</th>
<th>Evaluation Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>- 18,556 applicable tests</td>
<td>- Passed all 18,556 tests</td>
<td>- Passed all 18,556 tests</td>
</tr>
</tbody>
</table>

- **Test262 (Official Conformance Tests)**
- **Parsing tests**
- **Evaluation Tests**

<table>
<thead>
<tr>
<th>Version</th>
<th># Algo.</th>
<th>T: Total</th>
<th>L: Core Language Semantics</th>
<th>B: Built-in Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES7</td>
<td>2,105</td>
<td>10,471 / 10,982 (95.35%)</td>
<td>8,041 / 8,415 (95.56%)</td>
<td>2,430 / 2,567 (94.66%)</td>
</tr>
<tr>
<td>ES8</td>
<td>2,238</td>
<td>11,181 / 11,732 (95.30%)</td>
<td>8,453 / 8,811 (95.94%)</td>
<td>2,728 / 2,921 (93.39%)</td>
</tr>
<tr>
<td>ES9</td>
<td>2,370</td>
<td>11,849 / 12,393 (95.61%)</td>
<td>8,932 / 9,311 (95.93%)</td>
<td>2,917 / 3,082 (94.65%)</td>
</tr>
<tr>
<td>ES10</td>
<td>2,396</td>
<td>12,022 / 12,569 (95.65%)</td>
<td>9,073 / 9,456 (94.95%)</td>
<td>2,949 / 3,113 (94.73%)</td>
</tr>
<tr>
<td>ES11</td>
<td>2,521</td>
<td>12,505 / 13,047 (94.85%)</td>
<td>9,495 / 9,881 (96.09%)</td>
<td>3,010 / 3,166 (95.07%)</td>
</tr>
<tr>
<td>ES12</td>
<td>2,640</td>
<td>12,975 / 13,544 (95.80%)</td>
<td>9,717 / 10,136 (95.87%)</td>
<td>3,258 / 3,408 (95.60%)</td>
</tr>
<tr>
<td>Average</td>
<td>2,378</td>
<td>11,834 / 12,378 (95.61%)</td>
<td>8,952 / 9,335 (95.90%)</td>
<td>2,882 / 3,043 (94.71%)</td>
</tr>
</tbody>
</table>
JEST: N+1-version Differential Testing of Both JavaScript Engines

Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu

1. Mechanized Spec. Extraction
   - ECMA-262
   - [ASE'20] JISET
   - 1. Mechanized Spec. Extraction

2. Specification Validity Check
   - [ICSE'21] JEST
   - Conformance Test Synthesis

3. Derivation of Static Analyzers
   - JSAVER
   - [FSE'22] Derived Static Analyzer
   - Analysis Result
   - Type Analysis for Specification

- JavaScript Programs
- JSTARS
- [ASE'21] JEST
- [ICSE'21] JESTfs
JEST - Conformance with Engines

ECMA-262

Conform

QuickJS

GraalVM

moddable

JavaScript Engines
JEST - N+1-version Differential Testing

Synthesize

Test

An engine bug in

JavaScript Engines
JEST - N+1-version Differential Testing

A specification bug in ECMA-262
An engine bug in GraalVM.
In this thesis, we introduce a novel approach to automatically derive JavaScript static analyzers. We utilize mechanized specifications of JavaScript programs, thereby enabling researchers to correctly understand the behaviors of JavaScript programs. To automatically reason about them, researchers have developed various JavaScript static analyzers that conform to ECMAScript, the standard specification of JavaScript. For evaluation, we derive a JavaScript engine and conduct differential testing with JavaScript engines and JavaScript, mechanized specification extraction, and various JavaScript static analyzers in this thesis.
In this thesis, we introduce a novel approach to automatically derive JavaScript static analyzers for different tools to detect bugs in JavaScript specifications and engines. We utilize mechanized specifications using a meta-level static analysis. For evaluation, we derive a JavaScript static analyzer that automatically derives JavaScript static analyzers from mechanized specifications that automatically derives JavaScript static analyzers from mechanized specifications. This approach is known as JEST (JavaScript Engines and Specification Tester).
In this thesis, we introduce a novel approach to automatically derive JavaScript static analyzers. The approach includes three main steps: 1) program generation, 2) mechanized specification validity check, and 3) derivation of static analyzers. First, we automatically derive JavaScript static analyzers from the latest ECMAScript. Then, we present two different tools to detect bugs in JavaScript specifications and engines; 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 becomes more critical because the co-evolution of specifications, tests, and tools for programming languages.

For evaluation, we derive a JavaScript specification. Finally, we present the conformance tests in a sound way. We believe that the thesis would be the first step towards the mechanized specifications using a meta-level static analysis. The static analyzer from the latest ECMAScript shows that it successfully analyzes all applicable objects.
JEST - Assertion Injector (7 Kinds)

```javascript
var x = 1 + 2;

+ $assert.sameValue(x, 3);
```
1. Exceptions (Exc)

```javascript
+ // Throw
let x = 42;
function x() {};
```

2. Aborts (Abort)

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

3. Variable Values (Var)

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

4. Object Values (Obj)

```javascript
var x = {}, y = {}, z = { p: x, q: y };
+ $assert.sameValue(z.p, x);
+ $assert.sameValue(z.q, y);
```
5. **Object Properties (Desc)**

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

6. **Property Keys (Key)**

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

7. **Internal Methods and Slots (In)**

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

- JEST successfully synthesized 1,700 conformance tests from ES11

<table>
<thead>
<tr>
<th>Engines</th>
<th>Exc</th>
<th>Abort</th>
<th>Var</th>
<th>Obj</th>
<th>Desc</th>
<th>Key</th>
<th>In</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>V8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>GraalVM</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>QuickJS</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Moddable XS</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>21</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>17</td>
<td>0</td>
<td>44</td>
</tr>
</tbody>
</table>
JEST - Evaluation

- JEST successfully synthesized 1,700 conformance tests from ES11

### TABLE II: The number of engine bugs detected by JEST

<table>
<thead>
<tr>
<th>Engines</th>
<th>Exc</th>
<th>Abort</th>
<th>Var</th>
<th>Obj</th>
<th>Desc</th>
<th>Key</th>
<th>In</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>V8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>GraalVM</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>QuickJS</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Moddable XS</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>21</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>17</td>
<td>0</td>
<td>44</td>
</tr>
</tbody>
</table>

### TABLE III: Specification bugs in ECMAScript 2020 (ES11) detected by JEST

<table>
<thead>
<tr>
<th>Name</th>
<th>Feature</th>
<th>#</th>
<th>Assertion</th>
<th>Known</th>
<th>Created</th>
<th>Resolved</th>
<th>Existed</th>
<th>Days</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES11-1</td>
<td>Function</td>
<td>12</td>
<td>Key</td>
<td>O</td>
<td>2019-02-07</td>
<td>2020-04-11</td>
<td>429</td>
<td></td>
</tr>
<tr>
<td>ES11-2</td>
<td>Function</td>
<td>8</td>
<td>Key</td>
<td>O</td>
<td>2015-06-01</td>
<td>2020-04-11</td>
<td>1,776</td>
<td></td>
</tr>
<tr>
<td>ES11-3</td>
<td>Loop</td>
<td>1</td>
<td>Exc</td>
<td>O</td>
<td>2017-10-17</td>
<td>2020-04-30</td>
<td>926</td>
<td></td>
</tr>
<tr>
<td>ES11-4</td>
<td>Expression</td>
<td>4</td>
<td>Abort</td>
<td>O</td>
<td>2019-09-27</td>
<td>2020-04-23</td>
<td>209</td>
<td></td>
</tr>
<tr>
<td>ES11-5</td>
<td>Expression</td>
<td>1</td>
<td>Exc</td>
<td>O</td>
<td>2015-06-01</td>
<td>2020-04-28</td>
<td>1,793</td>
<td></td>
</tr>
<tr>
<td>ES11-6</td>
<td>Object</td>
<td>1</td>
<td>Exc</td>
<td>X</td>
<td>2019-02-07</td>
<td>2020-11-05</td>
<td>637</td>
<td></td>
</tr>
</tbody>
</table>

27 Bugs in Spec.

44 Bugs in Engines
JEST - Example in GraalVM.

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

“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.”

- A Developer of GraalVM
JSTAR: JavaScript Specification Type Analyzer using Refinement
Jihyeok Park, Seungmin An, Wonho Shin, Yusung Sim, and Sukyoung Ryu

ECMA-262 → JISET
1. Mechanized Spec.
Extraction

Mechanized
Specification

JEST
Conformance Test
Synthesis

JSAVER
3. Derivation of
Static Analyzers

Derived Static
Analyzer

Analysis
Result

[JEESE'20]

Type Analysis for
Specification

[ICSE'21]

[ASE'21]

[JSAVER]

[ICSE'21]

[JEST]
JSTAR - Types in Specification

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 < 0.5 and x > 0, return +0.
4. If x < 0 and x ≥ -0.5, return -0.

https://github.com/tc39/ecma262/tree/575149cfd77aebcf3a129e165bd89e14caafc31c
20.3.2.28  \textbf{Math.round} \((x)\) \quad x: (\text{String v Boolean v Number v Object v ...)

1. Let \(n\) be \(? \text{ToNumber}(x)\).
2. If \(n\) is an integral Number, return \(n\).
3. If \(x < 0.5\) and \(x > 0\), return \(+0\).
4. If \(x < 0\) and \(x \geq -0.5\), return \(-0\).
...
20.3.2.28 Math.round (x)

x: (String v Boolean v Number v Object v ...)

1. Let \( n \) be ? ToNumber (x).
   \( n: (\text{Number}) \land \text{ToNumber}(x): (\text{Number} v \text{Exception}) \)
2. If \( n \) is an integral Number, return \( n \).
3. If \( x < 0.5 \) and \( x > 0 \), return +0.
4. If \( x < 0 \) and \( x \geq -0.5 \), return -0.
...
20.3.2.28 Math.round (x) x: (String v Boolean v Number v Object v ...)

1. Let n be ? ToNumber(x). n: (Number) ∧ ToNumber(x): (Number v Exception)
2. If n is an integral Number, return n.
3. If $x < 0.5$ and $x > 0$, return +0.
4. If $x < 0$ and $x \geq -0.5$, return -0.

Type Mismatch for numeric operator `>`

Math.round(true) = ???
Math.round(false) = ???
JSTAR - Types in Specification

20.3.2.28 Math.round \((x)\)  
x: (String v Boolean v Number v Object v ...)

1. Let \(n\) be ?ToNumber\((x)\). n: (Number) ∧ ToNumber\((x)\): (Number v Exception)
2. If \(n\) is an integral Number, return \(n\).
3. If \(x < 0.5\) and \(x > 0\), return +0.
4. If \(x < 0\) and \(x ≥ -0.5\), return -0.

Type Mismatch for numeric operator `>`

Math.round(true) = ???
Math.round(false) = ???

3. If \(n < 0.5\) and \(n > 0\), return +0.
4. If \(n < 0\) and \(n ≥ -0.5\), return -0.

Math.round(true) = 1
Math.round(false) = 0

https://github.com/tc39/ecma262/tree/575149cfd77aebcf3a129e165bd89e14caafcc31c
In this thesis, we introduce a novel approach to automatically derive JavaScript static analyzers from the latest ECMAScript. We show that it successfully extracts a mechanized specification from ECMAScript. Then, we present a tool that automatically derives JavaScript static analyzers from 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 performs differential testing, specifica-

Abstract

...
In this thesis, we introduce a novel approach to automatically derive JavaScript static analyzers. The highly dynamic nature and complex semantics of JavaScript make it difficult to correctly understand various JavaScript static analyzers that conform to ECMAScript, the standard specification of JavaScript. Furthermore, since late 2014, JavaScript rapidly evolves with a yearly release cadence and open development process. This evolution makes the task of detecting bugs in JavaScript specifications and engines labor-intensive and error-prone.

Our approach consists of three steps: 1) mechanized specification extraction, 2) mechanized specification validity check, and 3) derivation of static analyzers. First, we derive a JavaScript specification. Finally, we present that it successfully extracts a mechanized specification from the latest ECMAScript. Then, we present the instrumented static analyzers that we have developed and evaluated. Our approach can be further extended to check the specification consistency with specifications available from a variety of sources.
Intermediate Representation for ECMAScript

Functions \[ F \ni f ::= \text{def } x(x^*, [x^*]) \ell \]

Instructions \[ I \ni i ::= \text{let } x = e \mid x = (e e^*) \mid \text{assert } e \mid \text{if } e \ell \ell \mid \text{return } e \mid r = e \]

References \[ r ::= x \mid r[e] \]

Expressions \[ e ::= t \{ [x : e]^* \} \mid [e^*] \mid e : \tau \mid r? \mid e \oplus e \mid \ominus e \mid r \mid c \mid p \]

Primitives \[ P \ni p ::= \text{undefined} \mid \text{null} \mid b \mid n \mid j \mid s \mid @s \]

Types \[ T \ni \tau ::= t \mid [] \mid [\tau] \mid js \mid \text{prim} \mid \text{undefined} \mid \text{null} \mid \text{bool} \mid \text{numeric} \mid \text{num} \mid \text{bigint} \mid \text{str} \mid \text{symbol} \]
Semantic Domains

States \[ d \in S = L \times C^* \times H \times E \]
Contexts \[ \kappa \in C = L \times E \times X \]
Heaps \[ h \in H = A \rightarrow O \]
Addresses \[ a \in A \]
Objects \[ o \in O = (T_t \times (V_s \rightarrow V)) \cup V^* \]
Nominal Types \[ t \in T_t \]
Environments \[ \sigma \in E = X \times V \]
Values \[ v \in V = F \cup A \cup V_c \cup P \]
Constants \[ c \in V_c \]
Strings \[ s \in V_s \]
B. Instructions: $[i]_i : S \rightarrow S$

- Variable Declarations:
  \[
  [\text{let } x = e]_i(d) = (\text{next}(l, \overline{\kappa}, h, \sigma[x \mapsto v])
  \]
  where\n  \[
  [e]_e(d) = ((l, \overline{\kappa}, h, \sigma), v)
  \]

- Function Calls:
  \[
  [x = (e_0 \ e_1 \cdots e_n)]_i(d) = (l_{\overline{\kappa}}, \kappa :: \overline{\kappa}, h, \sigma')
  \]
  where\n  \[
  [e_0]_e(d) = (d_0, \text{def } f(p_1, \cdots, p_m) l_{\overline{\kappa}})
  \]
  \[
  [e_1]_e(d_0) = (d_1, v_1) \wedge \cdots \wedge [e_n]_e(d_{n-1}) = (d_n, v_n) \wedge
  \]
  \[
  d_n = (l, \overline{\kappa}, h, \sigma) \wedge k = \min(n, m) \wedge
  \]
  \[
  \sigma' = [p_1 \mapsto v_1, \cdots, p_k \mapsto v_k] \wedge \kappa = (\text{next}(l), \sigma, x)
  \]

- Assertions:
  \[
  [\text{assert } e]_i(d) = d' \text{ if } [e]_e(d) = (d', \#t)
  \]

- Branches:
  \[
  [\text{if } e \ l_1 \ l_2]_i(d) = \begin{cases} (l_\kappa, \overline{\kappa}, h, \sigma) & \text{if } v = \#t \\ (l_\kappa, \overline{\kappa}, h, \sigma) & \text{if } v = \#f \\ \end{cases}
  \]
  where\n  \[
  [e]_e(d) = ((l, \overline{\kappa}, h, \sigma), v)
  \]

- Returns:
  \[
  [\text{return } e]_i(d) = (l, \overline{\kappa}, h, \sigma[x \mapsto v])
  \]
  where\n  \[
  [e]_e(d) = ((_, l, \sigma, x) :: \overline{\kappa}, h, _, v)
  \]

- Variable Updates:
  \[
  [x = e]_i(d) = (\text{next}(l), \overline{\kappa}, h, \sigma[x \mapsto v])
  \]
  where\n  \[
  [e]_e(d) = ((l, \overline{\kappa}, h, \sigma), v)
  \]

- Field Updates:
  \[
  [r[e_0] = e_1]_i(d) = (\text{next}(l), \overline{\kappa}, h[a \mapsto o'], \sigma)
  \]
  where\n  \[
  [r]_e(d) = (d', a) \wedge [e_0]_e(d') = (d_0, v_0) \wedge
  \]
  \[
  [e_1]_e(d_0) = ((l, \overline{\kappa}, h, \sigma), v_1) \wedge o = h(a) \wedge
  \]
  \[
  o' = \begin{cases} o_r & \text{if } o = (t, fs) \wedge v_0 = s \\ o_l & \text{if } o = [v'_1, \cdots, v'_m] \wedge v_0 = n \wedge
  \]
  \[
  o_r = (t, fs[s \mapsto v_1]) \wedge o_l = [\cdots, v'_{n-1}, v_1, v'_{n+1}, \cdots]
  \]
Semantics of Expressions

D. Expressions: $\llbracket e \rrbracket_e : S \rightarrow S \times V$

- Records:
  \[
  \llbracket t\ (x_1 : e_1, \ldots, x_n : e_n) \rrbracket_e(d) = (d', a)
  \]
  where
  \[
  \begin{align*}
  \llbracket e_1 \rrbracket_e(d) &= (d_1, v_1) \land \cdots \land \llbracket e_n \rrbracket_e(d_{n-1}) = (d_n, v_n) \land \\
  d_n &= (f, \kappa, h, \sigma) \land \llbracket e_n \rrbracket = [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \\
  a &\not\in \text{Domain}(h) \land d' = (f, \kappa, h[a \mapsto (t, \ell s)], \sigma)
  \end{align*}
  \]

- Lists:
  \[
  \llbracket [e_1, \ldots, e_n] \rrbracket_e(d) = (d', a)
  \]
  where
  \[
  \begin{align*}
  \llbracket e_1 \rrbracket_e(d) &= (d_1, v_1) \land \cdots \land \llbracket e_n \rrbracket_e(d_{n-1}) = (d_n, v_n) \land \\
  d_n &= (f, \kappa, h, \sigma) \land a \not\in \text{Domain}(h) \land \\
  d' &= (f, \kappa, h[a \mapsto [v_1, \ldots, v_n]], \sigma)
  \end{align*}
  \]

- Type Checks:
  \[
  \llbracket e : \tau \rrbracket_e(d) = (d', a)
  \]
  where
  \[
  \begin{align*}
  \llbracket e \rrbracket_e(d) &= (d', v) \land b = \begin{cases} 
  \#t & \text{if } v \text{ is a value of } \tau \\
  \#\ell & \text{otherwise}
  \end{cases}
  \end{align*}
  \]

- Variable Existence Checks:
  \[
  \llbracket x? \rrbracket_e(d) = (d, b)
  \]
  where
  \[
  d = (\_., \_., \ldots, \sigma) \land b = \begin{cases} 
  \#t & \text{if } x \in \text{Domain}(\sigma) \\
  \#\ell & \text{otherwise}
  \end{cases}
  \]

- Field Existence Checks:
  \[
  \llbracket r\ [e] ? \rrbracket_e(d) = (d'', b)
  \]
  where
  \[
  \begin{align*}
  \llbracket r\ [e] \rrbracket_e(d) &= (d', a) \land \llbracket e \rrbracket_e(d') = (d'', v) \land \\
  d'' &= (f, \kappa, h, \sigma) \land a = h(a) \land \\
  b &= \begin{cases} 
  \#t & \text{if } o = (t, \ell s) \land v = s \land s \in \text{Domain}(\ell s) \\
  \#\ell & \text{if } o = [v'_1, \ldots, v'_m] \land v = n \land 1 \leq n \leq m \\
  \#\ell & \text{otherwise}
  \end{cases}
  \end{align*}
  \]

- Binary Operations:
  \[
  \llbracket e \oplus e' \rrbracket_e(d) = (d'', v_0 \oplus v_1)
  \]
  where
  \[
  \begin{align*}
  \llbracket e_0 \rrbracket_e(d) &= (d', v_0) \land \llbracket e_1 \rrbracket_e(d') = (d'', v_1)
  \end{align*}
  \]

- Unary Operations:
  \[
  \llbracket e \ominus e' \rrbracket_e(d) = (d', \ominus v)
  \]
  where
  \[
  \begin{align*}
  \llbracket e \rrbracket_e(d) &= (d', v)
  \end{align*}
  \]

- References:
  \[
  \llbracket r \rrbracket_e(d) = [r]_r(d)
  \]

- Constants:
  \[
  \llbracket c \rrbracket_e(d) = (d, c)
  \]

- Primitives:
  \[
  \llbracket p \rrbracket_e(d) = (d, p)
  \]
## Abstract Semantic Domains

<table>
<thead>
<tr>
<th>Category</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract States</td>
<td>$\mathcal{d}^# \in \mathcal{S}^# = \mathcal{M} \times \mathcal{R}$</td>
</tr>
<tr>
<td>Result Maps</td>
<td>$\mathcal{m} \in \mathcal{M} = \mathcal{L} \times \mathcal{T}^* \rightarrow \mathcal{E}^#$</td>
</tr>
<tr>
<td>Return Point Maps</td>
<td>$\mathcal{r} \in \mathcal{R} = \mathcal{F} \times \mathcal{T}^* \rightarrow \mathcal{P}(\mathcal{L} \times \mathcal{T}^* \times \mathcal{X})$</td>
</tr>
<tr>
<td>Abstract Environments</td>
<td>$\mathcal{\sigma}^# \in \mathcal{E}^# = \mathcal{X} \rightarrow \mathcal{T}^#$</td>
</tr>
<tr>
<td>Abstract Types</td>
<td>$\mathcal{\tau}^# \in \mathcal{T}^# = \mathcal{P}(\mathcal{T})$</td>
</tr>
</tbody>
</table>
Abstract Semantic Functions

Then, we define the abstract semantics $\llbracket P \rrbracket^\#$ of a program $P$ as the least fixpoint of the abstract transfer $F^\# : S^\# \rightarrow S^\#$:

$$\llbracket P \rrbracket^\# = \lim_{n \to \infty} (F^\#)^n (d^\#)$$

$$F^\#(d^\#) = d^\# \sqcup \left( \bigsqcup_{(\ell, \tau) \in \text{Domain}(m)} \llbracket \text{inst}(\ell) \rrbracket^\#_i (\ell, \tau)(d^\#) \right)$$

where $d^\# = (m, _) \text{ and } d^\#_i$ denotes the initial abstract state.

B. Instructions: $\llbracket i \rrbracket^\#_i : (L \times T^*) \rightarrow S^\# \rightarrow S^\#$

- Variable Declarations:

$$\llbracket \text{let } x = e \rrbracket^\#_i(\ell, \tau)(d^\#) = (\{(\text{next}(\ell), \tau) \mapsto \sigma^\#_x\}, \emptyset)$$

where

$$d^\# = (m, _) \land \sigma^\# = m(\ell, \tau) \land$$

$$\sigma^\#_x = \sigma^\#[x \mapsto \llbracket e \rrbracket^\#_e(\sigma^\#)]$$
JSTAR - Evaluation

- Type Analysis for 864 versions of ECMA-262 in 3 years

---

### Table II: The analysis precision of JSTAR

<table>
<thead>
<tr>
<th>Checker</th>
<th>Bug Kind</th>
<th>Precision = (# True Bugs) / (# Detected Bugs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>no-refine</td>
</tr>
<tr>
<td>Reference</td>
<td>UnknownVar</td>
<td>62 / 106</td>
</tr>
<tr>
<td></td>
<td>DuplicatedVar</td>
<td>45 / 46</td>
</tr>
<tr>
<td>Aritty</td>
<td>MissingParam</td>
<td>4 / 4</td>
</tr>
<tr>
<td>Assertion</td>
<td>Assertion</td>
<td>4 / 56</td>
</tr>
<tr>
<td>Operand</td>
<td>NoNumber</td>
<td>22 / 113</td>
</tr>
<tr>
<td></td>
<td>Abrupt</td>
<td>20 / 48</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>92 / 279</td>
</tr>
</tbody>
</table>

**59.2% Precision**

**93 Bugs Detected**
JSTAR - Evaluation

- Type Analysis for 864 versions of ECMA-262 in 3 years

<table>
<thead>
<tr>
<th>Checker</th>
<th>Bug Kind</th>
<th>Precision = (# True Bugs) / (# Detected Bugs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>no-refine</td>
</tr>
<tr>
<td>Reference</td>
<td>UnknownVar</td>
<td>62 / 106</td>
</tr>
<tr>
<td></td>
<td>DuplicatedVar</td>
<td>45 / 46</td>
</tr>
<tr>
<td>Arity</td>
<td>MissingParam</td>
<td>4 / 4</td>
</tr>
<tr>
<td>Assertion</td>
<td>Assertion</td>
<td>4 / 56</td>
</tr>
<tr>
<td>Operand</td>
<td>NoNumber</td>
<td>2 / 65</td>
</tr>
<tr>
<td></td>
<td>Abrupt</td>
<td>20 / 48</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>92 / 279</td>
</tr>
</tbody>
</table>

Precision = (# True Bugs) / (# Detected Bugs)

<table>
<thead>
<tr>
<th>Name</th>
<th>Feature</th>
<th>#</th>
<th>Checker</th>
<th>Created</th>
<th>Life Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES12-1</td>
<td>Switch</td>
<td>3</td>
<td>Reference</td>
<td>2015-09-22</td>
<td>1,996 days</td>
</tr>
<tr>
<td>ES12-2</td>
<td>Try</td>
<td>3</td>
<td>Reference</td>
<td>2015-09-22</td>
<td>1,996 days</td>
</tr>
<tr>
<td>ES12-3</td>
<td>Arguments</td>
<td>1</td>
<td>Reference</td>
<td>2015-09-22</td>
<td>1,996 days</td>
</tr>
<tr>
<td>ES12-4</td>
<td>Array</td>
<td>2</td>
<td>Reference</td>
<td>2015-09-22</td>
<td>1,996 days</td>
</tr>
<tr>
<td>ES12-5</td>
<td>Async</td>
<td>1</td>
<td>Reference</td>
<td>2015-09-22</td>
<td>1,996 days</td>
</tr>
<tr>
<td>ES12-6</td>
<td>Class</td>
<td>1</td>
<td>Reference</td>
<td>2015-09-22</td>
<td>1,996 days</td>
</tr>
<tr>
<td>ES12-7</td>
<td>Branch</td>
<td>1</td>
<td>Reference</td>
<td>2015-09-22</td>
<td>1,996 days</td>
</tr>
<tr>
<td>ES12-8</td>
<td>Arguments</td>
<td>2</td>
<td>Operand</td>
<td>2015-12-16</td>
<td>1,910 days</td>
</tr>
</tbody>
</table>
Automatically Deriving JavaScript Static Analyzers from Language Specifications
Jihyeok Park, Seungmin An, and Sukyoung Ryu

1. Mechanized Spec. Extraction
[ASE'20] JISET
Mechanized Specification
2. Specification Validity Check
Conformance Test Synthesis

3. Derivation of Static Analyzers
[JSAVER] Derived Static Analyzer
Analysis Result

Type Analysis for Specification

JavaScript Programs

ECMA-262

[JEST] [ICSE'21]

[JSEA] [FSE'22]

[JSTAR] [ASE'21]

[JESTfs] [ICSE'21]
JSAVER - Meta-Level Static Analysis

ECMA-262

JavaScript Program → JS-IR Compiler → IR Program → IR Static Analyzer → Analysis Result

JavaScript Static Analyzer

Compiler-based approach (existing)

manual
JSAVER - Meta-Level Static Analysis

**Compiler-based approach (existing):**
- ECMA-262
- JavaScript Program
- JavaScript Static Analyzer
- JS-IR Compiler
- IR Program
- IR Static Analyzer
- Analysis Result

**Interpreter-based approach (ours):**
- ECMA-262
- JavaScript Program
- JavaScript Static Analyzer
- JS Interpreter (= IR Program)
- IR Static Analyzer
- Analysis Result
Abstract

The highly dynamic nature and complex semantics of JavaScript make it difficult to correctly understand its behaviors of JavaScript programs. To automatically reason about them, researchers have developed various JavaScript static analyzers that conform to ECMAScript, the standard specification of JavaScript.

The JavaScript language itself rapidly evolves with a yearly release cadence and open development process. To perform differential testing, specifica-

crion that automatically derives JavaScript static analyzers from that automatically extracts a mechanized specification from ECMAScript. We show that our approach consists of three steps:

1) mechanized specification
2) JavaScript Parser
3) Analysis Result
JSAVER - Evaluation

- Soundness / Precision / Performance
  - 18,556 applicable tests in Test262
  - 3,903 tests analyzable by all the three analyzers
JSAVER - Evaluation

- **Soundness / Precision / Performance**
  - 18,556 applicable tests in Test262
  - 3,903 tests analyzable by all the three analyzers

(a) Analysis results of TAJS
(b) Analysis results of SAFE
(c) Analysis results of JSAES12
My colleagues and I at Moddable appreciate your work to report so many issues. ... If you don’t mind me asking, my team is curious about the methodology you are using to find these issues. What can you share about that? As background, our primary testing is done with test262. It is an excellent resource but it is not truly comprehensive, as your results reinforce.
Contributing to this Specification

This specification is developed on GitHub with the help of the ECMAScript community. There are a number of ways to contribute to the development of this specification:

GitHub Repository: https://github.com/tc39/ecma262
Issues: All Issues, File a New Issue
Pull Requests: All Pull Requests, Create a New Pull Request
Test Suite: Test262
Editors:
- Shu-yu Guo (@_shu)
- Michael Ficarra (@smooshMap)
- Kevin Gibbons (@bakkoting)
During my time as editor of ECMA-262, one of my primary goals has been to make the specification easier to consume for automated analysis tools such as yours. So I'm very happy to see you've had success. I would like to know what difficulties you had in automatically consuming the specification, and what we can do to make it easier for tools like yours in the future. Have you looked at the changes we've made to the 2021 edition or the 2022 draft yet? There have been some fairly substantial changes. I would also like to know how we could integrate your tooling into our development and editing processes so they can co-evolve.
“Yeah, first of all, I want to, I can hardly express how amazing this work is, this is really impressive. I sat through the presentation with my mouth open the whole time. So thank you very much.”

“First, this is truly amazing work. My mind is blown. I tried to get screenshots, just to remember the slides and then was just taking screenshots of every slide. So I stopped.”

“I think this was an excellent presentation. In terms of committee feedback, what you’re hearing here, this is the committee in ecstatic mode. This is, this is the maximum that I’ve heard in terms of positive feedback for a presentation. So, so thank you very much.”
Presentation from KAIST research group

Presenter: KAIST research group (SAN, JHP, and SRU) & TC39 editor group (MF, KG, and SYG)

- slides
- JavaScript Static Analysis for Evolving Language Specifications: https://www.youtube.com/watch?v=3Ju_jnHB8g (only for the people with the link because the last part of the video is under paper review)
- ECMAScript Debugger: https://www.youtube.com/watch?v=syfZ3v6JNg8 (publicly available)

SRU: Hi everyone. I am SRU. I am a faculty at the School of Computing at KAIST. Today two of my students are going to present our recent work on automatically generating various tools about JavaScript programs from ES, the language specification written in English. So, first JHP is going to present a very brief version of his PhD dissertation, and then SAN is going to show you a pretty cool tool, a debugger for ES.

JHP: I'm JHP, a PhD candidate in the Programming Language Research Group at KAIST. I will briefly introduce our recent work with these slides. I will first explain why we started this work, and which tools we have developed so far.
We are thrilled to share that ESMeta is integrated into the CI of both ECMA-262 and Test262:

https://github.com/tc39/ecma262/pull/2926
https://github.com/tc39/test262/pull/3730
https://github.com/es-meta/esmeta

Now, each ECMA-262 PR will execute the ESMeta type checker, and any new or changed tests in a Test262 PR will be executed using the ESMeta interpreter.

Thank you all for making this happen especially 박지혁!
We are thrilled to share that ESMeta is integrated into the CI of both ECMA-262 and Test262:

https://github.com/tc39/ecma262/pull/2926
https://github.com/tc39/test262/pull/3730
https://github.com/es-meta/esmeta

Now, each ECMA-262 PR will execute the ESMeta type checker, and any new or changed tests in a Test262 PR will be executed using the ESMeta interpreter.

Thank you all for making this happen especially 박지혜!
Abstract

In this thesis, we introduce a novel approach to automatically derive JavaScript static analyzers that conform to ECMAScript, the standard specification of JavaScript. The highly dynamic nature and complex semantics of JavaScript make it difficult to correctly understand and implement static analyzers. We present a tool for mechanized specifications using a meta-level static analysis. For evaluation, we derive a JavaScript analyzer from the latest ECMAScript and show that it successfully analyzes all applicable parts.

1. Mechanized Spec. Extraction
   - ECMA-262
   - [ASE'20]
   - JISET

2. Specification Validity Check
   - Mechanized Specification
   - [FSE'22]
   - JSAVER

3. Derivation of Static Analyzers
   - Derived Static Analyzer
   - Analysis Result
   - JEST
   - [ICSE'21]
   - [FSE'22]
   - JSTAR
   - [ASE'21]

Type Analysis for Specification

Conformance Test Synthesis

- JEST
- [ICSE'21]
- JESTfs
- [PLDI'23]
**JEST [ICSE'21]**

**JavaScript Engines and Specification Tester**

**Coverage-guided Mutation**

- **Seed Synthesizer**
- **JS Programs**
- **Assertion Injector**
- **Program Mutator**

**Mechanized Specification**

**Conformance Bugs**

- **Specification Bugs**
- **Engine Bugs**

**Conformance Tests**

**JavaScript Engines**

**JEST**

---

Abstract

The highly dynamic nature and complex semantics of JavaScript make it difficult to correctly understand its specifications and engines. Conformance testing, which is widely used to detect bugs in specifications and engines, remains a challenge due to the rapid evolution of the language. In this paper, we present JEST (JavaScript Engines and Specification Tester), a tool for detecting bugs in JavaScript specifications and engines. JEST performs coverage-guided mutation on JavaScript programs to generate tests that are expected to fail according to the specifications. We then use a static analyzer to check the validity of the generated specifications and extract mechanized specifications from the latest version of ECMAScript. Our approach consists of three steps: 1) mechanized specification extraction, 2) mechanized specification validity check, and 3) derivation of static analyzers. First, we derive a JavaScript specification from any version of ECMAScript. Then, we present a JavaScript static analyzer from the latest ECMAScript and show that it successfully analyzes all applicable specifications. For evaluation, we derive a JavaScript specification using a meta-level static analysis. We believe that the thesis would be the first step towards the mechanization of JavaScript specifications and engines.
Conformance Testing of PL Implementations

Conformance Tests for L₁

Specification of L₁

Syntax

A ::= BC

[-] Semantics

L₁ Program + Assertion

Compiler
Interpreter
Static Analyzer
Symbolic Executor
...

Implementations of L₁

Conformance

Programming Language L₁

Quality?
Graph Coverage for Language Specification

- Specification of $L_1$
- Extract
- Mechanized Specification
- Build
- Control-flow Graph (CFG)
- Conformance Tests for $L_1$
- Quality of Conformance Tests
- Measure
- Graph Coverage
  - Node Coverage
  - Branch Coverage
  - Prime Path Coverage
  - Def-Use Coverage
  - ... Coverage Criteria

Test Requirements (TRs)

Problem: Are they sufficient?
Normal algorithms

**EvaluateStringOrNumericBinaryExpression** (leftOperand, opText, rightOperand)
7...
5. Return ?^9 ApplyStringOrNumericBinaryOperator^8 (lval, opText, rval).^10

**ApplyStringOrNumericBinaryOperator** (lval, opText, rval)
11...
4. Let rnum be ?^15 ToNumeric^14 (rval).
5. If Type(inum) is different from Type(rnum)^16, throw a TypeError exception. ^17

**ToNumeric** (value)
19...
2. If Type(trimValue) is BigInt^20, return trimValue. ^21

Fig. 3. Three normal algorithms transitively used in the semantics of AdditiveExpression in ES13

Built-in methods

**Number** (value)
1. If value is present^23, then
   a. Let prim be ?^25 ToNumeric^24 (value).

Fig. 4. Built-in method Number in ES13
**k-Feature-Sensitive (k-FS) Coverage**

- **k-Feature-Sensitive (k-FS) coverage criterion divides** the given TRs with the at most k-innermost enclosing language features.

**Implementation — JESTfs**

- **JEST** [1] is a JavaScript conformance test generator using Coverage-Guided Fuzzing.
- We implemented JESTfs as an extension of JEST with k-FS and k-FCPS coverage criteria.

**RQ2) Effectiveness of k-FS Coverage Criteria**

<table>
<thead>
<tr>
<th>Coverage Criteria Ck</th>
<th># Covered k-FCPS-TR (k)</th>
<th># Syn. Test</th>
<th># Bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-FS node-or-branch (0-fs)</td>
<td>10.0</td>
<td>5.6</td>
<td>15.6</td>
</tr>
<tr>
<td>1-FS node-or-branch (1-fs)</td>
<td>79.3</td>
<td>45.7</td>
<td>122.0</td>
</tr>
<tr>
<td>1-FCPS node-or-branch (1-FCPS)</td>
<td>179.7</td>
<td>97.6</td>
<td>277.3</td>
</tr>
<tr>
<td>2-FS node-or-branch (2-fs)</td>
<td>1,199.8</td>
<td>696.3</td>
<td>1,896.1</td>
</tr>
<tr>
<td>2-FCPS node-or-branch (2-FCPS)</td>
<td>2,323.1</td>
<td>1,297.6</td>
<td>3,620.7</td>
</tr>
</tbody>
</table>

**RQ3) Effectiveness of k-FCPS Coverage Criteria**

<table>
<thead>
<tr>
<th>Coverage Criteria Ck</th>
<th># Covered k-FCPS-TR (k)</th>
<th># Syn. Test</th>
<th># Bug</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-FS node-or-branch (0-fs)</td>
<td>10.0</td>
<td>5.6</td>
<td>15.6</td>
</tr>
<tr>
<td>1-FS node-or-branch (1-fs)</td>
<td>79.3</td>
<td>45.7</td>
<td>122.0</td>
</tr>
<tr>
<td>1-FCPS node-or-branch (1-FCPS)</td>
<td>179.7</td>
<td>97.6</td>
<td>277.3</td>
</tr>
<tr>
<td>2-FS node-or-branch (2-fs)</td>
<td>1,199.8</td>
<td>696.3</td>
<td>1,896.1</td>
</tr>
<tr>
<td>2-FCPS node-or-branch (2-FCPS)</td>
<td>2,323.1</td>
<td>1,297.6</td>
<td>3,620.7</td>
</tr>
</tbody>
</table>

# RQ1) Conformance Bug Detection

<table>
<thead>
<tr>
<th>Kind</th>
<th>Name</th>
<th>Version</th>
<th>Release</th>
<th># Detected Unique Bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td># New</td>
</tr>
<tr>
<td>Engine</td>
<td>V8</td>
<td>v10.8.121</td>
<td>2022.10.06</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>JSC</td>
<td>v615.1.10</td>
<td>2022.10.26</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>GraalJS</td>
<td>v22.2.0</td>
<td>2022.07.26</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>SpiderMonkey</td>
<td>v107.0b4</td>
<td>2022.10.24</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td>25</td>
</tr>
<tr>
<td>Transpiler</td>
<td>Babel</td>
<td>v7.19.1</td>
<td>2022.09.15</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>SWC</td>
<td>v1.3.10</td>
<td>2022.10.21</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>Terser</td>
<td>v5.15.1</td>
<td>2022.10.05</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Obfuscator</td>
<td>v4.0.0</td>
<td>2022.02.15</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td>58</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td>83</td>
</tr>
</tbody>
</table>
Coherence can be bypassed by an indirect impl for a trait object #57893

arielb1 opened this issue on Jan 25, 2019 · 50 comments

arielb1 commented on Jan 25, 2019 · edited by lcnr

Comments

The check for manual impl Object for Object only makes sure there is no direct impl Object for dyn Object - it does not consider such indirect impls. Therefore, you can write a blanket impl<T: ?Sized> Object for T that conflicts with the builtin impl Object for dyn Object.
The Verse Calculus: a Core Calculus for Functional Logic Programming

LENNART AUGUSTSSON, Epic Games, Sweden
JOACHIM BREITNER
KOEN CLAESSEN, Epic Games, Sweden
RANJIT JHALA, Epic Games, USA
SIMON PEYTON JONES, Epic Games, United Kingdom
OLIN SHIVERS, Epic Games, USA
TIM SWEENEY, Epic Games, USA

Functional logic languages have a rich literature, but it is tricky to give them a satisfying semantics. In this paper we describe the Verse calculus, VC, a new core calculus for functional logical programming. Our main contribution is to equip VC with a small-step rewrite semantics, so that we can reason about a VC program in the same way as one does with lambda calculus; that is, by applying successive rewrites to it.

This draft paper describes our current thinking about Verse. It is very much a work in progress, not a finished product. The broad outlines of the design are stable. However, the details of the rewrite rules may well change; we think that the current rules are not confluent, in tiresome ways. (If you are knowledgeable about confluence proofs, please talk to us!)

We are eager to engage in a dialogue with the community. Please do write to us.
WebAssembly?

**t.binop**

1. Assert: due to validation, two values of value type $t$ are on the top of the stack.
2. Pop the value $t.\ const\ c_2$ from the stack.
3. Pop the value $t.\ const\ c_1$ from the stack.
4. If $binop_t(c_1, c_2)$ is defined, then:
   a. Let $c$ be a possible result of computing $binop_t(c_1, c_2)$.
   b. Push the value $t.\ const\ c$ to the stack.
5. Else:
   a. Trap.

\[
(t.\ const\ c_1) \ (t.\ const\ c_2) \ t.\ binop \ \rightarrow \ (t.\ const\ c) \ (\text{if } c \in binop_t(c_1, c_2))
\]
\[
(t.\ const\ c_1) \ (t.\ const\ c_2) \ t.\ binop \ \rightarrow \ \text{trap} \ (\text{if } binop_t(c_1, c_2) = \{\})
\]
Dagstuhl Seminar 23101
Foundations of WebAssembly
(Mar 05 – Mar 10, 2023)

Permalink
Please use the following short url to reference this page: https://www.dagstuhl.de/23101

Organizers
- Karthikeyan Bhargavan (INRIA - Paris, FR)
- Jonathan Protzenko (Microsoft - Redmond, US)
- Andreas Rossberg (München, DE)
- Deian Stefan (University of California - San Diego, US)

Contact
- Andreas Dolzmann (for scientific matters)
- Christina Schwarz (for administrative matters)
timeline

2018 Wasm 1.0
  focus on low-level languages to get off the ground quickly

2022 Wasm 2.0
  multiple values, vector types, reference types, table ops, bulk ops, multiple tables

2023 Wasm 2.1
  tail calls, relaxed vector ops, deterministic profile

2024 Wasm 2.2+
  atomics, exceptions, typed references, garbage collection, 64-bit addresses, multiple memories

202X Wasm 2.X
  stack switching, threading, memory control, type imports,

https://github.com/WebAssembly/proposals
2018 Wasm 1.0
focus on low-level languages to get off the ground quickly

2022 Wasm 2.0
multiple values, vector types, reference types, tables

2023 Wasm 2.1
tail calls, relaxed vector ops, deterministic profile

2024 Wasm 2.2+
atomics, exceptions, typed references, garbage collection

202X Wasm 2.X
stack switching, threading, memory control, type checking

---

**Problem: Fast Evolving JavaScript**

- **1997 - ES1** First edition
- **1999 - ES3** RegEx, String, Try/catch, etc
- **2004**
- **2009 - ES5** getters/setters, strict mode, exceptions, etc
- **2011 - ES5.1** Editorial Changes
- **2015 - ES6** classes, modules, etc.
- **2017 - ES8** object manipulation, etc.
- **2019 - ES10**
- **2021 - ES12**
- **2022**

---

**ECMAScript 2021 (ES12) - 879 pages**
**t. binop**

1. Assert: due to validation, two values of value type \( t \) are on the top of the stack.
2. Pop the value \( t.\ const\ c_2 \) from the stack.
3. Pop the value \( t.\ const\ c_1 \) from the stack.
4. If \( \text{binop}_t(c_1, c_2) \) is defined, then:
   a. Let \( c \) be a possible result of computing \( \text{binop}_t(c_1, c_2) \).
   b. Push the value \( t.\ const\ c \) to the stack.
5. Else:
   a. Trap.

\[
\begin{align*}
(t.\ const\ c_1) (t.\ const\ c_2)\ t.\ binop & \to (t.\ const\ c) \quad (\text{if } c \in \text{binop}_t(c_1, c_2)) \\
(t.\ const\ c_1) (t.\ const\ c_2)\ t.\ binop & \to \text{trap} \quad (\text{if } \text{binop}_t(c_1, c_2) = \{\})
\end{align*}
\]

---

120  rule Step_pure/binop-val:
121  (CONST nt c_1) (CONST nt c_2) (BINOP nt binop) \( \rightsquigarrow \) (CONST nt c)
122  -- if \( \$\text{binop}(\text{binop}, \text{nt}, c_1, c_2) = c \) ;; TODO
123
124  rule Step_pure/binop-trap:
125  (CONST nt c_1) (CONST nt c_2) (BINOP nt binop) \( \rightsquigarrow \) TRAP
126  -- if \( \$\text{binop}(\text{binop}, \text{nt}, c_1, c_2) = \text{epsilon} \) ;; TODO
1. Assert: due to validation, two values of value type `t` are on the top of the stack.
2. Pop the value `t.const c2` from the stack.
3. Pop the value `t.const c1` from the stack.
4. If `binop_t(c1, c2)` is defined, then:
   a. Let `c` be a possible result of computing `binop_t(c1, c2)`.
   b. Push the value `t.const c` to the stack.
5. Else:
   a. Trap.

\[
\begin{align*}
(t.const \ c1) \ (t.const \ c2) \ t.binop \ &\mapsto \ (t.const \ c) \quad (\text{if} \ c \in \ binop_t(c1, c2)) \\
(t.const \ c1) \ (t.const \ c2) \ t.binop \ &\mapsto \ \text{trap} \quad (\text{if} \ binop_t(c1, c2) = \{\})
\end{align*}
\]

```
$ (cd ../spec && dune exec ../src/exe-watsup/main.exe -- *.watsup -v -l --sideconditions --animate --prose)
watsup 0.3 generator
== Parsing...
== Elaboration...
== IL Validation...
```
1.1.2 \textbf{binop} \textit{nt} \textit{binop}

1. Assert: Due to validation, a value of value type \textit{nt} is on the top of the stack.

2. Pop const \textit{nt} \textit{c}_2 from the stack.

3. Assert: Due to validation, a value of value type \textit{nt} is on the top of the stack.

4. Pop const \textit{nt} \textit{c}_1 from the stack.

5. Let \( r_0 \) be the result of computing \( \text{binop}(\text{binop}, \textit{nt}, \textit{c}_1, \textit{c}_2) \).

6. If the length of \( r_0 \) is 1, then:
   
   a. Let \([c]\) be the result of computing \( \text{binop}(\text{binop}, \textit{nt}, \textit{c}_1, \textit{c}_2) \).
   
   b. Push const \textit{nt} \textit{c} to the stack.

7. Let \( r_1 \) be the result of computing \( \text{binop}(\text{binop}, \textit{nt}, \textit{c}_1, \textit{c}_2) \).

8. If \( r_1 \) is [], then:
   
   a. Trap.

\[
\begin{align*}
\text{[E-BINOP-VAL]} & (nt.\text{const } c_1) (nt.\text{const } c_2) (nt.\text{binop}) \iff (nt.\text{const } c) \quad \text{if} \ binop_{\text{nt}}(c_1, c_2) = c \\
\text{[E-BINOP-TRAP]} & (nt.\text{const } c_1) (nt.\text{const } c_2) (nt.\text{binop}) \iff \text{trap} \quad \text{if} \ binop_{\text{nt}}(c_1, c_2) = \epsilon
\end{align*}
\]
Abstract

P4 is a language for programming the data plane of network devices. This document provides a precise definition of the P4\textsubscript{16} language, which is the 2016 revision of the P4 language (http://p4.org). The target audience for this document includes developers who want to write compilers, simulators, IDEs, and debuggers for P4 programs. This document may also be of interest to P4 programmers who are interested in understanding the syntax and semantics of the language at a deeper level.
Reviewer #2

This is the right order to design and document languages:

- first the semantics,
- then the implementation
- and documentation, ideally generated from the semantics.

Automatic Derivation of Static Analysis
"If you go and talk to an old person and he or she says, “We tried that 25 years ago and it didn’t work,” don’t take that for an answer, ask why it didn’t work and what’s different today.”

“Keep the big picture in mind; look for opportunities to bridge disciplines rather than rat holding on just one; don’t settle for just a 5% improvement, aim big; and finally read read read voraciously, read papers, read code, read books, absorb as much information as you can, you never can tell what you learned five years ago might suddenly be relevant now.”