Towards Tractable Verification of JavaScript Programs

Daiva Naudžiūnienė
Imperial College London

The Goal

Static Analysis / Verification Tool for JavaScript

Today’s talk

- Difficulties with static analysis tool directly for JavaScript
- JS-IVL, an intermediate verification language for JavaScript
- Translation from JavaScript to JS-IVL

An Example

```
'use strict';
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

obj.m();
```

An Example

```
'use strict';
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

obj.m();
```

false

Modified Example

```
'use strict';
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

var f = obj.m;
f();
```

Modified Example

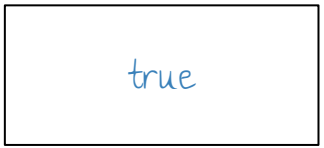
```
'use strict';
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

var f = obj.m;
f();
```

Modified Example in Non-Strict Mode

```
'use strict';
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

var f = obj.m;
f();
```

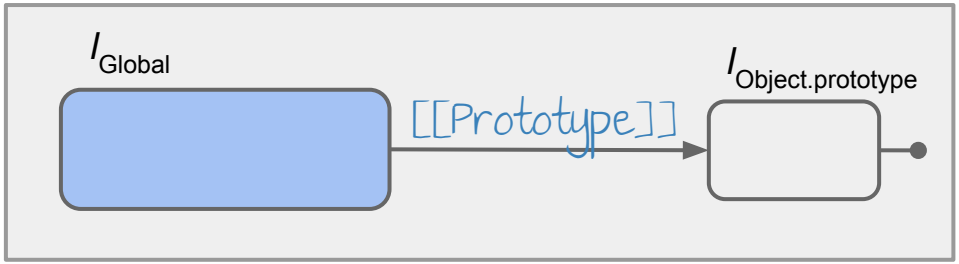
Modified Example in Non-Strict Mode

```
'use strict';
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

var f = obj.m;
f();
```



An Example

```
'use strict';
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

obj.m();
```

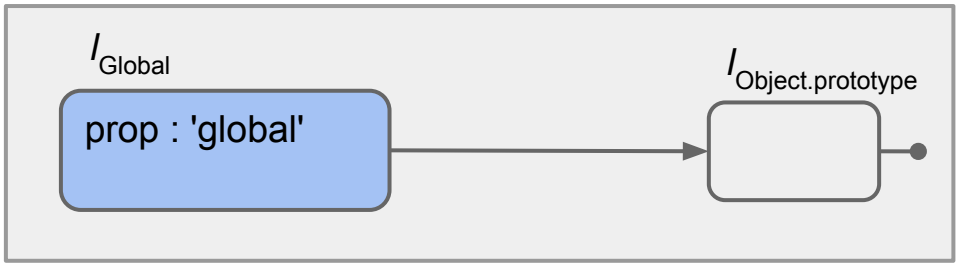


scope = [l_{Global}]

this = l_{Global}

An Example

```
'use strict';
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

obj.m();
```

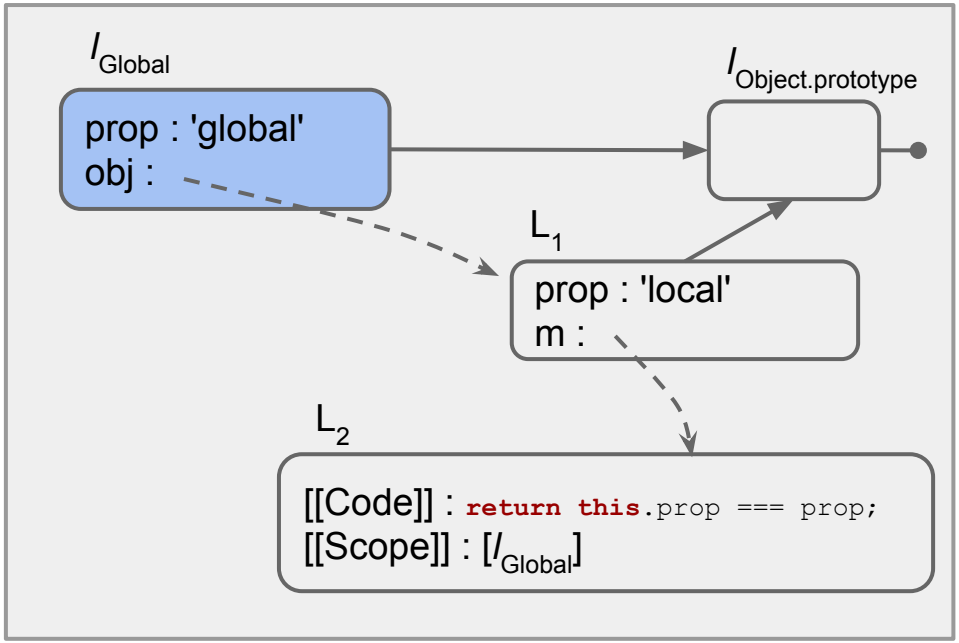


scope = [l_{Global}]

this = l_{Global}

An Example

```
'use strict';
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

obj.m();
```

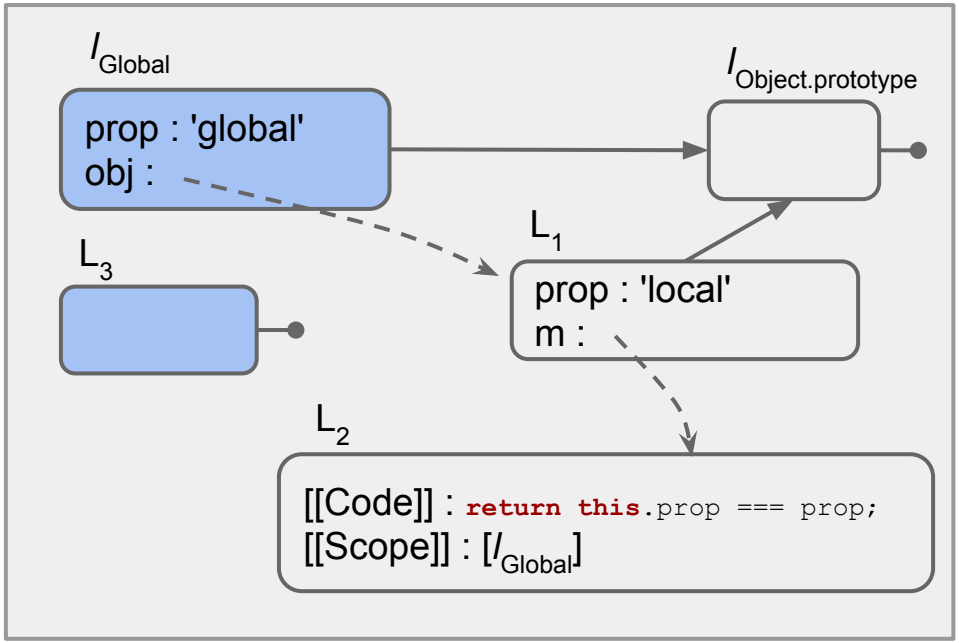


scope = [l_{Global}]

this = l_{Global}

An Example

```
'use strict';
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

obj.m();
```

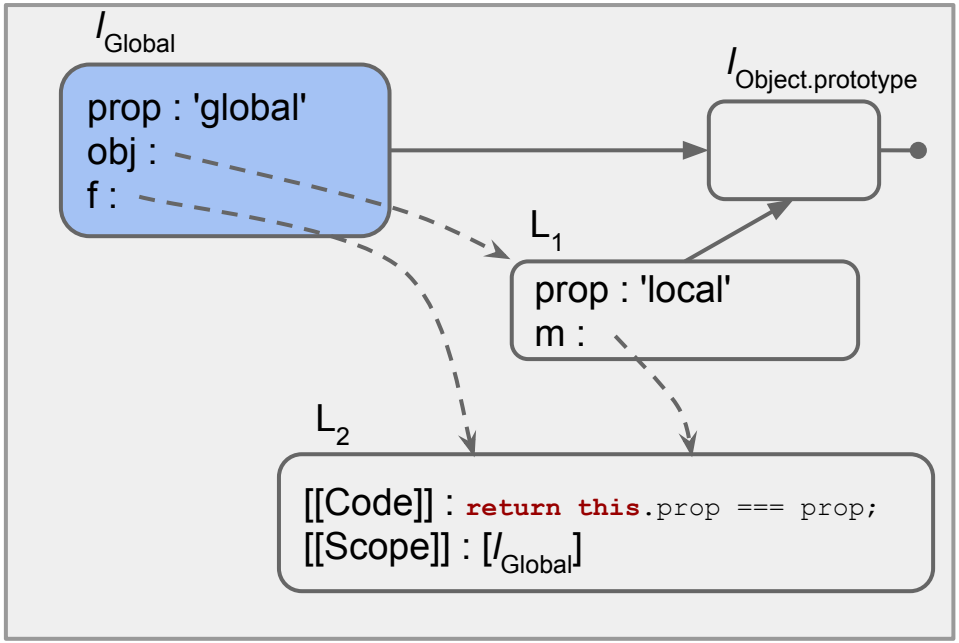


scope = [L_3; l_{Global}]

this = L_1

Modified Example

```
'use strict';
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

var f = obj.m;
f();
```

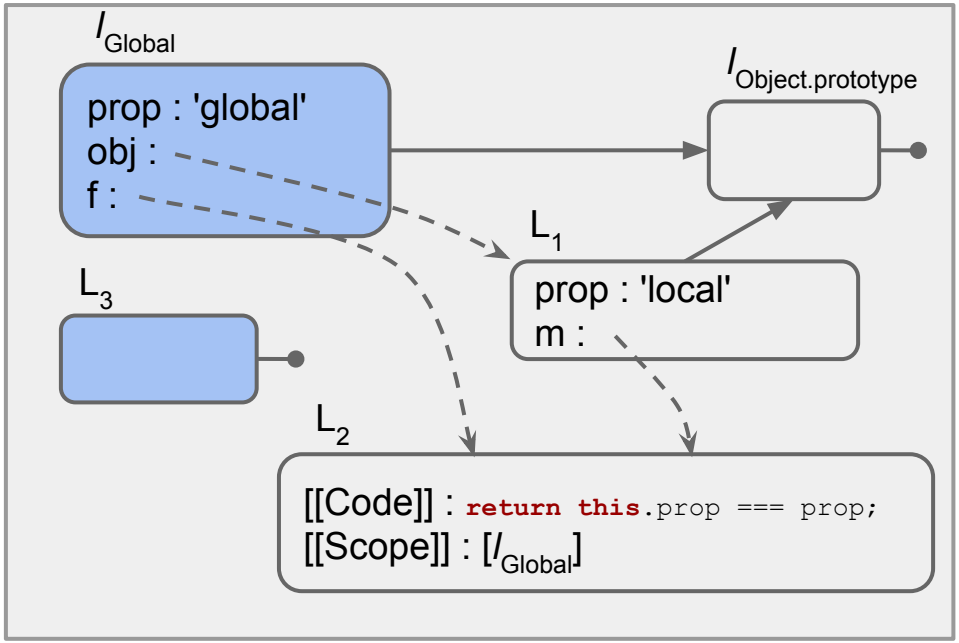


scope = [l_{Global}]

this = l_{Global}

Modified Example

```
'use strict';
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

var f = obj.m;
f();
```

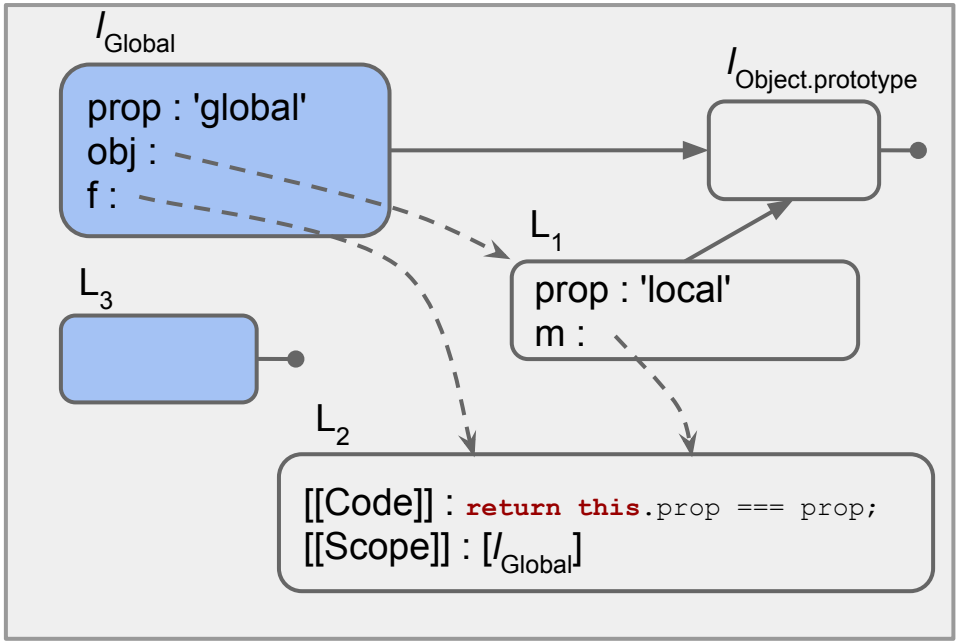


scope = [L_3 ; l_{Global}]

this = `undefined`

Modified Example in Non-Strict Mode

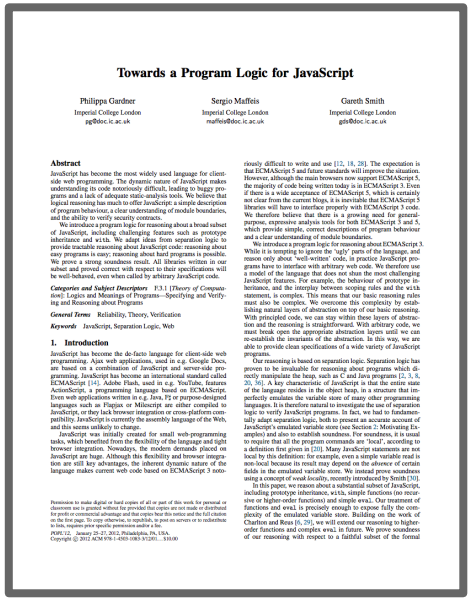
```
'use strict';
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

var f = obj.m;
f();
```



scope = [L_3 ; l_{Global}]

this = l_{Global}

Program Logic for JavaScript



Towards a Program Logic for JavaScript

Philippa Gardner
Imperial College London
pg@doc.ic.ac.uk

Sergio Maffeis
Imperial College London
rmaffeis@doc.ic.ac.uk

Gareth Smith
Imperial College London
gds@doc.ic.ac.uk

Abstract

JavaScript has become the most widely used language for client-side web programming. The dynamic nature of JavaScript makes understanding its code notoriously difficult, leading to buggy programs and a lack of adequate static-analysis tools. We believe that logical reasoning has much to offer JavaScript: a simple description of program behaviour, a clear understanding of module boundaries, and the ability to verify security contracts.

We introduce a program logic for reasoning about a broad subset of JavaScript, including challenging features such as prototype inheritance and `with`. We adapt ideas from separation logic to provide tractable reasoning about JavaScript code: reasoning about easy programs is easy; reasoning about hard programs is possible. We prove a strong soundness result. All libraries written in our subset and proved correct with respect to their specifications will be well-behaved, even when called by arbitrary JavaScript code.

Categories and Subject Descriptors F.3.1 [*Theory of Computation*]: Logics and Meanings of Programs—Specifying and Verifying and Reasoning about Programs

General Terms Reliability, Theory, Verification

Keywords JavaScript, Separation Logic, Web

1. Introduction

JavaScript has become the de-facto language for client-side web programming. Ajax web applications, used in e.g. Google Docs, are based on a combination of JavaScript and server-side programming. JavaScript has become an international standard called ECMAScript [14]. Adobe Flash, used in e.g. YouTube, features ActionScript, a programming language based on ECMAScript. Even web applications written in e.g. Java, Pj or purpose-designed languages such as Flapjax or Milescript are either compiled to JavaScript, or they lack browser integration or cross-platform compatibility. JavaScript is currently the assembly language of the Web, and this seems unlikely to change.

JavaScript was initially created for small web-programming tasks, which benefited from the flexibility of the language and tight browser integration. Nowadays, the modern demands placed on JavaScript are huge. Although this flexibility and browser integration are still key advantages, the inherent dynamic nature of the language makes current web code based on ECMAScript 3 notoriously difficult to write and use [12, 18, 28]. The expectation is that ECMAScript 5 and future standards will improve the situation. However, although the main browsers now support ECMAScript 5, the majority of code being written today is in ECMAScript 3. Even if there is a wide acceptance of ECMAScript 5, which is certainly not clear from the current blogs, it is inevitable that ECMAScript 5 libraries will have to interface properly with ECMAScript 3 code. We therefore believe that there is a growing need for general-purpose, expressive analysis tools for both ECMAScript 3 and 5, which provide simple, correct descriptions of program behaviour and a clear understanding of module boundaries.

We introduce a program logic for reasoning about ECMAScript 3. While it is tempting to ignore the 'ugly' parts of the language, and reason only about 'well-written' code, in practice JavaScript programs have to interface with arbitrary web code. We therefore use a model of the language that does not shun the most challenging JavaScript features. For example, the behaviour of prototype inheritance, and the interplay between scoping rules and the `with` statement, is complex. This means that our basic reasoning rules must also be complex. We overcome this complexity by establishing natural layers of abstraction on top of our basic reasoning. With principled code, we can stay within these layers of abstraction and the reasoning is straightforward. With arbitrary code, we must break open the appropriate abstraction layers until we can re-establish the invariants of the abstraction. In this way, we are able to provide clean specifications of a wide variety of JavaScript programs.

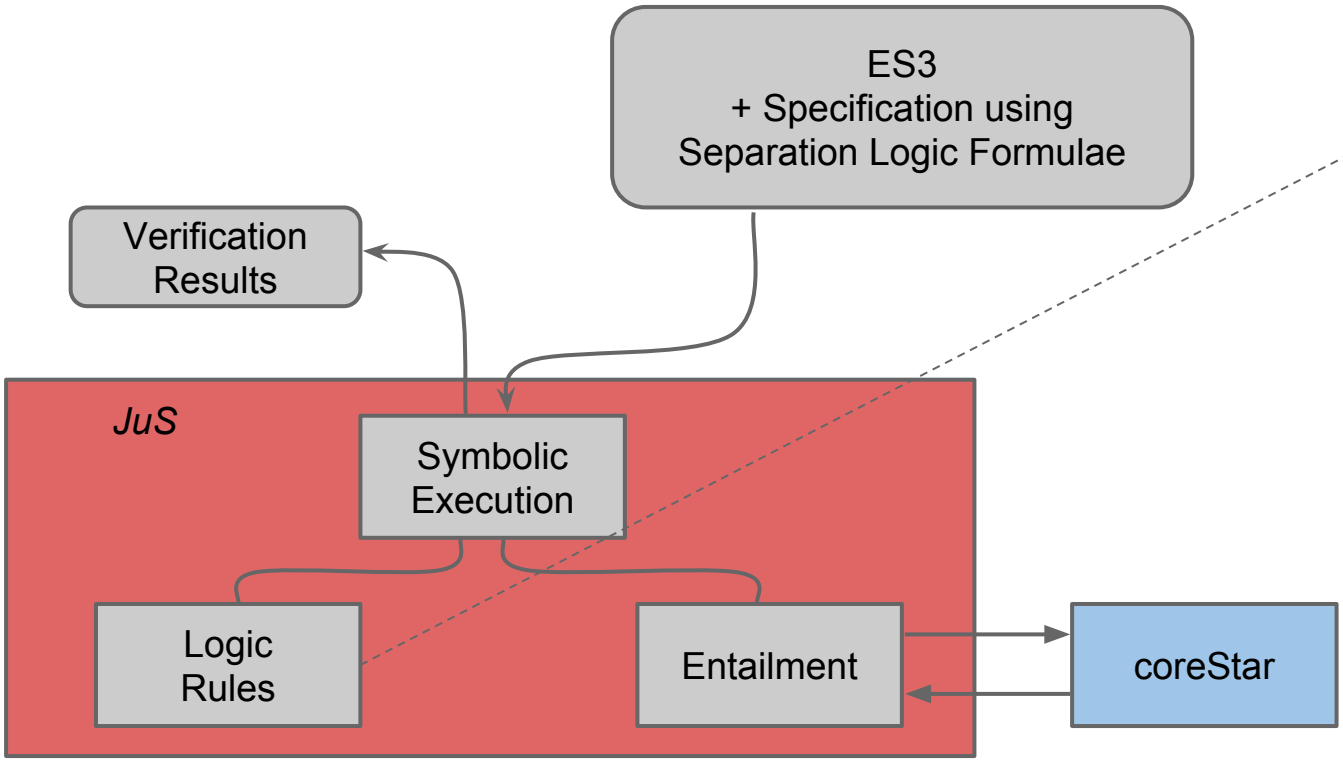
Our reasoning is based on separation logic. Separation logic has proven to be invaluable for reasoning about programs which directly manipulate the heap, such as C and Java programs [2, 3, 6, 20, 36]. A key characteristic of JavaScript is that the entire state of the language resides in the object heap, in a structure that imperfectly emulates the variable store of many other programming languages. It is therefore natural to investigate the use of separation logic to verify JavaScript programs. In fact, we had to fundamentally adapt separation logic, both to present an accurate account of JavaScript's emulated variable store (see Section 2: Motivating Examples) and also to establish soundness. For soundness, it is usual to require that all the program commands are 'local', according to a definition first given in [20]. Many JavaScript statements are not local by this definition: for example, even a simple variable read is non-local because its result may depend on the absence of certain fields in the emulated variable store. We instead prove soundness using a concept of *weak locality*, recently introduced by Smith [30].

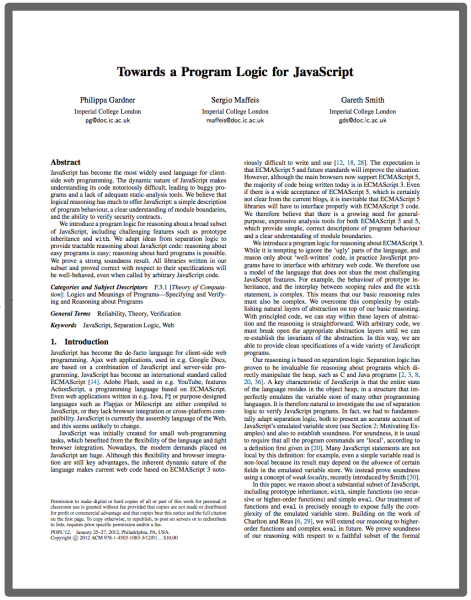
In this paper, we reason about a substantial subset of JavaScript, including prototype inheritance, `with`, simple functions (no recursive or higher-order functions) and simple `eval`. Our treatment of functions and `eval` is precisely enough to expose fully the complexity of the emulated variable store. Building on the work of Charlton and Reus [6, 29], we will extend our reasoning to higher-order functions and complex `eval`s in future. We prove soundness of our reasoning with respect to a faithful subset of the formal

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

Towards a Program Logic for JavaScript. Gardner, Maffeis, Smith. POPL'12

The JuS Tool





Towards a Program Logic for JavaScript

Philippa Gardner
Imperial College London
pg@doc.ic.ac.uk

Sergio Maffeis
Imperial College London
maffeis@doc.ic.ac.uk

Gareth Smith
Imperial College London
gds@doc.ic.ac.uk

Abstract

JavaScript has become the most widely used language for client-side web programming. The dynamic nature of JavaScript makes understanding its code notoriously difficult, leading to buggy programs and a lack of adequate static-analysis tools. We believe that logical reasoning has much to offer JavaScript; a simple description of program behaviour, a clear understanding of module boundaries, and the ability to verify security contracts.

We introduce a program logic for reasoning about a broad subset of JavaScript, including challenging features such as prototype inheritance and `with`. We adapt ideas from separation logic to provide tractable reasoning about JavaScript code: reasoning about easy programs is easy; reasoning about hard programs is possible. We prove a strong soundness result. All libraries written in our subset are proved correct with respect to their specifications will be well-behaved, even when called by arbitrary JavaScript code.

Categories and Subject Descriptors F.3.1 [*Theory of Computation*]: Logics and Meanings of Programs—Specifying and Verifying and Reasoning about Programs

General Terms Reliability, Theory, Verification

Keywords JavaScript, Separation Logic, Web

1. Introduction

JavaScript has become the de-facto language for client-side web programming. Ajax web applications, used in e.g. Google Docs, are based on a combination of JavaScript and server-side programming. JavaScript has become an international standard called ECMAScript [14]. Adobe Flash, used in e.g. YouTube, features ActionScript, a programming language based on ECMAScript. Even web applications written in e.g. Java, Pj or purpose-designed languages such as Pagius or Milescript are either compiled to JavaScript, or they lack browser integration or cross-platform compatibility. JavaScript is currently the assembly language of the Web, and this seems unlikely to change.

JavaScript was initially created for small web-programming tasks, which benefited from the flexibility of the language and tight browser integration. Nowadays, the modern demands placed on JavaScript are huge. Although this flexibility and browser integration are still key advantages, the inherent dynamic nature of the language makes current web code based on ECMAScript 3 notoriously difficult to write and use [12, 18, 28]. The expectation is that ECMAScript 5 and future standards will improve the situation. However, although the main browsers now support ECMAScript 5, the majority of code being written today is in ECMAScript 3. Even if there is a wide acceptance of ECMAScript 5, which is certainly not clear from the current blogs, it is inevitable that ECMAScript 5 libraries will have to interface properly with ECMAScript 3 code. We therefore believe that there is a growing need for general-purpose, expressive analysis tools for both ECMAScript 3 and 5, which provide simple, correct descriptions of program behaviour and a clear understanding of module boundaries.

We introduce a program logic for reasoning about ECMAScript 3. While it is tempting to ignore the 'ugly' parts of the language, and reason only about 'well-written' code, in practice JavaScript programs have to interface with arbitrary web code. We therefore use a model of the language that does not shun the most challenging JavaScript features. For example, the behaviour of prototype inheritance, and the interplay between scoping rules and the `with` statement, is complex. This means that our basic reasoning rules must also be complex. We overcome this complexity by establishing natural layers of abstraction on top of our basic reasoning. With principled code, we can stay within these layers of abstraction and the reasoning is straightforward. With arbitrary code, we must break open the appropriate abstraction layers until we can re-establish the invariants of the abstraction. In this way, we are able to provide clean specifications of a wide variety of JavaScript programs.

Our reasoning is based on separation logic. Separation logic has proven to be invaluable for reasoning about programs which directly manipulate the heap, such as C and Java programs [2, 3, 6, 20, 36]. A key characteristic of JavaScript is that the entire state of the language resides in the object heap, in a structure that imperfectly emulates the variable store of many other programming languages. It is therefore natural to investigate the use of separation logic to verify JavaScript programs. In fact, we had to fundamentally adapt separation logic, both to present an accurate account of JavaScript's emulated variable store (see Section 2: Motivating Examples) and also to establish soundness. For soundness, it is usual to require that all the program commands are 'local', according to a definition first given in [20]. Many JavaScript statements are not local by this definition: for example, even a simple variable read is non-local because its result may depend on the *absence* of certain fields in the emulated variable store. We instead prove soundness using a concept of *weak locality*, recently introduced by Smith [30].

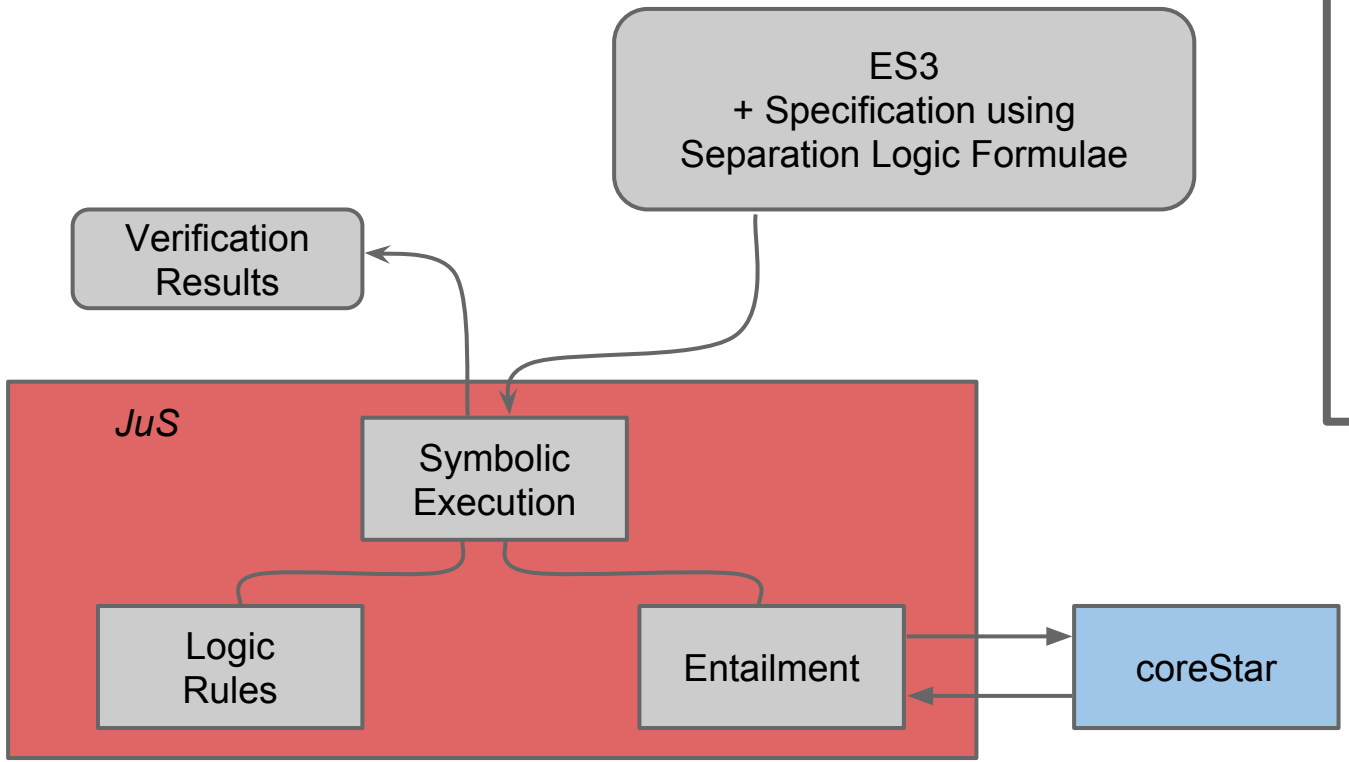
In this paper, we reason about a substantial subset of JavaScript, including prototype inheritance, `with`, simple functions (no recursive or higher-order functions) and simple `eval`. Our treatment of functions and `eval` is precisely enough to expose fully the complexity of the emulated variable store. Building on the work of Charlton and Reus [6, 29], we will extend our reasoning to higher-order functions and complex `eval` in future. We prove soundness of our reasoning with respect to a faithful subset of the formal

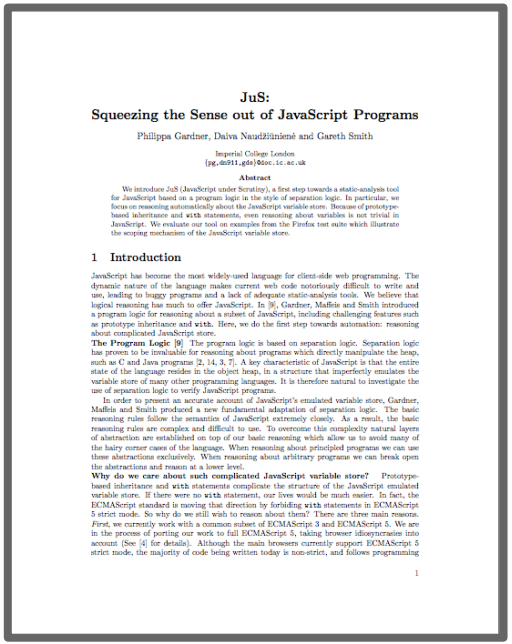
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1083-3/12/01...$10.00

Towards a Program Logic for JavaScript. Gardner, Maffeis, Smith. POPL'12

The JuS Tool





JuS:
Squeezing the Sense out of JavaScript Programs

Philippa Gardner, Daiva Naudžiūnienė and Gareth Smith

Imperial College London
{pg,dn911,gds}@doc.ic.ac.uk

Abstract

We introduce JuS (JavaScript under Scrutiny), a first step towards a static-analysis tool for JavaScript based on a program logic in the style of separation logic. In particular, we focus on reasoning automatically about the JavaScript variable store. Because of prototype-based inheritance and `with` statements, even reasoning about variables is not trivial in JavaScript. We evaluate our tool on examples from the Firefox test suite which illustrate the scoping mechanism of the JavaScript variable store.

1 Introduction

JavaScript has become the most widely-used language for client-side web programming. The dynamic nature of the language makes current web code notoriously difficult to write and use, leading to buggy programs and a lack of adequate static-analysis tools. We believe that logical reasoning has much to offer JavaScript. In [9], Gardner, Maffeis and Smith introduced a program logic for reasoning about a subset of JavaScript, including challenging features such as prototype inheritance and `with`. Here, we do the first step towards automation: reasoning about complicated JavaScript store.

The Program Logic [9] The program logic is based on separation logic. Separation logic has proven to be invaluable for reasoning about programs which directly manipulate the heap, such as C and Java programs [2, 14, 3, 7]. A key characteristic of JavaScript is that the entire state of the language resides in the object heap, in a structure that imperfectly emulates the variable store of many other programming languages. It is therefore natural to investigate the use of separation logic to verify JavaScript programs.

In order to present an accurate account of JavaScript's emulated variable store, Gardner, Maffeis and Smith produced a new fundamental adaptation of separation logic. The basic reasoning rules follow the semantics of JavaScript extremely closely. As a result, the basic reasoning rules are complex and difficult to use. To overcome this complexity natural layers of abstraction are established on top of our basic reasoning which allow us to avoid many of the hairy corner cases of the language. When reasoning about principled programs we can use these abstractions exclusively. When reasoning about arbitrary programs we can break open the abstractions and reason at a lower level.

Why do we care about such complicated JavaScript variable store? Prototype-based inheritance and `with` statements complicate the structure of the JavaScript emulated variable store. If there were no `with` statement, our lives would be much easier. In fact, the ECMAScript standard is moving that direction by forbiding `with` statements in ECMAScript 5 strict mode. So why do we still wish to reason about them? There are three main reasons. *First*, we currently work with a common subset of ECMAScript 3 and ECMAScript 5. We are in the process of porting our work to full ECMAScript 5, taking browser idiosyncrasies into account (See [4] for details). Although the main browsers currently support ECMAScript 5 strict mode, the majority of code being written today is non-strict, and follows programming

1

JuS: Squeezing the Sense out of JavaScript Programs. Gardner, Naudziuniene, Smith. JSTools@ECOOP'13

Takeaway from JuS

- JuS executes JS programs symbolically and checks if given specification is correct

- However, VERY complex Logic Rules do not scale
  - to implement **Bi-Abduction** to reduce annotations size
  - to support bigger and fuller subset of JavaScript

ES3 Logic Rule for Function Call

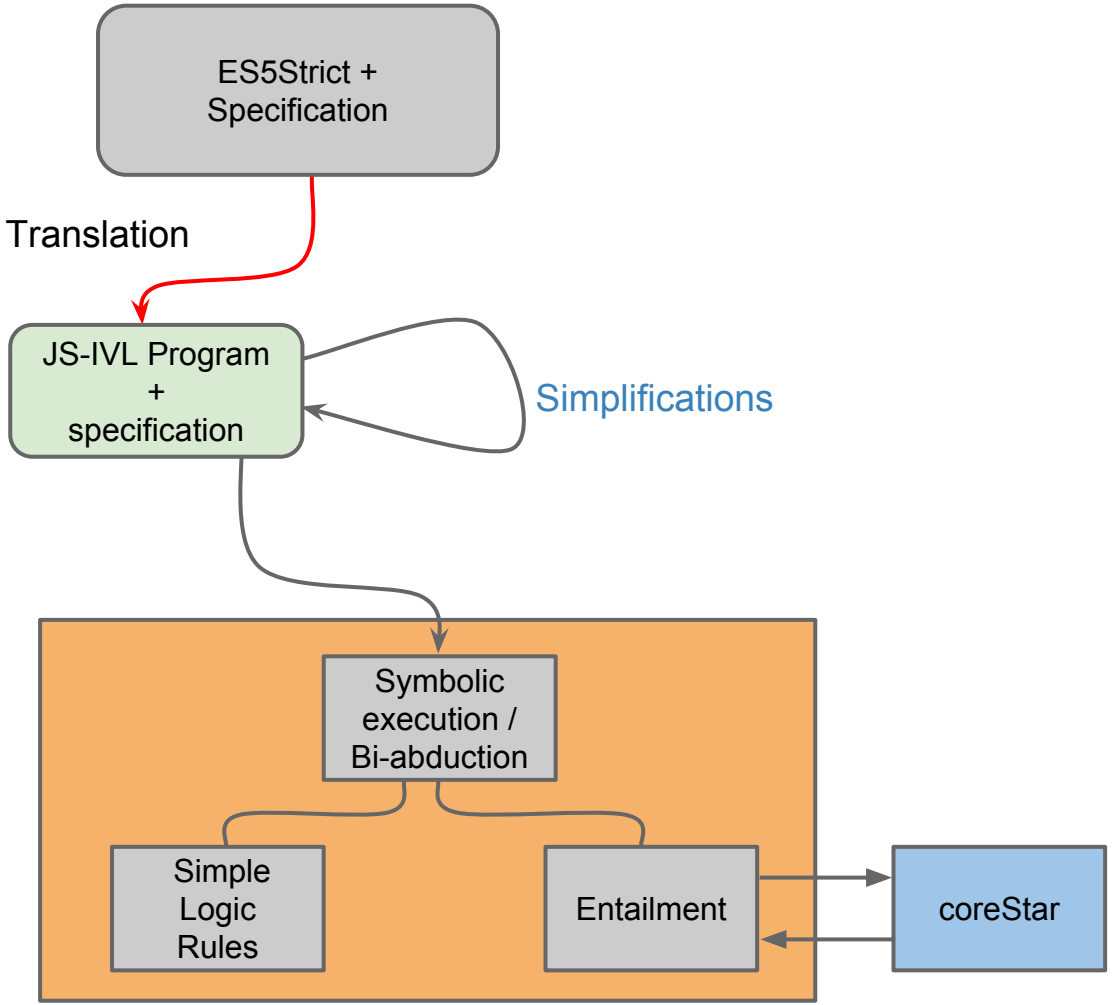
$$
\frac{
\begin{array}{l}
(1)\ \mathcal{Asm} \vdash \{P\}\ \mathtt{e}\ \{R_0 * \mathtt{r} \doteq F_1\} \\
(2)\ R_0 = \begin{pmatrix} S_0 \bowtie \mathsf{This}(F_1, T) \bowtie \gamma(Ls, F_1, F_2) * F_2 \neq l_e * \\ (F_2, @body) \mapsto \lambda X_1, \ldots, X_n.\mathtt{e}' * (F_2, @scope) \mapsto Ls'_V \end{pmatrix} \\
(3.1)\ \mathcal{Asm} \vdash \{R_0\}\ \mathtt{e1}\ \{R_1 * \boldsymbol{l} \doteq Ls_V * \mathtt{r} \doteq V_1\} \quad R_1 = S_1 * \gamma(Ls_1, V_1, V_1{}') \\
\vdots \\
(3.m)\ \mathcal{Asm} \vdash \{R_{m-1}\}\ \mathtt{em}\ \{R_m * \boldsymbol{l} \doteq Ls_V * \mathtt{r} \doteq V_m\} \quad R_m = S_m * \gamma(Ls_m, V_m, V_m{}') \\
(4)\ \forall j \in \{m+1 \ldots n\}\ .\ V_j{}' = \mathtt{\&undefined} \\
(5)\ R'_m = \begin{pmatrix} R_m * \exists \mathrm{L}\ .\ \boldsymbol{l} \doteq \mathrm{L}{:}Ls'_V * \\ (\mathrm{L}, X_1) \mapsto V_1{}' * \\ \vdots \\ (\mathrm{L}, X_n) \mapsto V_n{}' * \\ (\mathrm{L}, @this) \mapsto T * \\ (\mathrm{L}, @proto) \mapsto \mathtt{null} * \mathsf{defs}(\mathrm{L}, \mathtt{e}', [X_1, \ldots, X_n]) * \\ \mathsf{decls}(\mathtt{e}', YS) * \mathsf{newobj}_{\mathrm{L}}(\{@proto, @this, X_1, \ldots, X_n\} \cup YS) \end{pmatrix} \\
(6)\ \lambda \mathrm{X}_1 \ldots \mathrm{X}_n\ .\ \{\mathrm{P}_f\}\ \mathtt{e}'\ \{\mathrm{Q}_f\} \in \mathcal{Asm} \qquad (7)\ \boldsymbol{l} \notin \mathsf{fv}(Q) \cup \mathsf{fv}(R_m) \\
(8)\ (\lambda \mathrm{X}_1 \ldots \mathrm{X}_n\ .\ \{\mathrm{P}_f\}\ \mathtt{e}'\ \{\mathrm{Q}_f\})\ V'_1 \ldots V'_n \quad = \quad \{R'_m\}\ \mathtt{e}'\ \{\exists \mathrm{L}\ .\ Q * \boldsymbol{l} \doteq \mathrm{L}{:}Ls'_V\}
\end{array}
}{
\mathcal{Asm} \vdash \{P\}\ \mathtt{e(e1, \ldots em)}\ \{\exists \mathrm{L}\ .\ Q * \boldsymbol{l} \doteq Ls_V\}
}
$$

It’s complicated

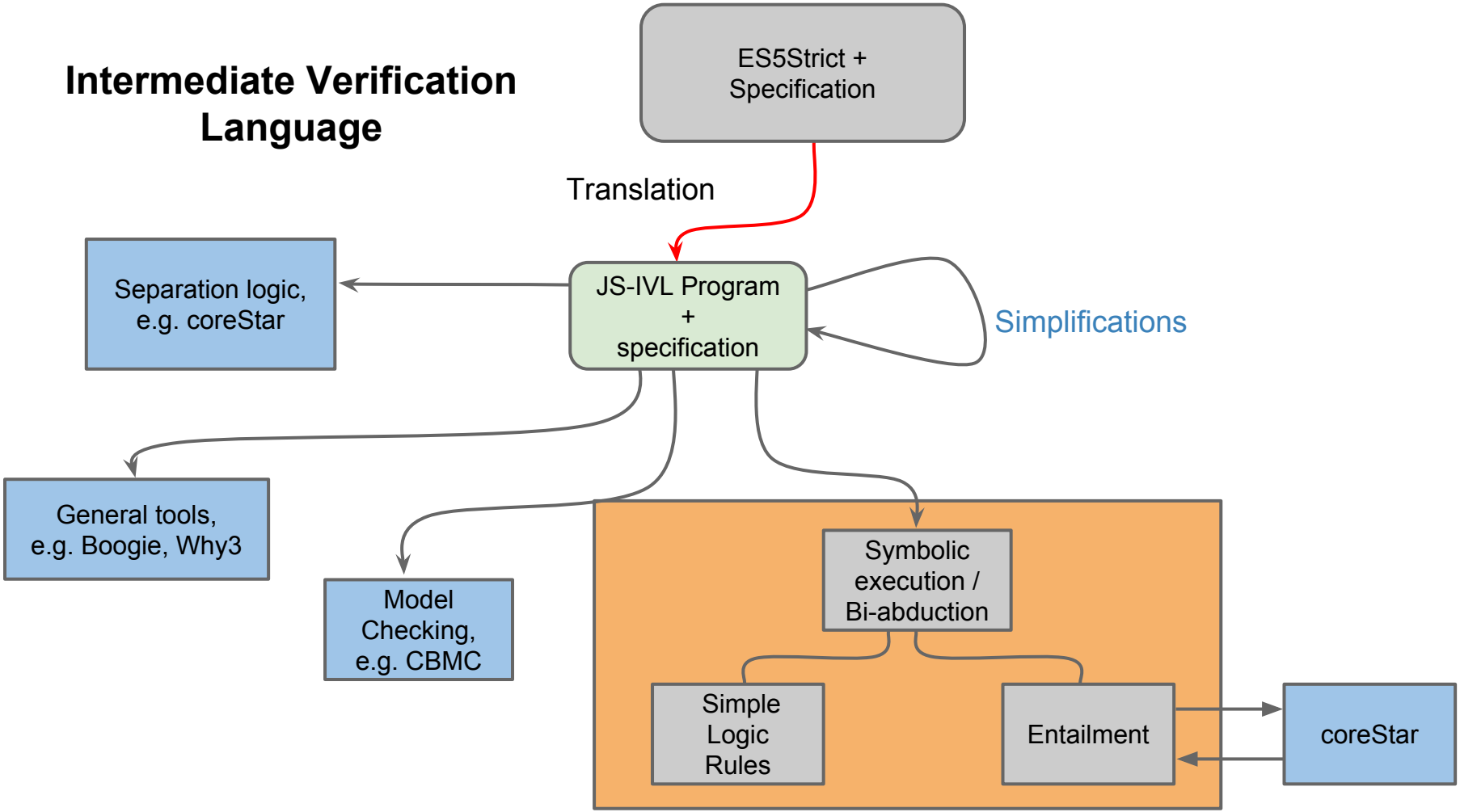
JavaScript

Static Analysis

Intermediate Verification Language



Intermediate Verification Language



JS-IVL

$$C \in \mathtt{Cmd} \triangleq \mathtt{r := E} \mid \mathtt{skip} \mid \mathtt{label\ l} \mid \mathtt{goto\ l} \mid \mathtt{goto\ [E]\ } l_1, l_2$$

Simple goto language, where l denotes a label, E is side effect free expression

JS-IVL

```
C ∈ Cmd ≜ r := E
  | skip | label l | goto l | goto [E] l1, l2
  | r := f(E,...,E) with l
```

Function calls, where $f \in \{ E, \text{eval}, \text{built-in-fid} \}$

JS-IVL

```
C ∈ Cmd ≜ r := E
  | skip | label l | goto l | goto [E] l1, l2
  | r := f(E,...,E) with l
  | r := new() | r := hasField (E, E)
  | r := [E, E]| [E, E] := E
  | r := delete (E, E)
```

JavaScript Heap

JS-IVL

```
C ∈ Cmd ≜ r := E
  | skip | label l | goto l | goto [E] l1, l2
  | r := f(E,...,E) with l
  | r := new() | r := hasField (E, E)
  | r := [E, E]| [E, E] := E
  | r := delete (E, E)
  | r := proto_field (E, E) | r := proto_obj (E, E)
```

Prototype-based Inheritance

JS-IVL

$C \in Cmd \triangleq r := E$
$\quad | \text{ skip } | \text{ label } l | \text{ goto } l | \text{ goto } [E] \; l_1, \; l_2$
$\quad | \; r := f(E,...,E)$
$\quad | \; r := \text{new}() \; | \; r := \text{hasField } (E, E)$
$\quad | \; r := [E, E] | \; [E, E] := E$
$\quad | \; r := \text{delete } (E, E)$
$\quad | \; r := \text{proto\_field } (E, E) \; | \; r := \text{proto\_obj } (E, E)$

$\text{Procedure} \triangleq \text{procedure id } (\text{rthis, rscope, } x_1,...,x_n)$
{
$\quad C_1;...;C_n$
}

JS-IVL Logic Rule for Function Call

(Function call for JS-IVL)

$\lambda x_1, \ldots, x_n.\{P\}\texttt{fid}\{Q * \texttt{r} \doteq V\} \in \mathcal{ASM}$

$\forall j \in \{m+1 \ldots n\} \,.\, E_j = \texttt{\&undefined}$

$\mathcal{ASM} \vdash \{P[E_1/x_1, \ldots, E_n/x_n]\}\, \texttt{x := fid(E}_1\texttt{, ...E}_m\texttt{) with lab}\, \{Q * \texttt{x} \doteq V[E_1/x_1, \ldots, E_n/x_n]\}$

Logic Rules of Function Call for JavaScript and JS-IVL

$$
\frac{
\begin{array}{l}
(1)\,\mathcal{Asm} \vdash \{P\}\,\mathtt{e}\,\{R_0 * \mathbf{r} \doteq F_1\} \\
(2)\; R_0 = \left(\begin{array}{l} S_0 \bowtie \mathsf{This}(F_1, T) \bowtie \gamma(Ls, F_1, F_2) * F_2 \not\doteq l_e * \\ (F_2, @body) \mapsto \lambda X_1, \ldots, X_n.\mathtt{e}' * (F_2, @scope) \mapsto Ls_V' \end{array}\right) \\
(3.1)\;\mathcal{Asm} \vdash \{R_0\}\,\mathtt{e1}\,\{R_1 * \boldsymbol{l} \doteq Ls_V * \mathbf{r} \doteq V_1\} \quad R_1 = S_1 * \gamma(Ls_1, V_1, V_1') \\
\vdots \\
(3.m)\;\mathcal{Asm} \vdash \{R_{m-1}\}\,\mathtt{em}\,\{R_m * \boldsymbol{l} \doteq Ls_V * \mathbf{r} \doteq V_m\} \quad R_m = S_m * \gamma(Ls_m, V_m, V_m') \\
(4)\;\forall j \in \{m+1 \ldots n\}\;.\; V_j' = \texttt{\&undefined} \\
(5)\; R_m' = \left(\begin{array}{l} R_m * \exists \mathrm{L}\;.\; \boldsymbol{l} \doteq \mathrm{L}{:}Ls_V' * \\ (\mathrm{L}, X_1) \mapsto V_1' * \\ \vdots \\ (\mathrm{L}, X_n) \mapsto V_n' * \\ (\mathrm{L}, @this) \mapsto T * \\ (\mathrm{L}, @proto) \mapsto \mathtt{null} * \mathsf{defs}(\mathrm{L}, \mathtt{e}', [X_1, \ldots, X_n]) * \\ \mathsf{decls}(\mathtt{e}', YS) * \mathsf{newobj}_{\mathrm{L}}(\{@proto, @this, X_1, \ldots, X_n\} \cup YS) \end{array}\right) \\
(6)\;\lambda \mathrm{X}_1 \ldots \mathrm{X}_n\;.\;\{\mathrm{P}_f\}\,\mathtt{e}'\,\{\mathrm{Q}_f\} \in \mathcal{Asm} \qquad (7)\;\boldsymbol{l} \notin \mathsf{fv}(Q) \cup \mathsf{fv}(R_m) \\
(8)\;(\lambda \mathrm{X}_1 \ldots \mathrm{X}_n\;.\;\{\mathrm{P}_f\}\,\mathtt{e}'\,\{\mathrm{Q}_f\})\; V_1' \ldots V_n' \quad = \quad \{R_m'\}\,\mathtt{e}'\,\{\exists \mathrm{L}\;.\; Q * \boldsymbol{l} \doteq \mathrm{L}{:}Ls_V'\}
\end{array}
}{
\mathcal{Asm} \vdash \{P\}\,\mathtt{e(e1,\ldots em)}\,\{\exists \mathrm{L}\;.\; Q * \boldsymbol{l} \doteq Ls_V\}
}
$$

$$
\frac{
\begin{array}{l}
\lambda x_1, \ldots, x_n.\{P\}\mathtt{fid}\{Q * \mathbf{r} \doteq V\} \in \mathcal{ASM} \\
\forall j \in \{m+1 \ldots n\}\;.\; E_j = \texttt{\&undefined}
\end{array}
}{
\mathcal{ASM} \vdash \{P[E_1/x_1, \ldots, E_n/x_n]\}\,\mathtt{x := fid(E_1, \ldots E_m) with lab}\,\{Q * \mathbf{x} \doteq V[E_1/x_1, \ldots, E_n/x_n]\}
}
$$

It’s simple

JS-IVL

Static Analysis

Complexity of the language

does not disappear, but has moved to the translation

ES5Strict → Translation → JS-IVL

```
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

obj.m()
```

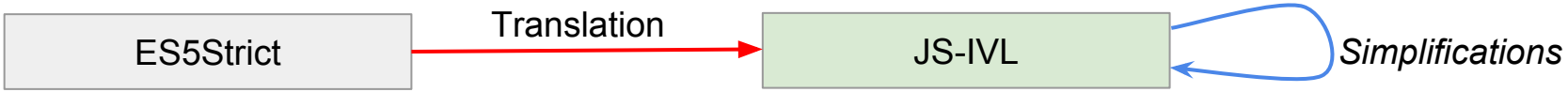
```
procedure anonymous0 (rthis,rscope) {
 0. main_scope := [rscope,"main"]
1. anonymous0_scope := new ()
2. [anonymous0_scope,"#proto"] := null
3. r507 := #empty
4. r509 := rthis
5. goto [typeOf (r509) = Reference] r580, r581
6. label r580
7. r511 := base (r509)
8. goto [r511 = #undefined] r583, r584
9. label r583
10. r525 := new ()
11. [r525,"#proto"] := #lrep
12. [r525,"#class"] := "Error"
13. r505 := r525
14. goto throw.r503
15. goto r585
16. label r584
17. r512 := field (r509)
18. goto [typeOf (r511) = Boolean or typeOf (r511) = Number or typeOf (r511) = String] r586, r587
19. label r586
20. goto [r511 = #undefined or r511 = null] r589, r590
21. label r589
22. r522 := new ()
23. [r522,"#proto"] := #ltep
24. [r522,"#class"] := "Error"
25. r505 := r522
26. goto throw.r503
27. goto r591
28. label r590
29. goto [typeOf (r511) < Object or typeOf (r511) = Object] r592, r593
30. label r592
31. r515 := r511
32. goto r594
33. label r593
34. goto [typeOf (r511) < Boolean or typeOf (r511) = Boolean] r595, r596
35. label r595
```

```
36. r515 := "#boolean_construct" (#empty, #empty, r511) with call_excep.r516
37. goto r517
38. label call_excep.r516
39. r505 := r515
40. goto throw.r503
41. label r517
42. goto r597
43. label r596
44. goto [typeOf (r511) < Number or typeOf (r511) = Number] r598, r599
45. label r598
46. r515 := "#number_construct" (#empty, #empty, r511) with call_excep.r518
47. goto r519
48. label call_excep.r518
49. r505 := r515
50. goto throw.r503
51. label r519
52. goto r600
53. label r599
54. r515 := "#string_construct" (#empty, #empty, r511) with call_excep.r520
55. goto r521
56. label call_excep.r520
57. r505 := r515
58. goto throw.r503
59. label r521
60. goto r600
61. label
62. goto
63. label
64. goto
65. label r594
66. goto r591
67. label r591
68. r523 := proto_field ( r515, r512 )
69. goto [r523 = #empty] r601, r602
```

...

1770 lines

```
1112. goto r490
1113. label r490
1114. goto r487
1115. label r486
1116. r148 := proto_field ( r145, r146 )
1117. goto [r148 = #empty] r494, r495
1118. label r494
1119. r144 := #undefined
1120. goto r496
1121. label r495
1122. r144 := r148
1123. goto r496
1124. label r496
1125. goto r487
1126. label r487
1127. goto r469
1128. label r469
1129. goto r466
1130. label r466
1131. goto r463
1132. label r462
1133. r144 := r138
1134. goto r463
1135. label r463
1136. goto [r144 = #empty] r497, r498
1137. label r497
1138. r160 := r96
1139. goto r499
1140. label r498
1141. r160 := r144
1142. goto r499
1143. label r499
1144. goto [r160 = #empty] r500, r501
                    r500
                    #undefined
                    r502
                    r501
1149. r3 := r160
1150. goto r502
1151. label r502
1152. goto return.r1
1153. label return.r1
1154. label throw.r0
}
```




```
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

obj.m()
```

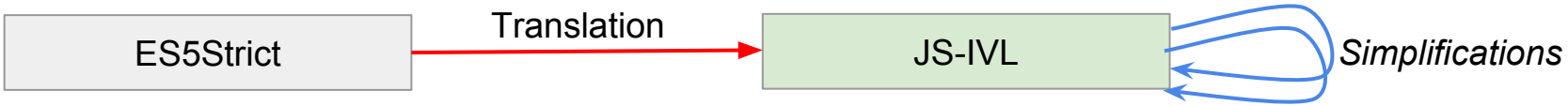
```
procedure anonymous0 (rthis,rscope) {
  0. label Entry
1. main_scope := [rscope,"main"]
2. anonymous0_scope := new ()
3. [anonymous0_scope,"#proto"] := null
4. r510 := rthis
5. goto [rthis = null] r616, r617
6. label r616
7. r527 := new ()
8. [r527,"#proto"] := #Itep
9. [r527,"#class"] := "Error"
10. r505 := r527
11. goto throw.r503
12. label r617
13. goto [rthis = #undefined] r619, r620
14. label r619
15. r527 := new ()
16. [r527,"#proto"] := #Itep
17. [r527,"#class"] := "Error"
18. r505 := r527
19. goto throw.r503
20. label r620
21. goto [r510 = #undefined] r625, r626
22. label r625
23. r544 := new ()
24. [r544,"#proto"] := #Irep
25. [r544,"#class"] := "Error"
26. r505 := r544
27. goto throw.r503
28. label r626
29. goto [typeOf (r510) = Boolean or
typeOf (r510) = Number or typeOf
(r510) = String] r628, r647
30. label r647
31. r533 := proto_field ( r510, "prop" )
32. goto [r533 = #empty] r655, r656
33. label r656
34. r529 := r533
35. goto r624
36. label r624
37. goto [main_scope = #undefined]
r661, r662
```

```
38. label r661
39. r561 := new ()
40. [r561,"#proto"] := #Irep
41. [r561,"#class"] := "Error"
42. r505 := r561
43. goto throw.r503
44. label r662
45. goto [typeOf (main_scope) =
Boolean or typeOf (main_scope) =
Number or typeOf (main_scope) =
String] r664, r682
46. label r682
47. goto [main_scope = #Ig] r685, r686
48. label r686
49. r546 := [main_scope,"prop"]
50. goto r660
51. label r660
52. goto [typeOf (r529) = typeOf (r546)]
r694, r695
53. label r695
54. r562 := false
55. goto r725
56. label r725
57. r506 := r562
58. goto return.r504
59. label r694
60. goto [typeOf (r529) = Undefined or
typeOf (r529) = Null] r697, r698
61. label r698
62. goto [typeOf (r529) = String or
typeOf (r529) = Object or typeOf (r529)
= Boolean] r700, r701
63. label r701
64. goto
r706, r7
65. label
66. r562
67. goto r725
68. label r706
69. goto [r529 = nan] r709, r710
70. label r710
71. goto [r546 = nan] r712, r713
72. label r713
73. goto [r529 = r546] r715, r716
```

...

```
1112. goto r490
164. goto throw.r0
165. label r413
166. r123 := "#number_construct"
(#empty, #empty, r99) with call_excep.
r126
167. goto r409
168. label call_excep.r126
169. r2 := r123
170. goto throw.r0
171. label r410
172. r123 := "#boolean_construct"
(#empty, #empty, r99) with call_excep.
r124
173. goto r409
174. label call_excep.r124
175. r2 := r123
176. goto throw.r0
177. label r407
178. r123 := r99
179. goto r409
180. label r404
181. r130 := new ()
182. [r130,"#proto"] := #Itep
183. [r130,"#class"] := "Error"
184. r2 := r130
185. goto throw.r0
186. label r348
187. r91 := new ()
188. [r91,"#proto"] := #Irep
189. [r91,"#class"] := "Error"
190. r2 := r91
                hrow.r0
              216
              new ()
              #proto"] := #Irep
195. [r26,"#class"] := "Error"
196. r2 := r26
197. goto throw.r0
198. label throw.r0
199. label return.r1
}
```

410 lines




```
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

obj.m()
```

```
procedure anonymous0 (rthis,rscope) {
1. main_scope := [rscope,"main"]
2. anonymous0_scope := new ()
3. goto [rthis = null] r616, r617
4. label r616
5. r527 := new ()
6. [r527,"#proto"] := #Itep
7. [r527,"#class"] := "Error"
8. r505 := r527
9. goto throw.r503
10. label r617
11. goto [rthis = #undefined] r619, r620
12. label r619
13. r527 := new ()
14. [r527,"#proto"] := #Itep
15. [r527,"#class"] := "Error"
16. r505 := r527
17. goto throw.r503
18. label r620
19. goto [typeOf (rthis) = Boolean or
typeOf (rthis) = Number or typeOf
(rthis) = String] r628, r647
20. label r647
21. r533 := proto_field ( rthis, "prop" )
22. goto [r533 = #empty] r655, r656
23. label r656
24. r529 := r533
25. goto r624
26. label r624
27. goto [main_scope = #lg] r685, r686
28. label r686
29. r546 := [main_scope,"prop"]
30. goto r660
31. label r660
32. goto [typeOf (r529) = typeOf (r546)]
r694, r695
33. label r695
34. r562 := false
35. goto r725
36. label r725
37. r506 := r562
38. goto return.r504
39. label r694
40. goto [typeOf (r529) = Undefined or
typeOf (r529) = Null] r697, r698
```

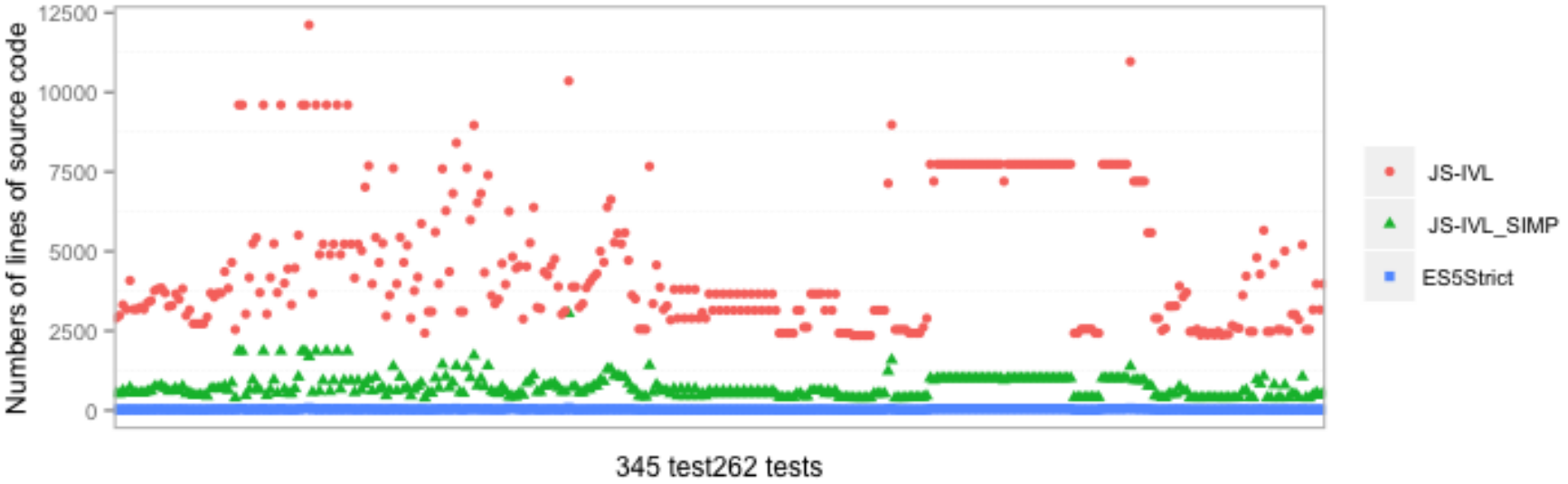
```
41. label r698
42. goto [typeOf (r529) = String or
typeOf (r529) = Object or typeOf (r529)
= Boolean] r700, r701
43. label r701
44. goto [typeOf (r529) = Number]
r706, r707
45. label r707
46. r562 := false
47. goto r725
48. label r706
49. goto [r529 = nan] r709, r710
50. label r710
51. goto [r546 = nan] r712, r713
52. label r713
53. goto [r529 = r546] r715, r716
54. label r716
55. goto [r529 = 0. and r546 = -0.] r718,
r719
56. label r719
57. goto [r529 = -0. and r546 = 0.] r721,
r722
58. label r722
59. r562 := false
60. goto r725
61. label r721
62. r562 := true
63. goto r725
64. label r718
65. r562 := true
66. goto r725
67. label r715
68. r562 := true
69.
70.
71.
72.
73. label r709
74. r562 := false
75. goto r725
76. label r700
77. goto [r529 = r546] r703, r704
78. label r704
79. r562 := false
80. goto r725
```

...

187 lines

```
procedure main (rthis,rscope) {
   [#lg,"prop"] := #undefined
   [#lg,"obj"] := #undefined
   [#lg,"prop"] := "global"
   r34 := new ()
   [r34,"#proto"] := #lop
   [r34,"#class"] := "Object"
   [r34,"prop"] := "local"
   r53 := new ()
   [r53,"#class"] := "Function"
   [r53,"#proto"] := #lfp
   r54 := new ()
   [r54,"#proto"] := #lop
   [r54,"constructor"] := r53
   [r53,"prototype"] := r54
   r55 := new ()
   [r55,"main"] := #lg
   [r53,"#fid"] := "anonymous0"
   [r53,"#constructid"] := "anonymous0"
   [r53,"length"] := 0
   [r53,"#scope"] := r55
   [r34,"m"] := r53

   [#lg,"obj"] := r34
   r102 := proto_field ( #lg, "obj" )
   r122 := proto_field ( r34, "m" )
   r138 := anonymous0 (r34, r55) with
call_excep
   goto r462

   label call_excep
                                  r138
                                  abel


   result_var := r138
   goto return_label

}
```

Simplification Evaluation



ES5Strict support

We support all of ES5Strict, except of:

- Getters / Setters
- Property attributes
- For statement
- Switch statement
- Arguments Object

Next todo

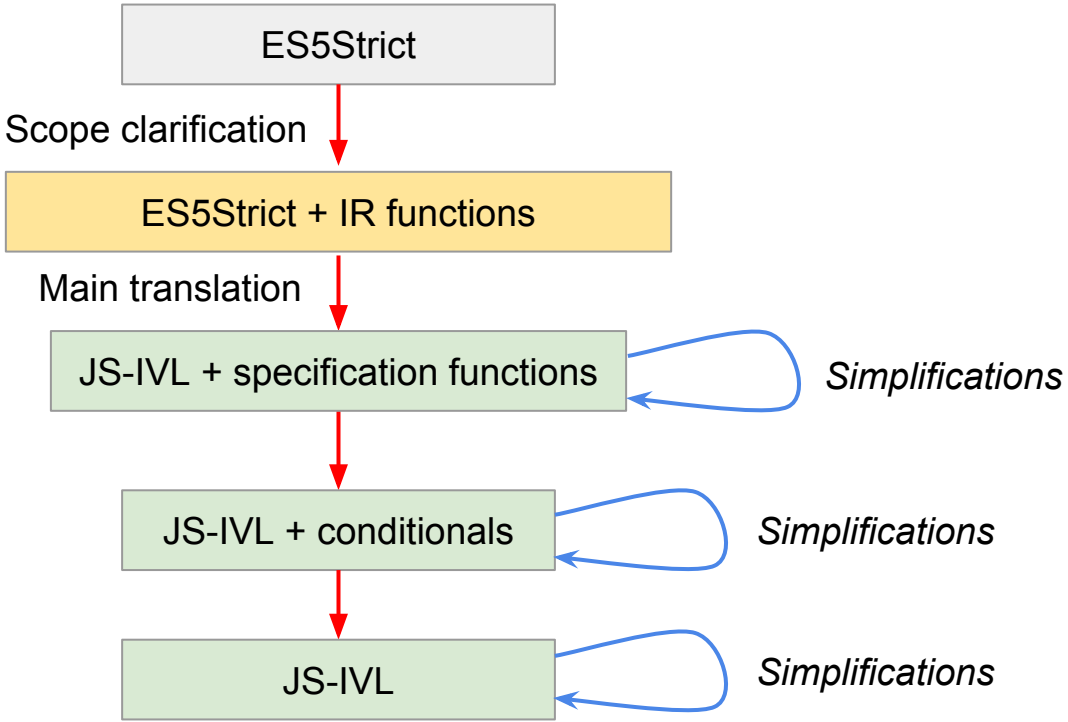
- For-in statement
- Indirect eval

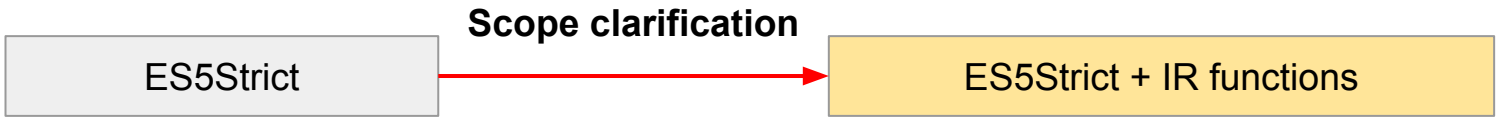
Not doing

- Array literals and library
- Regular expression literals and library
- Other libraries (chapter 15)

use existing implementation, e.g., V8

Translation






```
var prop = 'global';
var obj = {
  prop: 'local',
  m: function() {
    return this.prop === prop;
  }
};

obj.m()
```

```
IRfunction main () {
  var prop = 'global';
  var obj = {
    prop: 'local',
    m: function() /** @fid "anonymous0" **/
  };
  obj.m()
} []

IRfunction anonymous0 () {
  return this.prop === prop;
} [main: [prop, obj]]
```

ES5Strict + IR functions

Main Translation

JS-IVL

```
IRfunction main () {
  var prop = 'global';
  var obj = {
    prop: 'local',
    m: function()
      /** @fid "anonymous0" **/
  };
  obj.m()
} []
```

```
procedure main (rthis,rscope) {
    …
    ob := new ()
    [obj,"#proto"] := #ObjectPrototype
    [obj,"prop"] := "local"

    function_obj := new ()
    [function_obj,"#proto"] := #FunctionPrototype

    prototype_obj := new ()
    [prototype_obj,"#proto"] := #ObjectPrototype
    [function_obj,"prototype"] := prototype_obj

    scope := new ()
    [scope,"main"] := #GlobalObject

    [function_obj,"#fid"] := "anonymous0"
    [function_obj,"#scope"] := scope

    [obj,"m"] := function_obj
    …
}
```


```
IRfunction anonymous0 () {
  return this.prop === prop;
} [main: [prop, obj]]
```

```
procedure anonymous0 (rthis, rscope) {
  …
  goto [rthis = #undefined] ltrue, lfalse

  label ltrue
    error := new ()
    [error,"#proto"] := #lTypeErrorPrototype
    result_var := error
    goto throw_label

  label lfalse
    …
    r1 := proto_field (rthis, "prop")
    …
}
```

ExplainJS

ES5Strict

```
var prop = 'global';
var obj = {
 prop: 'local',
 m: function() {
  return this.prop === prop;
 }
};

obj.m();
```

JS-IVL

```
procedure anonymous0 (rthis, rscope) {
  …
  goto [rthis = #undefined] ltrue, lfalse
  label ltrue
    error := new ()
    [error,"#proto"] := #lTypeErrorPrototype
    result_var := error
    goto throw_label
  label lfalse
    …
    r1 := proto_field (rthis, "prop")
    …
}
procedure main (rthis, rscope) {
   …
   r2 := anonymous0 (obj, scope)
   …
}
```

ExplainJS

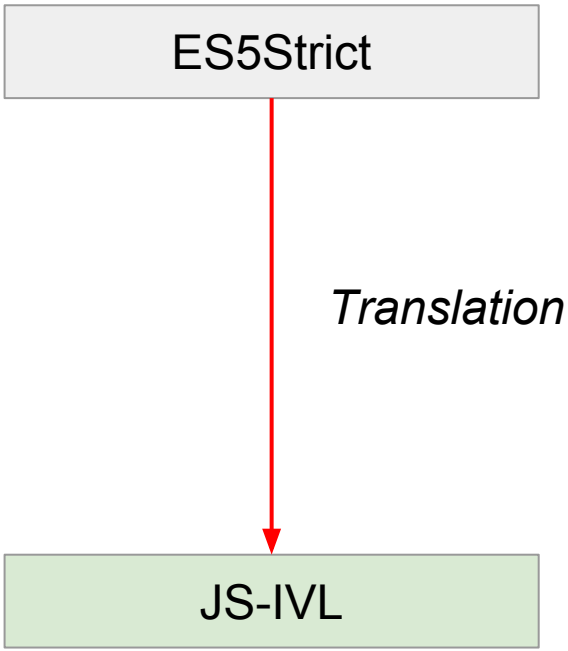
ES5Strict

```
var prop = 'global';
var obj = {
 prop: 'local',
 m: function() {
  return this.prop === prop;
 }
};

var f = obj.m;
f()
```

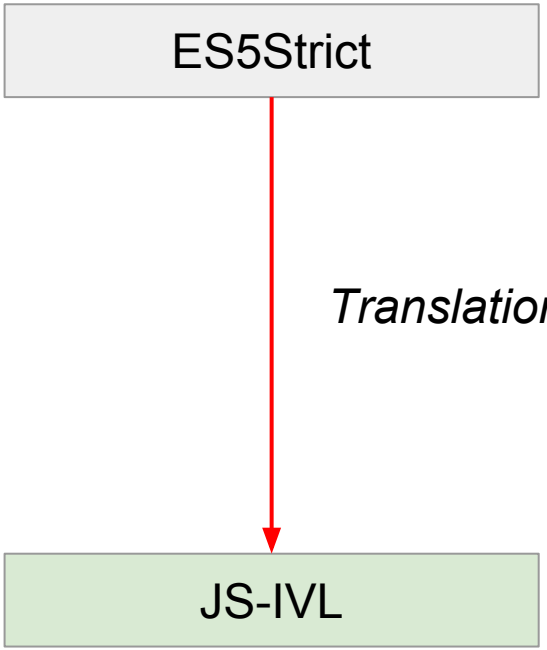
JS-IVL

```
procedure anonymous0 (rthis, rscope) {
  …
  goto [rthis = #undefined] ltrue, lfalse
  label ltrue
    error := new ()
    [error,"#proto"] := #lTypeErrorPrototype
    result_var := error
    goto throw_label
  label lfalse
    …
    r1 := proto_field (rthis, "prop")
    …
}
procedure main (rthis, rscope) {
  …
  r2 := anonymous0 (#undefined, scope)
  …
}
```

Reliable Translation

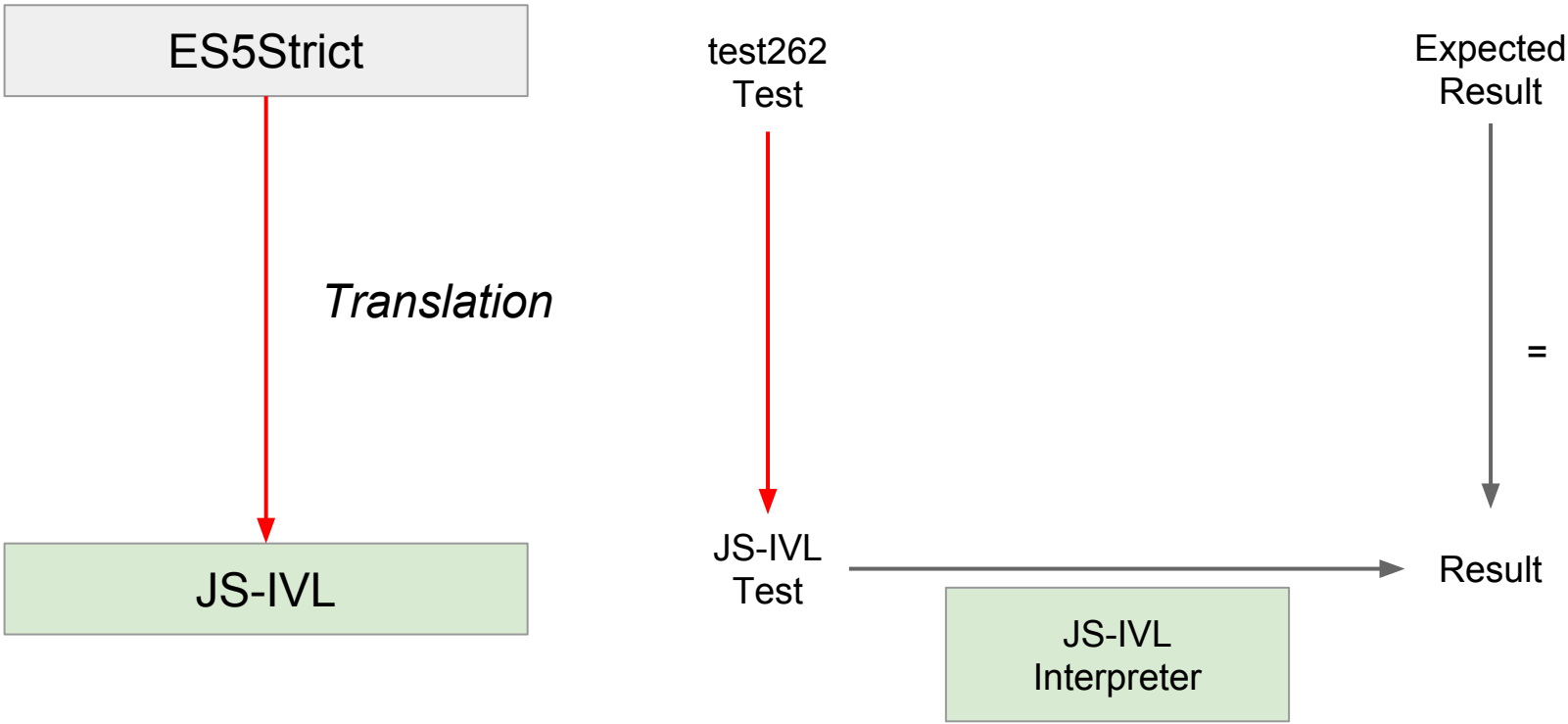


Reliable Translation

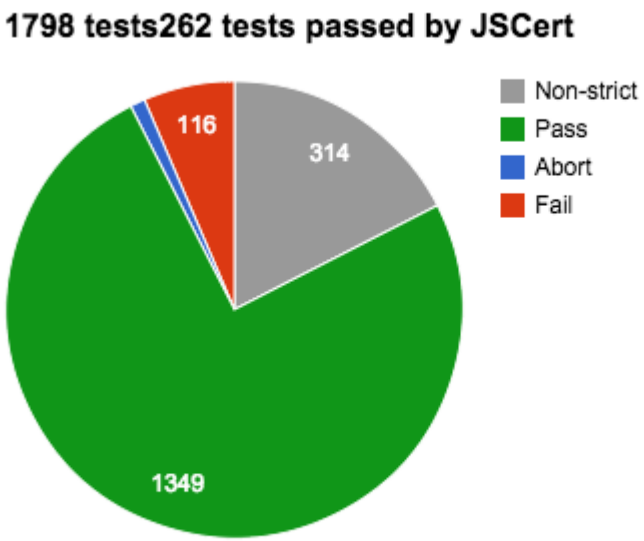


1. Prove that translation is correct using operational semantics
2. Test using test262

Reliable Translation Using Testing



Testing Results



Abort	19
Getters/Setters	13
Switch	6
Fail	116
Real failings	17
Arguments	54
Attributes	28
Parser failings	9
Array	4
Indirect eval	4

Conclusions

- JS-IVL, an intermediate verification language for ES5Strict
  - clear semantics
  - simpler logic rules
  - insight into the program
- Translation to JS-IVL that follows ES5Strict operational semantics
- Correctness of the translation using test262 tests

Future work

- Support all of ES5Strict
- Improve simplifications
- Build symbolic execution tool for JS-IVL
- Experiment with different backends, e.g. separation logic (coreStar), general tools (Boogie, Why3), model checking (CBMC)
- Prove correctness of the translation

Take JavaScript from the Web and find bugs

THE END