Understanding and Verifying JavaScript Programs

Philippa Gardner

Imperial College London

LFCS 30th Anniversary

JavaScript at Imperial

Philippa Gardner

José Fragoso Santos

Petar Maksimović

Daiva Naudžiūnienė

Azalea Raad

Thomas Wood

Mechanised Language Specification

Standard ML: formal definition Milner, Harper, MacQueen, Tofte, 1990 and 1997; mechanised definition Lee, Crary, Harper, 2006.

C: many partial definitions, some mechanised e.g. by Norrish 1998 , Leroy 2009; complete mechanised definition Ellison, Rosu, 2012; lots of recent work on C11, e.g. Batty's thesis, 2014.

Fragments of Java: many partial (large) definitions, first studied by Drossopoulou, Eisenbach, 1997; mechanised definition, Syme, 1998.

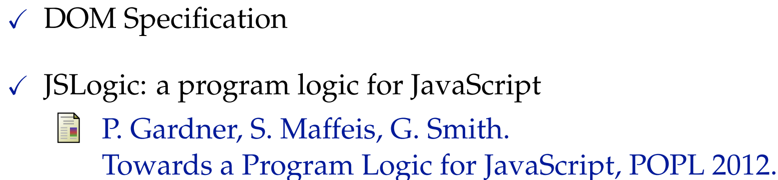
Javascript: complete formal definition of the ECMAScript 3 standard (ES3), Maffeis, Mitchell, Tally, 2008; mechanised definition of ES5, us, 2014; Park, Stefanescu, Rosu, 2015.

JavaScript at Imperial

✓ DOM Specification

P. Gardner, G. Smith, M. Wheelhouse, U. Zarfaty.
Local Hoare Reasoning about the DOM, PODS 2008.

JavaScript at Imperial

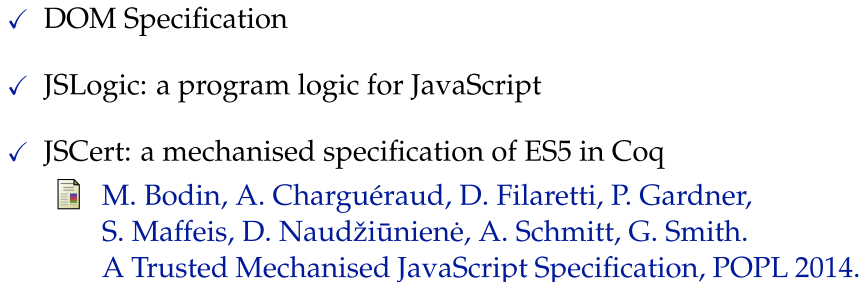


✓ DOM Specification

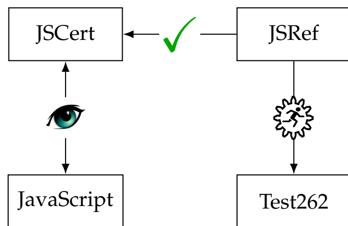
✓ JSLogic: a program logic for JavaScript

P. Gardner, S. Maffeis, G. Smith.
Towards a Program Logic for JavaScript, POPL 2012.

JavaScript at Imperial



- ✓ DOM Specification
- ✓ JSLogic: a program logic for JavaScript
- ✓ JSCert: a mechanised specification of ES5 in Coq
  - M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudžiūnienė, A. Schmitt, G. Smith. A Trusted Mechanised JavaScript Specification, POPL 2014.



Inria Collaborators

A. Schmitt

A. Charguéraud

M. Bodin

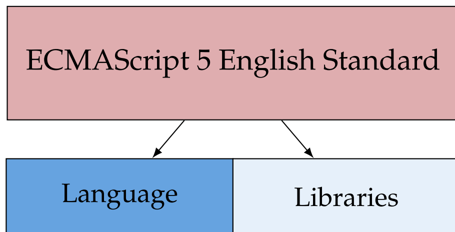
JavaScript at Imperial

- ✓ DOM Specification
- ✓ JSLogic: a program logic for JavaScript
- ✓ JSCert: a mechanised specification of ES5 in Coq
- JSIL: an intermediate language for JavaScript
- JSVerify: a verification tool for JavaScript

JavaScript and Verification

JavaScript

JavaScript and Verification



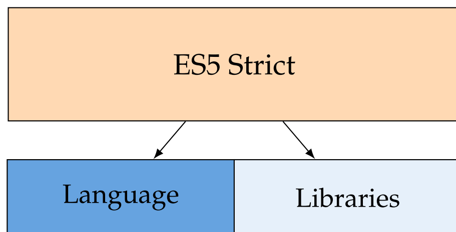
How is it organised?

Chapters 1-7: Overview, notation, parsing

Chapters 8-14: Language constructs

Chapter 15: Library functions

JavaScript and Verification



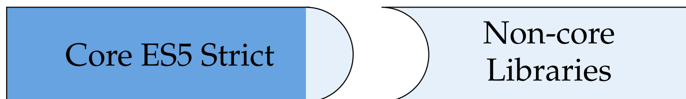
How is it organised?

Same as before; strict-only features throughout chapters 8-15

How is it different?

Better behavioural properties: lexicographic scoping, mandatory variable declarations, explicit error throwing...

JavaScript and Verification



What is Core ES5 Strict?

- All of the language constructs
- Core library functions

Non-core library functions definable using the core language

Why the core language?

Important for verification

Verifying Core ES5 Strict

- JSIL: simple intermediate goto language, good for verification, memory model similar to JavaScript
- Semantics-directed compilation from Core ES5 Strict to JSIL
- Core library functions implemented in JSIL

- JSVerify: Smallfoot-like verification tool for JSIL (in future for JavaScript)
- Core library functions specified and verified using JSVerify

JavaScript vs. JSIL Verification

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

JavaScript

Logic Rules of Function / Procedure Call

JSIL

$$
\frac{\mathtt{S}(\mathtt{fid}) = \lambda \mathtt{x}_1, \ldots, \mathtt{x}_n.\, \{P\}\, m\, \{Q * \mathtt{ret} \doteq \mathtt{E}\} \quad \forall_{k<j\leq n}\, \mathtt{E}_j = \mathsf{undefined}}{\mathtt{S} \vdash \{P[\mathtt{E}_1/\mathtt{x}_1, \ldots, \mathtt{E}_n/\mathtt{x}_n]\}\ \mathtt{x} := \mathtt{fid}(\mathtt{E}_1, \ldots, \mathtt{E}_k)\ \{Q[\mathtt{E}_1/\mathtt{x}_1, \ldots, \mathtt{E}_n/\mathtt{x}_n] * \mathtt{x} \doteq \mathtt{E}[\mathtt{E}_1/\mathtt{x}_1, \ldots, \mathtt{E}_n/\mathtt{x}_n]\}}
$$

Incorporating Non-core Libraries

Axiomatic specification to verify client programs

Does not follow the standard, which is operational

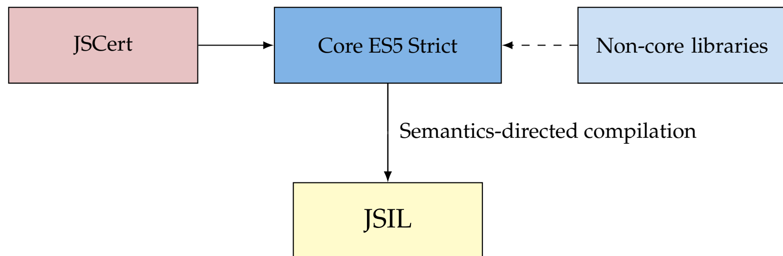
Justification of specifications

Informal appeal to the English standard

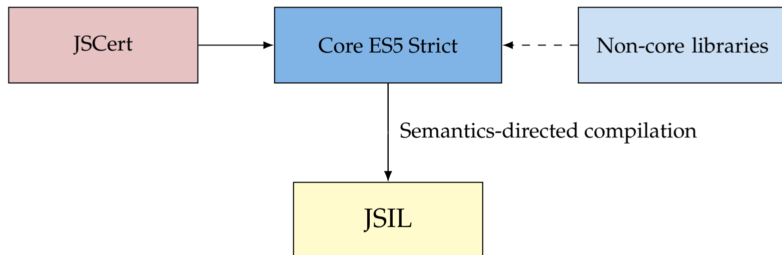
Reference implementation in JSIL or Core ES5 Strict, following the standard, verified with JSVerify, tested against Test262

Verification of industrial-strength library implementations

Introducing JSIL



Introducing JSIL



To be implemented

Attributes, for-in, getters/setters, the arguments object
Some core library functions

The Syntax of JSIL

Expressions:

$$\begin{aligned} e \;::=\; & v \mid x \mid \ominus\, e \mid e \oplus e \mid \mathsf{typeof}\,(e) \\ & \mid e \,._{\mathsf{o}}\, e \mid e \,._{\mathsf{v}}\, e \mid \mathsf{base}\,(e) \mid \mathsf{field}\,(e) \end{aligned}$$

Commands:

$$\begin{aligned} c \;::=\; & \mathsf{skip} \mid x := e \mid x := \mathsf{new}\,() \mid x := [e, e] \\ & \mid [e, e] := e \mid \mathsf{delete}\,(e) \mid x := \mathsf{hasField}\,(e, e) \\ & \mid x := \mathsf{protoField}\,(e, e) \mid x := \mathsf{protoObj}\,(e, e) \\ & \mid \mathsf{goto}\; i \mid \mathsf{goto}\; [e]\; i, j \mid x := e(\overline{e})\; \mathsf{with}\; j \end{aligned}$$

The Syntax of JSIL

Expressions:

$$
\begin{aligned}
\mathtt{e} \quad ::= \; & \mathtt{v} \mid \mathtt{x} \mid \ominus\, \mathtt{e} \mid \mathtt{e} \oplus \mathtt{e} \mid \mathsf{typeof}\,(\mathtt{e}) \\
& \mid \mathtt{e} \cdot_{\mathsf{o}} \mathtt{e} \mid \mathtt{e} \cdot_{\mathsf{v}} \mathtt{e} \mid \mathsf{base}\,(\mathtt{e}) \mid \mathsf{field}\,(\mathtt{e})
\end{aligned}
$$

Commands:

$$
\begin{aligned}
\mathtt{c} \quad ::= \; & \mathsf{skip} \mid \mathtt{x} := \mathtt{e} \mid \mathtt{x} := \mathsf{new}\,() \mid \mathtt{x} := [\mathtt{e}, \mathtt{e}] \\
& \mid [\mathtt{e}, \mathtt{e}] := \mathtt{e} \mid \mathsf{delete}\,(\mathtt{e}) \mid \mathtt{x} := \mathsf{hasField}\,(\mathtt{e}, \mathtt{e}) \\
& \mid \mathtt{x} := \mathsf{protoField}\,(\mathtt{e}, \mathtt{e}) \mid \mathtt{x} := \mathsf{protoObj}\,(\mathtt{e}, \mathtt{e}) \\
& \mid \mathsf{goto}\; i \mid \mathsf{goto}\; [\mathtt{e}]\; i, j \mid \mathtt{x} := \mathtt{e}(\overline{\mathtt{e}})\; \mathsf{with}\; j
\end{aligned}
$$

Extensible objects, dynamic fields

The Syntax of JSIL

Expressions:

$$\begin{aligned} \mathtt{e} \;::=\;& \mathtt{v} \mid \mathtt{x} \mid \ominus\, \mathtt{e} \mid \mathtt{e} \oplus \mathtt{e} \mid \mathsf{typeof}\,(\mathtt{e}) \\ & \mid \mathtt{e}.\mathsf{_o}\mathtt{e} \mid \mathtt{e}.\mathsf{_v}\mathtt{e} \mid \mathsf{base}\,(\mathtt{e}) \mid \mathsf{field}\,(\mathtt{e}) \end{aligned}$$

Commands:

$$\begin{aligned} \mathtt{c} \;::=\;& \mathsf{skip} \mid \mathtt{x} := \mathtt{e} \mid \mathtt{x} := \mathsf{new}\,() \mid \mathtt{x} := [\mathtt{e}, \mathtt{e}] \\ & \mid [\mathtt{e}, \mathtt{e}] := \mathtt{e} \mid \mathsf{delete}\,(\mathtt{e}) \mid \mathtt{x} := \mathsf{hasField}\,(\mathtt{e}, \mathtt{e}) \\ & \mid \mathtt{x} := \mathsf{protoField}(\mathtt{e}, \mathtt{e}) \mid \mathtt{x} := \mathsf{protoObj}(\mathtt{e}, \mathtt{e}) \\ & \mid \mathsf{goto}\ i \mid \mathsf{goto}\ [\mathtt{e}]\ i, j \mid \mathtt{x} := \mathtt{e}(\overline{\mathtt{e}})\ \mathsf{with}\ j \end{aligned}$$

Prototype chains

The Syntax of JSIL

Expressions:

$$\begin{aligned} \mathtt{e} \;::=\; & \mathtt{v} \mid \mathtt{x} \mid \ominus\, \mathtt{e} \mid \mathtt{e} \oplus \mathtt{e} \mid \mathsf{typeof}\,(\mathtt{e}) \\ & \mid \mathtt{e}._{\mathsf{o}}\mathtt{e} \mid \mathtt{e}._{\mathsf{v}}\mathtt{e} \mid \mathsf{base}\,(\mathtt{e}) \mid \mathsf{field}\,(\mathtt{e}) \end{aligned}$$

Commands:

$$\begin{aligned} \mathtt{c} \;::=\; & \mathsf{skip} \mid \mathtt{x} := \mathtt{e} \mid \mathtt{x} := \mathsf{new}\,() \mid \mathtt{x} := [\mathtt{e}, \mathtt{e}] \\ & \mid [\mathtt{e}, \mathtt{e}] := \mathtt{e} \mid \mathsf{delete}\,(\mathtt{e}) \mid \mathtt{x} := \mathsf{hasField}\,(\mathtt{e}, \mathtt{e}) \\ & \mid \mathtt{x} := \mathsf{protoField}\,(\mathtt{e}, \mathtt{e}) \mid \mathtt{x} := \mathsf{protoObj}\,(\mathtt{e}, \mathtt{e}) \\ & \mid \mathsf{goto}\; i \mid \mathsf{goto}\; [\mathtt{e}]\; i, j \mid \mathtt{x} := \mathtt{e}(\overline{\mathtt{e}})\; \mathsf{with}\; j \end{aligned}$$

Dynamic function choice

The Syntax of JSIL

Expressions:

$$
\begin{aligned}
\mathtt{e} \; ::= \; & \mathtt{v} \mid \mathtt{x} \mid \ominus\, \mathtt{e} \mid \mathtt{e} \oplus \mathtt{e} \mid \mathsf{typeof}\,(\mathtt{e}) \\
& \mid \mathtt{e} \cdot_{\mathsf{o}} \mathtt{e} \mid \mathtt{e} \cdot_{\mathsf{v}} \mathtt{e} \mid \mathsf{base}\,(\mathtt{e}) \mid \mathsf{field}\,(\mathtt{e})
\end{aligned}
$$

Commands:

$$
\begin{aligned}
\mathtt{c} \; ::= \; & \mathsf{skip} \mid \mathtt{x} := \mathtt{e} \mid \mathtt{x} := \mathsf{new}\,() \mid \mathtt{x} := [\mathtt{e}, \mathtt{e}] \\
& \mid [\mathtt{e}, \mathtt{e}] := \mathtt{e} \mid \mathsf{delete}\,(\mathtt{e}) \mid \mathtt{x} := \mathsf{hasField}\,(\mathtt{e}, \mathtt{e}) \\
& \mid \mathtt{x} := \mathsf{protoField}\,(\mathtt{e}, \mathtt{e}) \mid \mathtt{x} := \mathsf{protoObj}\,(\mathtt{e}, \mathtt{e}) \\
& \mid \mathsf{goto}\; i \mid \mathsf{goto}\; [\mathtt{e}]\; i, j \mid \mathtt{x} := \mathtt{e}(\overline{\mathtt{e}})\; \mathsf{with}\; j
\end{aligned}
$$

Procedures: $\mathtt{proc} \; ::= \; \mathsf{proc}\; m(\overline{\mathtt{x}})\{\overline{\mathtt{c}}\}$

Compiling ES5 Strict to JSIL

JavaScript Code

```
Object.prototype.foo = 1;
var bar = 2;

function f() {
    this.baz = this.bar + foo;
}

f.prototype.bar = 3
```

Three ways of calling `f`:

- Function call: `f()`
- Method call: `this.f()`
- Constructor call: `new f()`

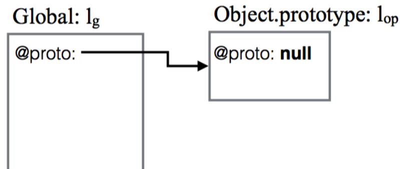
Compiling Functions

- Each function translated to a top-level procedure.
- Global code translated to a special procedure `main`.
- No nesting of procedures.
- Scope and the `this` object as first two parameters

JavaScript Code	JSIL Code
function f() { ... }	proc f(x_{sc}, x_{this})\{...\}
Global code	proc main()\{...\}

Compiling ES5 Strict to JSIL

Heap

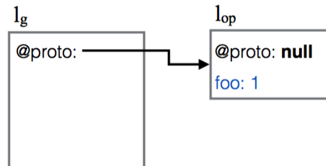


JavaScript Code

```
Object.prototype.foo = 1;
var bar = 2;

function f() {
    this.baz = this.bar + foo;
}

f.prototype.bar = 3
```

Compiling ES5 Strict to JSIL

Heap



JavaScript Code

```
Object.prototype.foo = 1;
var bar = 2;

function f() {
    this.baz = this.bar + foo;
}

f.prototype.bar = 3
```

JSIL Code

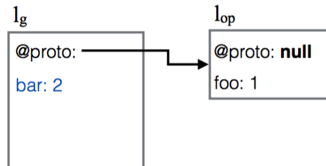
$[l_{op}, \text{"foo"}] := 1$

Compiling ES5 Strict to JSIL

JavaScript Code

```
Object.prototype.foo = 1;
var bar = 2;

function f() {
    this.baz = this.bar + foo;
}

f.prototype.bar = 3
```

Heap



JSIL Code

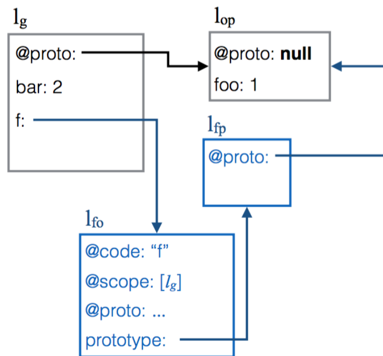
$[l_g, \text{“bar”}] := 2$

Compiling ES5 Strict to JSIL

JavaScript Code

```
Object.prototype.foo = 1;
var bar = 2;

function f() {
    this.baz = this.bar + foo;
}

f.prototype.bar = 3
```

Heap



JSIL Code

x_{fp} := new()
$[x_{fp}, @proto] := l_{op}$
x_{fo} := new()
$[x_{fo}, @code] :=$ "f"
$[x_{fo}, @scope] := [l_g]$
$[x_{fo}, @proto] := \ldots$
$[x_{fo}, \text{"prototype"}] := x_{fp}$
$[l_g, \text{"f"}] := x_{fo}$

Compiling ES5 Strict to JSIL

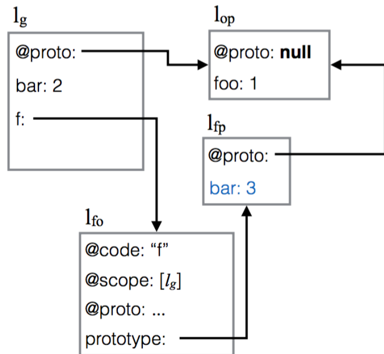
JavaScript Code

```
Object.prototype.foo = 1;
var bar = 2;

function f() {
    this.baz = this.bar + foo;
}

f.prototype.bar = 3
```

JSIL Code

$[x_{fp}, \text{"bar"}] := 3$

Heap



Constructor call: new f()

JavaScript Code

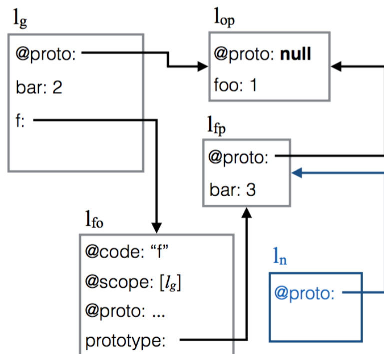
```
Object.prototype.foo = 1;
var bar = 2;

function f() {
    this.baz = this.bar + foo;
}

f.prototype.bar = 3
```

new f() ⟵ constructor call

JSIL Code

$x_n := \mathsf{new}()$
$x_{fo} := [l_g, \text{"f"}]$
$x_{fp} := [x_{fo}, \text{"prototype"}]$
$[x_n, @proto] := x_{fp}$
$x_{scope} := [x_{fo}, @scope]$
$x_{this} := x_n$
$x_f := [x_{fo}, @code]$
$x_{ret} := x_f(x_{scope}, x_{this})$

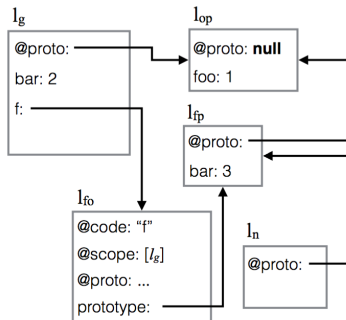
Heap



Constructor call: new f()

In this case, the `this` is bound to the newly created object.

```
this.baz = this.bar     +     foo
```



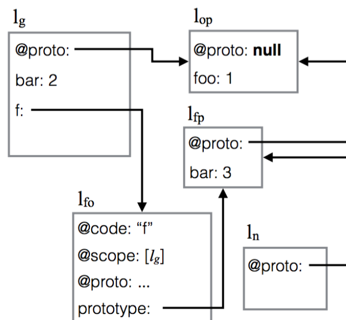
Constructor call: new f()

In this case, the `this` is bound to the newly created object.

```
this.baz = this.bar     +     foo
```

Step 1

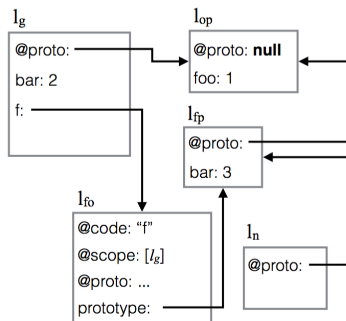
$l_{n.o}$baz



Constructor call: new f()

In this case, the `this` is bound to the newly created object.

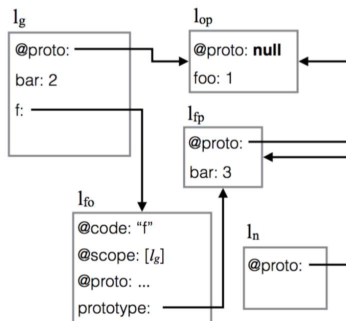
```
this.baz = this.bar      +      foo
   |          |
 Step 1     Step 2
   ↓          ↓
 ln.baz     ln.bar
```



Constructor call: new f()

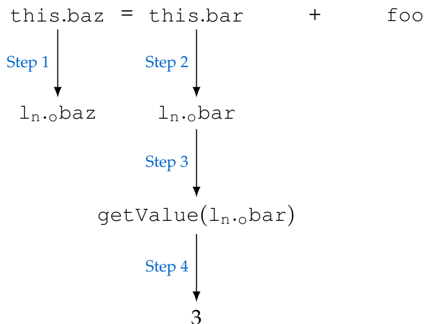
In this case, the `this` is bound to the newly created object.

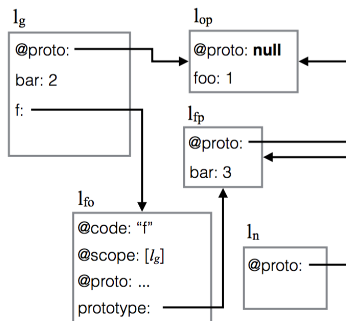
```
this.baz = this.bar      +      foo
   |          |
 Step 1     Step 2
   ↓          ↓
 ln.obaz    ln.obar
              |
            Step 3
              ↓
      getValue(ln.obar)
```



Constructor call: new f()

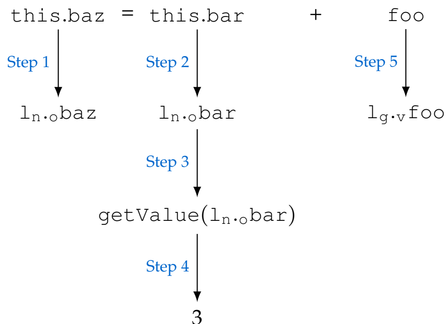
In this case, the `this` is bound to the newly created object.

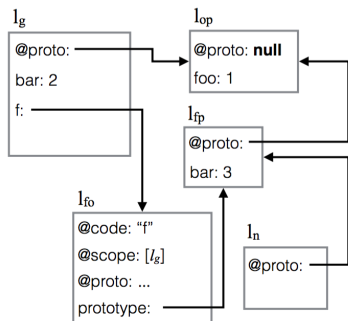




Constructor call: new f()

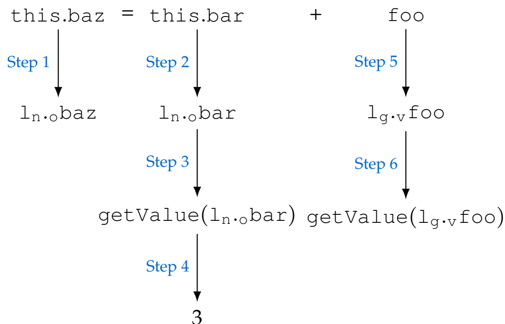
In this case, the `this` is bound to the newly created object.

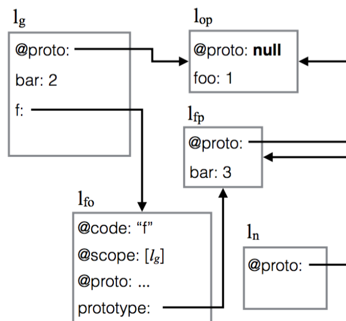




Constructor call: new f()

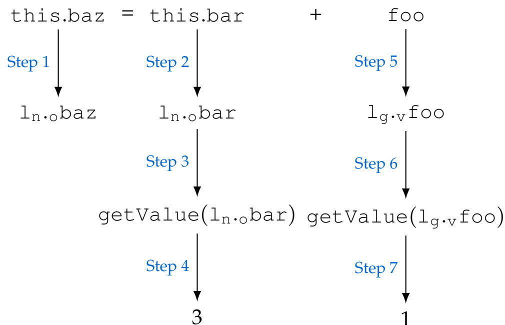
In this case, the `this` is bound to the newly created object.

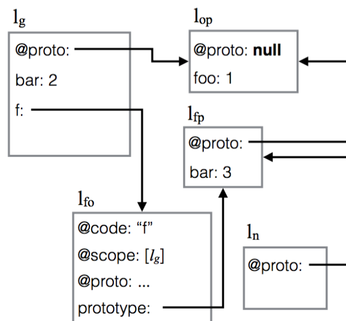




Constructor call: new f()

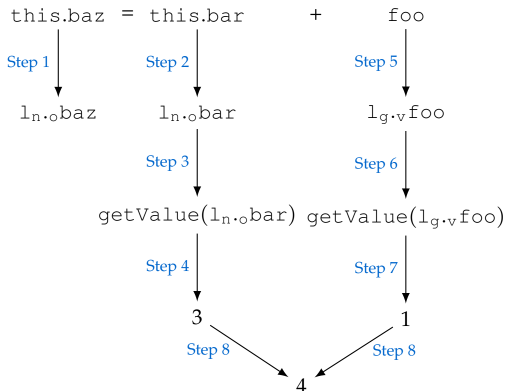
In this case, the `this` is bound to the newly created object.

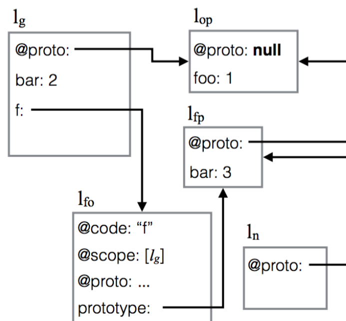




Constructor call: new f()

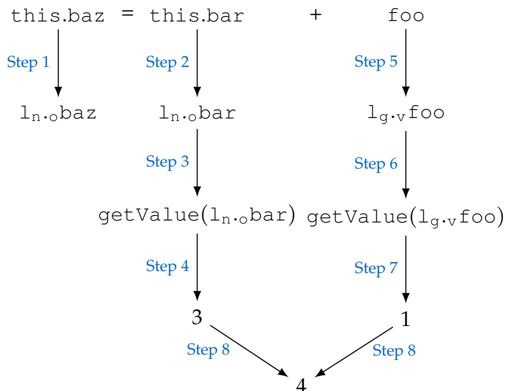
In this case, the `this` is bound to the newly created object.

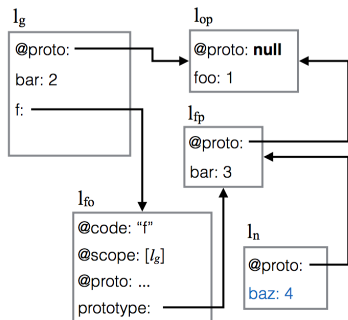




Constructor call: new f()

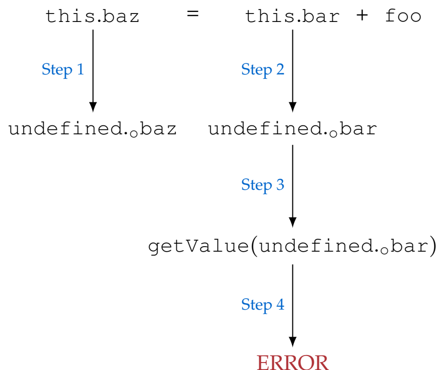
In this case, the `this` is bound to the newly created object.

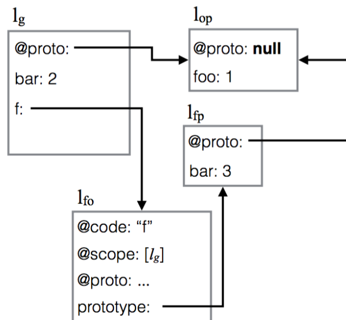




Function call: f()

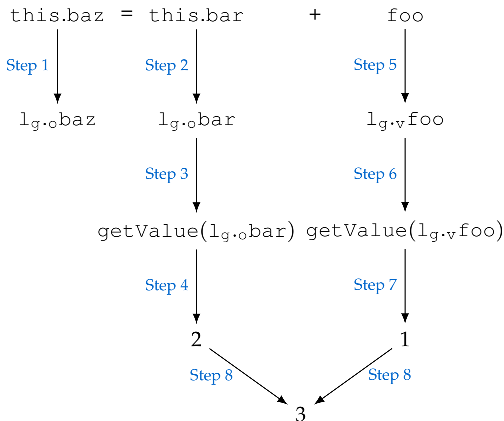
For function calls in strict mode, `this` is bound to `undefined`.

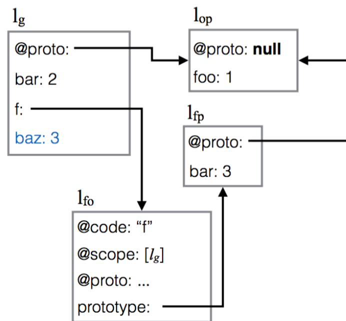




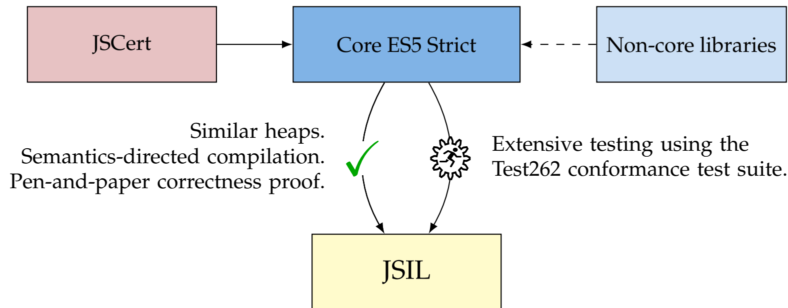
Method call: this.f()

In this case, `this` is bound to the global object.



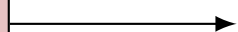


Trust: Proof and Testing



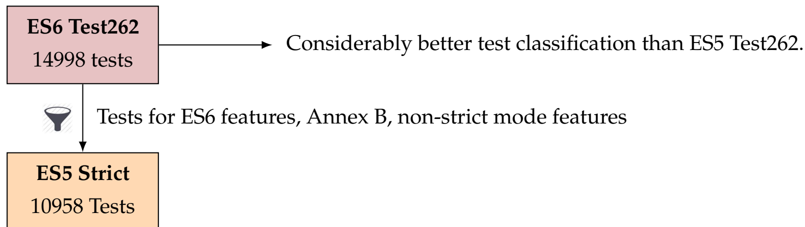
Testing Methodology and Results

ES6 Test262
14998 tests

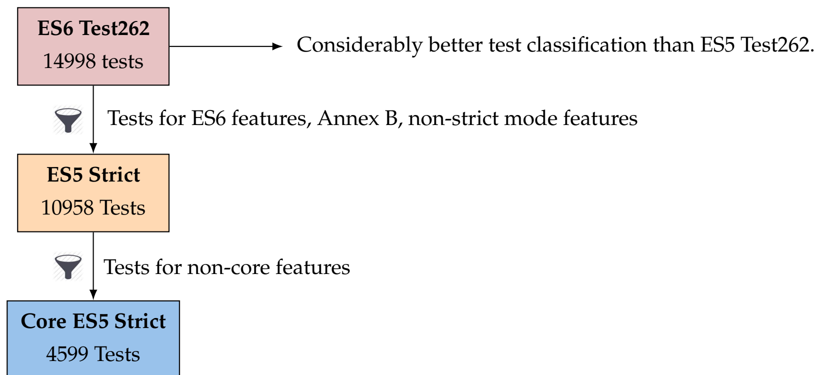


Considerably better test classification than ES5 Test262.

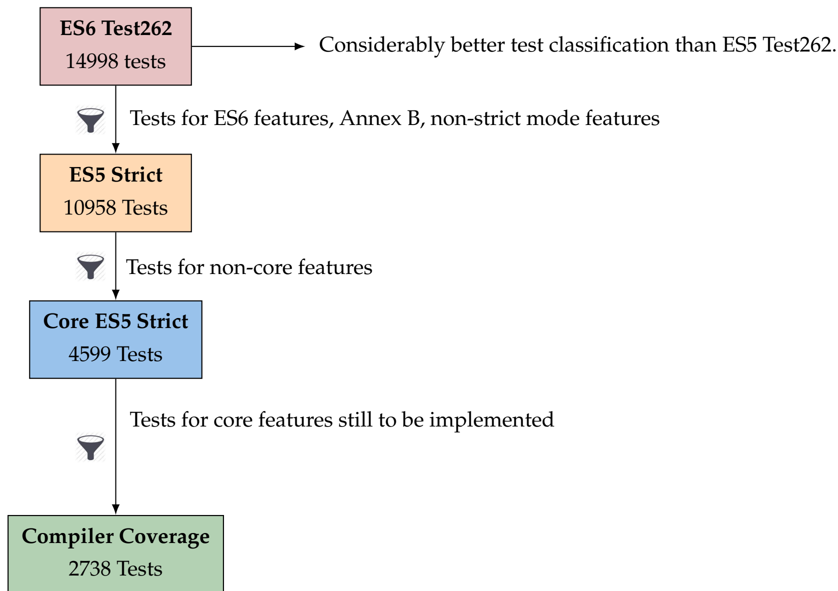
Testing Methodology and Results



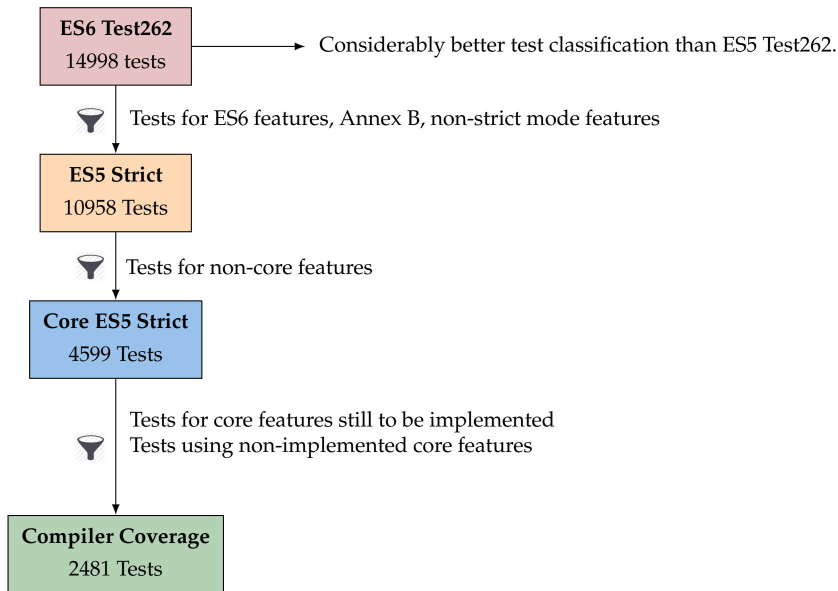
Testing Methodology and Results



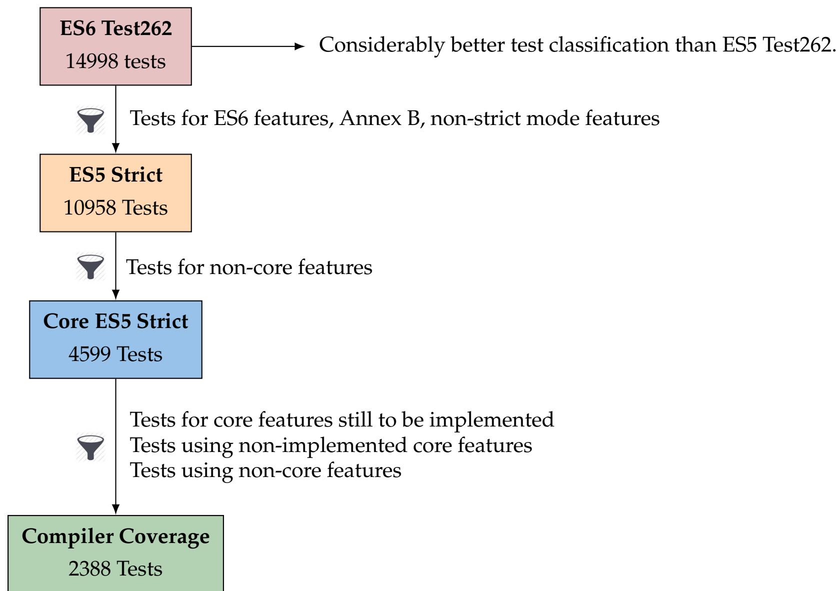
Testing Methodology and Results



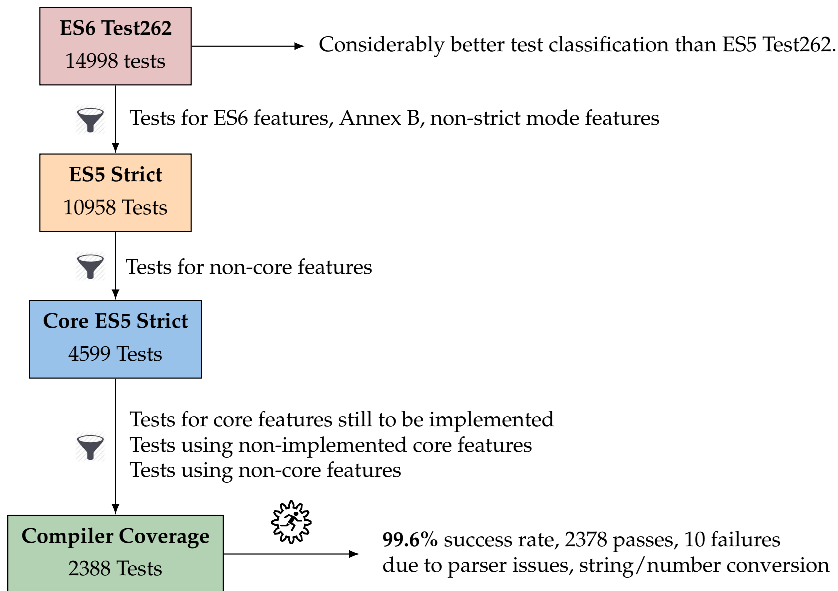
Testing Methodology and Results



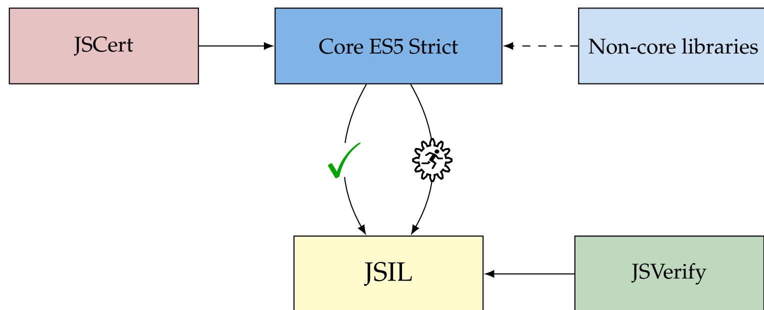
Testing Methodology and Results



Testing Methodology and Results



JSVerify: a Smallfoot-type Verification Tool for JSIL



JSVerify for Internal and Core Functions

JSVerify is used to specify and verify our JSIL implementations of the internal and core library functions, such as `getValue`.

Example: `getValue`

$$\{\ x \doteq L._{o}Y * \Pi(L :: LS, Y, Z)\ \}$$

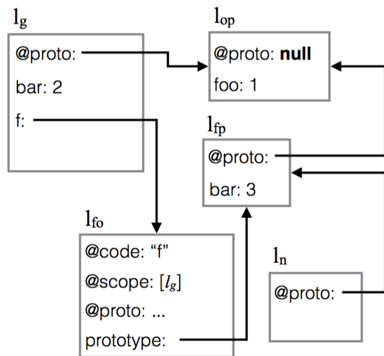
`getValue(x)`

$$\{\mathit{Precondition} * \texttt{ret} \doteq Z\}$$

`getValue(ln.obar)`

↓

3



JSVerify for Internal and Core Functions

JSVerify is used to specify and verify our JSIL implementations of the internal and core library functions, such as `getValue`.

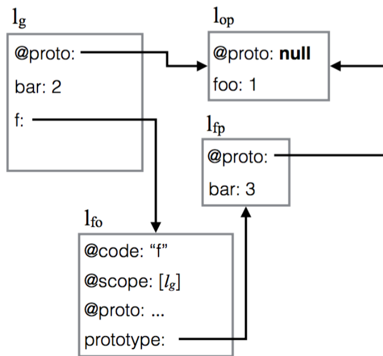
Example: `getValue`

$$\{\ \mathtt{x} \doteq \mathsf{undefined}._o\mathsf{Y}\ \}$$
$$\mathtt{getValue(x)}$$
$$\{\mathit{Precondition} * \mathtt{ret} \doteq \mathsf{error}\}$$

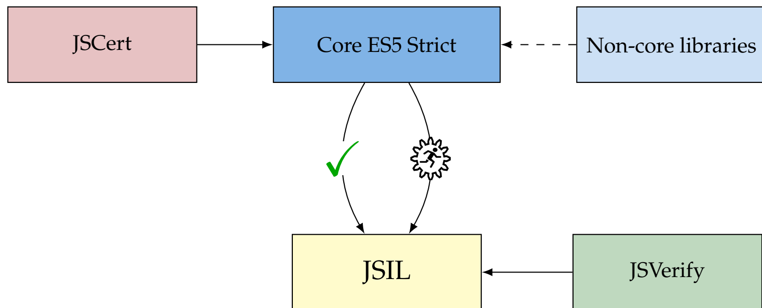
`getValue(undefined.`$_o$`bar)`

↓

ERROR



JSVerify for Non-core Functions



- Axiomatic specifications for non-core functions
- Reference implementation in JSIL or Core ES5 Strict, following the standard, verified with JSVerify, tested against Test262

JSIL as a Service