

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 JavaScript (the ECMAScript 3 standard, ES3), [Maffeis, Mitchell, Tally, 2008](#); mechanised definition of ES5, [us, 2014](#); [Park, Stefanescu, Rosu, 2015](#).

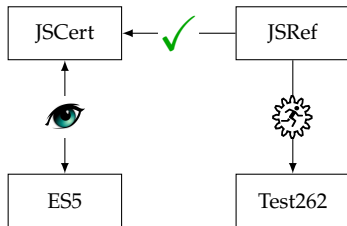
JavaScript at Imperial

- ✓ JSCert: a mechanised specification of ES5 in Coq



M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner,
S. Maffei, D. Naudžiūnienė, A. Schmitt, G. Smith.

A Trusted Mechanised JavaScript Specification, POPL 2014.



Inria Collaborators



A. Schmitt



A. Charguéraud



M. Bodin

JavaScript at Imperial

- ✓ JSCert: a mechanised specification of ES5 in Coq



M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner,
S. Maffei, D. Naudžiūnienė, A. Schmitt, G. Smith.
A Trusted Mechanised JavaScript Specification, POPL 2014.

- ✓ JSLogic: a program logic for JavaScript



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

JavaScript at Imperial

- ✓ JSCert: a mechanised specification of ES5 in Coq
- ✓ JSLogic: a program logic for JavaScript

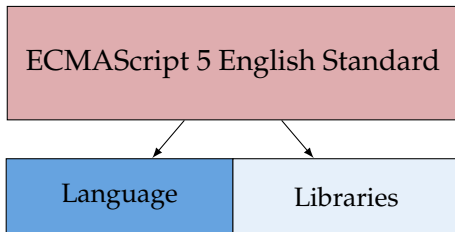
JSIL: an intermediate language for JavaScript

JSVerify: a verification tool for JavaScript

JavaScript and Verification

JavaScript

JavaScript and Verification



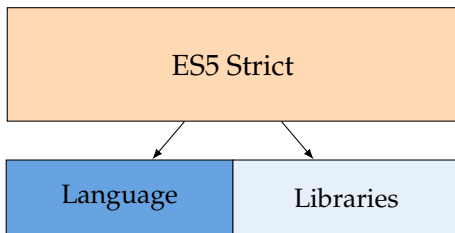
How is it organised?

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



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

JavaScript vs. JSIL Verification

- (1) $\mathcal{A}sm \vdash \{P\} e \{R_0 * r \doteq F_I\}$
- (2) $R_0 = \left(S_0 \models \text{This}(F_I, T) \models \gamma(Ls, F_I, F_E) * F_E \neq l_e * \right. \\ \left. (F_E, @body) \mapsto \lambda X_I, \dots, X_n. e' * (F_E, @scope) \mapsto Ls'_V \right)$
- (3.1) $\mathcal{A}sm \vdash \{R_0\} e_1 \{R_1 * l \doteq Ls_V * r \doteq V_I\} \quad R_1 = S_1 * \gamma(Ls_1, V_I, V_I')$
- $\vdots$
- (3.m) $\mathcal{A}sm \vdash \{R_{m-1}\} e_m \{R_m * l \doteq Ls_V * r \doteq V_m\} \quad R_m = S_m * \gamma(Ls_m, V_m, V_m')$
- (4) $\forall j \in \{m+1 \dots n\}. V_j' = \&undefined$
- (5) $R_m' = \left(\begin{array}{l} R_m * \exists L. l \doteq L : Ls'_V * \\ (L, X_I) \mapsto V_I' * \\ \vdots \\ (L, X_n) \mapsto V_n' * \\ (L, @this) \mapsto T * \\ (L, @proto) \mapsto \text{null} * \text{defs}(L, e', [X_I, \dots, X_n]) * \\ \text{decls}(e', YS) * \text{newobj}_L(\{ @proto, @this, X_I, \dots, X_n \} \cup YS) \end{array} \right)$
- (6) $\lambda X_1 \dots X_n. \{P_f\} e' \{Q_f\} \in \mathcal{A}sm$ (7) $l \notin \text{fv}(Q) \cup \text{fv}(R_m)$
- (8) $\lambda X_1 \dots X_n. \{P_f\} e' \{Q_f\} V_1' \dots V_n' = \{R_m'\} e' \{ \exists L. Q * l \doteq L : Ls'_V \}$
- $\mathcal{A}sm \vdash \{P\} e(e_1, \dots, e_m) \{ \exists L. Q * l \doteq Ls_V \}$

JavaScript

**Logic Rules of
Function / Procedure
Call**

JSIL

$$S(\text{fid}) = \lambda x_1, \dots, x_n. \{P\} m \{Q * \text{ret} \doteq E\} \quad \forall_{k < j \leq n} E_j = \text{undefined}$$

$$S \vdash \{P[E_1/x_1, \dots, E_n/x_n]\} x := \text{fid}(E_1, \dots, E_k) \{Q[E_1/x_1, \dots, E_n/x_n] * x \doteq E[E_1/x_1, \dots, E_n/x_n]\}$$

Incorporating Non-core Libraries

Non-core Libraries

Axiomatic specification to verify client programs

Does not follow the standard, which is operational

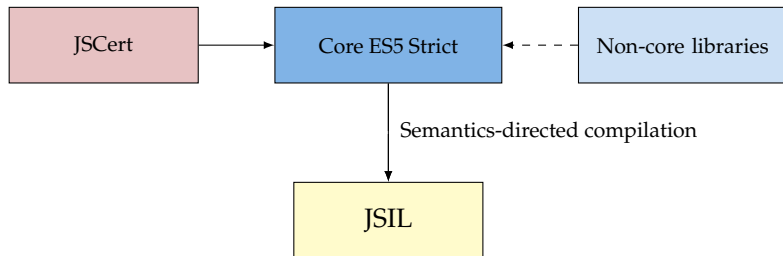
Justification of specifications

Informal appeal to the English standard

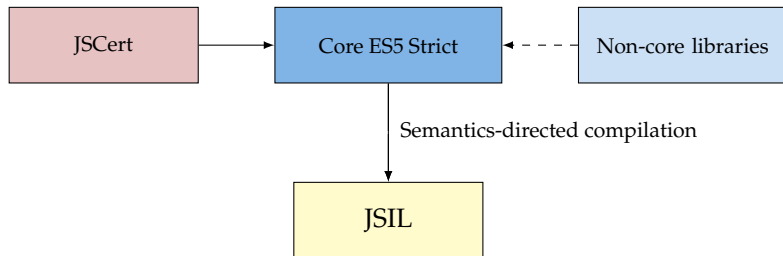
Reference implementation in JSIL or Core ES5 Strict,
following the standard, verified with JSVerify,
tested against Test262

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:

$$e ::= v \mid x \mid \ominus e \mid e \oplus e \mid \text{typeof}(e) \\ \mid e.o e \mid e.v e \mid \text{base}(e) \mid \text{field}(e)$$

Commands:

$$c ::= \text{skip} \mid x := e \mid x := \text{new}() \mid x := [e, e] \\ \mid [e, e] := e \mid \text{delete}(e) \mid x := \text{hasField}(e, e) \\ \mid x := \text{protoField}(e, e) \mid x := \text{protoObj}(e, e) \\ \mid \text{goto } i \mid \text{goto } [e] i, j \mid x := e(\bar{e}) \text{ with } j$$

The Syntax of JSIL

Expressions:

$$e ::= v \mid x \mid \ominus e \mid e \oplus e \mid \text{typeof}(e) \\ \mid e.o e \mid e.v e \mid \text{base}(e) \mid \text{field}(e)$$

Commands:

$$c ::= \text{skip} \mid x := e \mid x := \text{new}() \mid x := [e, e] \\ \mid [e, e] := e \mid \text{delete}(e) \mid x := \text{hasField}(e, e) \\ \mid x := \text{protoField}(e, e) \mid x := \text{protoObj}(e, e) \\ \mid \text{goto } i \mid \text{goto } [e] i, j \mid x := e(\bar{e}) \text{ with } j$$

Extensible objects, dynamic fields

The Syntax of JSIL

Expressions:

$$e ::= v \mid x \mid \ominus e \mid e \oplus e \mid \text{typeof}(e) \\ \mid e.o e \mid e.v e \mid \text{base}(e) \mid \text{field}(e)$$

Commands:

$$c ::= \text{skip} \mid x := e \mid x := \text{new}() \mid x := [e, e] \\ \mid [e, e] := e \mid \text{delete}(e) \mid x := \text{hasField}(e, e) \\ \mid x := \text{protoField}(e, e) \mid x := \text{protoObj}(e, e) \\ \mid \text{goto } i \mid \text{goto } [e] i, j \mid x := e(\bar{e}) \text{ with } j$$

Prototype chains

The Syntax of JSIL

Expressions:

$$e ::= v \mid x \mid \ominus e \mid e \oplus e \mid \text{typeof}(e) \\ \mid e.o e \mid e.v e \mid \text{base}(e) \mid \text{field}(e)$$

Commands:

$$c ::= \text{skip} \mid x := e \mid x := \text{new}() \mid x := [e, e] \\ \mid [e, e] := e \mid \text{delete}(e) \mid x := \text{hasField}(e, e) \\ \mid x := \text{protoField}(e, e) \mid x := \text{protoObj}(e, e) \\ \mid \text{goto } i \mid \text{goto } [e] i, j \mid x := e(\bar{e}) \text{ with } j$$

Dynamic function choice

The Syntax of JSIL

Expressions:

$$e ::= v \mid x \mid \ominus e \mid e \oplus e \mid \text{typeof}(e) \\ \mid e.o.e \mid e.v.e \mid \text{base}(e) \mid \text{field}(e)$$

Commands:

$$c ::= \text{skip} \mid x := e \mid x := \text{new}() \mid x := [e, e] \\ \mid [e, e] := e \mid \text{delete}(e) \mid x := \text{hasField}(e, e) \\ \mid x := \text{protoField}(e, e) \mid x := \text{protoObj}(e, e) \\ \mid \text{goto } i \mid \text{goto } [e] i, j \mid x := e(\bar{e}) \text{ with } j$$

Procedures: $\text{proc} ::= \text{proc } m(\bar{x})\{\bar{c}\}$

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

```
function f() { ... }
```

Global code

JSIL Code

```
proc f(xsc, xthis) { ... }
```

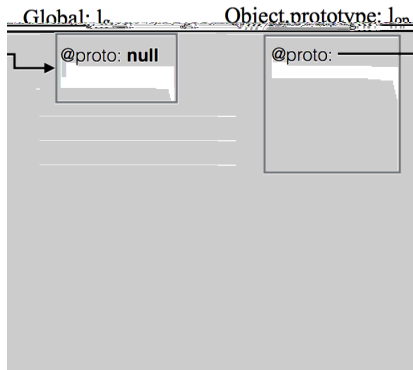
```
proc main() { ... }
```

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



Compiling ES5 Strict to JSIL

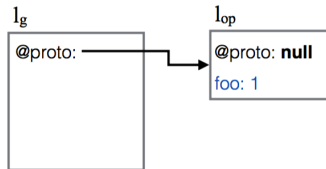
JavaScript Code

```
Object.prototype.foo = 1;  
var bar = 2;  
  
function f() {  
    this.baz = this.bar + foo;  
}  
  
f.prototype.bar = 3
```

JSIL Code

```
[lop, "foo"] := 1
```

Heap



Compiling ES5 Strict to JSIL

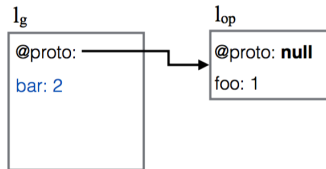
JavaScript Code

```
Object.prototype.foo = 1;  
var bar = 2;  
function f() {  
    this.baz = this.bar + foo;  
}  
f.prototype.bar = 3
```

JSIL Code

```
[lg, "bar"] := 2
```

Heap



Compiling ES5 Strict to JSIL

JavaScript Code

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

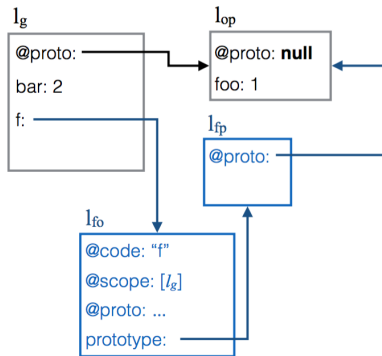
```
function f() {  
    this.baz = this.bar + foo;  
}
```

```
f.prototype.bar = 3
```

JSIL Code

```
xfp := new()  
[xfp, @proto] := lop  
xfo := new()  
[xfo, @code] := "f"  
[xfo, @scope] := [lg]  
[xfo, @proto] := ...  
[xfo, "prototype"] := xfp  
[lg, "f"] := xfo
```

Heap



Compiling ES5 Strict to JSIL

JavaScript Code

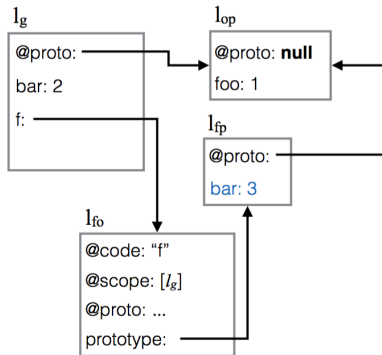
```
Object.prototype.foo = 1;  
var bar = 2;  
  
function f() {  
    this.baz = this.bar + foo;  
}
```

`f.prototype.bar = 3`

JSIL Code

`[xfp, "bar"] := 3`

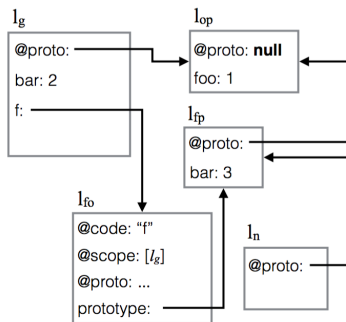
Heap



Constructor call: new f()

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

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



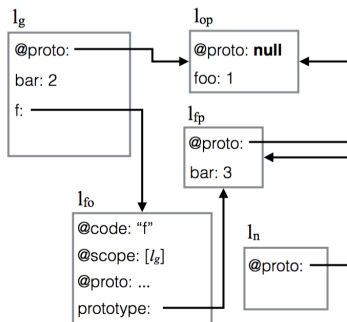
Constructor call: new f()

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

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

Step 1

```
l_n.o.baz
```



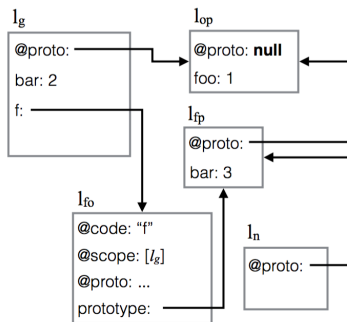
Constructor call: new f()

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

`this.baz = this.bar + foo`

Step 1 ↓ Step 2 ↓

`l_n.o.baz` `l_n.o.bar`



Constructor call: new f()

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

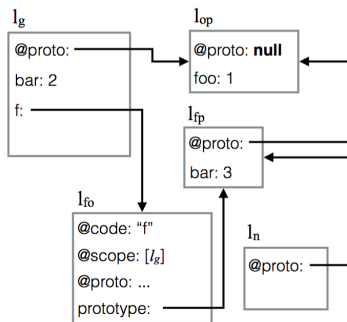
`this.baz = this.bar + foo`

Step 1 ↓ Step 2 ↓

`ln.obaz` `ln.obar`

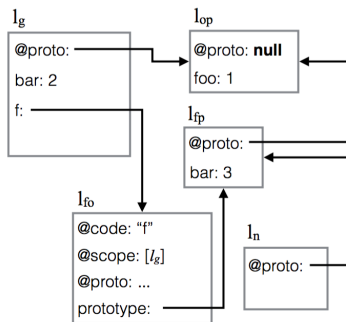
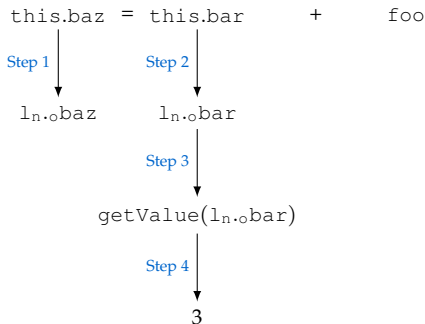
Step 3 ↓

`getValue(ln.obar)`



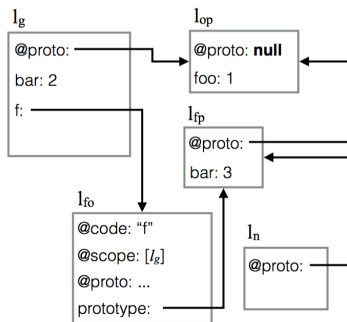
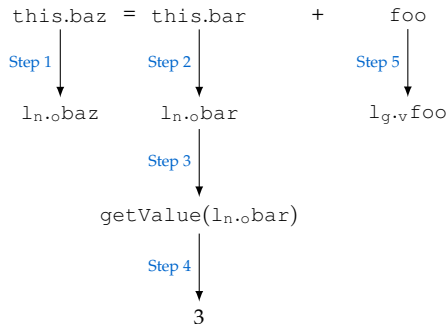
Constructor call: new f()

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



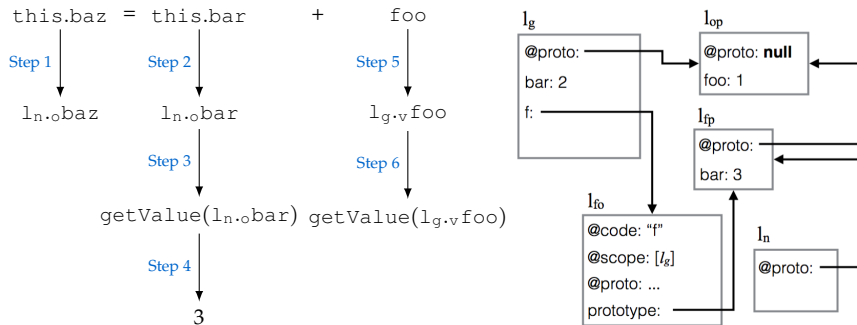
Constructor call: new f()

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



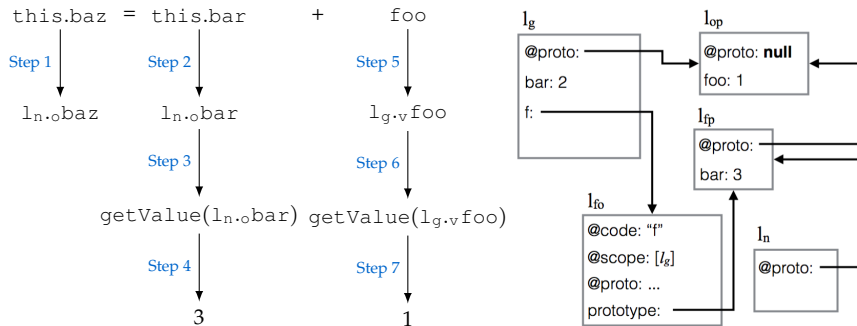
Constructor call: new f()

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



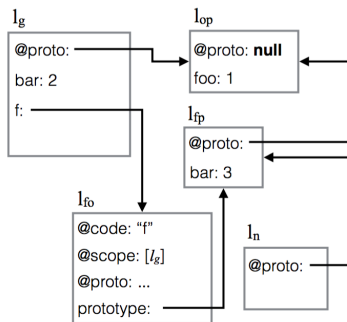
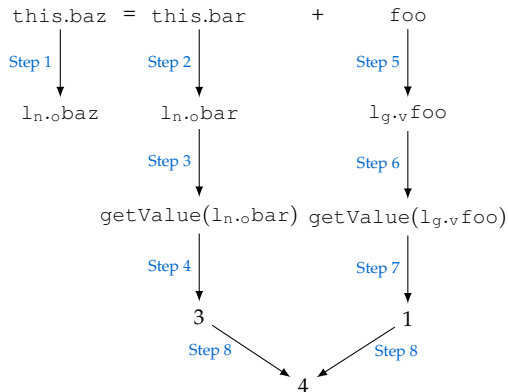
Constructor call: new f()

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



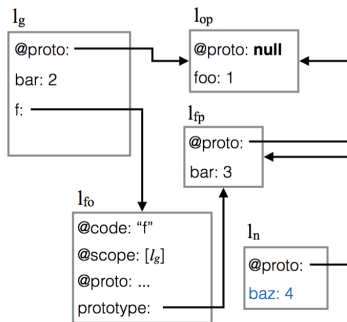
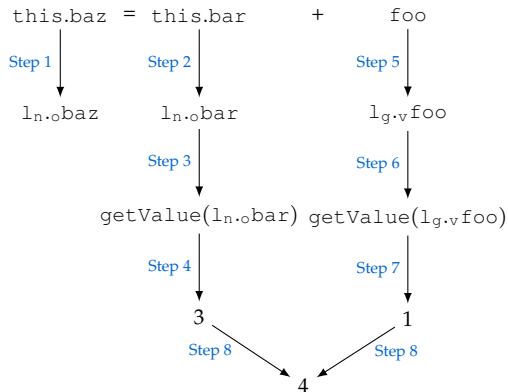
Constructor call: new f()

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



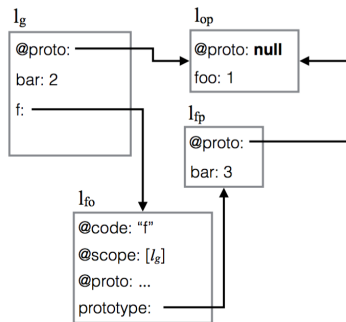
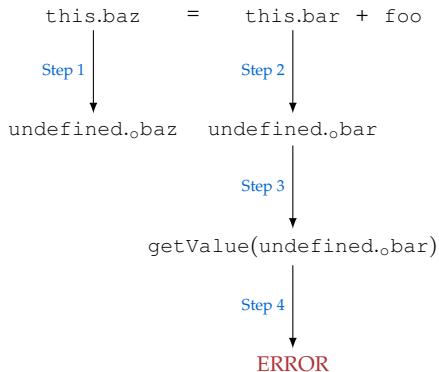
Constructor call: new f()

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



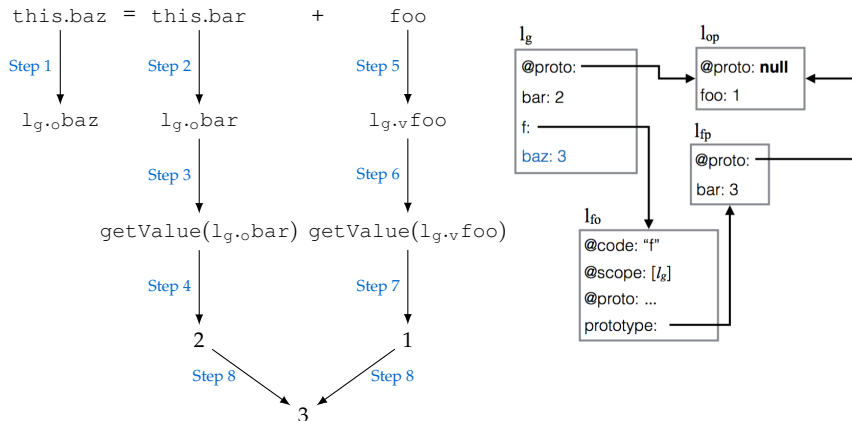
Function call: f()

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

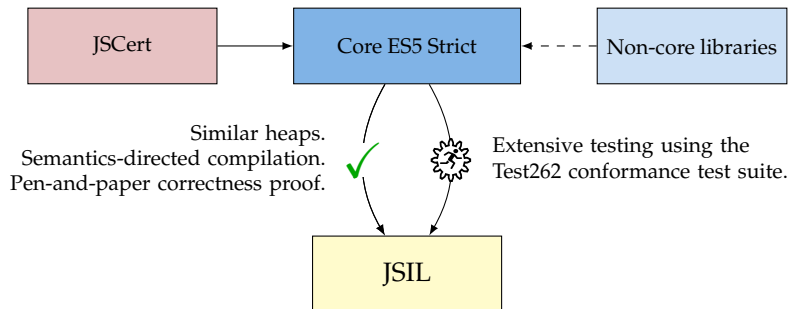


Method call: this.f()

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



Trust: Proof and Testing



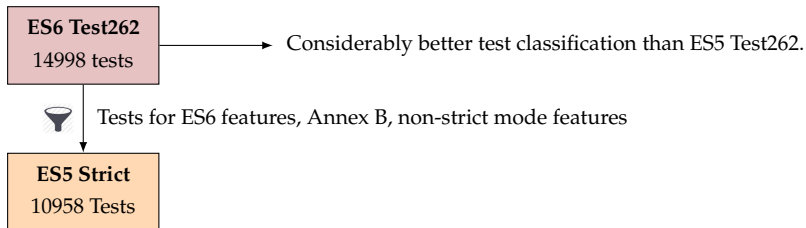
Testing Methodology and Results

ES6 Test262

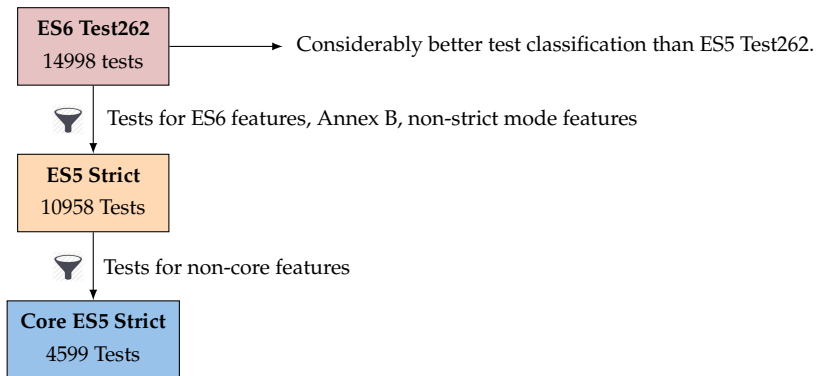
14998 tests

Considerably better test classification than ES5 Test262.

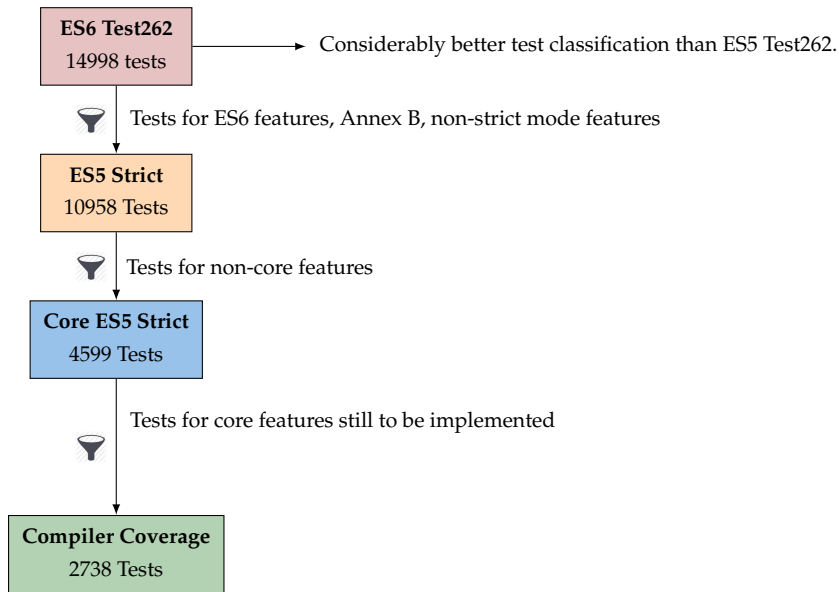
Testing Methodology and Results



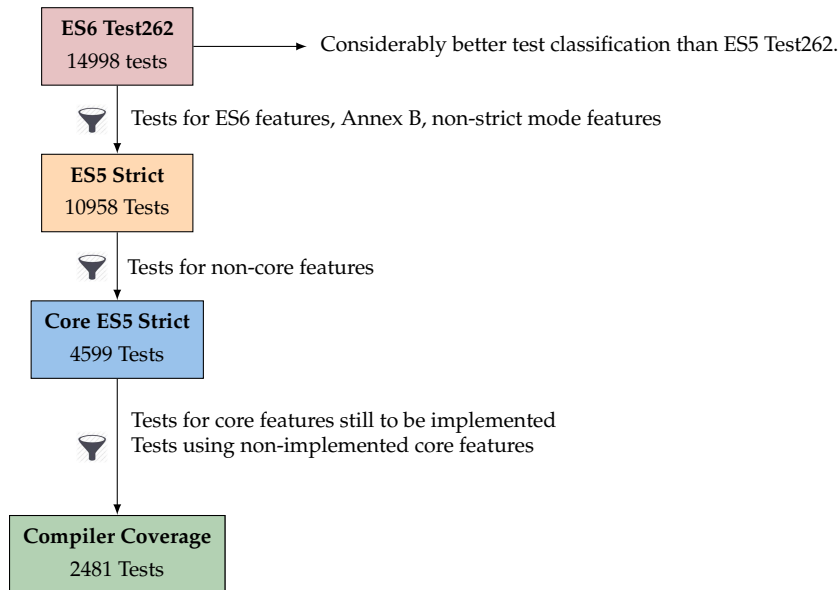
Testing Methodology and Results



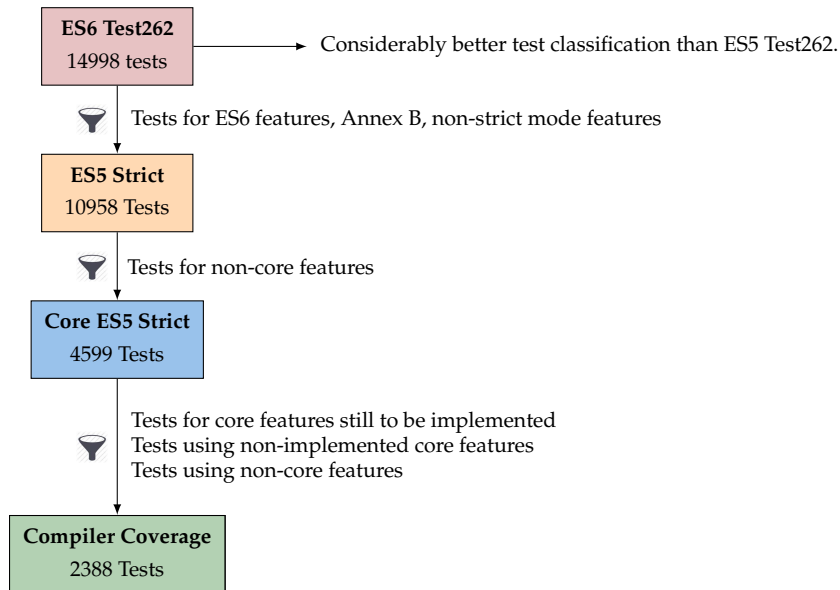
Testing Methodology and Results



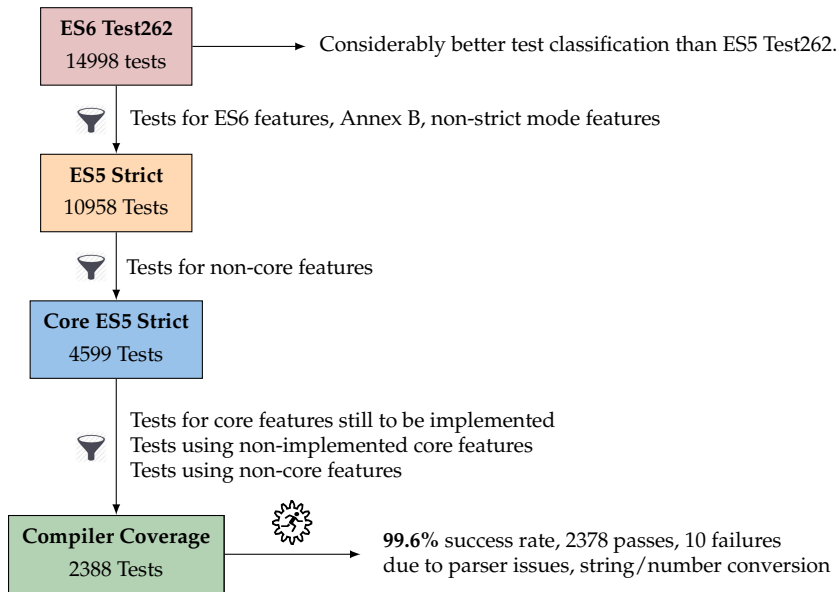
Testing Methodology and Results



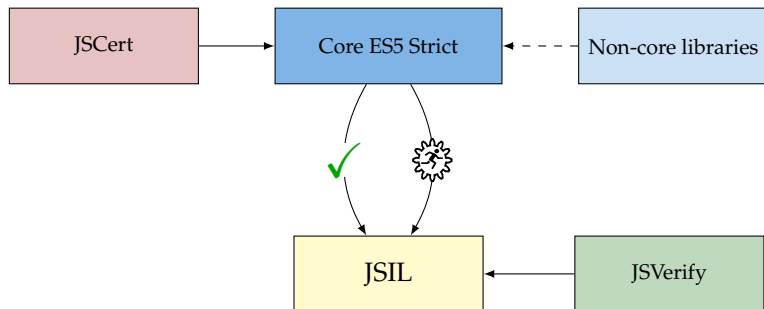
Testing Methodology and Results



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`

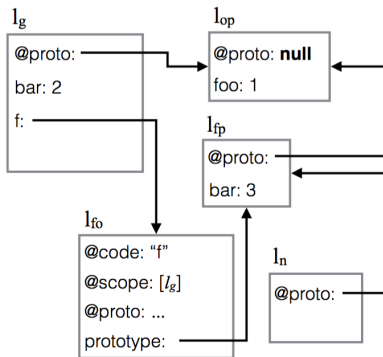
$$\{ \ x \doteq L.oY * \Pi(L :: LS, Y, Z) \ }$$

`getValue(x)`

$$\{ \textit{Precondition} * \textit{ret} \doteq Z \}$$

`getValue(ln.obar)`

↓
3



JSVerify for Internal and Core Functions

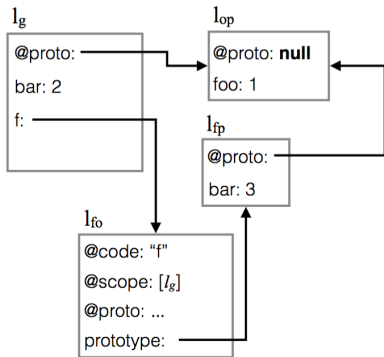
JSVerify is used to specify and verify our JSIL implementations of the internal and core library functions, such as `getValue`.

Example: `getValue`

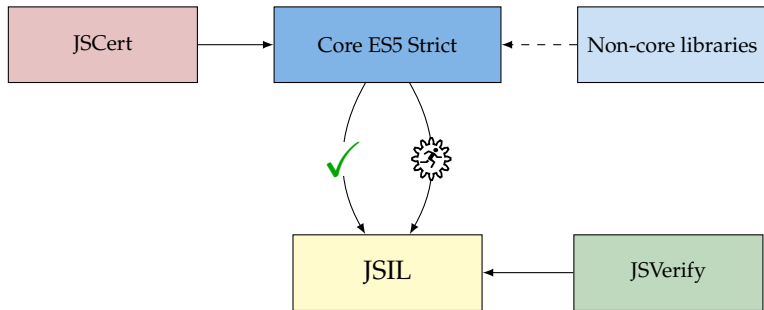
```
{ x ≐ undefined.oY }  
  getValue(x)  
{Precondition * ret ≐ error}
```

`getValue(undefined.o.bar)`

ERROR

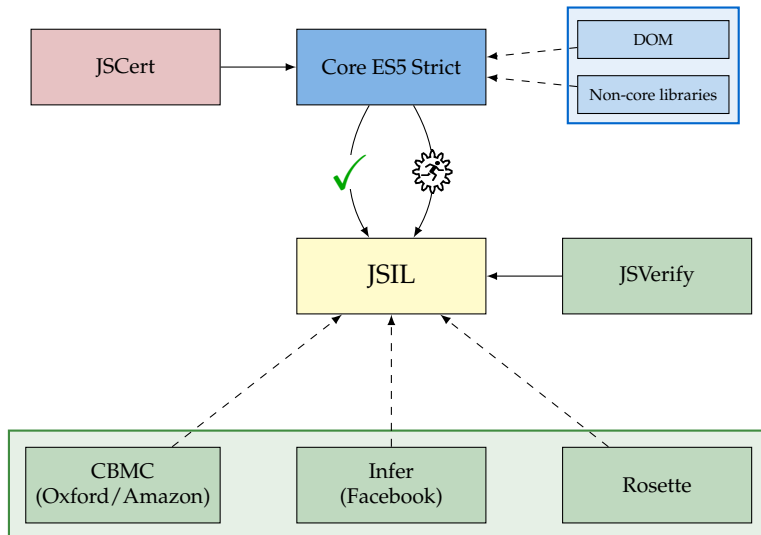


JSVerify for Non-core Functions



- 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



My supervisors



Gordon Plotkin



John Power