

# An Infrastructure for Tractable Verification of JavaScript

## Program Specification and Verification Group

Imperial College London



Philippa Gardner



José Frago Santos



Petar Maksimović



Daiva Naudžiūnienė

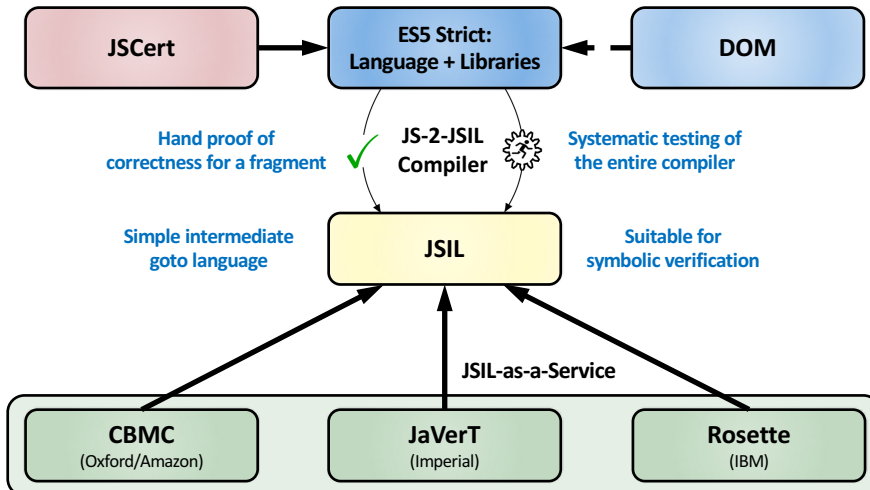


Azalea Raad



Thomas Wood

### Mechanised specification of the ES5 Standard



The dynamic nature of JavaScript and its complex semantics make it a difficult target for verification. To address this issue, we develop JS-2-JSIL, a compiler from JavaScript (ECMAScript 5 strict) to a simple intermediate goto language, JSIL. We design JS-2-JSIL to be step-by-step faithful to the ECMAScript standard, systematically testing it against the official ECMAScript conformance test suite, and proving it correct with respect to a fragment of the ES5 Strict operational semantics. We develop JSIL logic, a separation logic for JSIL that we prove sound with respect to its operational semantics, and a semi-automatic verification tool based on this logic. Together, these results allow us to verify Hoare triples for JavaScript using JSIL logic and JS-2-JSIL.

We establish the infrastructure required for verifying JavaScript code. We focus on the internal functions underlying JavaScript, which provide essential functionalities and are used in the semantics of all JavaScript commands. We implement all of these functions in JSIL, give functionally correct specifications for most of them, and verify, using our tool, that their JSIL implementations are correct with respect to these specifications. We are able to translate JavaScript triples to JSIL triples and verify the JSIL triples using our tool.

### JSIL

Goto language: extensible objects, dynamic field access and function calls

$m \in \text{Str}$   $n \in \text{Num}$   $b \in \text{Bool}$   $l \in \mathcal{L}$   
 $x \in \mathcal{X}_{\text{JSIL}}$   $\lambda \in \mathcal{L}it ::= n \mid b \mid m \mid \text{undefined} \mid \text{null}$   
 $t \in \text{Types} ::= \text{Num} \mid \text{Bool} \mid \text{Str} \mid \text{Undef} \mid \text{Null} \mid \text{Empty} \mid \text{Obj} \mid \text{List} \mid \text{Type}$   
 $v \in \mathcal{V}_{\text{JSIL}} ::= \lambda \mid l \mid \text{empty} \mid \text{error} \mid t \mid \bar{v}$   
 $e \in \mathcal{E}_{\text{JSIL}} ::= v \mid x \mid \odot e \mid e \odot e \mid \text{typeof}(e) \mid \bar{e} \mid \text{nth}(e, e)$   
 $bc \in \text{BCmd} ::= \text{skip} \mid x := e \mid x := \text{new}() \mid x := [e, e] \mid [e_1, e_2] := e_3 \mid \text{delete}(e, e) \mid x := \text{hasField}(e, e) \mid x := \text{getFields}(e)$   
 $c \in \text{Cmd} ::= bc \mid \text{goto } i \mid \text{goto } [e] \mid i, j \mid x := e \odot \bar{e} \mid \text{with } j \mid x := \phi(x)$

Built-in  $\phi$ -nodes provide support for SSA  
We have implemented all JavaScript internal functions and most of the built-in libraries directly in JSIL (~3Kloc)

### Testing

Systematic testing of the full JS-2-JSIL compiler against the official ECMAScript

Test262 test suite  
ES6 version of the test suite due to major deficiencies in the ES5 version  
Rigorous filtering and analysis  
Continuous-integration infrastructure, tests running en masse in parallel

ES5 Strict Tests	10469
Tests for non-impl. features	1297
Compiler Coverage	9160
ES5-ES6 differences	345
Tests using non-impl. features	30
Applicable Tests	8797
Passed tests	8797
Failed tests	0

### Separation-logic-based reasoning

#### JS Logic (POPL '12)

- ✓ Soundness proof
- ✓ Complex proof rules
- ! Difficult to automate

#### JSIL Logic (now)

- ✓ Soundness proof
- ✓ Simple proof rules
- ✓ Can be automated

The complexity of JavaScript has moved to the translation!

Procedure Call  
 $\text{Proc}(i) = x := e_0(e_1, \dots, e_n)$   $S(m) = \{P\} m(x_1, \dots, x_n) [Q \mid x_{n+1} := e] \quad \forall x_{n+1} \in e_j = \text{undefined}$   
 $\{P, S, i\} \vdash m, (P(e_1/x_1, \dots, e_n/x_n) \wedge e_0 \wedge m) \leadsto \{Q(e_1/x_1, \dots, e_n/x_n) \wedge e_0 \wedge m \mid x := e_0(e_1/x_1, \dots, e_n/x_n)\}$

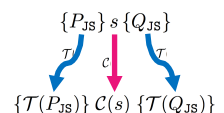
### Translating assertions from JS logic to JSIL logic

Non-trivial cases:

- Variable resolution
- Property descriptors
- Function objects
- The this object

#### Correctness of the translation for assertions

$H_{JS}, L, \epsilon_{JS} \models P_{JS} \Leftrightarrow H_{JSIL}, \rho, \tau(\epsilon_{JS}) \models \tau(P_{JS})$  when  $H_{JS}, L \sim_{\beta} H_{JSIL}, \rho$



### JS-2-JSIL

Compilation follows English standard line-by-line; correctness shown via hand proof and testing  
Correctness allows JSIL analysis to lift to JavaScript analysis

In the ES5 standard:

11.4.1 The Addition operator (+)  
The addition operator either performs string concatenation or numeric addition.  
The production  $\text{AdditiveExpression} ::= \text{AdditiveExpression} * \text{MultiplicativeExpression}$  is evaluated as follows:  
1. Let  $l$  be the result of evaluating  $\text{AdditiveExpression}$ .  
2. Let  $l$  be  $\text{getVValue}(l)$ .  
3. Let  $r$  be the result of evaluating  $\text{MultiplicativeExpression}$ .  
4. Let  $r$  be  $\text{getVValue}(r)$ .  
5. Let  $r$  be  $\text{toPrimitive}(r)$ .  
6. Let  $r$  be  $\text{toNumber}(r)$ .  
7. If  $\text{Type}(l)$  is String or Type( $r$ ) is String, then  
a. Return the result of concatenating  $\text{ToString}(l)$  and  $\text{ToString}(r)$ .  
8. Return the result of applying the addition operation to  $\text{ToNumber}(l)$  and  $\text{ToNumber}(r)$ . (See the Note below 11.4.1.)

Compiled to JSIL:

```
1 x1 := ref-v(lg, "+")
2 x1_v := getVValue(x1) with err
3 x2 := ref-v(lg, "r")
4 x2_v := getVValue(x2) with err
5 x1_p := toPrimitive(x1_v) with err
6 x2_p := toPrimitive(x2_v) with err
7 x_p := (typeof(x1_v) = String) or
8 (typeof(x2_v) = String)
9 then:
10 x1_s := toString(x1_p) with err
11 x2_s := toString(x2_p) with err
12 x_r := x1_s concat x2_s
13 goto done
14 else:
15 x1_n := toNumber(x1_p) with err
16 x2_n := toNumber(x2_p) with err
17 x_n := x1_n + x2_n
18 done: x_r := PH(K_r, then, K_n, x_n)
```

### JaVerT

Semi-automatic verification tool for JSIL  
Based on symbolic execution and JSIL Logic  
User specifies pre- and post-conditions, loop invariants, and predicate manipulations in JS Logic; all automatically translated to JSIL Logic  
Current state: JaVerT can verify a substantial fragment of JavaScript's internal functions and very simple JavaScript programs

#### JSIL-as-a-Service

JSIL front-end to enable verification tools to analyse JavaScript

CBMC, with Kroening and Tautschnig at Amazon

Rosette, with Julian Dolby at IBM

