

An Infrastructure for Tractable Verification of JavaScript

Program Specification and Verification Group

Imperial College London



Philippa Gardner



José Fragoso Santos



Petar Maksimović



Daiva Naudžiūnienė

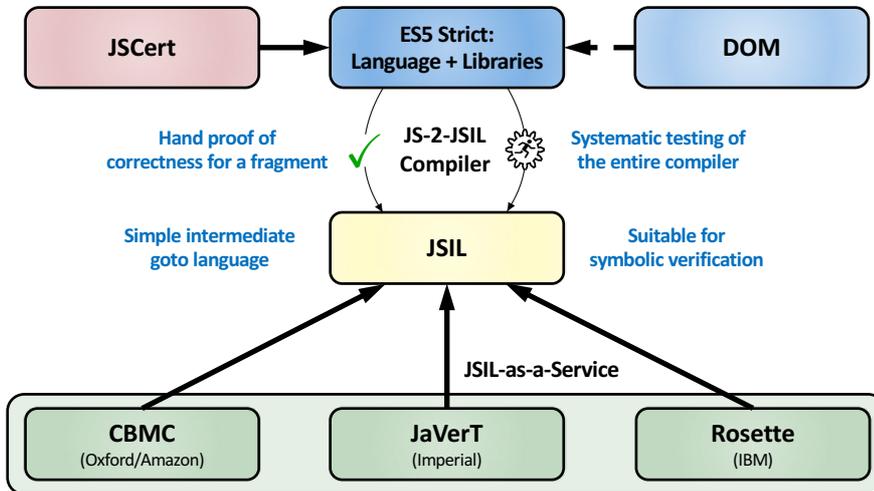


Azalea Raad



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

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

Built-in ϕ -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) **JSIL Logic (now)**
 ✓ Soundness proof ✓ Soundness proof
 ✓ Complex proof rules ✓ Simple proof rules
 ! Difficult to automate ✓ Can be automated

The complexity of JavaScript has moved to the translation!

(Function Call)

$$R_1 = \left(\begin{array}{l} \{P\} \text{e1} \{R_1 \rightarrow r \in F_1\} \\ S_1 \text{e1} \text{This}(F_1, T) \text{e1} \gamma(L_{S_1}, F_1, F_2) \text{e} \\ (F_2, \text{body}) \rightarrow \lambda X. \text{e2} \{F_2, \text{scope}\} \rightarrow L_{S_2} \\ \{R_1\} \text{e2} \{R_2 \rightarrow l \in L_{S_2} \wedge r \in V_1\} \quad R_2 = S_2 \gamma(L_{S_2}, V_1, V_2) \end{array} \right)$$

$$R_3 = \left(\begin{array}{l} R_2 \oplus \exists L. 1 \leq l \leq L. L_2 \oplus (L, X) \rightarrow V_2^* \\ (L, \text{this}) \rightarrow T^* \\ (L, \text{proto}) \rightarrow \text{null} \mid \text{def}(X, L, \text{e3}) \\ \text{newobj}(L, \text{proto}, \text{this}, X, \text{decs}(X, L, \text{e3})) \\ \{R_3\} \text{e3} \{ \exists L. Q \oplus 1 \leq l \leq L_2 \} \quad 1 \notin \text{fr}(Q, R_2) \end{array} \right)$$

{P} e1 (e2) {∃L. Q ⊕ 1 ≤ l ≤ L₂}

Procedure Call

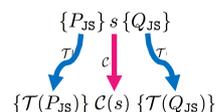
$$\text{Proc}(l) = x := \text{e}_1(\text{e}_2, \dots, \text{e}_n) \quad S(m) = \{P\} m(x_1, \dots, x_n) \{Q \oplus x_m \oplus e\} \quad \forall k \leq n. \text{e}_k = \text{undefined}$$

$$P, S, 1 \vdash m. \{P\}(\text{e}_1, x_1, \dots, \text{e}_n, \text{e}_m) \rightarrow m' \sim \{Q\}(\text{e}_1, x_1, \dots, \text{e}_n, \text{e}_m) \rightarrow m' \oplus x \oplus \text{e}(l) \mid x_1, \dots, \text{e}_n(x_n)$$

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

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

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 'AdditiveExpression : AdditiveExpression * MultiplicativeExpression' is evaluated as follows:
1. Let l be the result of evaluating 'AdditiveExpression'.
2. Let r be the result of evaluating 'MultiplicativeExpression'.
3. Let t be the result of evaluating 'AdditiveExpression'.
4. Let t be the result of evaluating 'AdditiveExpression'.
5. Let t be the result of evaluating 'AdditiveExpression'.
6. Let t be the result of evaluating 'AdditiveExpression'.
7. If Type(t) is String or Type(r) is String, then
   a. Return the String that is the result of concatenating ToString(l) and ToString(r).
8. Return the result of applying the addition operation to ToNumber(l) and ToNumber(r). See the Note below 11.4.5.
    
```

Compiled to JSIL:

```

1 x1 := ref+((l, r))
2 x1_v := getVValue(x1) with err
3 x2 := ref+((l, 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_p) = String) or
   (typeof(x2_p) = String)
8
9 goto done
10 then x1_s := toString(x1_p) with err
11 x2_s := toString(x2_p) with err
12 x_s := x1_s concat x2_s
13 goto done
14
15 x2_p := toNumber(x2_p) with err
16 x2_s := toString(x2_p) with err
17 x_s := x1_s concat x2_s
18
19 done: x_s := Phi(x_s, then, x_s, x1_p)
    
```

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