Formal JavaScript

Alan Schmitt

June 12, 2019
Thank you for the invitation
Motivation
How do you trust your software?
How do you trust your software?

- Manual verification
  - does not scale
How do you trust your software?

- Manual verification
  - does not scale
- Automatic bug finder
  - may miss some bugs

manual verification       bug finders
a long time ago           yesterday
How do you trust your software?

- Manual verification
  - does not scale
- Automatic bug finder
  - may miss some bugs
- Automatic, sound verifier
  - show the absence of bugs, may raise false alarms
  - ex: the Astrée static analyzer

manual verification  bug finders  sound verifiers
a long time ago  yesterday  today
How do you trust the tool that verifies your software?

- Manual verification
  - does not scale
- Automatic bug finder
  - may miss some bugs
- Automatic, sound verifier
  - show the absence of bugs, may raise false alarms
  - ex: the Astrée static analyzer

http://www.astree.ens.fr/

manual verification | bug finders | sound verifiers
---|---|---
a long time ago | yesterday | today

"Air France Airbus A380-800 F" by BriYYZ originally posted to Flickr as Arriving LAX north complex, credit BriYYZ, link
How do you trust the tool that verifies your software?

- **Manual verification**
  - does not scale
- **Automatic bug finder**
  - may miss some bugs
- **Automatic, sound verifier**
  - show the absence of bugs, may raise false alarms
    - ex: the Astrée static analyzer
- **Formally-verified verifier**
  - the verifier comes with a soundness proof
    - that is machine checked

---

Manual verification | Bug finders | Sound verifiers | Verified verifiers
---|---|---|---
A long time ago | Yesterday | Today | Tomorrow

http://www.astree.ens.fr/
Certified Analyses for JavaScript
Why JavaScript?

1. JavaScript is everywhere
2. JavaScript matters for web security
3. JavaScript is complex
4. JavaScript comes with a specification
What is JavaScript?
Netscape Communications realized that the Web needed to become more dynamic. Marc Andreessen, the founder of the company believed that HTML needed a “glue language” that was easy to use by Web designers and part-time programmers to assemble components such as images and plugins, where the code could be written directly in the Web page markup. – Wikipedia
JavaScript as Assembly of the Web

Emscripten is a toolchain for compiling to asm.js and WebAssembly, built using LLVM, that lets you run C and C++ on the web at near-native speed without plugins.

<table>
<thead>
<tr>
<th>Porting</th>
<th>APIs</th>
<th>Fast</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile your existing projects written in C or C++ and run them on all modern browsers.</td>
<td>Emscripten converts OpenGL into WebGL, and lets you use familiar APIs like SDL, or HTML5 directly.</td>
<td>Thanks to LLVM, Emscripten, asm.js and WebAssembly, code runs at near-native speed.</td>
</tr>
</tbody>
</table>

asm.js
an extraordinarily optimizable, low-level subset of JavaScript
JavaScript and Web Security
A Dangerous Situation
Mitigating Leaks

- sandbox (scripts cannot read or write files)
- same-origin policy (scripts cannot load scripts from somewhere else)
Hackers steal card data from 201 online campus stores from Canada and the US

Magecart group breached PrismRBS and modified the PrismWeb e-commerce platform.


A group of hackers has planted malicious JavaScript code that steals payment card details inside the e-commerce system used by colleges and universities in Canada and the US.

The malicious code was found on 201 online stores that were catering to 176 colleges and universities in the US and 21 in Canada, cyber-security Trend Micro said in a report released on Friday.

The attack is what security researchers call a Magecart attack --which consists of hackers placing malicious JavaScript code on the checkout and payment pages of online stores to record payment card data, which they later upload to their servers, and re-sell on underground cybercrime forums.
An attacker that manages to alter a JavaScript object prototype can severely impact how data is processed by the rest of the application, and open the door for more dangerous attacks, such as application crashes (denial of vulnerability bugs) or application hijacks (code execution flaws).
Javascript, the Language
Imperative and Functional

Variables

```javascript
var x = 4
x = (10 * 4) + 2
console.log(x)
```

⇒ 42

Functions are Values

```javascript
var f = function (g, x) { return (g(x) + 2) }

var fgx = f(function (y){ return (10 * y)}, 4)

console.log("f(g, x) = "+ fgx)
```

⇒ f(g, x) = 42
Objects

Literal Objects

```javascript
var obj = { a : 1, b : 2 } /* litéral */
console.log (obj.a) /* accès */
```

⇒ 1

Functions as Object Factories

```javascript
function f(a) { this.x = a }
var o = new f(42)
console.log (o.x)
```

⇒ 42
Prototypes and Object Factories

```javascript
function f(a) {
  this.x = a
}
var p = {y : 1}
f.prototype = p

var o = new f(42)

console.log("o.x = " + o.x + ", o.y = " + o.y)
```

⇒ o.x = 42, o.y = 1

**Warning:** the prototype field of a function becomes the `__proto__` field of the object created
A Language with (almost) no errors

Complex syntactic rules + automatic conversions ⇒ 😊

Blocks vs Objects

```javascript
var x = eval("{} + {}")
var y = eval("{} + {}")
console.log("x = " + x + "; y = " + y)
```

⇒ x = NaN; y = [object Object][object Object]

JSf*ck

```javascript
var x = (![]+[])[+++][]
    + (![]+[])[+++]+++++
    + (![]+[])[+++]+++++
    + (![]+[])[+++]+++++
    + (![]+[])[+++]+++++
console.log(x)
```

⇒ alan
A Language with (almost) no errors

Block vs Objects

```javascript
var x = eval("{} + {}")
var y = eval("({} + {})")
console.log("x = " + x + " y = " + y);
```

⇒ x = NaN; y = [object Object] [object Object]

JSF*ck

```javascript
var x = (![]+[])[+!+][]
  + (![]+[])[!+!+]
  + (![]+[])[+!+[]]
  + ([][[]]+[])[+!+[]]
console.log(x);
```

⇒ alan

Wat

@garybernhardt
(\texttt{![]}+[])\[+!+[]\] is 'a'

$$\begin{array}{|c|c|}
\hline
\text{[]} & \text{empty array} \\
\hline
\text{![]} & \text{nondation (converts to boolean)} \\
\hline
\text{[]+[]} & \text{concatenation (converts to string)} \\
\hline
\text{false} & \text{"false"} \\
\hline
\text{[]} & \text{empty array} \\
\hline
\text{+[]} & \text{conversion to number} \\
\hline
\text{true} & \text{0} \\
\hline
\text{!+[]} & \text{nondation} \\
\hline
\text{1} & \text{conversion to number} \\
\hline
\text{('+[]+[])\[+!+[]\]} & \text{array access} \\
\hline
\text{'a'} & \text{} \\
\hline
\end{array}$$

\(^1\text{Everything is true except false, 0, NaN, ",", null and undefined}\)
```javascript
var o = {};

o.toString = function () {
    o.toString = function () {
        return "😊"
    }
    return "😊"
}

console.log("I test : " + o);
console.log("It's all good, I can use it: " + o);

⇒ I test : 😊
    It's all good, I can use it: 😈
```
Integration to Web Pages

- navigation

```html
<input action="action" type="button" value="Back"
       onclick="history.go(-1);" />
```

- content modification (DOM)

```javascript
document.title = "New title"
var para = document.createElement("p")
para.appendChild(document.createTextNode("Hello World!"))
document.getElementsByTagName("body")[0].appendChild(para)
```

The Event Loop

- JavaScript is a **sequential** language (for the moment)
- Pervasive use of **callbacks** and **asynchronous promises**
JavaScript, the Specification
A quick history of JavaScript and ECMAScript

1995  Brendan Eich hired by Netscape to embed Scheme
May 1995  as Java is included in Netscape, scripting should have a similar syntax;
          JavaScript prototype developed in 10 days
Dec. 1995  JavaScript deployed in Netscape Navigator 2.0 beta 3
Aug. 1996  JScript deployed in Internet Explorer 3.0
A quick history of JavaScript and ECMAScript

1995  Brendan Eich hired by Netscape to embed Scheme

May 1995  as Java is included in Netscape, scripting should have a similar syntax; JavaScript prototype developed in 10 days

Dec. 1995  JavaScript deployed in Netscape Navigator 2.0 beta 3

Aug. 1996  JScript deployed in Internet Explorer 3.0

code is supposed to run identically in every browser

⇒ strong need for standardization
A quick history of JavaScript and ECMAScript

1995 Brendan Eich hired by Netscape to embed Scheme
May 1995 as Java is included in Netscape, scripting should have a similar syntax; JavaScript prototype developed in 10 days
Dec. 1995 JavaScript deployed in Netscape Navigator 2.0 beta 3
Aug. 1996 JScript deployed in Internet Explorer 3.0

code is supposed to run identically in every browser

⇒ strong need for standardization

Nov. 1996 JavaScript submitted to Ecma International
June 1997 first edition of ECMA-262 (110 pages)
A quick history of JavaScript and ECMAScript

- ES1
- ES2
- ES3
- ES4
- ES5
- ES2015
- ES2016
- ES2017
- ES2018
The specification

- new version every year
- 6 meetings of TC39 each year
- transparent process, on github
The specification

Stage 3

<table>
<thead>
<tr>
<th>Proposal</th>
<th>Author</th>
<th>Champion</th>
<th>Tests</th>
<th>Last Presented</th>
</tr>
</thead>
<tbody>
<tr>
<td>globalThis</td>
<td>Jordan Harband</td>
<td>Jordan Harband</td>
<td>✓</td>
<td>November 2018</td>
</tr>
<tr>
<td>import()</td>
<td>Domenic Denicola</td>
<td>Domenic Denicola</td>
<td>✓</td>
<td>November 2016</td>
</tr>
<tr>
<td>Legacy RegExp features in JavaScript</td>
<td>Claude Pache</td>
<td>Mark Miller, Claude Pache</td>
<td>✓</td>
<td>May 2017</td>
</tr>
<tr>
<td>BigInt</td>
<td>Daniel Ehrenberg</td>
<td>Daniel Ehrenberg</td>
<td>✓</td>
<td>May 2018</td>
</tr>
<tr>
<td>import.meta</td>
<td>Domenic Denicola</td>
<td>Domenic Denicola</td>
<td>✓</td>
<td>September 2017</td>
</tr>
<tr>
<td>Private instance methods and accessors</td>
<td>Daniel Ehrenberg</td>
<td>Daniel Ehrenberg, Kevin Gibbons</td>
<td>?</td>
<td>January 2019</td>
</tr>
</tbody>
</table>
The specification

- new version every year
- 6 meetings of TC39 each year
- transparent process, on github
- don’t break the web
Prototype Access and Mutation

```javascript
function f() {}
f.prototype = { y : 2 }

var x1 = new f()
var x2 = new f()

console.log("Before: x1.y = " + x1.y + "; x2.y = " + x2.y)

x1.__proto__.y = 3

console.log("After: x1.y = " + x1.y + "; x2.y = " + x2.y)

⇒ Before: x1.y = 2; x2.y = 2
⇒ After: x1.y = 3; x2.y = 3
```
Prototype Access and Mutation

```javascript
function f() {}
f.prototype = { y : 2 }

var x1 = new f()
var x2 = new f()

console.log("Before: x1.y = " + x1.y + "; x2.y = " + x2.y)

x1.__proto__.y = 3

console.log("After: x1.y = " + x1.y + "; x2.y = " + x2.y)
```

19.1.2.12 Object.getPrototypeOf ( O )

When the `getPrototypeOf` function is called with argument `O`, the following steps are taken:

1. Let `obj` be ? `ToObject(O)`.
2. Return `obj.[[GetPrototypeOf]]()`.
Test262: ECMAScript Test Suite (ECMA TR/104)

Test262 is the implementation conformance test suite for the latest drafts (or most recent published edition) of the following Ecma specifications:

- ECMA-262, ECMAScript Language Specification
- ECMA-402, ECMAScript Internationalization API Specification
- ECMA-404, The JSON Data Interchange Format (pdf)

Test262 itself is described in ECMA TR/104 and is included in ECMA-414 (pdf).

Goals & State of Test262

The goal of Test262 is to provide test material that covers every observable behavior specified in the ECMA-414 Standards Suite. Development of Test262 is an on-going process. As of October 2017, Test262 consisted of over 29272 individual test files covering the majority of the pseudo-code algorithms and grammar productions defined in the ECMA-414 Standards Suite. Each of these files contains one or more distinct test cases. This marks the most comprehensive ECMAScript test suite to date. While test coverage is broad, TC39 does not consider coverage to be complete and as with any software project there exists the possibility of omissions and errors. This project welcomes any contributions to Test262 that help make test coverage of existing features more comprehensive.
Memorandum of Understanding Between W3C and WHATWG

This MOU describes a collaboration process for the development of the HTML and DOM specifications published (in the past or future) by both W3C and WHATWG, where such specifications include specifications that are in the WHATWG versions of HTML and DOM but have been published separately at W3C. This MOU also sets forth certain publication mechanisms for the Parties around specifications published by W3C or WHATWG, and a transition plan for the W3C around the listed specifications. The Parties may expand the scope of the collaboration process set forth in this MOU beyond the HTML and DOM specifications only by a subsequent MOU that sets forth such expanded scope.
JavaScript, the Formalization
Certified Analyses

Program \rightarrow\text{Semantics}\rightarrow\text{Execution}
Certified Analyses

Semantics

Program

Execution

Property

States

Unsafe States
Certified Analyses

Diagram:
- Program
- Execution
- States
- Unsafe States
- Semantics
- Approximations
- Property
Certified Analyses

- Program
- Abstract Semantics
- Abstract Execution
- Execution
- States
- Unsafe States
- Semantics
- Property
Certified Analyses

- Program
- Execution
- States
- Semantics
- Property
- Abstract Execution
- Unsafe States

Analysis
Two JavaScript semantics in Coq

- **descriptive** given a program and a result, say if they are related
- **executable** given a program, compute the result

**Correctness**

If program P **executes** to v, then P and v are **related**

- 2 years, 8 people
- 18 klocs of Coq
Did we Formalize JavaScript?

Stay close to the specification text

Test It
Did we Formalize JavaScript?

Stay close to the specification text

Test It
JavaScript Formalizations
Semantics of While

\[\text{while } (\text{Expression}) \text{ Statement}\]

1. Let \( V = \text{empty} \).
2. Repeat
   1. Let \( \text{exprRef} \) be the result of evaluating \( \text{Expression} \).
   2. If \( \text{ToBoolean(GetValue(exprRef))} \) is false, return \((\text{normal}, V, \text{empty})\).
   3. Let \( \text{stmt} \) be the result of evaluating \( \text{Statement} \).
   4. If \( \text{stmt.value} \) is not empty, let \( V = \text{stmt.value} \).
   5. If \( \text{stmt.type} \) is not continue or \( \text{stmt.target} \) is not in the current label set, then
      1. If \( \text{stmt.type} \) is break and \( \text{stmt.target} \) is in the current label set, then Return \((\text{normal}, V, \text{empty})\).
      2. If \( \text{stmt} \) is an abrupt completion, return \( \text{stmt} \).
Let $V = \text{empty}$.

\begin{verbatim}
(* Step 1 *)
| red_stat_while : forall S C labs e1 t2 o,
  red_stat S C (stat_while_1 labs e1 t2 resvalue_empty) o ->
  red_stat S C (stat_while labs e1 t2) o
\end{verbatim}
Semantics of While

Repeat

1. Let \( exprRef \) be the result of evaluating Expression.
2. If \( ToBoolean(GetValue(exprRef)) \) is false, return \((normal, V, empty)\).

(* Steps 2a and 2b *)

| red_stat_while_1 : forall S C labs e1 t2 rv y1 o,
  red_spec S C (spec_expr_get_value_conv spec_to_boolean e1) y1 ->
  red_stat S C (stat_while_2 labs e1 t2 rv y1) o ->
  red_stat S C (stat_while_1 labs e1 t2 rv) o

(* Step 2b False *)

| red_stat_while_2_false : forall S0 S C labs e1 t2 rv,
  red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S false)) (out_ter S rv)
12.6.2 The while Statement

The production 

\[ \text{IterationStatement : while ( Expression ) Statement} \]

is evaluated as follows:

1. Let \( V \) be empty.
2. Repeat
   a. Let \( exprRef \) be the result of evaluating \( Expression \).
   b. If \( \text{ToBoolean}(\text{GetValue}(exprRef)) \) is \( \text{false} \), return \( (\text{normal}, V, \text{empty}) \).
   c. Let \( stmt \) be the result of evaluating \( Statement \).
   d. If \( stmt\text{.value} \) is not empty, let \( V = stmt\text{.value} \).
   e. If \( stmt\text{.type} \) is not \( \text{continue} \) and \( stmt\text{.target} \) is not in the current label set, then
      i. If \( stmt\text{.type} \) is \( \text{break} \) and \( stmt\text{.target} \) is in the current label set, then
         1. Return \( (\text{normal}, V, \text{empty}) \).
         2. If \( stmt \) is an abrupt completion, return \( stmt \).
JavaScript Formalizations

JSCERT

CoQ

JSRef

ECMAScript Language test262

Standard
Definition run_stat_while runs S C rv labs e1 t2 : result :=
  if_spec (run_expr_get_value runs S C e1) (fun S1 v1 =>
    Let b := convert_value_to_boolean v1 in
    if b then
      if_ter (runs_type_stat runs S1 C t2) (fun S2 R =>
        Let rv' := ifb res_value R <> resvalue_empty then res_value R else rv in
        Let loop := fun _ => runs_type_stat_while runs S2 C rv' labs e1 t2 in
        ifb res_type R <> restype_continue \~ res_label_in R labs then (
          ifb res_type R = restype_break \~ res_label_in R labs then
            res_ter S2 rv'
          else (  
            ifb res_type R <> restype_normal then
              res_ter S2 R
            else loop tt
          )
        ) else loop tt)
    else res_ter S1 rv).
let run_stat_while runs0 s c rv labs e1 t2 =
  if_spec (run_expr_get_value runs0 s c e1) (fun s1 v1 ->
    let_binding (convert_value_to_boolean v1) (fun b ->
      if b
        then if_ter (runs0.runs_type_stat s1 c t2) (fun s2 r ->
          let_binding
            (if not_decidable
              (resvalue_comparable r.res_value Coq_resvalue_empty)
              then r.res_value
              else rv) (fun rv' ->
                let_binding (fun x ->
                  runs0.runs_type_stat_while s2 c rv' labs e1 t2) (fun loop ->
                    if or_decidable
                      (not_decidable (restype_comparable r.res_type Coq_restype_continue))
                      (not_decidable (bool_decidable (res_label_in r labs)))).
                      then if and_decidable
                        (restype_comparable r.res_type Coq_restype_break)
                        (bool_decidable (res_label_in r labs)).
                        then res_ter s2 (res_normal rv')
                        else if not_decidable (restype_comparable r.res_type Coq_restype_normal)
                          then res_ter s2 r
                          else loop ()
                        else loop ()
                      else loop ()
                    else res_ter s1 (res_normal rv)))
            else loop ()
          else res_ter s1 (res_normal rv))
      else res_ter s1 (res_normal rv)))
Fairly Close to the Inductive Rules
JavaScript Formalizations

JSCERT

Coq

OCaml Extraction

Parser

JSREF

Correctness
JavaScript Formalizations

JSCERT → Correctness → JSREF

OCaml Extraction → Parser

CoQ → ¬ CoQ
JavaScript Formalizations

Correctness

JSCERT

CoQ

OCaml Extraction

Parser

JSRef

Bisect
Test Suite Coverage with Bisect

- good coverage of the core of ECMAScript 5.1
- code extraction from JSRef
  1. instrumented to report coverage
  2. run the test suite
  3. find places not executed (not tested)
  4. relate to parts of the spec not tested
  5. discover discrepancies between implementations
try ... finally in Chrome

```javascript
while(true) {
    try {
        "try";
        break
    } finally {
        "finally"
    }
}  
returns finally

while(true) {
    try {
        "try";
        break
    } finally {
        "finally"
    }
}  
    y = "done"
returns try
```
Impact on the Specification

More precise definitions

1. Let lprim be ? ToPrimitive(lval).
2. Let rprim be ? ToPrimitive(rval).
3. If Type(lprim) is String or Type(rprim) is String, then
   a. Let lstr be ? ToString(lprim).
   b. Let rstr be ? ToString(rprim).
   c. Return the string-concatenation of lstr and rstr.

Towards a typed specification

- Numbers vs mathematical integers
- Return values
An OCaml interpreter of JavaScript

- very close to the specification
- based on the extraction from JSRef
- uses a tiny subset of OCaml in monadic style
  - functions, tuples, shallow pattern matching, records
- request by Shu-yu Guo (Dagstuhl, 2014): a step by step execution of the spec
1. Let lprim be ? ToPrimitive(lval).
2. Let rprim be ? ToPrimitive(rval).
3. If Type(lprim) is String or Type(rprim) is String, then
   a. Let lstr be ? ToString(lprim).
   b. Let rstr be ? ToString(rprim).
   c. Return the string-concatenation of lstr and rstr.
4. Let lnum be ? ToNumber(lprim).
5. Let rnum be ? ToNumber(rprim).
6. Return the result of applying the addition operation to lnum and rnum.
and run_binary_op_add s0 c v1 v2 =
  let%prim (s1, w1) = to_primitive_def s0 c v1 in
  let%prim (s2, w2) = to_primitive_def s1 c v2 in
  if (type_compare (type_of (Value_prim w1)) Type_string) || (type_compare (type_of (Value_prim w2)) Type_string)
  then
    let%string (s3, str1) = to_string s2 c (Value_prim w1) in
    let%string (s4, str2) = to_string s3 c (Value_prim w2) in
    res_out (Out_ter (s4, (Res_val (Value_prim (Prim_string (strappend str1 str2))))))
  else
    let%number (s3, n1) = to_number s2 c (Value_prim w1) in
    let%number (s4, n2) = to_number s3 c (Value_prim w2) in
    res_out (Out_ter (s4, (Res_val (Value_prim (Prim_number (n1 +. n2))))))
• motivation: run it in a browser
• uses compiler-libs to generate a typed AST, which we translate
• target is a tiny subset of JS
  • functions, objects (no prototype), arrays, string, numbers
  • no type conversion

```javascript
var run_binary_op_add = function (s0, c, v1, v2) {
  return (if_prim(to_primitive_def(s0, c, v1), function(s1, w1) {
    return (if_prim(to_primitive_def(s1, c, v2), function(s2, w2) {
      if ((type_compare(type_of(Coq_value_prim(w1)), Coq_type_string()))
      || type_compare(type_of(Coq_value_prim(w2)), Coq_type_string()))
      { return (if_string(to_string(s2, c, Coq_value_prim(w1)), function(s3, str1) {
        return (if_string(to_string(s3, c, Coq_value_prim(w2)), function(s4, str2) {
          return (res_out(Coq_out_ter(s4, res_val(Coq_value_prim(Coq_prim_string(strappend(str1, str2))))))); }});});}
    } else { ... }})); });
};
```
and to Pseudo JavaScript

- to be readable while staying close to JavaScript
  - hide state and context
  - monadic extension of var
  - pattern matching
  - hide type changes

```javascript
var run_expr_binary_op =
    function (op, e1, e2) {
    switch (op) {
        case Coq_binary_op_and:
            return (run_binary_op_and(e1, e2));
        case Coq_binary_op_or:
            return (run_binary_op_or(e1, e2));
        default:
            var %run v1 = run_expr_get_value(e1);
            var %run v2 = run_expr_get_value(e2);
            return (run_binary_op(op, v1, v2));
    }
};

var run_binary_op_add = function (v1, v2) {
    var %prim w1 = toPrimitiveDef v1;
    var %prim w2 = toPrimitiveDef v2;
    if ((typeCMP(typeOf(w1), Type_string) || typeCMP(typeOf(w2), Type_string))) {
        var %string str1 = toString w1;
        var %string str2 = toString w2;
        return (strApp(str1, str2));
    } else {
        var %number n1 = toNumber w1;
        var %number n2 = toNumber w2;
        return (n1 + n2);
    }
};
```
instrument the generated JavaScript to record events
- Enter (enter a function)
- CreateCtx(ctx) (new function scope)
- Add(ident,value) (let binding)
- Return (return from a function)

executing the instrumented interpreter generates a trace of events

web tool to navigate these traces
Architecture

- Interpreter and libraries (OCaml)
- Libraries (JS)
- Interpreter with traces (JS)
- AST of interpreted program
- Esprima
- interpreted program
- web page
- trace
- generator
- tracing generator
9.1.7.1 OrdinaryHasProperty (O, P)

When the abstract operation OrdinaryHasProperty is called with Object O and with property key P, the following steps are taken:

1. Assert IsPropertyKey(P) is true.
2. Let hasOwn be ␥. ␥. [[GetOwnProperty]](P).
3. If hasOwn is not undefined, return true.
4. Let parent be ␥. ␥. [[GetPrototypeOf]](O).
5. If parent is not null, then
6. Return false.

9.1.8 [[Get]] (P, Receiver)

When the [[Get]] internal method of O is called with property key P and ECMAScript language value Receiver, the following steps are taken:


9.1.8.1 OrdinaryGet (O, P, Receiver)
Skeletal Semantics
The Problem with JSCert

(** If statement (12.5) *)

| red_stat_if : forall S C e1 t2 t3opt y1 o,
| red_spec S C (spec_expr_get_value_conv spec_to_boolean e1) y1 ->
| red_stat S C (stat_if_1 y1 t2 t3opt) o ->
| red_stat S C (stat_if e1 t2 t3opt) o

| red_stat_if_1_true : forall S0 S C t2 t3opt o,
| red_stat S C t2 o ->
| red_stat S0 C (stat_if_1 (vret S true) t2 t3opt) o

| red_stat_if_1_false : forall S0 S C t2 t3 o,
| red_stat S C t3 o ->
| red_stat S0 C (stat_if_1 (vret S false) t2 (Some t3)) o

| red_stat_if_1_false_implicit : forall S0 S C t2,
| red_stat S0 C (stat_if_1 (vret S false) t2 None) (out_ter S resvalue_empty)

- 900 mutually inductive rules
- inversion during an induction runs out of memory
Ingredients of a Semantics

Evaluate \textit{if } e \textit{ then } s1 \textit{ else } s2 \textit{ in state } \sigma

1. Let \( v \) the result of evaluating \( e \) in state \( \sigma \).
2. If \( v \) is true, let \( o \) the result of evaluating \( s1 \) in state \( \sigma \).
3. If \( v \) is false, let \( o \) the result of evaluating \( s2 \) in state \( \sigma \).
4. Return \( o \).

\[
\frac{\sigma, e \downarrow v \quad v = \text{tt}}{\sigma, \text{if } e \text{ then } s1 \text{ else } s2 \downarrow o} \quad \frac{\sigma, e \downarrow v \quad v = \text{ff}}{\sigma, \text{if } e \text{ then } s1 \text{ else } s2 \downarrow o}
\]
Ingredients of a Semantics

**Sequence**

\[
\begin{array}{c}
\sigma, e \downarrow v \\
v = \texttt{tt} \\
\sigma, s_1 \downarrow o
\end{array} \quad \quad \quad \quad \quad \quad \quad \quad
\begin{array}{c}
\sigma, e \downarrow v \\
v = \texttt{ff} \\
\sigma, s_2 \downarrow o
\end{array}
\]

**Evaluate if e then s1 else s2 in state σ**

1. Let \( v \) the result of evaluating \( e \) in state \( σ \).
2. If \( v \) is true, let \( o \) the result of evaluating \( s_1 \) in state \( σ \).
3. If \( v \) is false, let \( o \) the result of evaluating \( s_2 \) in state \( σ \).
4. Return \( o \).
Ingredients of a Semantics

Evaluate if e then s1 else s2 in state $\sigma$

1. Let $v$ the result of evaluating e in state $\sigma$.
2. If $v$ is true, let $o$ the result of evaluating $s1$ in state $\sigma$.
3. If $v$ is false, let $o$ the result of evaluating $s2$ in state $\sigma$.
4. Return $o$.
Ingredients of a Semantics

Evaluate if e then s1 else s2 in state $\sigma$

1. Let $v$ the result of evaluating $e$ in state $\sigma$.
2. If $v$ is true, let $o$ the result of evaluating $s_1$ in state $\sigma$.
3. If $v$ is false, let $o$ the result of evaluating $s_2$ in state $\sigma$.
4. Return $o$.
Ingredients of a Semantics

**Atom**

\[
\begin{align*}
\sigma, e \Downarrow v & \quad v = \text{tt} \quad \sigma, s_1 \Downarrow o \\
\sigma, \text{if } e \text{ then } s_1 \text{ else } s_2 \Downarrow o
\end{align*}
\]

\[
\begin{align*}
\sigma, e \Downarrow v & \quad v = \text{ff} \quad \sigma, s_2 \Downarrow o \\
\sigma, \text{if } e \text{ then } s_1 \text{ else } s_2 \Downarrow o
\end{align*}
\]

Evaluate if e then s1 else s2 in state \(\sigma\)

1. Let \(v\) the result of evaluating \(e\) in state \(\sigma\).
2. If \(v\) is true, let \(o\) the result of evaluating \(s_1\) in state \(\sigma\).
3. If \(v\) is false, let \(o\) the result of evaluating \(s_2\) in state \(\sigma\).
4. Return \(o\).
Ingredients of a Semantics

- Structure: sequence, recursion, choice
- Atoms

\[
\begin{align*}
\sigma, e \downarrow v & \quad v = \text{tt} & \sigma, s_1 \downarrow o \\
\sigma, \text{if } e \text{ then } s_1 \text{ else } s_2 \downarrow o
\end{align*}
\]

\[
\begin{align*}
\sigma, e \downarrow v & \quad v = \text{ff} & \sigma, s_2 \downarrow o \\
\sigma, \text{if } e \text{ then } s_1 \text{ else } s_2 \downarrow o
\end{align*}
\]

Evaluate if e then s1 else s2 in state \( \sigma \)

1. Let \( v \) the result of evaluating \( e \) in state \( \sigma \).
2. If \( v \) is true, let \( o \) the result of evaluating \( s_1 \) in state \( \sigma \).
3. If \( v \) is false, let \( o \) the result of evaluating \( s_2 \) in state \( \sigma \).
4. Return \( o \).
Size of Semantics (Number of Rules)

- $\lambda$-calculus: 3 rules
- CoreML: $\approx$ 50 rules
- C: $\approx$ 200 rules
- JavaScript: $\approx$ 900 rules
- R: $\approx$ 2,000 rules
Size of Semantics (Number of Atoms)

<table>
<thead>
<tr>
<th>Language</th>
<th>Number of Atoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>λ-calcul</td>
<td>1 atom</td>
</tr>
<tr>
<td>CoreML</td>
<td>~ 30 atoms</td>
</tr>
<tr>
<td>C</td>
<td>~ 120 atoms</td>
</tr>
<tr>
<td>JavaScript</td>
<td>~ 80 atoms</td>
</tr>
<tr>
<td>R</td>
<td>~ 160 atoms</td>
</tr>
</tbody>
</table>
Skeletons

Concrete Interpretation

Abstract Interpretation

While

λ-calcul

Consistency

generic

generic
Skeletal Semantics

- Simple framework capturing the structure of semantics
- Generic definition of interpretations
- Proof techniques to relate interpretations
- Generation of an OCaml interpreter
- Generation of an analyzer
- Needs to be applied to JavaScript
- Technical talk this afternoon
What’s Next
Beyond JSExplain and Skeletal Semantics

- Application to other languages (MLExplain)
- Generation of *-explain from Skeletal Semantics
- Analyses requested by TC39
  - Invariants
  - Object Capabilities
Formalizing semantics can be fun (and fruitful)!

JavaScript is an ideal candidate: complex and precise
Questions?

- https://tc39.github.io/
- http://www.jscert.org/
- https://gitlab.inria.fr/star-explain/
- http://skeletons.inria.fr/