A Structural Operational Semantics for JavaScript

Ankur Taly

Dept. of Computer Science, Stanford University
1. Need for a Formal Semantics?

2. Structural Operational Semantics for IMP
   - Formalizing the program state
   - Semantic Rules
   - Formal properties

3. Structural Operational Semantics for JavaScript
   - Motivation
   - Main features of JavaScript
   - Formalizing the Program state
   - Semantic Rules
   - Formal Properties

4. Applications of the Operational Semantics for JavaScript
   - BeamAuth
   - ADSafe

5. Conclusions and Future work
Syntax and Semantics

Every programming language has two broad components

- **Syntax**
  - All possible strings in the language.
  - Structure of the strings usually given by a Formal grammar.
  - Example: if \((x > 10)\) \(\{ \ x = x + 1; \}\)

- **Semantics**
  - Meaning of the strings in the language.
  - Actions that occur while interpreting these strings.
  - Meaning \([\text{if } (x > 10) \{ x = x + 1; \}]\) = If value of \(x\) is greater than 10 then \(x\) is incremented by 1.
Interpreter

- Convert Syntax to Machine Code (in accordance with the semantics).

\[
\text{Syntax} \xrightarrow{\text{Semantics}} \text{Machine Code}
\]

- Generate machine code so that the correct sequence of actions occur

**Example**: \(x = x + 10\)

  - Find where the variable \(x\) is stored in memory.
  - Add 10 to the value.
  - Place the new value back in memory.

We are not interested in this conversion process but only in specifying the semantics using some mathematical object.
Informal and Formal Semantics

Informal semantics:

- Meaning in English.
- Language standard or specification manual (JavaScript: ECMA 2.62)
- Sufficient for 'understanding' the language but insufficient for rigorously proving properties about the language
- Prove or Disprove: For all terms $t$, the execution of $t$ only depends on the values of the variables appearing in $t$.
  - Does meaning $[x = x + 10]$ only depend on value of $x$? in C? in JavaScript?
A corner case

```javascript
var y = "a";
var x = {toString : function(){ return y;}}
x = x + 10;
jasper> "a10"
```

- **Implicit type conversion** of an object to a string in JavaScript involves calling the `toString` function.
- Informal semantics fail to bring out such corner cases.
Informal and Formal Semantics

Formal Semantics:

- Specify meaning in a **Mathematically rigorous** way.
- Provides a framework for proving properties of the kind mentioned on the previous slide.
- **Task**: Convert informal semantics into a formal semantics.
- The very act of formalization reveals **subtle aspects** of the language.
Real World Example

Isolating two pieces of JavaScript code:

- Web pages today serve content coming from multiple locations:
  Example: Advertisement code obtained from various ad networks.
- Some of this content may be untrusted: Ad code might contain a malicious script

```javascript
var v = document.getElementsByTagName("password")[0].value
```

- Allow untrusted code to perform valuable interactions while at the same preventing intrusion and malicious damage.
Real World Example

Figure: Trusted and Untrusted code
Real World Example

**Figure:** Web Mashup (Valuable Interaction)
Formulating the problem

Problem 1
Write a static analyzer that can check an untrusted JavaScript program and determine if it is malicious.

Very difficult to statically determine the meaning of a JavaScript program because of the presence of functions that can convert string to code and vice versa, for eg: eval.

Problem 2
Find a well-defined and meaningful subset of JavaScript for which the above problem is solvable.

Seems approachable: Identify the bad guys like 'eval' and get rid of them.
Formal Semantics of JavaScript is the first step towards solving the above problems.
Formulating the problem

**Problem 1**
Write a static analyzer that can check an untrusted JavaScript program and determine if it is malicious.

Very difficult to statically determine the meaning of a JavaScript program because of the presence of functions that can convert string to code and vice versa, for eg: `eval`.

**Problem 2**
Find a well-defined and meaningful subset of JavaScript for which the above problem is solvable.

Seems approachable: Identify the bad guys like 'eval' and get rid of them.
Formal Semantics of JavaScript is the first step towards solving the above problems.
Applications of Formal Semantics

- Reveal inconsistencies in the language standard/spec.
- Prove properties about specific programs, static analysis of programs.
- Prove properties involving quantification over all possible programs.
- Proving program equivalence.
- Design sound compiler optimizations.
Operational Semantics

- Meaning of a program $\Leftrightarrow$ sequence of actions that are taken during its execution.
- Specify sequence of actions as transitions of an Abstract State machine
- States corresponds to
  - Term being evaluated
  - Abstract description of memory and other data structures involved in computation.
- A state transition denotes a partial evaluation of the term.
- Based on the abstract syntax of the language.
Another approach

**Denotational Semantics**:  
- Specify meaning of a program by specifying the partial function represented by each syntactic construct.  
- The function could be from one memory state to another.

**Example**: The denotation of the term

```javascript
y = 1; x = 10; while(x >= 1){y = x * y; x = x - 1};
```

is a partial function from one abstract memory state to another. The final state would be the initial state except that the value of \(x\) would be 0 and the value of \(y\) would be factorial(10).
Structural Operational Semantics (SOS).

- Systematic way of specifying operational semantics.
- Specify the transitions in a syntax oriented manner using the inductive nature of the abstract syntax.

**Example**: The state transition for the term $e_1 + e_2$ should be described using the transition for $e_1$ and the transition for $e_2$.

- We will first look at the SOS of a simple imperative language (IMP) and then extend it to JavaScript.
The IMP Language

Syntax

Aexp : a ::= n | X | a + a | a * a
Bexp : b ::= t | a <= a | not b | b and b
Stmt : s ::= skip | X := a | s; s |

- X is the set of variables (fixed) and n varies over \( \mathbb{N} \).
- Informal Semantics is very intuitive
Formal Semantics of IMP : Notion of a State

- Represented as a pair $S = \langle t, \sigma \rangle$
  - $t$ : term being evaluated (expressed using abstract syntax)
  - $\sigma$ : abstract description of memory.

- For IMP, we can abstract the memory as a store, which is a mapping from variable names to values ie $\sigma : X \rightarrow \mathbb{I}$

- Terminal states are those for which the term $t$ is empty or is a value.

- We will add more information to the state when we move to JavaScript.
Transition Rule : Example

A rule for Arithmetic Expressions

\[ \langle a_1, \sigma \rangle \rightarrow \langle a_1', \sigma \rangle \]
\[ \langle a_1 + a_2, \sigma \rangle \rightarrow \langle a_1' + a_2, \sigma \rangle \] \[ A_{3a} \]

\[ \langle a_2, \sigma \rangle \rightarrow \langle a_2', \sigma \rangle \]
\[ \langle n + a_2, \sigma \rangle \rightarrow \langle n + a_2', \sigma \rangle \] \[ A_{3b} \]

How to interpret this rule?

- If the term \( a_1 \) partially evaluates to \( a_1' \) then \( a_1 + a_2 \) partially evaluates to \( a_1' + a_2 \).
- Once the expression \( a_1 \) reduces to a value \( n \), then start evaluating \( a_2 \).

Example:
\[ \langle (10 + 12) + (13 + 20), \sigma \rangle \xrightarrow{A_{3a}} \langle 22 + (13 + 20), \sigma \rangle \xrightarrow{A_{3b}} \langle 22 + 33, \sigma \rangle \]
Transition Rule: Example

A rule for Statements

\[
\begin{align*}
\langle a, \sigma \rangle & \rightarrow \langle a', \sigma' \rangle & [C_3] \\
\langle x := a, \sigma \rangle & \rightarrow \langle x = a', \sigma' \rangle & [C_3] \\
\langle x := n, \sigma \rangle & \rightarrow \langle \sigma' \rangle & [C_2]
\end{align*}
\]

How to interpret this rule?

- If the arithmetic exp \( a \) partially evaluates to \( a' \) then the statement \( x = a \) partially evaluates to \( x = a' \).
- Rule \( C_2 \) applies when \( a \) reduces to a value \( n \).
- \( \text{Put}(\sigma, x, n) \) updates the value of \( x \) to \( n \).

Example: \( \langle (x := 10 + 12, \sigma) \xrightarrow{C_3} \langle x := 22, \sigma \rangle \xrightarrow{C_2} \langle \sigma' \rangle \)
Formally

General form of transition rule:

\[
\begin{align*}
&P_1, \ldots, P_n \\
\langle t, \sigma \rangle &\rightarrow \langle t', \sigma' \rangle \\
&P_1, \ldots, P_n \\
\langle t, \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]  

\(P_1, \ldots, P_n\) are the **conditions** that must hold for the transition to go through. Also called the **premise** for the rule. These could be:

- Other transitions corresponding to the sub-terms.
- Predicates that must be true.
- Calls to meta functions like:
  - \(\text{get}(\sigma, x) = v\) : Fetch the value of \(x\).
  - \(\text{put}(\sigma, x, n) = \sigma'\) : Update value of \(x\) to \(n\) and return new store.
Arithmetic Expressions

\[ A_{\text{exp}} : a ::= n \mid X \mid a + a \mid a \ast a \]

**Rules**

\[
\langle n, \sigma \rangle : \text{Terminal}[A_1] \\
\langle a_1, \sigma \rangle \rightarrow \langle a'_1, \sigma \rangle [A_4a] \\
\langle a_1 \ast a_2, \sigma \rangle \rightarrow \langle a'_1 \ast a_2, \sigma \rangle [A_4a] \\
\langle a_2, \sigma \rangle \rightarrow \langle a'_2, \sigma \rangle [A_4b] \\
\langle n \ast a_2, \sigma \rangle \rightarrow \langle n \ast a'_2, \sigma \rangle [A_4b]
\]

**Example**

Consider the store \( \sigma = \{ x : 10, y : 20 \} \). The state \( \langle x + y, \sigma \rangle \) has the following trace.

\[
\langle x \ast y, \sigma \rangle \xrightarrow{A_2} \langle 10 \ast y, \sigma \rangle \xrightarrow{A_2} \langle 10 \ast 20, \sigma \rangle
\]
Commands

Com : c ::= skip | X := a | c; c |
if b then c else c | while b do c

rules

\[
\begin{align*}
\langle a, \sigma \rangle & \rightarrow \langle a', \sigma' \rangle &[C_3] \\
\langle x := a, \sigma \rangle & \rightarrow \langle x = a', \sigma' \rangle &[C_2] \\
\langle s_1, \sigma \rangle & \rightarrow \langle s'_1, \sigma' \rangle &[C_{4a}] \\
\langle s_1; s_2, \sigma \rangle & \rightarrow \langle s'_1; s_2, \sigma' \rangle &[C_{4b}]
\end{align*}
\]
Commands

**If Then Else**

\[
\begin{align*}
\langle \text{if } \texttt{tt} \text{ then } s_1 \text{ else } s_2, \sigma \rangle & \rightarrow \langle s_1, \sigma \rangle \quad [C_{5a}] \\
\langle \text{if } \texttt{ff} \text{ then } s_1 \text{ else } s_2, \sigma \rangle & \rightarrow \langle s_2, \sigma \rangle \quad [C_{5b}] \\
\langle b, \sigma \rangle & \rightarrow \langle b', \sigma \rangle \\
\langle \text{if } b \text{ then } s_1 \text{ else } s_2, \sigma \rangle & \rightarrow \langle \text{if } b' \text{ then } s_1 \text{ else } s_2, \sigma \rangle \quad [C_{5c}] 
\end{align*}
\]

**While**

\[
\begin{align*}
\langle \text{while } b \text{ do } s, \sigma \rangle & \rightarrow \\
\langle \text{if } b \text{ then } s; \text{ while } b \text{ s else skip end}, \sigma \rangle & \rightarrow \langle \text{if } b' \text{ then } s_1 \text{ else } s_2, \sigma \rangle \quad [C_{6}] 
\end{align*}
\]
## Context Sensitive Rules (Felleisen)

### Similar rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A_3a]</td>
<td>[a_1, \sigma ] \rightarrow [a'_1, \sigma ]</td>
</tr>
<tr>
<td>[A_3b]</td>
<td>[a_2, \sigma ] \rightarrow [a'_2, \sigma ]</td>
</tr>
<tr>
<td>[A_4a]</td>
<td>[a_1 + a_2, \sigma ] \rightarrow [a'_1 + a_2, \sigma ]</td>
</tr>
<tr>
<td>[A_4b]</td>
<td>[n + a_2, \sigma ] \rightarrow [n + a'_2, \sigma ]</td>
</tr>
<tr>
<td>[A_5]</td>
<td>[a_1 \ast a_2, \sigma ] \rightarrow [a'_1 \ast a_2, \sigma ]</td>
</tr>
<tr>
<td>[A_6]</td>
<td>[n \ast a_2, \sigma ] \rightarrow [n \ast a'_2, \sigma ]</td>
</tr>
</tbody>
</table>

- The above rules have a similar premise:
- Combine them into a **single** rule of the following form:

\[
\begin{align*}
\langle a, \sigma \rangle &\rightarrow \langle a', \sigma \rangle \\
AC(a) &\rightarrow AC(a')
\end{align*}
\]

where \(AC:: -| - + an + -| - \ast an \ast -\)
Context Rules

- Reduces the number of rules.
- Replace all rules which have a transition in their premise by an appropriate context rule. Very helpful while proving properties by induction.
- Contexts can also be used to specify continuations (more on this when we move to JavaScript).
Progress

For all $a \in A_{exp}$, states $\sigma$:

$$a \text{ is not a Value } \Rightarrow \exists a', \sigma' : \langle a, \sigma \rangle \rightarrow \langle a', \sigma' \rangle$$

Proof Technique: Induction over the structure of $a$.

- Base Case: $a := n$ and $a := x$
- Inductive Cases: $a := a_1 + a_2$, $a := a_1 \ast a_2$.
  - Assume Induction hypothesis for $a_1$ and $a_2$
  - Show that a reduction rule applies to $a$ if there exists a reduction for $a_1$ and $a_2$. 
Let \( \text{Var}(t) \) denotes set of variable names appearing in term \( t \).
For Ex : \( \text{Var}(\text{if}(x \geq 0)\{y = y + 1\}) = \{x, y\} \).

**Reachability**

For all \( t, t' \in \text{Terms} \), states \( \sigma, \sigma' \) :

\[
\forall x \not\in \text{Var}(t) : \langle t, \sigma \rangle \rightarrow \langle t', \sigma' \rangle \Rightarrow \sigma(x) = \sigma'(x)
\]

Proof Technique : This is a property of a reduction step. **Induct over the set of transition rules.**

- Base Case : Rules with no transition in their premise.
- Inductive Case : Context rules.

Later, we will state a similar property for JavaScript.
1. Need for a Formal Semantics?

2. Structural Operational Semantics for IMP
   - Formalizing the program state
   - Semantic Rules
   - Formal properties

3. Structural Operational Semantics for JavaScript
   - Motivation
   - Main features of JavaScript
   - Formalizing the Program state
   - Semantic Rules
   - Formal Properties

4. Applications of the Operational Semantics for JavaScript
   - BeamAuth
   - ADSafe

5. Conclusions and Future work
Semantics of JavaScript: Motivation

- Widely used web programming language.
- Several subtle features that most programmers are unaware of.
- Very important to fully understand the semantics so as to reason about the security properties of programs written in it.

Example 1

```javascript
var b = 10;
var f = function(){
    var b = 5;
    function g(){
        var b = 8; return this.b;
    }
    g();
}
var result = f();
```
Semantics of JavaScript: Motivation

- Widely used web programming language.
- Several subtle features that most programmers are unaware of.
- Very important to fully understand the semantics so as to reason about the security properties of programs written in it.

Example 1

```javascript
var b = 10;
var f = function(){
    var b = 5;
    function g(){
        var b = 8;
        return this.b;
    }
    g();
}
var result = f();
```

Result = 10
Another Example

Example 2

```javascript
var f = function(){
    var a = g();
    function g() { return 1;};
    function g() { return 2;};
    var g = function() { return 3;}
    return a;
}
var result = f();
```

What is the final value of `result`?
Another Example

Example 2

```
var f = function() { var a = g();
    function g() { return 1;};
    function g() { return 2;};
    var g = function() { return 3;}
    return a;}
var result = f();
```

What is the final value of `result`?
`result = 2`
Informal Semantics

- **ECMA 2.62 Spec.**
- **Important Concepts**
  - Prototype based lookup for object properties
  - Function Closures.
  - Function calls involve parsing the body of the function before calling it.
  - `for (p in o){....} , eval(...), o[s]` allow string to be used as code and vice versa.
Basic JavaScript Syntax

Syntax

According to ECMA 2.62:

Expressions (e) ::= this | x | e OP e | e(e) | new e(e) | ...
Statement (s) ::= "s*" | if (e) s else s | while (e) s | with (e) s | ...
Programs (P) ::= s P | fd P
Function Decl (fd) ::= function x (x){ P }

Observation

Observe that according to the spec, declaring a function inside an 'if' block is a syntax error!
Basic JavaScript Syntax

Syntax

According to ECMA 2.62:

Expressions (e) :: this | x | e OP e | e(e) | new e(e) |...

Statement (s) :: "s*" | if (e) s else s | while (e) s | with (e) s | ...

Programs (P) :: s P | fd P

Function Decl (fd) :: function x (x){ P }

Observation

Observe that according to the spec, declaring a function inside an 'if' block is a syntax error!
JavaScript: Key Features

- Everything (including functions) is either an object or a primitive value.
- Activation records are normal JavaScript objects and the variable declarations are properties of this object.
- All computation happens inside a global object which is also the initial activation object.
- Instead of a stack of activation records, there a chain of activation records, which is called the scope chain.
- Arbitrary objects can be placed over the scope chain.
Formal Semantics: Program state

- All objects are passed by reference ⇒ The store must have information about **Heap locations**.
- Variables have different values in different scopes ⇒ State must include info about **current scope**.

State

Program state is represented as a triple $\langle H, l, t \rangle$.

- $H$ : Denotes the Heap, mapping from the set of locations ($\mathbb{L}$) to objects.
- $l$ : Location of the current scope object (or current activation record).
- $t$ : Term being evaluated.
Heap and Heap Reachability Graph

**Figure:** Heap and its reachability graph

**Heap Reachability Graph:** Heap addresses are the nodes. An edge from $l_i$ to $l_j$, if the object at address $l_i$ has property $p$ pointing to $l_j$.
Heap Operations

Each Heap object $o$ is a record $\{p_1 : o_{v1}, \ldots, p_n : o_{v_n}\}$ where $p_i$ are property names and $o_{v_i}$ could be a primitive value or another heap addresses.

- $Dot(H, l, p) = l_1$ : Access the property $p$ of object at heap location $l$.
- $Put(H, l, p, l_v) = H'$ : Update the property $p$ of object at $H(l)$ and return the new Heap.
- $H', l = alloc(H, o)$ : Allocate object $o$ to a new heap location $l$. 
Identifier Resolution

How do we find the value of an identifier ’x’ appearing in the program?

In the case of IMP we have the rule:

\[
\begin{align*}
\nu &= \text{get}(\sigma, x) \\
\langle x, \sigma \rangle \rightarrow \langle \nu, \sigma \rangle
\end{align*}
\]

For JavaScript it involves traversing the scope chain.

- **Recall**: following of access links to resolve local variables.
- In the case of JavaScript scope chain can also have arbitrary objects in addition to activation records.
- ”x” gets resolved to a pair \(1 \ast "x"\) where \(1\"hasproperty\"x\).
Some Notation

- \( \text{o "hasProperty" } p \iff \text{"p" is a property of object "o" or one of the ancestral prototypes of "o".} \)
- \( \text{o "hasOwnProperty" } p \iff \text{"p" is a property of object "o" itself.} \)
- A JavaScript reference type is pair denoted as \( l \ast p \) where \( l \) is heap address, also called the base type of the reference, and \( p \) is a property name.
Every scope chain has the global object at its base. Every prototype chain has `Object.prototype at the top`, which is a native object containing predefined functions such as `toString`, `hasOwnProperty` etc.
Scope lookup: Rules

ECMA 2.62:

1. Get the next object (l) in the scope chain. If there isn’t one, goto 4.
2. If l "HasProperty" x, return a reference type l*"x".
3. Else, goto 1
4. Return null*x.

Scope(H, l, "x") = ln
\[ \langle H, l, x \rangle \rightarrow \langle H, l, ln \ast \"x\" \rangle \]

HasProperty(H, l, m)
\[ \text{Scope}(H, l, m) = l \]

\neg(\text{HasProperty}(H, l, m))
\[ H(l).\text{@Scope} = ln \]
\[ \text{Scope}(H, l, m) = \text{Scope}(H, ln, m) \]

Scope(H, null, m) = null
Prototype lookup: Rules

ECMA 2.62:

1. If base type is null, throw a ReferenceError exception.
2. Else, Call the Get method, passing propName(x) and base type l as arguments.
3. Return result(2).

\[
\begin{align*}
H_2, l_{\text{excp}} &= \text{alloc}(H, o) \\
o &= \text{newNativeErr}("", \#\text{RefErrProt}) \\
\langle H, l, (\text{null} \ast m) \rangle &\rightarrow \langle H_2, l, \langle l_{\text{excp}} \rangle \rangle
\end{align*}
\]

\[
\begin{align*}
\text{Get}(H, l, m) &= \text{va} \\
\langle H, l, \text{ln} \ast m \rangle &\rightarrow \langle H, l, \text{va} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{HasOwnProperty}(H, l, m) \\
\text{Dot}(H, l, m) &= \text{va} \\
\text{Get}(H, l, m) &= \text{va}
\end{align*}
\]

\[
\neg(\text{HasOwnProperty}(H, l, m)) \\
H(l).\text{@prototype} = lp \\
\text{Get}(H, l, m) = \text{Get}(H, lp, m)
\]
Exceptions

- What if an intermediate step gives an exception?
- We need to stop further evaluation and throw the exception to the top level.

Example:

\[
\langle H, l, a_0 \rangle \rightarrow \langle H, l, \langle l_{\text{excp}} \rangle \rangle \\
\langle H, l, a_0 + a_1 \rangle \rightarrow \langle H, l, \langle l_{\text{excp}} \rangle + a_1 \rangle \\
\]

Stop evaluation of \( a_2 \).

Context Rule for Exception

\[
\langle H, l, eC[\langle l_{\text{excp}} \rangle] \rangle \rightarrow \langle H, l, \langle l_{\text{excp}} \rangle \rangle \\
\text{where } eC ::= _ | eC \text{ OP } e | \text{ va OP } eC | eC[e] | . . .
\]

We must force all other rules to NOT apply to terms with exception \( \langle l_{\text{excp}} \rangle \) so that determinism is preserved.
Exceptions

What if an intermediate step gives an exception?
We need to stop further evaluation and throw the exception to the top level.

Example:

\[
\begin{align*}
\langle H, l, a_0 \rangle & \rightarrow \langle H, l, \langle l_{\text{exc}} \rangle \rangle \\
\langle H, l, a_0 + a_1 \rangle & \rightarrow \langle H, l, \langle l_{\text{exc}} \rangle + a_1 \rangle
\end{align*}
\]

Stop evaluation of \( a_2 \).

Context Rule for Exception

\[
\langle H, l, eC[\langle l_{\text{exc}} \rangle] \rangle \rightarrow \langle H, l, \langle l_{\text{exc}} \rangle \rangle
\]

where \( eC ::= _- \mid eC \text{ OP } e \mid va \text{ OP } eC \mid eC[e] \mid \ldots \)

We must force all other rules to NOT apply to terms with exception \( \langle l_{\text{exc}} \rangle \) so that determinism is preserved.
With statement

With statement allows arbitrary objects to be placed on top of the scope chain.
Example:

```
var a = 5;
var o = {a:10}
with(o){ a; }
// 10
```

A simple rule for with is:

```
⟨H,l,with(l_{new})s⟩ \rightarrow ⟨H,l_{new},s⟩
```

Is the above rule correct?
Observe that once `with` completes, we need to restore the old scope back!
Continuation as contexts

Lets create a new context $\text{with}(l,\_)$ . The new rule is

$\langle H, l, \text{with}(l_{\text{new}})s \rangle \rightarrow \langle H, l_{\text{new}}, \text{with}(l, s) \rangle$

Then we have separate context rules.

$\langle H, l, s \rangle \rightarrow \langle H', l', s' \rangle$

$\langle H, l, \text{with}(l_{\text{old}}, s) \rangle \rightarrow \langle H', l', \text{with}(l_{\text{old}}, s') \rangle 
\left[ \text{With} - s \right]$

$\langle H, l, \text{with}(l_{\text{old}}, s) \rangle \rightarrow \langle H', l_{\text{old}}, \text{val} \rangle 
\left[ \text{With} - \text{end} \right]$

$\text{with}(l,\_)$ represents some intermediate state reached during the reduction of the with statement.
Formal Properties and Theorems

Notations and Definitions:

- \( Wf(\langle H, l, t \rangle) \): Predicate denoting well-formedness of state \( (\langle H, l, t \rangle) \)
- \( G(H) \): Heap reachability graph of \( H \).

Theorem 1

- Progress: \( Wf(S) \land S \) is not a terminal state \( \Rightarrow (\exists S': S \rightarrow S') \)
  Proof: Structural Induction.
- Preservation: \( Wf(S) \land S \rightarrow S' \Rightarrow Wf(S') \).
  Proof: Induction over the rules.
Garbage Collector for *JavaScript*

- Set of **live heap addresses**: \( \Delta(\langle H_0, l_0, t_0 \rangle) = \{l | l \in t_0\} \cup \{l_0\} \)  
  (Recall \( Var(t) \) in the case of IMP)

**Theorem 2**

Evaluation of a state \( S = \langle H, l, t \rangle \) only depends on the portion of the heap reachable from the Heap addresses \( \Delta(S) \).

- Immediate Consequence: Mark and sweep garbage collector.
- Garbage collect all heap addresses not reachable from \( \Delta(S) \) in the heap reachability graph.
- By theorem 2, the semantics of \( S \) is preserved during garbage collection.
Open Problem: $\alpha$-renaming

- $\alpha$ renaming a program refers to renaming all the identifiers appearing in the program.
- Done by FBJS as a security measure. Prepends all identifiers with user's application id.
- Will this change preserve the meaning of the program in C? in JavaScript?

No, think about eval!

Problem

Find a meaningful subset of JavaScript such that the semantics of any program written in that subset would be invariant under such a renaming.
Open Problem: $\alpha$-renaming

- $\alpha$ renaming a program refers to renaming all the identifiers appearing in the program.
- Done by FBJS as a security measure. Prepends all identifiers with user’s application id.
- Will this change preserve the meaning of the program? in C? in JavaScript?

No, think about eval!

Problem

Find a meaningful subset of JavaScript such that the semantics of any program written in that subset would be invariant under such a renaming.
1. Need for a Formal Semantics?

2. Structural Operational Semantics for IMP
   - Formalizing the program state
   - Semantic Rules
   - Formal properties

3. Structural Operational Semantics for JavaScript
   - Motivation
   - Main features of JavaScript
   - Formalizing the Program state
   - Semantic Rules
   - Formal Properties

4. Applications of the Operational Semantics for JavaScript
   - BeamAuth
   - ADSafe

5. Conclusions and Future work
Real Application: BeamAuth (Ben Adida)

- Two factor authentication mechanism, proposed by Ben Adida.
- Consider visiting the login page of your bank (*might be a phishing page!*).
- A piece of *JavaScript* sits in one of your bookmarks which contains your login information.
- When the bookmarklet is clicked, it verifies whether the source of the page is [www.bank.com](http://www.bank.com) and then fills in the login info on your behalf.

**Problem**

- Write a JavaScript program which can check whether a page belongs to the hostname bank.com.
Solution

Attempt 1 - Seems Easy

If (window.location.host === "bank.com") return "good page" else return "bad page"

Attack: var window = {location:{host: "bank.com"}}
Mitigation: Use this instead of window.

Attempt 2 - I can fix it

If (this.location.host === "bank.com") return "good page" else return "bad page"

Attack: window._defineGetter_("location", function () {return {host: "bank.com"}})
Mitigation: Use __lookupGetter__
Solution

Attempt 1 - Seems Easy

If (window.location.host === "bank.com") return "good page" else return "bad page"

Attack: var window = {location:{host: "bank.com"}}
Mitigation: Use this instead of window.

Attempt 2 - I can fix it

If (this.location.host === "bank.com") return "good page" else return "bad page"

Attack: window._defineGetter_("location", function () {return {host: "bank.com"}})
Mitigation: Use __lookupGetter__
Solution

Attempt 1 - Seems Easy

If (window.location.host === "bank.com") return "good page" else return "bad page"

Attack: var window = {location:{host: "bank.com"}}
Mitigation: Use this instead of window.

Attempt 2 - I can fix it

If (this.location.host === "bank.com") return "good page" else return "bad page"

Attack: window._defineGetter_("location", function () {return {host: "bank.com"}})
Mitigation: Use __lookupGetter__
Solution

Attempt 1 - Seems Easy

If (window.location.host === "bank.com") return "good page" else return "bad page"

Attack: var window = {location:{host: "bank.com"}}
Mitigation: Use this instead of window.

Attempt 2 - I can fix it

If (this.location.host === "bank.com") return "good page" else return "bad page"

Attack: window._defineGetter_("location", function () {return {host: "bank.com"}})
Mitigation: Use __lookupGetter__
Solution

Attempt 1 - Seems Easy

If (window.location.host === "bank.com") return "good page" else return "bad page"

Attack: var window = {location:{host: "bank.com"}}
Mitigation: Use this instead of window.

Attempt 2 - I can fix it

If (this.location.host === "bank.com") return "good page" else return "bad page"

Attack: window._defineGetter_("location", function (){return {host: "bank.com"}})
Mitigation: Use __lookupGetter__
Cat and mouse game went on

**Attempt 15 - Does this work?**

```javascript
function isNormalProperty(obj, propertyName){
    for (var p in obj){
        if (p === "__lookupGetter__") return false;
        if (p === "__lookupSetter__") return false; }
    if (typeof obj.__lookupGetter__("__lookupGetter__") !== undefined)
        return false;
    if (this.__lookupGetter__("__lookupSetter__") !== undefined)
        return false; }
    if (!isNormalProperty(this, "location")) return "Attack! ;
    if (!isNormalProperty("#", "host")) return Attack;
        this.location = ";
    if (this.__lookupGetter__("location") === "beamauth.org")
        doLoginStuff(SECRET_TOKEN);
    }
```

Attack : __lookupGetter__ could be redefined :-(

Ankur Taly | A Structural Operational Semantics for JavaScript
Cat and mouse game went on

Attempt 15 - Does this work?

```javascript
function isNormalProperty(obj, propertyName){
    for (var p in obj){
        if (p === "__lookupGetter__") return false;
        if (p === "__lookupSetter__") return false;
    }
    if (typeof obj.__lookupGetter__('__lookupGetter__') !== undefined)
        return false;
    if (this.__lookupGetter__('__lookupSetter__') !== undefined)
        return false;
    if (!isNormalProperty(this, "location")) return "Attack! ;
    if (!isNormalProperty("#", "host")) return Attack;
        this.location = ";
    if (this.__lookupGetter__('"location") === "beamauth.org")
        doLoginStuff(SECRET_TOKEN);

    Attack : __lookupGetter__ could be redefined :-(
```
May be its impossible !!!

- $H_{\text{good}}$: Heap state when actual `bank.com` is loaded.
- $H_{\text{bad}}$: Heap state when the phishing page is loaded.

Two notions of impossibility

- **Weak Notion**: For every BeamAuth script that we can write, there exists a bad page that can still go through.
- **Strong Notion**: There exists a phishing page for which the heap state $H_{\text{bad} - \text{strong}}$ is such that:
  - $H_{\text{bad} - \text{strong}}$ is observationally equivalent to $H_{\text{good}}$.
  - No BeamAuth can distinguish $H_{\text{bad} - \text{strong}}$ from $H_{\text{good}}$.

The Strong Notion is more powerful than the weak one.
The Operational semantics of JavaScript is the first step towards formally proving such results.
ADSafe (Douglas Crockford)

- **ADSafe** is a solution proposed by Yahoo for controlling the interaction between the trusted and untrusted code.

**Basic Idea:**

1. Represents a safe subset of JavaScript.
2. Wraps untrusted code inside a safe object called ADSafe object.
3. **All interaction with the trusted code happens only using the methods in the ADSafe object.**
4. Untrusted code can be statically checked to ensure that it only calls methods of the ADSafe object (Tool : JSLint).

More information on [http:www.adsafe.org](http:www.adsafe.org)
var ADSAFE = function () {
    ....
    var reject = function (object, name) {
        return object === window || typeof object !== 'object' ||
            (typeof name !== 'number' &&
             (typeof name !== 'string' || name.charAt(0) == '_'));
    }
    return {
        get: function (object, name) {
            var value;
            if (!reject(object, name) && object.hasOwnProperty(name)) {
                value = object[name];
                if (typeof value !== 'function' && value !== window) {
                    return value; }
            error(); },

        set: function (object, name, value) {
            ....
        }
    }();
}

Usage
To access property $p$ of object $o$: ADSAFE.get($o, p$)
Challenges and Issues

- Consider the following property: "All interaction with the trusted code happens only using the methods in the ADSafe object."
  Is this achievable?
- Consider the following code:
  ```javascript
  var o = {a:10};
  var arr = [10,11];
  arr[o];
  ```
  This function implicitly calls `Object.prototype.toString`, which is a function defined in the trusted space.
- What if `toString` in turn leaks out pointer to global object?

Conclusion

Besides the untrusted code, ADSafe has to impose restrictions on the native functions and objects present in the trusted space.
Challenges and Issues

- Consider the following property: "All interaction with the trusted code happens only using the methods in the ADSafe object."
  Is this achievable?
- Consider the following code:
  ```javascript
  var o = {a:10};
  var arr = [10,11];
  arr[o];
  ```
  This function implicitly calls `Object.prototype.toString`, which is a function defined in the trusted space.
  What if `toString` in turn leaks out pointer to global object?

Conclusion

Besides the untrusted code, ADSafe has to impose restrictions on the native functions and objects present in the trusted space.
Conclusions and Future work

Conclusions:

- First step towards formal analysis of whole of JavaScript.
- We have formalized the entire ECMA 2.62 language. Complete set of rules (in ASCII) span 70 pages.
- Prove basic soundness properties like progress and preservation for the semantics and the fact that JavaScript is garbage collectible.

Future Work:

- Add features like setters/getters (not present in ECMA 2.62), which are present in various browsers.
- Formalize interaction with the DOM.
- Apply the semantics for security analysis of applications such as ADSafe and BeamAuth.
Conclusions and Future work

Conclusions:

- First step towards formal analysis of whole of JavaScript.
- We have formalized the entire ECMA 2.62 language. Complete set of rules (in ASCII) span 70 pages.
- Prove basic soundness properties like progress and preservation for the semantics and the fact that JavaScript is garbage collectible.

Future Work:

- Add features like setters/getters (not present in ECMA 2.62), which are present in various browsers.
- Formalize interaction with the DOM.
- Apply the semantics for security analysis of applications such as ADSafe and BeamAuth.
Thank You!