A Structural Operational Semantics for JavaScript

Ankur Taly

Dept. of Computer Science, Stanford University

Joint work with Sergio Maffeis and John C. Mitchell

Outline

JavaScript

- Widely used web programming language.
- History :
  - Developed by Brendan Eich at Netscape.
  - Standardized for Browser Compatability : ECMAScript 262-edition 3
- Interesting and unusual features
  - First class functions
  - Prototype based language
  - Powerful modification capabilities : can convert string to code (eval), can redefine object methods !
- Very important to fully understand the Semantics so as to reason about the security properties of programs written in it.

Big Picture

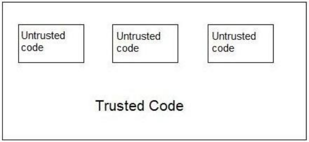

Figure: Trusted and Untrusted code

- Many websites include untrusted *JavaScript* content:
  - Third party advertisements
  - Social Networking sites : User written applications
  - Web Mashups

Isolation Goal

Allow untrusted code to perform valuable interactions at the same time prevent intrusion and malicious damage.

Real World Example

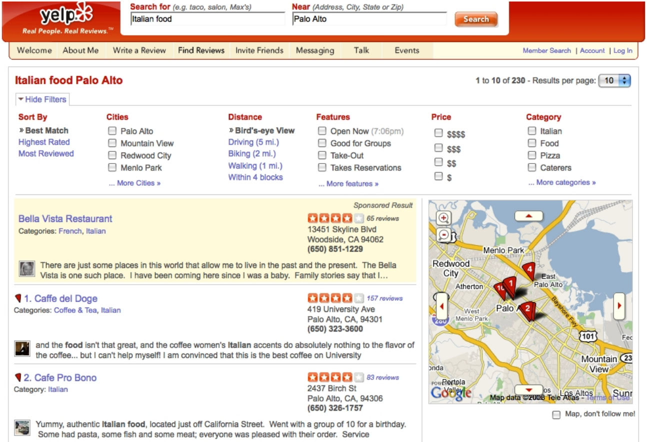

Figure: Web Mashup

Formulating the Problem

Static Analysis Problem

Given an untrusted JavaScript program P and a Heap H (corresponding to the trusted page), determine if P accesses any security critical portions of the Heap.

Very hard problem to solve for whole of JavaScript as all code that gets executed may not appear textually ! Example :

```
var m = "toString=func"; var n = "tion(){return undefined};";
eval(m + n);
```

Simplification : Solve the above problem for subsets of JavaScript that are more amenable to static analysis.

First step

Define a Formal semantics for complete JavaScript

Formulating the Problem

Static Analysis Problem

Given an untrusted JavaScript program P and a Heap H (corresponding to the trusted page), determine if P accesses any security critical portions of the Heap.

Very hard problem to solve for whole of JavaScript as all code that gets executed may not appear textually ! Example :

```
var m = "toString=func"; var n = "tion(){return undefined};";
eval(m + n);
```

Simplification : Solve the above problem for subsets of JavaScript that are more amenable to static analysis.

First step

Define a Formal semantics for complete JavaScript

Informal Semantics

- ECMA262-3 specification manual - currently in its third edition.
- 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.
  - Example : $\mathit{Meaning}[\mathrm{x} = \mathrm{x} + 10]$ only depend on value of x ?
  - in C ? Yes
  - in JavaScript ?

Example

```
var y = "a";
var x = {toString : function(){ return y;}}

x = x + 10;
js> "a10"
```

- Implicit type conversion of an object to a string in JavaScript involves calling the `toString` function.
- Informal semantics fail to emphasize such examples.

Formal Semantics

- Specify meaning in a Mathematically rigorous way.
- Provides a framework for proving properties of the kind mentioned on the previous slide.
- Our Goal
  - Convert Informal semantics(ECMA262-3) into a Formal semantics. (Done ! This talk)
  - Analyze existing safe subsets of JavaScript and formally prove the security properties that they entail. (Ongoing work)
- The very act of formalization revealed subtle aspects of the language.

Related work

- Giannini et al (ECOOP 05), Thiemann (ESOP 05)
  - Formalized a small (but non-trivial) subset of JavaScript
  - Provided a static type system but substantially simplified the semantics.
- We found examples of programs that are well-typed according to these simplified semantics but ill-typed according to the complete semantics and vice versa.

Our Contribution : A Structural Operational semantics for complete ECMA standard language.

Advantages:

- Ability to analyze semantics of arbitrary JavaScript code.
- Gives us a more systematic way of designing the subsets, parametric on the desired security properties.

Related work

- Giannini et al (ECOOP 05), Thiemann (ESOP 05)
  - Formalized a small (but non-trivial) subset of JavaScript
  - Provided a static type system but substantially simplified the semantics.
- We found examples of programs that are well-typed according to these simplified semantics but ill-typed according to the complete semantics and vice versa.

Our Contribution : A Structural Operational semantics for complete ECMA standard language.

Advantages:

- Ability to analyze semantics of arbitrary JavaScript code.
- Gives us a more systematic way of designing the subsets, parametric on the desired security properties.

Outline

Structural Operational Semantics (Gordon Plotkin)

- 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.
- Specify the transitions in a syntax oriented manner using the inductive nature of the abstract syntax.

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 ! However this allowed in all browsers

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 ! However this allowed in all browsers

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 is a chain of activation records, which is called the scope chain.
- Arbitrary objects can be placed over the scope chain - with(e) s construct.

JavaScript : Subtle Features

Example 2

```
var f = function(){if (true) {function g() { return 1;}; }
                   else {function g() { return 2;};}
                   var g = function() { return 3;}
                   return g();
                   function g(){ return 4;}}
var result = f();
```

What is the final value of result ?

JavaScript : Subtle Features

Example 2

```
var f = function(){if (true) {function g() { return 1;}; }
                   else {function g() { return 2;};}
                   var g = function() { return 3;}
                   return g();
                   function g(){ return 4;}}
var result = f();
```

What is the final value of result ?

- result = 2 (according to ECMA262-3)
- Function body is parsed to process all variable declarations before the function call is executed !
- Different implementations chose different declarations : Mozilla Spidermonkey : 4, Safari : 1 !

Formal Semantics : Program state

- All objects are passed by reference $\Rightarrow$ The store must have information about Heap locations.
- Variables have different values in different scopes $\Rightarrow$ 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.

Semantic Rules

- Three semantic functions $\xrightarrow{e}$, $\xrightarrow{s}$, $\xrightarrow{P}$ for expressions, statements and programs.
- Small step transitions : A semantic function transforms one state to another if certain conditions (premise) are true.
- General form : $\dfrac{\langle Premise \rangle}{S \xrightarrow{t} S'}$
- Atomic Transitions : Rules which do have another transition in their premise
- Context rules : Rules to apply atomic transitions in presence of certain specific contexts.

Heap and Heap Reachability Graph

l_0^A	@scope : #global
#global	"a" : l_1^A
l_1^A	"n" : l_2^A "p" : l_3^A
l_2^A	"value" : 10
l_3^A	"value" : 20
l_4^A	"@scope" : l_0^A

Heap

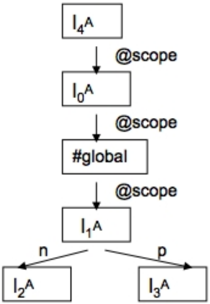

Figure: Heap and its reachability graph

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.

Scope and Prototype lookup

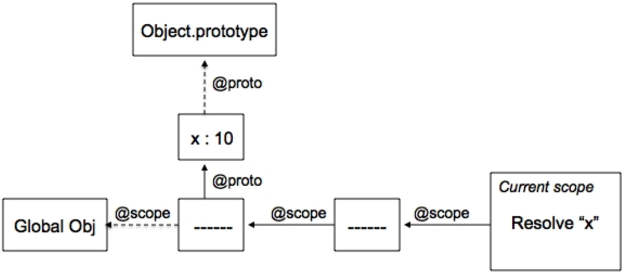

Figure: Scope and Prototype lookup

- 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 (essence):

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.

$$\frac{\texttt{Scope}(\texttt{H},\texttt{l},\text{"x"}) = \texttt{ln}}{\langle \texttt{H},\texttt{l},\texttt{x}\rangle \xrightarrow{\texttt{e}} \langle \texttt{H},\texttt{l},\texttt{ln} * \text{"x"}\rangle}$$

$$\frac{\texttt{HasProperty}(\texttt{H},\texttt{l},\texttt{m})}{\texttt{Scope}(\texttt{H},\texttt{l},\texttt{m}) = \texttt{l}}$$

$$\frac{\neg(\texttt{HasProperty}(\texttt{H},\texttt{l},\texttt{m})) \quad \texttt{H}(\texttt{l}).\texttt{@Scope} = \texttt{ln}}{\texttt{Scope}(\texttt{H},\texttt{l},\texttt{m}) = \texttt{Scope}(\texttt{H},\texttt{ln},\texttt{m})}$$

$$\texttt{Scope}(\texttt{H},\texttt{null},\texttt{m}) = \texttt{null}$$

Prototype lookup : Rules

ECMA 2.62 :

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

$$\frac{\mathtt{H_2}, \mathtt{l_{excp}} = \mathtt{alloc}(\mathtt{H}, \mathtt{o}) \quad \mathtt{o} = \mathtt{newNativeErr}(\text{""}, \#\mathtt{RefErrProt})}{\langle \mathtt{H}, \mathtt{l}, (\mathtt{null} * \mathtt{m}) \rangle \xrightarrow{\mathtt{e}} \langle \mathtt{H_2}, \mathtt{l}, \langle \mathtt{l_{excp}} \rangle \rangle}$$

$$\frac{\mathtt{Get}(\mathtt{H}, \mathtt{l}, \mathtt{m}) = \mathtt{va}}{\langle \mathtt{H}, \mathtt{l}, \mathtt{ln} * \mathtt{m} \rangle \xrightarrow{\mathtt{e}} \langle \mathtt{H}, \mathtt{l}, \mathtt{va} \rangle}$$

$$\frac{\mathtt{HasOwnProperty}(\mathtt{H}, \mathtt{l}, \mathtt{m}) \quad \mathtt{Dot}(\mathtt{H}, \mathtt{l}, \mathtt{m}) = \mathtt{va}}{\mathtt{Get}(\mathtt{H}, \mathtt{l}, \mathtt{m}) = \mathtt{va}}$$

$$\frac{\neg(\mathtt{HasOwnProperty}(\mathtt{H}, \mathtt{l}, \mathtt{m})) \quad \mathtt{H}(\mathtt{l}).\mathtt{@prototype} = \mathtt{lp}}{\mathtt{Get}(\mathtt{H}, \mathtt{l}, \mathtt{m}) = \mathtt{Get}(\mathtt{H}, \mathtt{lp}, \mathtt{m})}$$

Exceptions

- When an intermediate step gives an exception, stop further evaluation and throw the exception to the top level.
- Example :

$$\frac{\langle \mathtt{H}, \mathtt{l}, \mathtt{a}_0 \rangle \longrightarrow \langle \mathtt{H}, \mathtt{l}, \langle \mathtt{l}_{\mathtt{excp}} \rangle \rangle}{\langle \mathtt{H}, \mathtt{l}, \mathtt{a}_0 + \mathtt{a}_1 \rangle \longrightarrow \langle \mathtt{H}, \mathtt{l}, \langle \mathtt{l}_{\mathtt{excp}} \rangle + \mathtt{a}_1 \rangle}$$

  Stop evaluation of a_2.
- Use context based reduction rules (Felleisen)

Context Rule for Exceptions

$$\langle \mathtt{H}, \mathtt{l}, \mathtt{eC}[\langle \mathtt{l}_{\mathtt{excp}} \rangle] \rangle \longrightarrow \langle \mathtt{H}, \mathtt{l}, \langle \mathtt{l}_{\mathtt{excp}} \rangle \rangle$$

where `eC ::= _ | eC OP e | va OP eC | eC[e] | ...`

Exceptions

- When an intermediate step gives an exception, stop further evaluation and throw the exception to the top level.
- Example :

$$\frac{\langle \mathtt{H}, \mathtt{l}, \mathtt{a}_0 \rangle \longrightarrow \langle \mathtt{H}, \mathtt{l}, \langle \mathtt{l}_{\mathtt{excp}} \rangle \rangle}{\langle \mathtt{H}, \mathtt{l}, \mathtt{a}_0 + \mathtt{a}_1 \rangle \longrightarrow \langle \mathtt{H}, \mathtt{l}, \langle \mathtt{l}_{\mathtt{excp}} \rangle + \mathtt{a}_1 \rangle}$$

Stop evaluation of a_2.

- Use context based reduction rules (Felleisen)

Context Rule for Exceptions

$$\langle \mathtt{H}, \mathtt{l}, \mathtt{eC}[\langle \mathtt{l}_{\mathtt{excp}} \rangle] \rangle \longrightarrow \langle \mathtt{H}, \mathtt{l}, \langle \mathtt{l}_{\mathtt{excp}} \rangle \rangle$$

where `eC ::= _ | eC OP e | va OP eC | eC[e] | ...`

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 :

$$\langle \mathtt{H}, \mathtt{l}, \mathtt{with}(\mathtt{l}_{\mathtt{new}})\mathtt{s}\rangle \xrightarrow{\mathtt{s}} \langle \mathtt{H}, \mathtt{l}_{\mathtt{new}}, \mathtt{s}\rangle$$

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

Continuation as contexts

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

$$\langle \mathtt{H}, \mathtt{l}, \mathtt{with}(\mathtt{l}_{\mathtt{new}})\mathtt{s}\rangle \xrightarrow{\mathtt{s}} \langle \mathtt{H}, \mathtt{l}_{\mathtt{new}}, \mathtt{\$with}(\mathtt{l}, \mathtt{s})\rangle$$

Then we have separate context rules.

$$\frac{\langle \mathtt{H}, \mathtt{l}, \mathtt{s}\rangle \xrightarrow{\mathtt{s}} \langle \mathtt{H}', \mathtt{l}', \mathtt{s}'\rangle}{\langle \mathtt{H}, \mathtt{l}, \mathtt{\$with}(\mathtt{l}_{\mathtt{old}}, \mathtt{s})\rangle \xrightarrow{\mathtt{s}} \langle \mathtt{H}', \mathtt{l}', \mathtt{\$with}(\mathtt{l}_{\mathtt{old}}, \mathtt{s}')\rangle}[\mathit{With} - \mathit{s}]$$

$$\frac{\langle \mathtt{H}, \mathtt{l}, \mathtt{s}\rangle \xrightarrow{\mathtt{s}} \langle \mathtt{H}', \mathtt{l}', \mathtt{val}\rangle}{\langle \mathtt{H}, \mathtt{l}, \mathtt{\$with}(\mathtt{l}_{\mathtt{old}}, \mathtt{s})\rangle \xrightarrow{\mathtt{s}} \langle \mathtt{H}', \mathtt{l}_{\mathtt{old}}, \mathtt{val}\rangle}[\mathit{With} - \mathit{end}]$$

Summary

- We developed an operational semantics for the entire ECMA 2.62 language.
- Complete set of rules (in ASCII) span 70 pages.
- Semantics does not cover features beyond ECMA 2.62, like setters/getters etc, which are present in various browsers.
- We do not model interaction with the Document Object Model (DOM) of web browsers.
- The entire exercise also led to the discovery of several inconsistencies in the various browsers.

Outline

1. Motivation
   - Web Security problem
   - Informal and Formal Semantics
   - Related work
2. Formal Semantics for JavaScript (ECMA262-3)
   - Syntax
   - Main features
   - Semantic rules
3. Formal Properties
4. Conclusions and Future work

Formal Analysis

Contributions :

- Progress/Preservation Theorem
  - Evaluation of a state is never "stuck", and always progresses to a next state or a value or an exception.
  - Essential for any subsequent formal analysis to make sense
- Heap Reachability Theorem
  - Characterizing the reachable portion of the Heap
  - Showing that evaluation of a state does not depend on anything outside the reachable portion.
  - First step towards solving the static analysis problem of determining if a program can potentially access any security-critical portions.

Soundness

Notations and Definitions :

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

Theorem

- Progress :
  $Wf(S) \wedge S$ is not a terminal state $\Rightarrow (\exists S' : S \rightarrow S')$
  Proof Idea : Induction over the structure of terms.
- Preservation : $Wf(S) \wedge S \rightarrow S' \Rightarrow Wf(S')$.
  Proof Idea : Induction over the rules.

Heap Reachability

- Set of root addresses : $\Delta(\langle H_0, l_0, t_0 \rangle) = \{l | l \in t_0\} \cup \{l_0\}$ item $view_H(l)$: Subgraph of the Heap reachability graph consisting only of nodes reachable from l.

Theorem

Evaluation of a state $S = \langle H, l, t \rangle$ only depends on the Heap addresses corresponding to nodes reachable from the set of root nodes in the Heap reachability graph..

Proof Idea

Consider any two states S_1 and S_2. Define $S_1 \sim S_2$ iff

- There exists a heap address renaming function $f : dom(H) \rightarrow \mathbb{L}$.
- $\Delta(S_1) = f(\Delta(S_2))$.
- For all $l \in \Delta(S_1)$, $view_{H_1}(l) = view_{f(H_2)}(l)$

Theorem

$S_1 \sim S_2 \wedge S_1 \rightarrow S_1' \Rightarrow \exists S_2'.\, S_2 \rightarrow S_2' \wedge S_1' \sim S_2'$.

Proof Idea: Induction over the set of rules.

Garbage Collector for *JavaScript*

- Immediate Consequence of Reachability theorem.
- Mark and sweep garbage collector : For a particular state S, Garbage collect all heap addresses not reachable from $\Delta(S)$ in the heap reachability graph.
- By Reachability theorem , the semantics of S is preserved during garbage collection.

A glimpse of Ongoing work

- Central Problem : Design safe subset of JavaScript more amenable to static analysis.
- Main Idea
  - Filtering ⇔ Syntactic subset.
    Example : Forbid use of evil constructs like eval, Function, e[e] etc
  - Rewriting ⇔ Semantic subset.
    Example : Rename all identifiers appearing in the program to separate out the namespace of untrusted code.

A glimpse of Ongoing work

- Breakdown central problem into designing subsets with certain language properties
  - Design a subset of JavaScript such that for all programs in that subset, every property name that is accessed appears textually in the code.
  - Design a subset of JavaScript such that the semantics of any program in that subset does not change under renaming of identifiers.
  - . . .

Starting point for systematically designing and proving the desired language property for each of these subsets is the formal semantics for entire JavaScript.

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`

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 :

```
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 :

```
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 JavaScriptis garbage collectible.

Future Work :

- Add features like setters/getters (not present in ECMA 2.62) and formalize interaction with DOM.
- Encode the semantics in a machine readable format.
- Apply the semantics for security analysis of safe fragments of JavaScript such as AdSafe (Yahoo !), FBJS (FaceBook).

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 JavaScriptis garbage collectible.

Future Work :

- Add features like setters/getters (not present in ECMA 2.62) and formalize interaction with DOM.
- Encode the semantics in a machine readable format.
- Apply the semantics for security analysis of safe fragments of JavaScript such as AdSafe (Yahoo !), FBJS (FaceBook).

Thank You !