Program Analysis for JavaScript
– Challenges and Techniques

Anders Møller
Center for Advanced Software Analysis
Aarhus University

Joint work with
Esben Andreasen, Simon Holm Jensen, Peter A. Jonsson, Magnus Madsen, and Peter Thiemann
JavaScript needs program analysis

- **Testing** is still the only technique programmers have for finding errors in their code

- **Program analysis** can (in principle) be used for
  - bug detection (e.g. "x.p in line 7 always yields *undefined*")
  - code completion
  - optimization

![Percentage of websites using various client-side programming languages](chart.png)
JavaScript is a *dynamic language*

- Object-based, properties created on demand
- Prototype-based inheritance
- First-class functions, closures
- Runtime types, coercions
- ...

**NO STATIC TYPE CHECKING**
**NO STATIC CLASS HIERARCHIES**
The goal:

Catch type-related errors using program analysis

- Support the full language
- Aim for soundness
Statically detecting type-related errors in JavaScript programs
Likely programming errors

1. invoking a non-function value (e.g. undefined) as a function
2. reading an absent variable
3. accessing a property of `null` or `undefined`
4. reading an absent property of an object
5. writing to variables or object properties that are never read
6. calling a function object both as a function and as a constructor, or passing function parameters with varying types
7. calling a built-in function with an invalid number of parameters, or with a parameter of an unexpected type

etc.
Which way to go?

type inference?

prototype-based inheritance?

flow-sensitivity?

heap modeling?

call graph construction?

standard library?

coercion?
The TAJs approach

- Dataflow analysis (abstract interpretation) using the monotone framework
  [Kam & Ullman ’77]

- The recipe:
  1. construct a control flow graph for each function in the program to be analyzed
  2. define an appropriate dataflow lattice (abstraction of data)
  3. define transfer functions (abstraction of operations)

[Jensen, Møller, and Thiemann, SAS’09]
The dataflow lattice (simplified!)

- The analysis maintains an abstract state for each program point \(N\) and call context \(C\):
  \[ N \times C \rightarrow \text{State} \]
- Each abstract state provides an abstract value for each abstract object \(L\) and property name \(P\):
  \[ \text{State} = L \times P \rightarrow \text{Value} \]
- Each abstract value describes pointers and primitive values:
  \[ \text{Value} = \mathcal{P}(L) \times \text{Bool} \times \text{Str} \times \text{Num} \ldots \]
- Details refined through trial-and-error...

Key ideas:
- flow sensitivity
- context sensitivity (object sensitivity)
- pointer analysis with allocation site abstraction
- constant propagation
- recency abstraction
- lazy propagation
Transfer functions, example

A dynamic property read:  \( x[y] \)

1. Coerce \( x \) to objects
2. Coerce \( y \) to strings
3. Descend the object prototype chains to find the relevant properties
4. Join the property values
function Person(n) {
    this.setName(n);
    Person.prototype.count++;
}
Person.prototype.count = 0;
Person.prototype.setName = function(n) { this.name = n; }

function Student(n,s) {
    this.b = Person;
    this.b(n);
    delete this.b;
    this.studentid = s.toString();
}
Student.prototype = new Person;

var t = 100026;
var x = new Student("Joe Average", t++);
var y = new Student("John Doe", t);
y.setName("John Q. Doe");

does y have a setName method at this program point?
An abstract state (as produced by **TAJS**)

- **F_0**
  - length: 1
  - prototype: [F_0_PROTO]
  - [[Prototype]]: [OBJECT_PROTO]
  - [[Value]]: NaN

- **F_Person**
  - length: 1
  - prototype: [F_Person_PROTO]
  - [[Prototype]]: [FUNCTION_PROTO]
  - [[Scope]]: [[GLOBAL]]

- **GLOBAL**
  - Person: [F_Person]
  - Student: [F_Student]
  - t: 100027
  - x: [L0]
  - y: [L1]
  - [[Prototype]]: [OBJECT_PROTO]

- **L0**
  - studentid: "100026"
  - name: "Joe Average"
  - [[Prototype]]: [F_Student_PROTO]

- **L1**
  - studentid: "100027"
  - name: "Amy"
  - [[Prototype]]: [F_Student_PROTO]
JavaScript web applications

• Modeling JavaScript code is not enough...

• The environment of the JavaScript code:
  – the ECMAScript standard library
  – the browser API
  – the HTML DOM
  – the event mechanism

[Jensen, Madsen, and Møller, ESEC/FSE’11]
Eval in JavaScript

- `eval(S)`
  - parse the string $S$ as JavaScript code, then execute it

- Challenging for JavaScript static analysis
  - the string is dynamically generated
  - the generated code may have side-effects
  - and JavaScript has poor encapsulation mechanisms
Eval is evil

• ... but most uses of eval are not very complex
• So let’s transform eval calls into other code!
• How can we soundly make such transformations if we cannot analyze code with eval?

Which came first?

Analysis or transformation
Whenever **TAJS** detects new dataflow to `eval`, the `eval` transformer is triggered.

[Jensen, Jonsson, and Møller, ISSTA’12]
An example

```javascript
var y = "foo"
for (i = 0; i < 10; i++) {
    eval(y + "(" + i + ")")
}
```

The dataflow analysis propagates dataflow until the fixpoint is reached

- iteration 1:  y is "foo", i is 0
  ```javascript
eval(y + "(" + i + ")") \Rightarrow foo(0)
  ```
  (the dataflow analysis can now proceed into foo)

- iteration 2:  y is "foo", i is `AnyNumber`
  ```javascript
eval(y + "(" + i + ")") \Rightarrow foo(i)
  ```

- ... (would not work if i could be any string)
Ingredients in a static analyzer for JavaScript applications

We need to model

- the language semantics
- the standard library (incl. `eval`)
- the browser API (the HTML DOM, the event system, etc.)
Mission complete?
<table>
<thead>
<tr>
<th>Library</th>
<th>Absolute Usage Percentage</th>
<th>Market Share</th>
</tr>
</thead>
<tbody>
<tr>
<td>JQuery</td>
<td>57.5%</td>
<td>93.1%</td>
</tr>
<tr>
<td>MooTools</td>
<td>4.9%</td>
<td>8.0%</td>
</tr>
<tr>
<td>Modernizr</td>
<td>4.6%</td>
<td>7.5%</td>
</tr>
<tr>
<td>Prototype</td>
<td>2.7%</td>
<td>4.4%</td>
</tr>
<tr>
<td>ASP.NET Ajax</td>
<td>2.3%</td>
<td>3.7%</td>
</tr>
<tr>
<td>Script.aculo.us</td>
<td>2.1%</td>
<td>3.5%</td>
</tr>
<tr>
<td>YUI Library</td>
<td>0.8%</td>
<td>1.4%</td>
</tr>
<tr>
<td>Spry</td>
<td>0.5%</td>
<td>0.8%</td>
</tr>
<tr>
<td>Shadowbox</td>
<td>0.5%</td>
<td>0.8%</td>
</tr>
<tr>
<td>Dojo</td>
<td>0.1%</td>
<td>0.2%</td>
</tr>
<tr>
<td>Underscore</td>
<td>0.1%</td>
<td>0.2%</td>
</tr>
<tr>
<td>Ext JS</td>
<td>0.1%</td>
<td>0.1%</td>
</tr>
<tr>
<td>Backbone</td>
<td>0.1%</td>
<td>0.1%</td>
</tr>
<tr>
<td>AngularJS</td>
<td>less than 0.1%</td>
<td>0.1%</td>
</tr>
<tr>
<td>Knockout</td>
<td>less than 0.1%</td>
<td>0.1%</td>
</tr>
</tbody>
</table>

Percentages of websites using various JavaScript libraries
Note: a website may use more than one JavaScript library

W3Techs.com, 31 January 2014
Why use jQuery (or other libraries)?

★ Patches browser incompatibilities
★ CSS3-based DOM navigation
★ Event handling
★ AJAX (client-server communication)
★ UI widgets and animations
★ 1000s of plugins available
An appetizer

Which code fragment do you prefer?

```javascript
var checkedValue;
var elements = document.getElementsByTagName('input');
for (var n = 0; n < elements.length; n++) {
    if (elements[n].name == 'someRadioGroup' &&
        elements[n].checked) {
        checkedValue = elements[n].value;
    }
}
```

```javascript
var checkedValue = $('*[name="someRadioGroup"]:checked').val();
```
Investigating the beast

<table>
<thead>
<tr>
<th>jQuery version</th>
<th>LOC</th>
<th>load-LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0.0</td>
<td>996</td>
<td>272</td>
</tr>
<tr>
<td>1.1.0</td>
<td>1,141</td>
<td>300</td>
</tr>
<tr>
<td>1.2.0</td>
<td>1,504</td>
<td>296</td>
</tr>
<tr>
<td>1.3.0</td>
<td>2,150</td>
<td>648</td>
</tr>
<tr>
<td>1.4.0</td>
<td>2,851</td>
<td>737</td>
</tr>
<tr>
<td>1.5.0</td>
<td>3,610</td>
<td>924</td>
</tr>
<tr>
<td>1.6.0</td>
<td>3,923</td>
<td>1,003</td>
</tr>
<tr>
<td>1.7.0</td>
<td>4,096</td>
<td>1,118</td>
</tr>
<tr>
<td>1.8.0</td>
<td>4,075</td>
<td>1,157</td>
</tr>
<tr>
<td>1.9.0</td>
<td>4,122</td>
<td>1,161</td>
</tr>
<tr>
<td>1.10.0</td>
<td>4,144</td>
<td>1,193</td>
</tr>
<tr>
<td>2.0.0</td>
<td>3,775</td>
<td>1,101</td>
</tr>
</tbody>
</table>
Experimental results for jQuery with **WALA**:  
- can analyze a JavaScript program that **loads jQuery and does nothing else**  
- no success on jQuery 1.3 and beyond 😞

The **WALA** approach:  
1) dynamic analysis to infer *determinate* expressions that always have the same value in any execution (but for a specific calling context)  
2) exploit this information in context-sensitive pointer analysis

Example of imprecision that explodes

A dynamic property read:  \texttt{x[y]}

- if \( x \) may evaluate to the global object
- and \( y \) may evaluate to a unknown string
- then \( x[y] \) may yield eval, document, Array, Math, ...

consequence
jQuery: sweet on the outside, bitter on the inside

A representative example from the library initialization code:

```javascript
jQuery.each("ajaxStart ajaxStop ... ajaxSend".split(" "),
    function(i, o) {
        jQuery.fn[o] = function(f) {
            return this.on(o, f);
        };
    });
```

which could have been written like this:

```javascript
jQuery.fn.ajaxStart = function(f) { return this.on("ajaxStart", f); };
jQuery.fn.ajaxStop = function(f) { return this.on("ajaxStop", f); };
...
jQuery.fn.ajaxSend = function(f) { return this.on("ajaxSend", f); };
```
each: function (obj, callback, args) {
    var name, i = 0, length = obj.length,
    isObj = length === undefined || jQuery.isFunction(obj);
    if (args) {
        ... // (some lines omitted to make the example fit on one slide)
    } else {
        if (isObj) {
            for (name in obj) {
                if (callback.call(obj[name], name, obj[name]) === false) {
                    break;
                }
            }
        } else {
            for (; i < length ;) {
                if (callback.call(obj[i], i, obj[i++]) === false) {
                    break;
                }
            }
        }
    }
    return obj;
}
Our recent results, by improving TAJS

- **TAJS** can now analyze (in reasonable time)
  - the load-only program for 11 of 12 versions of jQuery
  - 27 of 71 small examples from a jQuery tutorial

- Very good precision for type analysis and call graphs
- Analysis time: 1-24 seconds (average: 6.5 seconds)

[Andreasen and Møller, OOPSLA’14]
TAJS analysis design

- Whole-program, flow-sensitive dataflow analysis
- Constant propagation
- Heap modeling using allocation site abstraction
- Object sensitivity (a kind of context sensitivity)
- Branch pruning (eliminate dataflow along infeasible branches)
- Parameter sensitivity
- Loop specialization
- Context-sensitive heap abstraction

[Andreasen and Møller, OOPSLA’14]
each: function (obj, callback, args) {
  var name, i = 0, length = obj.length,
  isObj = length === undefined || jQuery.isFunction(obj);
  if (args) {
    ...
  } else {
    if (isObj) {
      for (name in obj) {
        if (callback.call(obj[name], name, obj[name])) === false) {
          break;
        }
      }
    } else {
      for (; i < length ;) {
        if (callback.call(obj[i], i, obj[i++])) === false) {
          break;
        }
      }
    }
  }
  return obj;
}
The technical side...

- The analysis maintains an abstract state for each program point $N$ and call context $C$:
  $$N \times C \rightarrow \text{State}$$

- Old TAJS:
  $$C = \mathcal{P}(L) \quad \text{(object sensitivity)}$$
  $$L = N \quad \text{(L: abstract memory locations)}$$

- New TAJS:
  $$C = \mathcal{P}(L) \times (A \rightarrow \text{Value}) \times (B \rightarrow \text{Value})$$
  $$L = N \times C$$

Parameter sensitivity
(A: selected parameters)

Loop specialization
(B: selected local variables)

Context-sensitive heap abstraction
Conclusion

• JavaScript programmers need better tools!
• Static program analysis can detect type-related errors, find dead code, build call graphs, etc.
  – dataflow analysis to model the ECMAScript standard
  – model of the standard library, browser API, and HTML DOM
  – rewrite calls to eval during analysis
  – handle complex libraries by boosting analysis precision
• Progress, but far from a full solution...

Center for Advanced Software Analysis
http://cs.au.dk/CASA