Static Analysis for JavaScript – Challenges and Techniques

Anders Møller
Center for Advanced Software Analysis
Aarhus University

SAS 2015 – Saint-Malo
JavaScript
JavaScript needs static analysis

• **Testing** is still the main technique programmers have for finding errors in their code

• **Static analysis** can (in principle) be used for
  – bug detection (e.g. "x.p in line 7 always yields *undefined*")
  – security vulnerability detection
  – code completion and navigation in IDEs
  – optimization
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**
Type Analysis for JavaScript

Goals:

• Catch type-related errors using static analysis

• Support the full language

• Aim for soundness
TAJS in Eclipse
Related static analysis tools

SAFE: JS Analysis Framework

JSAl: A Static Analysis Platform for JavaScript
Type-related errors in JavaScript

```javascript
var x = [
    "Static","Analysis","Symposium"];
for (var i = 0; i < x.length; i++) {
    console.log(x[i]);
}
```
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.

See also The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript, Pradel & Sen, ECOOP 2015
Research methodology

1. Identify interesting problem
2. Design initial analysis
3. Implement, evaluate experimentally
4. Refine design and analysis
5. Identify bottleneck
6. Does it work perfectly? Too imprecise? Too slow?
Which way to go?

direction

- type inference?
- prototype-based inheritance?
- flow-sensitivity?
- heap modeling?
- call graph construction?
- standard library?
- coercion?
The **TAJS** approach

[Jensen, Møller, and Thiemann, SAS’09]

- Dataflow analysis / abstract interpretation using monotone frameworks
  [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)
Control flow graphs

- Convenient intermediate representation of JavaScript programs

- **Nodes** describe primitive instructions

- **Edges** describe *intra*-procedural control-flow

- Relatively **high-level** IR (unlike e.g. $\lambda_{JS}$)
The dataflow lattice (simplified!)

- For each program point $N$ and call context $C$, the analysis maintains an abstract state:
  $$N \times C \rightarrow \text{State}$$
- 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
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
A tiny example...

```javascript
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");
```

- declares a “class” named Person
- declares a “static field” named count
- declares a shared method named setName
- declares a “sub-class” named Student
- creates two Student objects...
- does y have a setName method at this program point?
An abstract state (as produced by TAJS)
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

  around 250 abstract objects with 500 properties and 200 functions...

[Jensen, Madsen, and Møller, ESEC/FSE’11]
Some experiments

Good results on analyzing small web applications from Chrome Experiments, IE 9 Test Drive, and 10K Challenge

Some ways to measure analysis precision:

• most call sites and property reads are safe
• most call sites are monomorphic
• most expressions have a unique type
• most spelling errors cause type-related errors

General observation: higher precision ⇒ faster analysis
The eval of JavaScript

• \texttt{eval}(S)
  – parse the string $S$ as JavaScript code, then execute it

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

```javascript
function _var_exists(name) {
    try {
        eval('var foo = ' + name + ';');
    } catch (e) {
        return false;
    }
    return true;
}

var Namespace = {
create: function(path) {
    var container = null;
    while (path.match(/^(\w+)/(?\.)\$/)) {
        var key = RegExp.$1;
        path = path.replace(/^(\w+)/(?\.)\$/m, "");
        if (!container) {
            if (!_var_exists(key)) {
                eval('window.' + key + ' = {};');
                eval('container = ' + key + ';');
            } else {
                if (!container[key]) container[key] = {};
                container = container[key];
            }
        } else {
            if (!container[key]) container[key] = {};
            container = container[key];
        }
    }
}

window[key] = {}; 
```

(returns `name` in window; also avoids conflicts if `name` is "name" or "foo")
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]
A simple example

```
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
  
  eval(y + "(" + i + ")") \implies foo(0)

  (the dataflow analysis can now proceed into foo)

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

- ... (would not work if i could be any string)
A real-world example

```javascript
get_cookie = function (name) {
    var ca = document.cookie.split(';');
    for (var i = 0, l = ca.length; i < l; i++) {
        if (eval("ca[i].match(/\b" + name + "=/)"))
            return decodeURIComponent(car[i].split('=')[1]);
    }
    return ');
}
get_cookie('clicky_olark')
get_cookie('no_tracky')
get_cookie('_jsuid')
```

**TAJS tells us that name is one of these three strings!**

```javascript
for (var i = 0, l = ca.length; i < l; i++) {
    if (eval("ca[i].match(/\b" + name + "=/)"))
        return decodeURIComponent(car[i].split('=')[1]);
}
return '';
```

```javascript
eval("ca[i].match(/\b" + name + "=/)"")
```
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?
jQuery
write less, do more.
<table>
<thead>
<tr>
<th>JavaScript Library</th>
<th>Absolute Usage Percentage</th>
<th>Market Share</th>
</tr>
</thead>
<tbody>
<tr>
<td>JQuery</td>
<td>65.6%</td>
<td>95.5%</td>
</tr>
<tr>
<td>Modernizr</td>
<td>9.0%</td>
<td>13.1%</td>
</tr>
<tr>
<td>Bootstrap</td>
<td>8.8%</td>
<td>12.9%</td>
</tr>
<tr>
<td>MooTools</td>
<td>3.9%</td>
<td>5.7%</td>
</tr>
<tr>
<td>ASP.NET Ajax</td>
<td>2.3%</td>
<td>3.4%</td>
</tr>
<tr>
<td>Prototype</td>
<td>2.2%</td>
<td>3.3%</td>
</tr>
<tr>
<td>Script.aculo.us</td>
<td>1.8%</td>
<td>2.7%</td>
</tr>
<tr>
<td>YUI Library</td>
<td>0.6%</td>
<td>0.9%</td>
</tr>
<tr>
<td>Shadowbox</td>
<td>0.5%</td>
<td>0.7%</td>
</tr>
<tr>
<td>Spry</td>
<td>0.4%</td>
<td>0.6%</td>
</tr>
<tr>
<td>Underscore</td>
<td>0.3%</td>
<td>0.4%</td>
</tr>
<tr>
<td>AngularJS</td>
<td>0.2%</td>
<td>0.3%</td>
</tr>
<tr>
<td>Backbone</td>
<td>0.1%</td>
<td>0.2%</td>
</tr>
<tr>
<td>Dojo</td>
<td>0.1%</td>
<td>0.2%</td>
</tr>
<tr>
<td>Knockout</td>
<td>0.1%</td>
<td>0.1%</td>
</tr>
<tr>
<td>Ext JS</td>
<td>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
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>

Lines executed when the library initializes itself after loading.
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
A dynamic property read: \( x[y] \)

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

**Example of imprecision that explodes**
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) {
        ...
    } 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)

- Perhaps not impressive, but progress 😊

[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]
with **parameter sensitivity**, these become constants

**constant propagation...**

**branch pruning** logically eliminates several branches

**specializing** on `i` effectively unrolls the loop

**context-sensitive heap abstraction** keeps the `ajaxStart`, `ajaxStop`, etc. functions separate
Observations

• The analysis is essentially executing the critical library code concretely!
  – but allowing abstract values, e.g. from the application code
• A kind of “static determinacy analysis”

Experiments show that
• all the tricks must be enabled to get positive results
• unhandled cases are likely not due to too much precision
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://casa.au.dk/