Toward Static Analysis of Real-World JavaScript Code

- or, The Curse of jQuery

Anders Møller

Center for Advanced Software Analysis

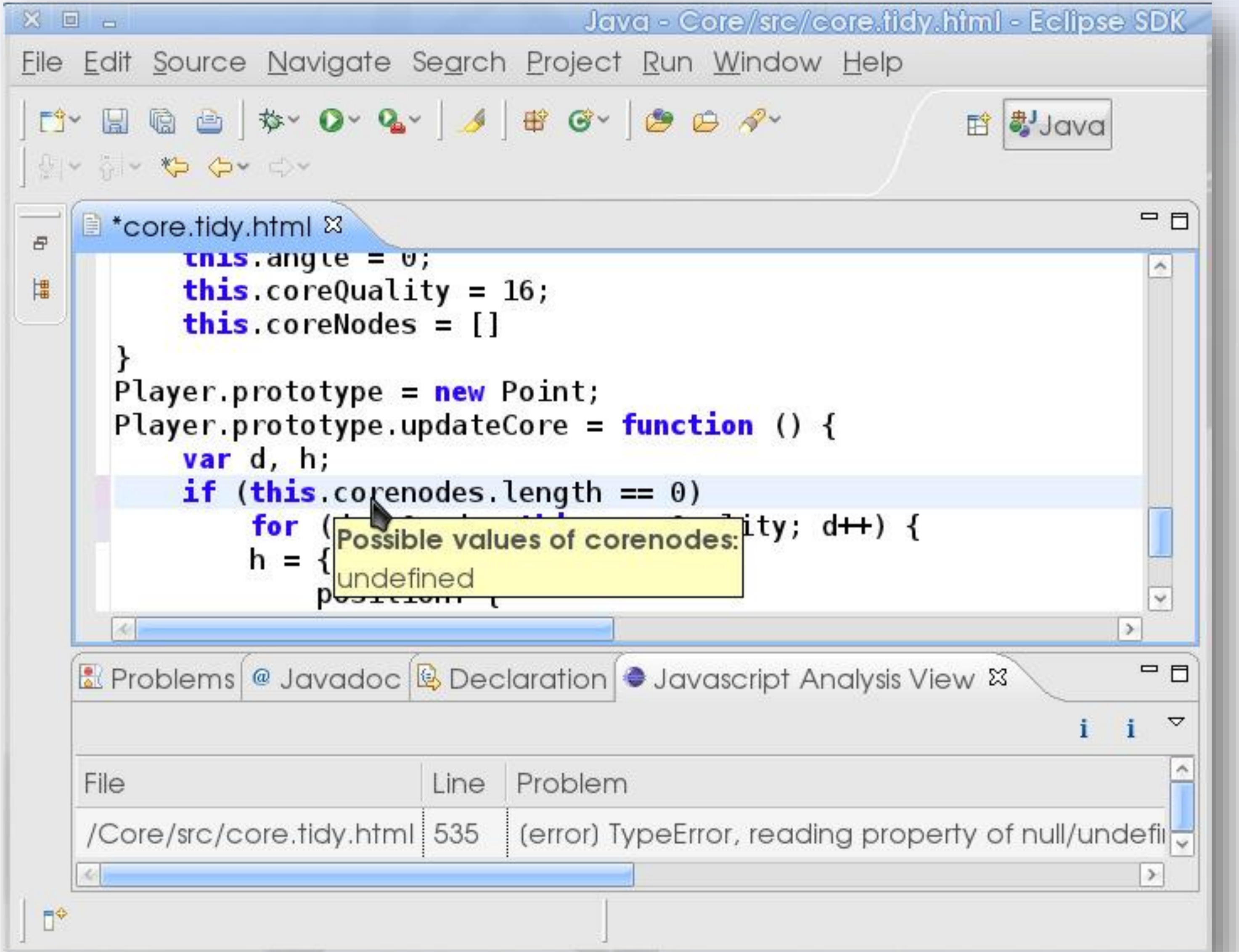
Aarhus University, Denmark

March 23 2015

JavaScript needs static analysis

- JavaScript is now everywhere
- Testing is still the only 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*")
  - code completion
  - optimization

TAJS in Eclipse



```
    this.angle = 0;
    this.coreQuality = 16;
    this.coreNodes = []
}
Player.prototype = new Point;
Player.prototype.updateCore = function () {
    var d, h;
    if (this.corenodes.length == 0)
        for (                       ity; d++) {
        h = {
            position: {
```

Possible values of corenodes:
undefined

File	Line	Problem
/Core/src/core.tidy.html	535	(error) TypeError, reading property of null/undefi

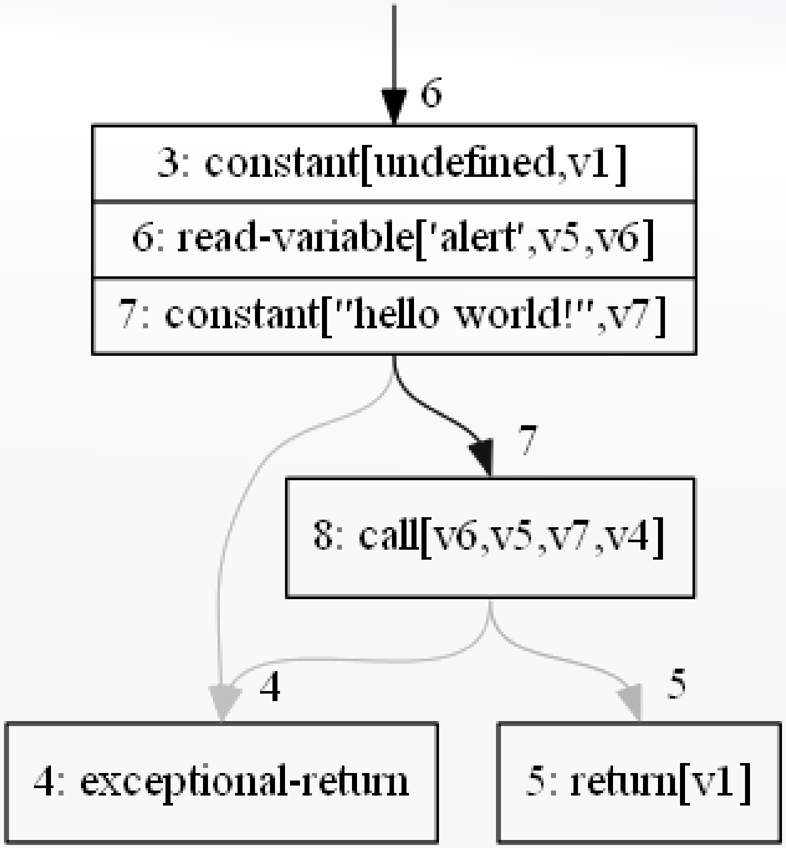
The TAJS approach

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

- 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)

Control flow graphs



- Convenient intermediate representation of JavaScript programs

- ***Nodes*** describe primitive instructions
- ***Edges*** describe *intra*-procedural control-flow

The dataflow lattice (simplified!)

- For each program point **N** an
  the analysis maintains an ab
  N × C → State
- Each abstract state provides
  for each abstract object **L** a
  State = L × P → Value
- Each abstract value describ
  primitive values:
  Value = $\mathcal{P}$(L) × Bool × Str × Num …
- *Details refined through trial-and-error...*

<u>Key ideas:</u>

- 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

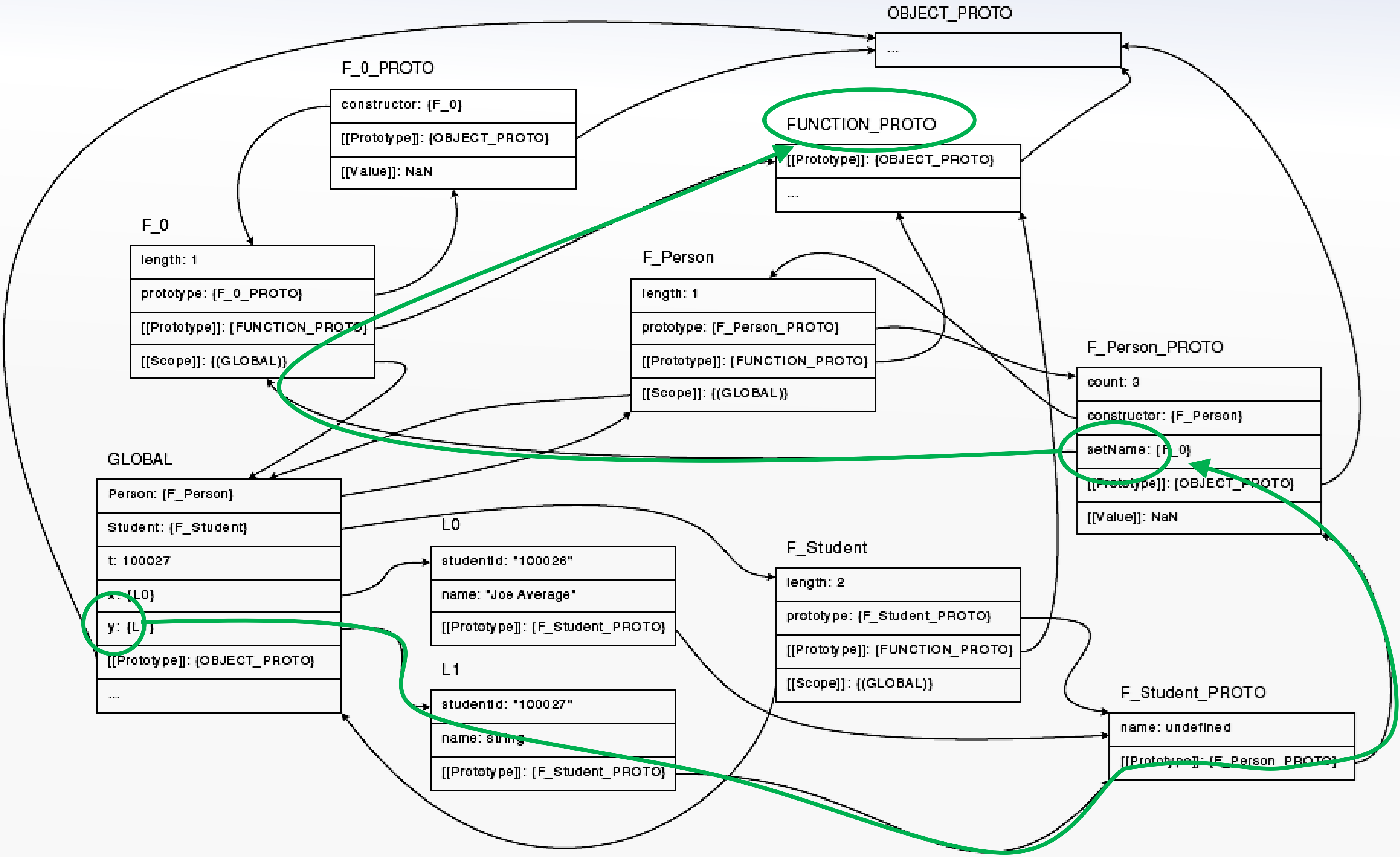
A tiny example...

```
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)



OBJECT_PROTO
...

F_0_PROTO
constructor: {F_0}
[[Prototype]]: {OBJECT_PROTO}
[[Value]]: NaN

FUNCTION_PROTO
[[Prototype]]: {OBJECT_PROTO}
...

F_0
length: 1
prototype: {F_0_PROTO}
[[Prototype]]: {FUNCTION_PROTO}
[[Scope]]: {(GLOBAL)}

F_Person
length: 1
prototype: {F_Person_PROTO}
[[Prototype]]: {FUNCTION_PROTO}
[[Scope]]: {(GLOBAL)}

F_Person_PROTO
count: 3
constructor: {F_Person}
setName: {F_0}
[[Prototype]]: {OBJECT_PROTO}
[[Value]]: NaN

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: string
[[Prototype]]: {F_Student_PROTO}

F_Student
length: 2
prototype: {F_Student_PROTO}
[[Prototype]]: {FUNCTION_PROTO}
[[Scope]]: {(GLOBAL)}

F_Student_PROTO
name: undefined
[[Prototype]]: {F_Person_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

around 250 abstract objects
with 500 properties
and 200 functions…

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

Ingredients in a static analyzer for JavaScript applications

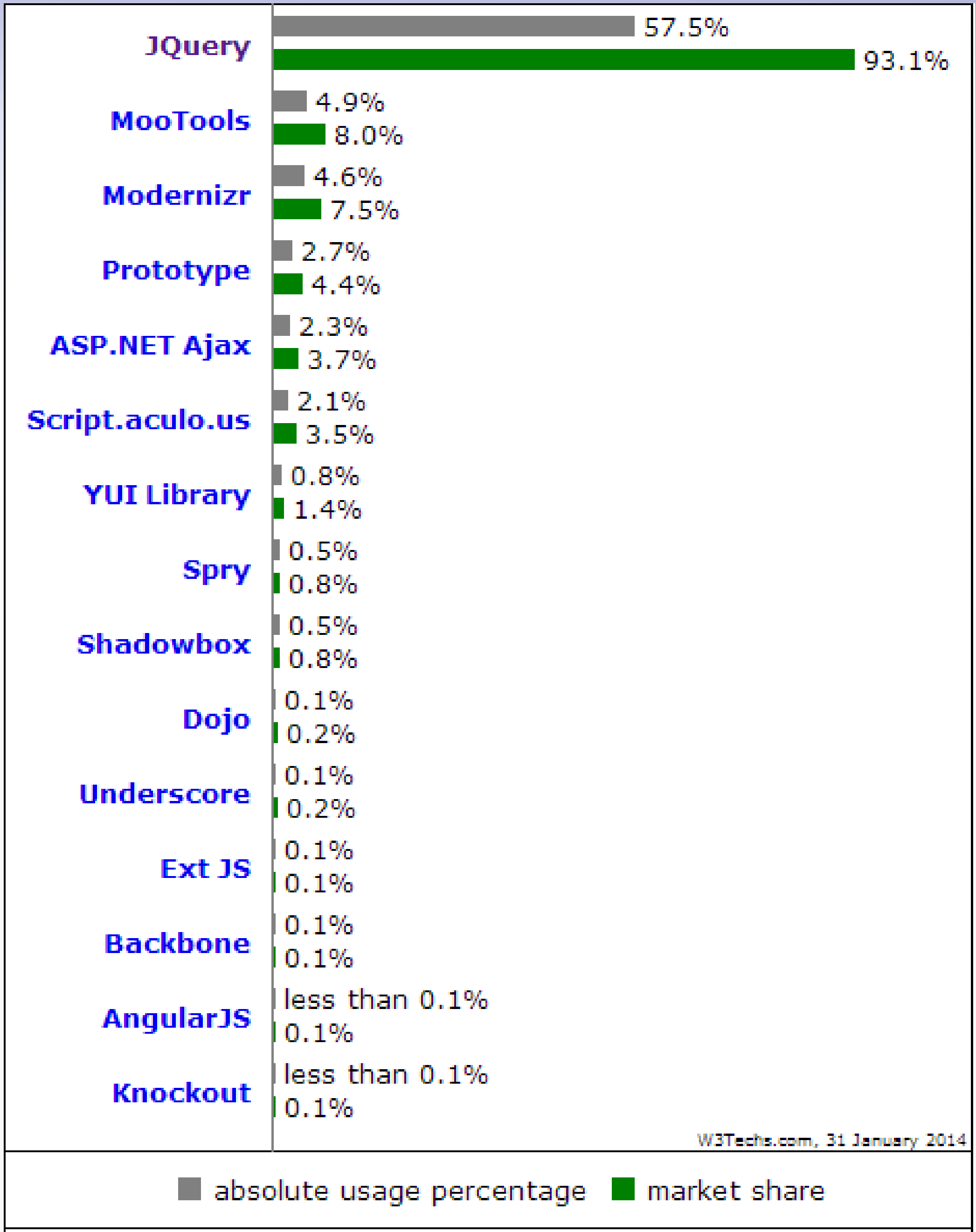
We need to model

- ☑ the language semantics
- ☑ the standard library
- ☑ the browser API (the HTML DOM, the event system, etc.)

Mission complete?

M[illegible] c[illegible]ple[illegible]?

write less, do more.



W3Techs.com, 31 January 2014

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

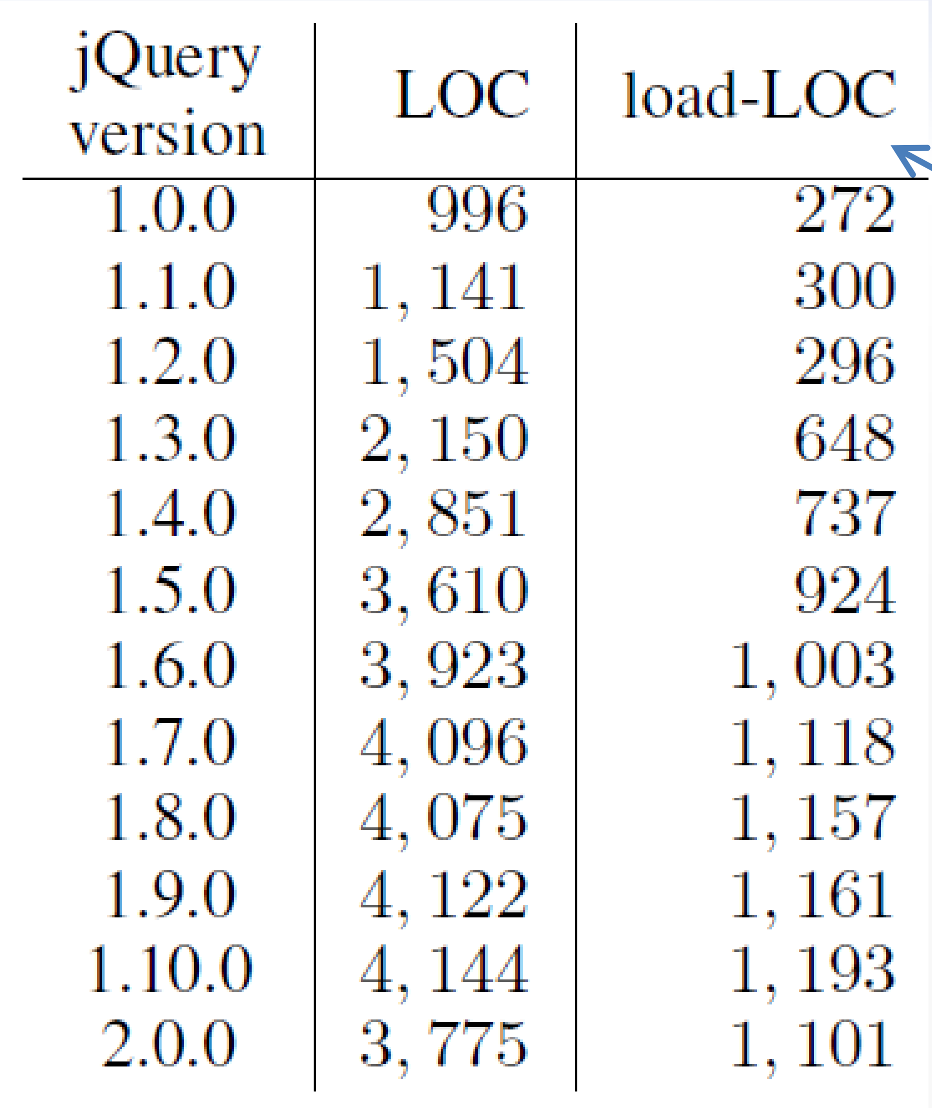
An appetizer

Which code fragment do you prefer?

```
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;
  }
}
```

```
var checkedValue = $('[name="someRadioGroup"]:checked').val();
```

Investigating the beast



jQuery version	LOC	load-LOC
1.0.0	996	272
1.1.0	1, 141	300
1.2.0	1, 504	296
1.3.0	2, 150	648
1.4.0	2, 851	737
1.5.0	3, 610	924
1.6.0	3, 923	1, 003
1.7.0	4, 096	1, 118
1.8.0	4, 075	1, 157
1.9.0	4, 122	1, 161
1.10.0	4, 144	1, 193
2.0.0	3, 775	1, 101

lines executed when the library initializes itself after loading

WALA

T. J. WATSON LIBRARIES FOR ANALYSIS

[Schäfer, Sridharan, Dolby, Tip. *Dynamic Determinacy Analysis*, PLDI'13]

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: `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:

```
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:

```
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;
}
```

Lots of

- **overloading**
- **reflection**
- **callbacks**

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**

NEW

[Andreasen and Møller, OOPSLA’14]

```
each: function (obj, callback, args) {
  var name, i = 0, length = obj.length,
  isObj = length === undefined || jQuery.isFuncti
  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;
}
```

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

The technical side...

- The analysis maintains an abstract state for each program point **N** and call context **C**:

  $$\mathbf{N} \times \mathbf{C} \to \mathbf{State}$$

- Old TAJS:

  $\mathbf{C} = \mathcal{P}(\mathbf{L})$ (object sensitivity)

  $\mathbf{L} = \mathbf{N}$ (**L**: abstract memory locations)

- New TAJS:

  parameter sensitivity
  (**A**: selected parameters)

  loop specialization
  (**B**: selected local variables)

  $\mathbf{C} = \mathcal{P}(\mathbf{L}) \times (\mathbf{A} \rightharpoonup \mathbf{Value}) \times (\mathbf{B} \rightharpoonup \mathbf{Value})$

  $\mathbf{L} = \mathbf{N} \times \mathbf{C}$

  context-sensitive heap abstraction

Conclusion

- Statically analyzing real-world JavaScript web applications must handle jQuery
- Progress – but far from a full solution...
  - our approach: boost precision, with inspiration from well-known ideas from static analysis
- How far can we push this?
  - nontrivial applications?
  - other libraries? plugins?

CENTER FOR ADVANCED SOFTWARE ANALYSIS

http://casa.au.dk/

European Research Council
Established by the European Commission