Program Analysis for JavaScript
– Challenges and Techniques

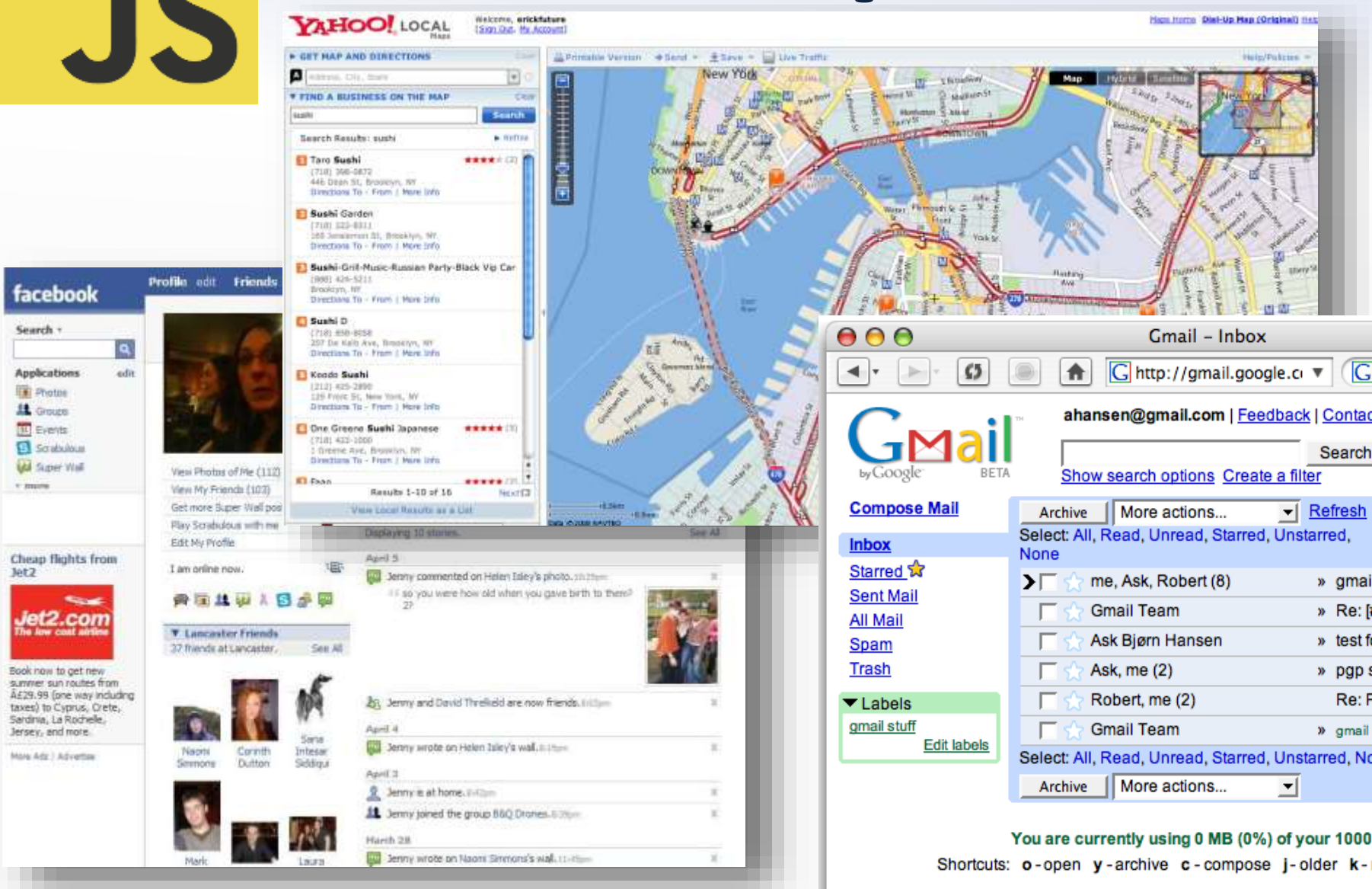
Anders Møller

Center for Advanced Software Analysis

Aarhus University

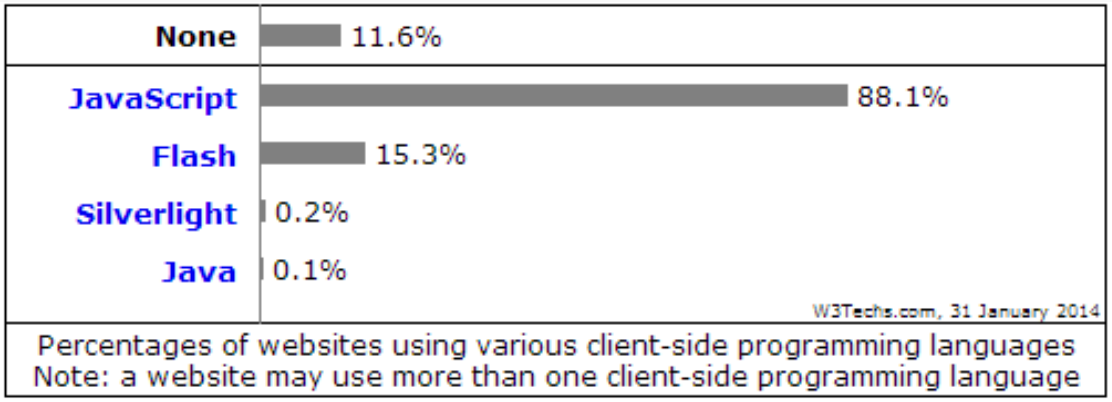
Joint work with
Esben Andreasen, Simon Holm Jensen, Peter A. Jonsson, Magnus Madsen, and Peter Thiemann

JavaScript


YAHOO! LOCAL
GET MAP AND DIRECTIONS
FIND A BUSINESS ON THE MAP
facebook
Profile
Friends
Gmail – Inbox
Gmail by Google BETA
Compose Mail
Inbox
Starred
Sent Mail
All Mail
Spam
Trash
Labels
gmail stuff
Edit labels
Search Mail
Show search options
Create a filter
Archive
More actions...
Refresh
You are currently using 0 MB (0%) of your 1000 MB.
Shortcuts: o - open y - archive c - compose j - older k - newer

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



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

TAJS Type Analysis for JavaScript

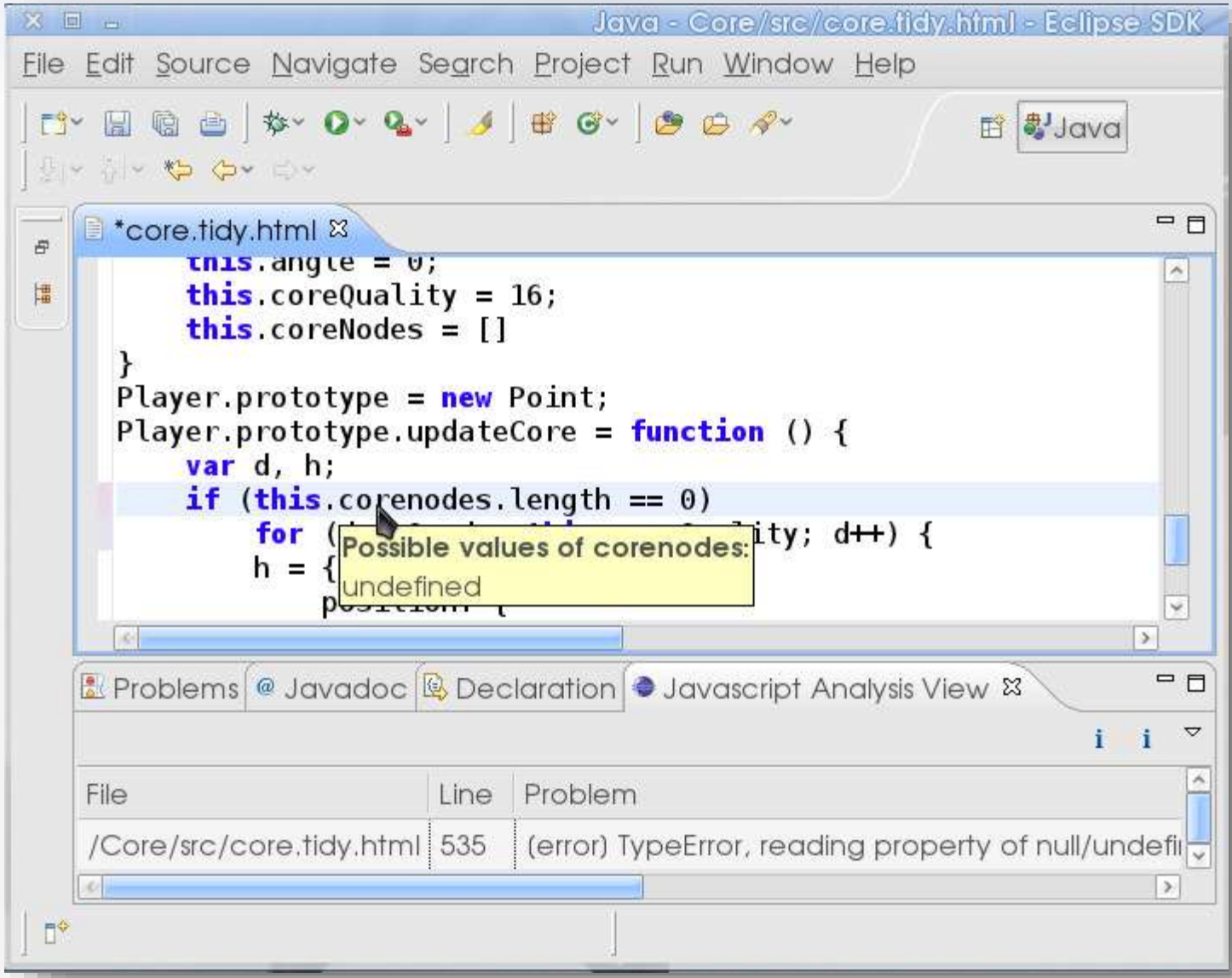
The goal:

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?

coercion?

standard library?

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)

The dataflow lattice (simplified!)

- The analysis maintains an abstrac
  each program point **N** and call co
  N × C → State
- Each abstract state provides an a
  for each abstract object **L** and p
  State = **L × P → Value**
- Each abstract value describes pointers and primitive values:
  Value = $\mathcal{P}$**(L) × Bool × Str × Num …**
- *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

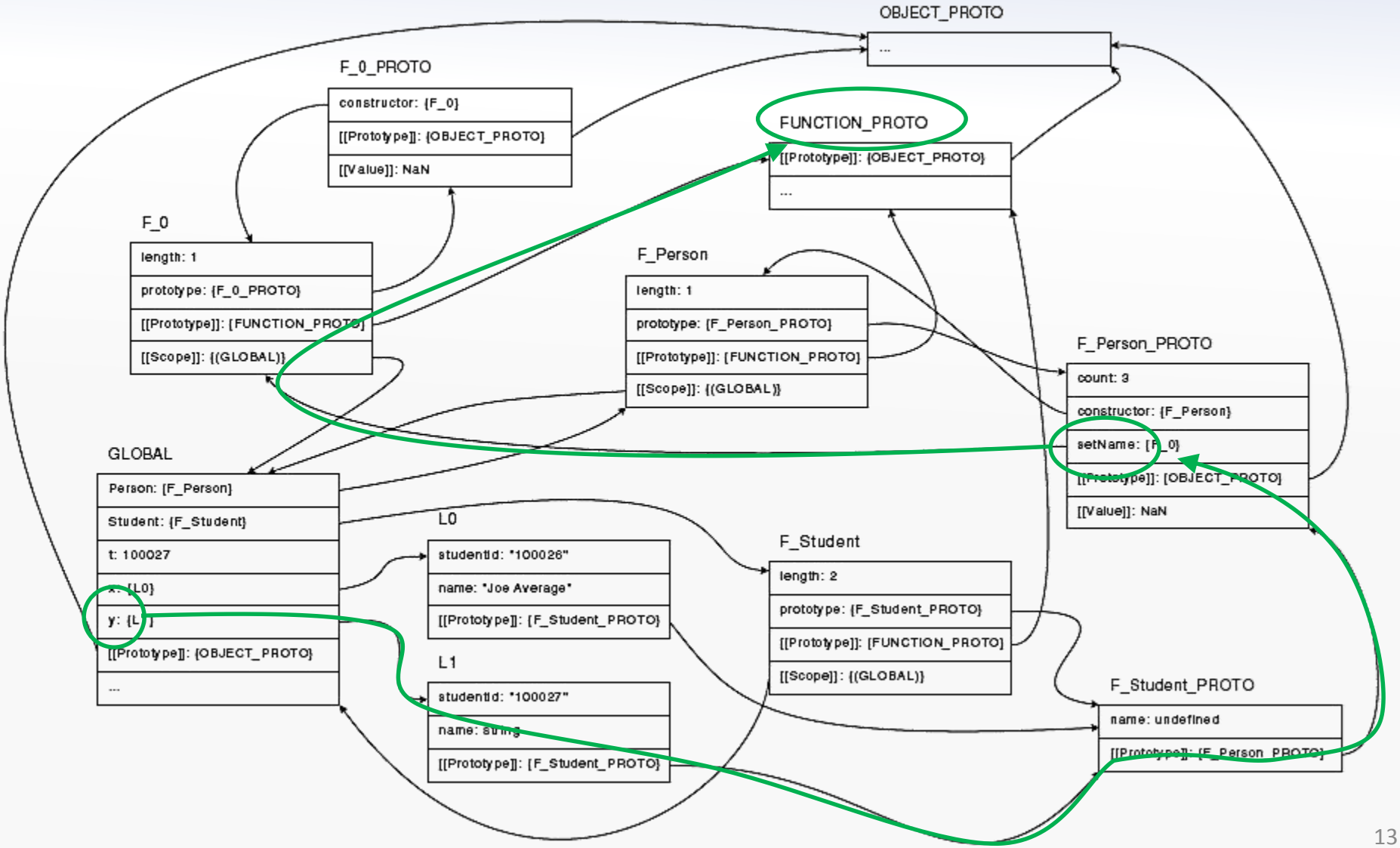
A tiny example...

```
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)



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]

Eval in JavaScript

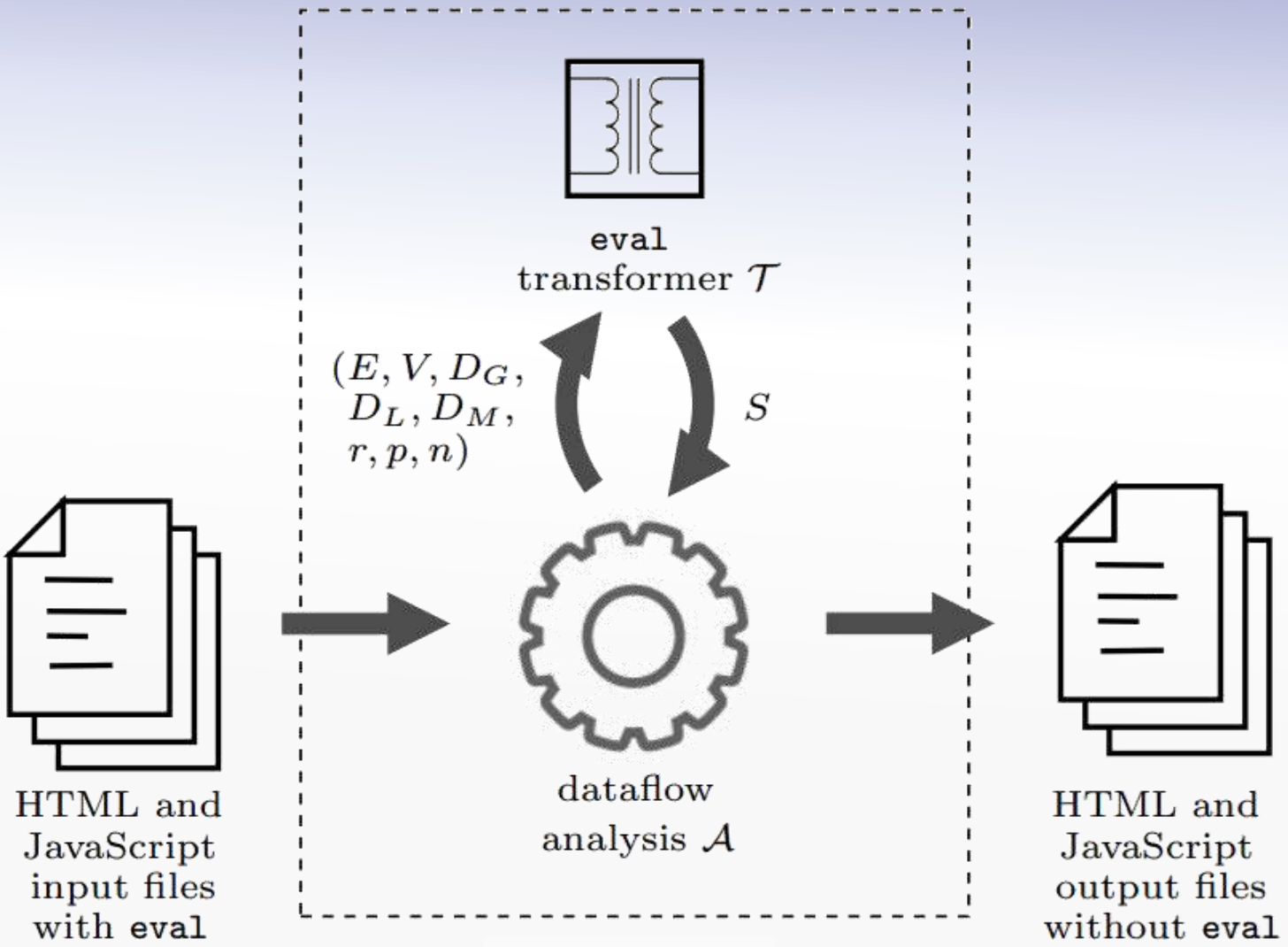
- `eval(`*s*`)`
  - parse the string *s* as JavaScript code, then execute it
- Challenging for JavaScript static analysis
  - the string is dynamically generated
  - the generated code may have side-effects
  - and JavaScript has poor encapsulation mechanisms

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

```
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 + ")")` ⇨ `foo(0)`

  (the dataflow analysis can now proceed into `foo`)

- iteration 2: `y` is `"foo"`, `i` is *AnyNumber*

  `eval(y + "(" + i + ")")` ⇨ `foo(i)`

- …

(would not work if `i` could be any string)

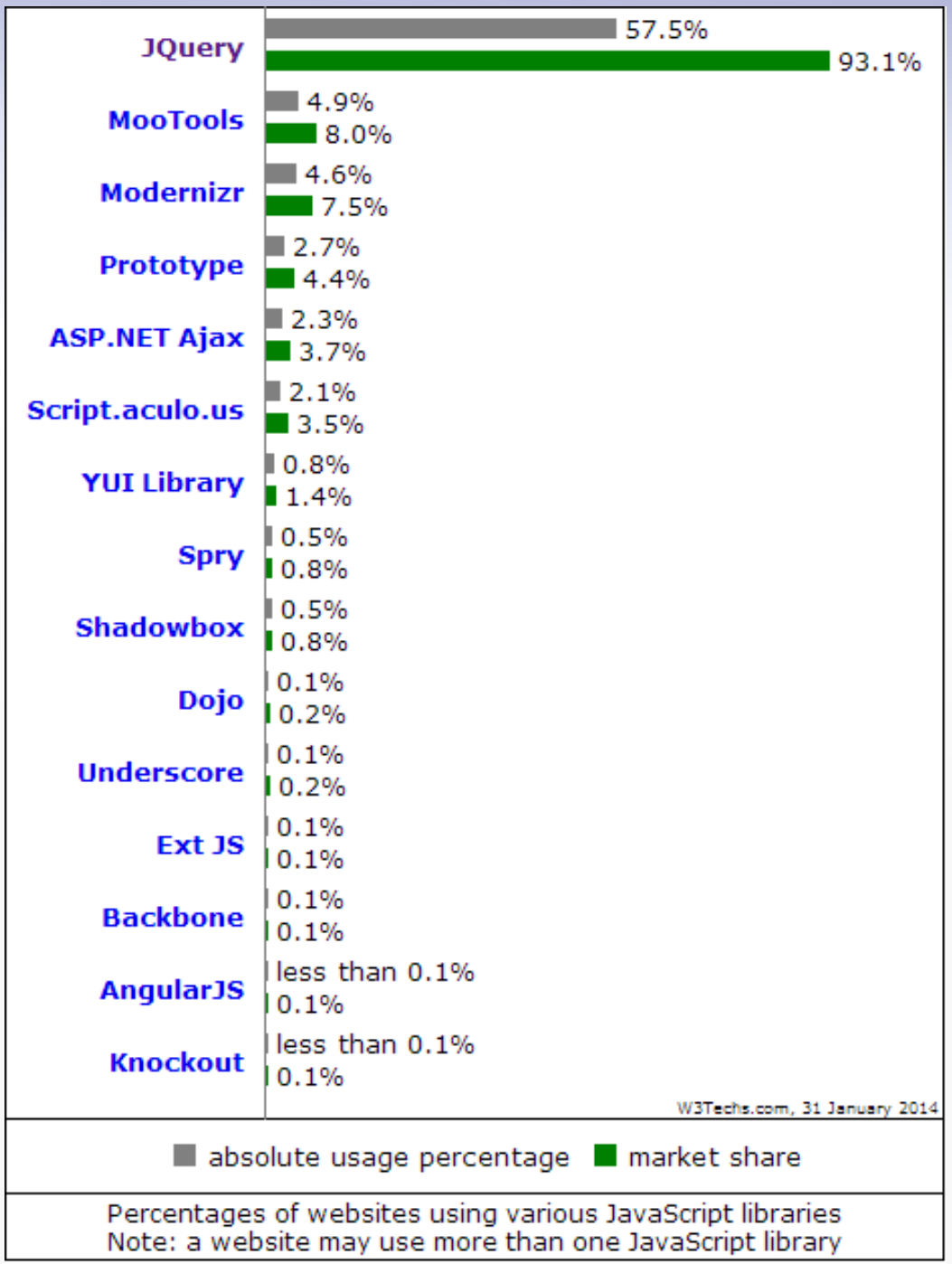
Ingredients in a static analyzer for JavaScript applications

We need to model

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

Mission complete?


jQuery
write less, do more.


JQuery
57.5%
93.1%
MooTools
4.9%
8.0%
Modernizr
4.6%
7.5%
Prototype
2.7%
4.4%
ASP.NET Ajax
2.3%
3.7%
Script.aculo.us
2.1%
3.5%
YUI Library
0.8%
1.4%
Spry
0.5%
0.8%
Shadowbox
0.5%
0.8%
Dojo
0.1%
0.2%
Underscore
0.1%
0.2%
Ext JS
0.1%
0.1%
Backbone
0.1%
0.1%
AngularJS
less than 0.1%
0.1%
Knockout
less than 0.1%
0.1%
W3Techs.com, 31 January 2014
absolute usage percentage
market share
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**

[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**:

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

- 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

- 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