Static Analysis for JavaScript

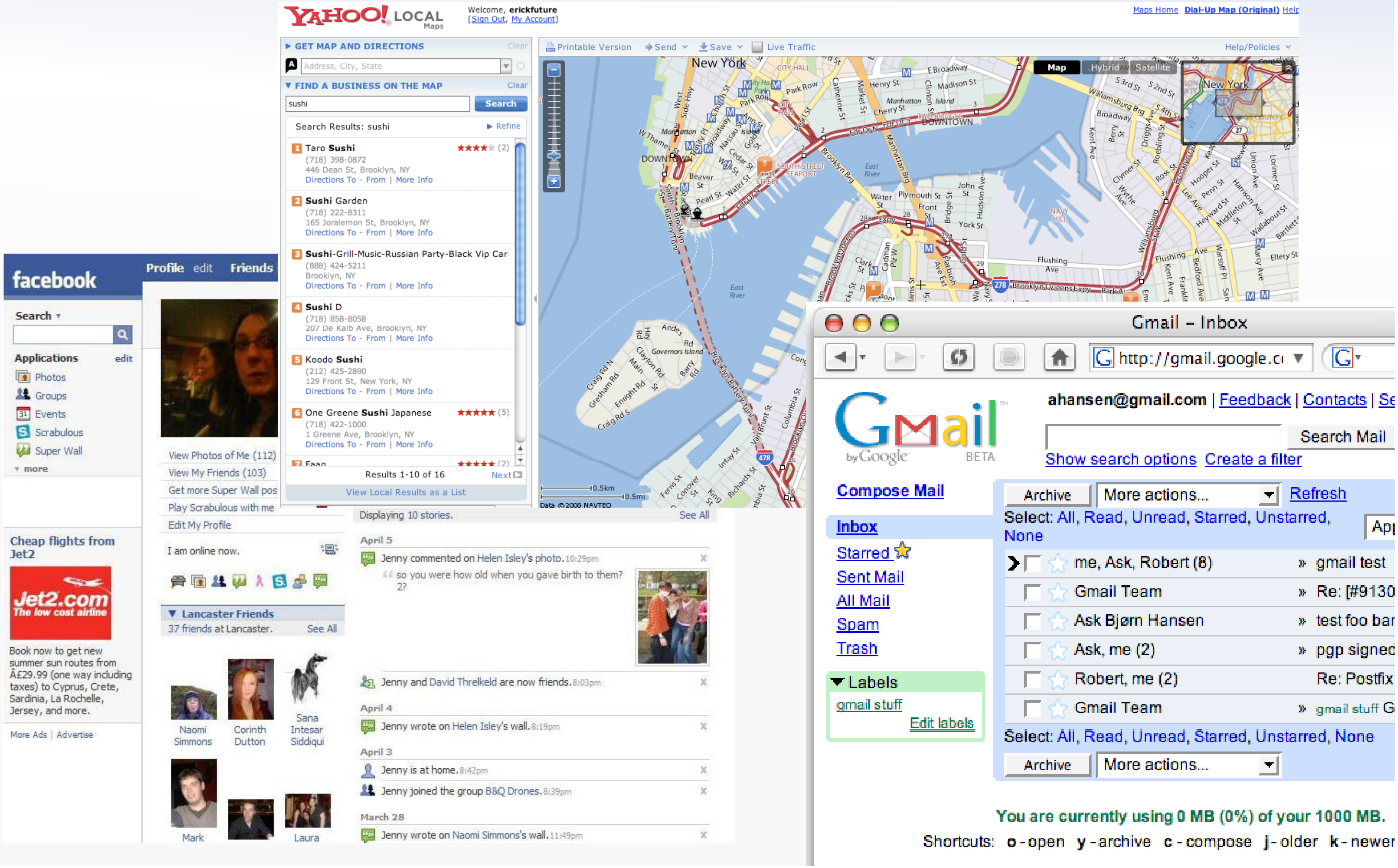
Anders Møller

Center for Advanced Software Analysis

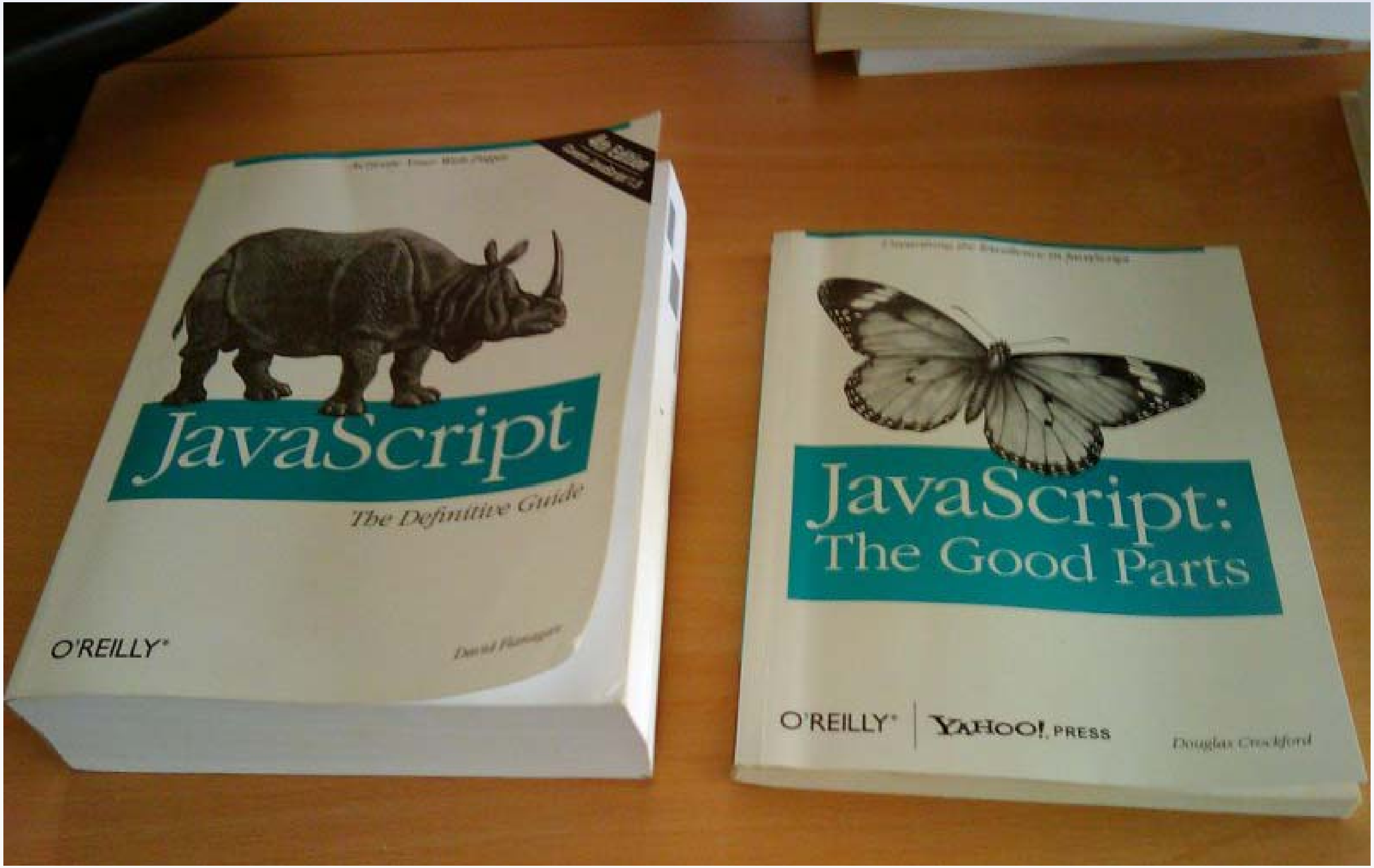
Aarhus University

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

JavaScript: the lingua franca of Web 2.0



The good parts of JavaScript?



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

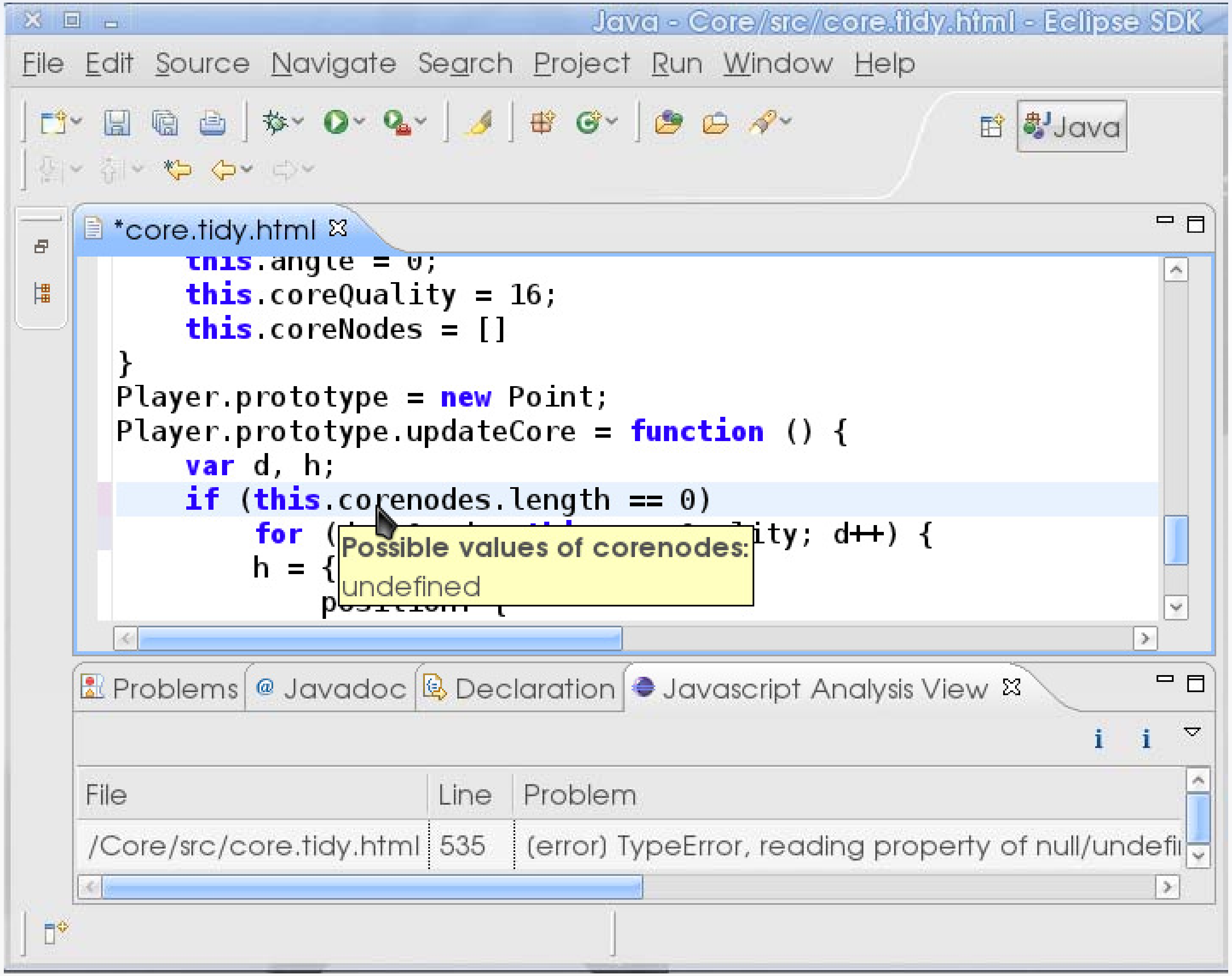
NO STATIC CLASS HIERARCHIES

TAJS: Type Analysis for JavaScript

- Catch type-related errors using **program analysis**

- Support **the full language** (including `eval`)
- 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…

Flow of control and data can be subtle

```
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

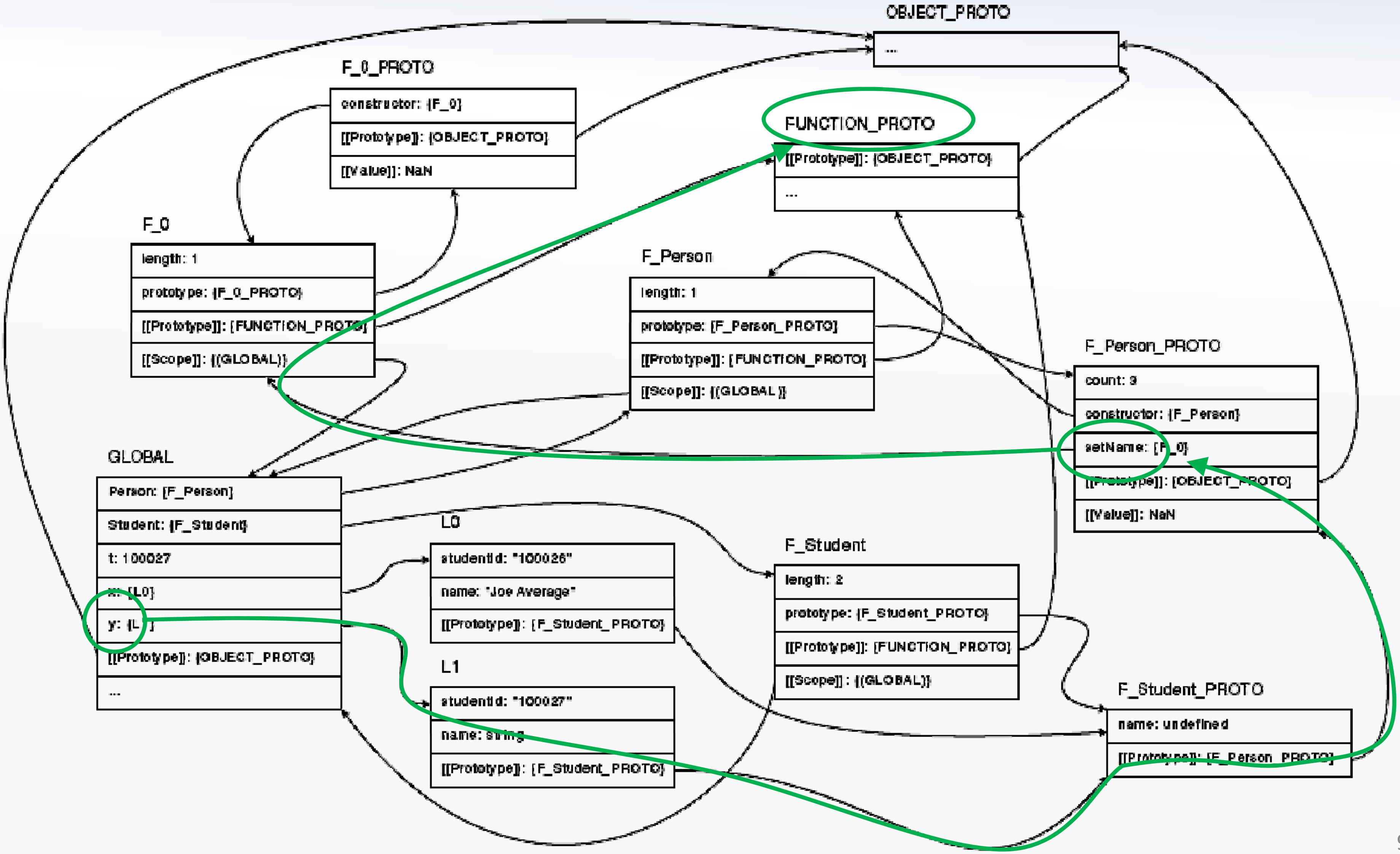
declares a shared method named setName

declares a “sub-class” named Student

creates two Student objects…

does `y` have a `setName` method at this program point?

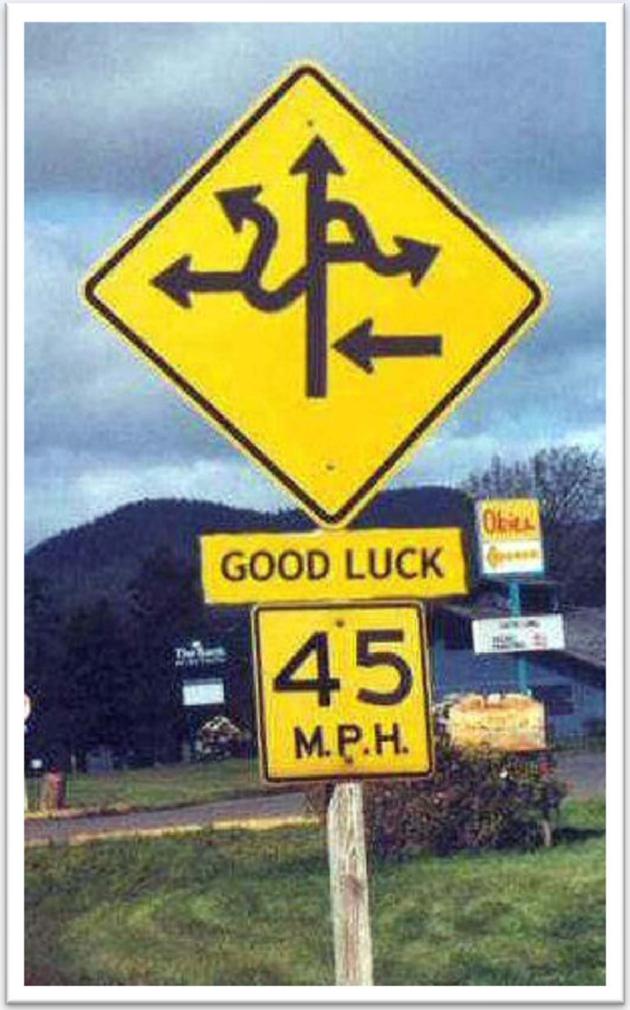
An abstract state (as produced by TAJS)



Which way to go?

We want

- heap analysis
- flow-sensitivity
- constant propagation
- on-the-fly call graph construction
- soundness



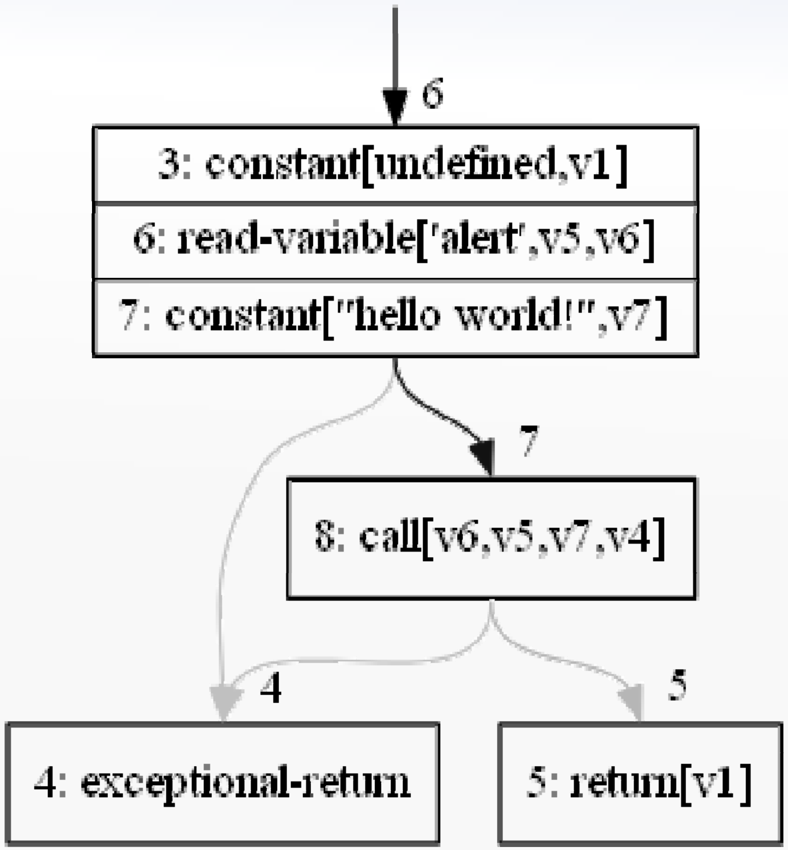
Our approach

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

- Abstract interpretation (dataflow analysis) using the monotone framework
  [Kam & Ullman '77]

- The recipe:
  1. construct a **control flow graph** for 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 representation of JavaScript programs

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

Analysis lattice

the analysis lattice

abstract states

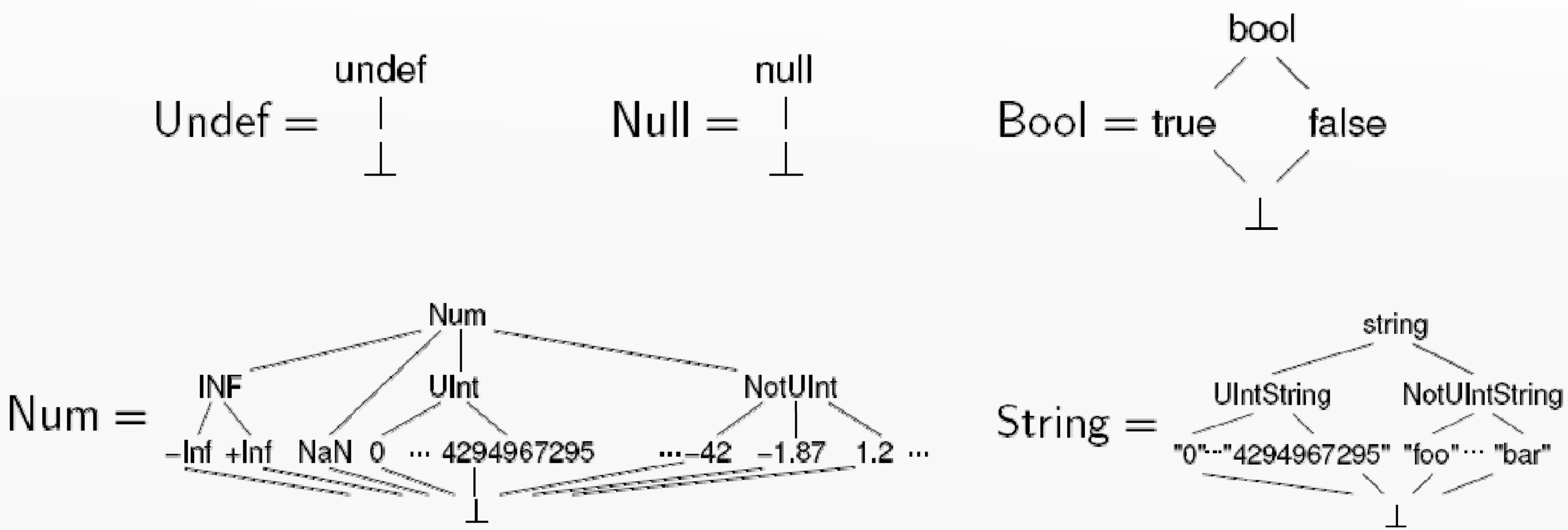
abstract objects

abstract values

Abstract values

object labels
(allocation sites)

$$\mathsf{Value} = \mathsf{Undef} \times \mathsf{Null} \times \mathsf{Bool} \times \mathsf{Num} \times \mathsf{String} \times \mathcal{P}(L)$$



Example: $(\perp, \text{null}, \text{true}, 42.0, \perp, \{\ell_7, \ell_9\})$

Abstract objects

property names including [[Prototype]]

$$\mathsf{Obj} = (P \hookrightarrow \mathsf{Value} \times \mathsf{Absent} \times \mathsf{Attributes} \times \mathsf{Modified}) \times \mathcal{P}(\mathsf{ScopeChain})$$

describes the [[Scope]] property

$$\mathsf{Absent} = \begin{array}{c} \text{absent} \\ | \\ \bot \end{array} \qquad \mathsf{Modified} = \begin{array}{c} \text{modified} \\ | \\ \bot \end{array}$$

$$\mathsf{Attributes} = \mathsf{ReadOnly} \times \mathsf{DontDelete} \times \mathsf{DontEnum}$$

$$\mathsf{ReadOnly} = \begin{array}{ccc} & \top & \\ / & & \backslash \\ \text{RO} & & \text{notRO} \\ \backslash & & / \\ & \bot & \end{array} \quad \mathsf{DontDelete} = \begin{array}{ccc} & \top & \\ / & & \backslash \\ \text{DD} & & \text{notDD} \\ \backslash & & / \\ & \bot & \end{array} \quad \mathsf{DontEnum} = \begin{array}{ccc} & \top & \\ / & & \backslash \\ \text{DE} & & \text{notDE} \\ \backslash & & / \\ & \bot & \end{array}$$

Abstract states

heap

$$\text{State} = (L \hookrightarrow \text{Obj}) \times \text{Stack} \times \mathcal{P}(L) \times \mathcal{P}(L)$$

(explained later, maybe...)

temporary variables

the current activation record

stack-reachable objects

$$\text{Stack} = (T \to \text{Value}) \times \mathcal{P}(\text{ExecutionContext}) \times \mathcal{P}(L)$$

$$\text{ExecutionContext} = \text{ScopeChain} \times L \times L$$

$$\text{ScopeChain} = L^*$$

the `this` object

the variable object, for variable declarations

for resolving variables

The analysis dataflow lattice

$$\mathsf{AnalysisLattice} = (C \times N \;\rightarrow\; \mathsf{State}) \times \mathsf{CallGraph}$$

contexts

(for context sensitivity)

the flow graph nodes

$$\mathsf{CallGraph} = \mathcal{P}(C \times N \times C \times F)$$

caller context and node

callee context and node

Transfer functions

Example: **read-property** $x = y[p]$

1. Coerce y to objects
2. Coerce p to strings
3. Descend the object prototype chains (using the [[Prototype]] property) to find the relevant properties
4. Join the property values
5. Assign the result to x

Weak vs. strong updates

Consider **write-property** $x[p] = y$

- x may refer to many abstract objects (identified by their allocation sites)
- ...and each may represent many concrete objects
- So **write-property** must conservatively be modeled by *joining* y into the existing value of $x[p]$ (i.e. a *weak update*)
  - bad for precision!

```
x = {a:”foo”}
x[”a”] = 42
// is x[”a”] == 42 here?
```

- *Strong update* (overwriting instead of joining) is possible whenever
  - x refers to only one abstract object
  - ...which is known to represent only one concrete object

Recency abstraction

[Balakrishnan and Reps, SAS'06]

- For each allocation site ℓ
  maintain **two** abstract objects:
  - $\ell^{@}$ corresponds to the *most recently* allocated object originating from ℓ
  - ℓ^{*} older objects from ℓ
- $\ell^{@}$ always describes at most one concrete object and hence permits strong updating!
- To make this work, we just need some extra bookkeeping in the transfer functions

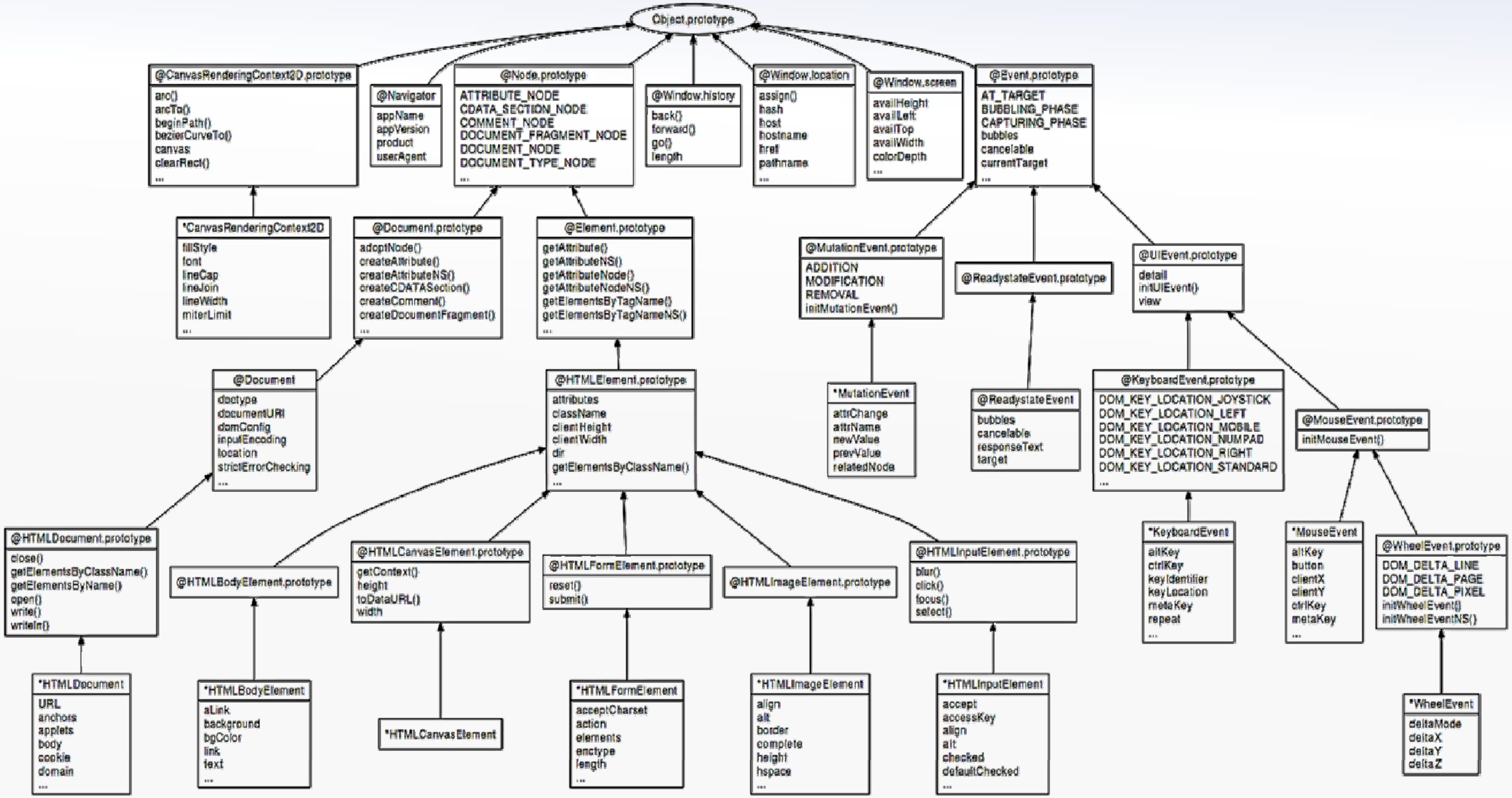
JavaScript web applications

- 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…

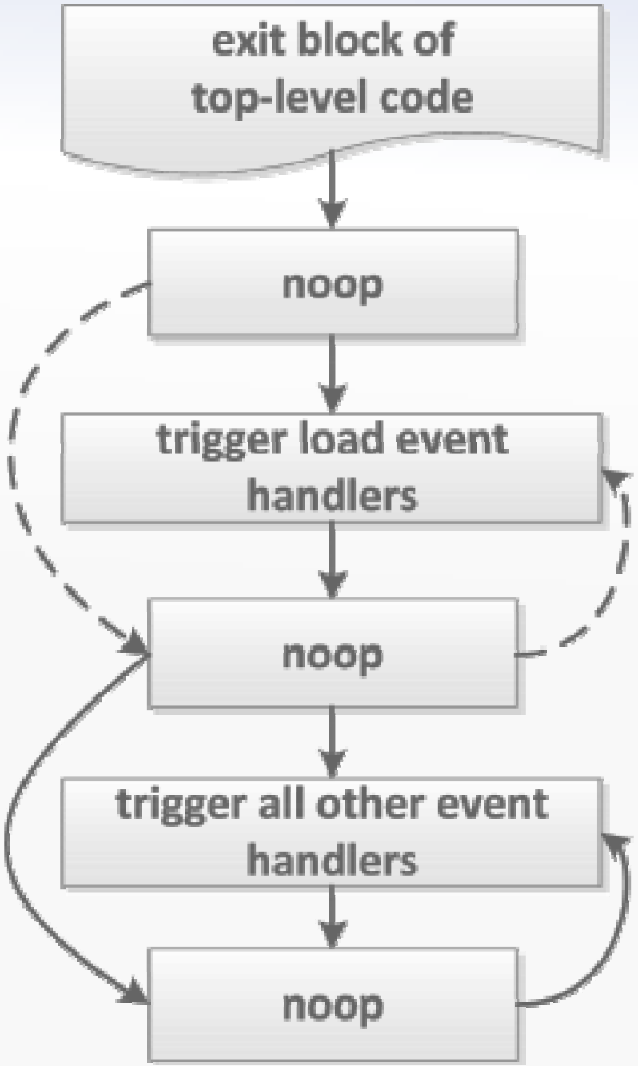
[Jensen, Madsen, and Møller, ESEC/FSE’11]

A small part of the HTML object hierarchy…



Modeling events

- Extend lattice and transfer functions to collect event handlers
- Trigger events non-deterministically
- Special treatment for *load* event handlers



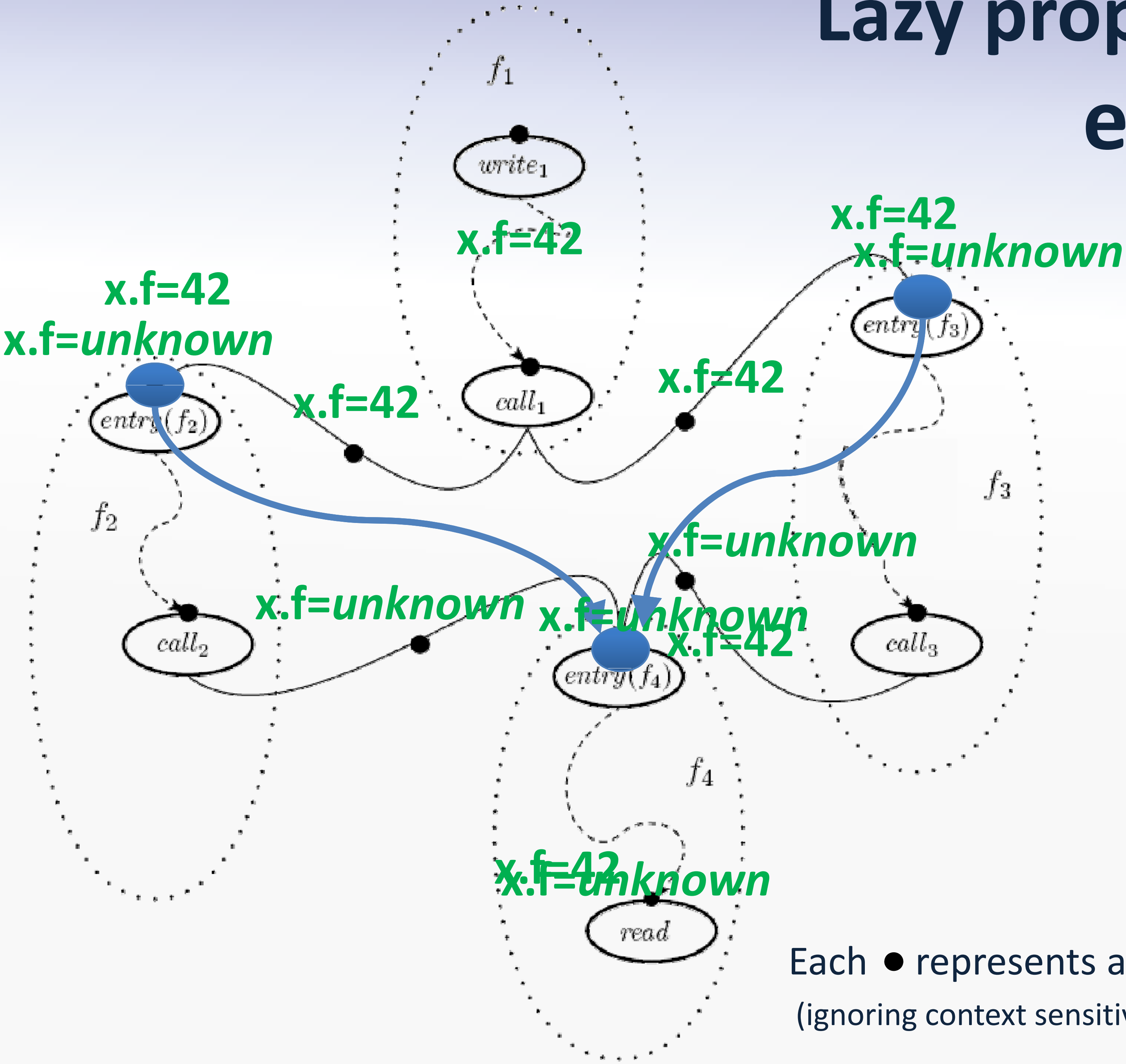
Lazy propagation

[Jensen, Møller, and Thiemann, SAS'10]

- Each abstract state is huge…
- Introducing *lazy propagation*:
  - When dataflow enters a function, assume initially that no object properties will be read by the function
  - Whenever an object property later is read, recover its value

  ⇒ **only relevant dataflow is propagated!**

Lazy propagation example



Each ● represents an abstract state

(ignoring context sensitivity)

Properties of lazy propagation

- Theoretical properties:
  - **Precision** is at least as good as before
  - **Soundness** (wrt. language semantics) is preserved
  - **Recovery** does not affect amortized complexity
- In practice:
  - **Much smaller abstract states!**
  - **Number of fixpoint iterations decreases**

Experiments

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

The analysis is able to show that

- 85-100% of all call sites are safe
- 80-100% of all property reads are safe
- most call sites are monomorphic
- most expressions have a unique type
- most spelling errors cause type-related errors

Eval in JavaScript

- `eval(`*s*`)`
  - parse the string *s* as JavaScript code, then execute it
- Challenging for **JavaScript static analysis**
  - the string may be dynamically generated
  - the generated code may have side-effects
  - and JavaScript has poor encapsulation mechanisms
- Existing analyses either ignore `eval` entirely or handle only the simplest cases

Eval in Practice

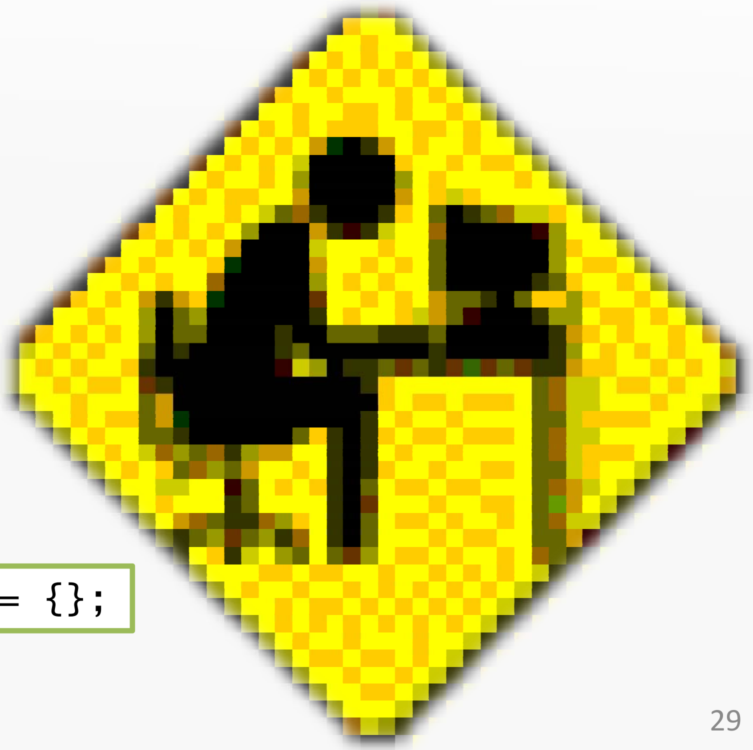
```
function _var_exists(name) {
  try {
    eval('var foo = ' + name + ';');
  } catch (e) {
    return false;
  }
  return true;
}

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

```
return name in window;
```

(if name is not "name" or "foo")

```
window[key] = {};
```



http://www.chromeexperiments.com/detail/canvas-cycle/

Eval is Evil (to Static Analysis)

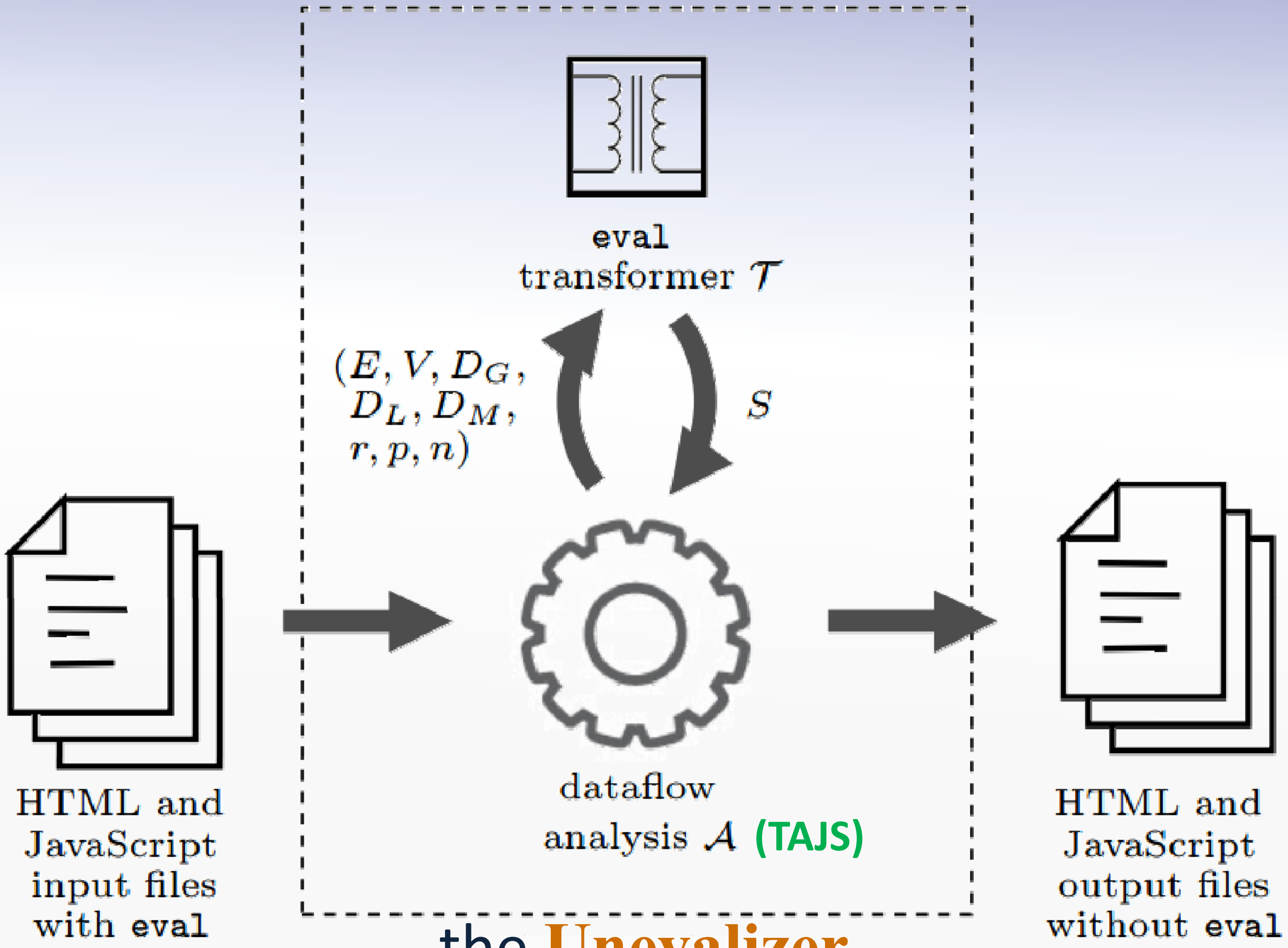
- ... but most uses of `eval` are not very complex
- So let's transform `eval` calls into other code!
- *How can we soundly make such transformations when we cannot analyze code with `eval`?*

Which came first?

Analysis or transformation

The Unevalizer to the Rescue!

- Removes calls to `eval` from input code
- Does not affect the behavior of the code
  - the resulting code is maybe not pretty
  - but it is analyzable!
- The idea:
  transform `eval` calls
  ***<u>during</u>* dataflow analysis**



the **Unevalizer**

Whenever the dataflow analysis 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)
  - …

More examples

- `eval("foo."+x)` ⇨ `foo[x]`
  if we know that x is a number or a string that is a valid identifier

- `eval("foo_"+x)` ⇨ `window["foo_"+x]`
  if we know that x is a string that consists of characters that are valid in identifiers, excluding the initial character, and that no local variables are named `foo_*`

... and one more example

```
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(ca[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!

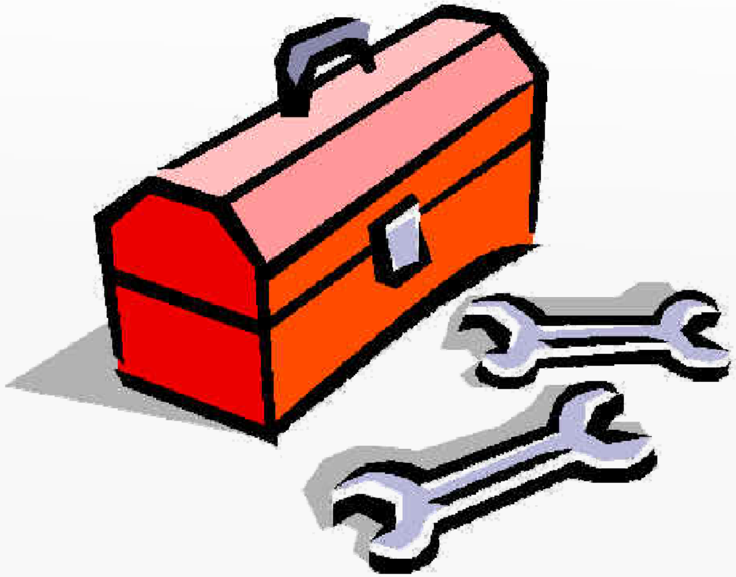
```
eval("ca[i].match(/\\b" + name + "=/)")
                      ⇩
name==="clicky_olark" ? ca[i].match(/\\bclicky_olark=/)
 : name==="no_tracky" ? ca[i].match(/\\bno_tracky=/)
                      : ca[i].match(/\\b_jsuid=/)
```

Remaining challenges in the TAJS project

- Improve **precision** further to eliminate false positives
- Improve **scalability** to handle JavaScript web applications that involve libraries, e.g. jQuery, MooTools, Dojo, ...
- Improve **IDE integration** (the Eclipse plug-in)
  – emphasize the most critical warnings
  – visualization of abstract states, call graphs, and inheritance hierarchies

Conclusion

- JavaScript programmers need better tools
- Static analysis can detect type-related errors
  - model of the standard library, the browser API, and the HTML DOM
  - recency abstraction
  - lazy propagation
  - rewrite calls to `eval` during analysis



CENTER FOR ADVANCED SOFTWARE ANALYSIS

http://cs.au.dk/CASA