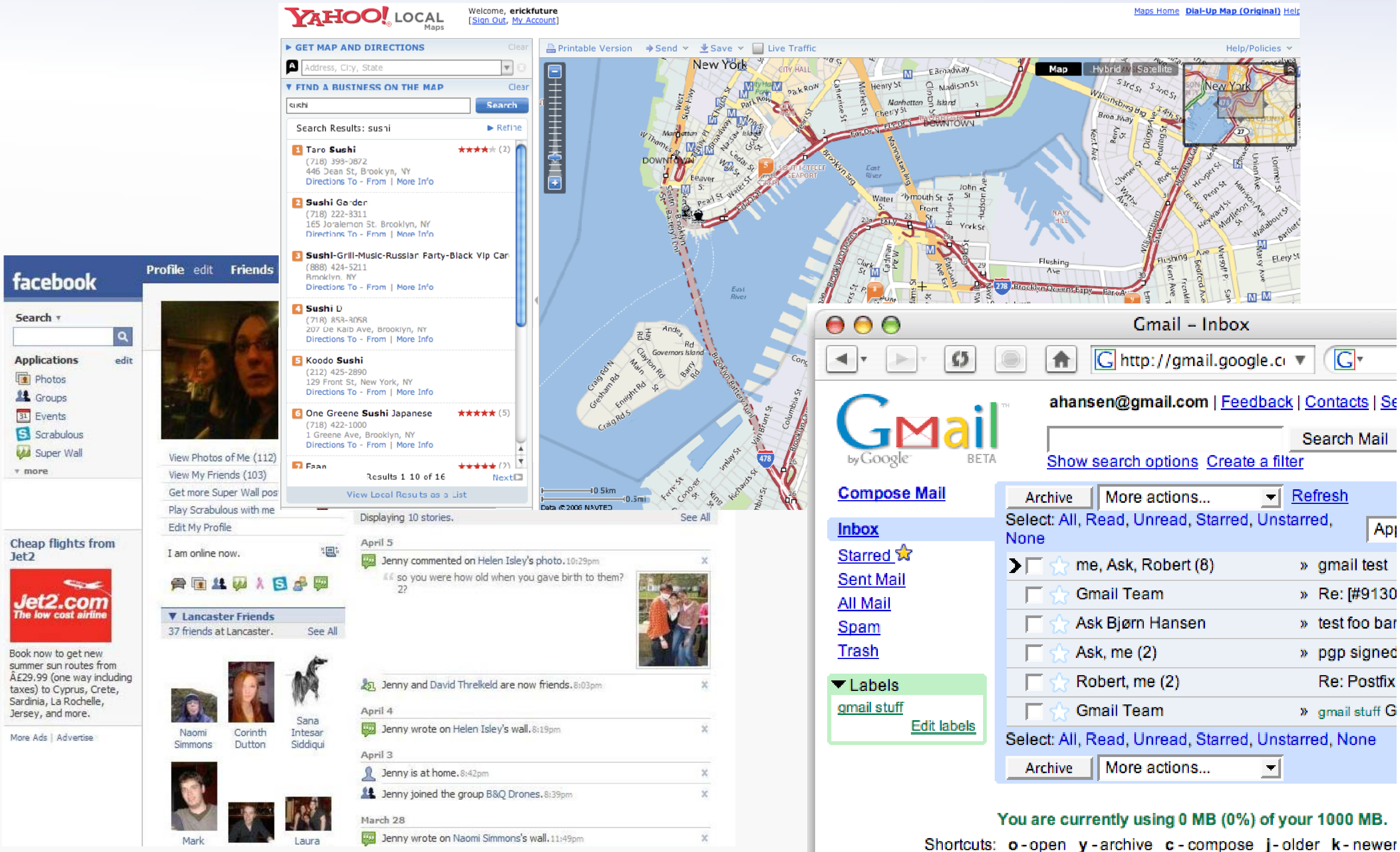
Analyzing JavaScript Web Applications

Anders Møller

Aarhus University

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

JavaScript: the lingua franca of Web 2.0



Search Results: sushi

1 Taro Sushi (2)
(718) 398-5872
446 Dean St, Brooklyn, NY
Directions To - From | More Info

2 Sushi Garden
(718) 222-3311
165 Joralemon St, Brooklyn, NY
Directions To - From | More Info

3 Sushi-Grill-Music-Russian Party-Black Vip Car
(888) 424-5211
Brooklyn, NY
Directions To - From | More Info

4 Sushi D
(718) 853-3058
207 De Kalb Ave, Brooklyn, NY
Directions To - From | More Info

5 Koodo Sushi
(212) 425-2890
129 Front St, New York, NY
Directions To - From | More Info

6 One Greene Sushi Japanese (5)
(718) 422-1000
1 Greene Ave, Brooklyn, NY
Directions To - From | More Info

Results 1 10 of 16 Next

View Local Results as a List

Cheap flights from Jet2

Jet2.com The low cost airline

Book now to get new summer sun routes from Â£29.99 (one way including taxes) to Cyprus, Crete, Sardinia, La Rochelle, Jersey, and more.

I am online now.

Lancaster Friends
37 friends at Lancaster.

Displaying 10 stories.

April 5
Jenny commented on Helen Isley's photo. 10:29pm
so you were how old when you gave birth to them? ??

Jenny and David Threlkeld are now friends. 8:03pm

April 4
Jenny wrote on Helen Isley's wall. 8:19pm

April 3
Jenny is at home. 8:42pm
Jenny joined the group B&Q Drones. 8:39pm

March 28
Jenny wrote on Naomi Simmons's wall. 11:49pm

ahansen@gmail.com

Select: All, Read, Unread, Starred, Unstarred, None

me, Ask, Robert (8)	» gmail test
Gmail Team	» Re: [#9130
Ask Bjørn Hansen	» test foo bar
Ask, me (2)	» pgp signed
Robert, me (2)	Re: Postfix
Gmail Team	» gmail stuff G

Select: All, Read, Unread, Starred, Unstarred, None

You are currently using 0 MB (0%) of your 1000 MB.

Shortcuts: o - open y - archive c - compose j - older k - newer

JavaScript is a *dynamic language*

- Object-based
- Prototype-based inheritance
- First-class functions, closures
- Runtime types
- …

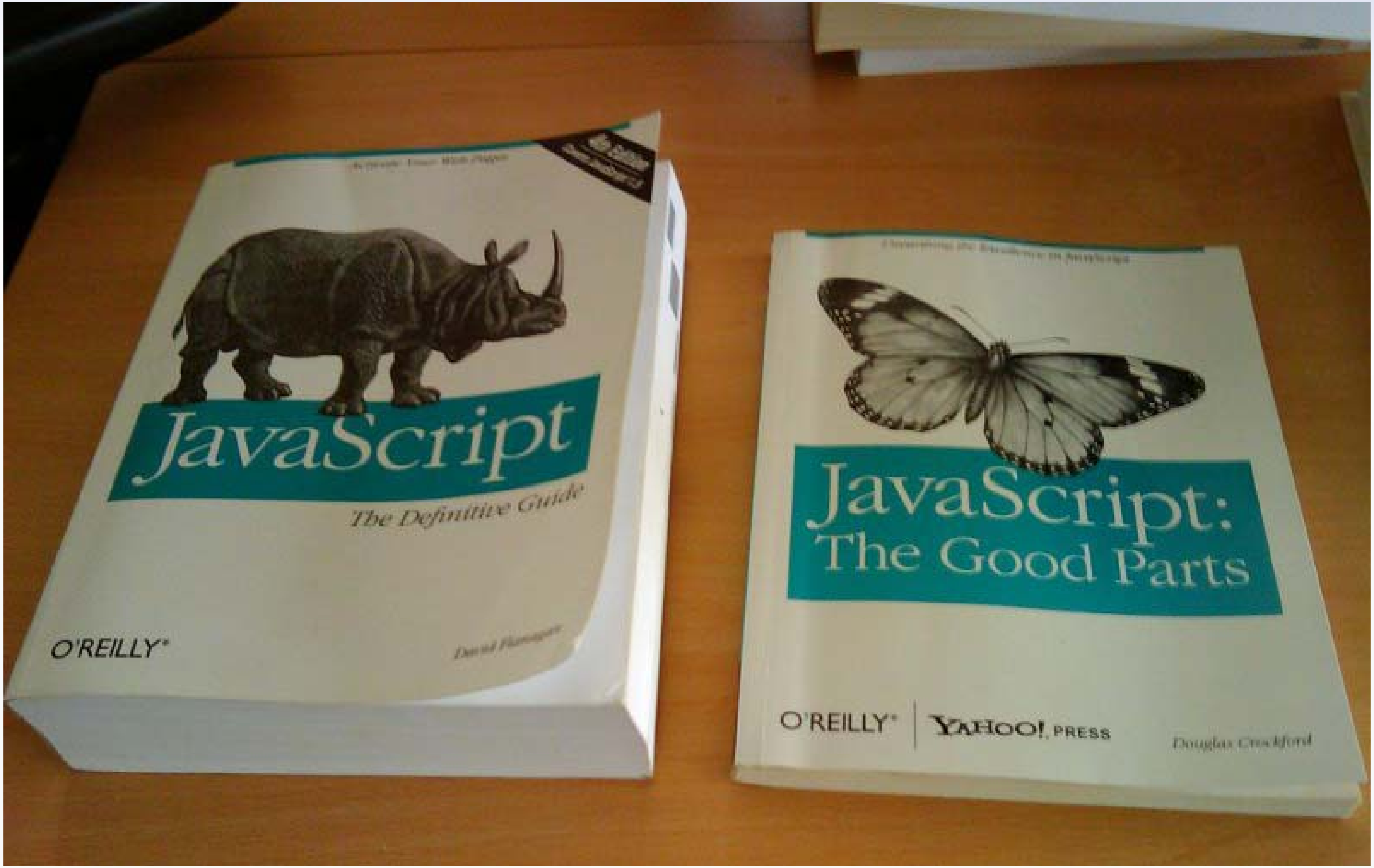
NO STATIC TYPE CHECKING

How JavaScript was designed

> “I hacked the JS prototype
> in ~1 week. And it showed!
> Mistakes were frozen early.”

– Brendan Eich, inventor of JavaScript

JavaScript


JavaScript
The Definitive Guide
O'REILLY
David Flanagan
JavaScript: The Good Parts
O'REILLY
YAHOO! PRESS
Douglas Crockford

Static type analysis to the rescue!

```
var x = {
  a: 42,
  b: function() {return a;}
}
print(x.b())
```

an *abstract state* for each program point,
analyze it further to detect likely programming errors

Potential 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 = 100026.0;
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

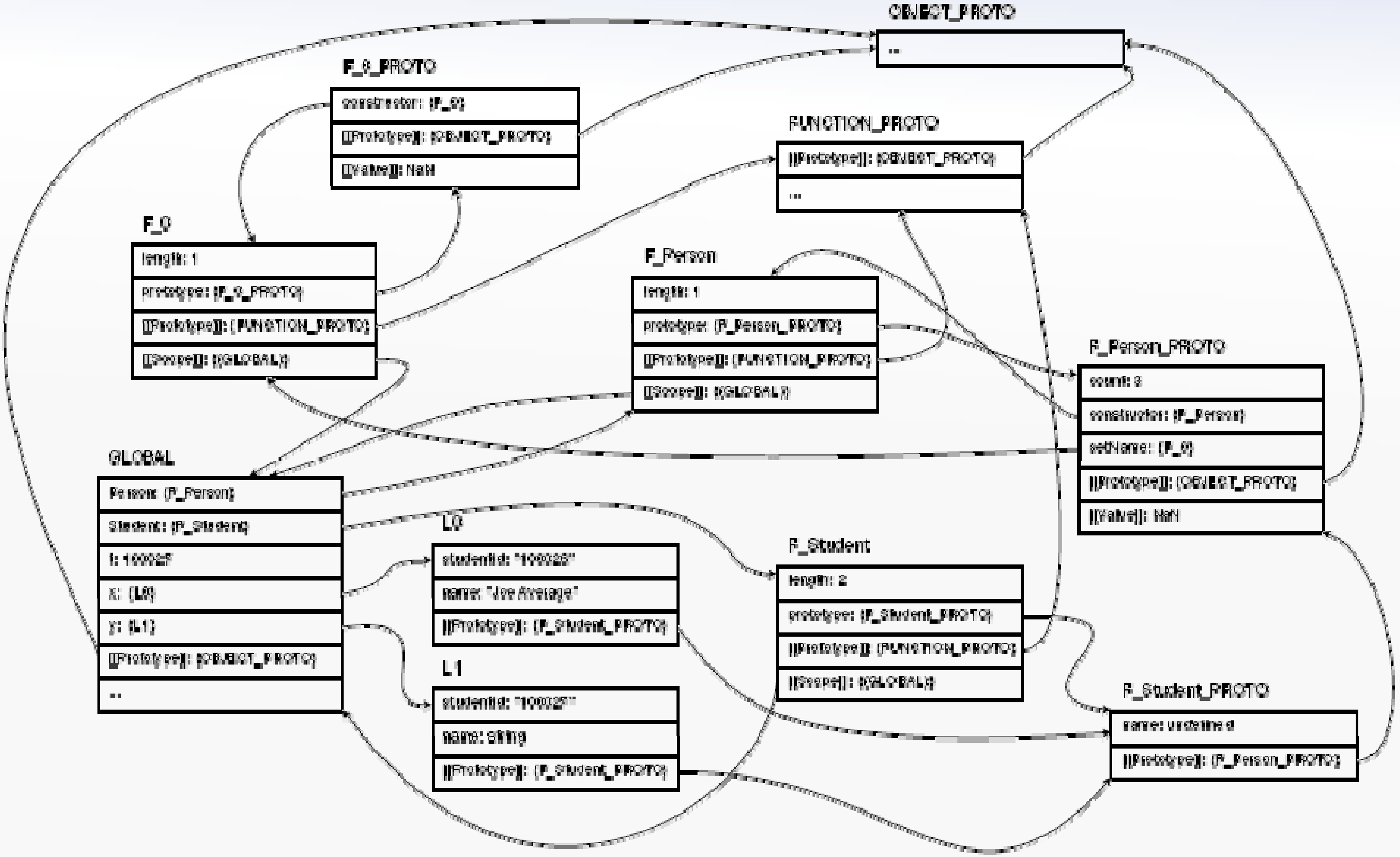
declares a shared method named setName

declares a “sub-class” named Student

creates two Student objects...

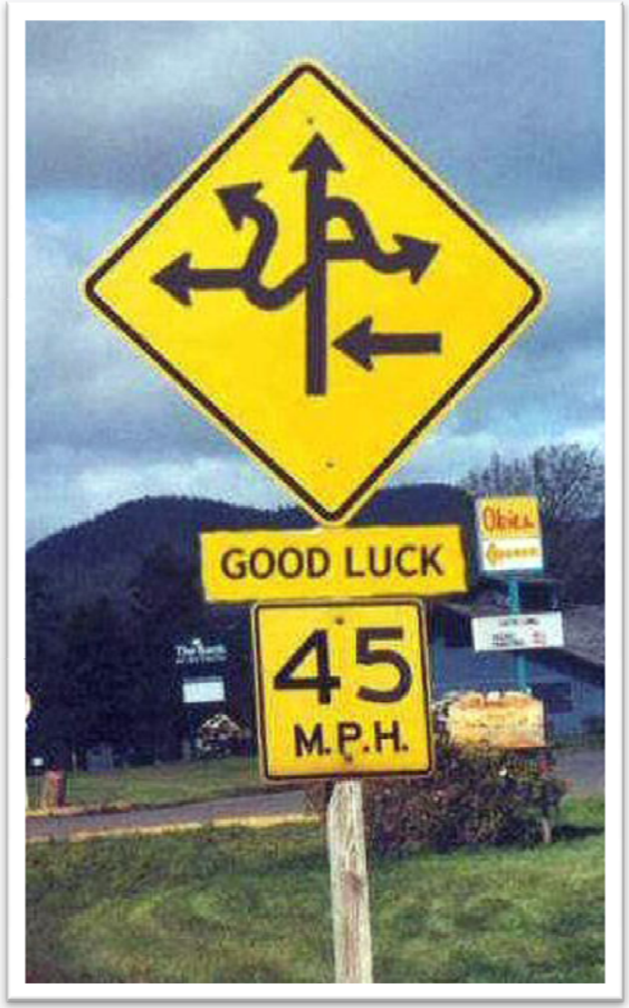
does y have a `setName` method at this program point?

An abstract state (as produced by our analysis)



General and widely used approaches

- *The monotone framework*
  [Kam & Ullman '77]
- *The functional approach*
  [Sharir & Pnueli '81]
- *IFDS*
  [Reps, Horwitz, and Sagiv '95]
- …



Our approach

- Abstract interpretation using the monotone framework

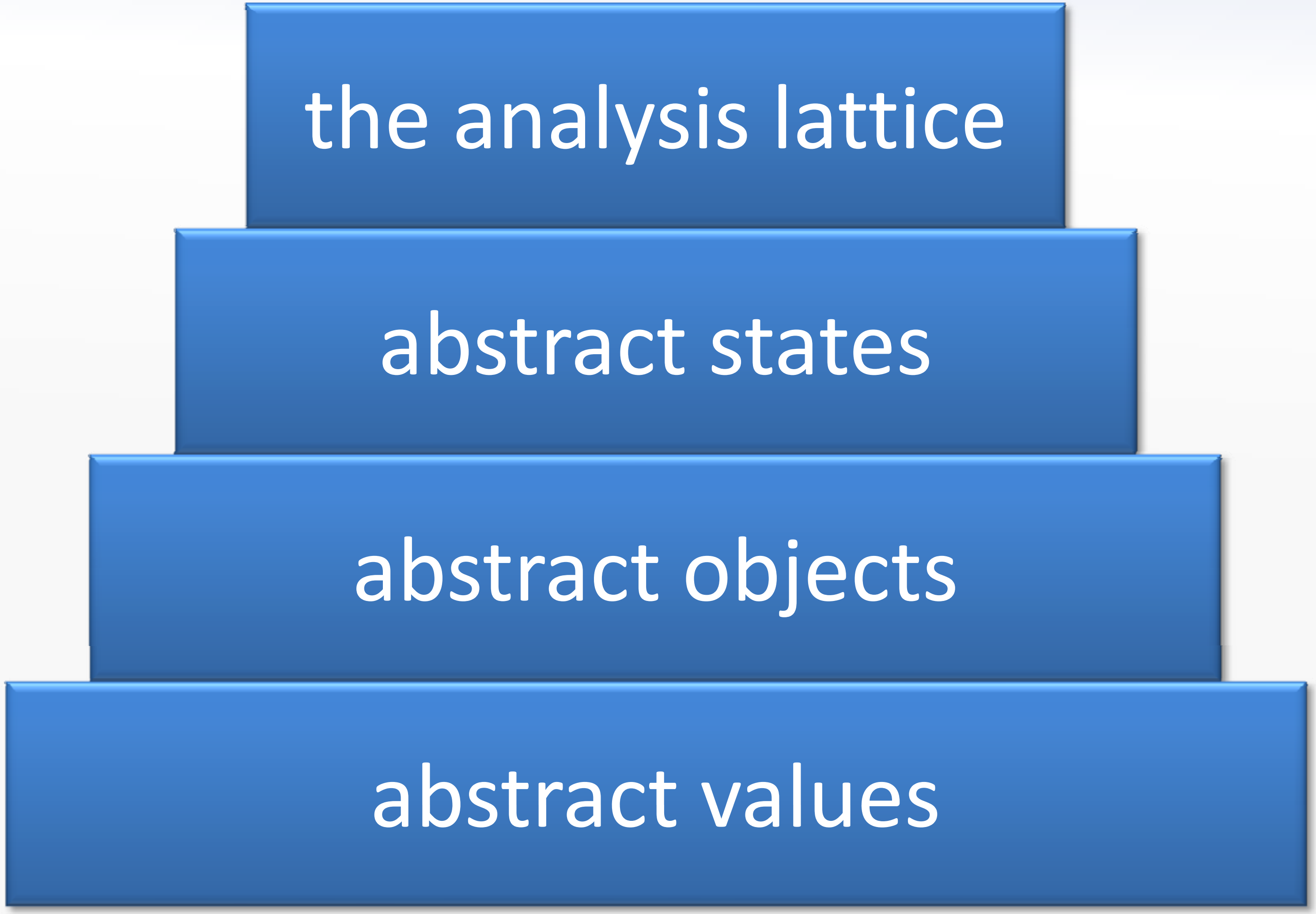
- The recipe:
  1. construct a **control flow graph** for the program to be analyzed
  2. define an appropriate **dataflow lattice** (abstraction of data)
  3. define **transfer functions** (abstraction of operations)

Control flow graphs

- declare-variable[x]
- read-variable[x,v]
- write-variable[v,x]
- constant[c,v]
- read-property[v_{obj},$v_{property}$,v_{result}]
- write-property[v_{obj},$v_{property}$,v_{value}]
- delete-property[v_{obj},$v_{property}$,v_{result}]
- if[v]
- entry[f,x_1,…,x_n], exit, exit-exc
- call[w,v_0,…,v_n], construct[w,v_0,…,v_n], after-call[v]
- return[v]
- throw[v], catch[x]
- <op>[v_1,v_2], <op>[v_1,v_2,v_3]
- …

- Convenient representation of JavaScript programs
- *Nodes* describe primitive instructions, *edges* describe control-flow
- Each x is a program variable
- Each v is a temporary variable (i.e. a register)

Analysis lattice



the analysis lattice

abstract states

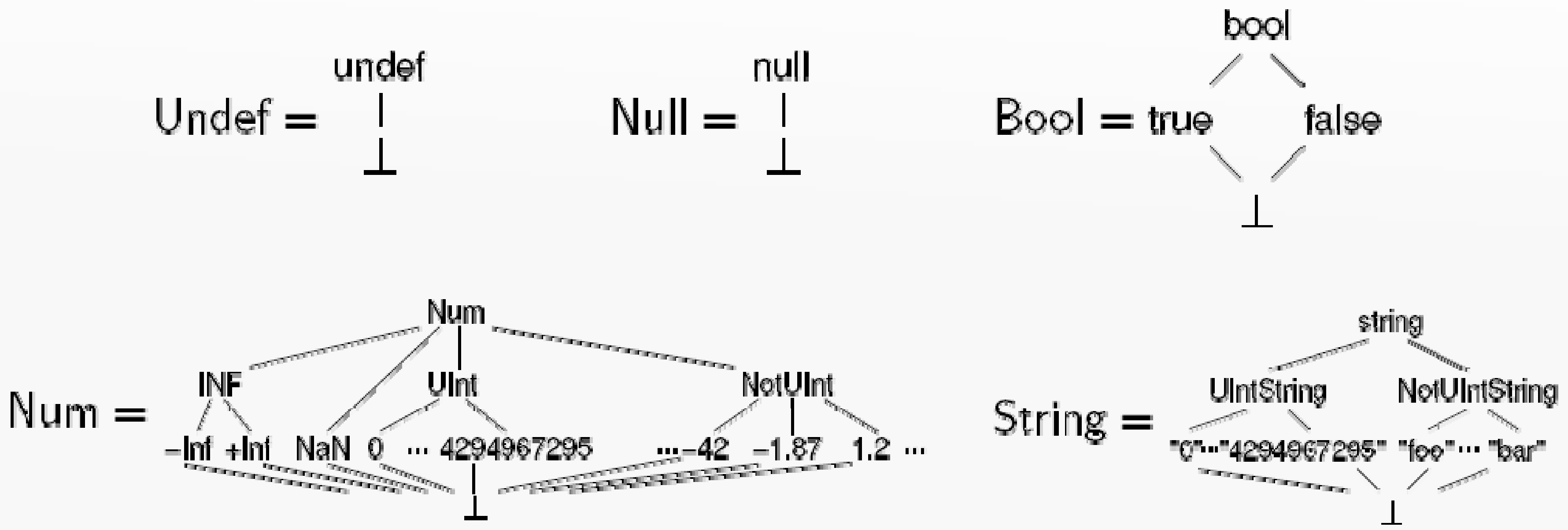
abstract objects

abstract values

Abstract values

object labels
(allocation sites)

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



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

Abstract objects

property names including [[Prototype]]

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

(explained later)

describes the [[Scope]] property

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

Abstract states

heap

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

(explained later)

temporary variables

the current activation record

stack-reachable objects

$$\mathsf{Stack} = (\mathcal{T} \rightarrow \mathsf{Value}) \times \mathcal{P}(\mathsf{ExecutionContext}) \times \mathcal{P}(\mathcal{L})$$

$$\mathsf{ExecutionContext} = \mathsf{ScopeChain} \times \mathcal{L} \times \mathcal{L}$$

$$\mathsf{ScopeChain} = \mathcal{L}^*$$

the `this` object

the variable object, for variable declarations

for resolving variables

The analysis lattice

$$\text{AnalysisLattice} = C \times N \rightarrow \text{State}$$

contexts

(for context sensitivity)

the flow graph nodes

Transfer functions

Example: **read-property**[v_{obj},$v_{property}$,v_{result}]

1. Coerce v_{obj} to objects
2. Coerce $v_{property}$ 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 v_{result}

Weak vs. strong updates

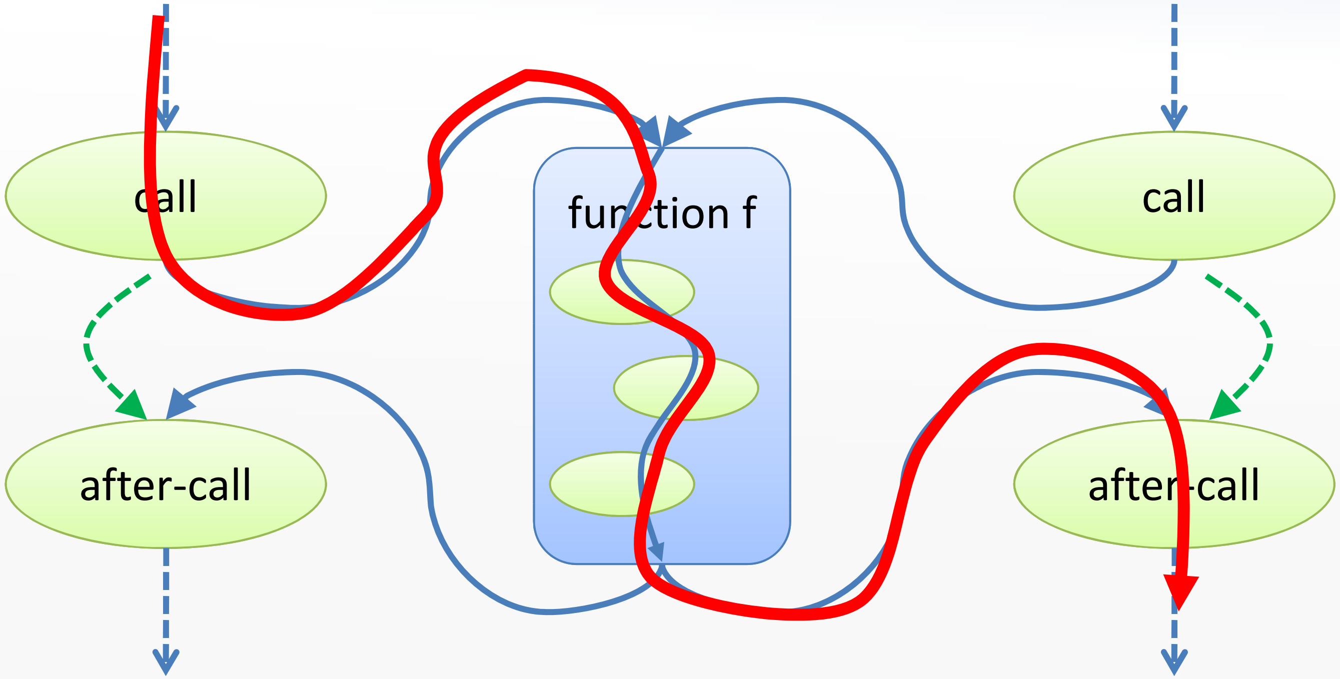
- For a **write-property**[v_{obj},$v_{property}$,v_{value}] node, v_{obj} refers to one or more abstract objects (identified by their allocation sites)
- Each abstract object generally describes *multiple* concrete objects
- So **write-property** must conservatively be modeled by *joining* v_{value} into the existing value of $v_{property}$ at v_{obj} (i.e. a *weak update*)
- This is bad for precision!
- *Strong update* (overwriting instead of joining) is possible whenever the abstract object is known to represent a single 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

Interprocedural analysis with *maybe-modified*



At function exits, **restore unmodified parts of the heap (and the stack)** from the call node

Observing redundancy

```
...
TaskControlBlock.prototype.markAsRunnable = function () {
  this.state = this.state | STATE_RUNNABLE;
};
...
```

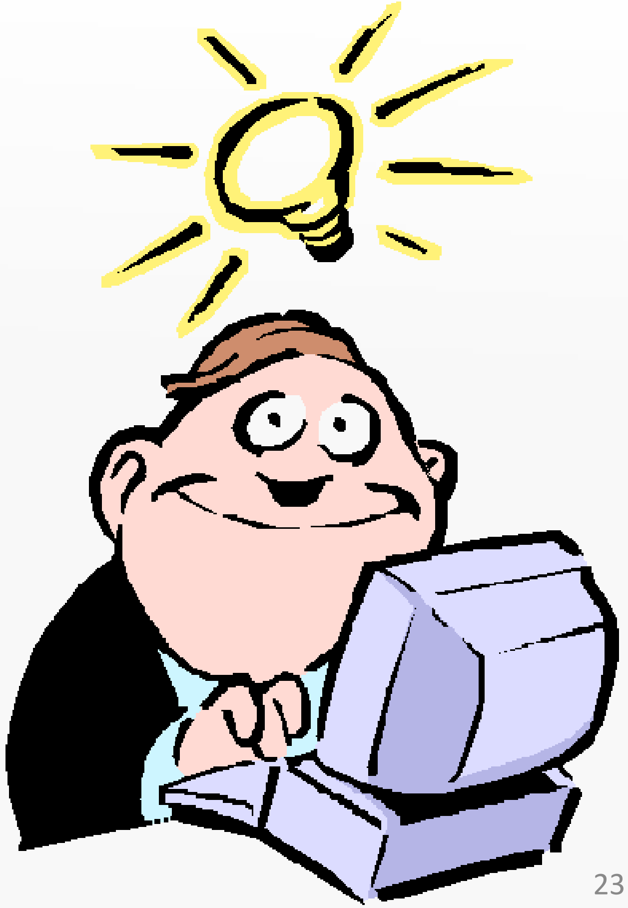
- Why is this function (from `richards.js`, V8) visited **18** times by the analyzer???
- Mostly, new dataflow that arrives at the function entry (and triggers re-analysis) is irrelevant to the function body!

Lazy propagation

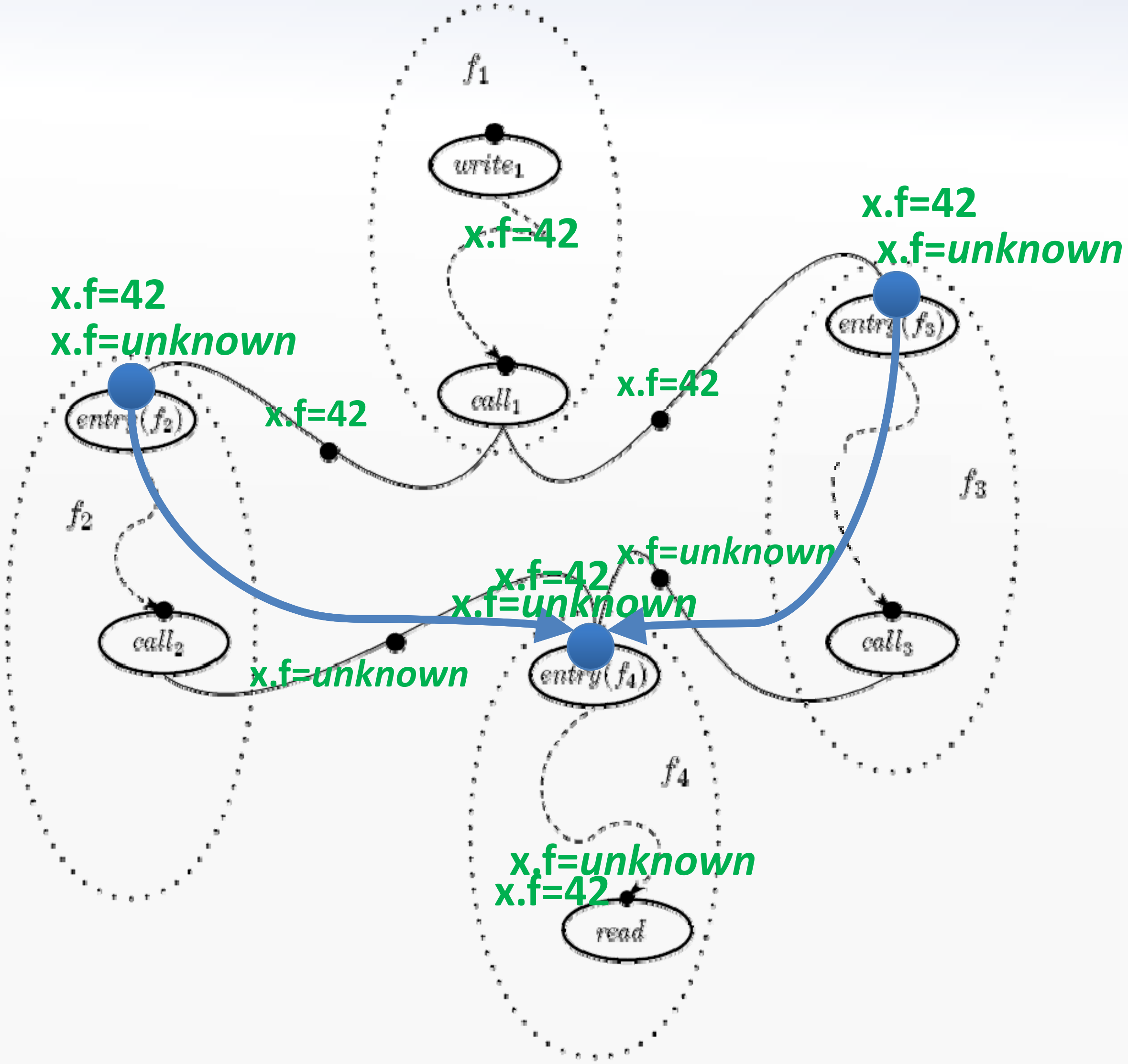
- Defer propagation of field values that are not known to be relevant to the current function
- Use a placeholder value: ***unknown***
- When analyzing a function, assume initially that no fields are referenced
- When a field is referenced, recover its proper value

⇒ irrelevant dataflow isn't propagated

⇒ *unknown* implies *unmodified*



An example



- Each • represents an abstract state
- For simplicity, no context sensitivity here

Formalization of lazy propagation

How do we express the idea more concisely and formally?

(necessary for reasoning about its properties and for obtaining a good implementation)

1) Start with a basic analysis framework where transfer functions are expressed via an **abstract data type** (ADT)
2) Introduce lazy propagation by a systematic modification of the ADT **(without touching the transfer functions!)**

The basic lattice (simplified)

object labels (allocation sites)

functions

$$\mathsf{Value} = \mathcal{P}(L) \times \mathcal{P}(F) \times \mathsf{Base}$$

$$\mathsf{Obj} = P \rightarrow \mathsf{Value}$$

property names (fields)

$$\mathsf{State} = L \rightarrow \mathsf{Obj}$$

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

set of call edges

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

contexts

nodes (primitive statements)

AnalysisLattice **as an abstract data type (ADT)**

reads a field value

$$\mathit{getfield} : C \times N \times L \times P \to \mathsf{Value}$$

$$\mathit{getcallgraph} : () \to \mathsf{CallGraph}$$

reads the call graph

reads an abstract state

$$\mathit{getstate} : C \times N \to \mathsf{State}$$

$$\mathit{propagate} : C \times N \times \mathsf{State} \to ()$$

intra-procedural flow

$$\mathit{funentry} : C \times N \times C \times F \times \mathsf{State} \to ()$$

$$\mathit{funexit} : C \times N \times C \times F \times \mathsf{State} \to ()$$

inter-procedural flow

- The transfer functions can only access the AnalysisLattice element through these operations
- (We’ll skip their definitions here…)

Introducing lazy propagation

– a systematic modification of the lattice and the ADT operations

property values can now be “unknown”!

$$\mathsf{Obj} = P \rightarrow (\mathsf{Value}{\downarrow}_{\mathsf{unknown}})$$

$$\mathsf{CallGraph} = C \times N \times C \times F \rightarrow (\mathsf{State}{\downarrow}_{\mathsf{none}})$$

$$\mathsf{AnalysisLattice} = (C \times N \rightarrow (\mathsf{State}{\downarrow}_{\mathsf{none}})) \times \mathsf{CallGraph}$$

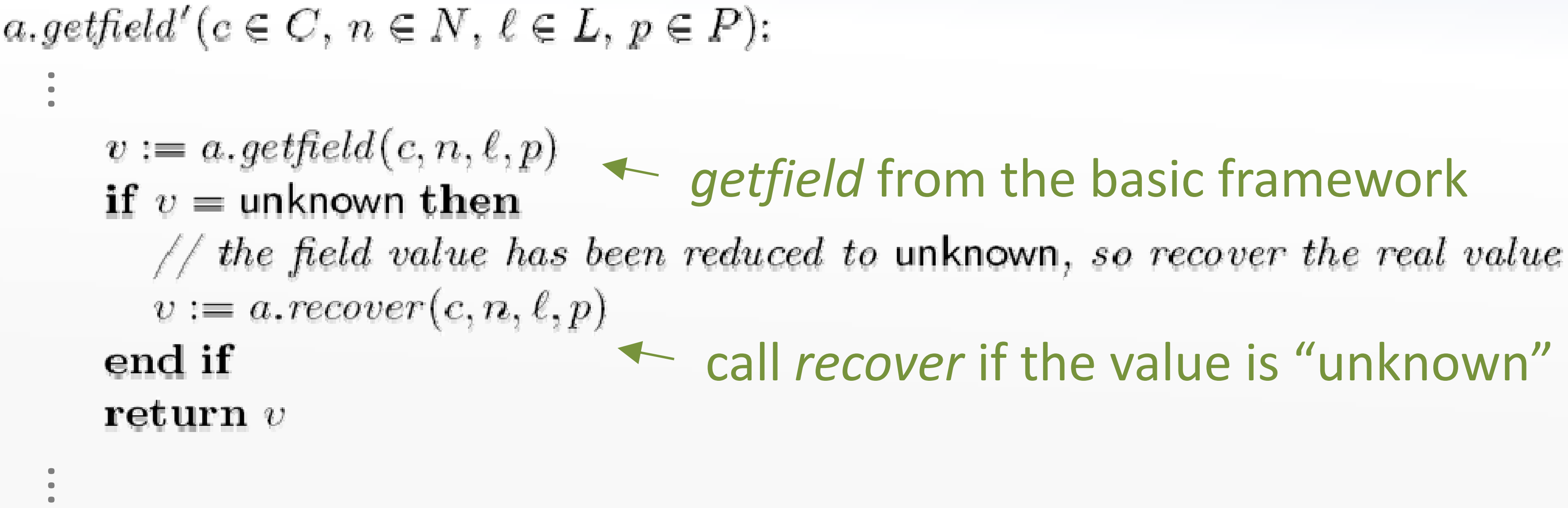
now distinguishing between the *unreachable* state and the all-*unknown* state

each call edge is now labelled with an abstract state

$$recover : C \times N \times L \times P \rightarrow \mathsf{Value}$$

used for recovering “unknown” values

getfield' (read a field)



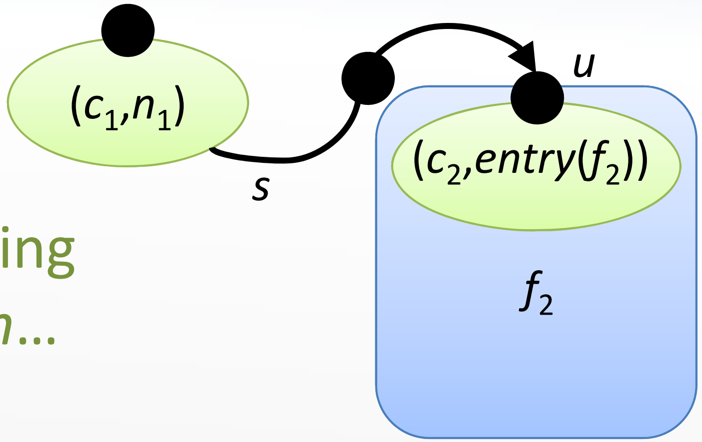
```
$a.getfield'(c \in C, n \in N, \ell \in L, p \in P)$:
  ⋮
    $v := a.getfield(c, n, \ell, p)$
    **if** $v$ = unknown **then**
      // the field value has been reduced to unknown, so recover the real value
      $v := a.recover(c, n, \ell, p)$
    **end if**
    **return** $v$
  ⋮
```

← *getfield* from the basic framework

← call *recover* if the value is "unknown"

funentry' (flow at function entry)

```
$a.funentry'(c_1 \in C, n_1 \in N, c_2 \in C, f_2 \in F, s \in \mathsf{State})$:
  let $(m, g) = a$ and $u = m(c_2, entry(f_2))$
  ⋮
  // introduce unknown field values
  $s' := \bot_{\mathsf{State}}$
  if $u \neq$ none then
    for all $\ell \in L, p \in P$ do
      if $u(\ell)(p) \neq$ unknown then
        // the field has been referenced
        $s'(\ell)(p) := s(\ell)(p)$
      end if
    end for
  end if
  // propagate the resulting state into the function entry
  $a.propagate'(c_2, entry(f_2), s')$
  ⋮
```

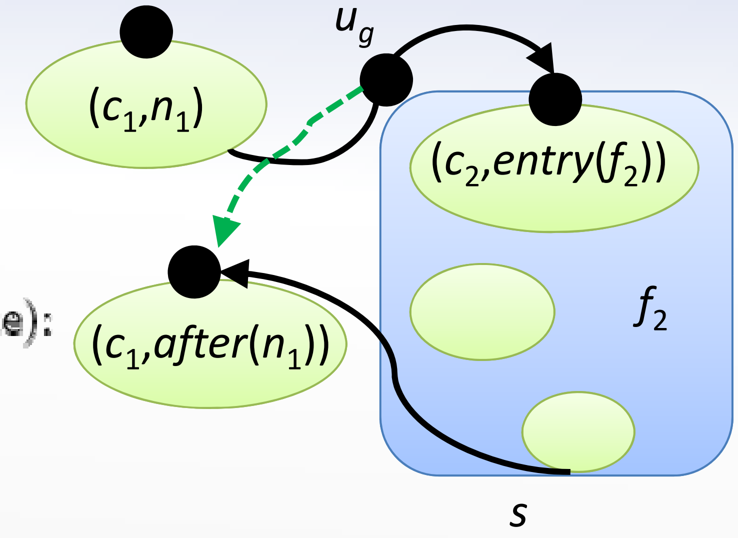


set everything to *unknown*…

… except fields that are known to be referenced

join *s'* into the function entry state

funexit' (flow at function exit)



```
$a.funexit'(c_1 \in C,\ n_1 \in N,\ c_2 \in C,\ f_2 \in F,\ s \in \mathsf{State})$:
  let $(\_, g) = a$ and $u_g = g(c_1, n_1, c_2, f_2)$
  $s' := \bot_{\mathsf{State}}$
  for all $\ell \in L, p \in P$ do
    if $s(\ell)(p) = \mathsf{unknown}$ then
      // the field has not been accessed, so restore its value from the call edge state
      $s'(\ell)(p) := u_g(\ell)(p)$
    else
      $s'(\ell)(p) := s(\ell)(p)$
    end if
  end for
  $a.propagate'(c_1, after(n_1), s')$
```

unknown
implies
not modified within the function

join *s'* into the node after the call

Theoretical properties of lazy prop.

- **Precision** is at least as good as before
- **Soundness** (wrt. language semantics) is preserved
- **Recovery** does not affect amortized complexity
- **Number of fixpoint iterations** increases in some situations and decreases in other

Experiments

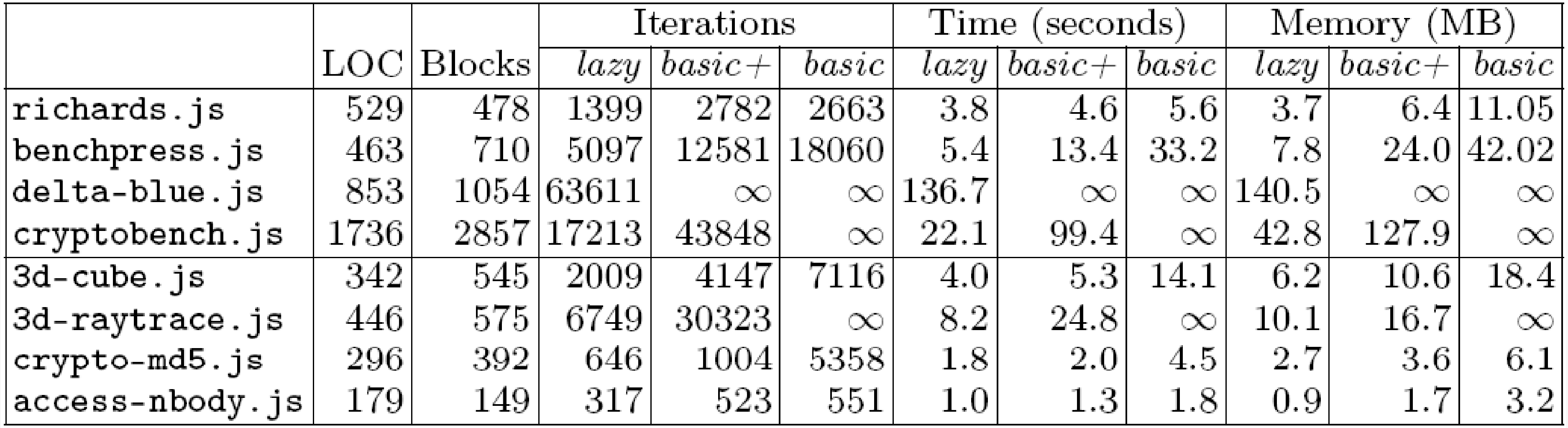
- >200 **small test cases**, to get into the obscure corner cases of JavaScript
- A few larger benchmarks: **Google's V8 benchmark suite** (500-1800 lines of code)
- Also tested on the **SunSpider benchmarks**

Experiments

Some results for `richards.js` from V8:

- the analysis guarantees for **95%** of the call/construct instructions that they always succeed
- **1** location where an absent variable is read,
  with **0** spurious warnings
- **93%** of all read/write/delete-property operations will never attempt to coerce null or undefined into an object
- **6** functions dead (guaranteed unreachable)

Experimental results



	LOC	Blocks	Iterations			Time (seconds)			Memory (MB)		
			lazy	*basic+*	*basic*	*lazy*	*basic+*	*basic*	*lazy*	*basic+*	*basic*
`richards.js`	529	478	1399	2782	2663	3.8	4.6	5.6	3.7	6.4	11.05
`benchpress.js`	463	710	5097	12581	18060	5.4	13.4	33.2	7.8	24.0	42.02
`delta-blue.js`	853	1054	63611	∞	∞	136.7	∞	∞	140.5	∞	∞
`cryptobench.js`	1736	2857	17213	43848	∞	22.1	99.4	∞	42.8	127.9	∞
`3d-cube.js`	342	545	2009	4147	7116	4.0	5.3	14.1	6.2	10.6	18.4
`3d-raytrace.js`	446	575	6749	30323	∞	8.2	24.8	∞	10.1	16.7	∞
`crypto-md5.js`	296	392	646	1004	5358	1.8	2.0	4.5	2.7	3.6	6.1
`access-nbody.js`	179	149	317	523	551	1.0	1.3	1.8	0.9	1.7	3.2

∞ means >512MB

basic: naive monotone framework

basic+: basic extended with *maybe-modified* (and *copy-on-write*)

lazy: basic extended with *lazy propagation*

Summary

- ★ ***Static analysis*** is a useful tool for reasoning about programs written in a scripting language such as JavaScript
- ★ ***Lazy propagation*** ensures that only relevant information is propagated from one function to another
  - – reduces the amount of data being propagated
  - – may improve precision: non-referenced fields respect interprocedurally realizable paths