“Type Analysis for JavaScript”

Presenter: *Austin Cory Bart*

Location: *Knowledge Works II, Room 2225*

Date: 9/22/2015

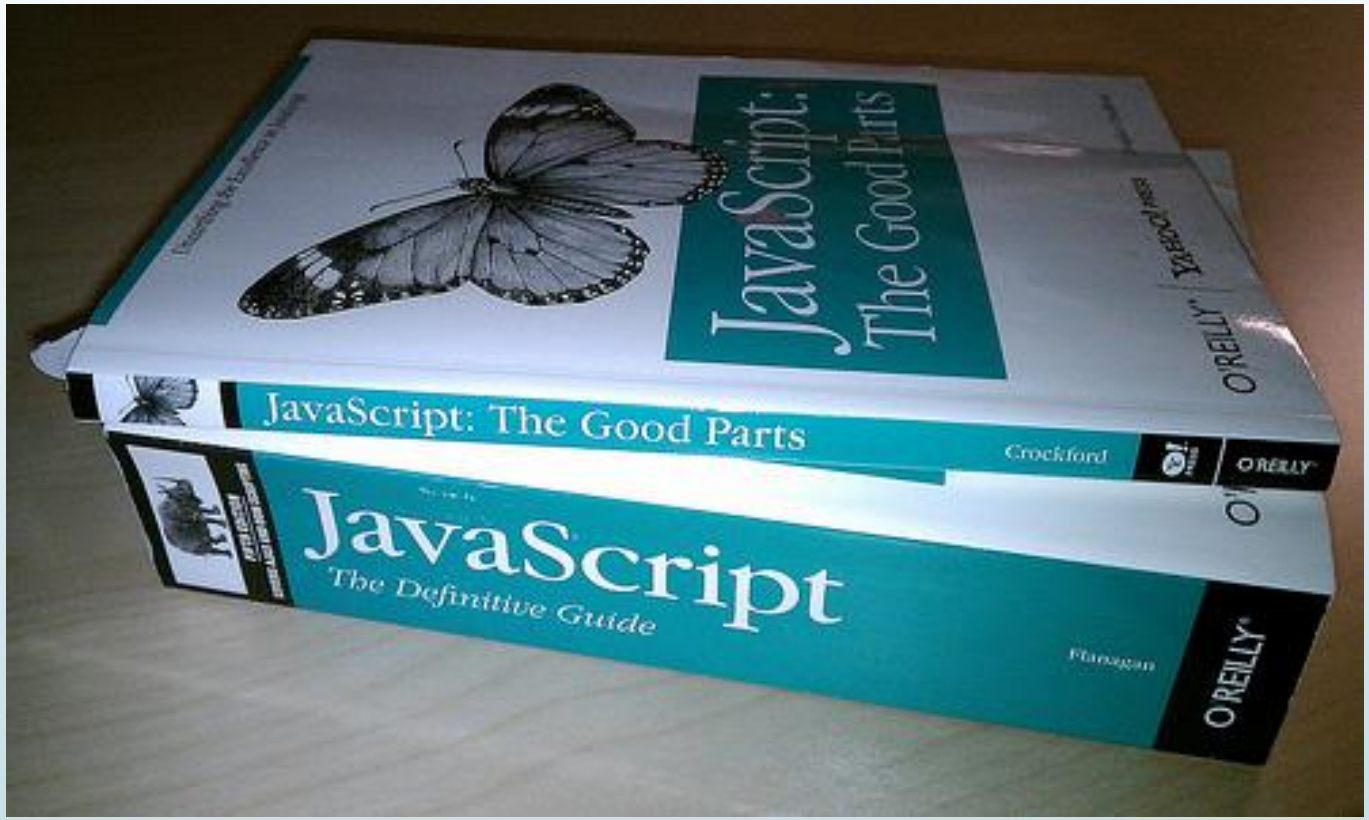
Original Authors: *S. H. Jensen, A. Moeller, & P Thiemann*

Original Venue: *SAS 2009*

Outline

1. What is JavaScript?
2. JS Analysis
3. Abstract Intepretation
4. Recency Abstraction
5. Results
6. Conclusion

What is this “JavaScript”?

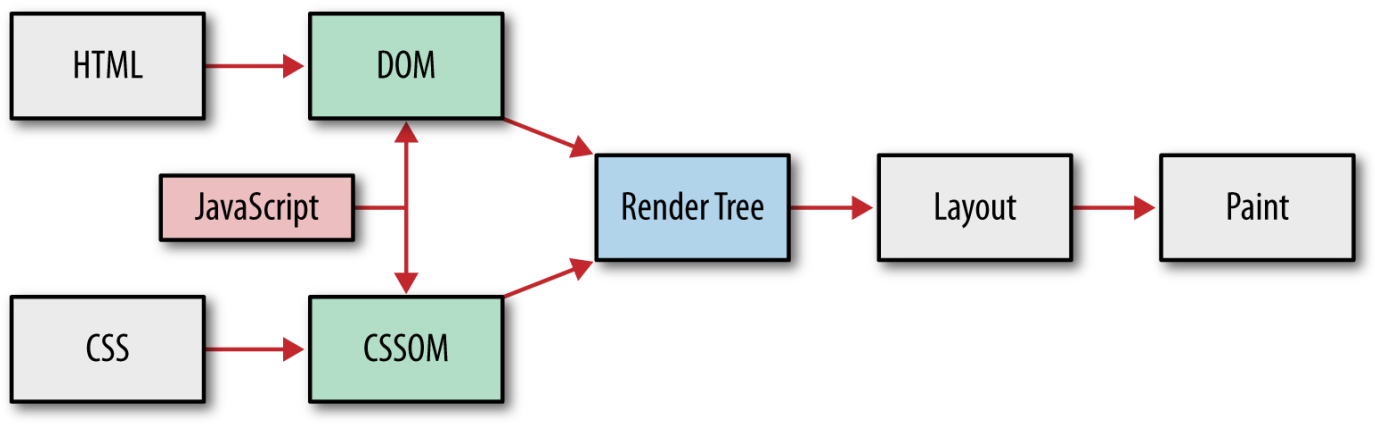


DM; TOBE

Mar 2015	Mar 2014	Change	Programming Language	Ratings	Change
1	1		C	16.642%	-0.89%
2	2		Java	15.580%	-0.83%
3	3		Objective-C	6.688%	-5.45%
4	4		C++	6.636%	+0.32%
5	5		C#	4.923%	-0.65%
6	6		PHP	3.997%	+0.30%
7	9	^	JavaScript	3.629%	+1.73%
8	8		Python	2.614%	+0.59%
9	10	^	Visual Basic .NET	2.326%	+0.46%
10	-	^^	Visual Basic	1.949%	+1.95%
11	12	^	F#	1.510%	+0.29%
12	13	^	Perl	1.332%	+0.18%
13	15	^	Delphi/Object Pascal	1.154%	+0.27%
14	11	v	Transact-SQL	1.149%	-0.33%
15	21	^^	Pascal	1.092%	+0.41%
16	31	^^	ABAP	1.080%	+0.70%
17	19	^	PL/SQL	1.032%	+0.32%
18	14	vv	Ruby	1.030%	+0.06%
19	20	^	MATLAB	0.998%	+0.31%
20	45	^^	R	0.951%	+0.72%

What is JavaScript?

- For manipulating the DOM



What is JavaScript

- C-style syntax

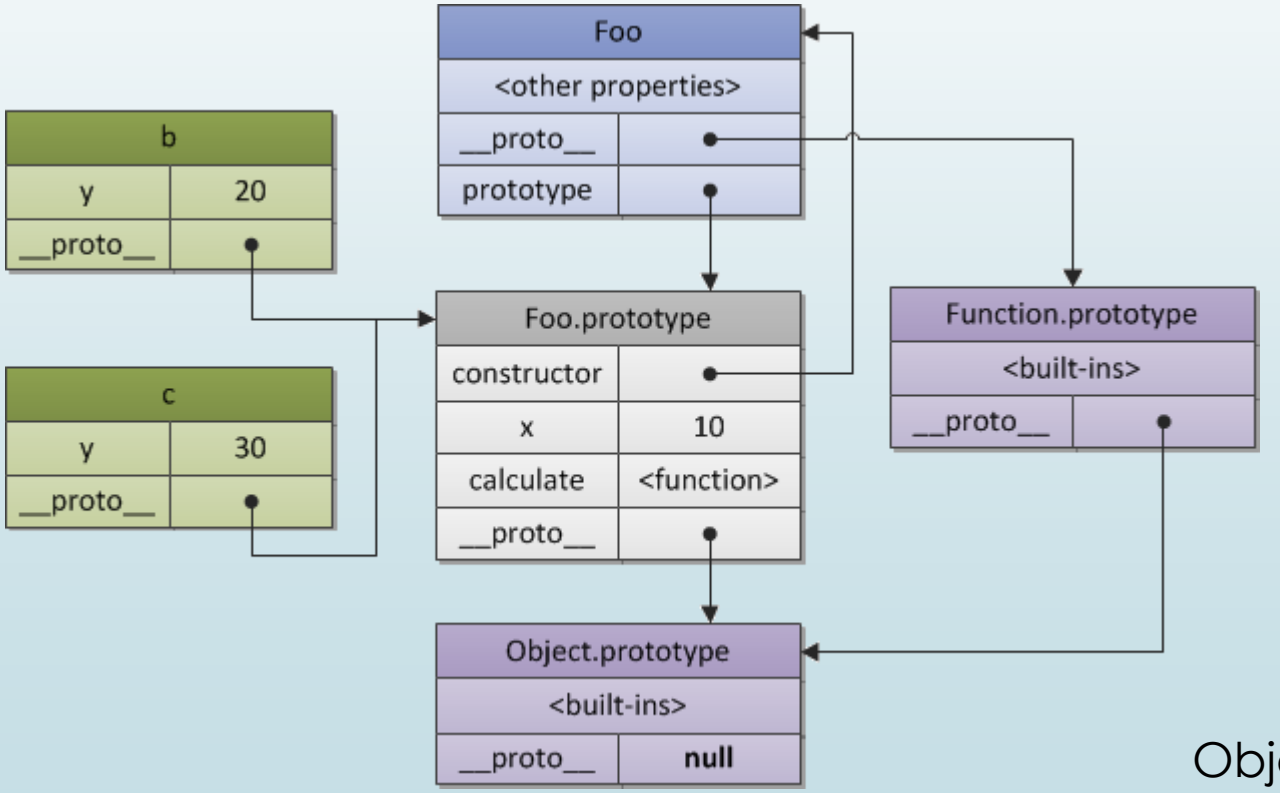
```
<SCRIPT LANGUAGE = "Javascript">

        var p1 = new Image();
        var p2 = new Image();
        var p3 = new Image();
        var p4 = new Image();
        var p5 = new Image();

        p1.src="pic_1.jpg";
        p2.src="pic_2.jpg";
        p3.src="pic_3.jpg";
        p4.src="pic_4.jpg";
        p5.src="pic_5.jpg";

        var imgArray = new Array(p1, p2, p3, p4, p5);

        var counter=0;
        var end = imgArray.length - 1;

        function scroll_backward() {

                if (counter == 0) {
                        alert("Start of pictures");
                }
                else {
                        counter--;
                }

                document.pic1.src = imgArray[counter].src;
        }

        function scroll_forward() {
                if (counter == end) {
                        alert("No more pictures");
                }
                else {
                        counter++;
                }
                document.pic1.src = imgArray[counter].src;
        }

</SCRIPT>
```

What is JavaScript?

- Absurd typing: (Almost) Nothing ever fails

What is JavaScript

- Prototype-based language



Object-level function overriding
Runtime prototype changing

What is JavaScript

- Dynamic

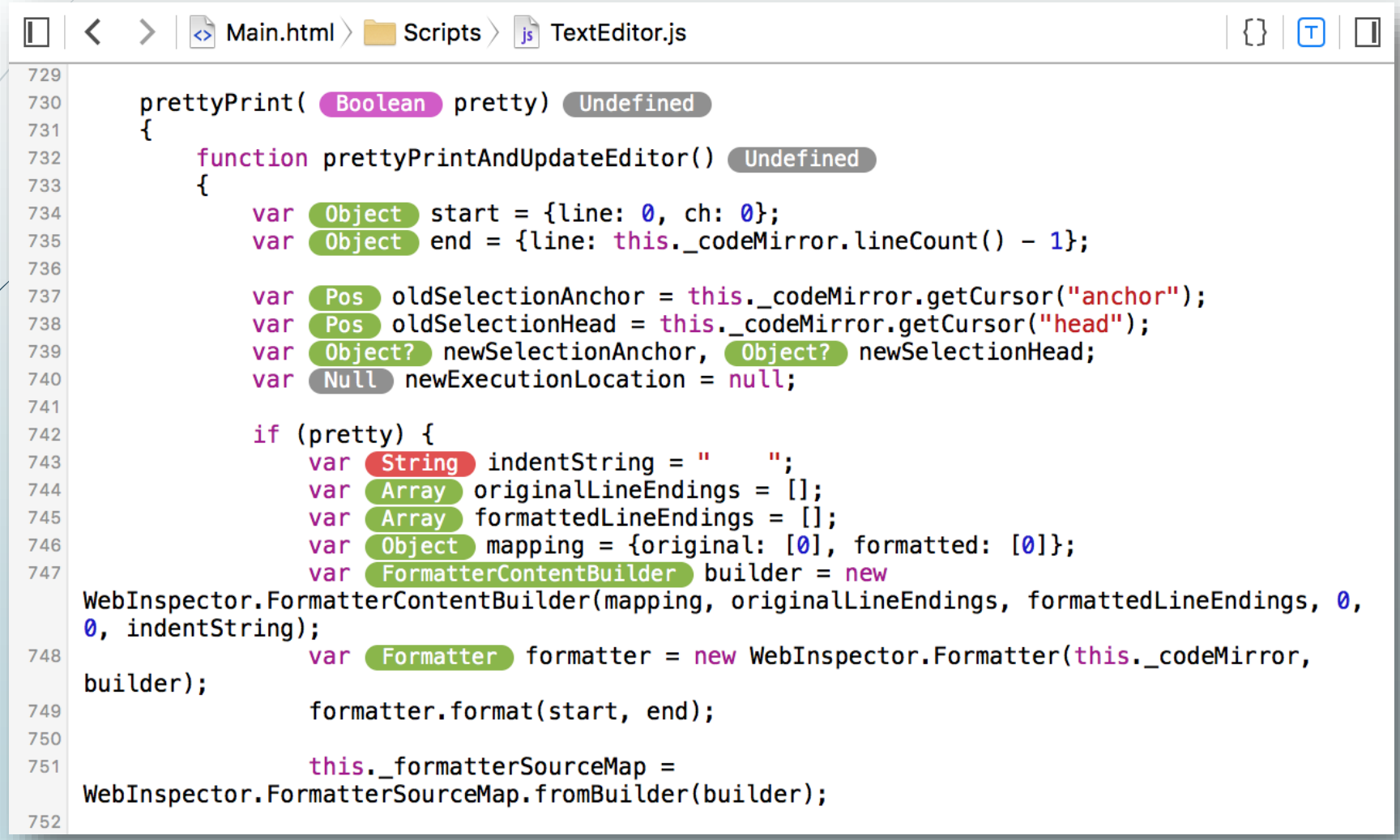
```
a = new A;
a.new_property = 5
a['what a crazy language'] = a.new_property
```

What is JavaScript

- Multiple levels of non-existence:
  - 0, false, “”
  - null
  - undefined

```
> a = {}
< Object {}
> a.b
< undefined
> a.b = 5
< 5
> a
< Object {b: 5}
> a.c = undefined
< undefined
> a
< Object {b: 5, c: undefined}
> a.c
< undefined
```

Type Analysis



Type Errors

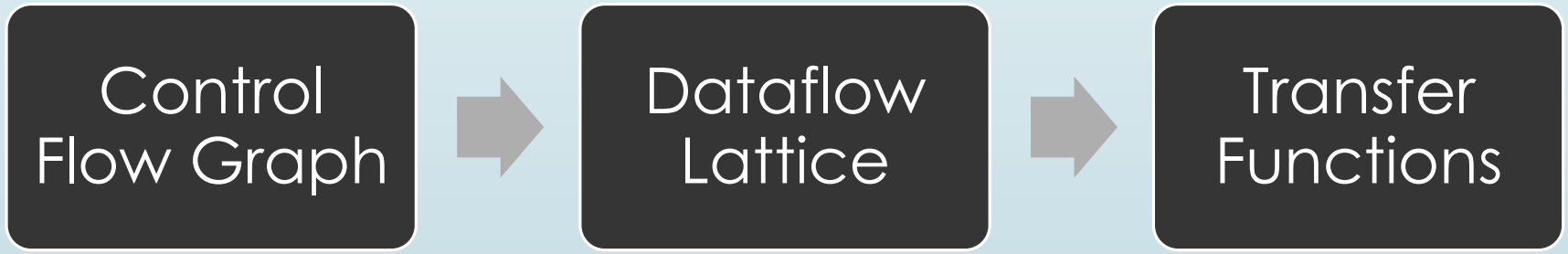
- Calling a property that's null
- Accessing the field of a null/undefined
- Reading an absent value

Analysis characteristics

- Handwritten, <1000 LOC
- Sound
- Automatic
- Full language (including `eval`)

Abstract Intepretation

- Dataflow through the Monotone Framework
  - Lattice + Set of Monotone Functions (L -> L)



(1) JS Control Flow Graph

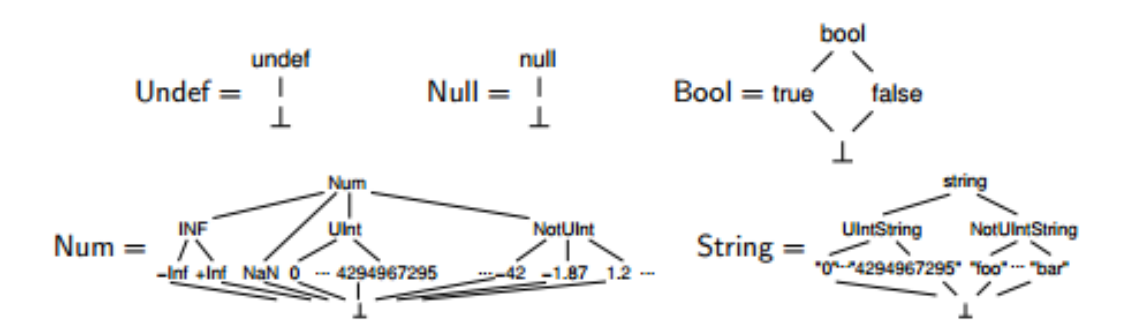
- declare-variable[*variable*]
- read-variable[*variable*, *temp*]
- write-variable[*temp*, *variable*]
- constant[*constant*, *variable*]
- read-property[*variable*, *field_name*, *temp*]
- write-property[*variable*, *field_name*, *temp*]
- delete-property[*variable*, *field_name*, *temp*]
- if[variable]
- entry[function, variable_1, ..., variable_n]/exit/exit-exc
- call[function, this, variable_1, ..., variable_n], construct, after_call
- return[variable]
- throw[variable] , catch[variable]
- <op>[variable_left, variable_right], binary and ternary operators

(2) DataFlow Lattice (A)

"These definitions are the culmination of tedious twiddling and experimentation"

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

L is the set of object labels



Examples:

$(\bot, \text{null}, \bot, \bot, \text{baz}, \emptyset)$

$(\text{undef}, \bot, \bot, \bot, \bot, \{\ell_{42}, \ell_{87}\})$

(2) Dataflow Lattice (B)

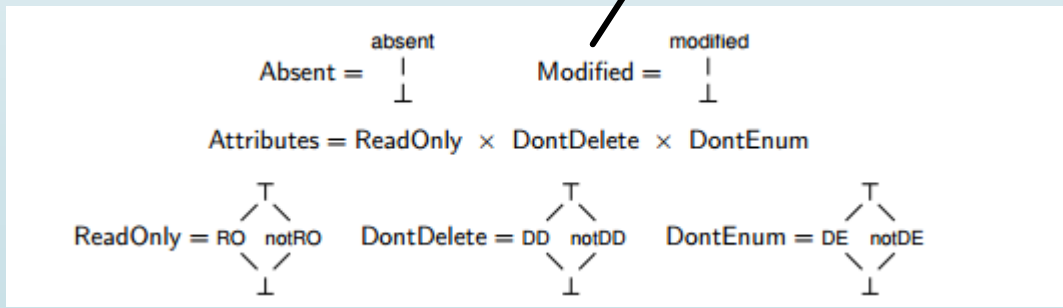
$$\text{Obj} = (P \hookrightarrow \text{Value} \times \text{Absent} \times \text{Attributes} \times \text{Modified}) \times \mathcal{P}(\text{ScopeChain})$$

All property labels

See part A

Handles closures

Modified since entry into function



(2) Dataflow Lattice (C)

Object Labels for handling intra-procedural analysis (definitely, maybe summarized)

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

Object Label

Temporaries

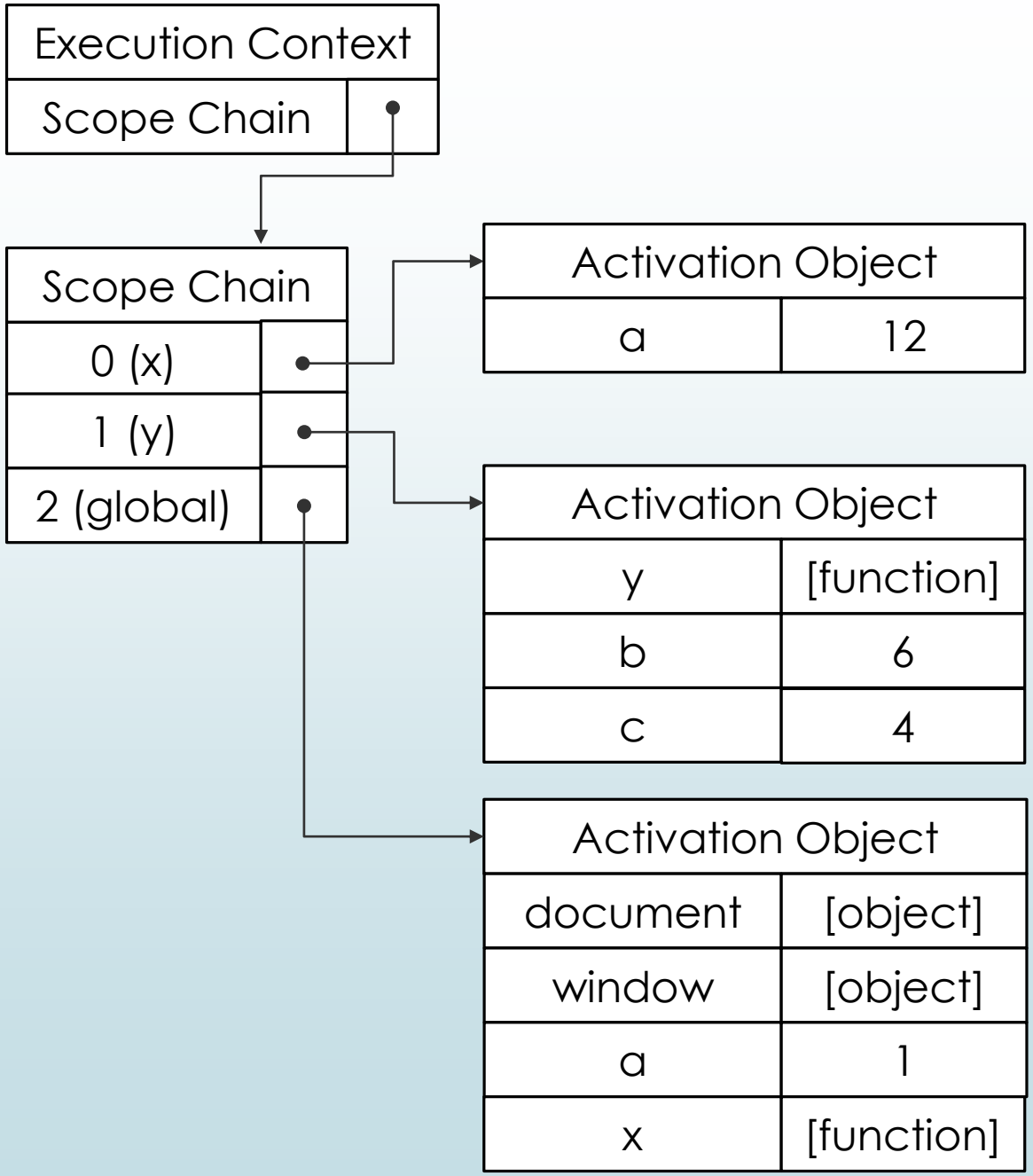
Stack Frames

All property labels in stack

$$\begin{aligned} \text{Stack} &= (T \rightarrow \text{Value}) \times \mathcal{P}(\text{ExecutionContext}) \times \mathcal{P}(L) \\ \text{ExecutionContext} &= \text{ScopeChain} \times L \times L \\ \text{ScopeChain} &= L^* \end{aligned}$$

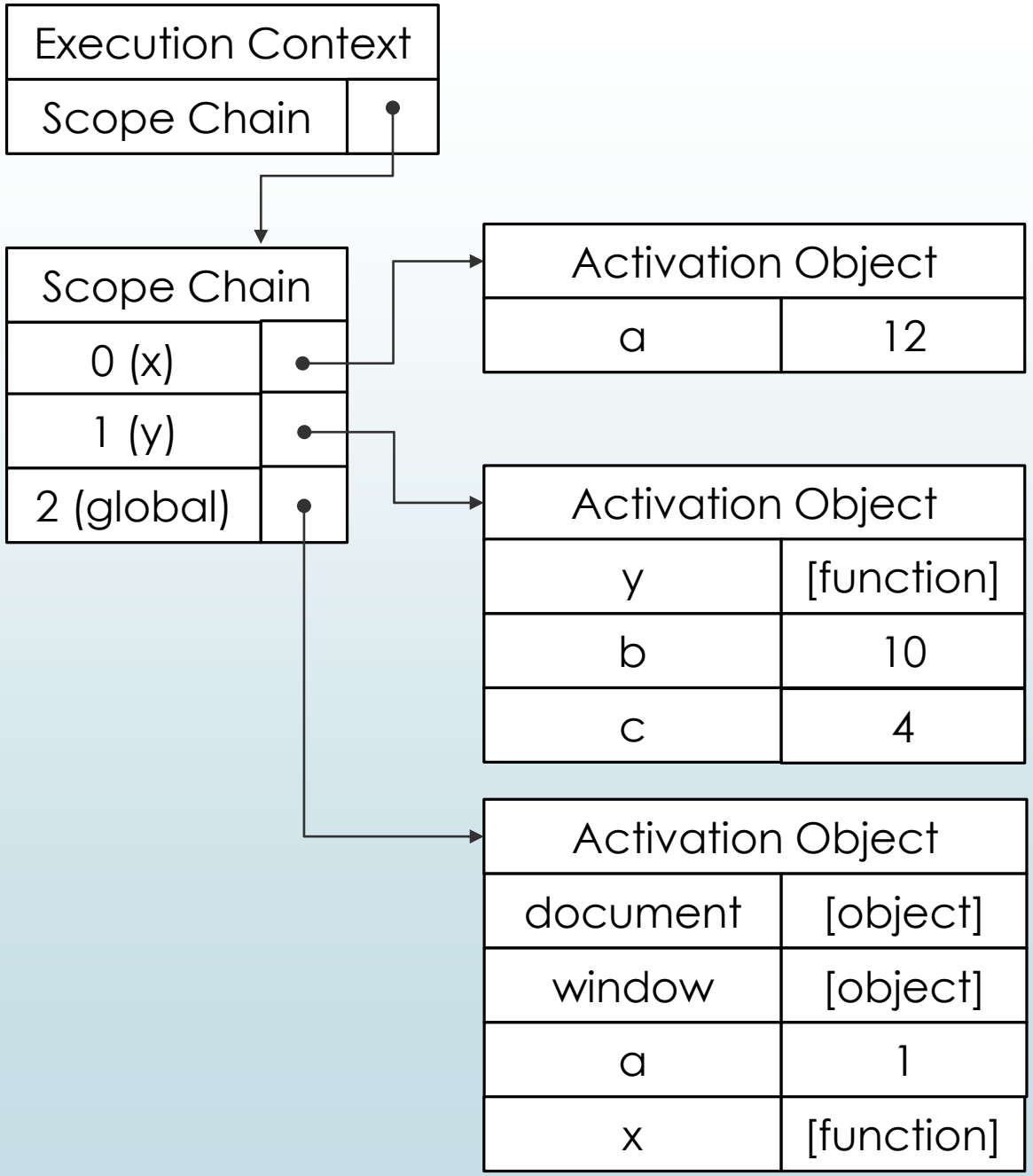
ScopeChain

```
var a = 1;

function x(c) {
    var b = a * 2 + c;

    function y() {
        var a = b * 2;
        b = a-2;
    }

    y();
}
x(4);
```



ScopeChain

```
var a = 1;

function x(c) {
    var b = a * 2 + c;

    function y() {
        var a = b * 2;
        b = a-2;
    }

    y();
}
x(4);
```

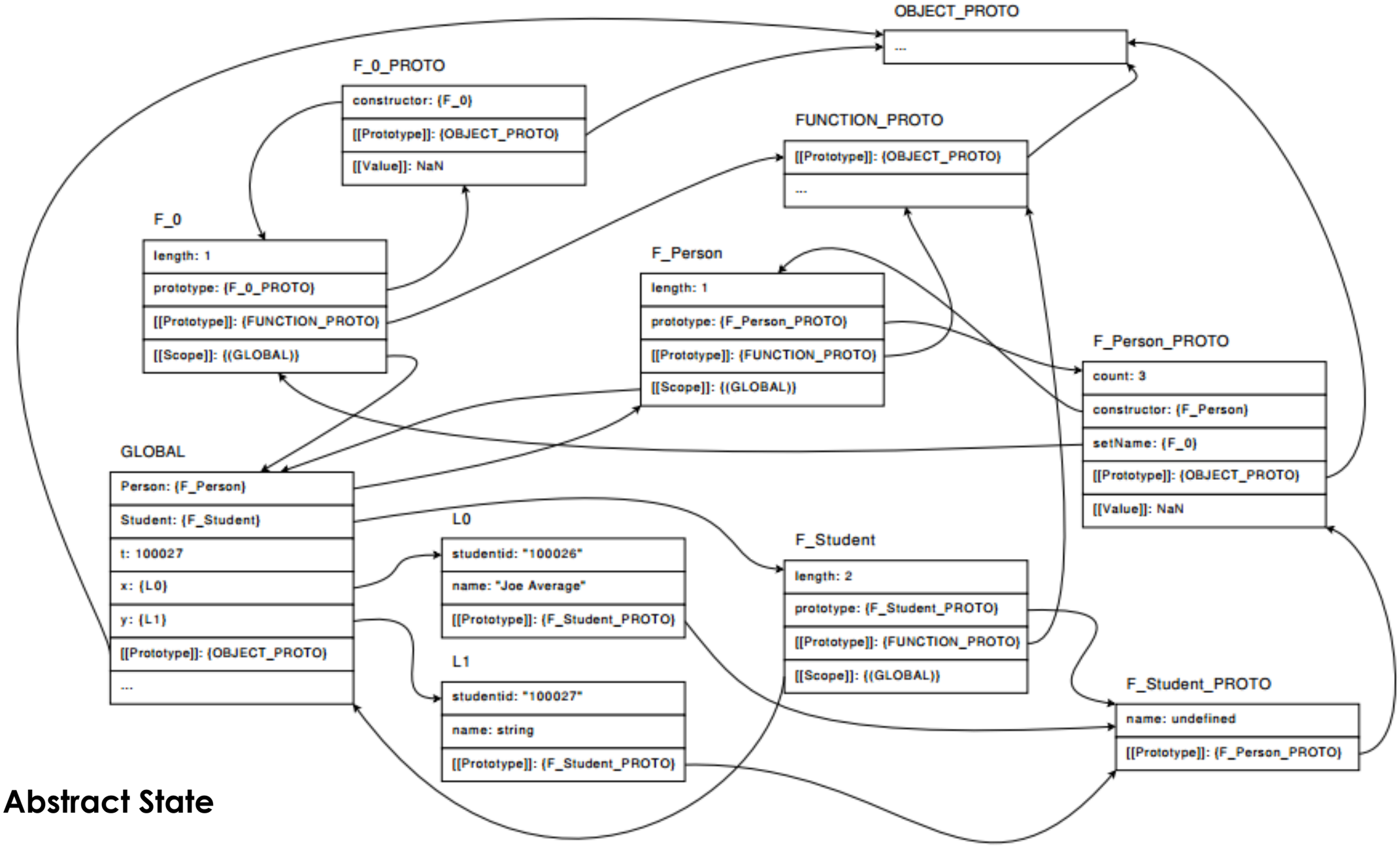


(2) Dataflow Lattice (D)

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

Version Labels (context sensitivity based on `this`)

Nodes



Abstract State

(3) Transfer Functions

- `read-property[ variable, field_name, temp]`

1. Force variable to be an object – if it is many objects, then they all have to be handled.
2. Force field_name to be a string
3. Travel the prototype chain to find the relevant properties and join them
4. Strong update to temp

“Strong” updates?

- Sometimes we are assigning to multiple possible abstract/concrete objects (as distinguished by their allocation site)
- `write-property[` *`variable`*`,` *`field_name`*`,` *`temp`*`]` suffers from this in particular

Recency Abstraction

- Each allocation site L gets two object labels
  - L@ - singleton (most recent)
  - L* - summary (the rest)
- Good for handling an allocation site in a loop or a call (flow-sensitive only for the latter)

- Some extra complexity required to track this!

Intra-procedural

```
call[ function, this, variable_1, ..., variable_n], after_call
```

1. Extract all function objects
2. Add call edges to the entry nodes
3. Add return edges to their exit nodes back in
4. Add exception edges

Boundedness

1. ScopeChain is bounded by the lexical depth
2. |abstract states| is based on context-sensitivity criteria
3. Object map is not a problem because of `default_index/default_other`.

- Result: The worst case may be bad, *but who programs like that?*

Testing

- 150 small programs (5-50 lines)
- Detects all errors, provides type information

Evaluation

- Google V8 Benchmark test suite
- *"we measure precision by counting the number of operations where the analysis does not produce a warning (for different categories), i.e. is capable of proving that the error cannot occur at that point"*

	lines	call / construct	variable read	property access	fixed-property read
richards.js	529	95%	100%	93%	87%
benchpress.js	463	100%	100%	89%	100%
delta-blue.js	853	78%	100%	82%	61%
3d-cube.js	342	100%	100%	92%	100%
3d-raytrace.js	446	99%	100%	94%	94%
crypto-md5.js	291	100%	100%	100%	100%
access-nbody.js	174	100%	100%	93%	100%

Fig. 2. Analysis precision.

Not listed – "cryptobench.js"
Causes OutOfMemory!

In most cases, the false positives appear to be caused by the lack of path sensitivity.

Conclusion

- Sound, detailed tool for type analysis in JS
- Monotone framework with lattice and transfer functions
- Recency Abstraction helps for some kinds of analyses

Future Work

- "We envision an IDE for JavaScript programming with features known from strongly typed languages, such as highlighting of type-related errors and support for precise content assists and safe refactorings"

- Modeling DOM