Dependent Types for JavaScript

Ravi Chugh Ranjit Jhala

University of California, San Diego

David Herman

Mozilla Research

Why JavaScript?

Pervasive

Used at
Massive Scale

Why Add Types?

```
var x = {};
x.f;
x.f.f;
```

produces `undefined` rather than error…

… but this raises `TypeError`

Why Add Types?

```
var x = {};
x.f;
x.f.f;
```

There **Are** Run-time Errors

reading from `undefined`, applying non-function values, …

Worse: Browsers Hide Them!

Why Add Types?

Prevent Errors

Prevent the Unexpected

Okay, But Who Cares?

Programmers

JSLint, Closure Compiler, TypeScript, Dart

Browser Vendors

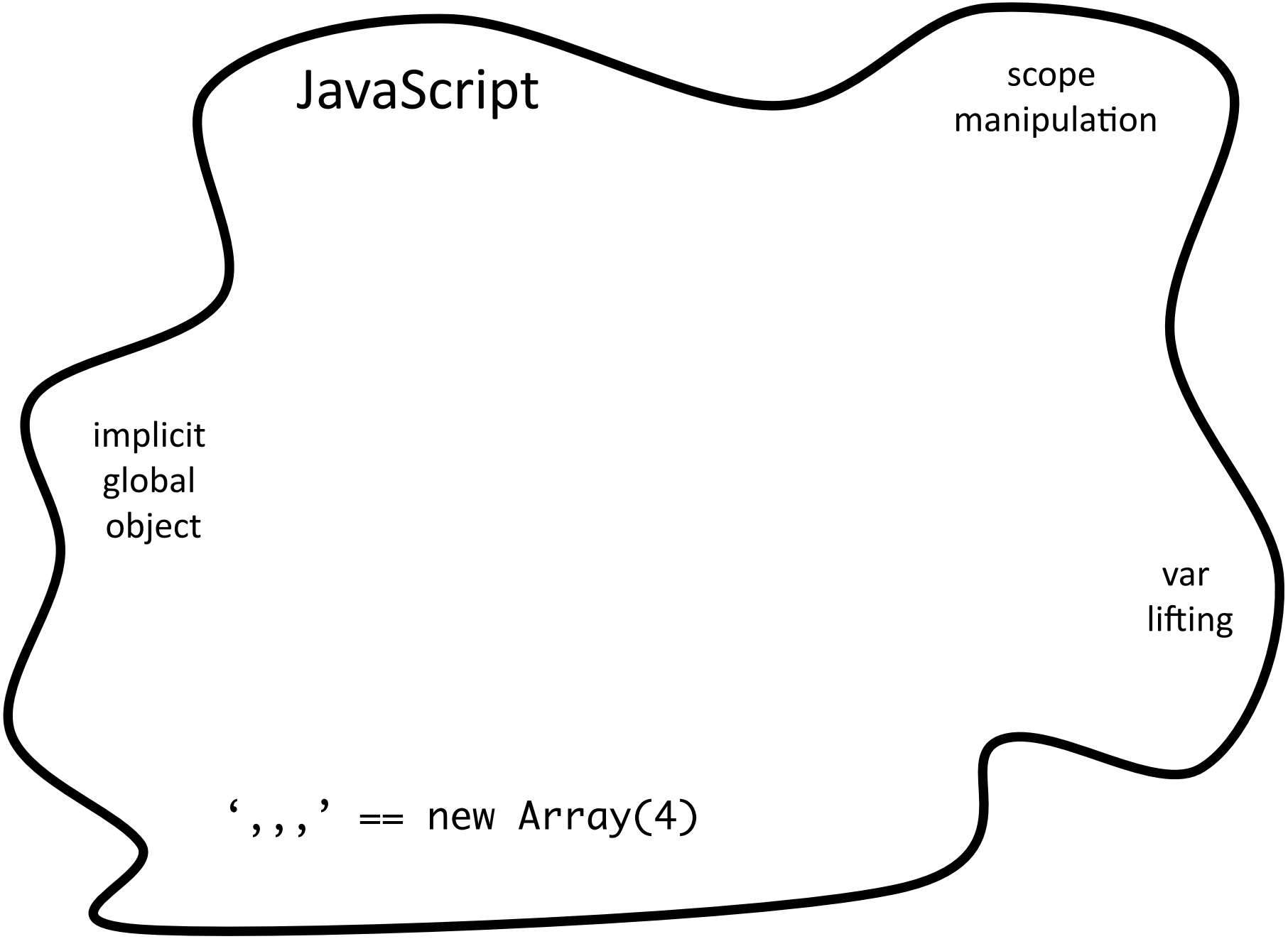
Compete based on performance, security, reliability

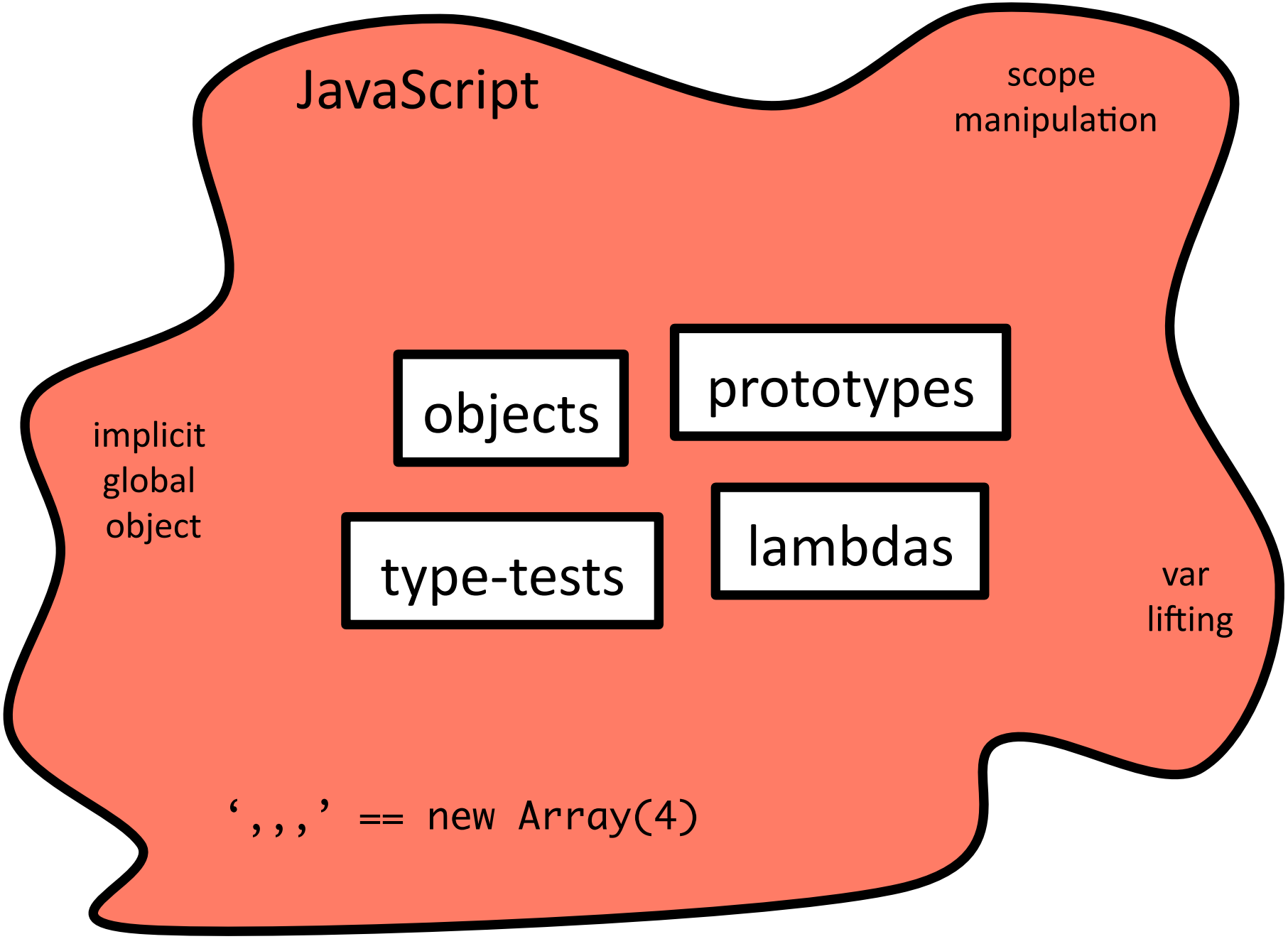
Standards Committees

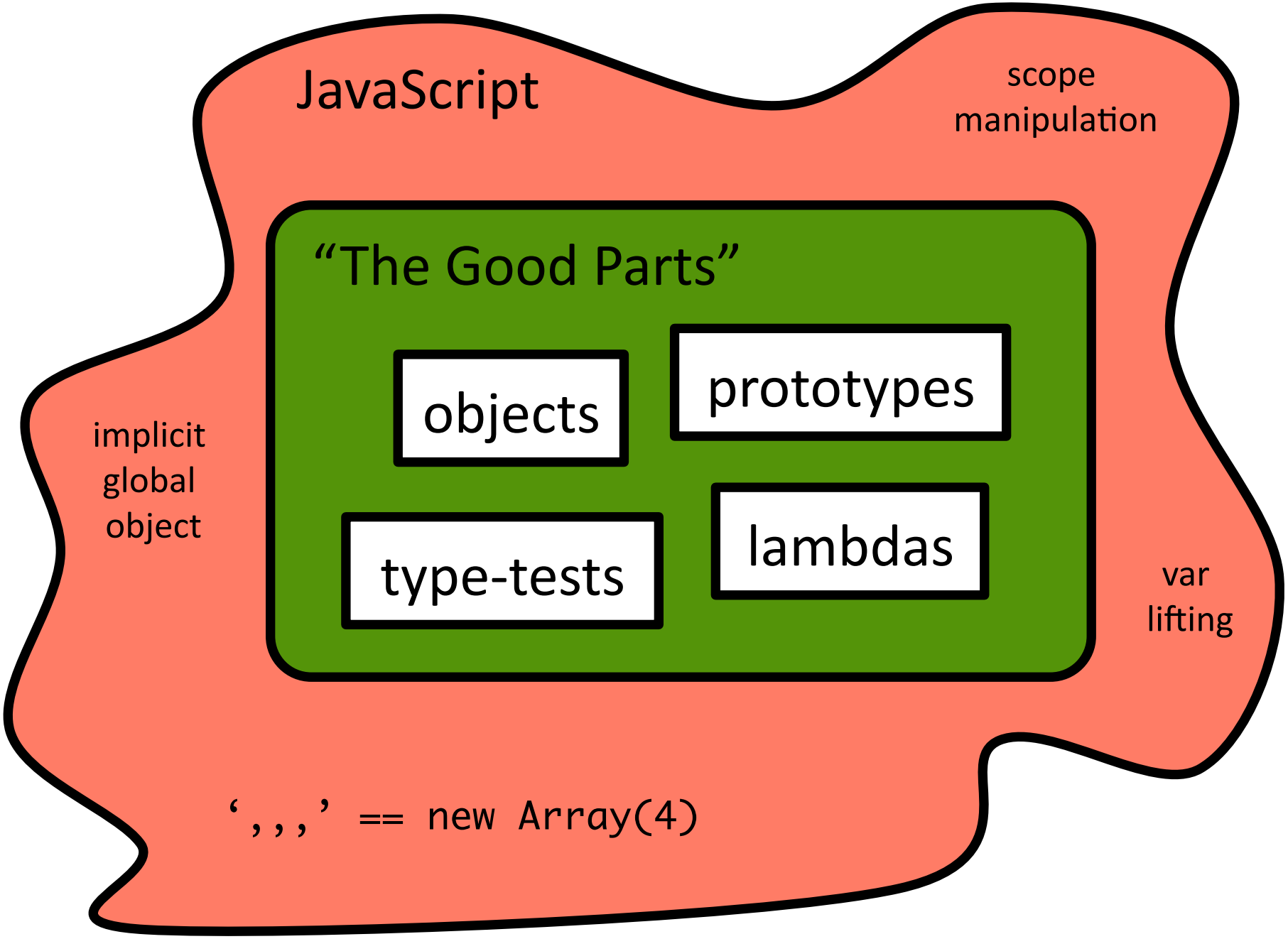
Actively evolving JavaScript and Web standards

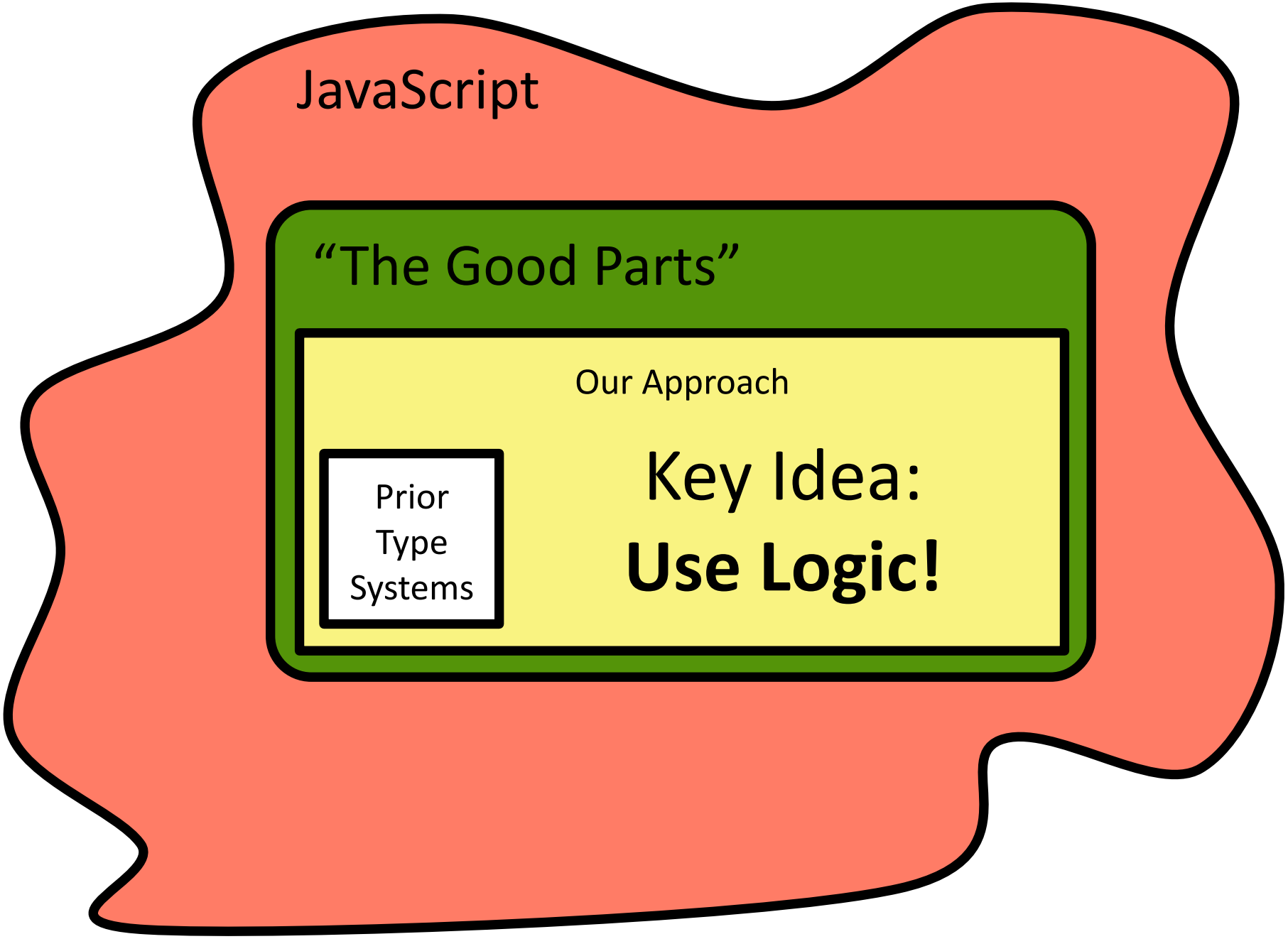
Why JavaScript?

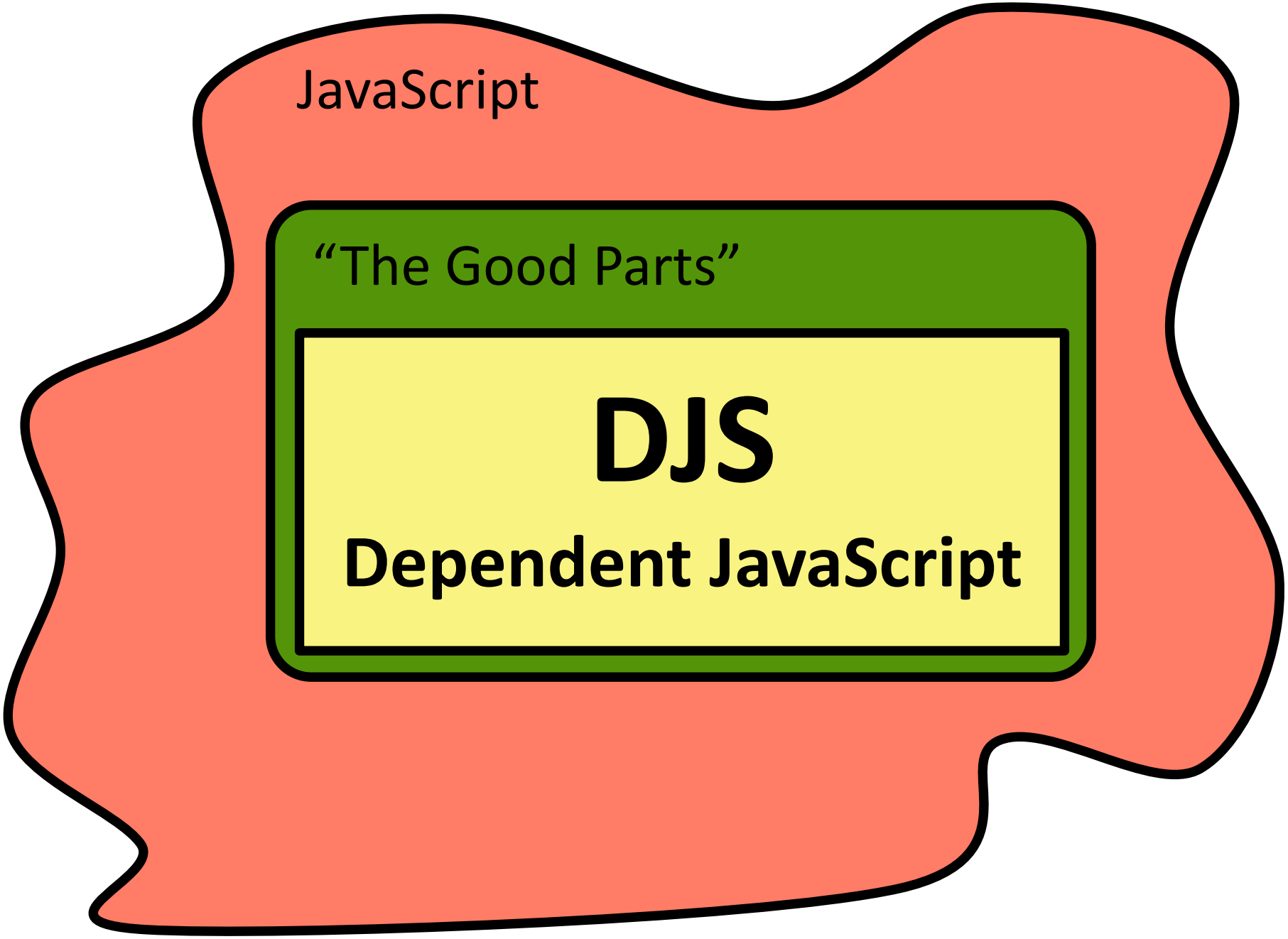
Isn’t the Language Terrible?


JavaScript
scope
manipulation
implicit
global
object
var
lifting
',,,' == new Array(4)


JavaScript
scope
manipulation
objects
prototypes
implicit
global
object
type-tests
lambdas
var
lifting
',,,' == new Array(4)


JavaScript
scope manipulation
"The Good Parts"
objects
prototypes
type-tests
lambdas
implicit global object
var lifting
',,,' == new Array(4)


JavaScript
“The Good Parts”
Our Approach
Prior Type Systems
Key Idea:
Use Logic!


JavaScript
“The Good Parts”
DJS
Dependent JavaScript

Outline

Motivation

Our Approach: Logic!

Evaluation

```
typeof true  // “boolean”

typeof 0.1   // “number”
typeof 0     // “number”

typeof {}    // “object”
typeof []    // “object”
typeof null  // “object”
```

`typeof` returns run-time **“tags”**

Tags are very coarse-grained **types**

“`undefined`”

“`boolean`”

“`string`”

“`number`”

“`object`”

“`function`”

Refinement Types

$$\{ x \mid p \}$$

"set of values x s.t. formula p is true"

```
      Num ≡ { n | tag(n) = "number" }
NumOrBool ≡ { v | tag(v) = "number" ∨ tag(v) = "boolean" }
      Int ≡ { i | tag(i) = "number" ∧ integer(i) }
      Any ≡ { x | true }
```

Refinement Types

Syntactic Sugar
for Common Types

$$
\begin{aligned}
\texttt{Num} &\equiv \{\, \texttt{n} \mid \texttt{tag}(\texttt{n}) = \text{“number”} \,\} \\
\texttt{NumOrBool} &\equiv \{\, \texttt{v} \mid \texttt{tag}(\texttt{v}) = \text{“number”} \vee \texttt{tag}(\texttt{v}) = \text{“boolean”} \,\} \\
\texttt{Int} &\equiv \{\, \texttt{i} \mid \texttt{tag}(\texttt{i}) = \text{“number”} \wedge \texttt{integer}(\texttt{i}) \,\} \\
\texttt{Any} &\equiv \{\, \texttt{x} \mid \texttt{true} \,\}
\end{aligned}
$$

Refinement Types

```
3 :: { n | n = 3 }
3 :: { n | n > 0 }
3 :: { n | tag(n) = "number" ∧ integer(n) }
3 :: { n | tag(n) = "number" }
```

Refinement Types

Subtyping is Implication

```
   { n | n = 3 }
<: { n | n > 0 }
<: { n | tag(n) = "number" ∧ integer(n) }
<: { n | tag(n) = "number" }
```

Refinement Types

Subtyping is Implication

$$
\begin{aligned}
& \texttt{n} = 3 \\
\Rightarrow\quad & \texttt{n} > 0 \\
\Rightarrow\quad & \texttt{tag(n)} = \text{“number”} \wedge \texttt{integer(n)} \\
\Rightarrow\quad & \texttt{tag(n)} = \text{“number”}
\end{aligned}
$$

Tag-Tests	Duck Typing	Mutable Objects	Prototypes	Arrays

```
var negate = function(x) {
  if (typeof x == "boolean")
    return !true // false
  else
    return 0 - x;
}
negate( true )
```

```
var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - 2    // -2
}
negate( 2 )
```

```
var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - []   // 0
}
negate( [] )
```

WAT?!

```
var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - x;
}
```

Use types to prevent implicit coercion

(-) :: (Num,Num) → Num

```
//: negate :: (x:Any) → Any
var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - x;
}
```

Function type annotation inside comments

```
//: negate :: (x:Any) → Any
var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - x;
}
```

x is boolean...
so negation
is well-typed

DJS is Path Sensitive

```
//: negate :: (x:Any) → Any
var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - x;
}
```

x is arbitrary non-boolean value… so DJS signals error!

DJS is Path Sensitive

```
//: negate :: (x:NumOrBool) → Any
var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - x;
}
```

```
//: negate :: (x:NumOrBool) → Any
var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - x;
}
```

this time,
x is a number…
so subtraction
is well-typed

```
//: negate :: (x:NumOrBool) → Any
var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - x;
}
```

but return
type is imprecise

```
//: negate :: (x:NumOrBool) → NumOrBool

var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - x;
}
```

```
/*: negate :: (x:NumOrBool)
            → { v | tag(v) = tag(x) } */
var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - x;
}
```

output type **depends** on input value

```
/*: negate :: (x:NumOrBool)
            → { v | tag(v) = tag(x) } */
```

Programmer chooses
degree of precision
in specification

What is “Duck Typing”?

```
if (duck.quack)
  return “Duck says ” + duck.quack();
else
  return “This duck can’t quack!”;
```

What is “Duck Typing”?

```
(+) :: (Num,Num) → Num
(+) :: (Str,Str) → Str
```

```
if (duck.quack)
  return “Duck says ” + duck.quack();
else
  return “This duck can’t quack!”;
```

What is “Duck Typing”?

Can dynamically test
the **presence** of a method
but not its **type**

```
if (duck.quack)
  return “Duck says ” + duck.quack();
else
  return “This duck can’t quack!”;
```

$$\{ d \mid tag(d) = \text{“Dict”} \wedge \boxed{has(d, \text{“quack”}) \Rightarrow sel(d, \text{“quack”})} :: Unit \rightarrow Str \}$$

Operators from McCarthy theory of arrays

```
if (duck.quack)
  return “Duck says ” + duck.quack();
else
  return “This duck can’t quack!”;
```

$\{ d \mid tag(d) = \text{“Dict”} \wedge$
$\quad has(d, \text{“quack”}) \Rightarrow$
$\quad\quad sel(d, \text{“quack”}) :: Unit \rightarrow Str \}$

Call produces `Str`, so concat well-typed

```
if (duck.quack)
  return “Duck says ” + duck.quack();
else
  return “This duck can’t quack!”;
```

DJS is Flow Sensitive

```
var x = {};
x.f = 7;
x.f + 2;
```

x_0:Empty

x_1:{dld = upd(x_0,"f",7)}

McCarthy operator

DJS verifies that `x.f` is definitely a number

DJS is Flow Sensitive

```
var x = {};
x.f = 7;
x.f + 2;
```

x_0:Empty

x_1:{dld = upd(x_0,"f",7)}

Strong updates to singleton objects

Weak updates to collections of objects

Tag-Tests	Duck Typing	Mutable Objects	Prototypes	Arrays

Tag-Tests	Duck Typing	Mutable Objects	Prototypes	Arrays

Typical "Dynamic" Features

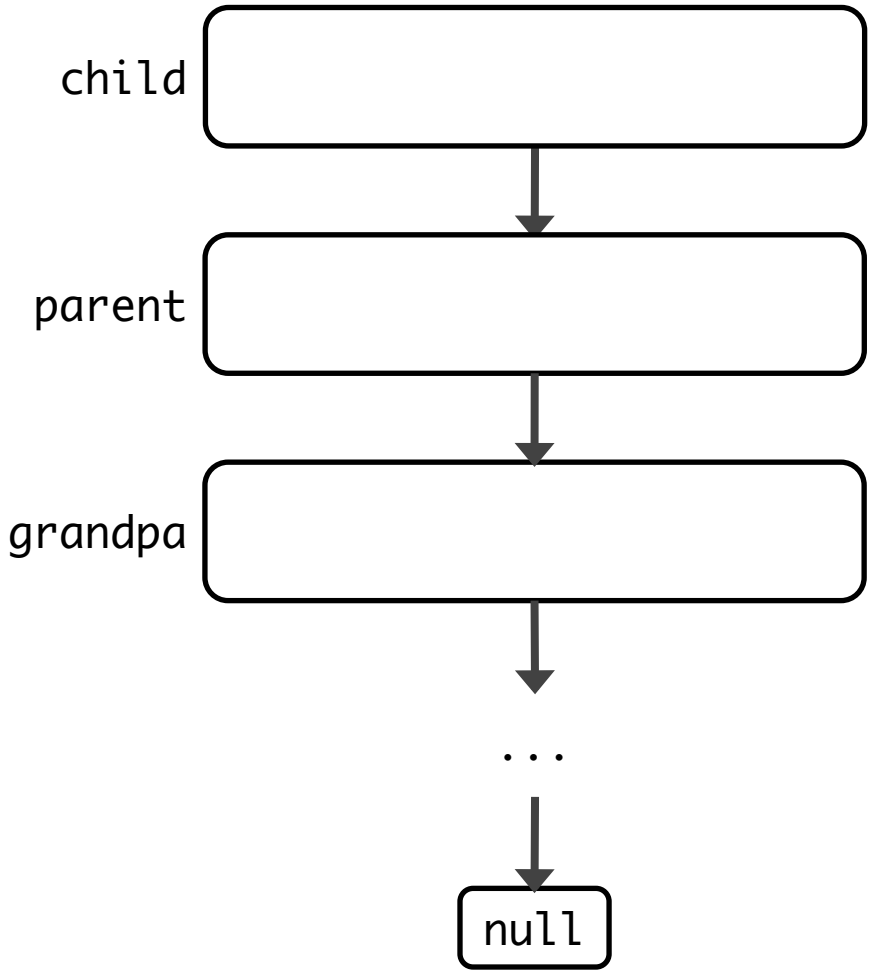
Tag-Tests	Duck Typing	Mutable Objects	Prototypes	Arrays

Typical "Dynamic" Features

JavaScript

`tl;dr`

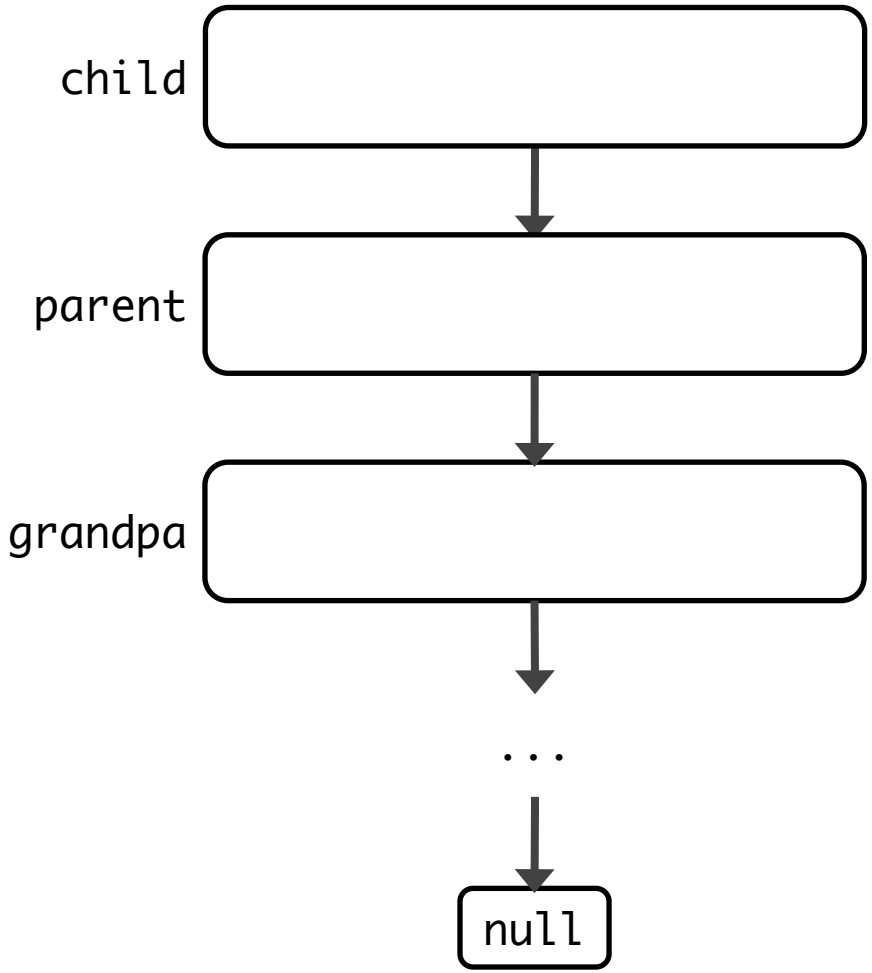
Harder, but
DJS handles them



Upon construction, each object links to a **prototype** object

Semantics of Key Lookup

`child[k];`



If `child` contains `k`, then
  Read `k` from `child`

Else if `parent` contains `k`, then
  Read `k` from `parent`

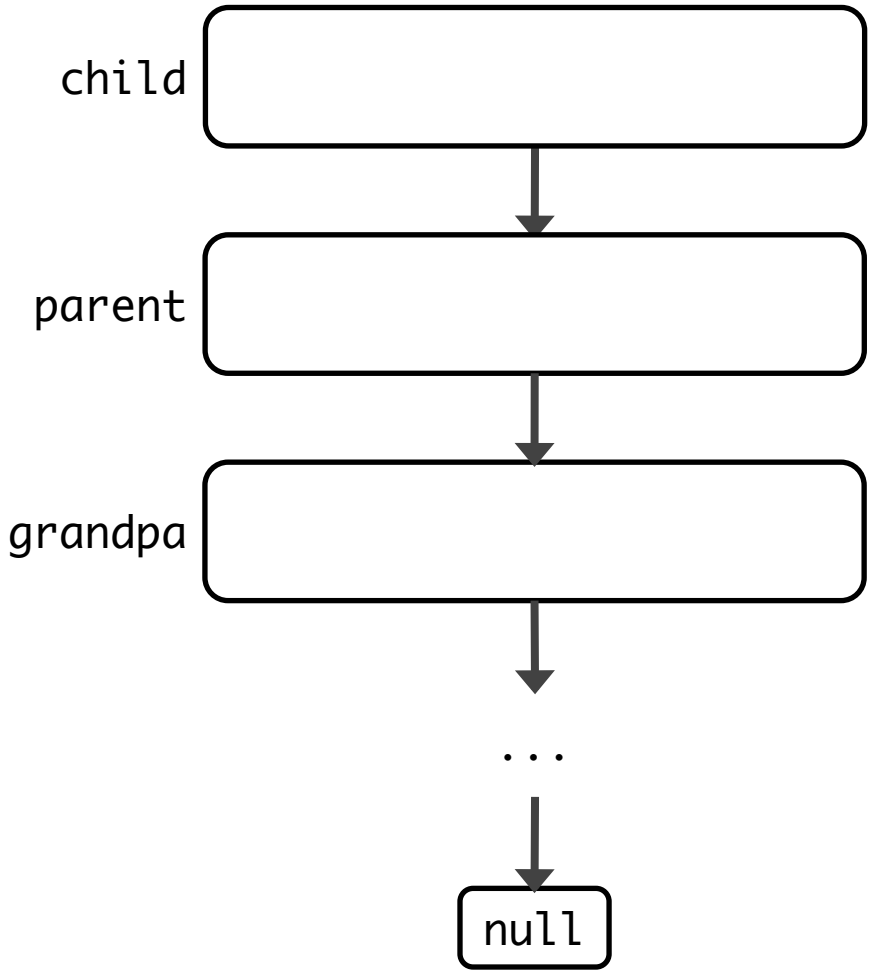
Else if `grandpa` contains `k`, then
  Read `k` from `grandpa`

Else if ...

Else
  Return `undefined`

Semantics of Key Lookup

`child[k];`




```
{v | if has(child,k) then
        v = sel(child,k)
```

Else if `parent` contains `k`, then
  Read `k` from `parent`

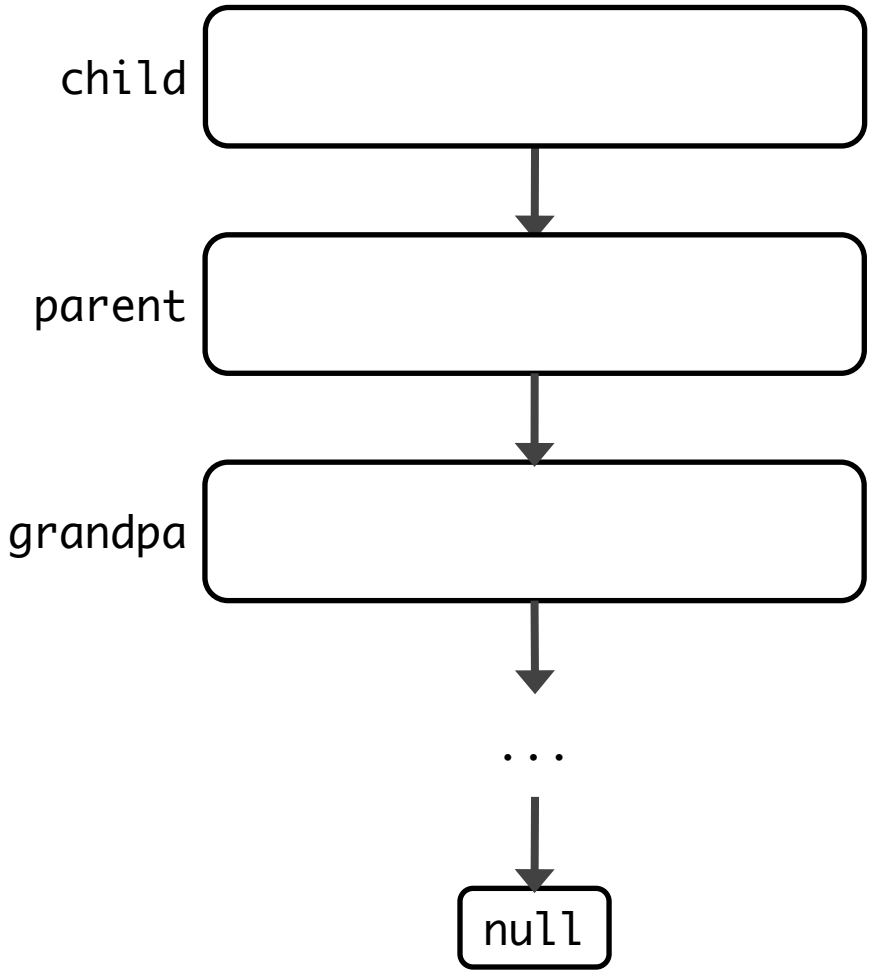
Else if `grandpa` contains `k`, then
  Read `k` from `grandpa`

Else if ...

Else
  Return `undefined`

Semantics of Key Lookup

`child[k];`




```
{v | if has(child,k) then
       v = sel(child,k)

     else if has(parent,k) then
       v = sel(parent,k)
```

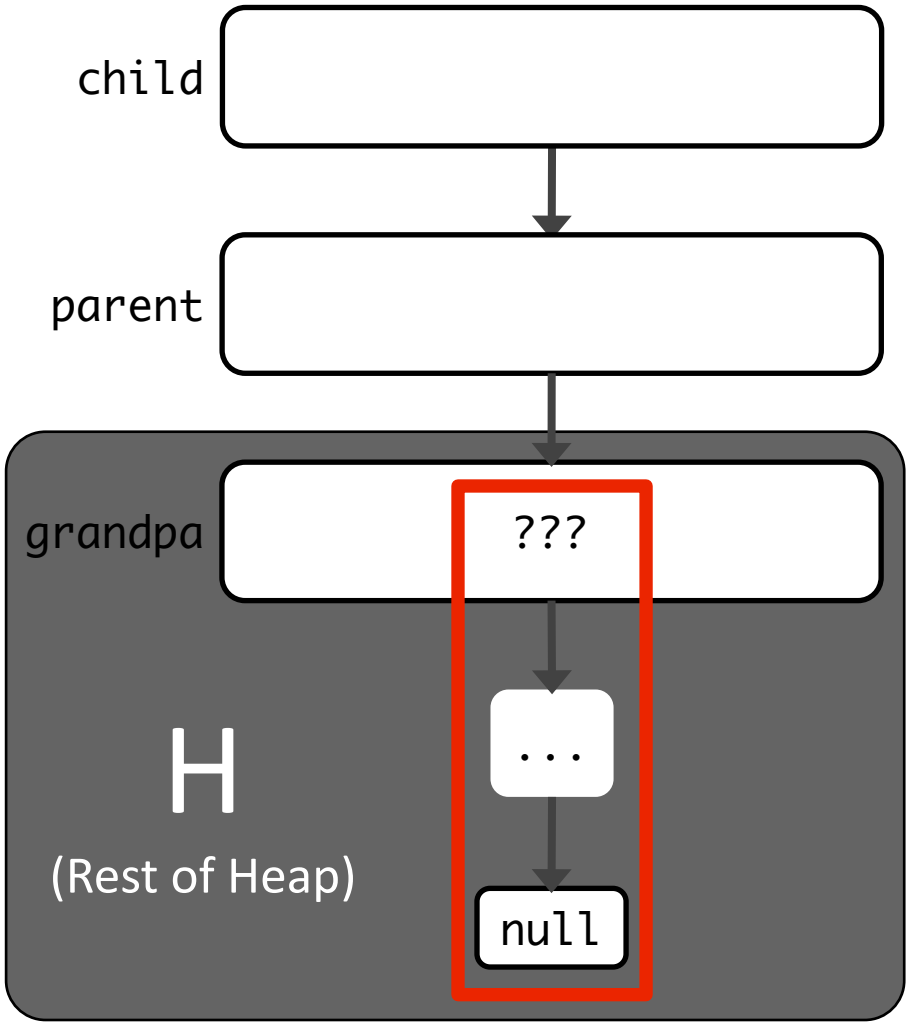
Else if grandpa contains k, then
  Read k from grandpa

Else if ...

Else
  Return undefined

Semantics of Key Lookup

`child[k];`




```
{v | if has(child,k) then
       v = sel(child,k)

     else if has(parent,k) then
       v = sel(parent,k)
```

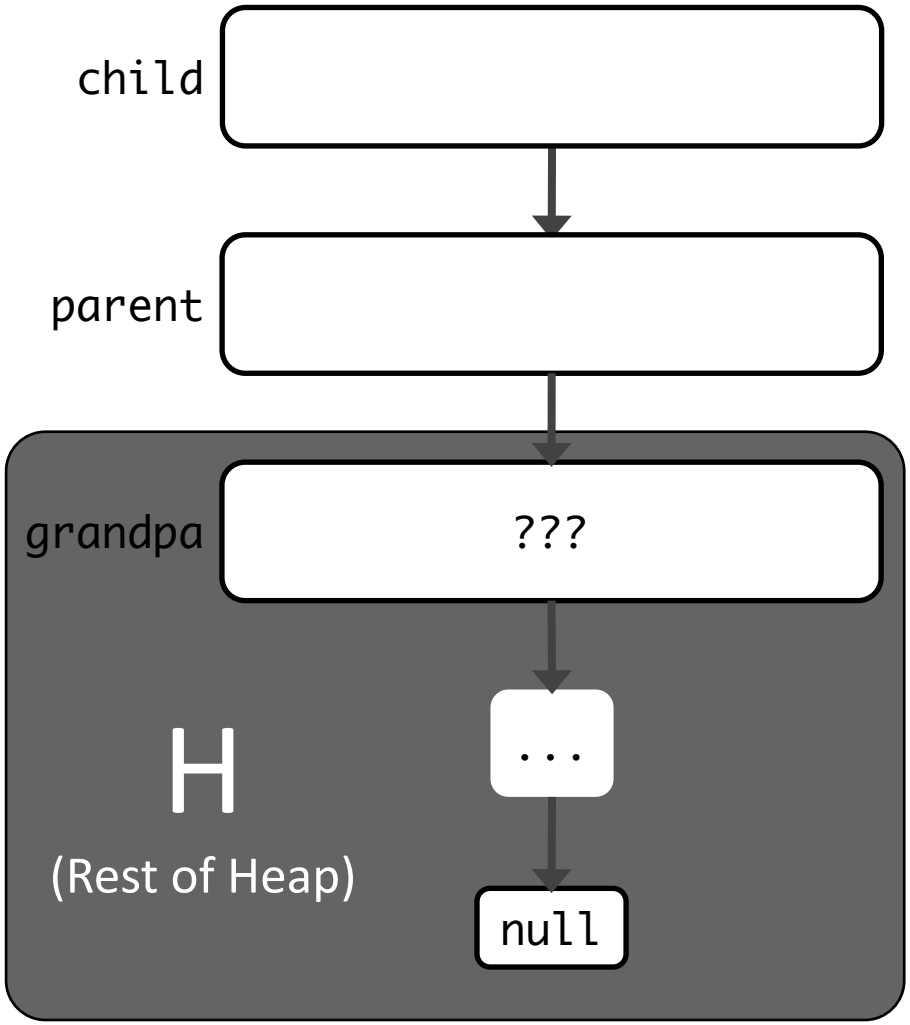
Else if grandpa contains k, then
  Read k from grandpa

Else if ...

Else
  Return undefined

Semantics of Key Lookup

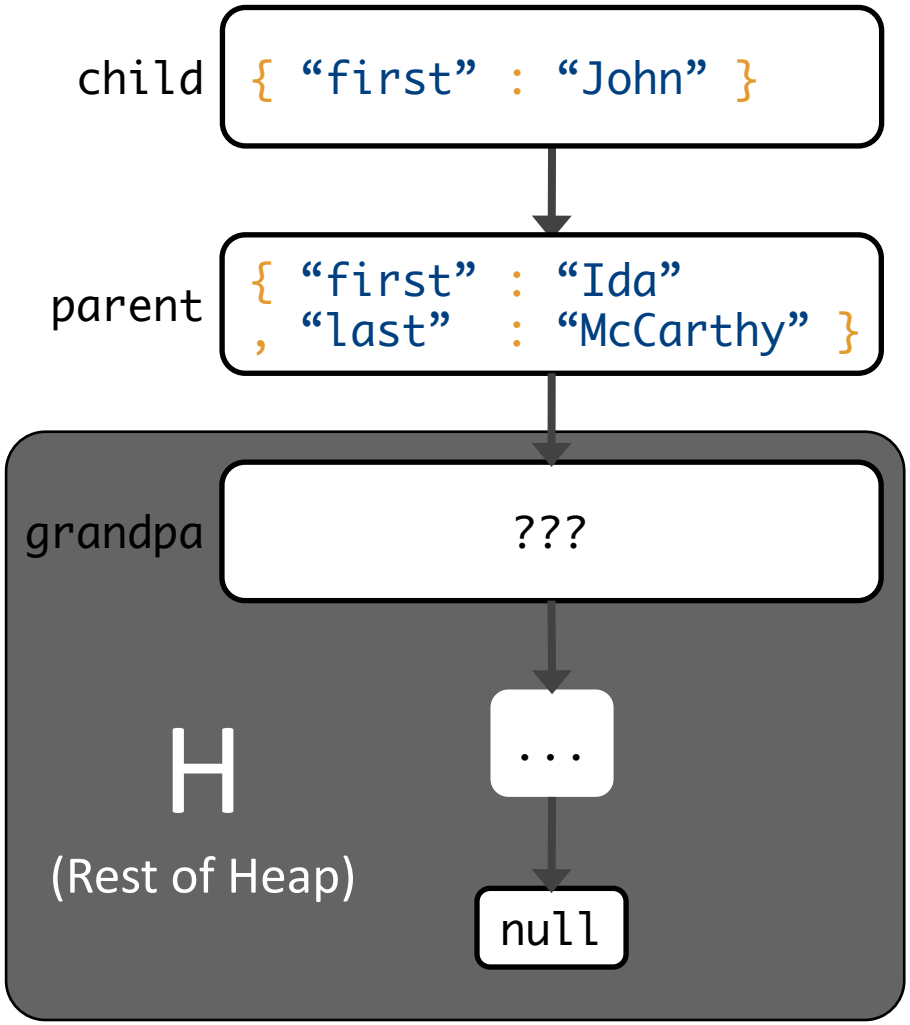
```
child[k];
```




```
{v | if has(child,k) then
       v = sel(child,k)

     else if has(parent,k) then
       v = sel(parent,k)

     else
       v = HeapSel(H,grandpa,k))}
```

Abstract predicate
to summarize the
unknown portion
of the prototype chain

```
var k = "first"; child[k];
```

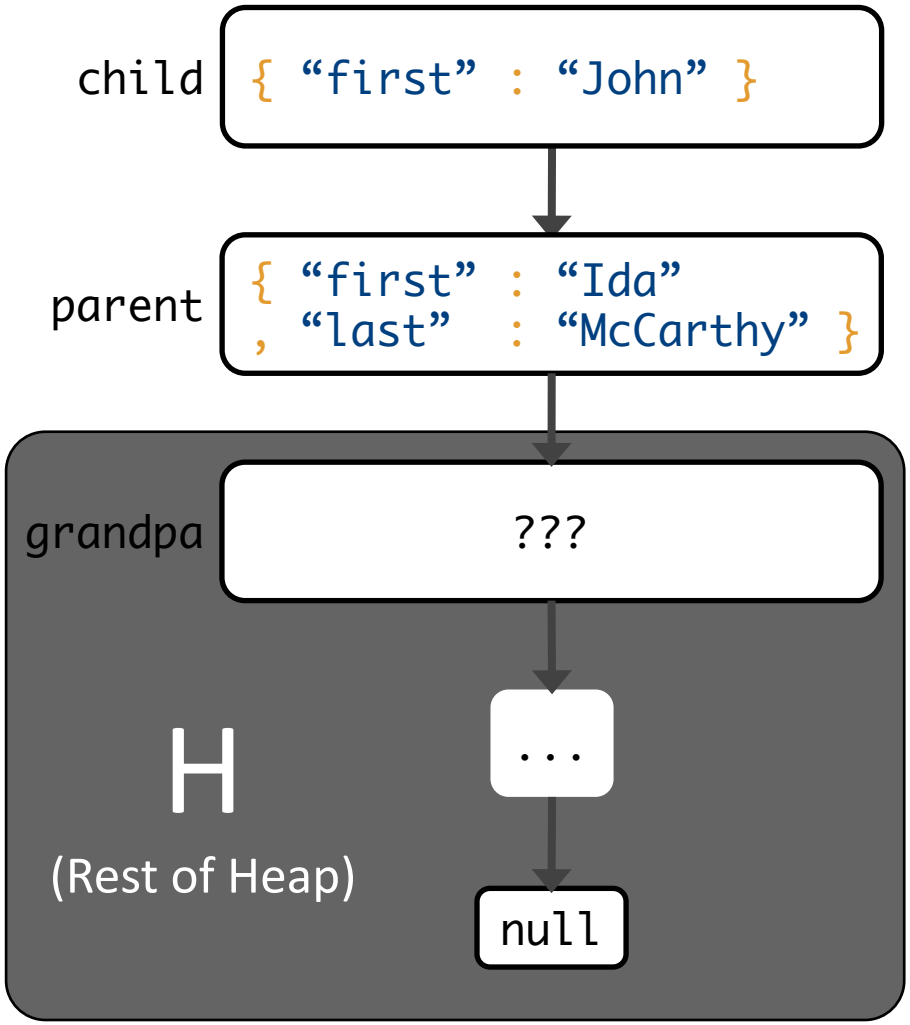



```
{ v  if has(child,k) then
       v = sel(child,k)

     else if has(parent,k) then
       v = sel(parent,k)

     else
       v = HeapSel(H,grandpa,k)) }
```

<:

{ v | v = "John" }

```
var k = "last"; child[k];
```




```
{ v  if has(child,k) then
       v = sel(child,k)
     else if has(parent,k) then
       v = sel(parent,k)
     else
       v = HeapSel(H,grandpa,k)) }
```

<:

{ v | v = "McCarthy" }

Prototype Chain Unrolling

Key Idea:

Reduce prototype semantics to **decidable** theory of arrays

```
var nums = [0,1,2]
while (…) {
  nums[nums.length] = 17
}
```

A finite tuple…

… extended to unbounded collection

```
var nums = [0,1,2]
while (…) {
  nums[nums.length] = 17
}

delete nums[1]

for (i = 0; i < nums.length; i++)
  sum += nums[i]
```

A “hole” in the array

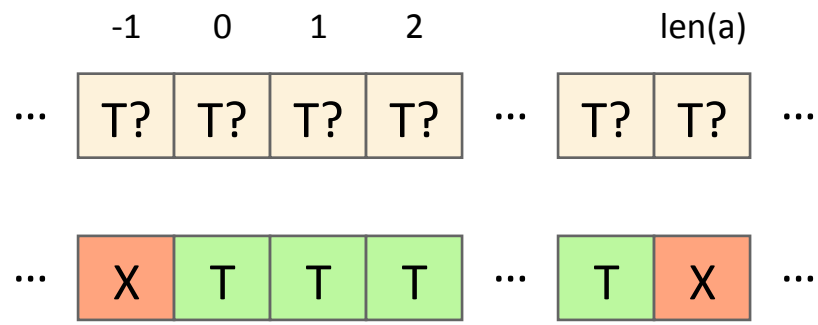
Missing element within “length”

Track **types**, "**packedness**," and **length** of arrays where possible

$\{ a \mid a :: \mathtt{Arr}(T)$

$\wedge\ \mathtt{packed}(a)$

$\wedge\ \mathtt{len}(a) = 10 \}$



$T? \equiv \{ x \mid T(x) \vee x = \mathtt{undefined} \}$

$X \equiv \{ x \mid x = \mathtt{undefined} \}$

Encode **tuples** as arrays

```
var tup = [17, “cacti”]
```

$$\{ a \mid a :: \mathrm{Arr}(\mathrm{Any}) \wedge \mathrm{packed}(a) \wedge \mathrm{len}(a) = 2 \wedge \mathrm{Int}(\mathrm{sel}(a,0)) \wedge \mathrm{Str}(\mathrm{sel}(a,1)) \}$$

```
var tup = [17, “cacti”]
tup[tup.length] = true
```

$$\{ a \mid a :: \mathrm{Arr}(\mathrm{Any}) \wedge \mathrm{packed}(a) \wedge \mathrm{len}(a) = 3 \wedge \ldots \}$$

DJS handles other **quirks**:

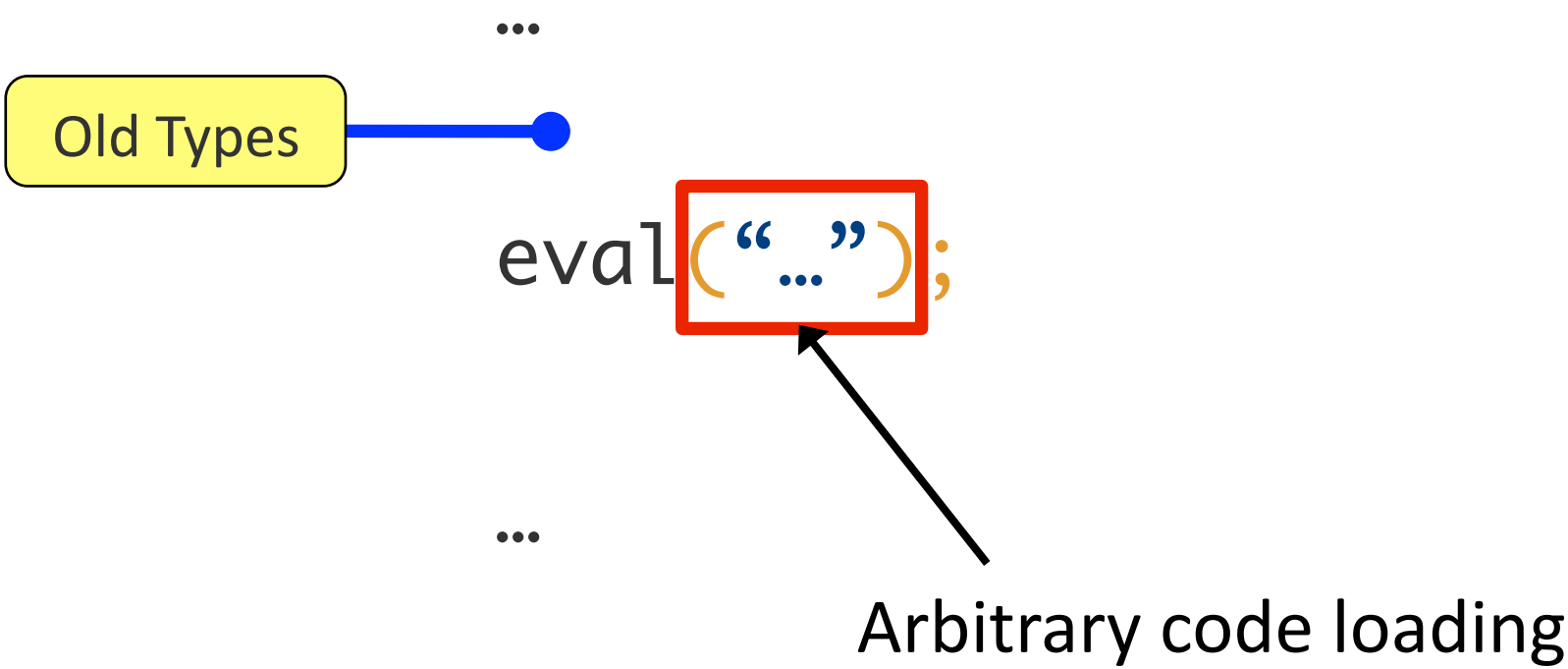
Special `length` property

`Array.prototype`

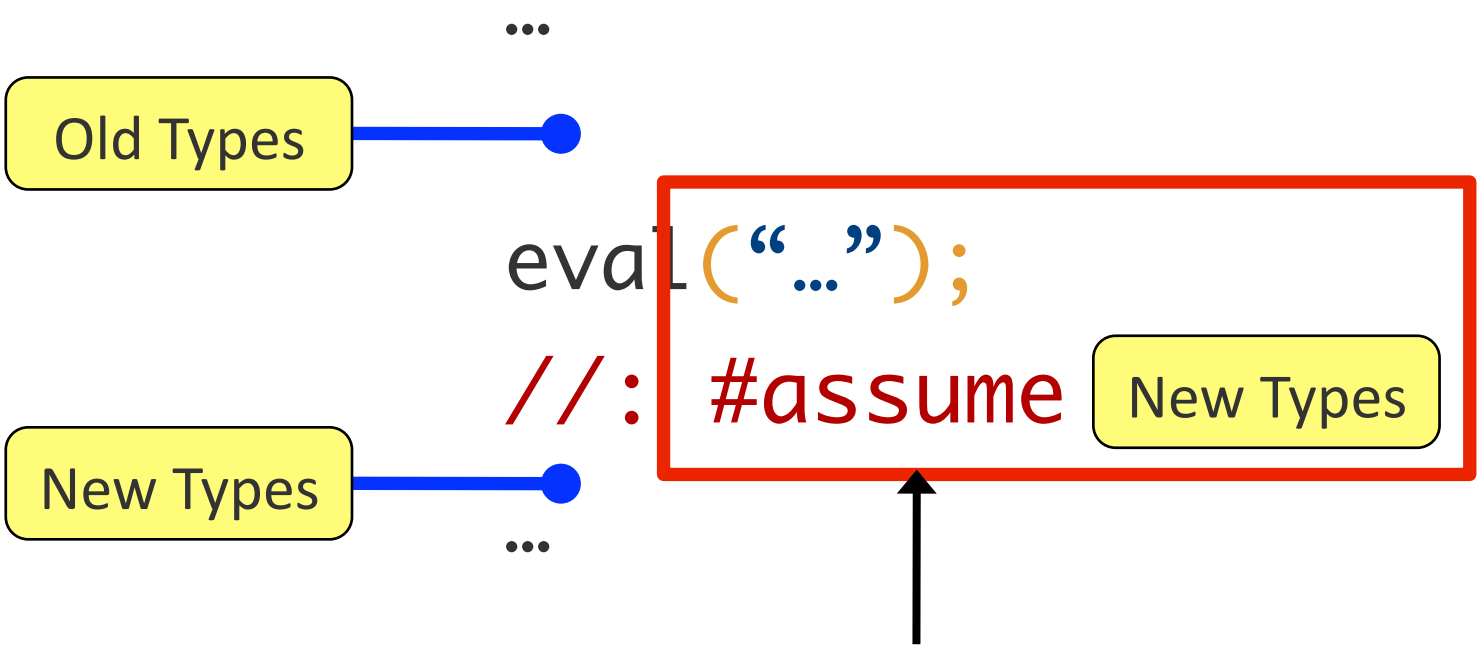
Non-integer keys

Tag-Tests	Duck Typing	Mutable Objects	Prototypes	Arrays

What About `eval`?



What About eval?



"Contract Checking" at Run-time
aka "Gradual Typing"

Recap of DJS Techniques

Logic!

Path and Flow Sensitivity

Prototype Unrolling

Syntactic Sugar

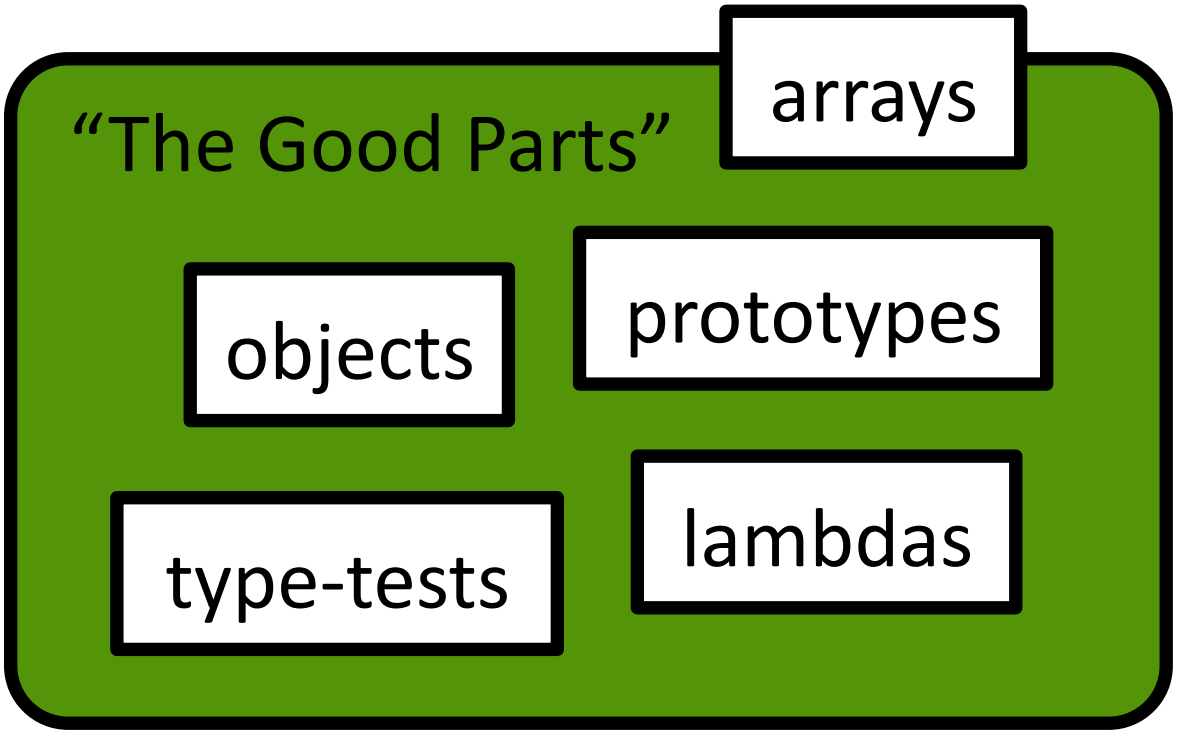
Outline

Motivation

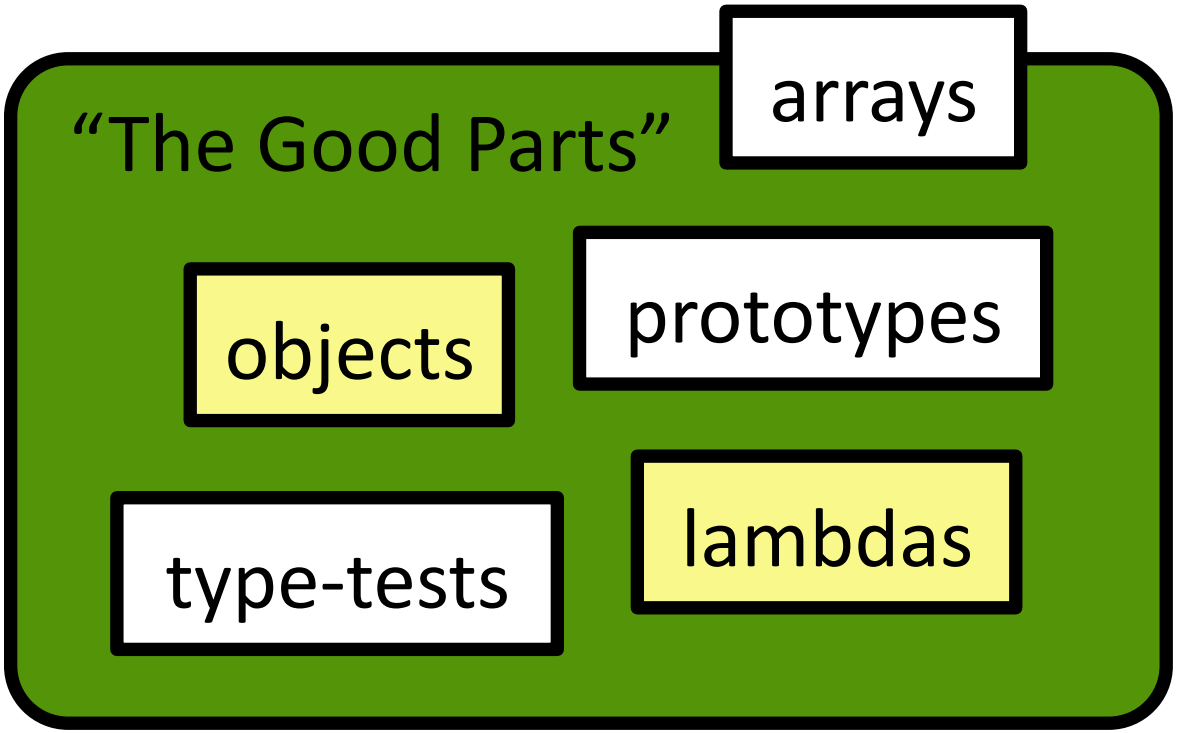
Our Approach: Logic!

Evaluation

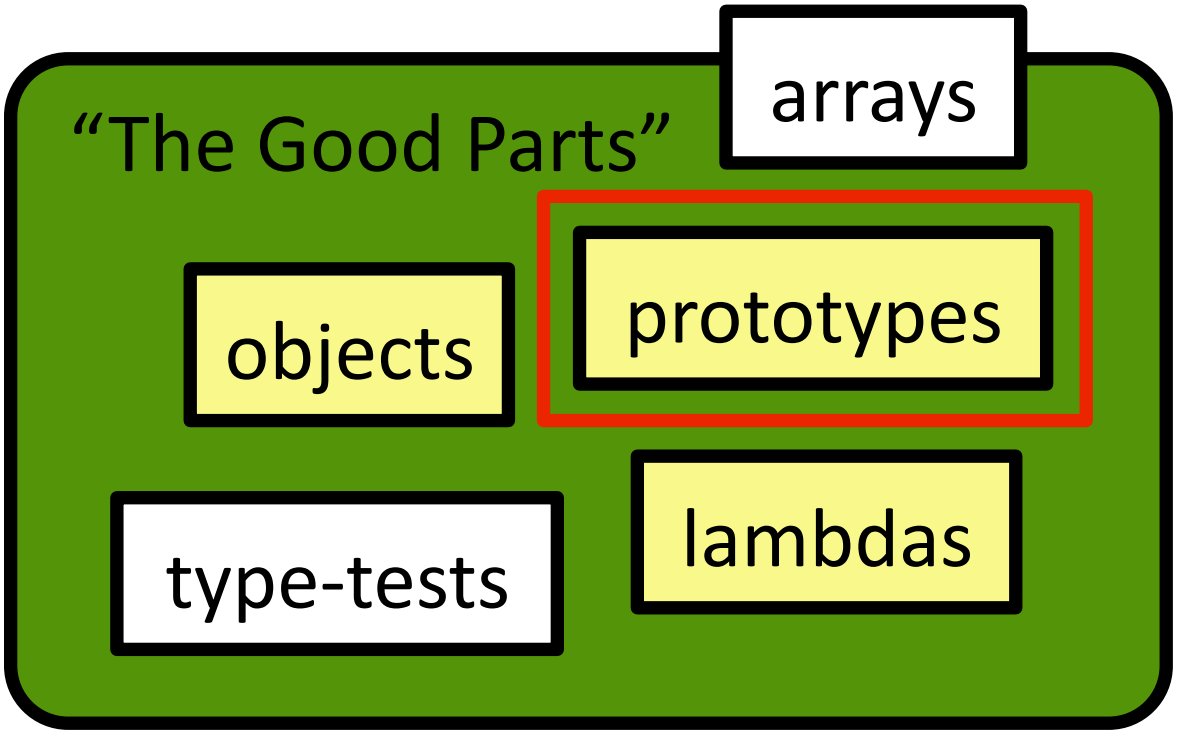
13 Benchmarks



13 Benchmarks

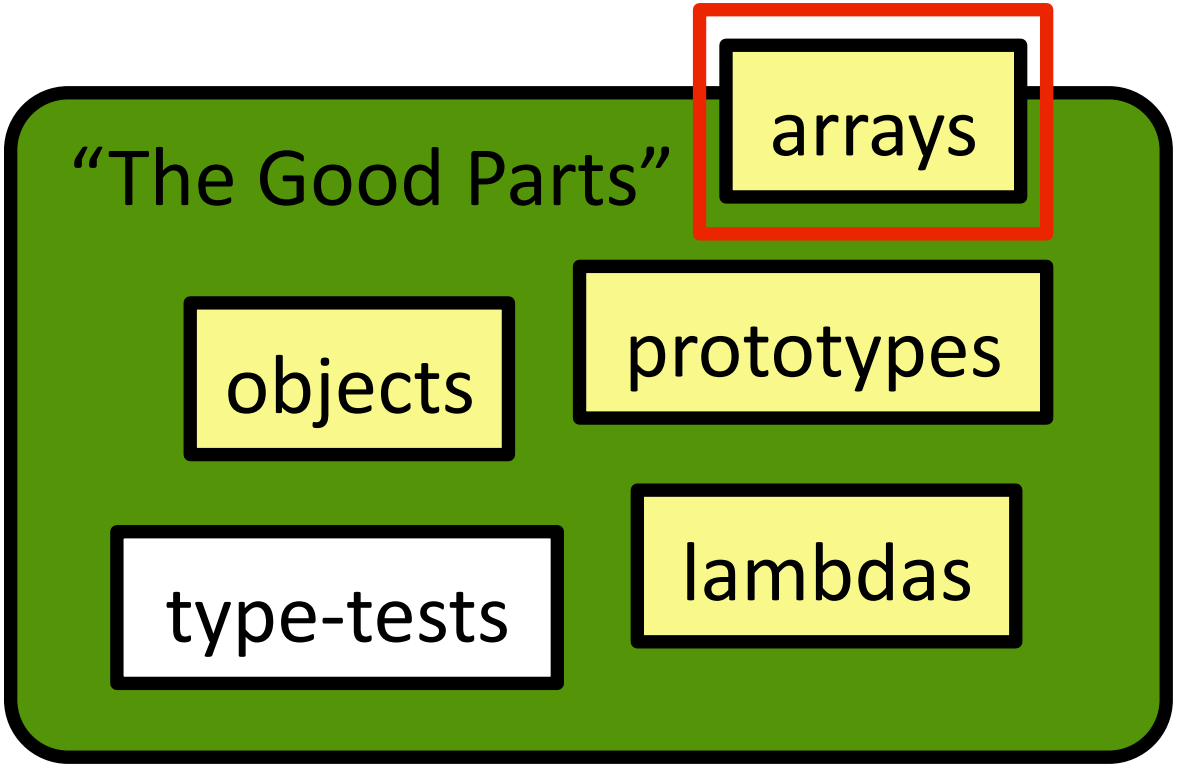


13 Benchmarks



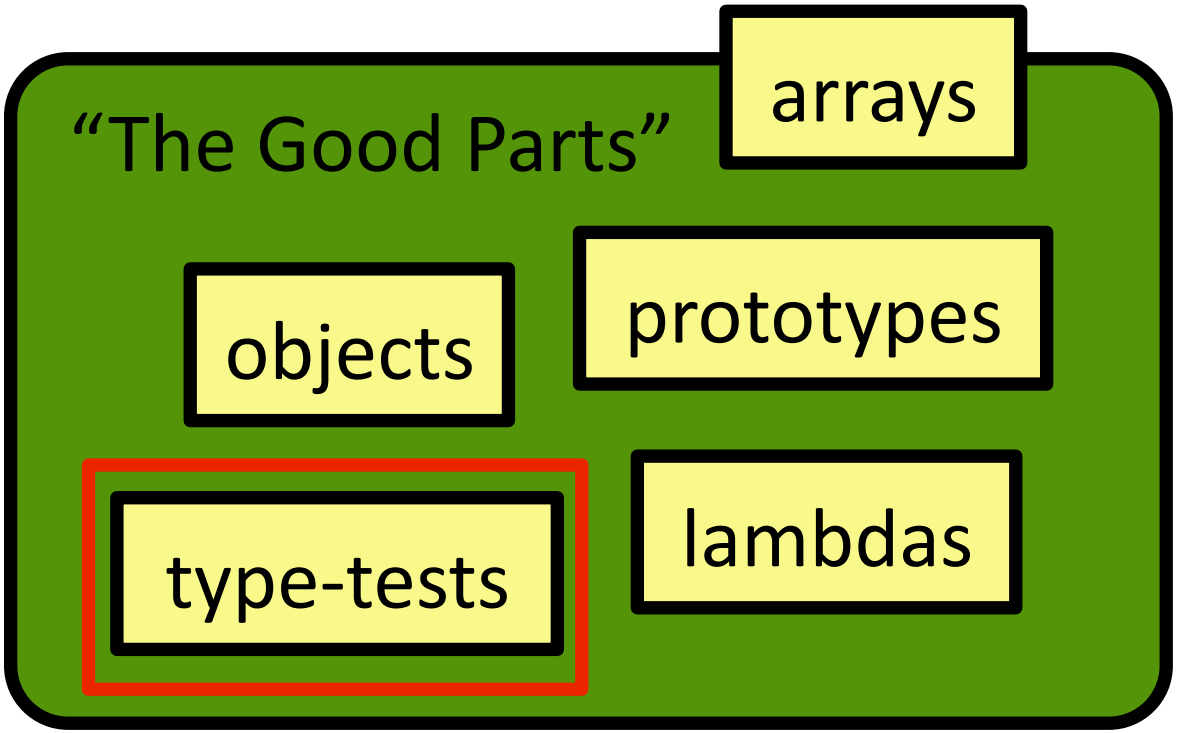
Four inheritance patterns from Crockford's *JS: The Good Parts*

13 Benchmarks



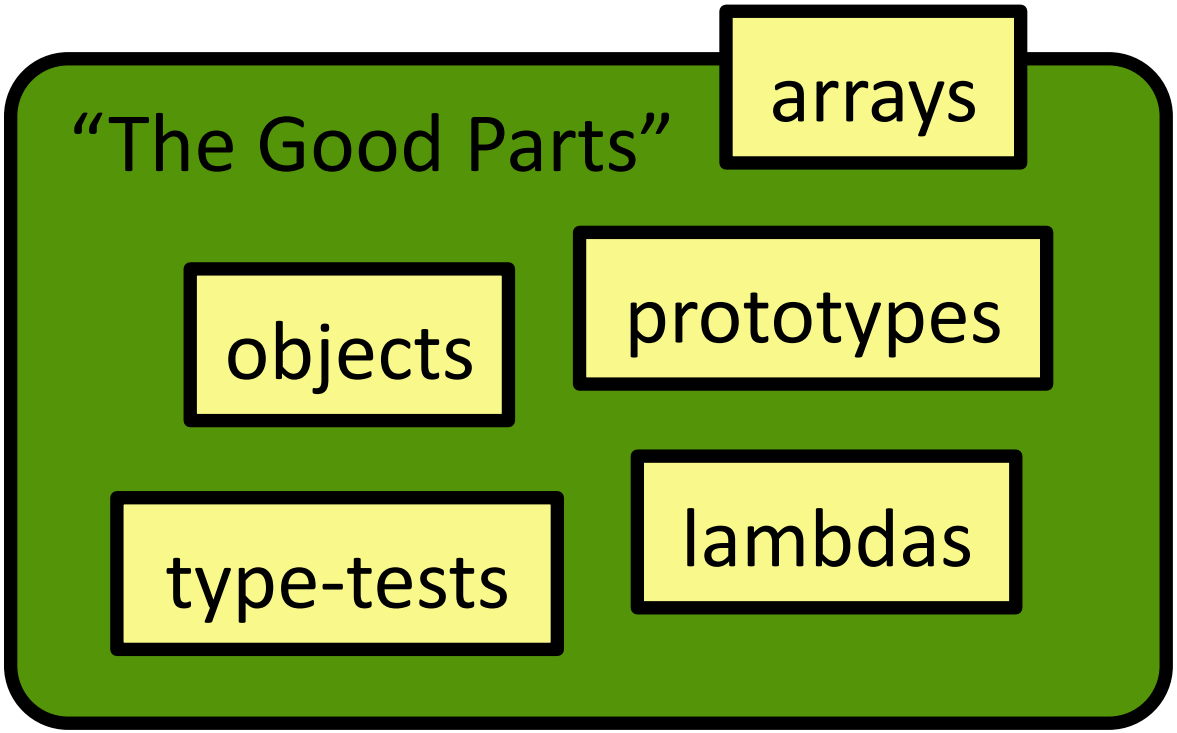
Array-manipulating examples from SunSpider

13 Benchmarks



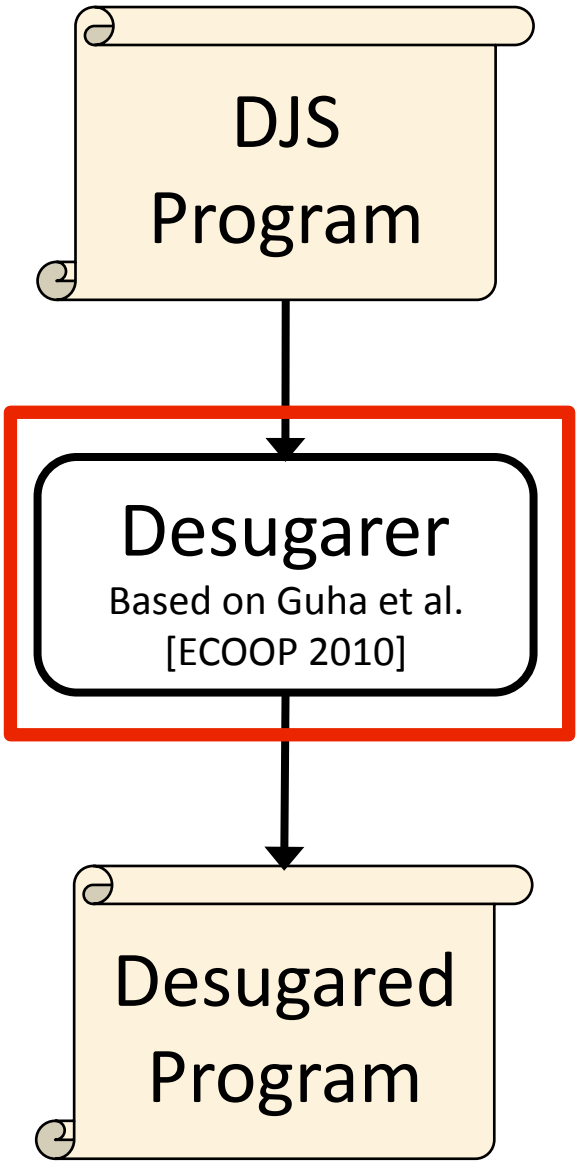
`typeOf()` function to replace `typeof`
from Closure Compiler

13 Benchmarks



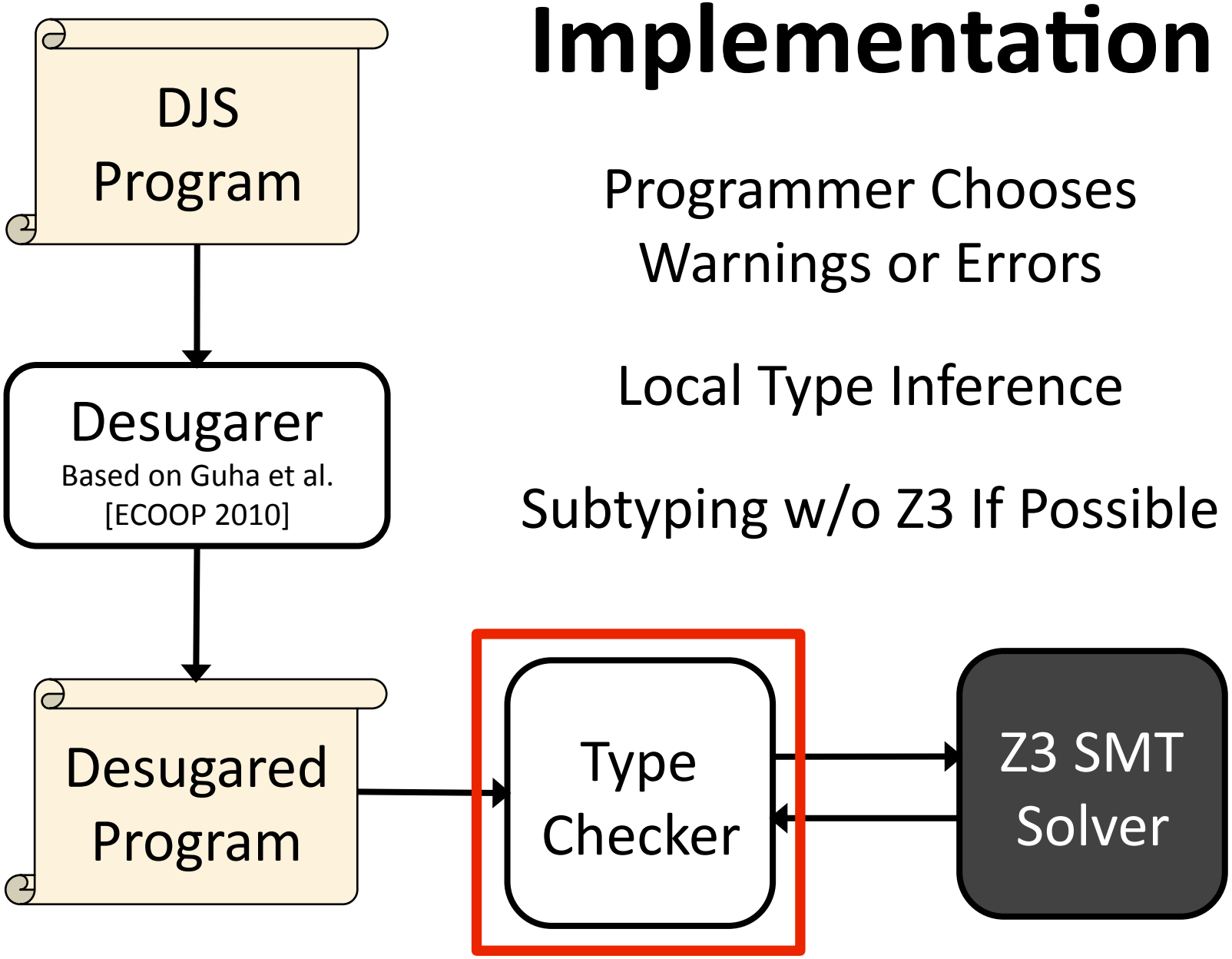
Well-typed programs
don't have run-time errors

Implementation



JavaScript → λ-Calculus + References + Prototypes

Implementation



Programmer Chooses
Warnings or Errors

Local Type Inference

Subtyping w/o Z3 If Possible

Annotation Burden

(Improved since paper)

	~300 LOC to start
+	~100 LOC annotations
=	~400 LOC total

33% Annotation Overhead

Common Cases **Simplified** via
Syntactic Sugar and Local Type Inference

Performance

(Improved since paper)

Total benchmark suite:

~10 seconds
~1100 Z3 queries

11/13 benchmarks in 0-1 s

Common Cases **Fast** via
Syntactic Reasoning When Possible

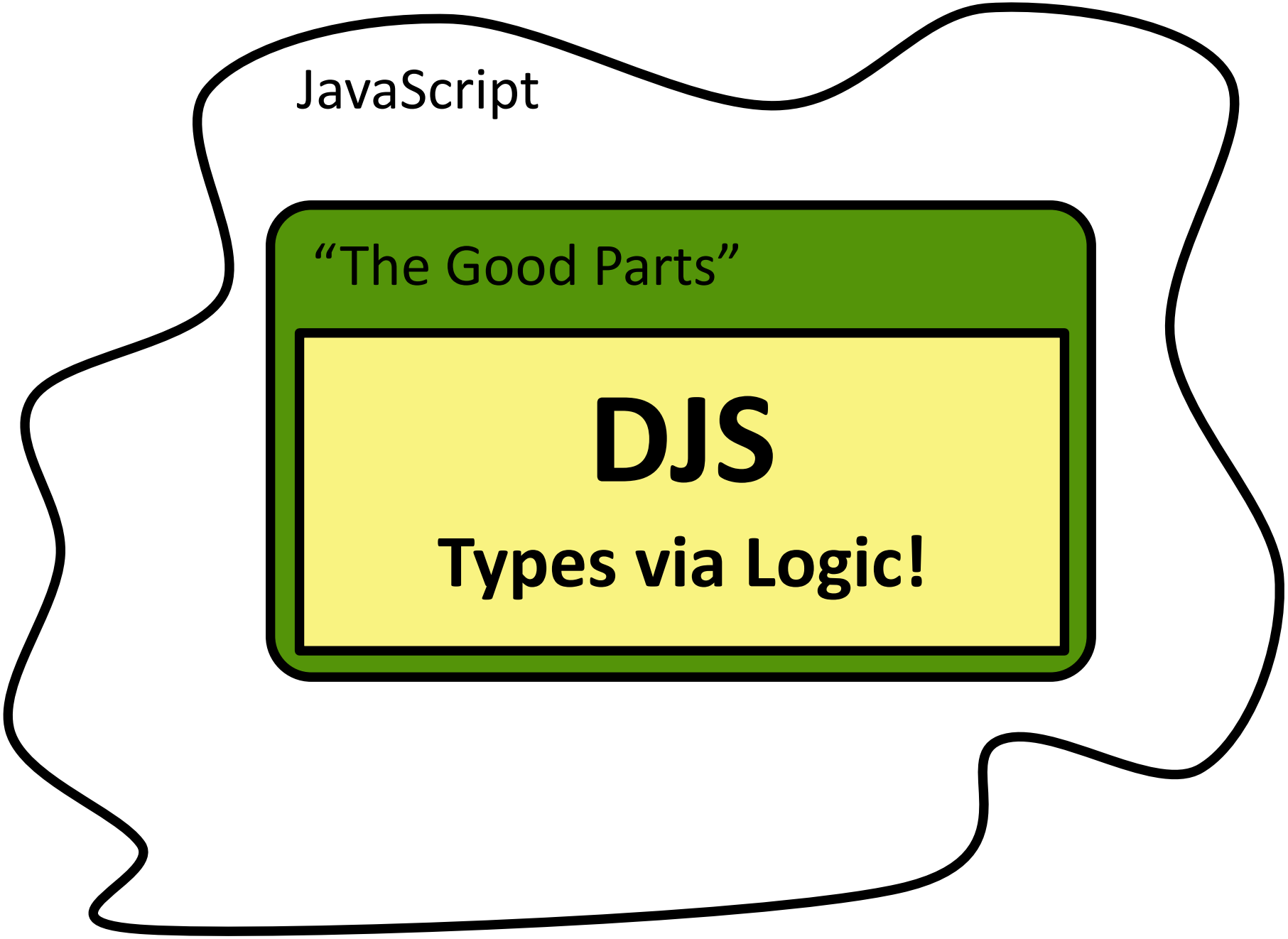
Future Work

Syntax, Inference, Performance

Larger Examples

Type Checker in JS; Run in Browser

IDE Support for Refactoring, etc.


JavaScript
“The Good Parts”
DJS
Types via Logic!

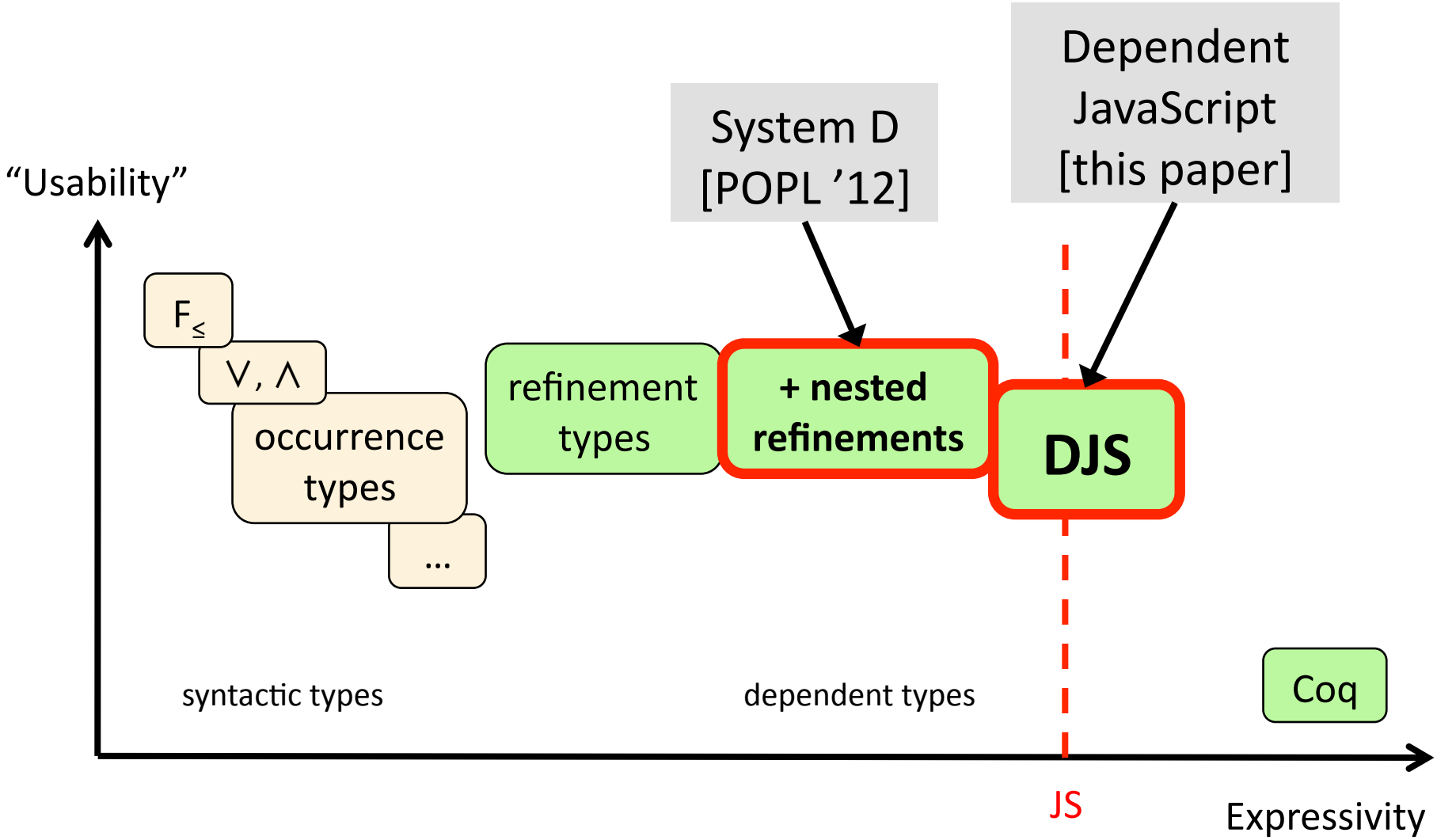
Thanks!

D::

ravichugh.com/djs

PS: I'm on the Job Market

Extra Slides


Dependent JavaScript [this paper]
System D [POPL '12]
"Usability"
$F_{\leq}$
∨, ∧
occurrence types
...
refinement types
+ nested refinements
DJS
syntactic types
dependent types
Coq
JS
Expressivity

System D

+ Types for JS Primitives
+ Strong Updates
+ Quirky JS Arrays
+ Prototype Inheritance

Dependent JavaScript (DJS)

$\{p\} \equiv \{v|p\}$

Function Types

```
/*: x:NumOrBool → {ite Num(x) Num(v) Bool(v)} */
function negate(x) {
  x = (typeof x == "number") ? 0 - x : !x
  return x
}
```

```
/*: x:Any → {v iff falsy(x)} */
function negate(x) {
  x = (typeof x == "number") ? 0 - x : !x
  return x
}
```

Function Types and Objects

$$x : T_1 / H_1 \rightarrow T_2 / H_2$$

input type

input heap

output type

output heap

```
ObjHas(d,k,H,d') ≡ has(d,k) v HeapHas(H,d',k)
```

```
/*: x:Ref / [x |-> d:Dict |> x.pro]
    -> {v iff ObjHas(d,"f",curHeap,x.pro)} / sameHeap */
function hasF(x) {
  return "f" in x
}
```

Function Types and Objects

$$x : T_1 / H_1 \rightarrow T_2 / H_2$$

input type

input heap

output type

output heap

```
ObjSel(d,k,H,d') ≡
        ite has(d,k) sel(d,k) HeapSel(H,d',k)
```

```
/*: x:Ref / [x |-> d:Dict |> x.pro]
    → {v = ObjSel(d,"f",curHeap,x.pro)} / sameHeap */
function readF(x) {
  return x.f
}
```

```
var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - "2"   // -2
}
negate( "2" )
```

Whoa, but perhaps okay…

```
var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - undefined // NaN
}
negate( undefined )
```

Error would be nicer, but okay…

```
var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - {}  // NaN
}
negate( {} )
```

Seems about right…

negate has good intentions

But too many corner cases in JS semantics!

```
var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - []   // 0
}

negate( [] )
```

WAT?!

$$\{ d \mid tag(d) = \text{“Dict”} \wedge has(d, \text{“quack”}) \Rightarrow \boxed{sel(d, \text{“quack”}) :: Unit \rightarrow Str} \}$$

Function types **nested** inside refinements

[POPL 2012]

```
if (duck.quack)
  return “Duck says ” + duck.quack();
else
  return “This duck can’t quack!”;
```

```
var x = {};
x.f;
x.f.f;
```

Programmer configures DJS to report either **warnings** or **errors** for:

1) Possible unbound keys

```
var x = {};
x.f;
x.f.f;
```

Programmer configures DJS to report either **warnings** or **errors** for:

1) Possible unbound keys

2) Possible run-time errors