Dependent Types for JavaScript

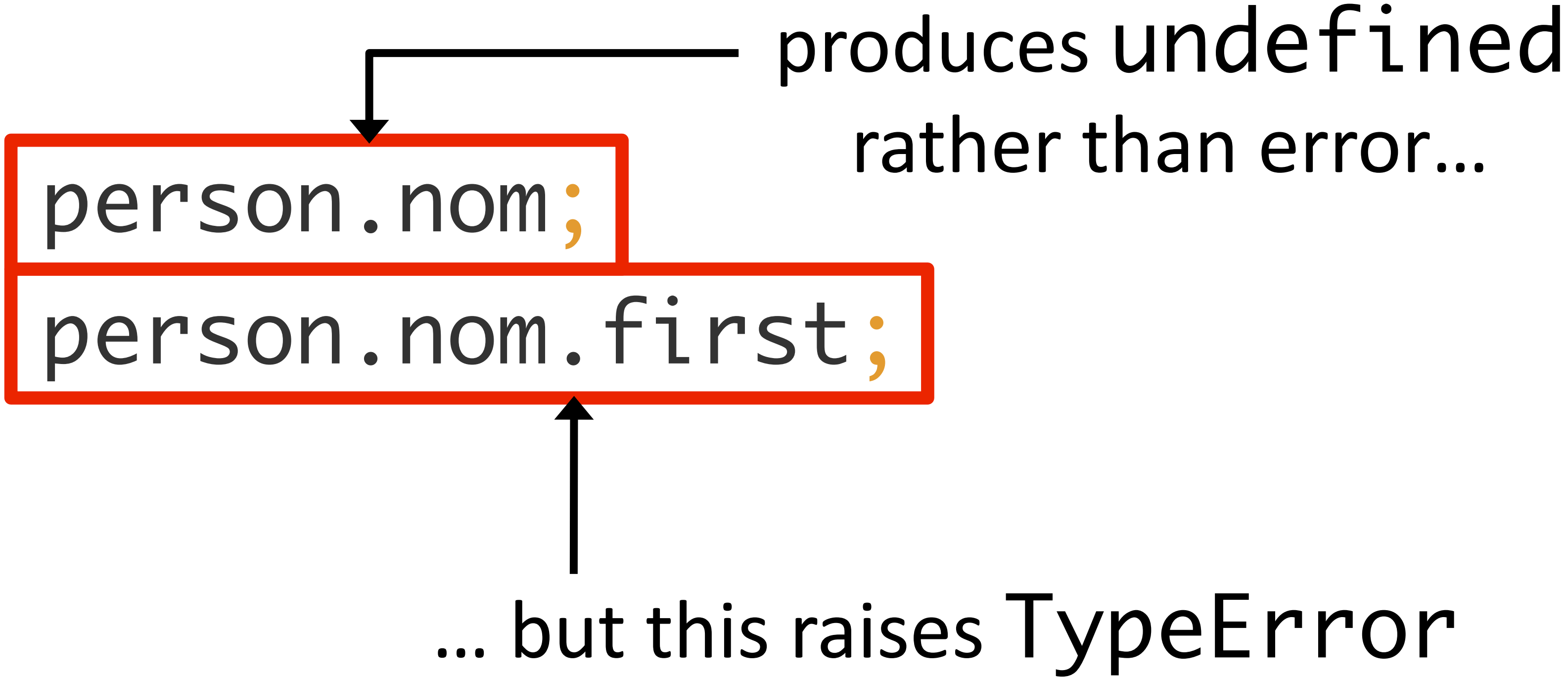
Ravi Chugh	UCSD
Ranjit Jhala	UCSD
Dave Herman	Mozilla
Pat Rondon	Google
Panos Vekris	UCSD

“Dynamic” Features
Facilitate Rapid Innovation

Types for JavaScript

1. Better Development Tools
2. Better Reliability
3. Better Performance

```
var person = {
  name : { first : “John”,
           last  : “McCarthy” }};
```



```
person.nom;
person.nom.first;
```

produces `undefined` rather than error…

… but this raises `TypeError`

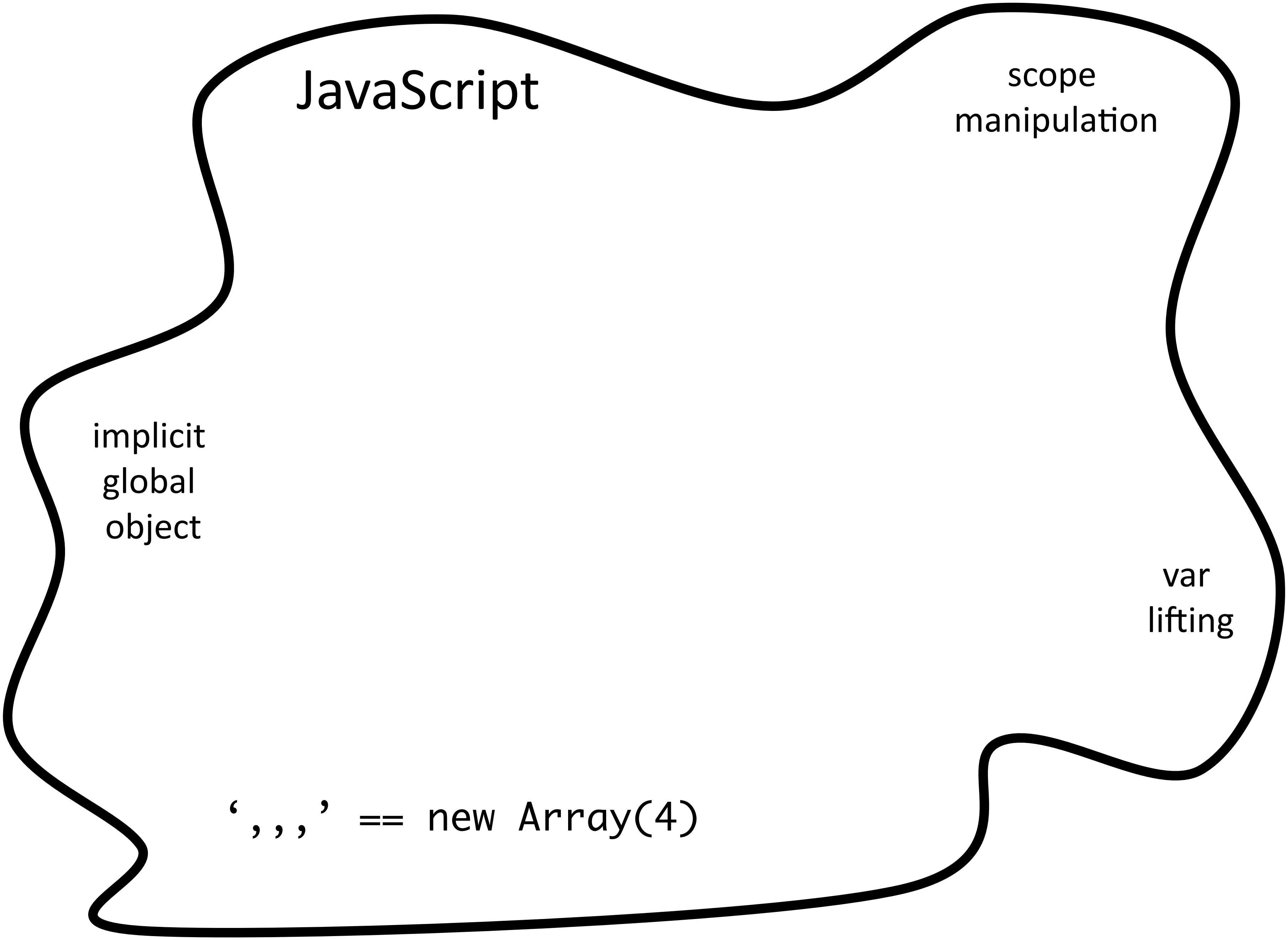
```
var person = {
  name : { first : “John”,
           last  : “McCarthy” }};

if (unlikely()) {
   person.nom;
   person.nom.first;
}
```

some errors hard to catch with testing

Types for JavaScript

Will Never Replace Need for
Testing and **Dynamic** Checking

But Want **Static** Checking When Possible



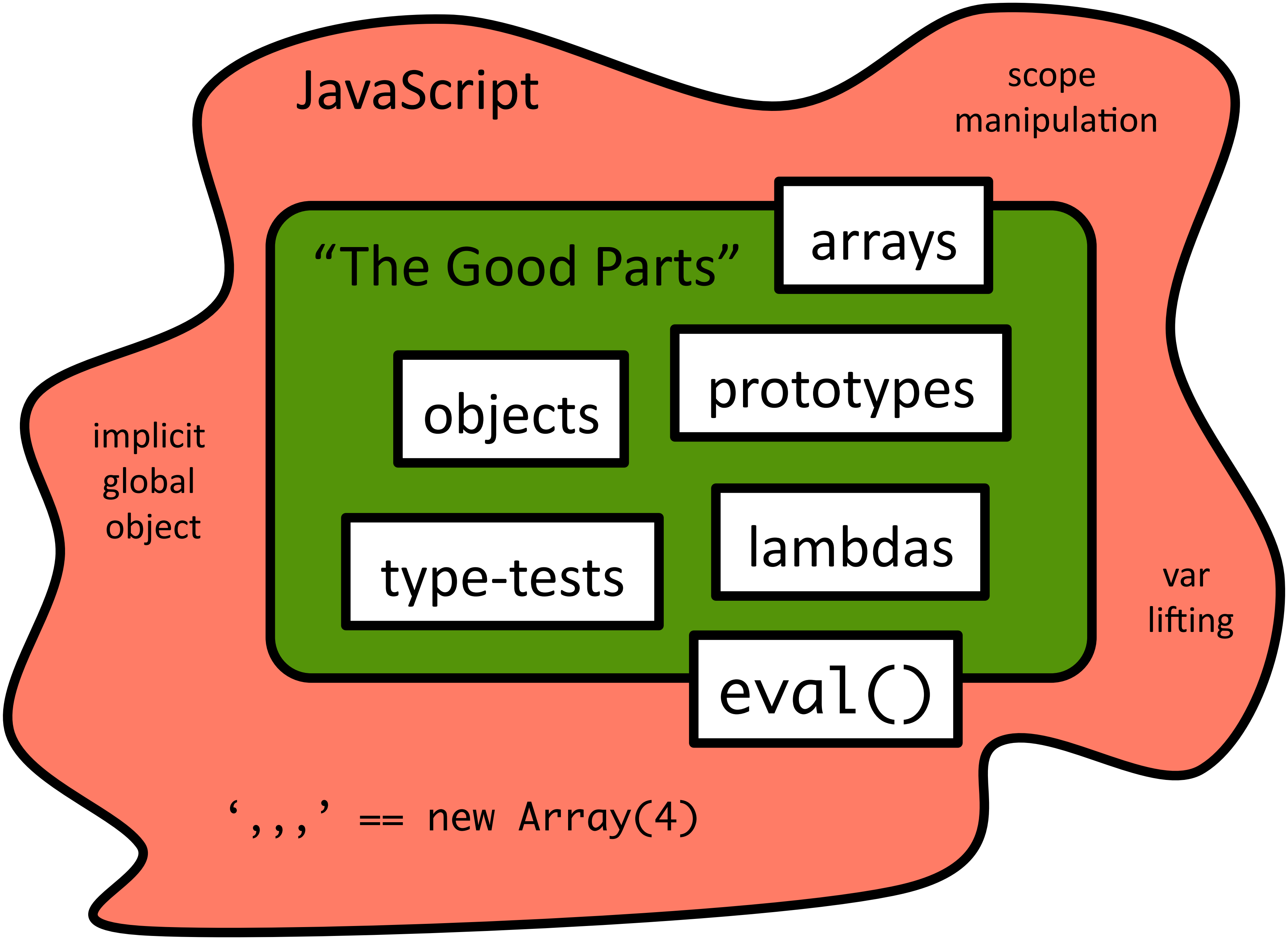
JavaScript

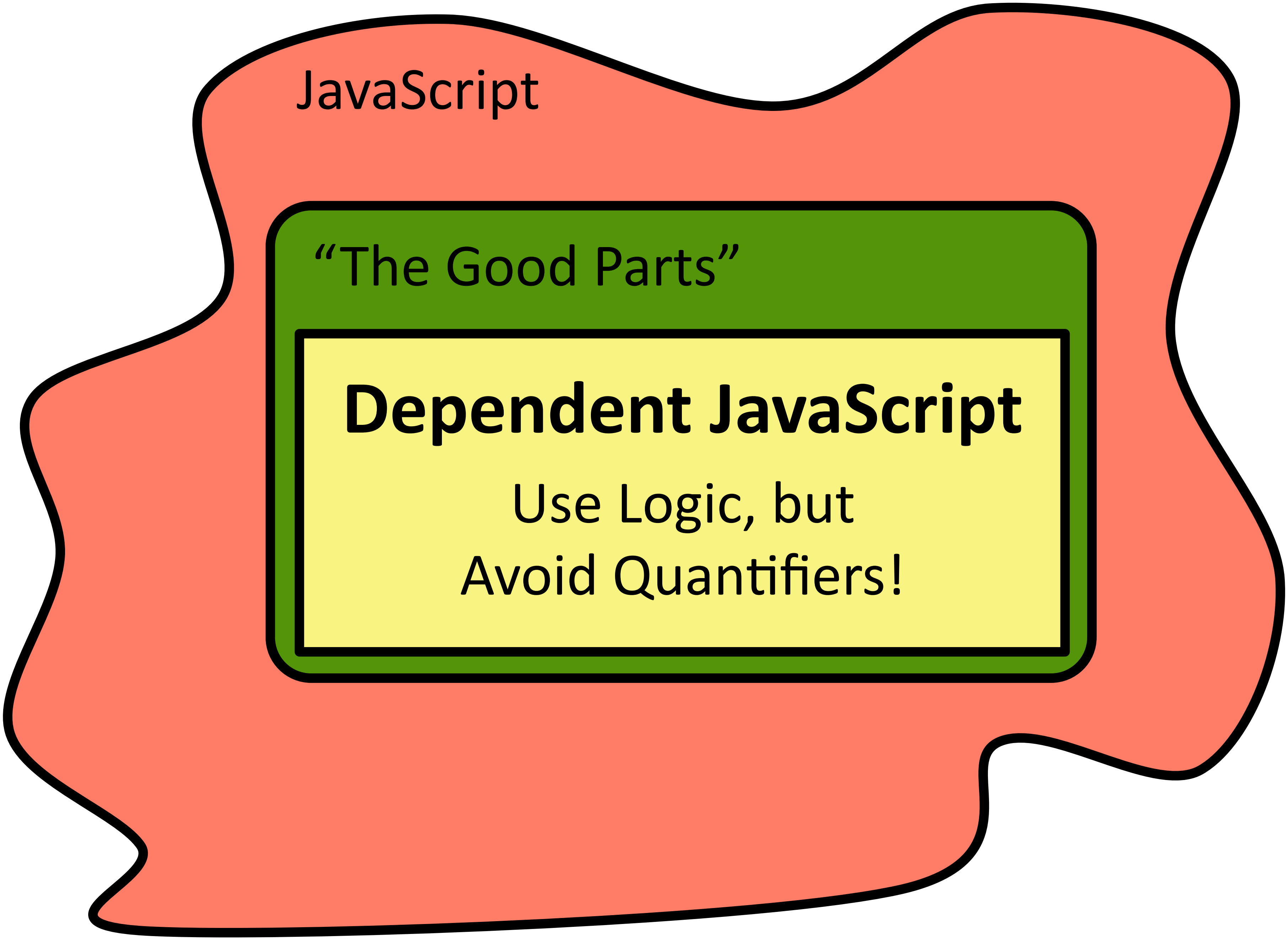
scope
manipulation

implicit
global
object

var
lifting

```
‘,,,’ == new Array(4)
```


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

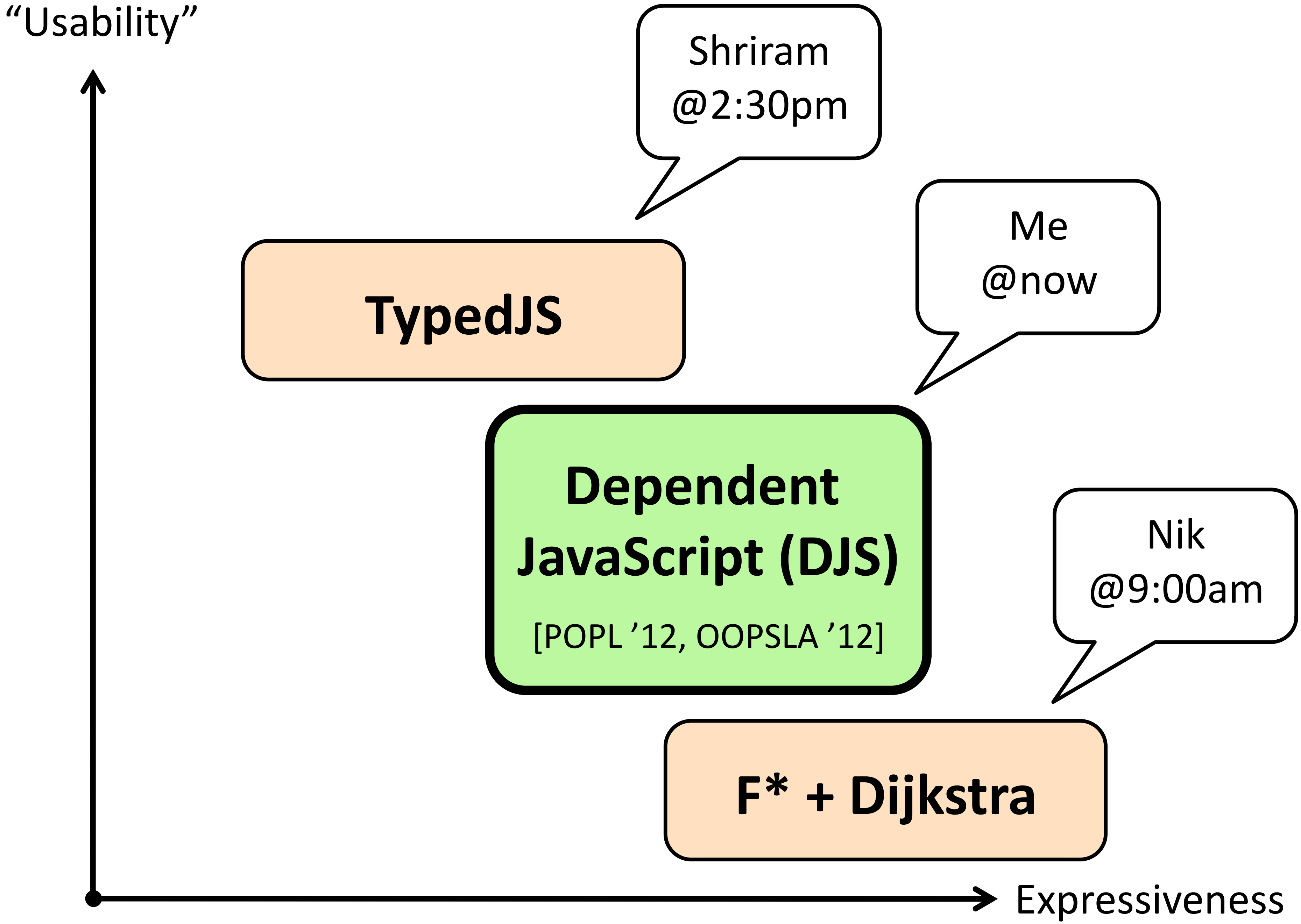


JavaScript

“The Good Parts”

Dependent JavaScript

Use Logic, but
Avoid Quantifiers!



"Usability"
Shriram @2:30pm
TypedJS
Me @now
Dependent JavaScript (DJS)
[POPL '12, OOPSLA '12]
Nik @9:00am
F* + Dijkstra
Expressiveness

DJS = Refinement Types
+ Several New
Quantifier-Free
Mechanisms

Me
@now

Dependent JavaScript (DJS)

[POPL ’12, OOPSLA ’12]

```
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

$$\texttt{Num} \equiv \{\, \texttt{n} \mid \texttt{tag}(\texttt{n}) = \text{“}\texttt{number}\text{”} \,\}$$

$$\texttt{NumOrBool} \equiv \{\, \texttt{v} \mid \texttt{tag}(\texttt{v}) = \text{“}\texttt{number}\text{”} \vee \texttt{tag}(\texttt{v}) = \text{“}\texttt{boolean}\text{”} \,\}$$

$$\texttt{Int} \equiv \{\, \texttt{i} \mid \texttt{tag}(\texttt{i}) = \text{“}\texttt{number}\text{”} \wedge \texttt{integer}(\texttt{i}) \,\}$$

$$\texttt{Any} \equiv \{\, \texttt{x} \mid \texttt{true} \,\}$$

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

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

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( [] )
```

?!?

```
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

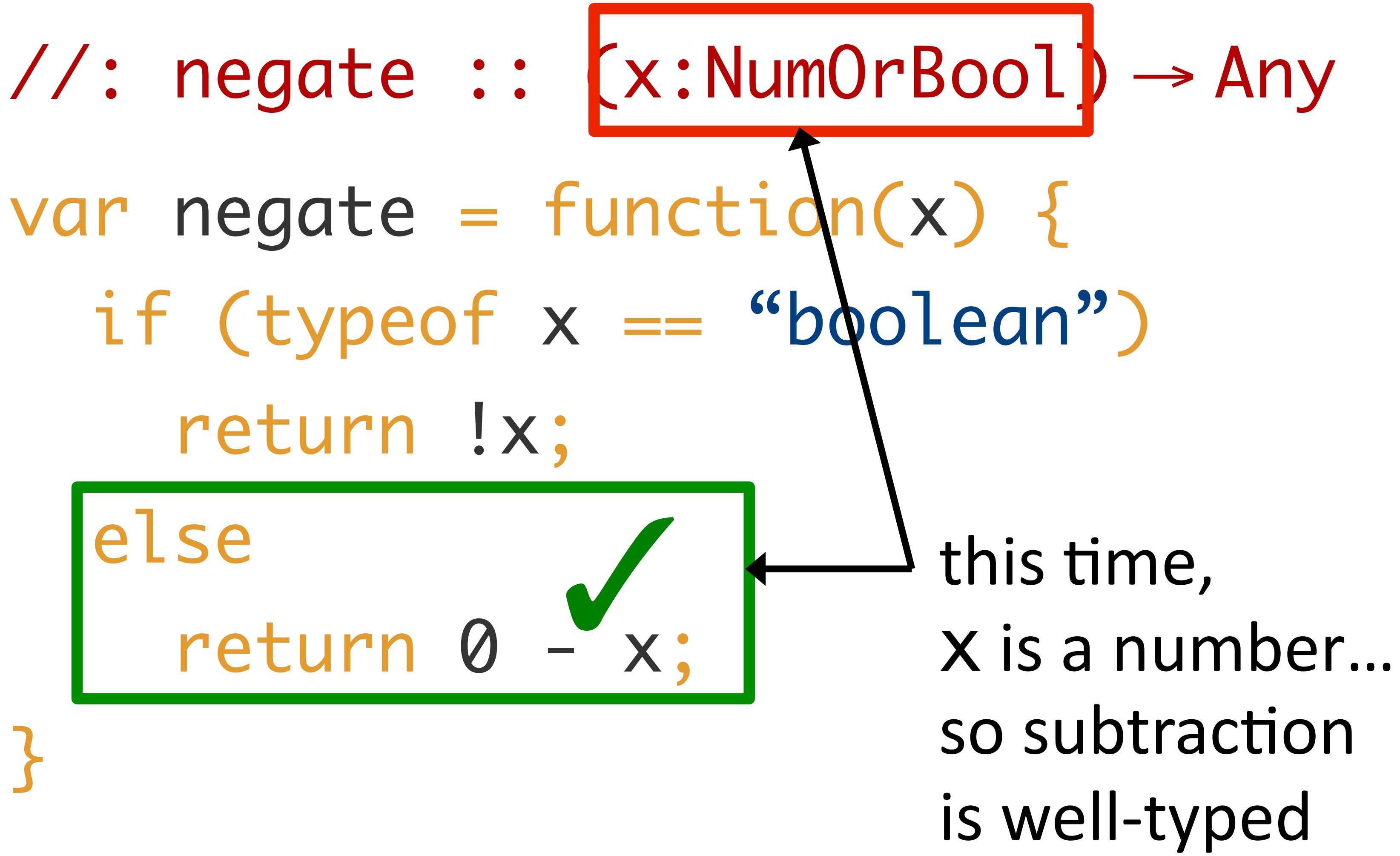
DJS is Path Sensitive

```
//: 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)
            → { y | tag(y) = tag(x) } */

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

✓

output type
depends on
input value

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!”;
```

$$\{\, d \mid \texttt{tag}(d) = \text{“}\texttt{object}\text{”} \land \boxed{\begin{array}{l}\texttt{has}(d, \text{“}\texttt{quack}\text{”}) \Rightarrow \\ \quad \texttt{sel}(d, \text{“}\texttt{quack}\text{”})\end{array}} :: \texttt{Unit} \rightarrow \texttt{Str} \,\}$$

Operators from McCarthy theory of arrays

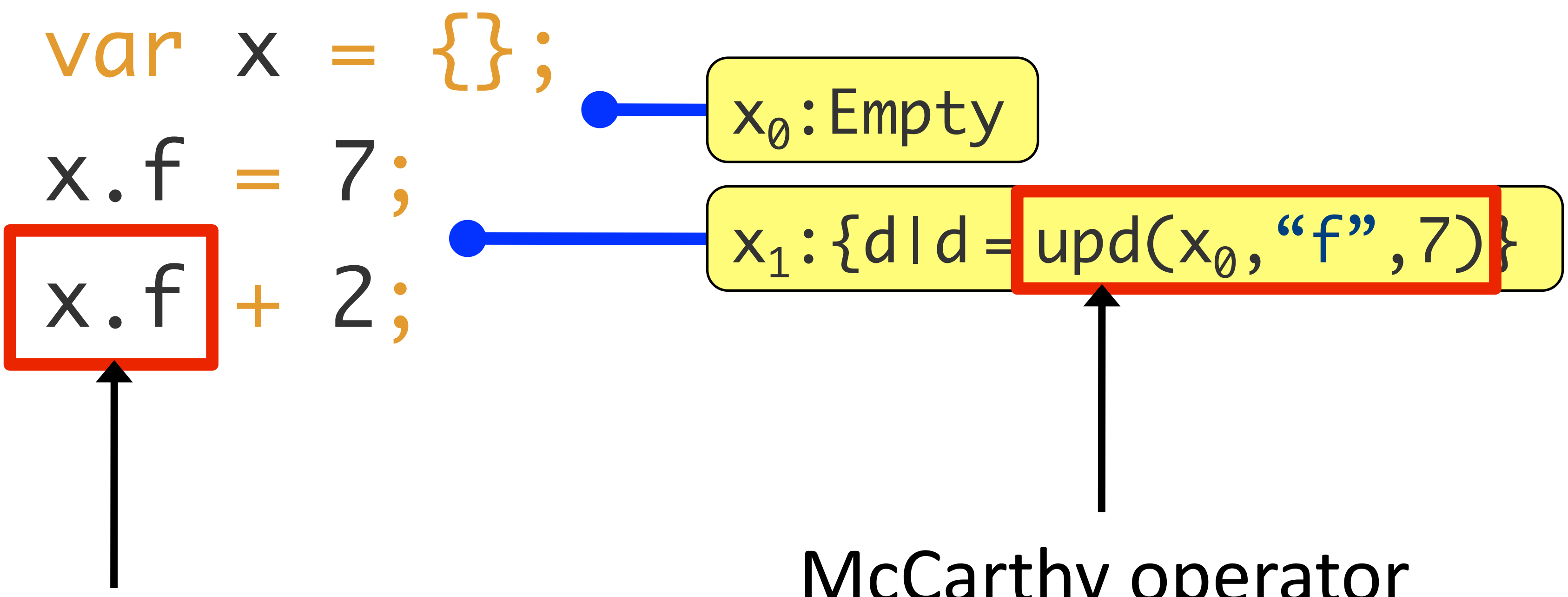
```
if (duck.quack)
  return “Duck says ” + duck.quack();
else
  return “This duck can’t quack!”;
```

{ d | tag(d) = “object” ∧
has(d,“quack”) ⇒
sel(d,“quack”) :: Unit → 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;
```

$x_0 : \mathrm{Empty}$

$x_1 : \{d \mid d = \boxed{\mathrm{upd}(x_0, \text{“f”}, 7)}\}$

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:{d|d = upd(x_0,“f”,7)}

Strong updates to singleton objects

Weak updates to collections of objects

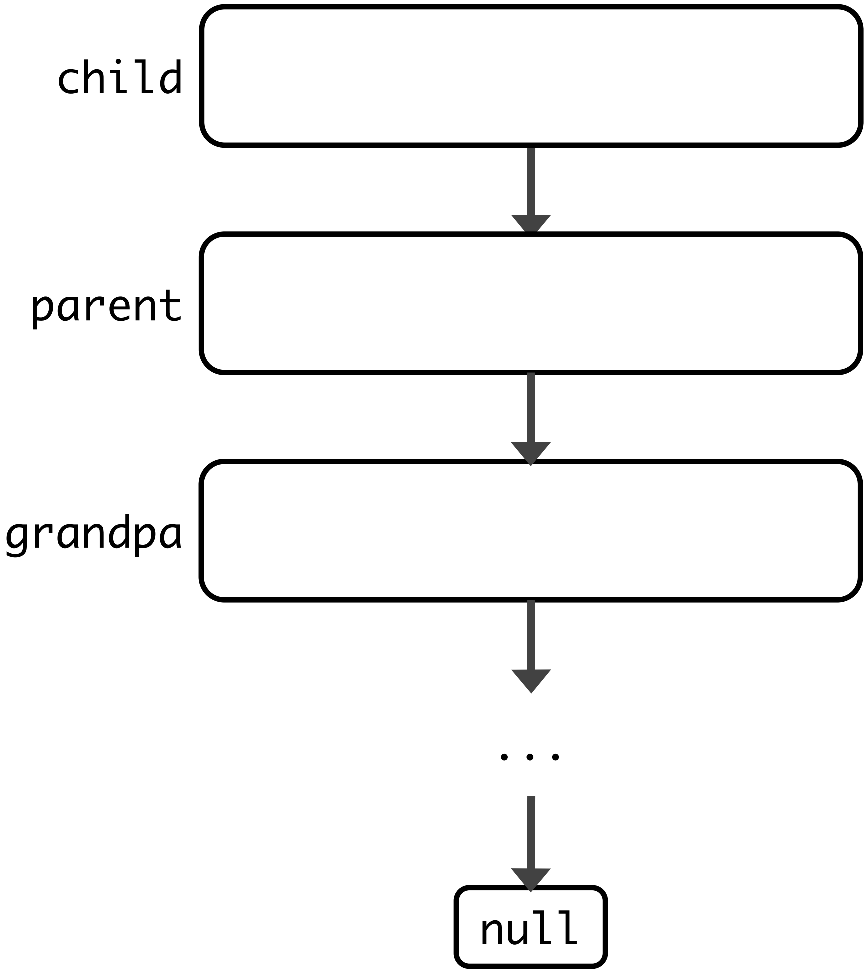
Tag-Tests	Duck Typing	Mutable Objects	Prototypes	Arrays

Typical “Dynamic” Features

Tag-Tests | Duck Typing | Mutable Objects | Prototypes | Arrays

Typical
“Dynamic”
Features

JavaScript



child

parent

grandpa

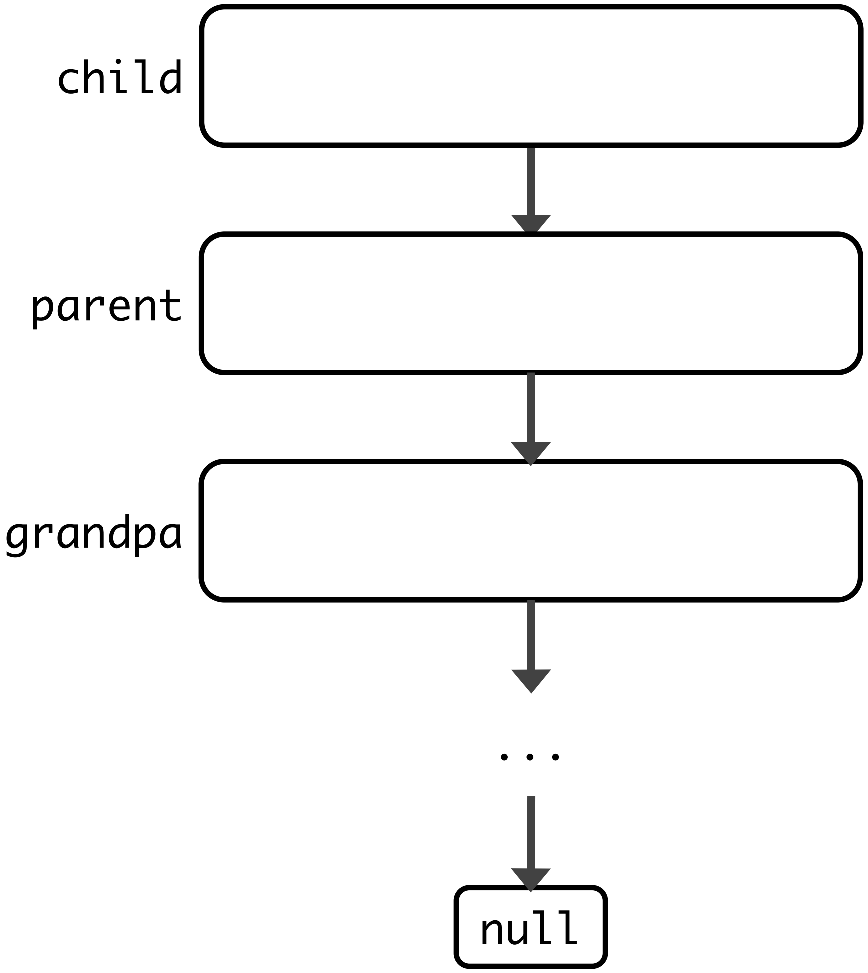
...

null

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

Semantics of Key Lookup

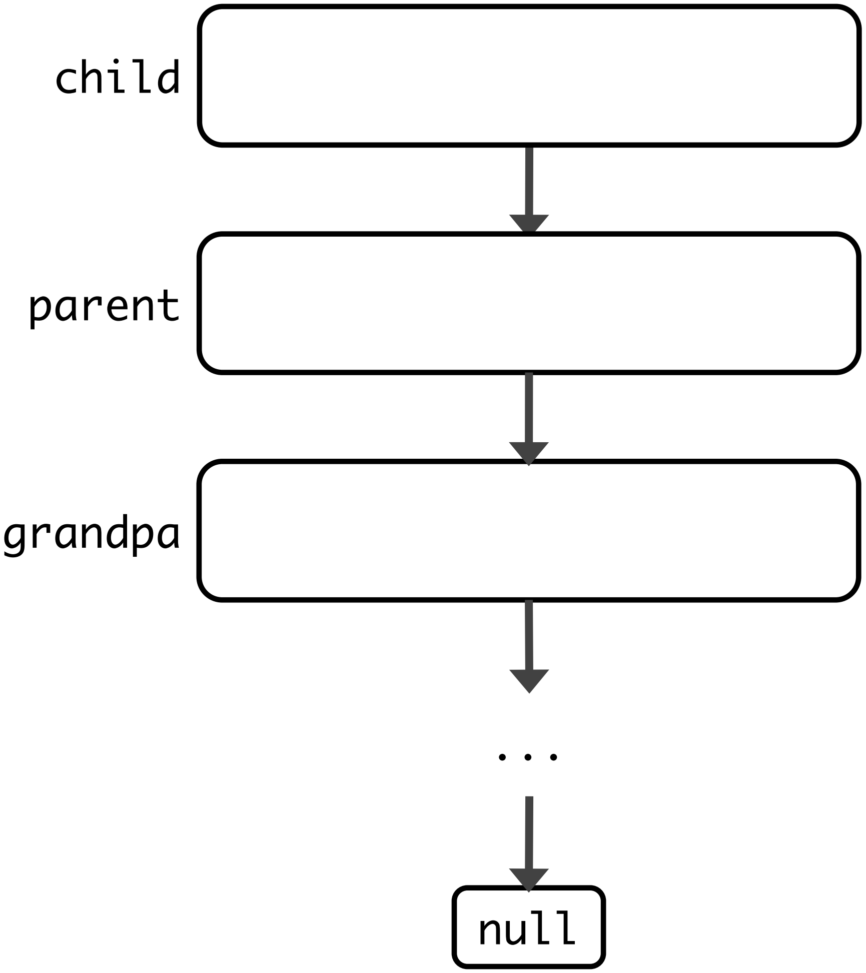
```
child[k];
```



```
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`

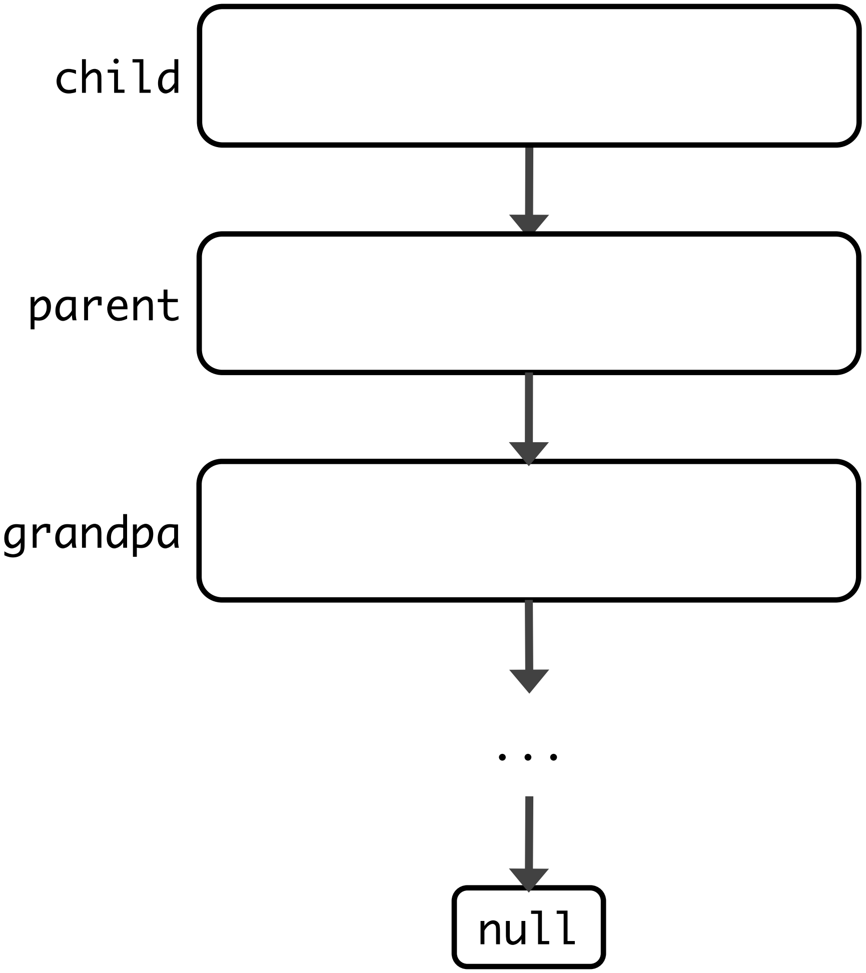
Else if `grandpa` contains k, then
  Read k from `grandpa`

Else if …

Else
  Return `undefined`

Semantics of Key Lookup

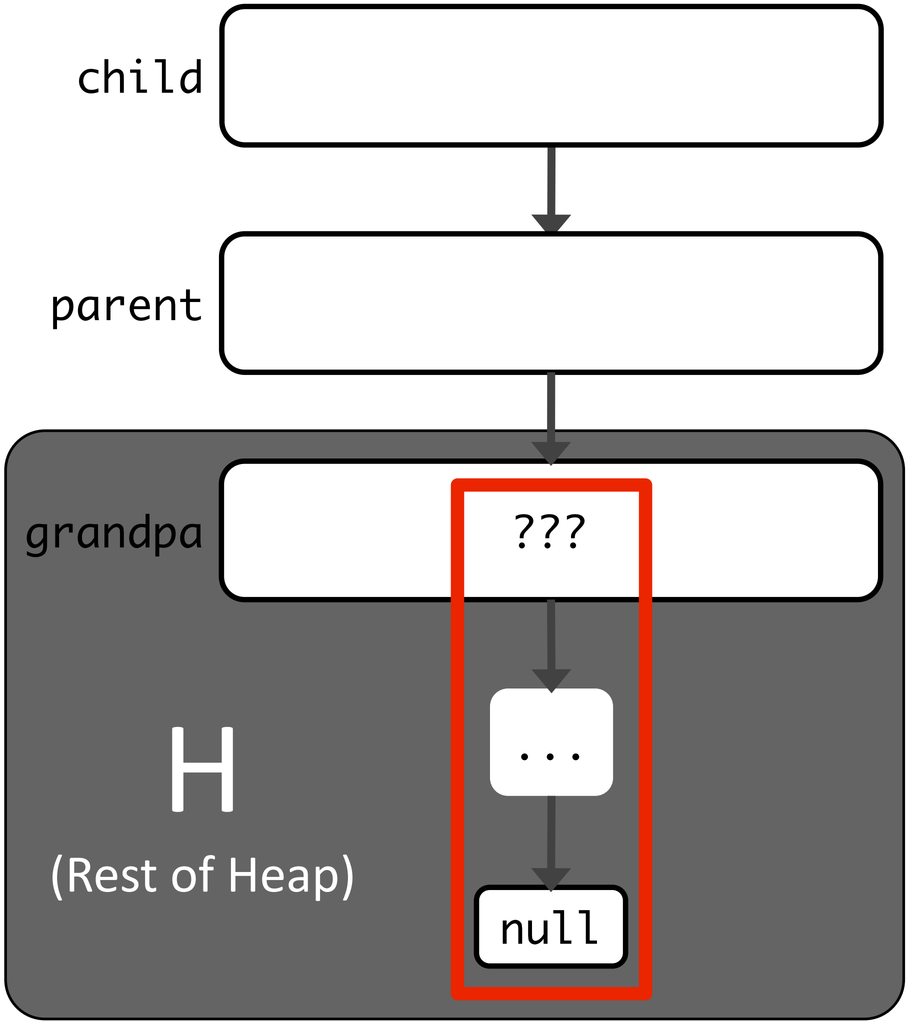
```
child[k];
```



```
{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

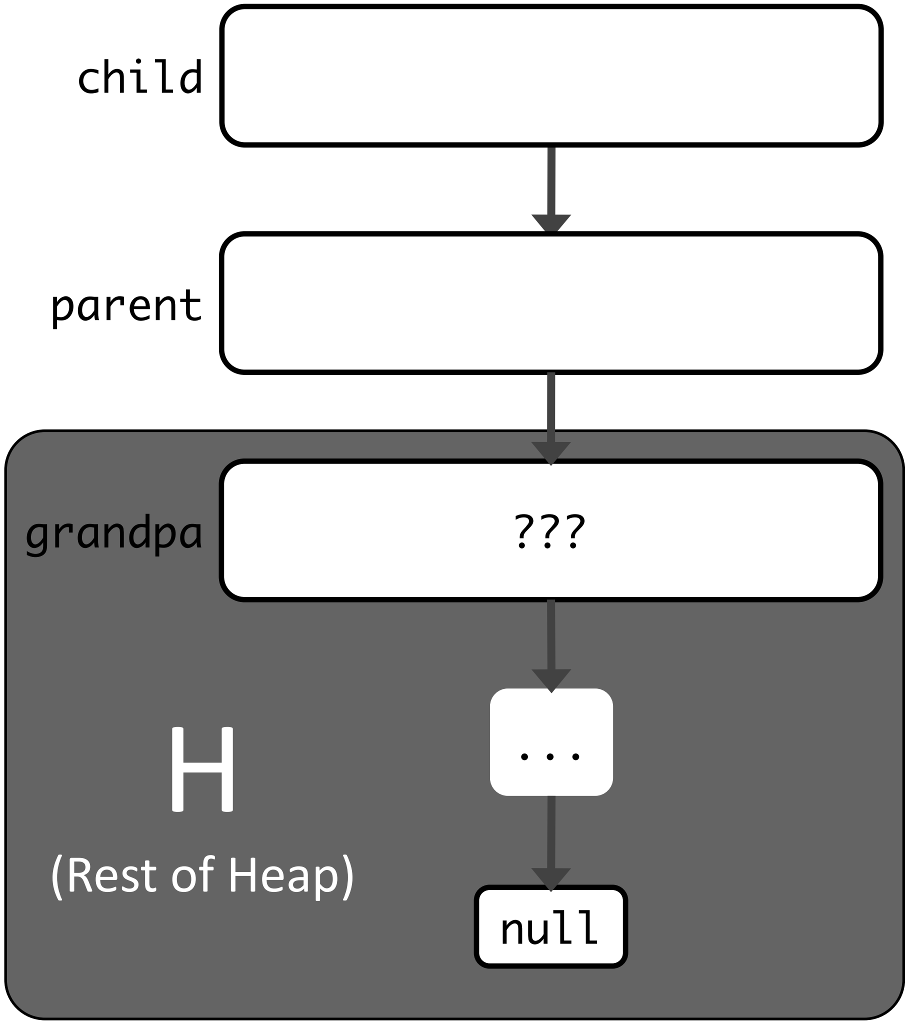
```
child[k];
```



```
{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

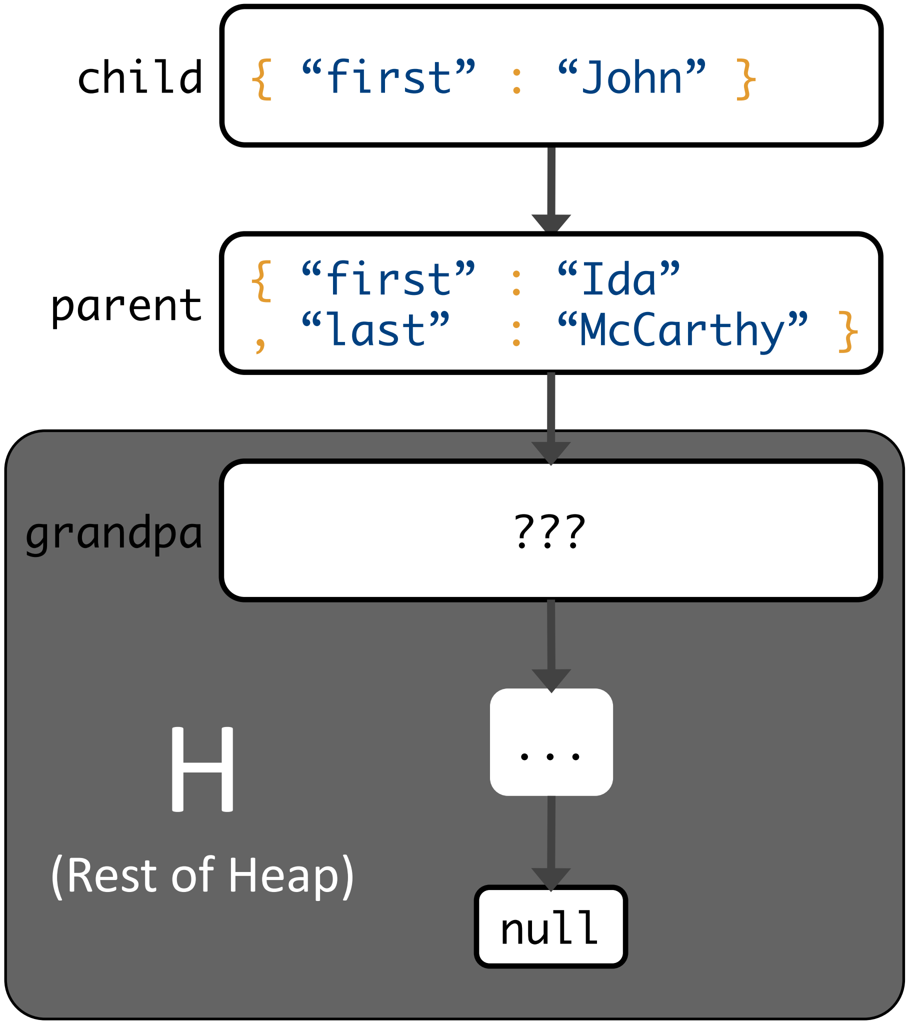
```
child[k];
```



```
{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];
```

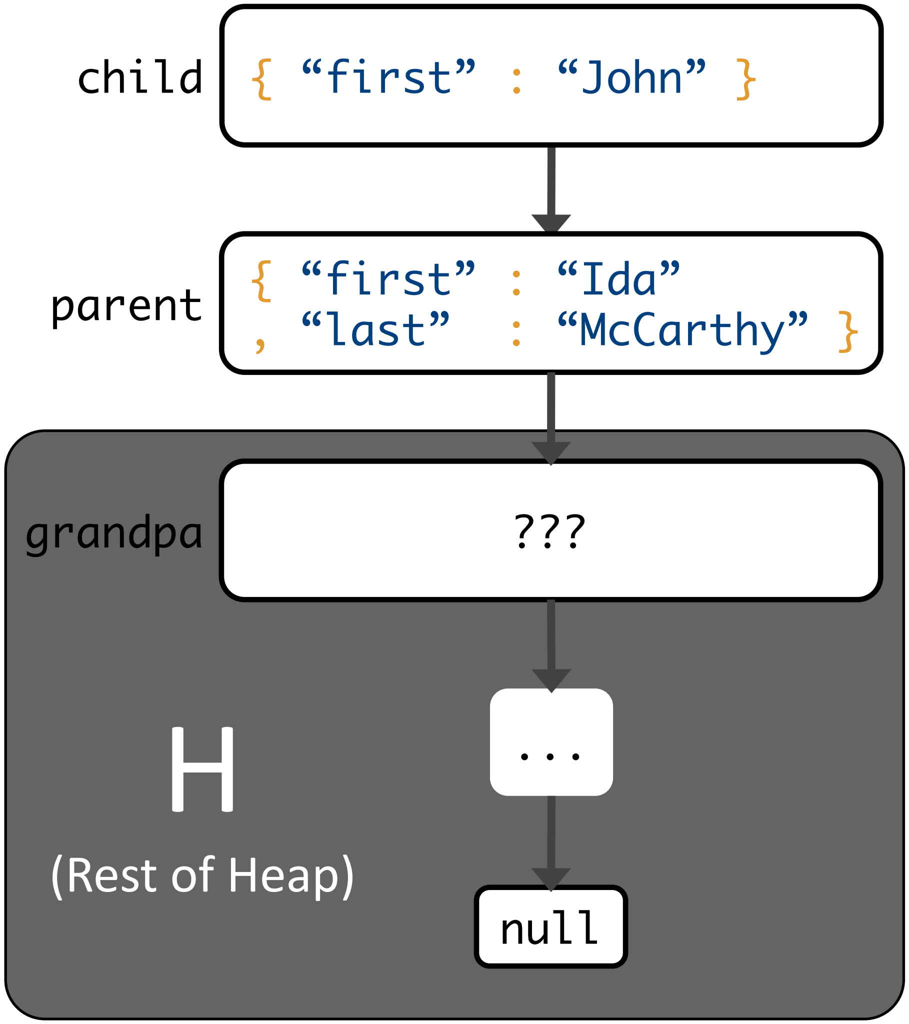


```
{v  if has(child,k) then
      v = sel(child,k)

    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

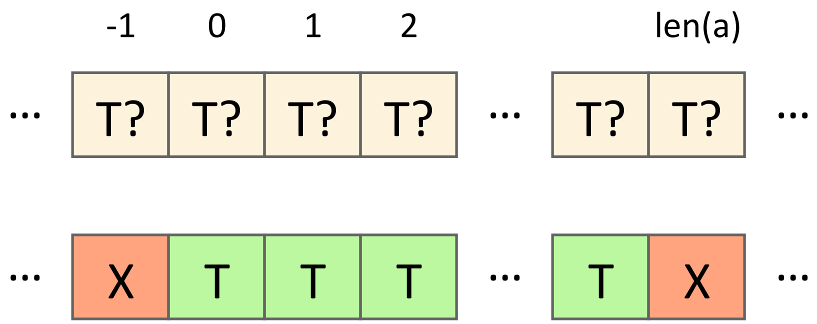
```
var nums = [0,1,2];
while (…) {
  nums[nums.length] = 17;
}

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

$$
\begin{aligned}
\{\, a \mid a :: \mathtt{Arr}(T) \\
\wedge\ \mathtt{packed}(a) \\
\wedge\ \mathtt{len}(a) = 10 \,\}
\end{aligned}
$$



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

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

Encode **tuples** as arrays

```
var tup = [17, “cacti”];
```

$$\begin{aligned}
&\{\, a \mid a :: \mathtt{Arr}(\mathtt{Any}) \\
&\qquad \land\ \mathtt{packed}(a) \land \mathtt{len}(a) = 2 \\
&\qquad \land\ \mathtt{Int}(\mathtt{sel}(a,0)) \\
&\qquad \land\ \mathtt{Str}(\mathtt{sel}(a,1)) \,\}
\end{aligned}$$

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

$$\{\, a \mid a :: \mathrm{Arr}(\mathrm{Any})$$

$$\wedge\ \mathrm{packed}(a) \wedge \mathrm{len}(a) = 3$$

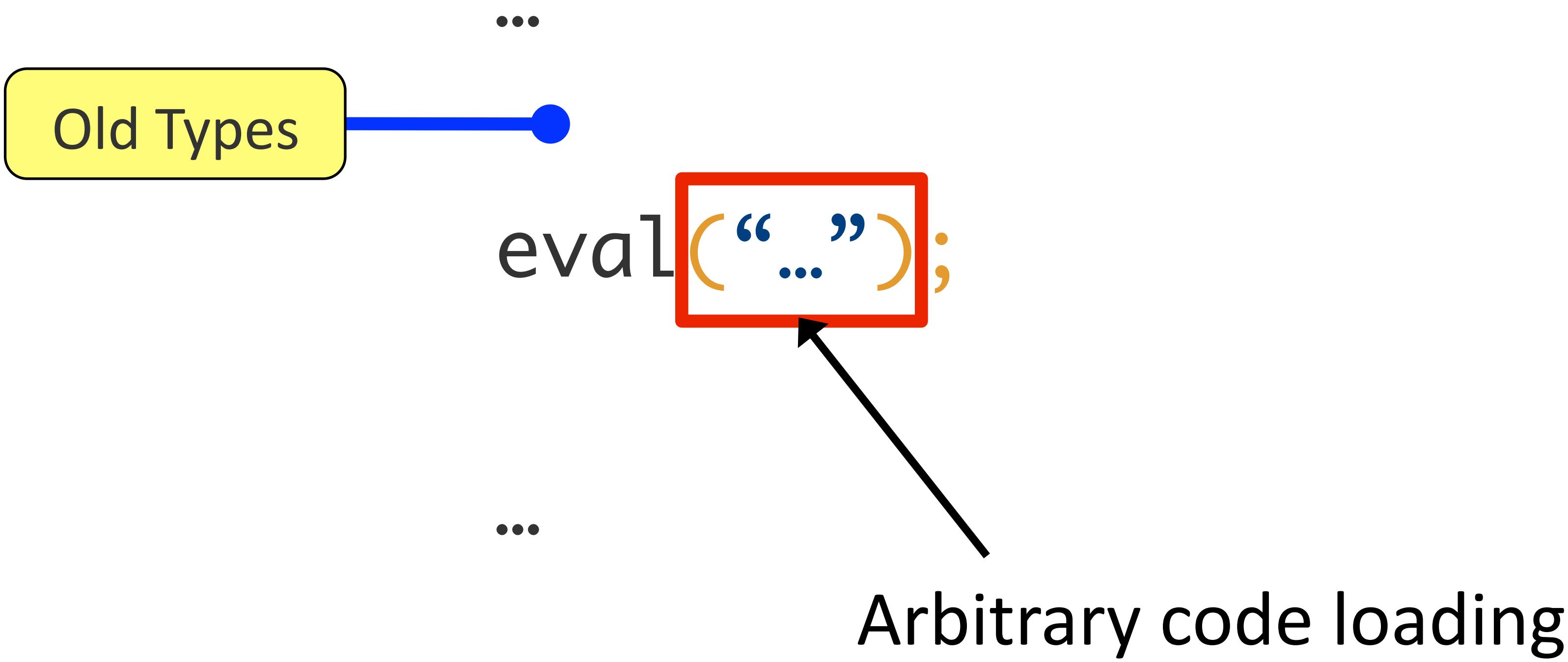
$$\wedge\ \ldots \}$$

DJS handles other array **quirks**:

- Special `length` property
- `Array.prototype`
- Non-integer keys

Tag-Tests	Duck Typing	Mutable Objects	Prototypes	Arrays

What About eval?

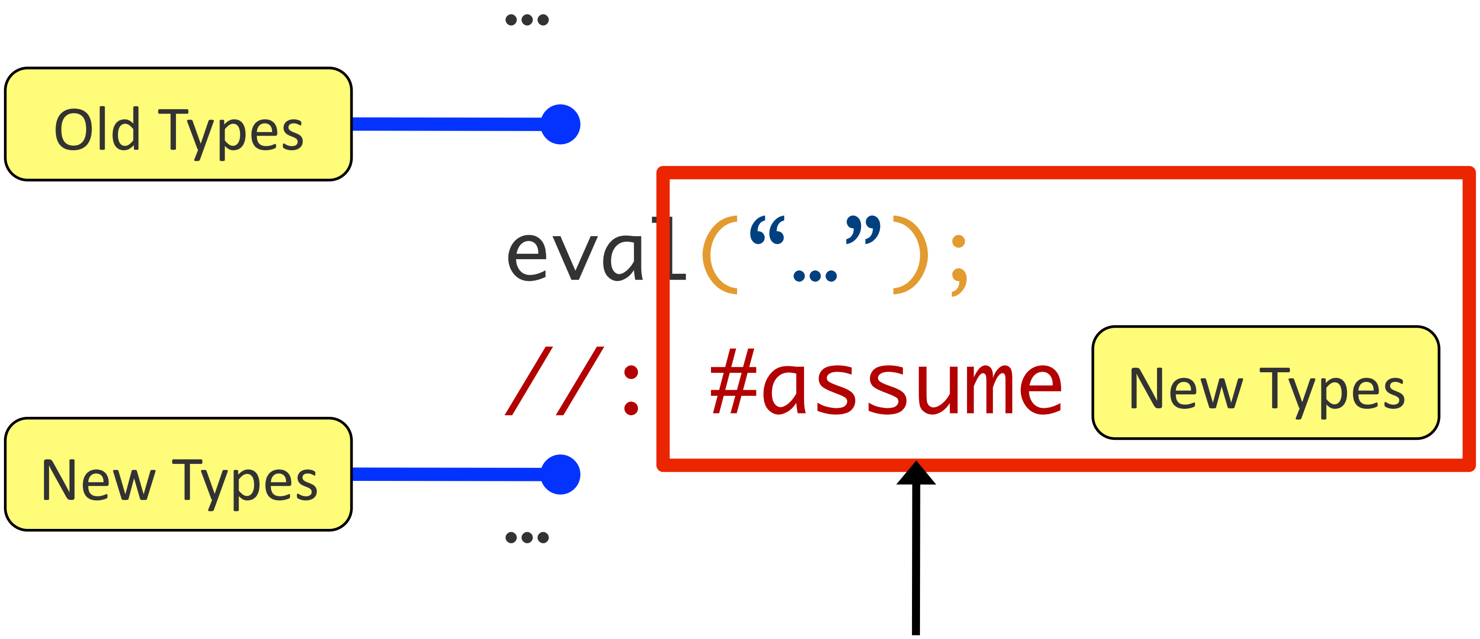


Old Types

```
…

eval("…");

…
```

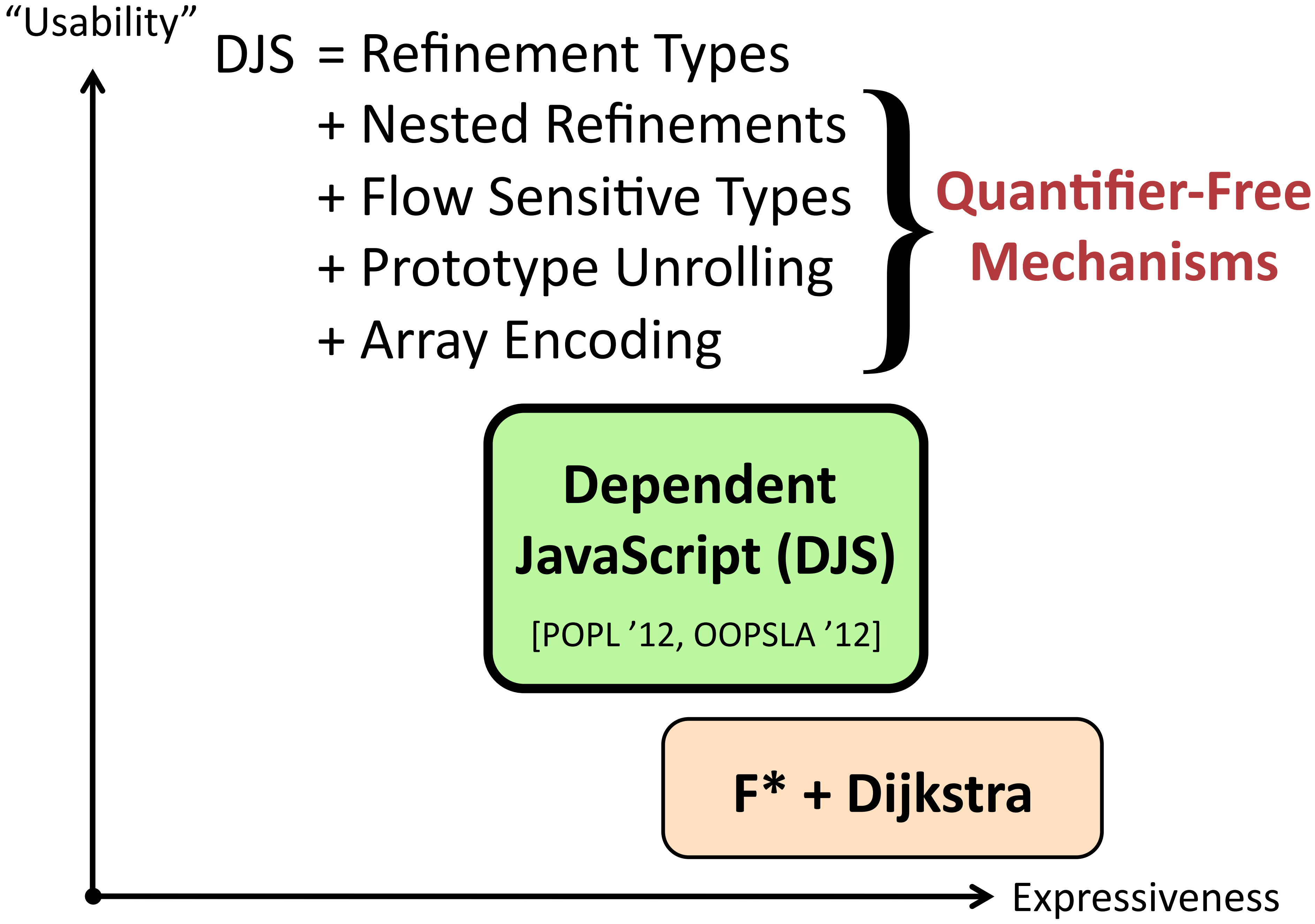
Arbitrary code loading

What About eval?



```
…
eval("…");
//: #assume New Types
…
```

Can Integrate DJS with
"Contract Checking" at Run-time
aka "Gradual Typing"



“Usability”

DJS = Refinement Types
+ Nested Refinements
+ Flow Sensitive Types
+ Prototype Unrolling
+ Array Encoding

Quantifier-Free Mechanisms

Dependent JavaScript (DJS)

[POPL ’12, OOPSLA ’12]

F* + Dijkstra

Expressiveness

Function Subtyping…

```
   { d | sel(d,“f”) :: (x:Any) → { y|y = x } }
<: { d | sel(d,“f”) :: (x:Num) → Num }
```

Function Subtyping…

```
   sel(d,“f”) :: (x:Any) → {y|y=x}
⇒  sel(d,“f”) :: (x:Num) → Num
```

Function Subtyping...

$$f :: (x{:}\mathrm{Any}) \to \{\, y \mid y = x \,\}$$

$$\Rightarrow \quad f :: (x{:}\mathrm{Num}) \to \mathrm{Num}$$

... With Quantifiers

$$\forall x, y.\ \ \mathrm{true} \ \land\ y = f(x) \ \Rightarrow\ y = x$$

$$\Rightarrow\ ✓\ \forall x, y.\ \ \mathrm{Num}(x) \ \land\ y = f(x) \ \Rightarrow\ \mathrm{Num}(y)$$

Valid, but First-Order Logic is Undecidable

Function Subtyping…

```
   f :: (x:Any) → {y|y=x}
⇒  f :: (x:Num) → Num
```

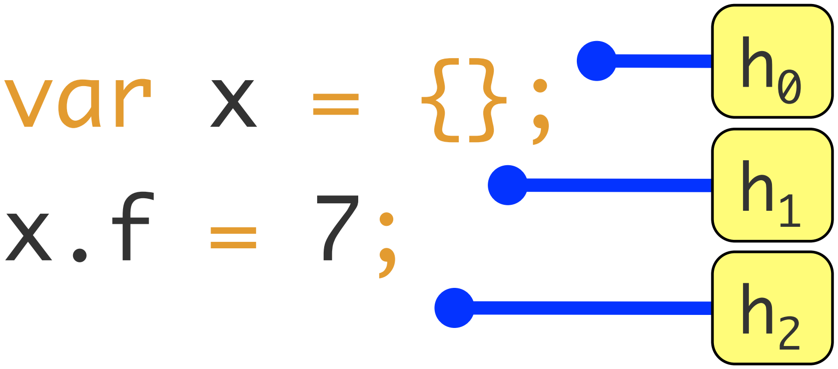
… Without Quantifiers!

Nested Refinements

Treat Function Types as **Uninterpreted**

Implication = SMT Validity + Syntactic Subtyping

Heap Updates…



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

… With Quantifiers

…

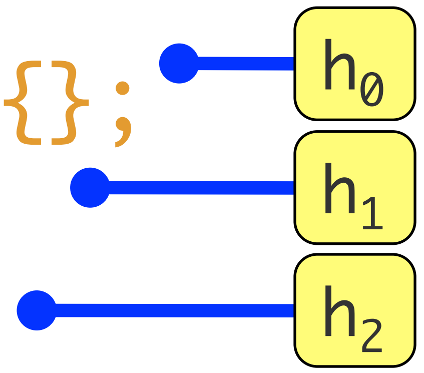
Encode Heap w/ McCarthy Operators

$$\wedge\ \mathtt{sel}(h_1, \mathtt{x}) = \mathtt{empty}$$
$$\wedge\ \forall \mathtt{y}.\ \mathtt{x} \neq \mathtt{y} \Rightarrow \mathtt{sel}(h_1, \mathtt{y}) = \mathtt{sel}(h_0, \mathtt{y})$$

$$\wedge\ \mathtt{sel}(h_2, \mathtt{x}) = \mathtt{upd}(\mathtt{sel}(h_1, \mathtt{x}), \text{“}\mathtt{f}\text{”}, 7)$$
$$\wedge\ \forall \mathtt{y}.\ \mathtt{x} \neq \mathtt{y} \Rightarrow \mathtt{sel}(h_2, \mathtt{y}) = \mathtt{sel}(h_1, \mathtt{y})$$

Heap Updates…

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



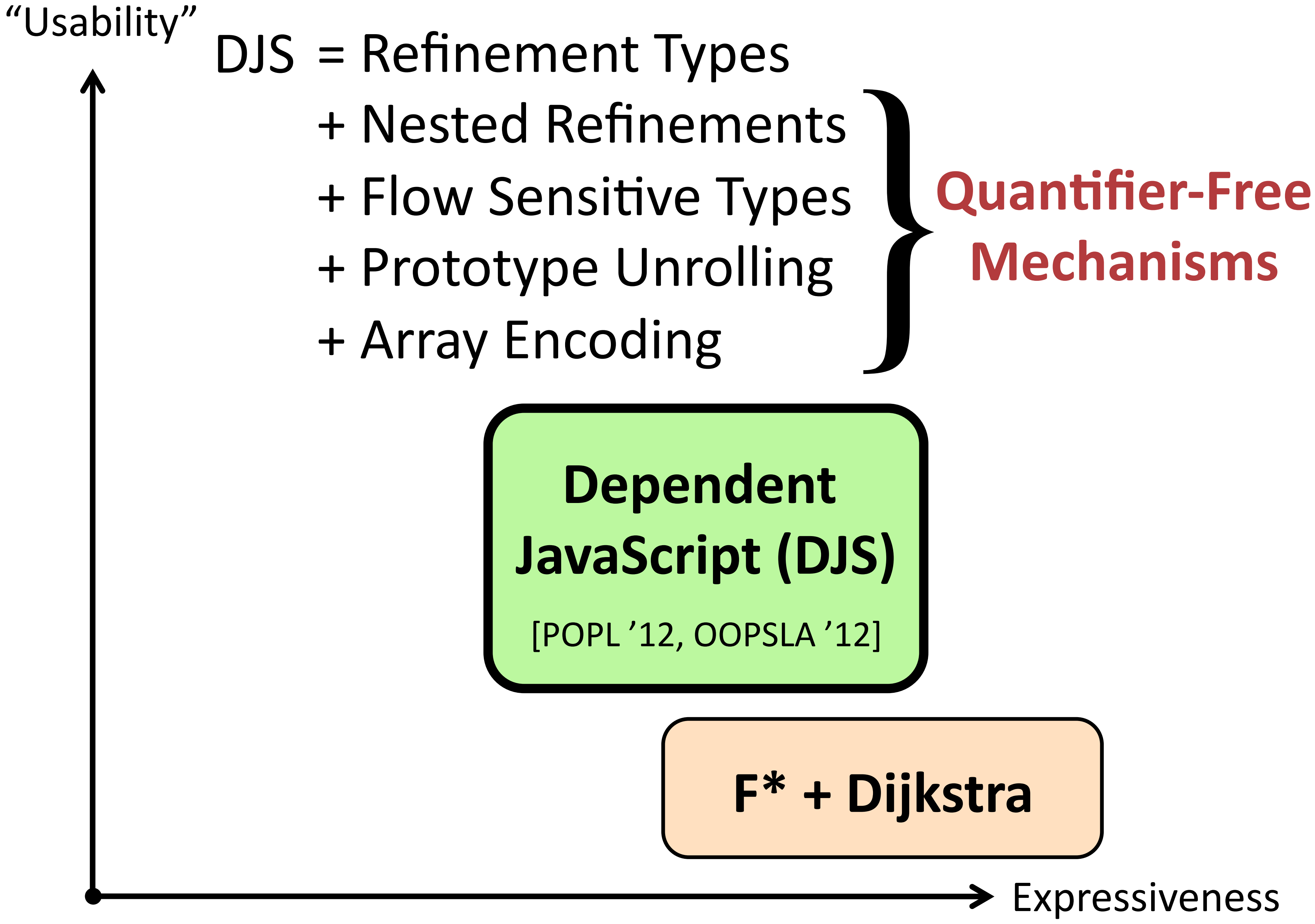
… Without Quantifiers!

Flow-Sensitive Types (à la Alias Types)

$$x : \underset{\substack{\uparrow \\ \text{input type}}}{T_1} / \underset{\substack{\uparrow \\ \text{input heap}}}{H_1} \rightarrow \underset{\substack{\uparrow \\ \text{output type}}}{T_2} / \underset{\substack{\uparrow \\ \text{output heap}}}{H_2}$$

Prototype Inheritance…
Array Semantics…

… Without Quantifiers!



“Usability”

DJS = Refinement Types
+ Nested Refinements
+ Flow Sensitive Types
+ Prototype Unrolling
+ Array Encoding

Quantifier-Free Mechanisms

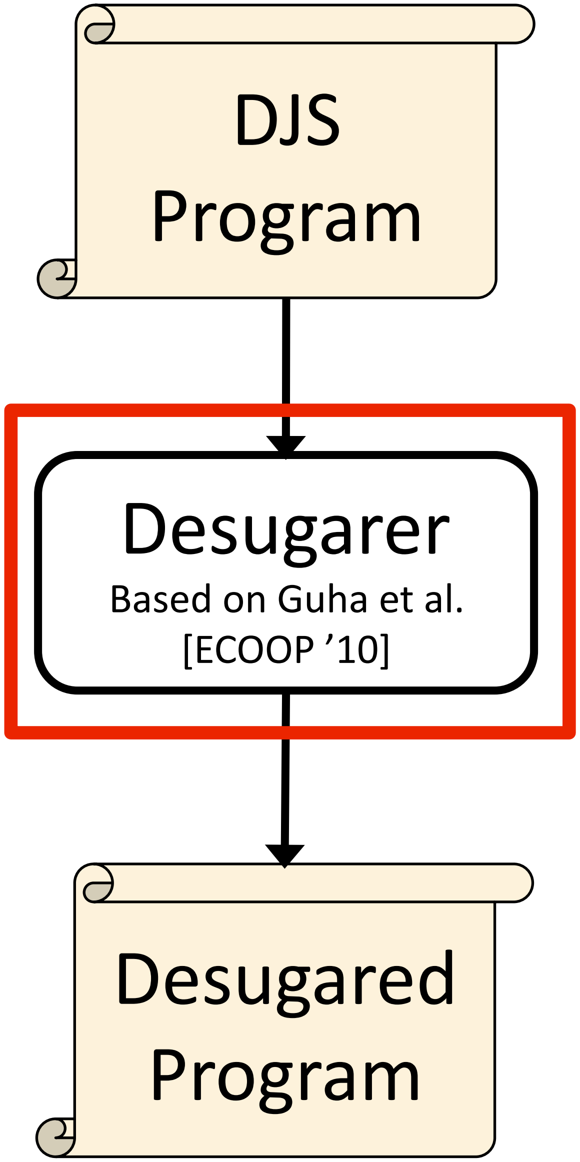
Dependent JavaScript (DJS)

[POPL ’12, OOPSLA ’12]

F* + Dijkstra

Expressiveness

Implementation



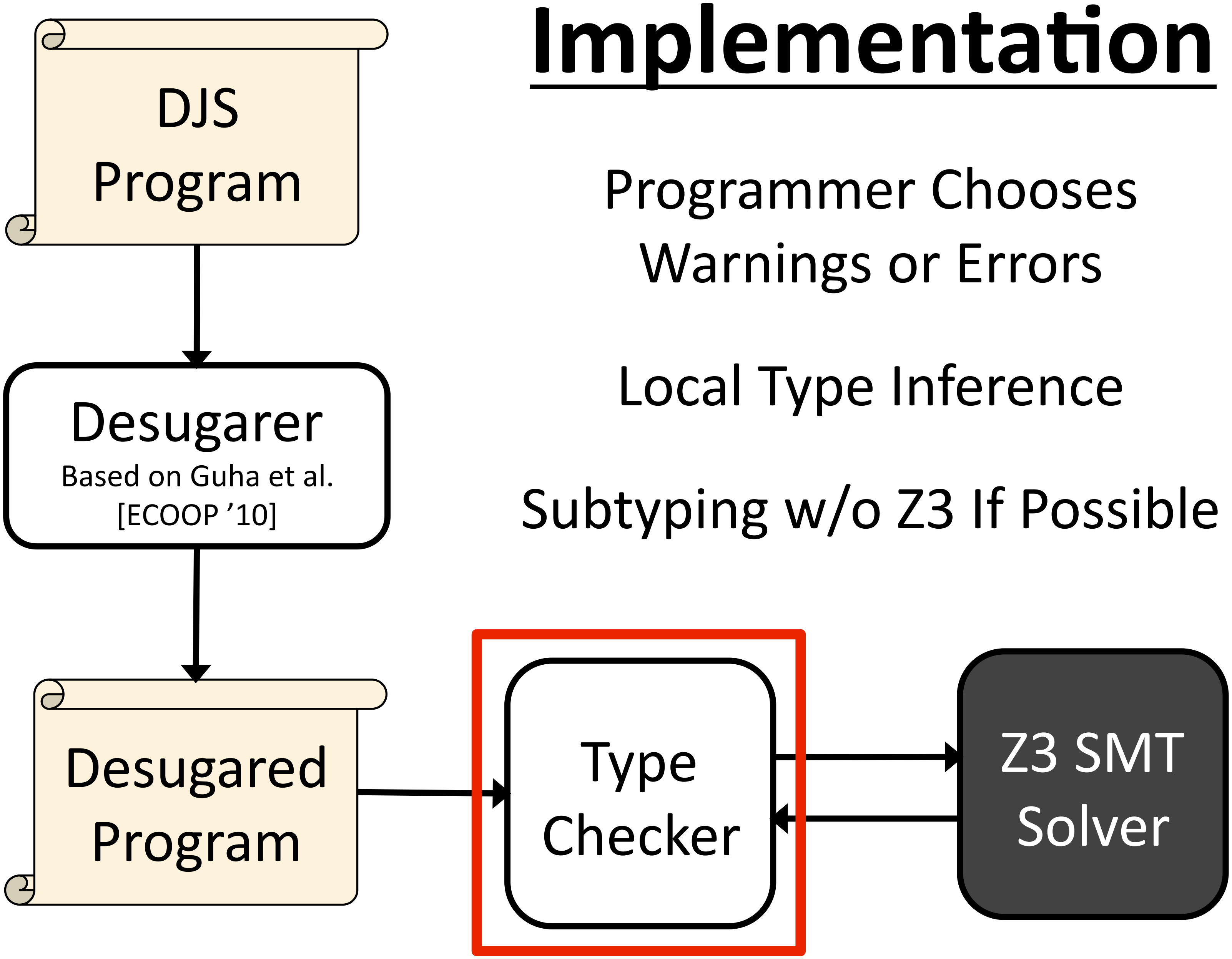
JavaScript → λ-Calculus + References + Prototypes

Implementation

Programmer Chooses Warnings or Errors

Local Type Inference

Subtyping w/o Z3 If Possible



Benchmarks	LOC before/after	
13 Excerpts from: *JavaScript, Good Parts* SunSpider Benchmark Suite Google Closure Library	306	408 (+33%)

Chosen to **Stretch** the Current Limits of DJS

Benchmarks	**LOC before/after**	
13 Excerpts from: *JavaScript, Good Parts* SunSpider Benchmark Suite Google Closure Library	306	408 (+33%)
9 Browser Extensions from: [Guha et al. Oakland ’11]	321	383 (+19%)
2 Examples from: Google Gadgets	1,003	1,027 (+2%)
TOTALS	**1,630**	**1,818** **(+12%)**

Already Improved by Simple **Type Inference** and **Syntactic Sugar**

Plenty of **Room for Improvement**

- Iterative Predicate Abstraction
- Bootstrap from **Run-Time Traces**

TOTALS	**1,630**	**1,818** **(+12%)**

Benchmarks	LOC before/after		Running Time
13 Excerpts from: *JavaScript, Good Parts* SunSpider Benchmark Suite Google Closure Library	306	408 (+33%)	10 sec
9 Browser Extensions from: [Guha et al. Oakland '11]	321	383 (+19%)	3 sec
2 Examples from: Google Gadgets	1,003	1,027 (+2%)	19 sec
TOTALS	**1,630**	**1,818 (+12%)**	**32 sec**

Already Improved by Simple **Optimizations**

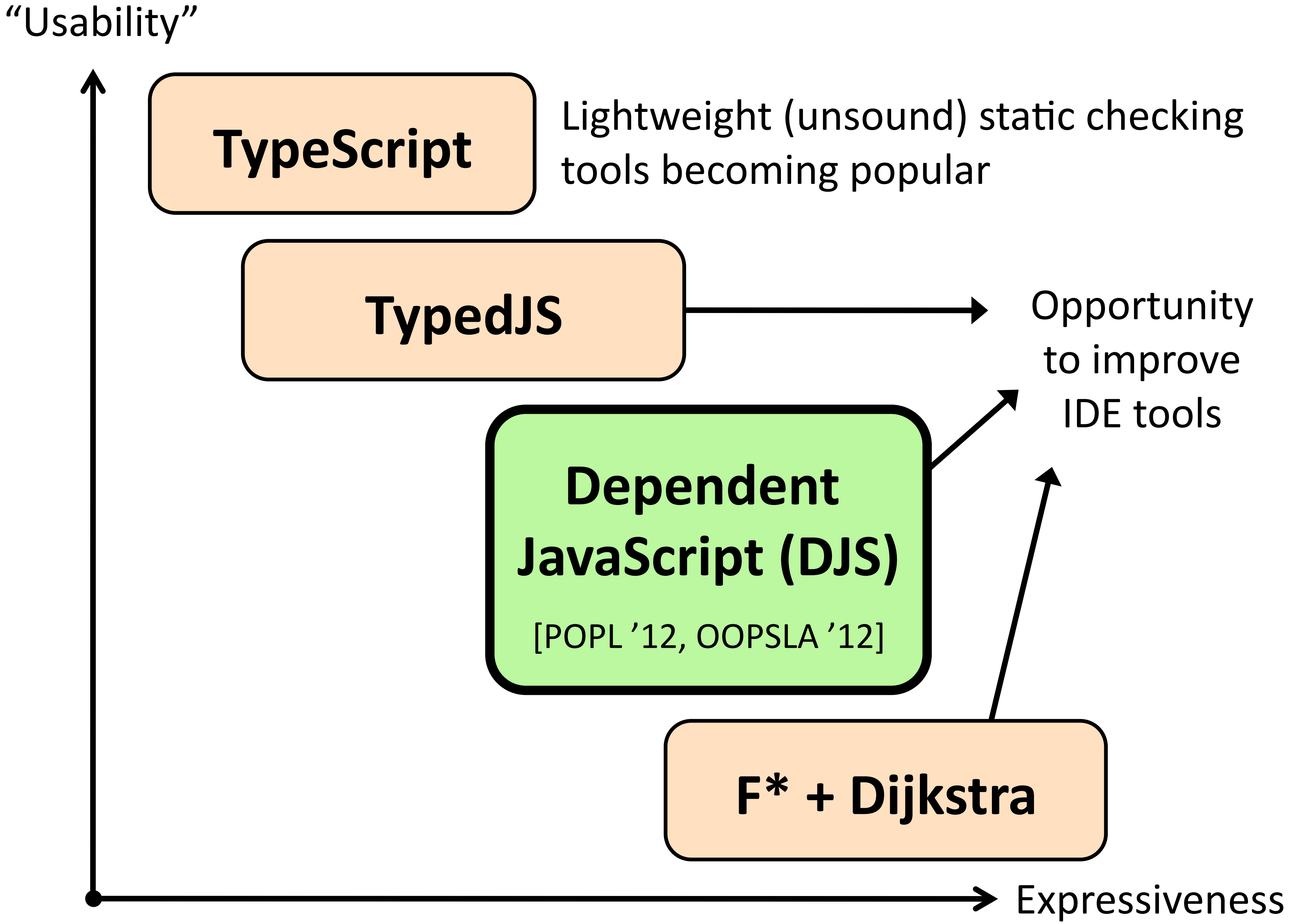
- Avoid SMT Solver When Possible
- Reduce Precision for Common Patterns

Plenty of **Room for Improvement**

TOTALS	**1,630**	**1,818** **(+12%)**	**32 sec**

Types for JavaScript

1. Better Development Tools
2. Better Reliability
3. Better Performance



"Usability"
TypeScript
Lightweight (unsound) static checking tools becoming popular
TypedJS
Opportunity to improve IDE tools
Dependent JavaScript (DJS)
[POPL '12, OOPSLA '12]
F* + Dijkstra
Expressiveness

Reliability / Security

- Refinement types for security in presence of untrusted code (e.g. browser extensions)
- Combine with static reasoning for JavaScript

Performance

- JITs use static analysis + profiling to optimize dynamic features (e.g. dictionaries, bignums)
- Opportunity to enable more optimizations

Thanks!

Types for JavaScript

1. Better Development Tools
2. Better Reliability
3. Better Performance

}

DJS is a Step Towards These Goals

`ravichugh.com/djs`