Towards Dependent Types for JavaScript

Ravi Chugh, David Herman, Ranjit Jhala

University of California, San Diego

Mozilla Research

Explicit

Decidable

A Large Subset of

Types for JavaScript

Features Common
to All Editions

Outline

Challenges

Our Approach

Preliminary Results

Challenge 1: Reflection

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

x should be "num-or-bool"

Challenge 2: Mutation

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

Different types stored in x

Challenge 3: Coercions

```
3   + 4    // 7
"3" + "4"  // "34"
3   + "4"  // "34"
```

Challenge 3: Coercions

```
!true   // false

!1      // false

!“”     // true
```

Challenge 3: Coercions

```
0 === 0    // true
0 === “”   // false
0 ==  “”   // true
```

Challenge 4: Objects

Mutable

```
var par   = {}
var child = Obj.create(par)
child.f   = 1
```

Dynamic Keys

```
var g     = "g"
child[g]  = 2
child.g  // 2
```

Prototypes

```
child.h  // undefined
par.h     = 3
child.h  // 3
```

Challenge 5: Arrays

Finite or Unknown “Length”

```
var nums = [0,1,2]
nums[0] + nums[1] + nums[2]
```

```
delete nums[1]
```

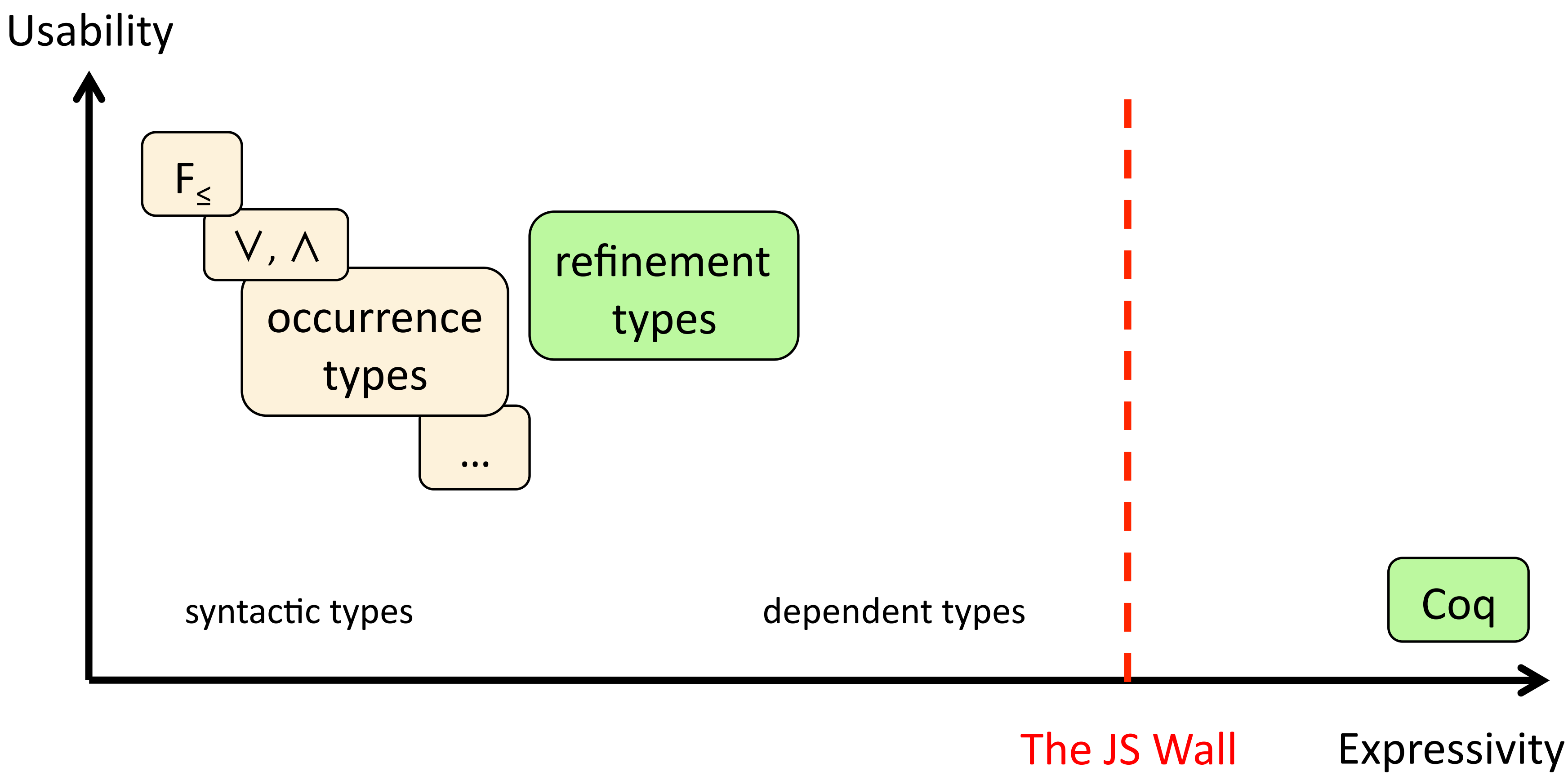
“Packed” or “Unpacked”

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

```
nums.push(42)
```

Prototype-based

Prior Approaches



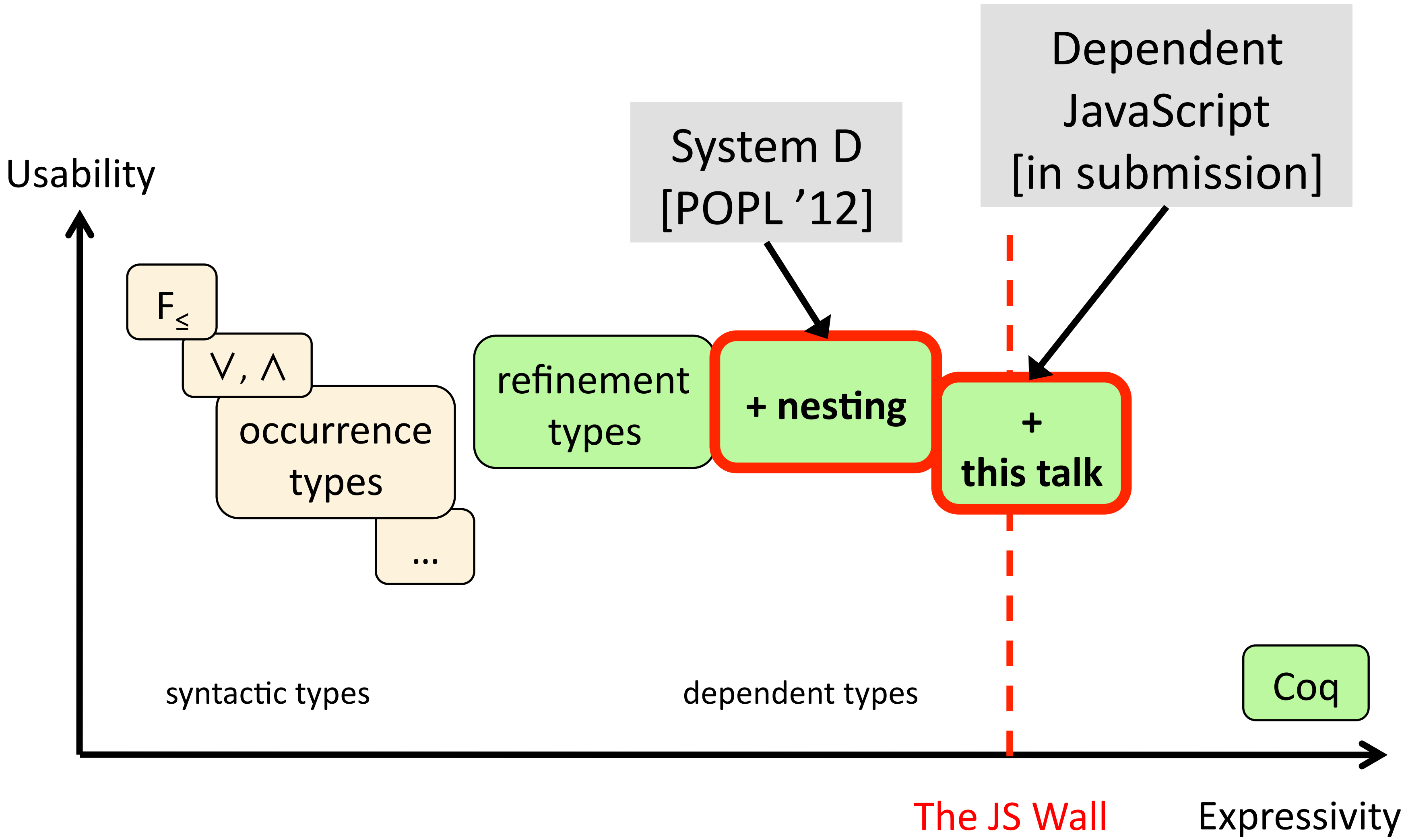
Outline

Challenges

Our Approach

Preliminary Results

Our Approach



Refinement Types

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

"value x such that formula p is true"

```
Bool ≡ { b | tag(b) = "boolean" }

 Num ≡ { n | tag(n) = "number" }

 Int ≡ { i | tag(i) = "number" ∧ integer(i) }

 Any ≡ { x | true }
```

Refinement Types

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

"value x such that formula p is true"

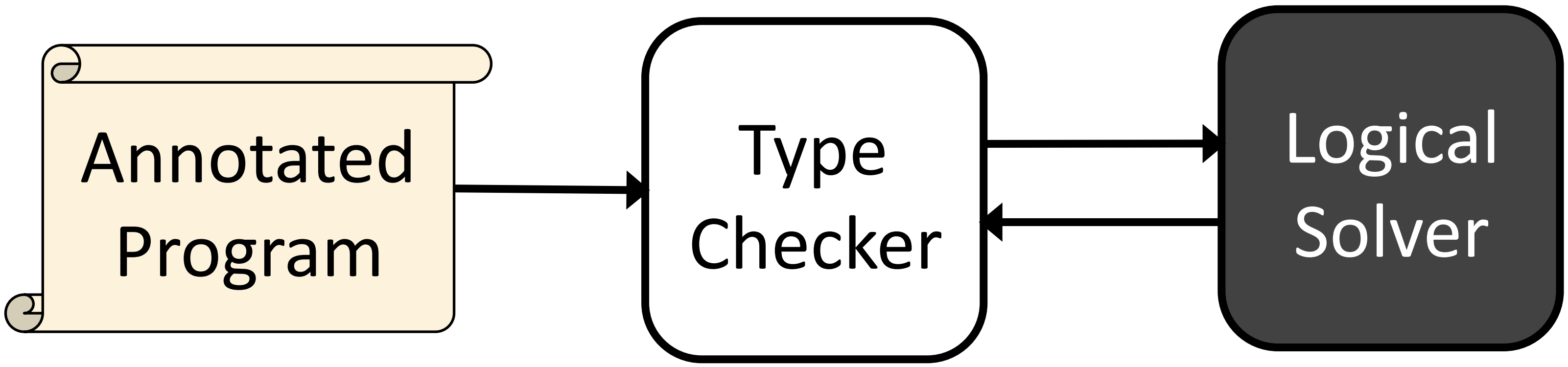
```
3 :: Num

3 :: Int

3 :: {i | i > 0}

3 :: {i | i = 3}
```

Subtyping is Implication

$\{ i \mid i = 3 \}$ <: $\{ i \mid i > 0 \}$ <: Int <: Num

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

Subtyping is Implication



System D [POPL 2012]

```
var obj = { "n": 17,
            "f": function (i) { return i + 5 }}
```

```
obj ::  { d | tag(d) = "Dict"
            ∧ tag(sel(d,"n")) = "number"
            ∧ sel(d,"f") :: Int → Int }
```

McCarthy's decidable theory of arrays

Great for dictionaries of base values

System D [POPL 2012]

```
var obj = { "n": 17,
            "f": function (i) { return i + 5 }}
```

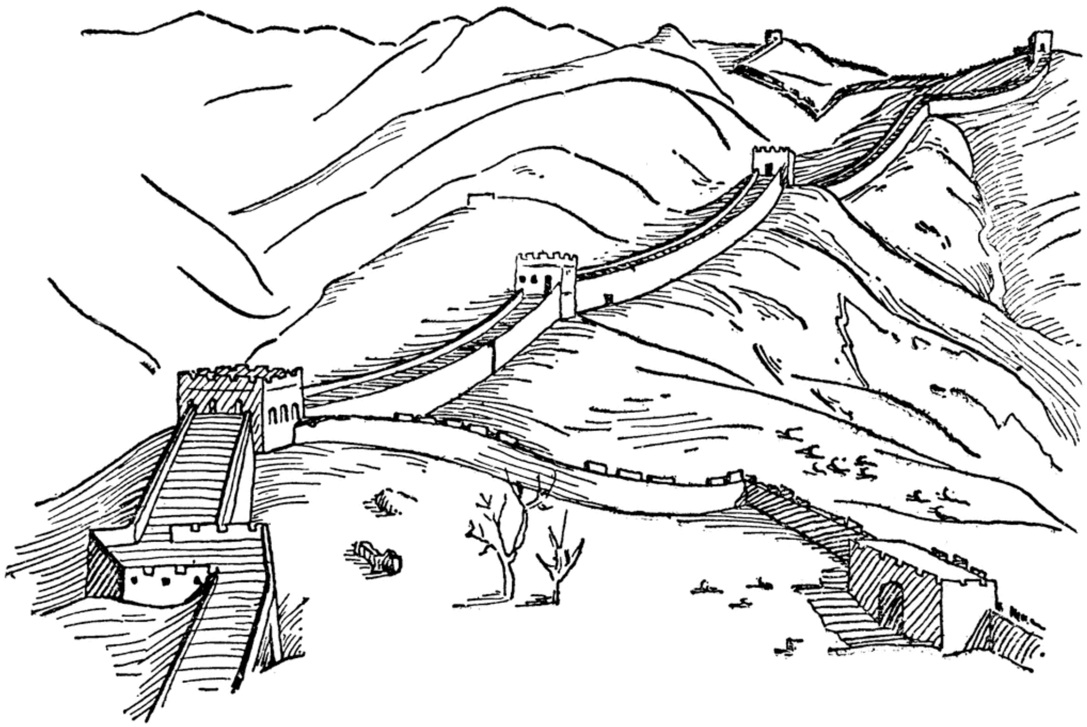
```
obj ::  { d | tag(d) = "Dict"
              ∧ tag(sel(d,"n")) = "number"
              ∧ sel(d,"f") :: Int → Int }
```

Uninterpreted "has-type" predicate

Type constructors in formulas

Subtyping algorithm retains precision and decidability

System D



JavaScript

photo courtesy of ClipArt ETC

System D

\+ Types for JS Primitives

\+ Strong Updates

\+ Prototype Inheritance

\+ Arrays

JavaScript

System D

\+ Types for JS Primitives

\+ Strong Updates

\+ Prototype Inheritance

\+ Arrays

Dependent JavaScript (DJS)

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

Choose degree of **precision** and **coercion**

```
!  ::    Bool → Bool
```

Choose degree of **precision** and **coercion**

```
! :: x:Bool → {b | if x = false
                   then b = true
                   else b = false }
```

Choose degree of **precision** and **coercion**

$$! :: x\!:\!\mathrm{Any} \rightarrow \{\, b \mid \text{if } \mathrm{falsy}(x) \text{ then } b = \mathrm{true} \text{ else } b = \mathrm{false} \,\}$$

$$\mathrm{falsy}(x) \equiv x = \mathrm{false} \vee x = 0 \vee x = \mathrm{null} \vee x = \text{“”} \vee x = \mathrm{undefined} \vee x = \mathrm{NaN}$$

Primitives	**Strong Updates**	Prototypes	Arrays

Flow Sensitive

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

Path Sensitive

(even with ordinary refinement types)

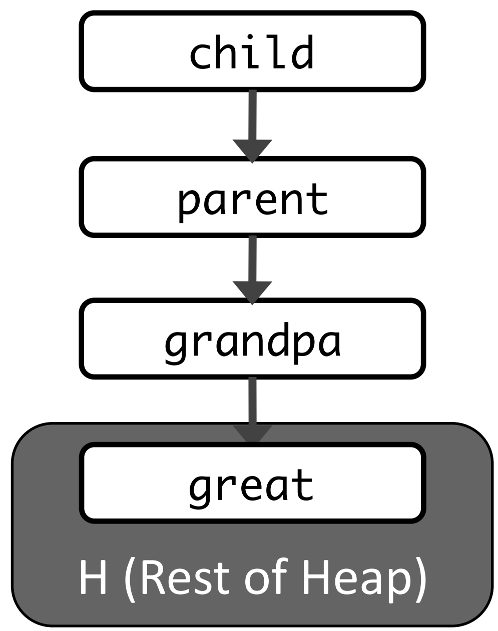
NumOrBool → NumOrBool

x:NumOrBool → {y | tag(y) = tag(x)}

Key Membership via Prototype Chain Unrolling

```
var grandpa = …,
    parent  = Object.create(grandpa),
    child   = Object.create(parent),
    b       = k in child,
```

$$b :: \{ v \mid v = \text{true iff } (\text{has}(\text{child}, k) \lor \text{has}(\text{parent}, k) \lor \text{has}(\text{grandpa}, k) \lor \text{HeapHas}(H, \text{great}, k)) \}$$



Key Lookup via Prototype Chain Unrolling

```
var grandpa = …,
    parent  = Object.create(grandpa),
    child   = Object.create(parent),
    b       = k in child,
    x       = child[k]
```

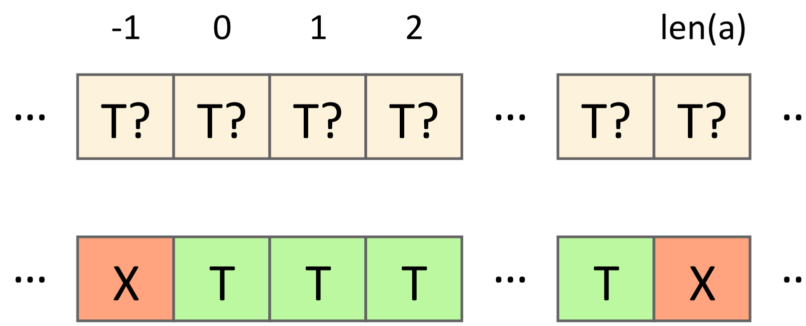
```
x :: {v | if has(child,k) then v = sel(child,k)
          elif has(parent,k) then v = sel(parent,k)
          elif has(grandpa,k) then v = sel(grandpa,k)
          elif HeapHas(H,great,k)) then v = HeapSel(H,great,k))
          else v = undefined }
```

Key Idea

Reduce prototype semantics
to decidable theory of arrays
via flow-sensitivity and unrolling

Track **types**, “**packedness**,” and **length** of arrays where possible

$$\{\, a \mid a :: \mathrm{Arr}(T) \;\land\; \mathrm{packed}(a) \;\land\; \mathrm{len}(a) = 10 \,\}$$



$$T? \;\equiv\; \{\, x \mid T(x) \lor x = \mathrm{undefined} \,\}$$

$$X \;\equiv\; \{\, x \mid x = \mathrm{undefined} \,\}$$

Encode **tuples** as arrays

```
var tup = [17, “ni hao”]
```

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

Re-use **prototype** mechanism

```
var tup = [17, "ni hao"]
tup.push(true)
```

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

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

$$\wedge\ \ldots \}$$

Recap of DJS Tricks

Uninterpreted Functions

Flow Sensitivity

Prototype Unrolling

Refinement Type Encodings

Outline

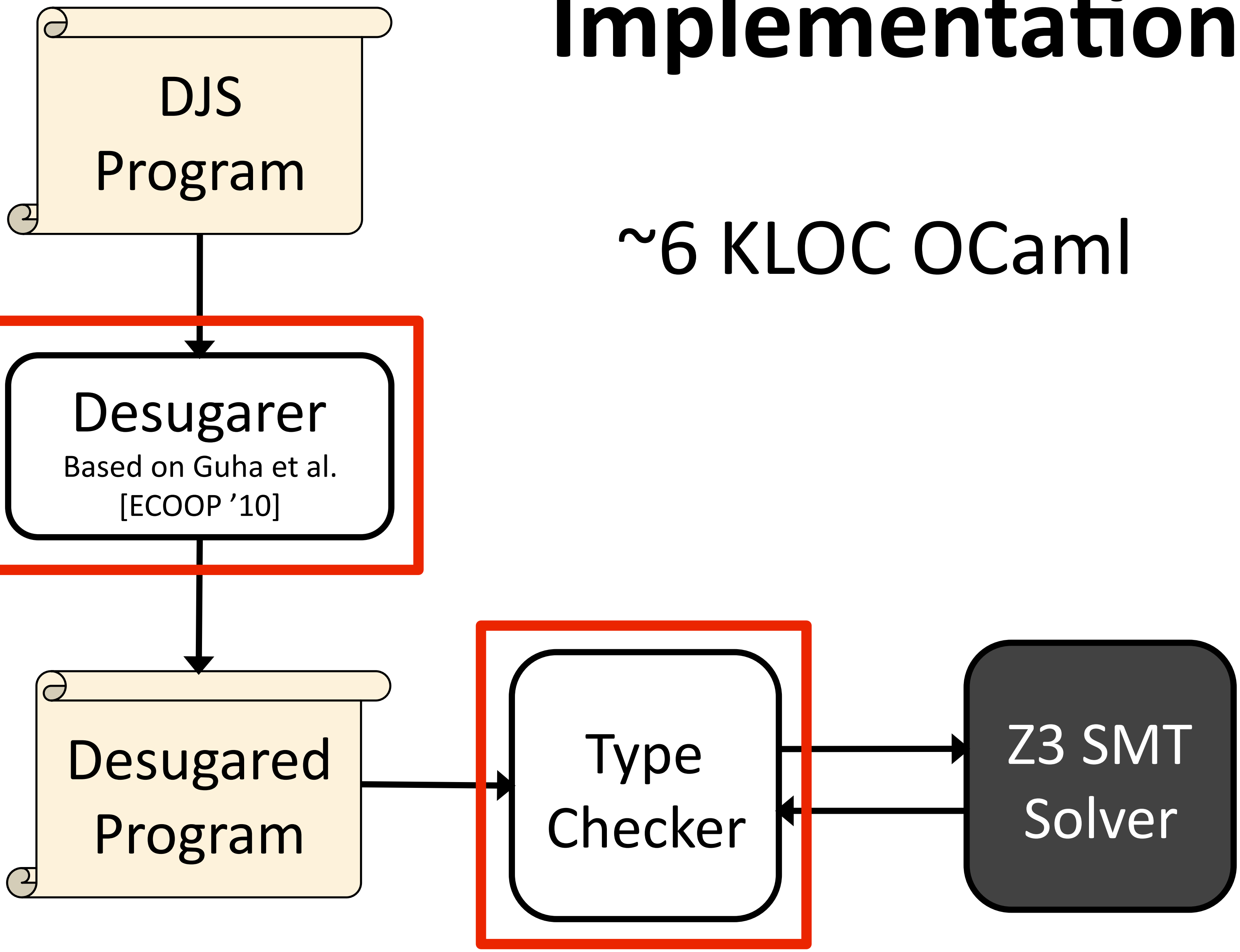
Challenges

Our Approach

Preliminary Results

Implementation

~6 KLOC OCaml

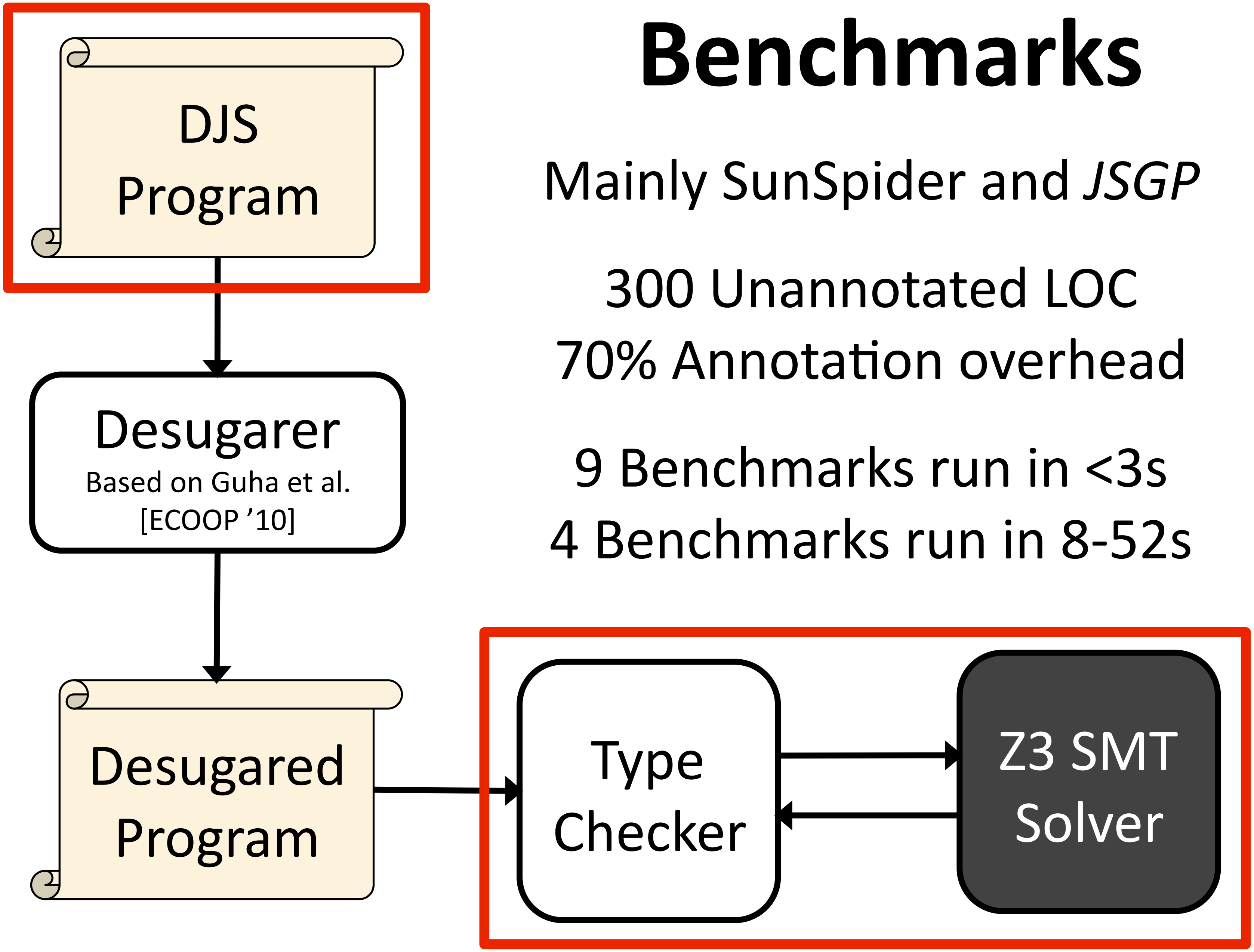


Benchmarks

Mainly SunSpider and *JSGP*

300 Unannotated LOC
70% Annotation overhead

9 Benchmarks run in <3s
4 Benchmarks run in 8-52s

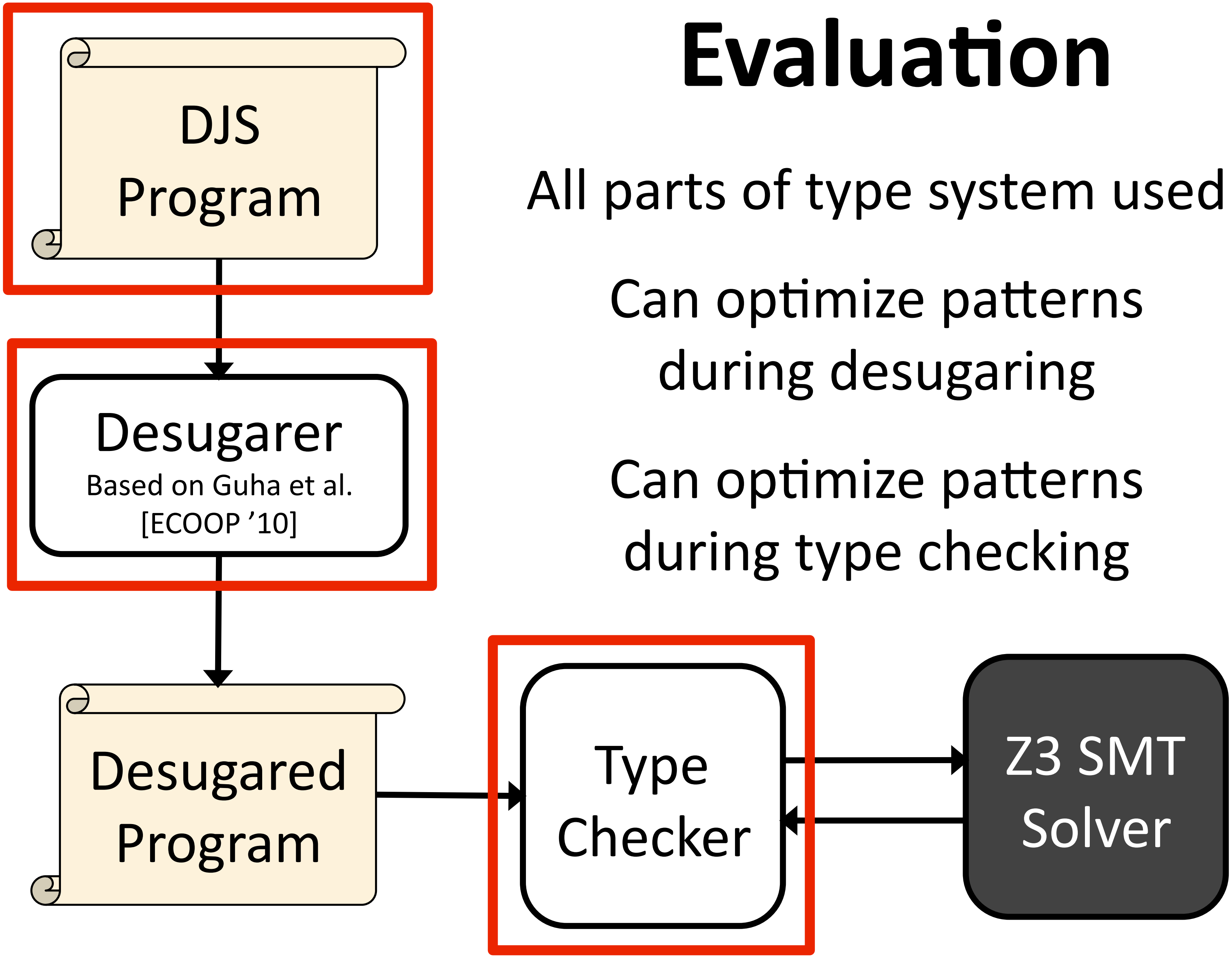


Evaluation

All parts of type system used

Can optimize patterns during desugaring

Can optimize patterns during type checking



Conclusion

DJS is a step **towards** climbing the JS Wall

Thanks!

ravichugh.com/nested

github.com/ravichugh/djs

$\{p\} \equiv \{v \mid p\}$

```
/*: 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) ∨ HeapHas(H,d’,k)

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

Q: What is “Duck Typing”?

Structural Object Types

+

Logical Reasoning ?