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

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

X should be "num-or-bool"
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**

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

**Dynamic Keys**

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

**Prototypes**

```javascript
child.h // undefined
par.h = 3;
child.h // 3
```
Challenge 5: Arrays

```javascript
var nums = [0, 1, 2];
```

```javascript
delete nums[1]  // “Packed” or “Unpacked"
```

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

```javascript
nums.push(42)  // Prototype-based
```

Finite or Unknown “Length”

“Packed” or “Unpacked”
Prior Approaches

Usability

syntactic types

occurrence types

\( F_\leq \)

\( \lor, \land \)

refinement types

dependent types

Expressivity

The JS Wall

Coq
Outline

Challenges

Our Approach

Preliminary Results
Our Approach

Dependent JavaScript [in submission]

System D [POPL '12]

+ nesting
+ this talk

F≤, V, ∧
occurrence types
... refinement types
dependent types
syntactic types

The JS Wall
Expressivity
Usability

Coq
Refinement Types

\{ x : \# p \}\n
"value X such that formula p is true"

Bool ≡ \{ b | \text{tag}(b) = \text{"boolean"} \}

Num ≡ \{ n | \text{tag}(n) = \text{"number"} \}

Int ≡ \{ i | \text{tag}(i) = \text{"number"} \land \text{integer}(i) \}

Any ≡ \{ x | \text{true} \}
Refinement Types

\{(x \mid p)\}

“value \(x\) such that formula \(p\) is true”

3 :: Num
3 :: Int
3 :: \(\{i \mid i > 0\}\)
3 :: \(\{i \mid i = 3\}\)
Subtyping is Implication

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

i = 3
⇒ i > 0
⇒ \text{tag}(i) = \text{“number”} \land \text{integer}(i)
⇒ \text{tag}(i) = \text{“number”}
Subtyping is Implication

- Annotated Program
- Type Checker
- Logical Solver
System D [POPL 2012]

```javascript
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
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 }

Type constructors in formulas
Subtyping algorithm retains precision and decidability

Uninterpreted “has-type” predicate
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)
<table>
<thead>
<tr>
<th>Primitives</th>
<th>Strong Updates</th>
<th>Prototypes</th>
<th>Arrays</th>
</tr>
</thead>
</table>

+ Types for JS Primitives
+ Strong Updates
+ Prototype Inheritance
+ Arrays
Choose degree of precision and coercion

! :: Bool → Bool
<table>
<thead>
<tr>
<th>Primitives</th>
<th>Strong Updates</th>
<th>Prototypes</th>
<th>Arrays</th>
</tr>
</thead>
</table>

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

! :: \(x: \text{Bool} \rightarrow \{ b \mid \text{if } x = \text{false} \text{ then } b = \text{true} \text{ else } b = \text{false} \} \)
Choose degree of precision and coercion

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

\[
falsy(x) = x = \text{false} \lor x = 0 \lor x = \text{null} \lor x = \text{undefined} \lor x = \text{NaN}
\]
function negate(x) {
  if (typeof x == "number")
    x = 0 - x
  else
    x = !x
  return x
}

NumOrBool → NumOrBool
x:NumOrBool → { y | tag(y) = tag(x) }


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

b :: { v | v = true iff  
(has(child,k) v  
(has(parent,k) v  
(has(grandpa,k) v  
HeapHas(H,great,k)) }  
```

Key Membership 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

\[
\{ a \mid a :: \text{Arr}(T) \} \quad \wedge \text{packed}(a) \quad \wedge \text{len}(a) = 10
\]

\[
\begin{array}{cccccc}
-1 & 0 & 1 & 2 & \ldots & \text{len}(a) \\
T? & T? & T? & T? & \ldots & T? \quad T? \quad \ldots \\
X & T & T & T & \ldots & T \quad X \quad \ldots
\end{array}
\]

\[
T? = \{ x \mid T(x) \lor x = \text{undefined} \}
\]

\[
X = \{ x \mid x = \text{undefined} \}
\]
Encode **tuples** as arrays

```plaintext
var tup = [17, “ni hao”]

{ a | a :: Arr(Any)
  ^ packed(a) ^ len(a) = 2
  ^ Int(sel(a,0))
  ^ Str(sel(a,1)) }
```
Re-use **prototype** mechanism

var tup = [17, “ni hao”]
tup.push(true)

{ a | a :: Arr(Any) ∧ packed(a) ∧ len(a) = 3 ∧ ...
}

Arrays
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
/**: 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**

\[
\text{x: } T_1/H_1 \rightarrow T_2/H_2
\]

- Input type
- Input heap
- Output heap
- Output type

\[
\text{ObjHas}(d,k,H,d') \equiv \text{has}(d,k) \lor \text{HeapHas}(H,d',k)
\]

/\*: x:Ref / [x \mapsto d:Dict |> ^x] \\
\rightarrow \{v \iff \text{ObjHas}(d,"f",\text{curHeap},^x)\} / \text{sameHeap} */

```javascript
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**

\[
\text{ObjSel}(d,k,H,d') \equiv \text{ite has}(d,k) \text{ sel}(d,k) \text{ HeapSel}(H,d',k)
\]

/\*: x:Ref / [x \mapsto d:Dict \mapsto ^x] \\
\rightarrow \{ v = \text{ObjSel}(d,"f",\text{curHeap},^x) \} / \text{sameHeap} */

```
function readF(x) {
    return x.f
}
```
Q: What is “Duck Typing”?

Structural Object Types

+Logical Reasoning?