Dependent Types for JavaScript

Ravi Chugh    Ranjit Jhala
University of California, San Diego

David Herman
Mozilla Research
Why JavaScript?

Pervasive

Used at Massive Scale
var x = {};  

x.f;  

x.f.f;  

produces undefined rather than error...

... but this raises TypeError
Why Add Types?

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

implicit global object

var lifting

scope manipulation

‘,,,’ == new Array(4)
JavaScript

implicit global object

var liYing',,, == new Array(4)

objects
prototypes
type-tests
lambda

scope manipulation

var lifting
JavaScript

"The Good Parts"

objects
prototypes

',,,' == new Array(4)
JavaScript

"The Good Parts"

Our Approach

Key Idea: Use Logic!

Prior Type Systems
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”

\[
\begin{align*}
\text{Num} &= \{ n \mid \text{tag}(n) = \text{“number”} \} \\
\text{NumOrBool} &= \{ v \mid \text{tag}(v) = \text{“number”} \lor \text{tag}(v) = \text{“boolean”} \} \\
\text{Int} &= \{ i \mid \text{tag}(i) = \text{“number”} \land \text{integer}(i) \} \\
\text{Any} &= \{ x \mid \text{true} \}
\end{align*}
\]
Refinement Types

Syntactic Sugar for Common Types

\[
\begin{align*}
\text{Num} & \equiv \{ n \mid \text{tag}(n) = \text{"number"} \} \\
\text{NumOrBool} & \equiv \{ v \mid \text{tag}(v) = \text{"number"} \lor \text{tag}(v) = \text{"boolean"} \} \\
\text{Int} & \equiv \{ i \mid \text{tag}(i) = \text{"number"} \land \text{integer}(i) \} \\
\text{Any} & \equiv \{ x \mid \text{true} \}
\end{align*}
\]
Refinement Types

3 :: \{ n \mid n = 3 \}
3 :: \{ n \mid n > 0 \}
3 :: \{ n \mid \text{tag}(n) = \text{"number"} \land \text{integer}(n) \}
3 :: \{ n \mid \text{tag}(n) = \text{"number"} \}
Refinement Types

Subtyping is Implication

\{ n \mid n = 3 \}
\lll \{ n \mid n > 0 \}
\lll \{ n \mid \text{tag}(n) = "number" \land \text{integer}(n) \}
\lll \{ n \mid \text{tag}(n) = "number" \}
Refinement Types

\[ n = 3 \]
\[ \Rightarrow n > 0 \]
\[ \Rightarrow \text{tag}(n) = \text{“number”} \land \text{integer}(n) \]
\[ \Rightarrow \text{tag}(n) = \text{“number”} \]

Subtyping is Implication
<table>
<thead>
<tr>
<th>Tag-Tests</th>
<th>Duck Typing</th>
<th>Mutable Objects</th>
<th>Prototypes</th>
<th>Arrays</th>
</tr>
</thead>
</table>

```javascript
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 - x; // -2
}

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

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

Use types to prevent implicit coercion

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

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

DJS is Path Sensitive
//: negate :: (x: Any) → Any

```javascript
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 :: \(x:\text{NumOrBool}\) \rightarrow \text{Any}

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

var negate = function(x) {
  if (typeof x == "boolean")
    return !x;
  else
    return 0 - x;
}
```
```javascript
/*: 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
Programmer chooses degree of precision in specification

```plaintext
/*: negate :: (x:NumOrBool) → {v | tag(v) = tag(x)} */
```
What is “Duck Typing”?

```javascript
if (duck.quack)
  return "Duck says " + duck.quack();
else
  return "This duck can’t quack!";
```
What is “Duck Typing”? 

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

(+) :: (Num,Num) → Num
(+) :: (Str,Str) → Str
What is “Duck Typing”?

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

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

\{ d \mid tag(d) = \text{“Dict”} \land \\
    \text{has}(d, \text{“quack”}) \Rightarrow \\
    \text{sel}(d, \text{“quack”}) :: \text{Unit} \to \text{Str} \}\n
Operators from McCarthy theory of arrays

| Tag-Tests | Duck Typing | Mutable Objects | Prototypes | Arrays |
\begin{align*}
\{ d \mid & \text{tag}(d) = \text{"Dict"} \land \\
& \text{has}(d, \text{"quack"}) \Rightarrow \\
& \text{sel}(d, \text{"quack"}) :: \text{Unit} \rightarrow \text{Str} \} \\
\end{align*}

Call produces \text{Str}, so concat well-typed

\begin{verbatim}
if (duck.quack)
    return "Duck says " + duck.quack();
else
    return "This duck can’t quack!";
\end{verbatim}
DJS is Flow Sensitive

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

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

McCarthy operator

```
x_0 : Empty
x_1 : {d | d = upd(x_0, "f", 7)}
```
DJS is **Flow** Sensitive

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

**Strong** updates to singleton objects

**Weak** updates to collections of objects
<table>
<thead>
<tr>
<th>Tag-Tests</th>
<th>Duck Typing</th>
<th>Mutable Objects</th>
<th>Prototypes</th>
<th>Arrays</th>
</tr>
</thead>
</table>

Typical "Dynamic" Features
Typical “Dynamic” Features

JavaScript

tl;dr
Harder, but DJS handles them
Upon construction, each object links to a prototype object.
Semantics of Key Lookup

```javascript
var k = "first";
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

\[
\text{child}[k];
\]

\[
\{ v \mid \text{if \ has(child, k) then} \\
\quad v = \text{sel(child, k)} \\
\text{Else if parent contains k, then} \\
\quad \text{Read k from parent} \\
\text{Else if grandpa contains k, then} \\
\quad \text{Read k from grandpa} \\
\text{Else if ...} \\
\text{Else} \\
\quad \text{Return undefined}
\]

child

→

parent

→

grandpa

→

null
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
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 \mid \text{if has}(\text{child}, k) \text{ then } v = \text{sel}(\text{child}, k) \\
    \text{else if has}(\text{parent}, k) \text{ then } v = \text{sel}(\text{parent}, k) \\
    \text{else } v = \text{HeapSel}(H, \text{grandpa}, k) \}\}

Abstract predicate to summarize the unknown portion of the prototype chain.
```javascript
var k = "first"; child[k];
```
```
var k = "last"; child[k];
```

```
child
{ "first" : "John" }

parent
{ "first" : "Ida" , "last" : "McCarthy" }

grandpa
???

H
(Rest of Heap)

null
```

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

*... extended to unbounded collection...*
```javascript
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

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

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

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

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

```java
var tup = [17, “cacti”]
```

```java
{ a | a :: Arr(Any) ∧ packed(a) ∧ len(a) = 2 ∧ Int(sel(a,0)) ∧ Str(sel(a,1)) }
```
var tup = [17, "cacti"]
tup[tup.length] = true

{ a | a :: Arr(Any) ∧ packed(a) ∧ len(a) = 3 ∧ ... }
DJS handles other **quirks**:

- Special length property
- `Array.prototype`
- Non-integer keys
<table>
<thead>
<tr>
<th>Tag-Tests</th>
<th>Duck Typing</th>
<th>Mutable Objects</th>
<th>Prototypes</th>
<th>Arrays</th>
</tr>
</thead>
</table>

What About `eval`?

... Old Types

`eval("...");`

... Arbitrary code loading
What About eval?

```
... eval("...");
//: #assume
```

"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

“The Good Parts”
- arrays
- objects
- prototypes
- type-tests
- lambdas
13 Benchmarks

“The Good Parts”

arrays

objects

prototypes

type-tests

lambdas
13 Benchmarks

Four inheritance patterns from Crockford’s *JS: The Good Parts*
13 Benchmarks

Array-manipulating examples from SunSpider
type0f() function to replace typeof from Closure Compiler
13 Benchmarks

Well-typed programs don’t have run-time errors
Implementation

DJS Program

Desugarer
Based on Guha et al.
[ECOOP 2010]

Desugared Program

JavaScript $\rightarrow$ $\lambda$-Calculus + References + Prototypes
Implementation

Programmer Chooses Warnings or Errors

Local Type Inference

Subtyping w/o Z3 If Possible

DJS Program

Desugarer
Based on Guha et al. [ECOOP 2010]

Desugared Program

Type Checker

Z3 SMT Solver
Annotation Burden
(Improved since paper)

$\sim 300$ LOC to start
$+ \sim 100$ LOC annotations
= $\sim 400$ 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
Coq refinement types

Dependent types

Expressivity

System D [POPL ’12]

Dependent JavaScript [this paper]

"Usability"

syntactic types

occurrence types

F≤

∨, ∧

refinement types

DJS

+ nested refinements

Coq

Dependent

Expressivity

JS
System D

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

Dependent JavaScript (DJS)
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

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

\[
\text{hasF}(x) = \{ v \text{ iff ObjHas}(d, "f", \text{curHeap}, x.\text{pro}) \} / \text{sameHeap}
\]

```javascript
function hasF(x) {
  return "f" in x
}
```
Function Types and Objects

\[ x : T_1/H_1 \rightarrow T_2/H_2 \]

\[ \text{input type} \quad \uparrow \quad \text{input heap} \quad \uparrow \quad \text{output type} \quad \uparrow \quad \text{output heap} \]

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

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

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

negate("2")
```
Whoa, but perhaps okay…
### Tag-Tests

| Duck Typing | Mutable Objects | Prototypes | Arrays |

---

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

Error would be nicer, but okay...
```javascript
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!

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

negate([])
```

WAT?!
if (duck.quack)
    return “Duck says “ + duck.quack();
else
    return “This duck can’t quack!”;
var x = {};

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

1) Possible unbound keys
```javascript
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