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;
```

... but this raises `TypeError`

produces undefined rather than error...
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

implicit global object

var lifting

scope manipulation

‘,,,’ == new Array(4)
JavaScript

"The Good Parts"

arrays
objects
prototypes
type-tests

lambda

eval()

‘,,,’ == new Array(4)
JavaScript

“The Good Parts”

Dependent JavaScript

Use Logic, but
Avoid Quantifiers!
Usability

Expressiveness

TypedJS

Dependent JavaScript (DJS) [POPL ’12, OOPSLA ’12]

F* + Dijkstra

Shriram @2:30pm

Me @now

Nik @9:00am
DJS = Refinement Types
+ Several New
Quantifier-Free
Mechanisms

Dependent
JavaScript (DJS)
[POPL ’12, OOPSLA ’12]

Me
@now
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”

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

Syntactic Sugar for Common Types

\[
\text{Num} = \{ n \mid \text{tag}(n) = \text{"number"} \}
\]

\[
\text{NumOrBool} = \{ v \mid \text{tag}(v) = \text{"number"} \lor \text{tag}(v) = \text{"boolean"} \}
\]

\[
\text{Int} = \{ i \mid \text{tag}(i) = \text{"number"} \land \text{integer}(i) \}
\]

\[
\text{Any} = \{ x \mid \text{true} \}
\]
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

\[
\begin{align*}
\{ n \mid n = 3 \} \\
<: \{ n \mid n > 0 \} \\
<: \{ n \mid \text{tag}(n) = \text{“number”} \land \text{integer}(n) \} \\
<: \{ n \mid \text{tag}(n) = \text{“number”} \}
\end{align*}
\]
Refinement Types

Subtyping is Implication

\[
\begin{align*}
  n &= 3 \\
  &\implies n > 0 \\
  &\implies \text{tag}(n) = \text{“number”} \land \text{integer}(n) \\
  &\implies \text{tag}(n) = \text{“number”}
\end{align*}
\]
<table>
<thead>
<tr>
<th>Tag-Tests</th>
<th>Duck Typing</th>
<th>Mutable Objects</th>
<th>Prototypes</th>
<th>Arrays</th>
</tr>
</thead>
</table>

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

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

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

```
//: negate :: (x:Any) ➞ Any
```

Function type annotation inside comments
```javascript
var negate = function(x) {
  if (typeof x === "boolean")
    return !x;
  else
    return 0 - x;
}
```

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

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

DJS is Path Sensitive

X is arbitrary non-boolean value...
so DJS signals error!
//: 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
var negate = function(x) {
    if (typeof x === "boolean")
        return !x;
    else
        return 0 - x;
}

`: negate :: (x:NumOrBool) \rightarrow Any`
//: 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”?

```javascript
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

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

\[
\{ d \mid \text{tag}(d) = \text{"object"} \land \\
\text{has}(d, \text{"quack"}) \Rightarrow \\
\text{sel}(d, \text{"quack"}) : : \text{Unit} \rightarrow \text{Str} \}
\]
DJS is Flow Sensitive

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

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

McCarthy operator

```
x0: Empty
x1: {d | d = \text{upd}(x_0, "f", 7)}
```
DJS is **Flow Sensitive**

```javascript
var x = {};  // x₀: Empty
x.f = 7;    // x₁: {d | d = upd(x₀, "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
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 | \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 if grandpa contains } k, \text{ then Read } k \text{ from } \text{grandpa} \\
\text{Else if } \ldots \\
\text{Else} \\
\text{Return undefined} \}
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
  Return undefined

```
{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
  Return undefined
```
Semantics of Key Lookup

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

\[
\{ v \mid \text{if has(child, k) then} \vspace{3mm} \\
\quad v = \text{sel(child, k)} \\
\text{else if has(parent, k) then} \vspace{3mm} \\
\quad v = \text{sel(parent, k)} \\
\text{else} \vspace{3mm} \\
\quad v = \text{HeapSel}(H, grandpa, k) \}
\]

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

```javascript
{ 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))
}
```

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

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

```json
grandpa
???
```

```javascript
H
(Rest of Heap)
```

```json
null
```
```javascript
var k = "last"; child[k];
```
Prototype Chain Unrolling

Key Idea:
Reduce prototype semantics to \textit{decidable} theory of arrays
```javascript
var nums = [0,1,2];
while (...) {
    nums[nums.length] = 17;
}
```

A finite tuple...

... 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

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

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

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

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

```javascript
var tup = [17, "cacti"];```

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

{ a | a :: Arr(Any)
  ∧ packed(a) ∧ len(a) = 3
  ∧ ...
}
```
DJS handles other array 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?

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

Can Integrate DJS with “Contract Checking” at Run-time aka “Gradual Typing”
Dependent JavaScript (DJS)

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

"Usability"

Quantifier-Free Mechanisms

F* + Dijkstra

Expressiveness

[POPL '12, OOPSLA '12]
Function Subtyping...

\[
\{ d \mid \text{sel}(d, "f") :: (x: \text{Any}) \rightarrow \{ y \mid y = x \} \} <:\{ d \mid \text{sel}(d, "f") :: (x: \text{Num}) \rightarrow \text{Num} \}\]
Function Subtyping...

\[
\text{sel}(d, "f") :: (x: \text{Any}) \rightarrow \{ y \mid y = x \} \\
\Rightarrow \text{sel}(d, "f") :: (x: \text{Num}) \rightarrow \text{Num}
\]
Function Subtyping...

\[
\begin{align*}
f & : (x:\text{Any}) \to \{ y \mid y = x \} \\
\Rightarrow f & : (x:\text{Num}) \to \text{Num}
\end{align*}
\]

... With Quantifiers

\[
\begin{align*}
\forall x, y. \, \text{true} \land y = f(x) & \Rightarrow y = x \\
\Rightarrow \forall x, y. \, \text{Num}(x) \land y = f(x) & \Rightarrow \text{Num}(y)
\end{align*}
\]

Valid, but First-Order Logic is Undecidable
Function Subtyping...

\[ f :: (x:\text{Any}) \rightarrow \{ y \mid y = x \} \]

\[ \Rightarrow f :: (x:\text{Num}) \rightarrow \text{Num} \]

... Without Quantifiers!

Nested Refinements

Treat Function Types as Uninterpreted

Implication = SMT Validity + Syntactic Subtyping
Heap Updates...

```javascript
var x = {};

x.f = 7;
```

... With Quantifiers

 Encode Heap w/ McCarthy Operators

\[
\begin{align*}
& \land \text{sel}(h_1,x) = \text{empty} \\
& \land \forall y. \ x \neq y \Rightarrow \text{sel}(h_1,y) = \text{sel}(h_0,y) \\
& \land \text{sel}(h_2,x) = \text{upd}(\text{sel}(h_1,x), "f", 7) \\
& \land \forall y. \ x \neq y \Rightarrow \text{sel}(h_2,y) = \text{sel}(h_1,y)
\end{align*}
\]
Heap Updates...

\[
\text{var } x = \{\};
\]
\[
x.f = 7;
\]

... Without Quantifiers!

Flow-Sensitive Types (à la Alias Types)

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

input type

input heap

output type

output heap
Prototype Inheritance...
Array Semantics...

... Without Quantifiers!
Dependent JavaScript (DJS) = Refinement Types
+ Nested Refinements
+ Flow Sensitive Types
+ Prototype Unrolling
+ Array Encoding

Quantifier-Free Mechanisms

Dependent JavaScript (DJS) [POPL ’12, OOPSLA ’12]

F* + Dijkstra

“Usability”

Expressiveness
Implementation

DJS Program

Desugarer
Based on Guha et al. [ECOOP ‘10]

Desugared Program

JavaScript → λ-Calculus + References + Prototypes
Desugared Program

Desugarer
Based on Guha et al. [ECOOP ‘10]

Desugared Program

DJS Program

Implementation

Programmer Chooses Warnings or Errors

Local Type Inference

Subtyping w/o Z3 If Possible

Type Checker

Z3 SMT Solver
Benchmarks

13 Excerpts from:
*JavaScript, Good Parts*
SunSpider Benchmark Suite
Google Closure Library

**LOC**
before/after

<table>
<thead>
<tr>
<th>Location before</th>
<th>Location after</th>
</tr>
</thead>
<tbody>
<tr>
<td>306</td>
<td>408 (+33%)</td>
</tr>
</tbody>
</table>

Chosen to **Stretch** the Current Limits of DJS
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>LOC before/after</th>
</tr>
</thead>
<tbody>
<tr>
<td>13 Excerpts from:</td>
<td>306</td>
</tr>
<tr>
<td><em>JavaScript, Good Parts</em></td>
<td>408 (+33%)</td>
</tr>
<tr>
<td>SunSpider Benchmark Suite</td>
<td></td>
</tr>
<tr>
<td>Google Closure Library</td>
<td></td>
</tr>
<tr>
<td>9 Browser Extensions from:</td>
<td>321</td>
</tr>
<tr>
<td>[Guha et al. Oakland ’11]</td>
<td>383 (+19%)</td>
</tr>
<tr>
<td>2 Examples from:</td>
<td>1,003</td>
</tr>
<tr>
<td>Google Gadgets</td>
<td>1,027 (+2%)</td>
</tr>
<tr>
<td>TOTALS</td>
<td>1,630</td>
</tr>
<tr>
<td></td>
<td>1,818 (+12%)</td>
</tr>
</tbody>
</table>
Already Improved by Simple **Type Inference** and **Syntactic Sugar**

Plenty of **Room for Improvement**
- Iterative Predicate Abstraction
- Bootstrap from **Run-Time Traces**
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>LOC before/after</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>13 Excerpts from: <em>JavaScript, Good Parts</em>&lt;br&gt; SunSpider Benchmark Suite&lt;br&gt; Google Closure Library</td>
<td>306&lt;br&gt; 408 (+33%)</td>
<td>10 sec</td>
</tr>
<tr>
<td>9 Browser Extensions from: [Guha et al. Oakland ’11]</td>
<td>321&lt;br&gt; 383 (+19%)</td>
<td>3 sec</td>
</tr>
<tr>
<td>2 Examples from: Google Gadgets</td>
<td>1,003&lt;br&gt; 1,027 (+2%)</td>
<td>19 sec</td>
</tr>
<tr>
<td>TOTALS</td>
<td>1,630&lt;br&gt; 1,818 (+12%)</td>
<td>32 sec</td>
</tr>
</tbody>
</table>
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
Expressiveness

“Usability”

TypeScript

TypedJS

Dependent JavaScript (DJS)
[POPL ’12, OOPSLA ’12]

F* + Dijkstra

Lightweight (unsound) static checking tools becoming popular

Opportunity to improve IDE tools

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