Nested Refinement Types for JavaScript

Ravi Chugh

Types for JavaScript

“Dynamic” or Untyped Features Enable Rapid Prototyping
“Dynamic” or Untyped Features Enable Rapid Prototyping

JavaScript

<table>
<thead>
<tr>
<th>Count</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>411,345</td>
<td>C#</td>
</tr>
<tr>
<td>361,080</td>
<td>Java</td>
</tr>
<tr>
<td>337,896</td>
<td>PHP</td>
</tr>
<tr>
<td>321,757</td>
<td>JavaScript</td>
</tr>
<tr>
<td>175,765</td>
<td>C++</td>
</tr>
<tr>
<td>160,768</td>
<td>Python</td>
</tr>
<tr>
<td>83,486</td>
<td>C</td>
</tr>
<tr>
<td>64,345</td>
<td>Ruby</td>
</tr>
<tr>
<td>9,827</td>
<td>Haskell</td>
</tr>
<tr>
<td>1,347</td>
<td>OCaml</td>
</tr>
</tbody>
</table>

http://stackoverflow.com/tags
02-03-13
Types for JavaScript

“Dynamic” or Untyped Features Enable Rapid Prototyping

1. Development Tools
2. Reliability & Security
3. Performance
1. Development Tools

optional “type systems”
unsound but provide
some IDE support

```javascript
2 function foo(obj: {n: number}) {
3     return 42 + obj.n;
4 }
5 var i: number = foo({n: "bad arg"});
6 var j: number = foo({n: undefined});
```

/*
* Returns a random number greater than or equal to @code a and less than
* @code b.
* @param {number} a The lower bound for the random number (inclusive).
* @param {number} b The upper bound for the random number (exclusive).
* @return {number} A random number N such that a <= N < b.
*/
```
```javascript
42 goog.math.uniformRandom = function(a, b) {
43     return a + Math.random() * (b - a);
44 }
```
2. Reliability & Security

- Lightweight static checkers
- Run-time invariant checkers
- Secure language subsets

- JSLint
- contracts.coffee
- Facebook Developers
- Safe
- Google Caja: Compiler for making third-party HTML, CSS and JavaScript safe for embedding
3. Performance

fast language subset to facilitate optimizations

Just-In-Time (JIT) engines perform static/dynamic analysis to predict types
Types for JavaScript

Why So Hard?
JavaScript

implicit global object

var lifting

scope manipulation

' , , , ' == new Array(4)
“The Good Parts”

- lambdas
- reflection
- objects
- arrays
- eval()*

scope manipulation

implicit global object

JavaScript

var lifting

‘,,,’ == new Array(4)
JavaScript

“The Good Parts”

- lambdas
- reflection
- objects
- arrays
- `eval()`*
Thesis Statement:
“Refinement types can reason about dynamic languages to verify the absence of run-time errors ...
Usability

Expressiveness

Types + Logic

Idioms of Dynamic Languages

Thesis Statement:

... without resorting to undecidable logics”

Verification

Hoare Logics
Model Checking
Abstract Interpretation
Theorem Proving
...

13
Usability

Types + Logic

Idioms of Dynamic Languages

Dependent JavaScript (DJS)
[POPL 2012, OOPSLA 2012]

Verification
Hoare Logics
Model Checking
Abstract Interpretation
Theorem Proving
...

Expressiveness
Outline

Motivation and Thesis

Challenges of Dynamic Languages

Tour of Dependent JavaScript

Technical Contributions

Evaluation

Looking Ahead
JavaScript

"The Good Parts"

- lambdas
- reflection
- objects
- arrays
- `eval()`*
Core Technical Challenges

- reflection
- objects
Core Technical Challenges

JavaScript vs. Java
function negate(x) {
    if (typeof x == "number") {
        return 0 - x;
    } else {
        return !x;
    }
}
function negate(x) {
    if (typeof x == "number") {
        return 0 - x;  // X should be numeric on then-branch...
    } else {
        return !x;  // but boolean on else-branch
    }
}
function negate(x) {
    if (typeof x == "number") {
        return 0 - x;
    } else {
        return !x;
    }
}
interface NumOrBool {
    NumOrBool negate();
}

class Bool implements NumOrBool {
    boolean data;
    Bool(boolean b) { this.data = b; }
    Bool negate() {
        return new Bool(!this.data);
    }
}

class Num implements NumOrBool {
    ...}

declared union type
function negate(x) {
    if (typeof x == "number") {
        ...
    } else {
        ...
    }
}

interface NumOrBool {
    NumOrBool negate();
}

class Bool implements NumOrBool {
    boolean data;
    Bool(boolean b) { this.data = b; }
    Bool negate() {
        return new Bool(!this.data);
    }
}
class Num implements NumOrBool { ... }

ad-hoc union type

declared union type
function negate(x) {
  if (typeof x === "number") {
    ...
  } else {
    ...
  }
}
```javascript
function negate(x) {
    if (typeof x === "number") {
        ...
    } else {
        ...
    }
}
```

Lightweight Reflection

ad-hoc union type
Java

```java
class Person {
  String first;
  String last;

  Person (String s1, String s2) {
    this.first = s1;
    this.last = s2;
  }

  void greeting(...) { ... }
}

var john = new Person("John", "McCarthy");
var s = john.first + "…";
```

JavaScript

```javascript
var john = {
  "first": "John",
  "last": "McCarthy",
  "greeting": ...
};

var s = john.first + "…";
```

**keys are arbitrary **computed **strings**

**field names are declared**
Objects are **Dictionaries**

\[
\text{obj.f means obj["f"]}
\]
Java

class Person {
    String first;
    String last;
    Person (String s1, String s2) {
        this.first = s1;
        this.last = s2;
    }
    void greeting(…) { … }
}
var john = new Person("John", "McCarthy");
var s = john.first + "…";

JavaScript

var john = {};
john.first = "John";
john.last = "McCarthy";
john.greeting = …;

incrementally extend object

“sealed” after initialization
JavaScript

```
var john = {};
john.first = "John";
john.last = "McCarthy";
john.greeting = …;

// much later in the code
john.anotherKey = …;

addMoreKeys(john);

if (...) {
    addEvenMoreKeys(john);
}
```

Java

```java
class Person {
    String first;
    String last;
    Person (String s1, String s2) {
        this.first = s1;
        this.last = s2;
    }
    void greeting(...) { … }
}
var john = new Person("John", "McCarthy");
var s = john.first + "…";
```
class Person {
    String first;
    String last;

    Person (String s1, String s2) {
        this.first = s1;
        this.last = s2;
    }

    void greeting(...) {
    }
}

var john = new Person("John", "McCarthy");
var s = john.first + "…";

class TuringWinner extends Person {
    int year;
    TuringWinner (...) {
    }
}

var twJohn = new TuringWinner(...);
twJohn.greeting();
Java

```java
class Person {
    String first;
    String name;
    Person (String s1, String s2) {
        this.first = s1;
        this.last = s2;
    }
    void greeting(…) { … }
}
var john = new Person("John", "McCarthy");
...
var twJohn = new TuringWinner(…);
twJohn.greeting();
```

JavaScript

```javascript
var john = {
    "first": "John",
    "last": "McCarthy",
    "greeting": …
};
var twJohn = {
    "year": 1971,
    "__proto__": john
},
twJohn.greeting();
```
Objects Inherit from Prototypes

```javascript
var john = {
  "first" : "John",
  "last" : "McCarthy",
  "greeting" : ...
};

var twJohn = {
  "year" : 1971,
  "__proto__" : john
};

twJohn.greeting();
```
Core Technical Challenges

- Lightweight Reflection
- Objects are Dictionaries
- Objects are Extensible
- Objects Inherit from Prototypes
Dependent JavaScript (DJS)
[POPL ’12, OOPSLA ’12]

- lightweight reflection
- objects are dictionaries
- objects are extensible
- prototype inheritance

<table>
<thead>
<tr>
<th>Java</th>
<th>C#</th>
<th>Haskell</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

- ✓: True
- ×: False
- ✓*: Partially True
Outline

Motivation and Thesis
Challenges of Dynamic Languages
Tour of Dependent JavaScript
Technical Contributions
Evaluation
Looking Ahead
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 \}\n
“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

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

3 :: \{ n \mid n = 3 \}\n3 :: \{ n \mid n > 0 \}\n3 :: \{ n \mid \text{tag}(n) = \text{“number”} \land \text{integer}(n) \}\n3 :: \{ n \mid \text{tag}(n) = \text{“number”} \}\
Refinement Types

Subtyping is Implication

\[
\begin{align*}
\{ n \mid n = 3 \} & \subseteq \{ n \mid n > 0 \} \\
\subseteq & \{ n \mid \text{tag}(n) = \text{"number"} \land \text{integer}(n) \} \\
\subseteq & \{ n \mid \text{tag}(n) = \text{"number"} \}
\end{align*}
\]
Refinement Types

Subtyping is Implication

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

Subtyping is Implication

Refinement Type Checker

$p \Rightarrow q$?

Yes / No

SMT Solver or Theorem Prover
<table>
<thead>
<tr>
<th>Tag-Tests</th>
<th>Dictionary Objects</th>
<th>Mutable Objects</th>
<th>Prototypes</th>
</tr>
</thead>
</table>

function negate(x) {
    if (typeof x === "number") {
        return 0 - x;
    } else {
        return !x;
    }
}
//: negate :: (x:NumOrBool) → NumOrBool

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

\{ x | \text{tag}(x) = "number" \} \subseteq \{ x | \text{tag}(x) = "number" \}

tag(x) = "number" \implies \text{tag}(x) = "number"
//: negate :: (x:NumOrBool) → NumOrBool

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

“expected args”

{ x | not(tag(x) = “number”) 
       ∧ (tag(x) = “number” ∨ tag(x) = “boolean”) } ⊆

{ x | tag(x) = “boolean” }
//: negate :: (x:NumOrBool) → NumOrBool

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

Output type depends on input value. The output type depends on the input value. For example, if the input is a number, the output is the negation of that number; if the input is a boolean, the output is the negation of that boolean value.
What is “Duck Typing”? 

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

if ("quack" in duck) 
  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**

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

\[
\{ d \mid \text{tag}(d) = \text{“object”} \land \\
\text{has}(d, \text{“quack”}) \Rightarrow \\
\text{sel}(d, \text{“quack”}) :: () \to \text{Str}\}
\]
if ("quack" in duck)
  return "Duck says ” + duck.quack();
else
  return "This duck can’t quack!”;
DJS is **Flow Sensitive**

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

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

```
x0: Empty
x1: {d | d = upd(x0, "f", 7)}
```

McCarthy operator
DJS is **Flow Sensitive**

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

**Strong** updates to singleton objects

**Weak** updates to collections of objects
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

null
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 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}(child, k) \text{ then} \\
\quad v = \text{sel}(child, k) \\
\text{elseif has}(parent, k) \text{ then} \\
\quad v = \text{sel}(parent, k) \\
\text{elseif grandpa contains k, then} \\
\quad \text{Read k from grandpa} \\
\text{elseif ...} \\
\text{else} \\
\quad \text{Return undefined} \}
child[k];

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

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

{ "first" : "John" }

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

???

null

H
(Rest of Heap)

{ v | v = "John" }

\:;

\{ 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" \}
```javascript
var k = "last"; child[k];
```

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

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

```
grandpa = ???
```

```
H = {
  Rest of Heap
}
```

```
null
```

```
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

“The Good Parts”

- lambdas ✓
- reflection ✓
- objects ✓
- arrays
- eval() *

EXTRA SLIDES
Outline

Motivation and Thesis

Challenges of Dynamic Languages

Tour of Dependent JavaScript

Technical Contributions

Evaluation

Looking Ahead
Our Strategy:
Build an expressive type system for core language
Dependent JavaScript (DJS)

- lambdas
- dictionaries
- tag tests
- mutation
- prototypes

System D
System !D
System DJS

(POPL 2012)
(OOPSLA 2012)
(OOPSLA 2012)
Prior Refinement Types

Types

\[ T ::= \{ \ x : B \mid p \} \]

- base type (e.g. Int, Bool, Str)

- dependent function type

- object type (a.k.a. record, struct)

- dictionary type (a.k.a. map, hash table)

- reference type (a.k.a. pointer)

Predicates refine only base types
<table>
<thead>
<tr>
<th>Prior Refinement Types</th>
<th>System DJS</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Java</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Haskell</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Scala</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feature</th>
<th>C#</th>
<th>Java</th>
<th>Haskell</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>lightweight reflection</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>objects are dictionaries</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>objects are extensible</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>prototype inheritance</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Nested Refinements</td>
<td>Flow-Sensitive Refinements</td>
<td>JavaScript Encodings</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------</td>
<td>----------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>objects are dictionaries</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>objects are extensible</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>prototype inheritance</td>
<td>✗</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Key Challenge:

Function Subtyping as Logical Implication
42 + negate(17)

\[ \{ f \mid f :: (x: \text{NumOrBool}) \rightarrow \{ y \mid \text{tag}(y) = \text{tag}(x) \} \} \subseteq \{ f \mid f :: (x: \text{Num}) \rightarrow \text{Num} \} \]

“expected function”
How to Prove

\[ f :: (x \text{: } \text{NumOrBool}) \rightarrow \{ y | \text{tag}(y) = \text{tag}(x) \} \]

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

How to Encode Type System \textbf{Inside} Logic?

Prior Refinement Systems \textbf{Disallow} Function Types \text{Inside} Logic!
\[ T ::= \{ x \mid p \} \]
\[ \mid T_1 \rightarrow T_2 \]

Refinement Logic

[Prior State-of-the-Art]

Satisfiability Modulo Theories (SMT)

SAT + Decidable Theories

\[ p \Rightarrow q \]

SMT Solver (e.g. Z3)
### Types

\[
T ::= \{ x \mid p \} \quad \mid \quad T_1 \rightarrow T_2
\]

### Refinement Logic

\[
p ::= p \land q \mid p \lor q \mid \neg p \quad \mid \quad x = y \mid x > y \mid \ldots
\]

\[
\mid \quad \text{tag}(x) = \text{“number”} \mid \ldots
\]

\[
\mid \quad \text{sel}(x, k) = 17 \mid \ldots
\]

- Boolean Connectives
- Equality
- Linear Arithmetic
- Uninterpreted Functions
- McCarthy Arrays

---

<table>
<thead>
<tr>
<th>Nested Refinements</th>
<th>Flow-Sensitive Refinements</th>
<th>JavaScript Encodings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>Refinement Logic</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JavaScript Encodings</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flow-Sensitive Refinements</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Nested Refinements</td>
<td></td>
</tr>
</tbody>
</table>
Types

\[ T ::= \{ x \mid p \} \]
\[ \mid T_1 \rightarrow T_2 \]

Refinement Logic

\[ p ::= p \land q \mid p \lor q \mid \neg p \]
\[ \mid x = y \mid x > y \mid \ldots \]
\[ \mid \text{tag}(x) = \text{"number"} \mid \ldots \]
\[ \mid \text{sel}(x,k) = 17 \mid \ldots \]

Enables Union Types and **Lightweight Reflection**

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

\[ T ::= \{ x \mid p \} \]
\[ \mid T_1 \rightarrow T_2 \]

Refinement Logic

\[ p ::= p \land q \mid p \lor q \mid \neg p \]
\[ \mid x = y \mid x > y \mid \ldots \]
\[ \mid \text{tag}(x) = \text{"number"} \mid \ldots \]
\[ \mid \text{sel}(x, k) = 17 \mid \ldots \]

Enables **Dictionary Types** with Dynamic Keys ...  

But **Disallows** Functions in Dictionaries!

\[ \{ d \mid \text{tag}(\text{sel}(d, k)) = \text{"boolean"} \land \text{tag}(\text{sel}(d, \text{"f"})) = \text{"function"} \land \text{sel}(d, \text{"f"}) ??? \} \]
Types

\[
T ::= \{ x \mid p \} \mid T_1 \rightarrow T_2
\]

Refinement Logic

\[
p ::= p \land q \mid p \lor q \mid \neg p \\
\mid x = y \mid x > y \mid \ldots \\
\mid \text{tag}(x) = \text{"number"} \mid \ldots \\
\mid \text{sel}(x,k) = 17 \mid \ldots \\
\mid \text{sel}(x,k) :: T_1 \rightarrow T_2 \mid \ldots
\]

Goal: Refer to Type System Inside Logic Without Giving up Decidability

Solution: Nested Refinements!
Nested Refinements [POPL ’12]

1. Decidable Encoding
2. Precise Subtyping
3. Type Soundness Proof
Decidable Encoding

\[ f \colon (x: \text{NumOrBool}) \rightarrow \{ y \mid \text{tag}(y) = \text{tag}(x) \} \]

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

Uninterpreted Predicate in Logic

Distinct Uninterpreted Constants in Logic…
Decidable Encoding

\[ f : (x:\text{NumOrBool}) \rightarrow \{ y | \text{tag}(y) = \text{tag}(x) \} \]

\[ \Rightarrow \]

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

**Uninterpreted Predicate in Logic**

**Distinct Uninterpreted Constants in Logic... w/ Types Nested Inside**
# Decidable Encoding

$$f :: (x: \text{NumOrBool}) \rightarrow \{ y | \text{tag}(y) = \text{tag}(x) \}$$

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

Trivially Decidable, but **Imprecise**
Precise Subtyping

\[ \{ f \mid p \} \subseteq \{ f \mid f :: (x:T_1) \rightarrow T_2 \} \]

Step 1: Find \((x:S_1) \rightarrow S_2\) such that

\[ p \Rightarrow f :: (x:S_1) \rightarrow S_2 \]

Step 2: Compare \((x:S_1) \rightarrow S_2\) and \((x:T_1) \rightarrow T_2\)

“contra-variance”

\[ T_1 \subseteq S_1 \]

“co-variance”

\[ x : T_1 \land S_2 \subseteq T_2 \]
Precise Subtyping

\[
\{ f \mid f = \text{negate} \} \subseteq \{ f \mid f :: (x:\text{Num}) \rightarrow \text{Num} \}
\]

Step 1:
\[
f = \text{negate} \Rightarrow f :: (x:\text{NumOrBool}) \rightarrow \{ y \mid \text{tag}(y) = \text{tag}(x) \}
\]

Step 2:
Precise Subtyping

\[ \{ f \mid f = \text{negate} \} \subseteq \{ f \mid f :: (x:\text{Num}) \rightarrow \text{Num} \} \]

**Step 1:**

\[ f = \text{negate} \Rightarrow f :: (x:\text{NumOrBool}) \rightarrow \{ y \mid \text{tag}(y) = \text{tag}(x) \} \]

**Step 2:**

\[ \text{Num} \subseteq \text{NumOrBool} \]

\[ x:\text{Num} \land \{ y \mid \text{tag}(y) = \text{tag}(x) \} \subseteq \text{Num} \]
Precise Subtyping

\[ \{ f \mid f = \text{negate} \} \subseteq \{ f \mid f :: (x:\text{Num}) \rightarrow \text{Num} \} \]

Step 1: Decidable Reasoning via SMT

Step 2: Precise Function Subtyping via Syntactic Techniques
Type Soundness Proof

Conventional Proofs Require Logic to be an Oracle ...

... but Nesting Introduces Circularity That Prevents Typical Inductive Proofs
Type Soundness Proof

Proof Technique: 
**Stratify** into Infinite Hierarchy

- Type System $\infty$
  - Type System $0$
    - Logic $0$
  - Type System $1$
    - Logic $1$
  - Type System $2$
    - Logic $2$
  - ...
Nested Refinements [POPL ’12]

Key to Encode Dictionary Objects

(Even in Statically Typed Languages!)
Key Issue:

“Invariant” Reference Types

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

```
let x = new {} in
  (*x).f = 7;
  (*x).f + 2;
```

Allocates ptr $x :: \text{Ref(Dict)}$

Heap update doesn’t affect type

So key lookup doesn’t type check!
Allocates **heap location** $L_x$ and $x :: \text{Ref}(L_x)$

Type of pointer doesn’t change ...

But types of heap locations can!

```javascript
let x = new {} in
  (*x).f = 7;
  (*x).f + 2;
```

$L_x \mapsto d_0: \{d \mid d = \text{empty}\}$

$L_x \mapsto d_1: \{d \mid d = \text{upd}(d_0, "f", 7)\}$
$x : T_1 \rightarrow T_2$

- Input type: $x : T_1$
- Output type: $T_2$
\[ x : T_1 \xrightarrow{h_1} T_2 \]

- **input type**: \( T_1 \)
- **input heap**: \( h_1 \)
- **output type**: \( T_2 \)
- **output heap**: \( h_2 \)
\[ x : T_1 / h_1 \rightarrow T_2 / h_2 \]

Our Formulation Extends:

**Alias Types** (Smith et al., ESOP 2000)

*syntactic types* for *higher-order* programs

**Low-Level Liquid Types** (Rondon et al., POPL 2010)

*refinement types* for *first-order* programs
let swap (x, y) =
  let tmp = *x in
  *x = *y;
  *y = tmp in ...

///: swap :: (x:Ref(Lx), y:Ref(Ly))
///:   / H + (Lx |-> a:Any) + (Ly |-> b:Any)
///:   → Void
///:   / H + (Lx |-> a’:Any{a’=b})
///:   + (Ly |-> b’:Any{b’=a})
Prototype Inheritance

```javascript
//: getKey :: (x:Ref(L_x), k:Str)
//:         / H + (L_x l-> d:Dict > L_x')
//:         → {y | if has(d,k)
//:             then y = sel(d, k)
//:             else y = HeapHas(H, L_x', k)}
//:         / H + (L_x l-> d':Dict{d' = d} > L_x')
```
Prototype Inheritance

Primitive Operators
(Avoid Implicit Coercion)

JavaScript Arrays
(Several Unusual Issues)
Outline

Motivation and Thesis
Challenges of Dynamic Languages
Tour of Dependent JavaScript
Technical Contributions

Evaluation

Looking Ahead
### Benchmarks

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

<table>
<thead>
<tr>
<th>LOC</th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>313</td>
<td>421</td>
</tr>
<tr>
<td></td>
<td>(+35%)</td>
<td></td>
</tr>
</tbody>
</table>

Chosen to **Stretch** the Limits of DJS
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>LOC before/after</th>
</tr>
</thead>
<tbody>
<tr>
<td>14 Excerpts from:</td>
<td></td>
</tr>
<tr>
<td><em>JavaScript, Good Parts</em></td>
<td>313 (313)</td>
</tr>
<tr>
<td>SunSpider Benchmark Suite</td>
<td>421 (+35%)</td>
</tr>
<tr>
<td>Google Closure Library</td>
<td></td>
</tr>
<tr>
<td>9 Browser Extensions from:</td>
<td></td>
</tr>
<tr>
<td>2 Examples from:</td>
<td>383 (+19%)</td>
</tr>
<tr>
<td>Google Gadgets</td>
<td></td>
</tr>
<tr>
<td>2 Examples from:</td>
<td>1,003 (1,003)</td>
</tr>
<tr>
<td>Google Gadgets</td>
<td>1,027 (+2%)</td>
</tr>
<tr>
<td>TOTALS</td>
<td>1,637 (1,637)</td>
</tr>
<tr>
<td></td>
<td>1,831 (+12%)</td>
</tr>
<tr>
<td>Benchmarks</td>
<td>LOC before/after</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>14 Excerpts from:</td>
<td></td>
</tr>
<tr>
<td><em>JavaScript, Good Parts</em></td>
<td>313</td>
</tr>
<tr>
<td>SunSpider Benchmark Suite</td>
<td>421 (+35%)</td>
</tr>
<tr>
<td>Google Closure Library</td>
<td></td>
</tr>
<tr>
<td>9 Browser Extensions from:</td>
<td></td>
</tr>
<tr>
<td>2 Examples from:</td>
<td></td>
</tr>
<tr>
<td>Google Gadgets</td>
<td>1,003</td>
</tr>
<tr>
<td>TOTALS</td>
<td>1,637</td>
</tr>
<tr>
<td></td>
<td>1,831 (+12%)</td>
</tr>
</tbody>
</table>
Reducing Annotation Burden

• Bidirectional (local) type inference
  – Several aspects unique to this setting

• Optional `var` declaration invariants
  – Auto add to types of subsequent closures

```plaintext
var i = 0;
...
//: f :: T₁ / H₁
//: → T₂ / H₂
var f(...) { .. i .. }
```

function refers to mutable variable `i`, so heaps must describe the location

104
Reducing Annotation Burden

- Bidirectional (local) type inference
  - Several aspects unique to this setting

- Optional `var` declaration invariants
  - Auto add to types of subsequent closures

```java
var i /*: Int */ = 0;
...
//: f :: T₁ / H₁
//:  → T₂ / H₂
var f(...) { ... i ... }
```

**copy** the annotation, if any
Reducing Annotation Burden

• Bidirectional (local) type inference
  – Several aspects unique to this setting

• Optional var declaration invariants
  – Auto add to types of subsequent closures

```javascript
var i /*: Int */ = 0;
...
//: f :: T₁ / H₁ + (Lᵢ ↦ i₁:Int)
//: → T₂ / H₂ + (Lᵢ ↦ i₂:Int)
var f(...) { ... i ... }
```

**copy** the annotation, if any
Reducing Annotation Burden

• Bidirectional (local) type inference
  – Several aspects unique to this setting

• Optional var declaration invariants
  – Auto add to types of subsequent closures

```java
var i = 0;
...

//: f :: T₁ / H₁ + (L_i |-> i₁:{x | x = 0})
//:    → T₂ / H₂ + (L_i |-> i₂:{x | x = 0})
var f(...) { ... i ... }
```

no annotation means “do not modify”
Improving Performance

• Avoid making SMT queries when possible

\[
17 :: \{ x : \text{Int} \mid x = 17 \} \\
\text{track syntactic types for specialized cases ...}
\]

\[
\text{if } b \text{ then "hi" else } 17 :: \{ x \mid b = \text{true} \Rightarrow x = \text{"hi"} \\
\quad \land b = \text{false} \Rightarrow x = 17 \} \\
\text{... fall back on refinement types for general case}
\]
Improving Performance

• Can reduce precision for if-expressions

```javascript
var n = 0;
if (b) {
    n = n + 1;
} else {
    n = n + 2;
}
```

“exact joins” by default ...

... annotation on `var` triggers “inexact joins”

\[ L_n \mid\rightarrow m:\{ x \mid b = \text{true} \Rightarrow x = 1 \wedge b = \text{false} \Rightarrow x = 2 \} \]
Improving Performance

• Can reduce precision for if-expressions

```javascript
var n /*: Int */ = 0;
if (b) {
    n = n + 1;
} else {
    n = n + 2;
}
```

“exact joins” by default ...

... annotation on `var` triggers “inexact joins”
Relatively Modest Optimizations Have Made **Significant** Impact

Plenty of **Room for Improvement**

| TOTALS | 1,637 | 1,831 (+12%) | 33 sec |
Outline

Motivation and Thesis

Challenges of Dynamic Languages

Tour of Dependent JavaScript

Technical Contributions

Evaluation

Looking Ahead
Immediate Directions

• Type Soundness
  – proven for System D, but not for !D or DJS

• Expressiveness
  – e.g. mutual recursion, recursion through heap

• Usability
  – annotation burden, performance, error messages

• Integration with Run-Time Contracts
  – invariants before and after eval()
“Gradual Gradual Typing”

Usability

- **TypeScript**
  Lightweight (unsound) static checking tools becoming popular

- **Dependent JavaScript (DJS)**
  Translate for additional checking and IDE support

Verification

Expressiveness
Web Development Tools

Way Behind Tools for Java, etc.

• Static Types in DJS Enable Refactoring, Documentation Tools, Code Completion

• Web-Based Interfaces to **Collect Statistics** about Programming Patterns, Errors
Iterative Inference

Global inference (e.g. via Liquid Types) would be nice, but no basic type structure to start with
Browser Extension Security

Security = Refinement Type Checking

à la Swamy et al. (ESOP 2010)
and Guha et al. (Oakland 2011)
Thanks!

Nested Refinements + Dependent JavaScript

Ranjit Jhala
Pat Rondon
Dave Herman
Panos Vekris

Other Projects

Sorin Lerner
Jan Young
Jeff Meister
Nikhil Swamy
Juan Chen
EXTRA SLIDES
JavaScript

“The Good Parts”

- lambdas
- reflection
- objects
- arrays
- `eval()`*
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

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

\[
\begin{array}{cccccccc}
-1 & 0 & 1 & 2 & \cdots & \text{len}(a) \\
T? & T? & T? & T? & \cdots & T? & T? & \cdots \\
\hline
X & T & T & T & T & \cdots & T & X & \cdots \\
\end{array}
\]

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

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

```javascript
var tup = [17, “cacti”];
```

{ 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
  ∧ ... }
```
JavaScript

“The Good Parts”

- lambdas
- reflection
- objects
- arrays
- eval( )*
What About `eval`?

Old Types

... `eval`(`"..."`)...

Arbitrary code loading
What About `eval`?

```
... eval(“…”);

//: #assume
```

- **Old Types**
- **New Types**

**Future Work:** Integrate DJS with “Contract Checking” at Run-time aka “Gradual Typing”
Usability

Expressiveness

Idioms of Dynamic Languages

Verification
Hoare Logics
Model Checking
Abstract Interpretation
Theorem Proving
...

129
Higher-Order Refinements

Function Subtyping...

\[
\{ d \mid \text{sel}(d, "f") :: (x: Any) \to \{ y \mid y = x \} \} <:\{ d \mid \text{sel}(d, "f") :: (x: Num) \to \text{Num} \}
\]
Higher-Order Refinements

Function Subtyping...

\[
\text{sel}(d, "f") :: (x: \text{Any}) \to \{ y \mid y = x \} \\
\Rightarrow \text{sel}(d, "f") :: (x: \text{Num}) \to \text{Num}
\]
Higher-Order Refinements

Function Subtyping...

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

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

... With Quantifiers

\[ \forall x, y. \text{true} \land y = f(x) \Rightarrow y = x \]

\[ \Rightarrow \forall x, y. \text{Num}(x) \land y = f(x) \Rightarrow \text{Num}(y) \]

Valid, but First-Order Logic is Undecidable
Higher-Order Refinements

Heap Updates...

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

... With Quantifiers

```
∧ sel(h₁,x) = empty
∧ ∀y. x ≠ y ⇒ sel(h₁,y) = sel(h₀,y)
∧ sel(h₂,x) = upd(sel(h₁,x),"f",7)
∧ ∀y. x ≠ y ⇒ sel(h₂,y) = sel(h₁,y)
```

Encode Heap w/ McCarthy Operators
Subtyping with Nesting

1) Convert $q$ to CNF clauses $(q_{11} \lor \ldots) \land \ldots \land (q_{n1} \lor \ldots)$

2) For each clause, discharge some literal $q_{ij}$ as follows:

base predicate: $p \Rightarrow q_{ij}$

anything except HasType($x$, U)

E.g. tag($x$) = tag($y$)

$\text{tag}(\text{sel}(d,k)) =$ “number”
Subtyping with Nesting

1) Convert $q$ to CNF clauses $(q_{11} \lor \ldots) \land \ldots \land (q_{n1} \lor \ldots)$

2) For each clause, discharge some literal $q_{ij}$ as follows:

base predicate: $p \Rightarrow q_{ij}$

$$p \Rightarrow x :: U$$

$$p \Rightarrow x :: U'$$
Type Soundness with Nesting
**Substitution Lemma**

| If | \( x : T_x, \Gamma \vdash e :: T \) |
| and | \( \vdash v :: T_x \) |
| then | \( \Gamma[v/x] \vdash e[v/x] :: T[v/x] \) |

independent of \( \emptyset \), and just echoes the binding from the environment

\[
\begin{align*}
\text{f} & \{ v | v :: \text{Int} \rightarrow \text{Int} \} \vdash 0 :: \{ v | f :: \text{Int} \rightarrow \text{Int} \} \\
& \vdash \lambda x. x + 1 :: \{ v | v :: \text{Int} \rightarrow \text{Int} \} \\
& \vdash 0 :: \{ v | \lambda x. x + 1 :: \text{Int} \rightarrow \text{Int} \}
\end{align*}
\]
Substitution Lemma

If \( x: T_x, \Gamma \vdash e :: T \)
and \( \vdash v :: T_x \)
then \( \Gamma[v/x] \vdash e[v/x] :: T[v/x] \)

1\textsuperscript{st} attempt

\[ v = 0 \quad \times \quad \lambda x.x+1 :: \text{Int} \rightarrow \text{Int} \]

\[ 0 :: \{ v \mid v = 0 \} \quad \{ v \mid v = 0 \} < \{ v \mid \lambda x.x+1 :: \text{Int} \rightarrow \text{Int} \} \]

\[ \vdash 0 :: \{ v \mid \lambda x.x+1 :: \text{Int} \rightarrow \text{Int} \} \]
Substitution Lemma

If \( x : T_x, \Gamma \vdash e :: T \) and \( \vdash \nu :: T_x \) then \( \Gamma[\nu/x] \vdash e[\nu/x] :: T[\nu/x] \)

\[ \nu = 0 \implies \nu :: U' \]

2\textsuperscript{nd} attempt

Arrow \( U' <: \text{Int} \rightarrow \text{Int} \)

\[ 0 :: \{ \nu \mid \nu = 0 \} \]

\[ \{ \nu \mid \nu = 0 \} < \{ \nu \mid \lambda x. x+1 :: \text{Int} \rightarrow \text{Int} \} \]

\[ \vdash 0 :: \{ \nu \mid \lambda x. x+1 :: \text{Int} \rightarrow \text{Int} \} \]
• Rule not closed under substitution

• Interpret formulas by “hooking back” into type system

• Stratification to create ordering for induction

[S-Valid-Uninterpreted]  
\[ \Gamma \land \ p \Rightarrow q \]

\[ \Gamma \vdash \{ \nu \mid \nu = p \} < \{ \nu \mid \nu = q \} \]

[S-Valid-Interpreted]  
\[ I_n \models \Gamma \land \ p \Rightarrow q \]

\[ \Gamma \vdash_n \{ \nu \mid \nu = p \} < \{ \nu \mid \nu = q \} \]

\[ I_n \models \lambda x. x+1 :: \text{Int} \rightarrow \text{Int} \]

iff

\[ \vdash_{n-1} \lambda x. x+1 :: \{ \nu \mid \nu :: \text{Int} \rightarrow \text{Int} \} \]
Type Soundness with Nesting

Stratified Substitution Lemma

If \( x : T_x, \Gamma \vdash_n e :: T \) and \( \vdash_n v :: T_x \), then \( \Gamma[v/x] \vdash_{n+1} e[v/x] :: T[v/x] \)

“Level 0” for type checking source programs, using only [S-Valid-Uninterpreted]

Stratified Preservation

If \( \vdash_0 e :: T \) and \( e \rightarrow v \), then \( \vdash_m v :: T \) for some \( m \)

artifact of the metatheory
</Ph.D.>