Static Verification for Web Scripting Languages

Ravi Chugh
Web Platform
Web Platform

- Untrusted Code
- Scripting Languages
My Goal: Rigorous Foundations

Web Platform

Static Types and Logic!

Untrusted Code

Scripting Languages
1995  Web Page = Static HTML 
+ Links 
+ Client-Side Code

```
<html>
  <script type="javascript">
    loginButton.onclick = function () {
      if (passwordBox.value == "") {
        alert("Enter password!");
      } else {
        ...
      }
    }
  </script>
</html>
```
1995

No Static Types

“Dynamic” Features Make it Easy to Experiment...

... an easy-to-use “scripting language” as a companion to Java

Brendan Eich, JavaScript creator
2013
Asking whether Gmail would ever open up an API for developers to code add-ons and plug-ins with more official support and stability, Kowitz suggested it was unlikely. **Gmail is based on “hundreds of thousands” of lines of JavaScript**, Kowitz said, and it's a code base that “changes so quickly.” That said, Jackson said the Gmail team "loves" to see third-party functionality being developed, and the team tries to make developers aware of changes they know will affect those products.
Asked whether Gmail would ever open up an API for developers to code add-ons and plug-ins with more official support and stability, Kowitz suggested it was unlikely. **Gmail is based on “hundreds of thousands” of lines of JavaScript** it’s a code base that “changes so quickly.” That said, Jackson said the Gmail team "loves" to see third-party functionality being developed, and the team tries to make developers aware of changes they know will affect those products.

“Scripting” ?!?
## 2013

**JavaScript is Everywhere**

<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](http://stackoverflow.com/tags)

02-03-13
2013

JavaScript is Everywhere

Popular Server-Side Scripting Languages

<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>
</tbody>
</table>

http://stackoverflow.com/tags
02-03-13
2013

JavaScript is Everywhere
2013

JavaScript is Everywhere

Banking  Shopping  News

Third-Party Ads
2013

JavaScript is Everywhere

Third-Party Ads

Banking  Shopping  News

Third-Party Apps
2013

Third-Party
JavaScript is
Everywhere
2013

No longer just “scripting”

... an easy-to-use “scripting language” as a companion to Java
2013

No longer just “scripting”

... an easy-to-use “scripting language” as a companion to **Java**

JavaScript is **much harder** than Java
... New Yorker article, “The Itch,”...
ANNALS OF MEDICINE

THE ITCH

Its mysterious power may be a clue to a new theory about brains and bodies.

BY ATUL GAWANDE

JUNE 30, 2008

It was still shocking to M. how much a few wrong turns could change your life. She had graduated from Boston College with a degree in psychology, married at twenty-five, and had two children, a son and a daughter. She and her family settled in a town on Massachusetts’ southern shore. She worked for thirteen years in health care, becoming the director...
It was still shocking to M. how much a few wrong turns could change your life. She had graduated from Boston College with a degree in psychology, married at twenty-five, and had two children, a son and a daughter. She and her family settled in a town on Massachusetts’ southern shore. She worked for thirteen years in health care, becoming the director of a residence program for men who’d suffered severe head injuries. But she and her husband began fighting. There were betrayals. By the time she was thirty-two, her marriage had disintegrated. In the divorce, she lost possession of their home, and, amid her financial and psychological struggles, she saw that she was losing her children, too. Within a few years, she was drinking. She began dating someone, and they drank together. After a while, he brought some drugs home, and she tried them. The drugs got harder. Eventually, they were doing heroin, which turned out to be readily available from a street a block away from her apartment.
I want every article in print mode
Browser Extensions

Third-Party Developers

Customize Browser
Browser Extensions

PRINT
NEW YORKER
for (var elt in doc.getElementsByTagName("a")) {
    var url = elt.getAttribute("href");
    if (url.match(/^newyorker.com$/)) {
        elt.setAttribute("href", url + "?printable=true")
    }
}
Document (DOM)

host-site.com

newyorker.com?
printable=true

Browser Extensions

PRINT
NEW YORKER
Document (DOM)

host-site.com

evil.com

Network Manager

Passwords Bank Accts History

JavaScript Engine

Other Sensitive APIs

Browser Extensions

PRINT

NEW YORKER

EVIL

BUGGY
Enforcing JavaScript Security?

Dynamic Checking...
Expensive
Misses Attacks
Can Blame Wrong Party

Goal: Static Verification
My Dissertation

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

Static Types for Dynamic Languages
My Dissertation

New Techniques Apply to Traditional Languages

Static Types for Dynamic Languages

Many Challenges of Web are Independent of Language

Working with JavaScript Can Have Long-Term Impact
Static Types

Describe sets of run-time values

“booleans”

“integers”

“objects with foo field”

“non-null pointers”

“urls that are safe to visit”
Static Types
Prevent classes of run-time errors

“multiply int and bool”

“foo field not found”

“null pointer exception”

“redirect to evil.com”
f(x)

“actual args”
Type₁

⊂

“expected args”
Type₂

“function call produces no error”
"function call may produce error"
Refinement Types

Types + Logic

```
{ x \mid p } \subseteq \{ x \mid q \}
```

“actual args” ⊆ “expected args”

\[ p \implies q \]
Refinement Types

Types

\{ x \mid p \}\quad \text{Logic} \quad f :: \{ y \mid q \}

Nested Refinements [POPL ’12]
Key to Encode Dynamic Features
Expressiveness

Usability

Types + Logic

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

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

Expressiveness
<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
</table>

**Outline**

Why is JavaScript Important?  
Why is JavaScript Hard?  
Static Types via Logic  
Security via Logic  
Looking Ahead
Outline

Why is JavaScript Important?

Why is JavaScript Hard?

Static Types via Logic

Security via Logic

Looking Ahead
Outline

Why is JavaScript Hard?

- Lightweight Reflection
- Dictionary Objects
- Additional Challenges
<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
</table>

```javascript
function negate(x) {
    if (typeof x === "number") {
        Q: What is my “type”?  
    }
}
```
Q: What is my “type”?  

All JavaScript Values

<table>
<thead>
<tr>
<th>“number”</th>
<th>“object”</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1 3.14</td>
<td>{} {}</td>
</tr>
<tr>
<td>17 42 ...</td>
<td>{ “f”:17 } { “f”:42, “g”:true }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>“boolean”</th>
</tr>
</thead>
<tbody>
<tr>
<td>true false</td>
</tr>
</tbody>
</table>
function negate(x) {
    if (typeof x === "number") {
        // Code inside if block
    }
}

Q: What is my “type”?  
A: Described by “tag”
function negate(x) {
  if (typeof x === "number") {
    return 0 - x;
  } else {
    return !x;
  }
}
Outline

Why is JavaScript Hard?

- Lightweight Reflection
- Dictionary Objects
- Additional Challenges
obj.f  means  obj["f"]

Objects are "Dictionaries"
that map
string keys
to values

a.k.a.
"maps"
"hash tables"
"associative arrays"
function dispatch(table, op, i, j) {
    method
    table
    method
to invoke
function dispatch(table, op, i, j) {
  var fn = table[op];
  return fn(i, j);
}

var integerOps =
  {
    "+" : function (n, m) { ... },
    "-" : function (n, m) { ... }
  };

dispatch(integerOps, "*", 17, 42);
function dispatch(table, op, i, j) {
    var fn = table[op];
    return fn(i, j);
}

"expected args"

dependency between types of arguments

table object with an op key that stores a function on numbers
Outline

Why is JavaScript Hard?

- Lightweight Reflection
- Dictionary Objects
- Additional Challenges
Outline

Why is JavaScript Hard?

- Lightweight Reflection
- Dictionary Objects
- Extensible Objects
- Prototype Inheritance
- Array Semantics
Outline

Why is JavaScript Hard?

Unsupported in Prior Work

Lightweight Reflection

Dictionary Objects
Extensible Objects
Prototype Inheritance
Array Semantics
<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
</table>

Outline

Why is JavaScript Important?

Why is JavaScript Hard?

Static Types via Logic

Security via Logic

Looking Ahead
<table>
<thead>
<tr>
<th>&quot;number&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
</tr>
<tr>
<td>3.14</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>42</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

All JavaScript Values

```javascript
{}   { "f": 17 }
...
{ "f": 42, "g": true }
```

true    false
### Why JavaScript?

#### Types via Logic

<table>
<thead>
<tr>
<th>x</th>
<th>tag(x) = “number&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>3.14</td>
</tr>
<tr>
<td>17</td>
<td>42</td>
</tr>
</tbody>
</table>

#### true, false

| x | tag(x) = “boolean" |

| f:17 |
| f:42, g:true |

---

### All JavaScript Values

#### Security via Logic

#### Challenges

#### Looking Ahead
### Types via Logic

| {x | x = true} | true | false |

| {x | x > 0}     | 3.14 | ...   |

| {x | integer(x)} | -1   | 17    | 42    |

| {}             | {“f”: 17} |

| ...            | {“f”: 42, “g”: true} |

All JavaScript Values
Refinement Types

\{ x \mid p \}

“set of values X s.t. formula p is true”
Why JavaScript? | Challenges | Types via Logic | Security via Logic | Looking Ahead

\[ f(x) \]

"actual args"

```
Type_1
```

\[ \subseteq \]

"expected args"

```
Type_2
```
Why JavaScript? | Challenges | Types via Logic | Security via Logic | Looking Ahead

---

**f(x)**

"actual args" \[ \{ x \mid p \} \] \[ \subseteq \] \[ \{ x \mid q \} \] "expected args"

\( p \implies q \)
Subtyping is Set Inclusion is Implication

\[
\left\{ x \mid p \right\} \subseteq \left\{ x \mid q \right\}
\]

\[ p \quad \Rightarrow \quad q \]
Outline

Static Types via Logic

- Lightweight Reflection
- Dictionary Objects
- Key Technical Innovations
- Evaluation
//: negate :: (x:NumOrBool) → NumOrBool

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

Type Annotation in Comments

Syntactic Sugar for Common Types

```
NumOrBool ≡ { v | tag(v) = "number" v tag(v) = "boolean" }
```
//: negate :: (x:NumOrBool) → NumOrBool

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

{ x | tag(x) = "number" } ⊆ { x | tag(x) = "number" }

“expected args”

tag(x) = "number" ⇒ tag(x) = "number"
//: negate :: (x:NumOrBool) → NumOrBool

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

{ x | not(tag(x) == “number”) ∧ (tag(x) == “number” ∨ tag(x) == “boolean”) } ⊆ { x | tag(x) == “boolean” }
function negate(x) {
  if (typeof x == "number") {
    return 0 - x;
  } else {
    return !x;
  }
}
//: negate :: (x:NumOrBool) → \{ y | tag(y) = tag(x) \}

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

Static Types via Logic

- Lightweight Reflection
- Dictionary Objects
- Key Technical Innovations
- Evaluation
### All JavaScript Values

$$\{ x \mid \text{tag}(x) = \text{"object"} \}$$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{}</code></td>
<td><code>{f: true}</code></td>
<td><code>{f: 17}</code></td>
</tr>
<tr>
<td><code>{f: 42, g: null}</code></td>
<td></td>
<td><code>{f: 42, g: null}</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
### All JavaScript Values

<table>
<thead>
<tr>
<th>{}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ x \mid \text{tag}(x) = &quot;object&quot; \land \text{has}(x, &quot;f&quot;) }</td>
</tr>
<tr>
<td>{ &quot;f&quot;:17 }</td>
</tr>
<tr>
<td>{ &quot;f&quot;:true }</td>
</tr>
<tr>
<td>{ &quot;f&quot;:42, &quot;g&quot;:null }</td>
</tr>
<tr>
<td>\…</td>
</tr>
</tbody>
</table>
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

All JavaScript Values

\{ x \mid \text{tag}(x) = \text{“object”} \land \\
\text{has}(x, \text{“f”}) \land \\
\text{tag}(\text{sel}(x, \text{“f”})) = \text{“number”} \}\n
\{ “f”: true \}

\{ “f”: 17 \}

\{ “f”: 42, “g”: null \}

\ldots
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

All JavaScript Values

```javascript
{ x |
  sel(x, "f") = true ∧
  dom(x) = { "f" } }

{ "f": true }

{ "f": 17 }

{ "f": 42, "g": null }

...
```
//: dispatch ::
//:   ({{table | tag(table) = “object”},
//:     {{op | tag(op) = “string”
//:       ∧ has(table, op)
//:       ∧ sel(table, op) :: (Num, Num) → Num}
//:     ,
//:     {{i | tag(i) = “number”},
//:     {{j | tag(j) = “number”}})
//:   → Num

function dispatch(table, op, i, j) {
    var fn = table[op];
    return fn(i, j);
}
```javascript
function dispatch(table, op, i, j) {
    var fn = table[op];
    return fn(i, j);
}
```
var integerOps = { "+" : ..., "-" : ... };
dispatch (integerOps, "*", 17, 42);

"actual args"

\{ op \mid op = "*" \} \subseteq \{ op \mid \text{has}(integerOps, op) \}

op = "*" \implies \neg \text{has}(integerOps, op)

Type Checking Prevents Run-Time Error
Outline

Static Types via Logic

Lightweight Reflection
Dictionary Objects
Key Technical Innovations
Evaluation
<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
<tbody>
<tr>
<td>prior refinement types</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>lightweight reflection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>dictionary objects</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>extensible objects</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>array semantics</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Dependent JavaScript (DJS)  
[POPL ’12, OOPSLA ’12]
Why JavaScript? | Challenges | Types via Logic | Security via Logic | Looking Ahead
---|---|---|---|---

Key Challenge:
Function Subtyping as Logical Implication
### Why JavaScript?

**Types via Logic**

- **Security via Logic**

### Challenges

- **Looking Ahead**

#### 42 + negate (17)

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

#### \[
\{ f \mid f :: (x:\text{Num}) \rightarrow \text{Num} \} \\
\text{“expected function”}
\]
How to Prove

How to Encode Type System **Inside** Logic?

Prior Refinement Systems **Disallow** Function Types Inside Logic!
Why JavaScript?

Types via Logic

Security via Logic

Looking Ahead

Types

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

\[ \mid T_1 \rightarrow T_2 \]

Refinement Logic

[SAT + Decidable Theories]

[Satisfiability Modulo Theories (SMT)]

SMT Solver (e.g. Z3)

\[ p \Rightarrow q \]

\[ \checkmark \]

\[ \times \]

[Prior State-of-the-Art]
### Types

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

### Refinement Logic

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

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

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

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

\[ \text{NumOrBool} = \{ v \mid \text{tag}(v) = \text{“number”} \lor \text{tag}(v) = \text{“boolean”} \} \]
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"}) \text{???} \}
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

Types

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

\[ \vdash T_1, T_2 \]

Refinement Logic

\[ p ::= p \land q \mid p \lor q \mid \neg p \]

\[ \vdash x = y \mid x > y \mid \ldots \]

\[ \vdash \text{tag}(x) = \text{"number"} \mid \ldots \]

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

\[ \vdash \text{sel}(x, k) :: T_1 \rightarrow T_2 \mid \ldots \]

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

\[
\begin{align*}
\text{Uninterpreted Predicate in Logic} & \quad \Rightarrow \quad \text{Distinct Uninterpreted Constants in Logic...} \\
\Rightarrow & \quad \text{f} \\
(x:\text{NumOrBool}) \rightarrow \{ y | \text{tag}(y) = \text{tag}(x) \} \\
(x:\text{Num}) \rightarrow \text{Num}
\end{align*}
\]
Decidable Encoding

Uninterpreted Predicate in Logic

Distinct Uninterpreted Constants in Logic...

w/ Types Nested Inside

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

\( f :: (x:Num) \rightarrow \text{Num} \)
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 \textbf{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

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

**Step 1:**

\[ f = \text{negate} \]

**Step 2:**

\[ f :: (x:\text{NumOrBool}) \to \{ y \mid \text{tag}(y) = \text{tag}(x) \} \]
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
Nested Refinements [POPL ’12]
Key to Encode Dictionary Objects
(Even in Statically Typed Languages!)
<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Prior Refinement Types</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dependent JavaScript (DJS)</td>
<td>[POPL ’12, OOPSLA ’12]</td>
<td></td>
</tr>
</tbody>
</table>

- lightweight reflection: ✓ ✓
- dictionary objects: ✗ ✓
- extensible objects: ✗ ✓
- prototype inheritance: ✗ ✓
- array semantics: ✗ ✓
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

JavaScript, Python, Ruby

lightweight reflection

dictionary objects

extensible objects

prototype inheritance

array semantics

Dependent JavaScript (DJS)

[POPL ’12, OOPSLA ’12]
<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
</table>

# Outline

## Static Types via Logic

- Lightweight Reflection
- Dictionary Objects
- Key Technical Innovations
- Evaluation
Benchmarks

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

LOC
before/after

| 306 | 408 (+33%) |

Chosen to **Stretch** the Limits of DJS

Why JavaScript?  Challenges  Types via Logic  Security via Logic  Looking Ahead
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>LOC before/after</th>
</tr>
</thead>
<tbody>
<tr>
<td>13 Excerpts from:</td>
<td></td>
</tr>
<tr>
<td><em>JavaScript, Good Parts</em></td>
<td></td>
</tr>
<tr>
<td>SunSpider Benchmark Suite</td>
<td>306</td>
</tr>
<tr>
<td>Google Closure Library</td>
<td>408 (+33%)</td>
</tr>
<tr>
<td>9 Browser Extensions from:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>383 (+19%)</td>
</tr>
<tr>
<td>2 Examples from:</td>
<td></td>
</tr>
<tr>
<td>Google Gadgets</td>
<td>1,003</td>
</tr>
<tr>
<td></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** [STOP ’11]

| TOTALS  | 1,630 | 1,818 (+12%) |
Why JavaScript?  |  Challenges  |  Types via Logic  |  Security via Logic  |  Looking Ahead

---

Type Inference Will Enable Larger Examples

TOTALS

1,630  |  1,818 (+12%)
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>LOC before/after</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>13 Excerpts from:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>JavaScript, Good Parts</em></td>
<td>306 (306)</td>
<td>10 sec</td>
</tr>
<tr>
<td>SunSpider Benchmark Suite</td>
<td>408 (+33%)</td>
<td></td>
</tr>
<tr>
<td>Google Closure Library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 Browser Extensions from:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Guha et al. Oakland ’11]</td>
<td>321 (321)</td>
<td>3 sec</td>
</tr>
<tr>
<td>2 Examples from:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Google Gadgets</td>
<td>1,003 (1,003)</td>
<td>19 sec</td>
</tr>
<tr>
<td><strong>TOTALS</strong></td>
<td>1,630 (1,630)</td>
<td>32 sec</td>
</tr>
<tr>
<td></td>
<td>1,818 (+12%)</td>
<td></td>
</tr>
</tbody>
</table>
Already Improved by Simple Optimizations

- Avoid SMT Solver When Possible
- Reduce Precision for Common Patterns

Plenty of **Room for Improvement**
Outline

Static Types via Logic

Lightweight Reflection
Dictionary Objects
Key Technical Innovations
Evaluation
Outline

Why is JavaScript Important?
Why is JavaScript Hard?
Static Types via Logic
Security via Logic
Looking Ahead
Outline

Security via Logic

Browser Extensions

Types for Fine-Grained Policies
Document (DOM)

host-site.com

Browser Extensions

PRINT
NEW YORKER

EVIL

BUGGY

Network Manager

Passwords
Bank Accts
History

JavaScript
Engine

Other
Sensitive
APIs
Access **All** Resources

Policy Expressiveness

---

| Why JavaScript? | Challenges | Types via Logic | Security via Logic | Looking Ahead |
Access Some Resources

can access DOM
cannot access History
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

Policy Expressiveness

coarse-grained and uninformative!

can access DOM

cannot access History
<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
</table>

**Policy Expressiveness**

- coarse-grained and uninformative!
- similar situation for other app platforms
GOAL

Fine-Grained Policies

Policy Expressiveness

Why JavaScript? Challenges Types via Logic Security via Logic Looking Ahead

App Store

Android

Apple
GOAL  Fine-Grained Policies

can write “a” elements but only “href” attribute when original “href” value starts with “newyorker.com” and is prefix of new value
Outline

Security via Logic

Browser Extensions
Types for Fine-Grained Policies
Types for Security
+ JavaScript Verification

= Fine-Grained Web Security
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

Static Verification (DJS)

Developer writes policy in **logic**!

Security Expert reviews **only** policy

User reads **informative** policy
Network Manager
request, ...

JS Engine
eval, ...

DOM
getAttr, setAttr, ...

request ::
  url: Str
→ Str
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

Network Manager

request, ...

JS Engine

eval, ...

DOM

getAttr, setAttr, ...

request ::

{url: Str | WhiteListed(url)}

→ Str
Network Manager request, ...

JS Engine `eval, ...

DOM `getAttr, `setAttr, ...

eval ::
  s:Str
→ Void
<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Manager request, ...</td>
<td>JS Engine eval, ...</td>
<td>DOM getAttr, setAttr, ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
eval :: 
{s:Str | not(Tainted(s))} 
→ Void
```
<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Manager request, ...</td>
<td>JS Engine eval, ...</td>
<td>DOM getAttr, setAttr, ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

getAttr ::

\[
\text{elt:Element} \\
\rightarrow \text{attr:Str} \\
\rightarrow \text{Str}
\]
<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Manager request, ...</td>
<td>JS Engine eval, ...</td>
<td>DOM getAttr, setAttr, ...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

getAttr ::

\[
\text{elt:Element} 
\rightarrow \{ \text{attr:Str} \mid \text{CanRead(elt, attr)} \}\}
\rightarrow \text{Str}
\]
setAttr ::
  elt:Element
  →  attr:Str
  →  y:Str
  →  Void
Network Manager request, ...

JS Engine eval, ...

DOM getAttr, setAttr, ...

```
setAttr ::
  elt:Element
  attr:Str
  → {y:Str | CanWrite(elt, attr, y)}
  → Void
```
<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Network Manager</td>
<td>JS Engine</td>
<td>DOM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>request, ...</td>
<td>eval, ...</td>
<td>getAttr, setAttr, ...</td>
<td></td>
</tr>
</tbody>
</table>

**Refined APIs Mediate Access**

**Typecheck Extension with Policy**
forall url.
    not (url = "evil.com")
⇒ WhiteListed (url)

var s = request("evil.com");

finite blacklist
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

forall url.
    not (url = “evil.com”)
⇒ WhiteListed (url)

//: url:Str

var s = request(url);
forall url.
not (url = "evil.com")
⇒ WhiteListed (url)

if (url != "evil.com") {
  var s = request(url);
}

Why JavaScript?
Challenges
Types via Logic
Security via Logic
Looking Ahead
forall url.
not (url = "evil.com")
⇒ WhiteListed (url)

if (url !="evil.com") {
    var s = request(url);
    eval(s);
}

s:Str

might be tainted!
Why JavaScript?

Security via Logic

Types via Logic

Looking Ahead

Challenges

if (url != "evil.com") {
    var s = request(url);
    s = sanitize(s);
    eval(s);
}

forall url.
    not (url = "evil.com")
⇒ WhiteListed (url)

STATICALLY VERIFIED!
WhiteListed ("ny.com")
WhiteListed ("golf.com")

forall url.

\[\text{WhiteListed (url)} \Rightarrow \neg \text{not (Tainted (request (url)))}\]

trust that whitelisted servers return untainted strings
Why JavaScript?  Challenges  Types via Logic  Security via Logic  Looking Ahead

WhiteListed ("ny.com")
WhiteListed ("golf.com")

forall url.
    WhiteListed (url)
⇒ not (Tainted (request (url)))

var s = request("ny.com");
eval(s);

no need to sanitize

STATICALLY VERIFIED!
forall e, oldUrl, newUrl.

ValueOf (e, “href”, oldUrl) ∧
StrPrefix (“newyorker.com”, oldUrl) ∧
StrPrefix (oldUrl, newUrl)

⇒ CanWrite (e, “href”, newUrl)

for (var elt in doc.getEltsByTagName(“a”)) {
    var url = elt.getAttr(“href”);
    if (url.match(/^newyorker.com$/)) {
        elt.setAttr(“href”, url + “?printable=true”);
    }
}
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

✓ ValueOf (elt, “href”, url) \∧
StrPrefix ("newyorker.com", url) \∧
StrPrefix (url, url + "?printable=true")

⇒ CanWrite (e, “href”, url+"?printable=true")

for (var elt in doc.getEltsByTagName("a")) {
  var url = elt.getAttr("href")
  if (url.match(/\^newyorker.com/)) {
    elt.setAttr("href", url + "?printable=true");
  }
}
ValueOf (elt, “href”, url) ∧
StrPrefix ("newyorker.com", url) ∧
StrPrefix (url, url + "?printable=true")
⇒ CanWrite (e, “href”, url + "?printable=true")

for (var elt in doc.getEltsByTagName("a")) {
    var url = elt.getAttr("href");
    if (url.match(/^newyorker.com$/)) {
        elt.setAttr("href", url + "?printable=true");
    }
}
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

ValueOf (elt, “href”, url) ∧
StrPrefix (“newyorker.com”, url) ∧
StrPrefix (url, url + “?printable=true”) ⇒ CanWrite (e, “href”, url+“?printable=true”)

for (var elt in doc.getEltsByTagName(“a”)) {
    var url = elt.getAttr(“href”);
    if (url.match(/^newyorker.com/)) {
        elt.setAttr(“href”, url + “?printable=true”);
    }
}
Why JavaScript? Challenges Types via Logic Security via Logic Looking Ahead

(ValueOf (elt, "href", url) ∧ StrPrefix ("newyorker.com", url) ∧ StrPrefix (url, url + "?printable=true") ⇒ CanWrite (e, "href", url + "?printable=true")

for (var elt in doc.getEltsByTagName("a")) {
    var url = elt.getAttr("href");
    if (url.match(/^newyorker.com$/)) {
        elt.setAttr("href", url + "?printable=true");
    }
}

STATICALLY VERIFIED!
ValueOf (elt, “href”, url) ∧
StrPrefix ("newyorker.com", url) ∧
StrPrefix (url, url + “?printable=true”) ⇒ CanWrite (e, “href”, url+“?printable=true”)

for (var elt in doc.getEltsByTagName("a")) {
  var url = elt.getAttr("href");
  if (url.match(/\^newyorker.com/)) {
    elt.setAttr("href", "evil.com");
  }
}
ValueOf (elt, “href”, url) ∧
StrPrefix (“newyorker.com”, url) ∧
StrPrefix (url, “evil.com”)
⇒ CanWrite (elt, “href”, “evil.com”)

for (var elt in doc.getEltsByTagName(“a”)) {
  var url = elt.getAttr(“href”);
  if (url.match(/^newyorker.com$/)) {
    elt.setAttr(“href”, “evil.com”);
  }
}
### Why JavaScript? Security via Logic

#### Challenges

<table>
<thead>
<tr>
<th>✓ ValueOf (elt, “href”, url) ∧</th>
<th>✓ StrPrefix (“newyorker.com”, url) ∧</th>
</tr>
</thead>
<tbody>
<tr>
<td>❌ StrPrefix (url, “evil.com”)</td>
<td>⇒ CanWrite (e, “href”, “evil.com”)</td>
</tr>
</tbody>
</table>

```javascript
for (var elt in doc.getEltsByTagName("a")) {
  var url = elt.getAttr("href");
  if (url.match(/^newyorker.com$/)) {
    elt.setAttr("href", "evil.com")
  }
}
```


Preliminary Benchmarks

Secure Extensions Written in OCaml

Types for Security
+ JavaScript Verification

= Fine-Grained Web Security

[ESOP ’10, Guha et al. ’11]
[POPL ’12, OOPSLA ’12]
[current]
Secure Extensions Written in OCaml

Ported to DJS

~400 LOC so far

Well-typed programs don’t have run-time errors and conform to security policies

Guha et al. ’11
<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
</table>

**Outline**

**Security via Logic**

- Browser Extensions
- Types for Fine-Grained Policies
Outline

Why is JavaScript Important?
Why is JavaScript Hard?
Static Types via Logic
Security via Logic
Looking Ahead
Web Platform

Dependent JavaScript

Untrusted Code

Scripting Languages
Usability

**TypeScript**

- Lightweight (unsound) static checking tools becoming popular
- Translate for additional checking and IDE support

**Dependent JavaScript (DJS)**

**Verification**

Expressiveness
Web Platform

- Untrusted Code
- Scripting Languages

Dependent JavaScript Tool Support
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
<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
</table>

### Web Platform

- **Untrusted Code**
- **Scripting Languages**

- Dependent JavaScript
- Tool Support
- Policy Specification

---

149
Why JavaScript? | Challenges | Types via Logic | Security via Logic | Looking Ahead

**Static Verification (DJS)**

- Developer must **write** policy
- Security Expert
- and
- User must **read** policy
<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
</table>

**Developer must write policy**
- Infer from traces
- Incentives from platform

**Security Expert must read policy**
- Modeling tools to analyze policies

**User must read policy**
- Traces to demonstrate behavior
- Community review-and-recommend services
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

Dependent JavaScript

Tool Support

Policy Specification

Dynamic Enforcement

Web Platform

Untrusted Code

Scripting Languages

---

152
Sta2c Verifica2on (DJS)

Why JavaScript? | Challenges | Types via Logic | Security via Logic | Looking Ahead

Developer

Security Expert

User
**Why JavaScript?**

- Types via Logic
- Security via Logic

**Challenges**

```javascript
... eval("..."); ...
```

**Dynamic Code Loading**

Impossible to *statically* analyze code you don’t have!
### Why JavaScript?

<table>
<thead>
<tr>
<th>Why JavaScript?</th>
<th>Challenges</th>
<th>Types via Logic</th>
<th>Security via Logic</th>
<th>Looking Ahead</th>
</tr>
</thead>
</table>

### Static Verification (DJS)
- Developer
- Security Expert
- User

### Dynamic Enforcement
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

Dynamic Enforcement

Old Types

New Types

```
... eval("...");
//: #assume New Types
...
```
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

eval("...");

//: #assume New Types

• Dynamic Checking

• Staged Analysis [PLDI ’09]
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

Static Verification (DJS)

Developer

Security Expert reviews only policy

User

Dynamic Enforcement
Why JavaScript?

Challenges

Types via Logic

Security via Logic

Looking Ahead

Static Verification (DJS)

Security Expert reviews only policy

SMT Solver

trust
Eliminate from TCB via Formal Verification

Static Verification (DJS) → trust

Security Expert

trust

SMT Solver

Eliminate from TCB à la Proof-Carrying Code [PLDI ’10]
Why JavaScript?

- Types via Logic
- Security via Logic

Looking Ahead

Web Platform

- Static Types and Logic!
- Untrusted Code
- Scripting Languages

Dependent JavaScript
Tool Support
Policy Specification
Dynamic Enforcement
Trust Base
Static Types for:
Reliability / Security
Development Tools
Performance
• Additional Just-in-Time Optimizations
• Compile to Custom Architectures
Dataflow Analysis and Race Detection

Concurrent Programs

Scripting Languages

Run-Time Instrumentation

Untrusted Code

Program Analysis via Set Constraints

Static Types and Logic!

[PLDI '08]

[ESOP '10, PLDI '10]
[POPL '12, OOPSLA '12]

[STOP '11]

[PLDI '09]
Thanks!

Scripting Languages are Pervasive

DJS: Precise Static Types via Logic

Towards Fine-Grained Web Security