Information Flow Analysis for a Dynamically Typed Functional Language with Staged Metaprogramming

Martin Lester\textsuperscript{1}   Luke Ong\textsuperscript{1}   Max Schäefer\textsuperscript{2}

\textsuperscript{1}Department of Computer Science, University of Oxford

\textsuperscript{2}IBM T.J. Watson Research Center

2012–11–22
Information Flow Analysis for JavaScript

Martin Lester\textsuperscript{1}  Luke Ong\textsuperscript{1}  Max Schäefer\textsuperscript{2}

\textsuperscript{1}Department of Computer Science, University of Oxford

\textsuperscript{2}IBM T.J. Watson Research Center

2012–11–22
Motivation

Web applications written in JavaScript regularly handle sensitive data.

- Reasoning about their security properties is an important problem.
- JavaScript is a difficult language to reason about.

Why is JavaScript difficult?

- Poorly understood, quaint semantics.
- Many features: mutable state, exceptions, dynamic types, prototype-based inheritance, type coercion, first-class functions . . .
Motivation

Web applications written in JavaScript regularly handle sensitive data.
▶ Reasoning about their security properties is an important problem.
▶ JavaScript is a difficult language to reason about.

Why is JavaScript difficult?
▶ Poorly understood, quaint semantics.
▶ Many features: mutable state, exceptions, dynamic types, prototype-based inheritance, type coercion, first-class functions . . . and `eval`.

What have we done about it?
▶ Produced an information flow analysis for a language with many of JavaScript’s features, including run-time code generation.
Many research papers claim that `eval` is used rarely or only in trivial ways. A recent survey shows otherwise. Examples include:

- concatenating strings to form variable names;
- simulating higher order functions;
- bizarre or seemingly pointless invocations.
Staged Metaprogramming

JavaScript’s `eval` is a form of metaprogramming: it allows construction, manipulation and evaluation of program code at run-time. But metaprogramming is not new:

- Lisp allows quoting and unquoting of code.
- This restricts manipulation to plugging holes in abstract syntax trees.

Example

Plug: \(<x>\)

into: \((\text{fun}(<\text{--}>)(x + 1))\)

to get: \((\text{fun}(x)(x + 1))\)

✓
Staged Metaprogramming

JavaScript’s `eval` is a form of metaprogramming: it allows construction, manipulation and evaluation of program code at run-time. But metaprogramming is not new:

- Lisp allows quoting and unquoting of code.
- This restricts manipulation to plugging holes in abstract syntax trees.

Example

Plug: \( \langle x \rangle \) into: \((\text{fun}(\langle-\rangle)\{x + 1\})\)

to get: \((\text{fun}(x)\{x + 1\})\) ✓

Plug: \( \langle \rangle \)(\text{fun}(y)\{y\}) into: \((\text{fun}(x)\{\langle-\rangle\})\)

to get: \((\text{fun}(x)\{})\)(\text{fun}(y)\{y\}) x
Staged Metaprogramming

JavaScript’s `eval` is a form of metaprogramming: it allows construction, manipulation and evaluation of program code at run-time. But metaprogramming is not new:

- Lisp allows quoting and unquoting of code.
- This restricts manipulation to plugging holes in abstract syntax trees.

Example

Plug: `⟨x⟩` into: `(fun(⟨−⟩){x + 1})` to get: `(fun(x){x + 1})`

Plug: `⟨{}⟩(fun(y){y})` into: `(fun(x){⟨−⟩})` to get: `(fun(x){})(fun(y){y})`

In an attempt to understand better the behaviour of `eval`, we study a language with staged metaprogramming in the style of Lisp.

- Syntactically, staged just means that quotes can be nested.

✓

×
Definition of Noninterference

Consider a setting where inputs and outputs to a program are marked with security levels, such as $H$ for high security and $L$ for low security.

Example

In a Web application, high/low input/output channels might be:

- high input — a “password” input box
- low input — any other text box
- high output — encrypted connection to webserver
- low output — unencrypted connection to webserver

- If the high inputs of a program cannot affect the low outputs, the program satisfies noninterference.
- This means an attacker who can only view low outputs cannot gain any information about high inputs.
- Noninterference is a popular information security property.
Consider the program:

\[ \text{if} (h) \{ \text{true} \} \text{ else } \{ / \} \]

The result of the program can be either true or /.

- As the value / can flow directly to the result, we say that there is a direct flow from / to the result.
- If / is false, then the result of the program is equal to \( h \). As this dependency arises only through control flow, we say that there is an indirect flow from \( h \) to the result.
Verifying Noninterference

Some early work on noninterference added a monitor to a program to track and enforce security levels of variables. Unfortunately:

- A 1-safety property of a program is one for which a violation can be shown by a single, finite trace of a program.
- Monitoring is good for enforcing 1-safety properties.
- Showing violation of a 2-safety property requires two traces.
- Noninterference is a 2-safety property.

Some recent research revisits monitoring and works around this by combining it with simple static analysis.

- Monitoring is good at handling metaprogramming.
- But it is still a dynamic (rather than static) analysis.

There is a large body of research on verifying noninterference with type systems, for example in ML.

- This is difficult to apply to dynamically typed languages.
Information Flow Analysis

▶ An information flow analysis tells us, for any variable $x$, whether it is used in the computation of another variable $y$.

▶ Alternatively, in our setting with marked security levels, we can check whether any value labelled with level $H$ is used to compute other variables (or the result of a program).

▶ “$x$ is not used to compute $y$” is a stronger claim than “the value of $x$ does not affect the value of $y$”.

Example

This program clearly uses $x$ in its computation of $y$, but $y$ is always 0, so $x$ does not affect its value.

```
x := 10;
while (x <> 0) {
x := x - 1;
}
y := x;
```

▶ We can use an information flow analysis to verify noninterference.
Outline

Motivation
  Metaprogramming
  Noninterference and Information Flow

Outline

Syntax and Semantics of SLamJS

Information Flow Analysis for SLamJS
  CFA for SLamJS
  Information Flow for SLamJS
  Proof Outline

Implementation and Examples

Future Work

Conclusion
Syntax of SLamJS

Booleans
\[ b ::= \text{true} \mid \text{false} \]

Strings
\[ s \in \text{String} \]

Numbers
\[ n \in \text{Number} \]

Names
\[ x \in \text{Name} \]

Constants
\[ k ::= \text{undef} \mid \text{null} \mid b \mid s \mid n \]

Expressions
\[ e ::= k \mid \{ s : e \} \mid x \mid \text{fun}(x)\{e\} \mid e(e) \mid \text{box} \ e \]
\[ \mid \text{unbox} \ e \mid \text{run} \ e \mid \text{if}(e)\{e\} \text{ else }\{e\} \mid e[e] \]
\[ \mid e[e] = e \mid \text{del} \ e[e] \mid (e, \rho) \mid \text{run} \ e \text{ in } \rho \]

Values
\[ v, v^0 ::= (\text{fun}(x)\{e\}, \rho) \]
\[ v^n ::= k \mid \{ s : v^n \} \mid (\text{box} \ v^{n+1}) \]
\[ v^{n+1} ::= x \mid (\text{fun}(x)\{v^{n+1}\}) \mid (v^{n+1}(v^{n+1})) \]
\[ \mid (\text{run} \ v^{n+1}) \mid (\text{if}(v^{n+1})\{v^{n+1}\} \text{ else} \{v^{n+1}\}) \]
\[ \mid (v^{n+1}[v^{n+1}]) \mid (v^{n+1}[v^{n+1}] = v^{n+1}) \mid (\text{del} \ v^{n+1}[v^{n+1}]) \]
\[ v^{n+2} ::= (\text{unbox} \ v^{n+1}) \]

Environments
\[ \rho \in \text{Name} \xrightarrow{\text{fin}} v^0 \]
Syntax of SLamJS

Booleans  \( b \) \( ::= \) true | false
Strings  \( s \) \( \in \) String
Numbers  \( n \) \( \in \) Number
Names  \( x \) \( \in \) Name
Constants  \( k \) \( ::= \) undef | null | \( b \) | \( s \) | \( n \)
Expressions  \( e \) \( ::= \) \( k \) | \( x \) | fun\((x)\){\( e \)} | \( e(e) \) | box \( e \)

\[ \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \qua
Semantics of SLamJS

We define evaluation contexts and an evaluation relation $\triangleright$ in a typical way. Here are some illustrations of its behaviour:

- $(\text{fun}(x)\{x\})(1) \xrightarrow{\triangleright} 1$
- $((\text{fun}(x)\{\text{fun}(y)\{x\}\})(1))(2) \xrightarrow{\triangleright} 1$
- if(true){1} else{false} $\xrightarrow{\triangleright} 1$
- $(\text{fun}(x)\{\text{run(box x)}\})(0) \xrightarrow{\triangleright} 0$
- run(box(if(unbox(box true)){1} else{false})) $\xrightarrow{\triangleright} 1$
Staged Metaprogramming in SLamJS

SLamJS allows staged metaprogramming with these constructs:

- **box** — turns an expression into a code value;
- **unbox** — marks a hole in a code value that can be filled by another code value;
- **run** — executes a code value as code.

For example:

```plaintext
let y = box x in
let z = box (1 + (unbox y)) in
let x = 1 in
run z
```
Staged Metaprogramming in SLamJS

SLamJS allows staged metaprogramming with these constructs:

- **box** — turns an expression into a code value;
- **unbox** — marks a hole in a code value that can be filled by another code value;
- **run** — executes a code value as code.

For example:

```plaintext
let y = box x in
let z = box (1 + (unbox y)) in
let x = 1 in
run z
```

After evaluation, the program will become:

```plaintext
let x = 1 in
run (box (1 + x))
```

Static and dynamic scoping.
No $\alpha$-equivalence.
Staged Metaprogramming in SLamJS

SLamJS allows staged metaprogramming with these constructs:

- **box** — turns an expression into a code value;
- **unbox** — marks a hole in a code value that can be filled by another code value;
- **run** — executes a code value as code.

For example:

```plaintext
let y = box x in
let z = box (1 + (unbox y)) in
let x = 1 in
run z
```

```plaintext
let z = box (1 + (unbox (box x))) in
let x = 1 in
run z
```

```plaintext
let z = box (1 + x) in
let x = 1 in
run z
```

Static and dynamic scoping. No α-equivalence.
Staged Metaprogramming in SLamJS

SLamJS allows staged metaprogramming with these constructs:

- **box** — turns an expression into a code value;
- **unbox** — marks a hole in a code value that can be filled by another code value;
- **run** — executes a code value as code.

For example:

```plaintext
let y = box x in
let z = box (1 + (unbox y)) in
let x = 1 in
run z

let z = box (1 + x) in
let x = 1 in
run z
```

Static and dynamic scoping.
No \(\alpha\)-equivalence.
Staged Metaprogramming in SLamJS

SLamJS allows staged metaprogramming with these constructs:

- **box** — turns an expression into a code value;
- **unbox** — marks a hole in a code value that can be filled by another code value;
- **run** — executes a code value as code.

For example:

```plaintext
let y = box x in
let z = box (1 + (unbox y)) in
let x = 1 in
run z
```

```plaintext
let z = box (1 + (unbox (box x))) in
let x = 1 in
run z
```

```plaintext
let z = box (1 + x) in
let x = 1 in
run z
```

```plaintext
let x = 1 in
run (box (1 + x))
```

```plaintext
let x = 1 in
1 + x
```

```plaintext
1 + 1
2
```

Static and dynamic scoping. No $\alpha$-equivalence.
Some Reduction Rules

(LOOKUP) \((x, \rho) \quad \longrightarrow \quad \rho(x)\)

(APPLY) \(((\text{fun}(x)\{e\}, \rho)(v))) \quad \longrightarrow \quad (e, \rho[x \mapsto v])\)

(UNBOX) \((\text{unbox} (\text{box} \; v^1)) \quad \longrightarrow \quad (v^1)\)

(RUN) \((\text{run} (\text{box} \; v^1) \; \text{in} \; \rho) \quad \longrightarrow \quad (v^1, \rho)\)

(IFTRUE) \((\text{if} (\text{true})\{e_1\} \; \text{else} \{e_2\}) \quad \longrightarrow \quad e_1\)

(IFFALSE) \((\text{if} (\text{false})\{e_1\} \; \text{else} \{e_2\}) \quad \longrightarrow \quad e_2\)

There is only a single rule for \(\rightarrow^n\):

\[
C^m_n\langle e \rangle \quad \overset{m}{\longrightarrow} \quad C^m_n\langle e' \rangle \quad \text{if} \; e \quad \overset{n}{\longrightarrow} \quad e'
\]

We write \(\centernot\overset{\sqcap}{\longrightarrow}\) for the union over all \(n\) of \(\overset{n}{\longrightarrow}\), and \(\overset{\sqcap}{\longrightarrow}^*\) for its reflexive, transitive closure.
Information Flow in SLamJS

To allow us to express information flow in SLamJS, we augment it with explicit security level markers:

Markers \( m \in \text{Marker} \)
Expressions \( e ::= \ldots | (m : e) \)

Direct flows from a marked value are tracked by the marker being part of the value:

\[
\text{if} (\text{false}) \{ \text{true} \} \text{ else} \{L : l\} \xrightarrow{\text{H}} L : l
\]

No further treatment is needed.

Markers block reductions that might result in an indirect flow:

\[
\text{if} (H : h) \{ \text{true} \} \text{ else} \{L : l\} \not\xrightarrow{\text{H}}
\]

We introduce lift rules that move markers towards the top level of an expression:

\[
\text{if} (H : h) \{ \text{true} \} \text{ else} \{L : l\} \xrightarrow{\text{H}} H : (\text{if} (h) \{ \text{true} \} \text{ else} \{L : l\})
\]

Effectively, an indirect flow is turned into a direct one.
Erasure captures what it means for a value *not* to be used in a computation.

- The *M*-erasure of \( e \), written \( \lfloor e \rfloor_M \), is \( e \) with all subexpressions marked by \( m \notin M \) replaced with \( _\).
- \( _\) behaves like an unbound variable.
- \( \lfloor \text{if}(\text{true})\{\text{false}\} \text{ else}\{H : h\} \rfloor_L = \text{if}(\text{true})\{\text{false}\} \text{ else}\{\_\} \xrightarrow{\mathcal{M}} \text{false} \)
- \( \lfloor \text{if}(H : h)\{\text{true}\} \text{ else}\{L : l\} \rfloor_L = \text{if}(\_\)\{\text{true}\} \text{ else}\{L : l\} \not\xrightarrow{\mathcal{M}} \)

**Theorem (Stability)**

*Consider an expression \( e_1 \) (which may use \( _\)) and a \( _\)-free expression \( e_2 \) such that \( e_1 \xrightarrow{\mathcal{M}}^* e_2 \). Then for every \( M \subseteq \text{Marker} \) such that \( \lfloor e_2 \rfloor_M = e_2 \), it follows that \( \lfloor e_1 \rfloor_M \xrightarrow{\mathcal{M}}^* \lfloor e_2 \rfloor_M \).

- This means that if \( e_2 \) is not marked by \( m \), we can safely erase it from \( e_1 \).
Our information flow analysis for SLamJS comprises two phases:

1. We perform CFA to determine which functions and code values can be bound where.
2. We generate and solve information flow constraints using the results of the CFA.

- Handling code values in CFA requires some special treatment.
- The key observation in CFA is that data and control flow influence each other, so both must be handled in a single analysis. As information flow does not affect data or control flow, it can be separate.

- Because our analysis extends CFA, we believe our technique could easily be adapted to other CFA-style analyses.
CFA for SLamJS

0CFA is a standard analysis that operates by:

1. **labelling** each subexpression of a program;
2. generating **constraints** between the values occurring at each label (and each variable);
3. **solving** these constraints.

0CFA conflates variables with the same name bound in different functions.

- For most languages, this is not a problem, as we can simply $\alpha$-convert them.
- SLamJS does not respect $\alpha$-equivalence, so the analysis must track explicitly where names are bound.
0CFA can be derived from abstract interpretation over a suitable domain. Our abstract domain is:

Abstract values \( \nu \in \text{AbsVal} \) ::=

- NULL
- UNDEF
- BOOL
- NUM
- STR
- FUN(\( x, e \))
- BOX(\( e \))
- REC(\( \ell \))

The abstract value BOX(\( e \)) is inhabited by:

- the expression \( \text{box } e \);
- any expression that \( \text{box } e \) evaluates to.

The range of code values in a program may be infinite. This permissive definition of BOX(\( e \)) ensures that a finite solution to the constraints is always possible.
Sample CFA rules

\[ \Gamma, \varrho \models k^\ell \quad \text{if} \quad \lfloor k \rfloor \in \Gamma(\ell) \]

\[ \Gamma, \varrho \models x^\ell \quad \text{if} \quad \varrho(x) \subseteq \Gamma(\ell) \]

\[ \Gamma, \varrho \models (\text{box } e)^\ell \quad \text{if} \quad \Gamma, \varrho \models e \]

\[ \quad \text{and} \quad \exists \nu \in \Gamma(\ell). \Gamma, \varrho \models \nu \approx \text{box } e \]

\[ \Gamma, \varrho \models (\text{unbox } e)^\ell \quad \text{if} \quad \Gamma, \varrho \models e \]

\[ \quad \text{and} \quad \forall \text{BOX}(e') \in \Gamma(lbl(e)). \Gamma(lbl(e')) \subseteq \Gamma(\ell) \]

\[ \Gamma, \varrho \models (\text{if}(e_1\{e_2\} \text{ else } \{e_3\}))^\ell \quad \text{if} \quad \Gamma, \varrho \models e_1 \land \Gamma, \varrho \models e_2 \land \Gamma, \varrho \models e_3 \]

\[ \quad \text{and} \quad \Gamma(lbl(e_2)) \subseteq \Gamma(\ell) \land \Gamma(lbl(e_3)) \subseteq \Gamma(\ell) \]
CFA Example

Consider:

$(((\text{fun}(x)\{I : (\text{fun}(y)\{x\})\})(H : 1))(L : 2), \epsilon) \xrightarrow{\Pi}^* (I : (H : 1))$

labelled as:

$(((\text{fun}(x)\{(I : (\text{fun}(y)\{x^0\})^1)^2\})(H : 1^4)^5 (L : 2^7)^8)^9$

Solution of the CFA constraints gives:

$0 \mapsto \{\text{NUM}\} \quad 1 \mapsto \{\text{FUN}(y, (x)^0)\} \quad 2 \mapsto \{\text{FUN}(y, (x)^0)\}$
$3 \mapsto \{\text{FUN}(x, ((I : (\text{fun}(y)\{(x)^0\})^1)^2))\} \quad 4 \mapsto \{\text{NUM}\} \quad 5 \mapsto \{\text{NUM}\}$
$6 \mapsto \{\text{FUN}(y, (x)^0)\} \quad 7 \mapsto \{\text{NUM}\} \quad 8 \mapsto \{\text{NUM}\} \quad 9 \mapsto \{\text{NUM}\}$

As expected, the result of evaluation (labelled 9) is a number.
Information Flow for SLamJS

The information flow analysis uses the results of CFA to generate constraints on two relations between markers, labelled program points and variables:

- $\rightarrow\rightarrow$ tracks direct flows;
- $\rightarrow\leftarrow$ tracks indirect flows.

If an expression marked by $m$ is used in computing an expression labelled $\ell$ then, taking $\rightarrow\rightarrow = \rightarrow\rightarrow \cup \rightarrow\leftarrow$, the analysis ensures $m \rightarrow\star \ell$.

**Theorem (Information Flow Soundness)**

Suppose $\rightarrow\rightarrow$ has been computed for $t^\ell$ by the information flow analysis. Then if $t^\ell \stackrel{\Pi}{\rightarrow}\star \nu^\ell'$, where $\nu$ is a stage-0 value composed only of markers and constants, and $M = \{m \in \text{Marker} \mid m \rightarrow\star \ell\}$, it follows that $[\nu]_M = \nu$.

The key parts of this theorem have been mechanised in Coq.
### Sample Information Flow Rules

<table>
<thead>
<tr>
<th>Expression e $\models_{IF} e$ holds:</th>
<th>Subexpressions if:</th>
<th>Direct and:</th>
<th>Indirect and:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k^\ell$</td>
<td>$-$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$x^\ell$</td>
<td>$-$</td>
<td>$x \rightsquigarrow \ell$</td>
<td>$-$</td>
</tr>
<tr>
<td>$(if(t_1^{\ell_1}){ t_2^{\ell_2} } else{ t_3^{\ell_3} })^{\ell_4}$</td>
<td>$\land_{i=1}^{3} \models_{IF} t_i^{\ell_i}$</td>
<td>$\ell_2 \rightsquigarrow \ell_4 \land \ell_3 \rightsquigarrow \ell_4$</td>
<td>$\ell_1 \not\leftrightarrow \ell_4$</td>
</tr>
<tr>
<td>$(box \ t^{\ell_1})^{\ell_2}$</td>
<td>$\models_{IF} t^{\ell_1}$</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$(unbox \ t^{\ell_1})^{\ell_2}$</td>
<td>$\models_{IF} t^{\ell_1}$</td>
<td>$\forall BOX(t'^{\ell'}) \in \Gamma(\ell_1). \ell' \rightsquigarrow \ell_2$</td>
<td>$\ell_1 \not\leftrightarrow \ell_2$</td>
</tr>
</tbody>
</table>
Information Flow Analysis Example

Recall:

\[
(((\text{fun}(x)\{I : (\text{fun}(y)\{x\})\})(H : 1))(L : 2), \epsilon) \xrightarrow{\ast}(I : (H : 1))
\]

labelled as:

\[
(((\text{fun}(x)\{(I : (\text{fun}(y)\{x^0\}))^1\}^2))^3(H : 1^4)^5(L : 2^7)^8)^9
\]

The information flow constraints are:

\[
\begin{align*}
4 & \rightsquigarrow 5 & \rightsquigarrow x & \rightsquigarrow 0 & \downarrow \\
H & \nearrow I & \nearrow 3 & \nearrow 6 & \nearrow 9 & \nearrow L \\
1 & \rightsquigarrow 2 & \nearrow
\end{align*}
\]

We have \(H \rightsquigarrow^* 9\) and \(I \rightsquigarrow^* 9\) and \(L \not\rightsquigarrow^* 9\). This means the result (labelled 9) has information flows from \(H\) and \(I\), but not \(L\).
Proof Outline

First we prove the coherence of both stages of the analysis with evaluation:

**Theorem (CFA Coherence)**

\[ \text{If } \Gamma, \varrho \models e \text{ and } e \xrightarrow{n} e', \text{ then } \Gamma, \varrho \models e'. \]

**Theorem (Information Flow Coherence)**

\[ \text{If } \Gamma, \varrho, \varrho \models_{\text{IF}} e_1 \text{ and } e_1 \xrightarrow{m} e_2, \text{ then } \Gamma, \varrho, \varrho \models_{\text{IF}} e_2. \text{ Furthermore, } \text{lbl}(e_2) \xrightarrow{*} \text{lbl}(e_1). \]

We want to prove:

**Theorem (Information Flow Soundness)**

\[ \text{Suppose } \models \text{ has been computed for } t^\ell \text{ by the information flow analysis. Then if } t^\ell \xrightarrow{\Pi^*} v^\ell', \text{ where } v \text{ is a stage-0 value composed only of markers and constants, and } M = \{ m \in \text{Marker} \mid m \xrightarrow{*} \ell \}, \text{ it follows that } \lfloor v \rfloor_M = v. \]

Consider \( \Gamma, \varrho, \varrho \models_{\text{IF}} t^\ell \) with \( t^\ell \xrightarrow{\Pi^*} v^\ell' \). Using Information Flow Coherence, we can show \( \Gamma, \varrho, \varrho \models_{\text{IF}} v^\ell' \) with \( \ell' \xrightarrow{*} \ell \).
Proof Outline — Continued

So: $Γ$, $ϱ$, $|$−$|$→ $t \ell$, and $t \ell \smallfrown \ast \nu \ell'$ with $Γ$, $ϱ$, $|$−$|$→ $v$ and $\ell' \sim \ast \ell$. Let $M = \{ m \in Marker \mid m \sim \ast \ell \}$. 

The definition of the erasure $[v]_M$ means that if:

$▷$ for every marker $m$ that occurs in $v$, we have $m \in M$

then $[v]_M = v$. So try to prove this.

$v$ is a value composed only of markers and constants, so the information flow analysis rules enforce that:

$▷$ for every marker $m$ that occurs in $v$, $m \sim \ast \ell'$.

Then we have $m \sim \ast \ell' \sim \ast \ell$, so $m \sim \ast \ell$. Hence $m \in M$. This gives the main theorem.

To apply Information Flow Soundness we must usually invoke Stability.
We have implemented our analysis in OCaml.

Example

```ocaml
let c = box x in
let x = L : 1 in
let eval = fun(b){run b} in
let x = H : 2 in
eval(c)
```

Depends on: L
We have implemented our analysis in OCaml.

Example

```ocaml
let c = box x in
let x = L : 1 in
let eval = fun(b){run b} in
let x = H : 2 in
eval(c)
```

Depends on: L

```ocaml
let x = if(true){box f} else{box g} in
let f = fun(y){1} in
let g = fun(z){L : true} in
run (box ((unbox x)(H : undef)))
```

Depends on: L
Future Work

- Extend the analysis to handle other JavaScript features, such as mutable state and exceptions.
- Improve the precision of analysis of object reads and writes by extending the abstract string domain.
- Transfer our ideas to a CFA2 analysis for improved precision with higher order flow.
- Apply recent work on analysing `eval` directly to transform uses of `eval` into staged metaprogramming.
Conclusion

Our contributions:

▶ We have developed an information flow analysis for a JavaScript-like language with staged metaprogramming.
▶ We have mechanised the proof of soundness for our analysis using Coq.
▶ We have implemented our analysis in OCaml.
▶ Online material: http://mjolnir.cs.ox.ac.uk/web/slamjs/.

We believe that we now have all the technical tools for an interesting information flow analysis of JavaScript with `eval`. 
Conclusion

Our contributions:

▶ We have developed an information flow analysis for a JavaScript-like language with staged metaprogramming.
▶ We have mechanised the proof of soundness for our analysis using Coq.
▶ We have implemented our analysis in OCaml.
▶ Online material: http://mjolnir.cs.ox.ac.uk/web/slamjs/.

We believe that we now have all the technical tools for an interesting information flow analysis of JavaScript with `eval`.

▶ Thanks for listening. Questions are welcome.