2013 IEEE 26th Computer Security Foundations Symposium

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

Martin Lester
C.-H. Luke Ong
Department of Computer Science
University of Oxford
Oxford, UK
{*martin.lester, luke.ong*}@*cs.ox.ac.uk*

Max Schäfer
School of Computer Engineering
Nanyang Technological University
Singapore
schaefer@ntu.edu.sg

***Abstract*—Web applications written in JavaScript are regularly used for dealing with sensitive or personal data. Consequently, reasoning about their security properties has become an important problem, which is made very difficult by the highly dynamic nature of the language, particularly its support for runtime code generation. As a first step towards dealing with this, we propose to investigate security analyses for languages with more principled forms of dynamic code generation. To this end, we present a static information flow analysis for a dynamically typed functional language with prototype-based inheritance and staged metaprogramming. We prove its soundness, implement it and test it on various examples designed to show its relevance to proving security properties, such as noninterference, in JavaScript. To our knowledge, this is the first fully static information flow analysis for a language with staged metaprogramming, and the first formal soundness proof of a CFA-based information flow analysis for a functional programming language.**

***Keywords*-noninterference; staged metaprogramming; CFA; information flow; dynamically typed languages; JavaScript; static analysis**

I. Introduction

An *information flow* analysis determines which values in a program can influence which parts of the result of the program. Using an information flow analysis, we can, for instance, prove that program inputs that are deemed high security do not influence low security outputs; this important security property is known as *noninterference* [1].

Early work on noninterference focused mainly on applications in a military or government setting, where there might be strict rules about security clearance and classification of documents. More recently, there has been increased interest in information security (and hence its analysis) for Web applications, particularly for Web 2.0 applications written in JavaScript.

We have developed a static information flow analysis for a dynamically typed, pure, functional language with stage-based metaprogramming [2]; we call the language SLamJS (Staged Lambda JS) because it exhibits a number of JavaScript's interesting features in an idealised, lambda calculus-based setting [3]. The analysis is based on the idea of extending a constraint-based formulation of the analysis 0CFA [4] with constraints to track information flow. We believe that the idea could be extended to other CFA-style analyses (such as CFA2 [5]) for improved precision. We have formally proved the correctness of our analysis; we have also implemented it and tested it on a number of examples.

Supporting material, which includes mechanisations of our key results in the theorem prover Coq and an implementation of our analysis in OCaml, is available online at http://mjolnir.cs.ox.ac.uk/web/slamjs/.

The structure of the remainder of the paper is as follows. In Section II, we present SLamJS: we begin with an explanation of why we believe our chosen combination of language features is relevant to information security in Web applications. Next, we present the semantics of SLamJS and explain, using an augmented semantics, what information flow means in this language. Section III explains how the analysis works and how we proved its correctness. We discuss our implementation and some examples on which we have tested the analysis in Section IV. In Section V, we examine the gap between our work and a practical analysis for real-world Web applications. We also discuss other research on analysis of information flow and staged metaprogramming, before concluding in Section VI.

II. The Language SLamJS

A. Motivation

The new arena of Web applications presents many interesting challenges for information flow analysis. While there is an extensive body of research on information flow in statically typed languages [6], there is little tackling dynamically typed languages. The semantics of JavaScript are complex and poorly understood [7], which makes any formal analysis difficult. Web applications frequently comprise code from multiple sources (including libraries and adverts), written by multiple authors in an ad-hoc style. They are often interactive (so cannot be viewed as a single execution with inputs and outputs) and it might not be known in advance which code will be loaded.

© 2013, Martin Lester. Under license to IEEE.
DOI 10.1109/CSF.2013.21

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.

The **eval** construct of JavaScript, which allows execution of arbitrary code strings, is particularly troublesome, to the extent that many analyses just ignore it. However, a recent survey shows that real JavaScript code uses **eval** extensively [8]. Its uses vary widely from straightforward (loading data via JSON) through ill-informed (accessing fields of an object without using array notation) to subtle (changing scoping behaviour) and complex (emulating higher order functions). We think that it is important to develop techniques for analysing this notorious construct.

So that we might reasonably work formally, we have developed a simplified language called SLamJS. The language is heavily influenced by λ_{JS}, a "core calculus" for JavaScript [3]. Like JavaScript, SLamJS is dynamically typed and features first-class functions and objects with prototype-based inheritance. Like JavaScript, it allows code to be constructed, passed around and executed at runtime. Unlike JavaScript, this is achieved using Lisp-style code quotations rather than code strings [9]. Recent work indicates that real-world usage of **eval** is often of a form that could be expressed using code quotations [10]. Thus analysis of programs with executable code quotations is an important step towards analysis of programs with executable code strings.

B. *Syntax and Semantics of SLamJS*

1) Syntax: SLamJS is a functional language with atomic constants, records, branching, first-class functions and staged metaprogramming; the syntax is given in Fig. 1.

The language has five types of atomic constant: booleans, strings, numbers and two special values (**undef** and **null**) to indicate undefined or null values. A record $\{\overline{s:v}\}$ is a finite mapping from fields (named by strings) to values. Fields can be read ($e[e]$), updated or replaced ($e[e] = e$) and deleted (**del** $e[e]$). Records support prototype-based lookup: a read from an undefined field of a record is redirected to the corresponding field on the record held in its `"__proto__"` field, if there is one.

Branching on boolean values is enabled by the **if**$(e)\{e\}$ **else**$\{e\}$ construct. Functions can be defined (**fun**$(x)\{e\}$) and applied ($e(e)$).

Staged metaprogramming is supported through use of the **box**, **unbox** and **run** constructs in the style of Choi et al. [9]. **box** e_1 turns e_1 into a "quoted" or "boxed" code value, which can be executed using **run**. The use of **unbox** e_2 within a boxed expression e_1 forces evaluation of e_2 to a boxed value, which is spliced into e_1 *before it becomes a boxed value*.

Expressions of the form (e, ρ) and **run** e **in** ρ only arise as intermediate terms during execution: the former represents an explicit substitution [2], [11] where all free variables of the expression e are given their value by the environment ρ; the latter represents an expression to be unboxed and evaluated in environment ρ.

Values exist at all stages. Constants, records with constant fields and constant code quotations are values v^n at every stage n; closures are only values v^0 at stage zero. Other constructs may be values at higher stages (v^{n+1}, v^{n+2} for $n \geq 0$), provided that their subexpressions are values at the appropriate stage. We generally omit the stage superscript for values of stage zero (writing v instead of v^0).

2) Semantics: We give a small-step operational semantics with evaluation contexts and explicit substitutions for SLamJS. There are two reduction relations, $\overset{n}{\dashrightarrow}$ and $\xrightarrow{n}$, each annotated with a level n. The former is for top-level reduction, while the latter is for evaluation under a context.

Evaluation contexts In a staged setting, evaluation contexts may straddle stage boundaries, hence they are annotated with stage subscripts and superscripts. A context C_n^m denotes a hole at stage n inside an expression at stage m. For a context C_n^m and an expression e, we denote by $C_n^m\langle e\rangle$ the expression obtained by plugging e into the hole contained in C_n^m. The grammar of some key evaluation contexts is given in Fig. 2; full details are in Appendix A.

Reduction rules Top-level reduction rules fall into two categories: environment propagation rules for pushing explicit substitutions inwards (Fig. 3), and proper reduction rules (Fig. 4). The former are fairly straightforward, so full details are left for Appendix A. Note that explicit substitutions only apply at stage zero, hence (x, ρ) evaluates to x at level $n+1$ without looking up x in ρ. Furthermore, observe that (**run** e, ρ) pushes its environment into e, allowing boxed code values to capture variables from outside.

The proper reduction rules are also quite standard [9], except for the field access rules, which are designed to mimic JavaScript semantics as far as possible.

In particular, every record is expected to have a `"__proto__"` field, which holds either the value **null** or another record, giving rise to a chain of prototype objects that ultimately ends in **null**. Reading a record field follows this chain by rule (Read2), until the field is either found (Read1), or the top of the chain is reached, where (Read3) yields **undef**. Note that the reduction $\overset{0}{\dashrightarrow}$ can get stuck, for example, when applying a non-function, or branching on a non-boolean.

There is only a single rule for $\xrightarrow{m}$:

$$C_n^m\langle e\rangle \quad \xrightarrow{m} \quad C_n^m\langle e'\rangle \quad \text{if } e \overset{n}{\dashrightarrow} e'$$

We write $\xrightarrow{\text{⌑}}$ for the union over all m of $\xrightarrow{m}$, and $\xrightarrow{\text{⌑}}{}^{*}$ for its reflexive, transitive closure.

Example 1: Here is an evaluation trace of a simple **if** statement. We use ϵ to stand for the empty environment.

$$\begin{array}{ll} & (\textbf{if}(\textbf{true})\{\textbf{false}\}\ \textbf{else}\{1\}, \epsilon) \\ \xrightarrow{0} & \textbf{if}((\textbf{true}, \epsilon))\{(\textbf{false}, \epsilon)\}\ \textbf{else}\{(1, \epsilon)\} \\ \xrightarrow{0} & \textbf{if}(\textbf{true})\{(\textbf{false}, \epsilon)\}\ \textbf{else}\{(1, \epsilon)\} \\ \xrightarrow{0} & (\textbf{false}, \epsilon) \xrightarrow{0} \textbf{false} \end{array}$$

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.

$$
\begin{array}{llcl}
\text{Booleans} & b & ::= & \textbf{true} \mid \textbf{false} \\
\text{Strings} & s & \in & \textit{String} \\
\text{Numbers} & n & \in & \textit{Number} \\
\text{Names} & x & \in & \textit{Name} \\
\text{Constants} & k & ::= & \textbf{undef} \mid \textbf{null} \mid b \mid s \mid n \\
\text{Expressions} & e & ::= & k \mid \{\overline{s:e}\} \mid x \mid \textbf{fun}(x)\{e\} \mid e(e) \mid \textbf{box}\ e \mid \textbf{unbox}\ e \mid \textbf{run}\ e \\
 & & \mid & \textbf{if}(e)\{e\}\,\textbf{else}\{e\} \mid e[e] \mid e[e] = e \mid \textbf{del}\ e[e] \mid (e,\rho) \mid \textbf{run}\ e\ \textbf{in}\ \rho \\
\text{Values}\ldots & v, v^0 & ::= & (\textbf{fun}(x)\{e\}, \rho) \qquad \ldots\text{at stage 0 only} \\
\ldots\text{at any stage} & v^n & ::= & k \mid \{\overline{s:v^n}\} \mid (\textbf{box}\ v^{n+1}) \\
\ldots\text{at higher} & v^{n+1} & ::= & x \mid (\textbf{fun}(x)\{v^{n+1}\}) \mid (v^{n+1}(v^{n+1})) \mid (\textbf{run}\ v^{n+1}) \\
\quad\text{stages only} & & \mid & (\textbf{if}(v^{n+1})\{v^{n+1}\}\,\textbf{else}\{v^{n+1}\}) \\
 & & \mid & (v^{n+1}[v^{n+1}]) \mid (v^{n+1}[v^{n+1}] = v^{n+1}) \mid (\textbf{del}\ v^{n+1}[v^{n+1}]) \\
 & v^{n+2} & ::= & (\textbf{unbox}\ v^{n+1}) \\
\text{Environments} & \rho & \in & \textit{Name} \xrightarrow{\text{fin}} v^0
\end{array}
$$

Figure 1. Syntax of SLamJS

$$
\begin{array}{lllll}
C_n^m & ::= & [\,] & \in & \mathcal{C}_n^n \\
 & \mid & (\textbf{fun}(x)\{C_n^{m+1}\}) & \in & \mathcal{C}_n^{m+1} \\
 & \mid & (C_n^m(e)) & \in & \mathcal{C}_n^m \\
 & \mid & (v^m(C_n^m)) & \in & \mathcal{C}_n^m \\
 & \mid & (\textbf{unbox}\ C_n^m) & \in & \mathcal{C}_n^{m+1} \\
 & \mid & (\textbf{run}\ C_n^m\ \textbf{in}\ \rho) & \in & \mathcal{C}_n^m
\end{array}
\qquad
\begin{array}{llll}
\mid & (\textbf{if}(C_n^m)\{e\}\,\textbf{else}\{e\}) & \in & \mathcal{C}_n^m \\
\mid & (\textbf{if}(v^{m+1})\{C_n^{m+1}\}\,\textbf{else}\{e\}) & \in & \mathcal{C}_n^{m+1} \\
\mid & (\textbf{if}(v^{m+1})\{v^{m+1}\}\,\textbf{else}\{C_n^{m+1}\}) & \in & \mathcal{C}_n^{m+1} \\
\mid & (\textbf{box}\ C_n^{m+1}) & \in & \mathcal{C}_n^m \\
\mid & (\textbf{run}\ C_n^m) & \in & \mathcal{C}_n^m
\end{array}
$$

Figure 2. Selected evaluation contexts

$$
\begin{array}{rcl}
(k,\rho) & \overset{n}{\dashrightarrow} & k \\
(\textbf{fun}(x)\{e\},\rho) & \overset{n+1}{\dashrightarrow} & (\textbf{fun}(x)\{(e,\rho)\}) \\
(\textbf{box}\ e,\rho) & \overset{n}{\dashrightarrow} & (\textbf{box}\ (e,\rho)) \\
(\textbf{run}\ e,\rho) & \overset{0}{\dashrightarrow} & (\textbf{run}\ (e,\rho)\ \textbf{in}\ \rho)
\end{array}
\qquad
\begin{array}{rcl}
(x,\rho) & \overset{n+1}{\dashrightarrow} & x \\
(e_1(e_2),\rho) & \overset{n}{\dashrightarrow} & ((e_1,\rho)((e_2,\rho))) \\
(\textbf{unbox}\ e,\rho) & \overset{n}{\dashrightarrow} & (\textbf{unbox}\ (e,\rho)) \\
(\textbf{run}\ e,\rho) & \overset{n+1}{\dashrightarrow} & (\textbf{run}\ (e,\rho))
\end{array}
$$

$$
(\textbf{if}(e_1)\{e_2\}\,\textbf{else}\{e_3\},\rho) \overset{n}{\dashrightarrow} (\textbf{if}((e_1,\rho))\{(e_2,\rho)\}\,\textbf{else}\{(e_3,\rho)\})
$$

Figure 3. Selected environment propagation rules

$$
\begin{array}{lrcll}
(\textsc{Lookup}) & (x,\rho) & \overset{0}{\dashrightarrow} & \rho(x) & \\
(\textsc{Apply}) & ((\textbf{fun}(x)\{e\},\rho)(v)) & \overset{0}{\dashrightarrow} & (e,\rho[x\mapsto v]) & \\
(\textsc{Unbox}) & (\textbf{unbox}\ (\textbf{box}\ v^1)) & \overset{1}{\dashrightarrow} & (v^1) & \\
(\textsc{Run}) & (\textbf{run}\ (\textbf{box}\ v^1)\ \textbf{in}\ \rho) & \overset{0}{\dashrightarrow} & (v^1,\rho) & \\
(\textsc{IfTrue}) & (\textbf{if}(\textbf{true})\{e_1\}\,\textbf{else}\{e_2\}) & \overset{0}{\dashrightarrow} & e_1 & \\
(\textsc{IfFalse}) & (\textbf{if}(\textbf{false})\{e_1\}\,\textbf{else}\{e_2\}) & \overset{0}{\dashrightarrow} & e_2 & \\
(\textsc{Read1}) & (\{\overline{s:v}, s_i : v_i, \overline{s:v}'\}[s_i]) & \overset{0}{\dashrightarrow} & v_i & \\
(\textsc{Read2}) & (\{\overline{s:v}, \texttt{"\_proto\_"} : \{\overline{s:v'}\}, \overline{s:v''}\}[s_x]) & \overset{0}{\dashrightarrow} & (\{\overline{s:v'}\}[s_x]) & \text{if } s_x \notin \overline{s} \cup \overline{s}'' \\
(\textsc{Read3}) & (\{\overline{s:v}, \texttt{"\_proto\_"} : \textbf{null}, \overline{s:v''}\}[s_x]) & \overset{0}{\dashrightarrow} & \textbf{undef} & \text{if } s_x \notin \overline{s} \cup \overline{s}'' \\
(\textsc{Write1}) & (\{\overline{s:v}, s_i : v_i, \overline{s:v}'\}[s_i] = v_i') & \overset{0}{\dashrightarrow} & \{\overline{s:v}, s_i : v_i', \overline{s:v}'\} & \\
(\textsc{Write2}) & (\{\overline{s:v}\}[s_x] = v_x) & \overset{0}{\dashrightarrow} & \{\overline{s:v}, s_x : v_x\} & \text{if } s_x \notin \overline{s} \\
(\textsc{Del1}) & (\textbf{del}\ \{\overline{s:v}, s_i : v_i, \overline{s:v}'\}[s_i]) & \overset{0}{\dashrightarrow} & \{\overline{s:v}, \overline{s:v}'\} & \\
(\textsc{Del2}) & (\textbf{del}\ \{\overline{s:v}\}[s_x]) & \overset{0}{\dashrightarrow} & \{\overline{s:v}\} & \text{if } s_x \notin \overline{s}
\end{array}
$$

Figure 4. Proper reduction rules

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.

$$
\begin{array}{lrcl}
(\textsc{Lift-App}) & ((\mathfrak{m} : e), \rho)(v) & \overset{0}{\dashrightarrow} & (\mathfrak{m} : ((e, \rho)(v))) \\
(\textsc{Lift-If}) & (\textbf{if}(\mathfrak{m} : v)\{e_1\}\,\textbf{else}\{e_2\}) & \overset{0}{\dashrightarrow} & (\mathfrak{m} : (\textbf{if}(v)\{e_1\}\,\textbf{else}\{e_2\})) \\
(\textsc{Lift-Unbox}) & \textbf{unbox}\,(\mathfrak{m} : v) & \overset{1}{\dashrightarrow} & (\mathfrak{m} : (\textbf{unbox}\ v)) \\
(\textsc{Lift-RunIn}) & \textbf{run}\,(\mathfrak{m} : v)\,\textbf{in}\ \rho & \overset{0}{\dashrightarrow} & (\mathfrak{m} : (\textbf{run}\ v\,\textbf{in}\ \rho)) \\
(\textsc{Lift-ReadSel}) & (v_1[\mathfrak{m} : v_2]) & \overset{0}{\dashrightarrow} & (\mathfrak{m} : (v_1[v_2])) \\
(\textsc{Lift-ReadRec}) & ((\mathfrak{m} : v_1)[v_2]) & \overset{0}{\dashrightarrow} & (\mathfrak{m} : (v_1[v_2])) \\
(\textsc{Lift-WriteSel}) & (v_1[\mathfrak{m} : v_2] = v_3) & \overset{0}{\dashrightarrow} & (\mathfrak{m} : (v_1[v_2] = v_3)) \\
(\textsc{Lift-WriteRec}) & ((\mathfrak{m} : v_1)[v_2] = v_3) & \overset{0}{\dashrightarrow} & (\mathfrak{m} : (v_1[v_2] = v_3)) \\
(\textsc{Lift-DelSel}) & (\textbf{del}\ v_1[\mathfrak{m} : v_2]) & \overset{0}{\dashrightarrow} & (\mathfrak{m} : (\textbf{del}\ v_1[v_2])) \\
(\textsc{Lift-DelRec}) & (\textbf{del}\,(\mathfrak{m} : v_1)[v_2]) & \overset{0}{\dashrightarrow} & (\mathfrak{m} : (\textbf{del}\ v_1[v_2]))
\end{array}
$$

Figure 5. Semantic rules for lifts

Example 2: The staging constructs in SLamJS allow fragments of code to be treated as values and spliced together or evaluated at run-time, as shown in this evaluation trace.

$$
\begin{array}{l}
\quad(\textbf{run}\,(\textbf{box}\,(\textbf{if}(\textbf{unbox}\,(\textbf{box}\,(\textbf{true})))\{\textbf{false}\}\,\textbf{else}\{1\})), \epsilon) \\
\xrightarrow{0} \textbf{run}\,(\textbf{box}\,(\textbf{if}(\textbf{unbox}\,(\textbf{box}\,(\textbf{true})))\{\textbf{false}\} \\
\qquad\qquad \textbf{else}\{1\}), \epsilon)\,\textbf{in}\,\epsilon \\
\xrightarrow{0}{}^{*} \textbf{run}\,(\textbf{box}\,(\textbf{if}(\textbf{unbox}\,(\textbf{box}\,(\textbf{true})))\{(\textbf{false}, \epsilon)\} \\
\qquad\qquad \textbf{else}\{(1, \epsilon)\}))\ \textbf{in}\ \epsilon \\
\xrightarrow{0} \textbf{run}\,(\textbf{box}\,(\textbf{if}(\textbf{true})\{(\textbf{false}, \epsilon)\}\,\textbf{else}\{(1, \epsilon)\}))\,\textbf{in}\,\epsilon \\
\xrightarrow{0}{}^{*} \textbf{run}\,(\textbf{box}\,(\textbf{if}(\textbf{true})\{\textbf{false}\}\,\textbf{else}\{1\}))\,\textbf{in}\,\epsilon \\
\xrightarrow{0} (\textbf{if}(\textbf{true})\{\textbf{false}\}\,\textbf{else}\{1\}, \epsilon) \\
\xrightarrow{0} \textbf{if}(\textbf{true}, \epsilon)\{(\textbf{false}, \epsilon)\}\,\textbf{else}\{(1, \epsilon)\} \\
\xrightarrow{0} \textbf{if}(\textbf{true})\{(\textbf{false}, \epsilon)\}\,\textbf{else}\{(1, \epsilon)\} \xrightarrow{0} (\textbf{false}, \epsilon) \xrightarrow{0} \textbf{false}
\end{array}
$$

Example 3: Our staging constructs allow variables to be captured by code values originating outside their scope. Here, the code value **box** y is outside the scope of y, but captures it during evaluation.

$$
\begin{array}{l}
\quad(((\textbf{fun}(x)\{(\textbf{fun}(y)\{\textbf{run}\ x\})\})(\textbf{box}\ y))(\textbf{true}), \epsilon) \\
\xrightarrow{0}{}^{*} (\textbf{fun}(y)\{\textbf{run}\ x\}, \langle x \mapsto \textbf{box}\ y\rangle)(\textbf{true}) \\
\xrightarrow{0} (\textbf{run}\ x, \langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\ y\rangle) \\
\xrightarrow{0} \textbf{run}\,(x, \langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\ y\rangle)\,\textbf{in}\,\langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\ y\rangle \\
\xrightarrow{0} \textbf{run}\,(\textbf{box}\ y)\,\textbf{in}\,\langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\ y\rangle \\
\xrightarrow{0} (y, \langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\ y\rangle) \\
\xrightarrow{0} \textbf{true}
\end{array}
$$

This useful feature is vital for modelling certain uses of **eval**; the above code corresponds to this JavaScript:

```
((function (x) {return function (y) {
    return (eval(x));}})("y"))(true);
```

However, the power comes at a price: the usual alpha equivalence property of λ-calculus does not hold in SLamJS [2], which makes reasoning about programs harder.

C. Augmented Semantics of SLamJS

The result of a program can depend on its component values in essentially two different ways. Consider programs operating on two variables l and h. The program $(\textbf{if}(l)\{h\}\,\textbf{else}\{1\})$ may evaluate to the value of h (if l is **true**); we say that there is a *direct flow* from h to the result. Conversely, the program $(\textbf{if}(h)\{\textbf{true}\}\,\textbf{else}\{1\})$ cannot evaluate to h. However, the result of evaluation tells us whether h was **true** or **false** because h influences the control flow of the program; there is an *indirect flow* from h to the result of the program.

In order to track the dependency of a result on its component subexpressions, we augment the language with explicit *dependency markers* [12], [13]. We also introduce new rules for lifting markers into their parent expressions to avoid losing information about dependencies. The augmented semantics is not intended for use in the execution of programs; rather, we use it for analysing and reasoning about dependencies in the original language. We begin by adding markers to the syntax:

$$
\begin{array}{llll}
\text{Markers} & \mathfrak{m} & \in & \textit{Marker} \\
\text{Expressions} & e & ::= & \ldots \mid (\mathfrak{m} : e) \\
\text{Values} & v^n & ::= & \ldots \mid (\mathfrak{m} : v^n)
\end{array}
$$

We extend contexts to allow evaluation within a marked expression:

$$
C_n^m \quad ::= \quad \ldots \quad \mid \quad (\mathfrak{m} : C_n^m) \quad \in \quad \mathcal{C}_n^m
$$

We allow propagation of environments within marked expressions:

$$
(\mathfrak{m} : e, \rho) \quad \overset{n}{\dashrightarrow} \quad (\mathfrak{m} : (e, \rho))
$$

In Fig. 5 we introduce lifts to maintain a record of indirect flows. Note there is no need for a lift rule on the right of an assignment (i.e., $v_1[v_2] = (\mathfrak{m} : v_3) \overset{0}{\dashrightarrow} (\mathfrak{m} : v_1[v_2] = v_3)$), since the flow from v_3 is direct.

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.

Example 4: Recall Example 1. Suppose we add markers to each of the components of the **if**. The evaluation trace now becomes:

$$\begin{array}{ll} & (\textbf{if}((\textsc{h} : \textbf{true}))\{(\textsc{l} : \textbf{false})\}\,\textbf{else}\{(\textsc{i} : 1)\}, \epsilon) \\ \xrightarrow{0}{}^{*} & \textbf{if}((\textsc{h} : \textbf{true}))\{((\textsc{l} : \textbf{false}), \epsilon)\}\,\textbf{else}\{((\textsc{i} : 1), \epsilon)\} \\ \xrightarrow{0} & (\textsc{h} : (\textbf{if}(\textbf{true})\{((\textsc{l} : \textbf{false}), \epsilon)\}\,\textbf{else}\{((\textsc{i} : 1), \epsilon)\})) \\ \xrightarrow{0} & (\textsc{h} : ((\textsc{l} : \textbf{false}), \epsilon)) \\ \xrightarrow{0}{}^{*} & (\textsc{h} : (\textsc{l} : \textbf{false})) \end{array}$$

Note how the markers H and L in the result indicate that it depends on the marked values $(\textsc{h} : \textbf{true})$ and $(\textsc{l} : \textbf{false})$.

Example 5: Here is an example of marked evaluation with functions:

$$(((\textbf{fun}(x)\{\textsc{i} : (\textbf{fun}(y)\{x\})\})(\textsc{h} : 1))(\textsc{l} : 2), \epsilon) \xrightarrow{0}{}^{*} (\textsc{i} : (\textsc{h} : 1))$$

Observe that the result depends on I because the function $(\textsc{i} : (\textbf{fun}(y)\{x\}))$ was used to compute it, but not on L, as $(\textsc{l} : 2)$ is discarded by that function.

Simulation: Consider a function *unmark*, defined in the obvious way, which strips an expression of all markers. Clearly if $\mathit{unmark}(e_1) = f_1 \xrightarrow{n} f_2$, then for some e_2 such that $e_1 \xrightarrow{n}{}^{*} e_2$, we have $\mathit{unmark}(e_2) = f_2$.

III. Information Flow Analysis for SLamJS

A. Overview

Before we can define an information flow analysis, we need to define what information flow is. Following Pottier and Conchon [12], we use the idea that if information does not flow from a marked expression into a value resulting from evaluation, then erasing that marked expression or replacing it with a dummy value should not affect the result of evaluation. (We use only their proof technique; their type-based analysis is not applicable to our language.) We begin in Section III-B by defining erasure and establishing some results about its behaviour.

Our information flow analysis is built on top of a 0CFA-style analysis capable of handling our staging constructs. Two variants of such an analysis are explained in Section III-C; mechanised correctness proofs in Coq are available online.

In Section III-D, we present the information flow analysis itself. A key idea in CFA is that control flow influences data flow and vice versa. Information flow depends on control and data flow, but the reverse is not true. Therefore it is possible to treat information flow analysis as an addition to CFA, rather than a completely new combined analysis. We have two versions of the CFA, each of which yields an information flow analysis. We sketch a correctness proof of the simpler analysis; complete mechanised proofs of both are available online.

Finally, in Section III-E, we prove soundness of the information flow analysis. We also discuss its relationship with noninterference.

B. Erasure and Stability

1) Erasure and Prefixes: We extend the language with a "hole" that behaves like an unbound variable:

$$\begin{array}{llll} \text{Expressions} & e & ::= & \ldots \mid \_ \\ \text{Values} & v^n & ::= & \ldots \mid \_ \end{array}$$

$$(\_, \rho) \overset{n}{\dashrightarrow} \_$$

Now for $M \subseteq \mathit{Marker}$, define the M*-erasure* of e, written $\lfloor e \rfloor_M$, to be: e with any subexpression $(\mathfrak{m} : e')$ where $\mathfrak{m} \notin M$ replaced by _. A full definition is in Appendix A.

2) Prefixing and Monotonicity: We say that e_1 is a *prefix* of e_2 or write $e_1 \preccurlyeq e_2$ if replacing some subexpressions of e_2 with _ gives e_1.

Evaluation is monotonic with respect to prefixing: if $e_1 \preccurlyeq e_2$ and $e_1 \xrightarrow{⌑}{}^{*} f$, where f contains no _, then $e_2 \xrightarrow{⌑}{}^{*} f$.

Lemma 1 (Step Stability): If $e_1 \overset{n}{\dashrightarrow} e_2$, then either $\lfloor e_1 \rfloor_M \overset{n}{\dashrightarrow} \lfloor e_2 \rfloor_M$ or the reduction rule applied to derive this is a lift (Lift-*) of a marker $\mathfrak{m} \notin M$.

Proof: By induction over the rules defining $\overset{n}{\dashrightarrow}$. ■

Theorem 1 (Stability): Consider an expression e_1 (which may use _) and a _-free expression e_2 such that $e_1 \xrightarrow{⌑}{}^{*} e_2$. Then for every $M \subseteq \mathit{Marker}$ such that $\lfloor e_2 \rfloor_M = e_2$, it follows that $\lfloor e_1 \rfloor_M \xrightarrow{⌑}{}^{*} \lfloor e_2 \rfloor_M$.

Proof: Consider any e_2 and M with $\lfloor e_2 \rfloor_M = e_2$. Aim to prove, for any e_1 with $e_1 \xrightarrow{⌑}{}^{*} e_2$, that $\lfloor e_1 \rfloor_M \xrightarrow{⌑}{}^{*} e_2$. Argue by induction over the length k of derivations of $e_1 \xrightarrow{⌑}{}^{*} e_2$.

Base case: $k = 0$. So $e_1 = e_2$. We have $\lfloor e_2 \rfloor_M = e_2$, so trivially $\lfloor e_1 \rfloor_M = e_2$.

Inductive step: $k = k' + 1$. Given $e_1 \xrightarrow{n} e \xrightarrow{⌑}{}^{k'} e_2$, aim to prove $\lfloor e_1 \rfloor_M \xrightarrow{⌑}{}^{*} e_2$. Assume by the induction hypothesis that $\lfloor e \rfloor_M \xrightarrow{⌑}{}^{k'} e_2$. Let $e_1 = C_n^m\langle f_1 \rangle$ and $e = C_n^m\langle f \rangle$ with $f_1 \overset{n}{\dashrightarrow} f$. Case split on if $f_1 \overset{n}{\dashrightarrow} f$ is a lift of a marker $\mathfrak{m} \notin M$.

If it is such a lift, then let $f = (\mathfrak{m} : f')$. Now $\lfloor f \rfloor_M = \_$, so $\lfloor f \rfloor_M \preccurlyeq \lfloor f_1 \rfloor_M$. Thus $\lfloor C_n^m\langle f \rangle \rfloor_M \preccurlyeq \lfloor C_n^m\langle f_1 \rangle \rfloor_M$; that is, $\lfloor e \rfloor_M \preccurlyeq \lfloor e_1 \rfloor_M$. We already have (from the induction hypothesis) that $\lfloor e \rfloor_M \xrightarrow{⌑}{}^{k'} e_2$. Now, applying Monotonicity, we get $\lfloor e_1 \rfloor_M \xrightarrow{⌑}{}^{*} e_2$.

Otherwise, apply the Step Stability Lemma to get $\lfloor f_1 \rfloor_M \overset{n}{\dashrightarrow} \lfloor f \rfloor_M$. It follows that $\lfloor C_n^m\langle f_1 \rangle \rfloor_M \xrightarrow{n} \lfloor C_n^m\langle f \rangle \rfloor_M$; that is, $\lfloor e_1 \rfloor_M \xrightarrow{n} \lfloor e \rfloor_M$. Using the induction hypothesis gives $\lfloor e_1 \rfloor_M \xrightarrow{n} \lfloor e \rfloor_M \xrightarrow{⌑}{}^{k'} e_2$, as required. ■

Example 6: Recall that in Example 5, the result depended on H and I, but not L. Applying $\lfloor - \rfloor_{\{\mathrm{H,I}\}}$ and evaluating the initial expression gives:

$$(((\textbf{fun}(x)\{\textsc{i} : (\textbf{fun}(y)\{x\})\})(\textsc{h} : 1))(\_), \epsilon) \xrightarrow{0}{}^{*} (\textsc{i} : (\textsc{h} : 1))$$

That is, the result of evaluation is unchanged.

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.

C. 0CFA for SLamJS

We use a context-insensitive, flow-insensitive control flow analysis (0CFA [4]) to approximate statically the set of values to which individual expressions in a program may evaluate at runtime.

As far as 0CFA is concerned, the only non-standard feature of SLamJS is its staging constructs. Roughly speaking, **box** and **unbox**/**run** act like function abstraction and application, except that they use a dynamic (instead of static) scoping discipline.[1]

We present two variants of 0CFA for SLamJS: a simple, but somewhat imprecise formulation that does not distinguish like-named variables bound by different abstractions, and a more complicated one that does.

Simple Analysis: Following Nielson, Nielson and Hankin [14], we formalise our analysis by means of an *acceptability judgement* of the form $\Gamma, \varrho \models e$, where Γ is an *abstract cache* associating sets of abstract values with labelled program points, and ϱ is an *abstract environment* mapping local variables and record fields to sets of abstract values. Intuitively, the purpose of this judgement is to ensure that $\Gamma(\ell)$ soundly over-approximates all possible values to which the expression at program point ℓ can evaluate, and ϱ does the same for variables and record fields.

More precisely, we assume that all expressions in the program are labelled with labels drawn from a set *Label*. An abstract cache is a mapping $\mathit{Label} \to \mathcal{P}(\mathit{AbsVal})$ associating a set of abstract values with every program point; similarly, an abstract environment $\varrho\colon \mathit{AbsVar} \to \mathcal{P}(\mathit{AbsVal})$ maps abstract variables to sets of abstract values, where an abstract variable is either a simple name x (representing a function parameter), or a field name of the form $\ell.p$, where ℓ is a label representing a record, and p is the name of a field of that record.

Our domain of abstract values (Fig. 6) is mostly standard, with, e.g., an abstract value NULL to represent the concrete **null** value, an abstract NUM value representing any number, and abstract values $\mathtt{FUN}(x, e)$, $\mathtt{BOX}(e)$ and $\mathtt{REC}(\ell)$ representing, respectively, a function value, a quoted piece of code, and a record allocated at program point ℓ. For an abstract environment ϱ and a label ℓ we define $\mathtt{proto}(\ell)_\varrho$ to be the smallest set $P \subseteq \mathit{Label}$ such that $\ell \in P$ and for every $p \in P$ and $\mathtt{REC}(\ell') \in \varrho(p.\texttt{"\_proto\_"})$ also $\ell' \in P$.

The acceptability judgement is now defined using syntax-directed rules, some of which are shown in Fig. 7 (the remaining rules, which are standard, are given in Appendix A of the extended version of this paper [15]).

We write t^ℓ to represent an expression of the syntactic form t, labelled with ℓ. Thus, k^ℓ means an expression consisting of a literal k labelled ℓ, and the first rule simply says that in order for Γ and ϱ to constitute an acceptable analysis of k^ℓ, $\Gamma(\ell)$ must contain the abstract value $\lceil k \rceil$ representing k. Similarly, the second rule requires Γ and ϱ to be consistent in the abstract values they assign to variables and references to them. The rules for dealing with function abstractions and records are standard and so are elided here for brevity.

The rule for **box** e requires Γ and ϱ to be an acceptable analysis of the single sub-expression e, and for $\Gamma(\ell)$ to include an abstract value ν approximating **box** e, which is written as $\Gamma, \varrho \models \nu \approx$ **box** e. This judgement holds if $\nu = \mathtt{BOX}(e)$, but we must be slightly more flexible: during evaluation, unboxing may splice new code fragments into e, changing its syntactic shape to some new expression e'. In order for the flow analysis to be effectively computable, we want the set of abstract values to be finite, so we cannot expect every such $\mathtt{BOX}(e')$ to be part of our abstract domain. Instead, we close the approximation judgement under reduction, that is, if $\Gamma, \varrho \models \nu \approx t$ and $t^\ell \xrightarrow{n} t'^{\ell'}$, then also $\Gamma, \varrho \models \nu \approx t'$; the full definition of the approximation judgement appears in Fig. 8.

The rule for **unbox** e is surprisingly simple: all that is required is that for any abstract value $\mathtt{BOX}(e')$ that the analysis thinks can flow into $\mathit{lbl}(e)$ (i.e., the label of expression e) every abstract value flowing into its body e' also flows into the unboxing expression. Note that this models the name capture associated with dynamic scoping, since our abstract environment ϱ does not distinguish between different variables of the same name. The rule for **run** is the same as for **unbox**.

Finally, we show the rule for **if**, which is standard: any abstract value that either of the branches can evaluate to is also a possible result of the entire **if** expression.

To show this acceptability judgement makes sense, we prove its coherence with evaluation:

Theorem 2 (CFA Coherence): If $\Gamma, \varrho \models e$ and $e \xrightarrow{n} e'$, then $\Gamma, \varrho \models e'$.

The proof of this theorem is fairly technical and is elided here. A full formalisation in Coq is available online in our supporting material.

Owing to its syntax-directed nature, the definition of the acceptability relation can quite easily be recast as constraint rules; by generating and solving all constraints for a given program, an acceptable flow analysis can be derived.

Note that, while there may be infinitely many abstract values of the form $\mathtt{BOX}(e)$ and $\mathtt{FUN}(e)$ that are relevant to a particular program, the closure of the approximation judgement under reduction means that the analysis need only consider those corresponding to subexpressions e of the original program, not those that may arise during execution. That is, the analysis need only solve a finite set of constraints over a finite set of abstract values and a finite set of labels and abstract variables, so it can be guaranteed to terminate.

[1]This intuition is made more precise in Choi et al.'s work on static analysis of staged programs [9], where staging constructs are translated into function abstraction and application; we prefer to work directly on the staged language for simplicity.

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.

Abstract values	$\nu \in \mathit{AbsVal}$	$::=$	$\texttt{NULL} \mid \texttt{UNDEF} \mid \texttt{BOOL} \mid \texttt{NUM} \mid \texttt{STR}$
		$\mid$	$\texttt{FUN}(x, e) \mid \texttt{BOX}(e) \mid \texttt{REC}(\ell)$
Abstract variables	$\xi \in \mathit{AbsVar}$	$::=$	$x \mid \ell.p$
Abstract caches	Γ	$:$	$\mathit{Label} \to \mathcal{P}(\mathit{AbsVal})$
Abstract environments	ϱ	$:$	$\mathit{AbsVar} \to \mathcal{P}(\mathit{AbsVal})$

Figure 6. Abstract domains

$$\begin{array}{lll}
\Gamma, \varrho \models k^\ell & \text{if} & \lceil k \rceil \in \Gamma(\ell) \\
\Gamma, \varrho \models x^\ell & \text{if} & \varrho(x) \subseteq \Gamma(\ell) \\
\Gamma, \varrho \models (\textbf{box}\ e)^\ell & \text{if} & \Gamma, \varrho \models e \\
 & \text{and} & \exists \nu \in \Gamma(\ell). \Gamma, \varrho \models \nu \approx \textbf{box}\ e \\
\Gamma, \varrho \models (\textbf{unbox}\ e)^\ell & \text{if} & \Gamma, \varrho \models e \\
 & \text{and} & \forall \texttt{BOX}(e') \in \Gamma(\mathit{lbl}(e)). \Gamma(\mathit{lbl}(e')) \subseteq \Gamma(\ell) \\
\Gamma, \varrho \models (\textbf{if}(e_1)\{e_2\}\,\textbf{else}\{e_3\})^\ell & \text{if} & \Gamma, \varrho \models e_1 \wedge \Gamma, \varrho \models e_2 \wedge \Gamma, \varrho \models e_3 \\
 & \text{and} & \Gamma(\mathit{lbl}(e_2)) \subseteq \Gamma(\ell) \wedge \Gamma(\mathit{lbl}(e_3)) \subseteq \Gamma(\ell)
\end{array}$$

Figure 7. Some rules for the 0CFA acceptability judgement

$$\begin{array}{lll}
\Gamma, \varrho \models \lceil k \rceil \approx k & & \text{for any literal } k \\
\Gamma, \varrho \models \texttt{FUN}(x, e) \approx \textbf{fun}(x)\{e\} & & \\
\Gamma, \varrho \models \texttt{BOX}(e) \approx \textbf{box}\ e & & \\
\Gamma, \varrho \models \texttt{REC}(\ell') \approx \{\overline{s : t^\ell}\} & \text{if} & \forall i. \exists \nu_i \in \varrho(\ell'.s_i). \Gamma, \varrho \models \nu_i \approx t_i \\
\Gamma, \varrho \models \nu \approx t' & \text{if} & \Gamma, \varrho \models \nu \approx t \wedge t^\ell \xrightarrow{n} t'^\ell \\
\Gamma, \varrho \models \nu \approx (t, \rho) & \text{if} & \Gamma, \varrho \models \nu \approx t \wedge \Gamma, \varrho \models \rho
\end{array}$$

For a literal k, let $\lceil k \rceil$ be its abstract value:

$\lceil \textbf{null} \rceil$	$=$	$\texttt{NULL}$	
$\lceil \textbf{undef} \rceil$	$=$	$\texttt{UNDEF}$	
$\lceil b \rceil$	$=$	$\texttt{BOOL}$	for boolean b
$\lceil n \rceil$	$=$	$\texttt{NUM}$	for number n
$\lceil s \rceil$	$=$	$\texttt{STR}$	for string s

Figure 8. The approximation judgement $\Gamma, \varrho \models \nu \approx t$ and the abstract value operation $\lceil k \rceil$

Example 7: Recall again Example 5. Our implementation of the analysis labels the expression as follows:

$$(((\textbf{fun}(x)\{(\textsc{i} : (\textbf{fun}(y)\{x^0\})^1)^2\})^3(\textsc{h} : 1^4)^5)^6(\textsc{l} : 2^7)^8)^9$$

By generating and solving constraints it gives the following solution for Γ:

$$\begin{array}{llll}
0 \mapsto \{\texttt{NUM}\} & 1 \mapsto \{\texttt{FUN}(y, (x)^0)\} & & 2 \mapsto \{\texttt{FUN}(y, (x)^0)\} \\
\multicolumn{3}{l}{3 \mapsto \{\texttt{FUN}(x, ((\textsc{i} : (\textbf{fun}(y)\{(x)^0\})^1)^2)\}} & 4 \mapsto \{\texttt{NUM}\} \\
5 \mapsto \{\texttt{NUM}\} & 6 \mapsto \{\texttt{FUN}(y, (x)^0)\} & & 7 \mapsto \{\texttt{NUM}\} \\
8 \mapsto \{\texttt{NUM}\} & 9 \mapsto \{\texttt{NUM}\} & &
\end{array}$$

while $\varrho = \{x \mapsto \{\texttt{NUM}\}, y \mapsto \{\texttt{NUM}\}\}$. As expected, the result of evaluation (labelled 9) is a number.

Improved Analysis: The analysis presented so far is not very precise, since abstract environments do not distinguish identically named parameters of different functions. Ordinarily, this is not a problem, as one can rename them apart, but this is not possible for SLamJS, which does not enjoy alpha conversion.

To restore analysis precision in the absence of alpha conversion, we introduce an *abstract context* Ξ that keeps track of name bindings. In a single-staged language, such an abstract context would simply map a name x to the innermost enclosing function abstraction whose parameter is x. In a multi-staged setting, we need to distinguish between bindings at different stages, hence the abstract context maintains one such mapping per stage. Thus Ξ is a stack of frames, one for each stage; a frame maps each variable name to the label of its binding context.

For instance, the two uses of x in the SLamJS expression $\textbf{fun}(x)\{\textbf{box}(\textbf{fun}(x)\{(\textbf{unbox}\,x)(x)\})\}$ are at different stages, and hence bound by different abstractions: the first x by the outer abstraction, the second by the inner one.

The acceptability judgement for the improved analysis is now of the form $\Gamma, \varrho, \Xi \models e$, and the derivation rules include additional bookkeeping to adjust Ξ when analysing subexpressions at different stages. While conceptually simple, this change somewhat complicates the formalism, so we do not present it in detail here; a full formalisation is available in the supporting material.

D. Information Flow for SLamJS

Assume we have already analysed a program using 0CFA and found environments Γ, ϱ that over-approximate the values flowing to each labelled expression. We use information about which functions and boxed values may occur to assist in determining what direct and indirect flows occur between labels of the expression.

By recursing over the structure of an expression, we generate constraints on a relation $\rightsquigarrow$:

$$\rightsquigarrow\ :\ (\mathit{Label} \uplus \mathit{Name} \uplus \mathit{Marker}) \to (\mathit{Label} \uplus \mathit{Name} \uplus \mathit{Marker})$$

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.

Expression e $\models_{\mathrm{IF}} e$ holds:	*Subexpressions* if:	*Direct Flows* and:	*Indirect Flows* and:
k^{ℓ}	$-$	$-$	$-$
x^{ℓ}	$-$	$x \rightsquigarrow \ell$	$-$
$\textbf{fun}(x)\{t^{\ell_1}\}^{\ell_2}$	$\models_{\mathrm{IF}} t^{\ell_1}$	$-$	$-$
$(t_1^{\ell_1}(t_2^{\ell_2}))^{\ell}$	$\models_{\mathrm{IF}} t_1^{\ell_1} \wedge \models_{\mathrm{IF}} t_2^{\ell_2}$	$\forall \mathtt{FUN}(x, t_3^{\ell_3}) \in \Gamma(\ell_1) \,.\, \ell_2 \rightsquigarrow x \wedge \ell_3 \rightsquigarrow \ell$	$\ell_1 \looparrowright \ell$
$(\textbf{if}(t_1^{\ell_1})\{t_2^{\ell_2}\}\,\textbf{else}\{t_3^{\ell_3}\})^{\ell_4}$	$\bigwedge_{i=1}^{3} \models_{\mathrm{IF}} t_i^{\ell_i}$	$\ell_2 \rightsquigarrow \ell_4 \wedge \ell_3 \rightsquigarrow \ell_4$	$\ell_1 \looparrowright \ell_4$
$(t_1, \rho)^{\ell_1}$	$\models_{\mathrm{IF}} {t_1}^{\ell_1} \wedge \bigwedge_{(x \mapsto t^{\ell}) \in \rho} \models_{\mathrm{IF}} t^{\ell}$	$-$	$-$
$(\mathfrak{m} : t^{\ell_1})^{\ell_2}$	$\models_{\mathrm{IF}} t^{\ell_1}$	$\ell_1 \rightsquigarrow \ell_2 \wedge \mathfrak{m} \rightsquigarrow \ell_2$	$-$
$(t_1^{\ell_1}[t_2^{\ell_2}])^{\ell}$	$\models_{\mathrm{IF}} t_1^{\ell_1} \wedge \models_{\mathrm{IF}} t_2^{\ell_2}$	$\forall \mathtt{REC}(\ell') \in \Gamma(\ell_1) \,.\, \forall \ell'' \in \mathtt{proto}(\ell')_{\varrho} \,.$ $\forall s \,.\, \ell''.s \rightsquigarrow \ell$	$\ell_1 \looparrowright \ell$ $\ell_2 \looparrowright \ell$
$\{s_1 : t_1^{\ell_1}, \ldots, s_n : t_n^{\ell_n}\}^{\ell}$	$\bigwedge_{i=1}^{n} \models_{\mathrm{IF}} e_n$	$\exists \mathtt{REC}(\ell') \in \Gamma(\ell) \,.\, \forall i \,.\, \ell_i \rightsquigarrow \ell'.s_i$	$-$
$(t_1^{\ell_1}[t_2^{\ell_2}] = t_3^{\ell_3})^{\ell}$	$\bigwedge_{i=1}^{3} \models_{\mathrm{IF}} t_i^{\ell_i}$	$\ell_1 \rightsquigarrow \ell \wedge \forall \mathtt{REC}(\ell') \in \Gamma(\ell_1) \,.\, \forall s \,.\, \ell_3 \rightsquigarrow \ell'.s$	$\ell_2 \looparrowright \ell$
$(\textbf{del}\ t_1^{\ell_1}[t_2^{\ell_2}])^{\ell}$	$\models_{\mathrm{IF}} t_1^{\ell_1} \wedge \models_{\mathrm{IF}} t_2^{\ell_2}$	$\ell_1 \rightsquigarrow \ell$	$\ell_2 \looparrowright \ell$
$(\textbf{box}\ t^{\ell_1})^{\ell_2}$	$\models_{\mathrm{IF}} t^{\ell_1}$	$-$	$-$
$(\textbf{unbox}\ t^{\ell_1})^{\ell_2}$	$\models_{\mathrm{IF}} t^{\ell_1}$	$\forall \mathtt{BOX}(t'^{\ell'}) \in \Gamma(\ell_1) \,.\, \ell' \rightsquigarrow \ell_2$	$\ell_1 \looparrowright \ell_2$
$(\textbf{run}\ t^{\ell_1})^{\ell_2}$	$\models_{\mathrm{IF}} t^{\ell_1}$	$\forall \mathtt{BOX}(t'^{\ell'}) \in \Gamma(\ell_1) \,.\, \ell' \rightsquigarrow \ell_2$	$\ell_1 \looparrowright \ell_2$
$(\textbf{run}\ t^{\ell_1}\ \textbf{in}\ \rho)^{\ell_2}$	$\models_{\mathrm{IF}} t^{\ell_1} \wedge \models_{\mathrm{IF}} \rho$	$\forall \mathtt{BOX}(t'^{\ell'}) \in \Gamma(\ell_1) \,.\, \ell' \rightsquigarrow \ell_2$	$\ell_1 \looparrowright \ell_2$

Figure 9. Rules for generating information flow constraints

As an expression, the labels, variable names and markers occurring within an expression and the abstract values in the results of 0CFA for an expression are all finite, the process will terminate.

We express constraints between labels, variable names and markers as either direct flows ($x \rightsquigarrow y \implies x \leadsto y$) or indirect flows ($x \looparrowright y \implies x \leadsto y$). (The distinction between direct and indirect is for clarity of exposition; there is no practical difference between them with regard to the resulting analysis.)

Note that if instead we interpret $x \rightsquigarrow y$ and $x \looparrowright y$ as (elements of) relations and define $\leadsto\ =\ \rightsquigarrow \cup \looparrowright$, then $\leadsto$ satisfies the constraints.

We say that $\Gamma, \varrho, \leadsto \models_{\mathrm{IF}} e$ if $\Gamma, \varrho \models e$ and the conditions in Fig. 9 hold. As Γ, ϱ and $\leadsto$ are constant throughout the definition, we abbreviate $\Gamma, \varrho, \leadsto \models_{\mathrm{IF}} e$ to $\models_{\mathrm{IF}} e$ for clarity.

We now prove the coherence of our information flow analysis with evaluation. Like the corresponding proof for our 0CFA, this is lengthy and technical, so we only sketch it here. A mechanisation of the proof is available online.

Lemma 2 (Reduction Preserves Satisfaction): If we have $\Gamma, \varrho, \leadsto \models_{\mathrm{IF}} t_1^{\ell_1}$ and also $t_1^{\ell_1} \overset{n}{\dashrightarrow} t_2^{\ell_2}$, then $\Gamma, \varrho, \leadsto \models_{\mathrm{IF}} t_2^{\ell_2}$. Furthermore, $\ell_2 \leadsto^* \ell_1$.

Proof: By case analysis on the rules defining $\overset{n}{\dashrightarrow}$. ■

Theorem 3 (Information Flow Coherence): If we have $\Gamma, \varrho, \leadsto \models_{\mathrm{IF}} e_1$ and also $e_1 \xrightarrow{m} e_2$, then $\Gamma, \varrho, \leadsto \models_{\mathrm{IF}} e_2$. Furthermore, $\mathit{lbl}(e_2) \leadsto^* \mathit{lbl}(e_1)$.

Proof: Sketch: Unfolding the definition of $\xrightarrow{m}$, we let

$$\begin{array}{cccccccccccccc}
4 & \rightsquigarrow & 5 & \rightsquigarrow & x & \rightsquigarrow & 0 & \searrow & & 7 & \rightsquigarrow & 8 & \rightsquigarrow & y \\
\textsc{h} & \nearrow & \textsc{i} & \searrow & 3 & \looparrowright & 6 & \looparrowright & 9 & \textsc{l} & \nearrow & & & \\
 & & 1 & \rightsquigarrow & 2 & \nearrow & & & & & & & &
\end{array}$$

Figure 10. Information flow constraints for Example 5

$e_1 = C_n^m\langle t_1^{\ell_1}\rangle$ and $e_2 = C_n^m\langle t_2^{\ell_2}\rangle$ with $t_1^{\ell_1} \overset{n}{\dashrightarrow} t_2^{\ell_2}$.

Observe that $\Gamma, \varrho, \leadsto \models_{\mathrm{IF}} t_1^{\ell_1}$ and hence, applying Lemma 2, $\Gamma, \varrho, \leadsto \models_{\mathrm{IF}} t_2^{\ell_2}$, with $\ell_2 \leadsto^* \ell_1$. Observe further that constraints generated by C_n^m and the contents of its hole interact only at that hole, labelled ℓ_2 or ℓ_1. Thus, using $\ell_2 \leadsto^* \ell_1$, they must be satisfied in the conclusion, giving $\Gamma, \varrho \leadsto \models_{\mathrm{IF}} C_n^m\langle t_2^{\ell_2}\rangle$ as required.

The claim that $\mathit{lbl}(e_2) \leadsto^* \mathit{lbl}(e_1)$ is trivial for all non-empty contexts, as $\mathit{lbl}(e_2) = \mathit{lbl}(e_1)$. For the empty context, it follows directly from the similar claim in Lemma 2. ■

Note that, while the 0CFA and information flow analysis phases are conceptually distinct, correctness of the latter depends on correctness of the former. Therefore, for the sake of simplicity, our mechanisation of the proof concerns a combined formulation of the analyses in which both phases are performed simultaneously.

Example 8: Recall once more Example 5. Using the results of 0CFA, our implementation generates the relations $\rightsquigarrow$ and $\looparrowright$ as depicted in Fig. 10.

Setting $\leadsto = \rightsquigarrow \cup \looparrowright$, we have $\textsc{h} \leadsto^* 9$ and $\textsc{i} \leadsto^* 9$ and $\textsc{l} \not\leadsto^* 9$. As expected, this means the result (labelled 9) has information flows from $\textsc{h}$ and $\textsc{i}$, but not $\textsc{l}$.

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.

E. Information Flow Soundness

Theorem 4 (Information Flow Soundness): Suppose $\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} t^{\ell}$. Then if $t^{\ell} \xrightarrow{\square}{}^{*} v^{\ell'}$, where v is a stage-0 value composed only of markers and constants, then $\lfloor v \rfloor_M = v$ where $M = \{\mathfrak{m} \in \mathit{Marker} \mid \mathfrak{m} \rightsquigarrow^{*} \ell\}$.

Proof: First show that $\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} v^{\ell'}$ with $\ell' \rightsquigarrow^{*} \ell$. Argue by a simple induction over the derivation of $t^{\ell} \xrightarrow{\square}{}^{*} v$.

Base case: $\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} t^{\ell}$ follows immediately from the theorem's premise.

Inductive step: Assume that $\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} e_1$ and $\mathit{lbl}(e_1) \rightsquigarrow^{*} \ell$, with $e_1 \xrightarrow{\square} e_2$ the next step in the derivation. Apply Theorem 3 to show that $\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} e_2$ and $\mathit{lbl}(e_2) \rightsquigarrow^{*} \mathit{lbl}(e_1)$; hence $\mathit{lbl}(e_2) \rightsquigarrow^{*} \ell$.

Now we have $\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} v$ and $\ell' \rightsquigarrow^{*} \ell$. Observe from the definition of $\lfloor v \rfloor_M$ that if for every marker $\mathfrak{m}$ that occurs in v we have $\mathfrak{m} \in M$, then $\lfloor v \rfloor_M = v$.

But v is a value composed only of markers and constants, so for every marker $\mathfrak{m}$ that occurs in v (by examination of the $\models_{\mathrm{IF}}$ constraint rules) it must be the case that $\mathfrak{m} \rightsquigarrow^{*} \ell'$. Thus, as $\ell' \rightsquigarrow^{*} \ell$, $\mathfrak{m} \rightsquigarrow^{*} \ell$. Hence, from the definition of M, $\mathfrak{m} \in M$. So it is indeed true that $\lfloor v \rfloor_M = v$. ■

Relationship with Noninterference: Our information flow analysis can be used to verify the security property noninterference. Noninterference asserts that the values of any "high-security" inputs must not affect the values of any "low-security" outputs. In order for this assertion to be meaningful, we must have notions of input, output and high- and low-security levels.

For example, assume elements of *Marker* represent different levels of security, such as L for low security and H for high security. For input, assume two relations $\xrightarrow{\text{low}}$ and $\xrightarrow{\text{high}}$, which take an expression and set the values of low and high inputs respectively. For low-security output, just take the value to which an expression evaluates.

Say that expression t^{ℓ} satisfies noninterference analysis if $\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} t^{\ell}$ and $\mathrm{H} \not\rightsquigarrow^{*} \ell$. Further, require that $\xrightarrow{\text{low}}$ and $\xrightarrow{\text{high}}$ satisfy the following conditions:

$$\begin{aligned}
\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} t^{\ell} \wedge t \xrightarrow{\text{low}} t' &\quad\Longrightarrow\quad \Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} t'^{\ell} \\
\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} t^{\ell} \wedge t \xrightarrow{\text{high}} t' &\quad\Longrightarrow\quad \Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} t'^{\ell} \\
\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} t^{\ell} \wedge t \xrightarrow{\text{high}} t' &\quad\Longrightarrow\quad \lfloor t \rfloor_{\{\mathrm{L}\}} = \lfloor t' \rfloor_{\{\mathrm{L}\}}
\end{aligned}$$

Claim: If t^{ℓ} satisfies noninterference analysis, then in the following situation:

$$t^{\ell} \xrightarrow{\text{low}} t'^{\ell} \qquad t'^{\ell} \xrightarrow{\text{high}} t_1^{\ell} \qquad t'^{\ell} \xrightarrow{\text{high}} t_2^{\ell} \qquad t_1^{\ell} \xrightarrow{\square}{}^{*} u^{\ell'}$$

where $u^{\ell'}$ is a value composed only of markers and constants, it follows that $t_2^{\ell} \xrightarrow{\square}{}^{*} u^{\ell'}$. That is, the low output u is independent from the values of the high inputs for t selected using $\xrightarrow{\text{high}}$.

Proof: By the condition on $\xrightarrow{\text{low}}$, observe we have $\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} t'$. By the first condition on $\xrightarrow{\text{high}}$, it then follows that $\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} t_1^{\ell}$ and $\Gamma, \varrho, \rightsquigarrow \models_{\mathrm{IF}} t_2^{\ell}$. As $\mathrm{H} \not\rightsquigarrow^{*} \ell$, by soundness of information flow, we have $u = \lfloor u \rfloor_{\{\mathrm{L}\}}$. So using stability, we get $\lfloor t_1 \rfloor_{\{\mathrm{L}\}} \xrightarrow{\square}{}^{*} u$. But, by the second condition on $\xrightarrow{\text{high}}$, we have $\lfloor t' \rfloor_{\{\mathrm{L}\}} = \lfloor t_1 \rfloor_{\{\mathrm{L}\}} = \lfloor t_2 \rfloor_{\{\mathrm{L}\}}$. So $\lfloor t_2 \rfloor_{\{\mathrm{L}\}} \xrightarrow{\square}{}^{*} u$. Then by monotonicity, $t_2 \xrightarrow{\square}{}^{*} u$.

The conditions on $\xrightarrow{\text{low}}$ and $\xrightarrow{\text{high}}$ seem reasonable. As an example, $\xrightarrow{\text{low}}$ and $\xrightarrow{\text{high}}$ that can only replace constants marked as L and H respectively and can only replace them with constants of the same type (integer, boolean or string) satisfy these conditions.

IV. Evaluation

We have implemented our analysis in OCaml and tested it on a range of examples. The most expensive part of the analysis computationally is 0CFA, which runs in time $\mathcal{O}(n^3)$ in the size of the program [16]; consequently, it runs quickly on all our examples and we expect it to scale well to large programs. The source code for our analysis tool and the examples are available online. We now present some of these examples.

For each example, we list the markers on which our simple analysis says the result may *depend*. Where the *improved* analysis gives a more precise result, we list that too. To improve readability, we write **let** $x = v$ **in** e as a shorthand for **fun**$(x)\{e\}v$. Our implementation extends SLamJS (and its analysis) as presented in this paper with primitive arithmetic, equality and **typeof** operators, which we use in some of our examples. It can also handle mutable references in the style of λ_{JS} and a subset of actual JavaScript syntax. Many of our examples are inspired by patterns of **eval** usage common in Web applications, as surveyed by Richards et al. [8] and discussed by Jensen et al. [10].

Example 9: Depends on: H, L.
if(H : **true**){L : **false**} **else**{1}
We begin with a classic example where branching on a value introduces an indirect flow from it. As our analysis does not track specific boolean values, it would give the same result if the branch were on (H : **false**). We could resolve this imprecision by extending our abstract value domain with abstract values for **true** and **false**.

Example 10: Depends on: H, I, L. *Depends (improved)*: H, L.
let ctrue = **fun**$(x)\{$**fun**$(y)\{x\}\}$ **in**
let cif = **fun**$(x)\{$**fun**$(y)\{$**fun**$(z)\{(x(y))(z)\}\}\}$ **in**
$((\mathit{cif}(\mathrm{H} : \mathit{ctrue}))(\mathrm{L} :$ **false**$))(\mathrm{I} : 1))$
Conversely, if we present the previous example using the standard Church-encodings of **if** and **true** as functions, our analysis is precise enough to determine that the result does not depend on I. Note that we need the improved analysis to distinguish the bindings of x and y in ctrue and cif.

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.

Example 11: Depends on: $\textsc{l}$.
$\textbf{let } x = \textbf{if}(\textbf{true})\{\textbf{box } f\}\,\textbf{else}\{\textbf{box } g\}\textbf{ in}$
$\textbf{let } f = \textbf{fun}(y)\{1\}\textbf{ in}$
$\textbf{let } g = \textbf{fun}(z)\{\textsc{l} : \textbf{true}\}\textbf{ in}$
$\textbf{run } (\textbf{box } ((\textbf{unbox } x)(\textsc{h} : \textbf{undef})))$
This is modelled on the following JavaScript usage [10]:
```
if (...) x = "f"; else x = "g";
eval(x + "()");
```
f and g are bound to functions; x is set to a code value of either f or g; a function argument is added to the code value and the result executed. In this example, both f and g ignore their argument ($\textsc{h}$: **undef**), so the result does not depend on $\textsc{h}$; our analysis correctly identifies this.

Example 12: Depends on: $\textsc{h}, \textsc{l}$. *Depends (improved)*: $\textsc{l}$.
$\textbf{let } c = \textbf{box } x\textbf{ in}$
$\textbf{let } x = \textsc{l} : 1\textbf{ in}$
$\textbf{let } eval = \textbf{fun}(b)\{\textbf{run } b\}\textbf{ in}$
$\textbf{let } x = \textsc{h} : 2\textbf{ in}$
$eval(c)$
JavaScript programmers sometimes use **eval** to execute code within a different scope. SLamJS does not aim to emulate all the quirks of **eval**, but scoping of staged code can still have interesting behaviour, as shown in this example. In the scope of the definition of the function bound to $eval$, x is 1. So when it evaluates the code value c, which contains just the variable x, this is the value it returns; note that x was not bound at all where c was defined. The second binding of x is unused; our analysis correctly determines this.

Example 13: Depends on: $\textsc{h}, \textsc{i}, \textsc{l}$.
$\textbf{let } i = \textsc{i}$:
$\{\texttt{"\_proto\_"} : \textbf{null}, \texttt{"}x\texttt{"} : (\textsc{h} : 1), \texttt{"}y\texttt{"} : (\textsc{l} : 2)\}\textbf{ in}$
$\textbf{let } s = \textbf{fun}(id)\{\textbf{let } f = \textbf{box } (i[\textbf{unbox } id])\textbf{ in run } f\}\textbf{ in}$
$s(\textbf{box } \texttt{"}y\texttt{"})$
Some programmers use **eval** to construct variable names, as in (`var n = 5; eval ("f_" + n);`) to access `f_5`. We cannot express this directly in SLamJS as there are no facilities to manipulate variable names. Another common practice is to use **eval** to access object properties, often because of the programmer's ignorance of JavaScript's indirect object field access syntax; this example models that practice in SLamJS. Because our analysis does not model the values strings may take, its handling of field reads and writes is rather coarse, so it cannot tell the result will not depend on $\textsc{h}$; this could be addressed refining our abstract value domain.

Example 14: Depends on: $\textsc{h}$.
$\textbf{let } fst = \textbf{fun}(x)\{\textbf{fun}(y)\{x\}\}\textbf{ in}$
$\textbf{let } f = \textbf{if}(\textbf{false})\{fst\}\,\textbf{else}\{\textbf{box } fst\}\textbf{ in}$
$\textbf{let } x = (\textsc{h} : 1)\textbf{ in}$
$\textbf{let } y = (\textsc{l} : \textbf{true})\textbf{ in}$
$\textbf{if}(\textbf{typeof } f = \texttt{"function"})\{(f(x))(y)\}$
$\textbf{else}\{\textbf{run } (\textbf{box } (((\textbf{unbox } f)(x))(y)))\}$
This example models the JavaScript usage pattern:
```
if (f instanceof Function) f(x);
else eval (f + "(x)");
```
which may arise when using **eval** to emulate higher-order functions. Here, our analysis shows the same precision on a boxed value representing a function as when dealing with a real function.

Example 15: Depends on: $\textsc{h}, \textsc{l}$. *Depends (improved)*: $\textsc{l}$.
$\textbf{let } pair = \textbf{fun}(x)\{\textbf{fun}(y)\{\textbf{fun}(z)\{\textbf{run } z\}\}\}\textbf{ in}$
$\textbf{let } fst = \textbf{fun}(z)\{z(\textbf{box } x)\}\textbf{ in}$
$\textbf{let } snd = \textbf{fun}(z)\{z(\textbf{box } y)\}\textbf{ in}$
$\textbf{let } bp = \textbf{box } ((pair(\textsc{l} : (\textbf{box } (1))))(\textsc{h} : (\textbf{box } (\textbf{true}))))\textbf{ in}$
$\textbf{let } boxfst = \textbf{box } ((fst)(\textbf{unbox } bp))\textbf{ in}$
$\textbf{run } (\textbf{run } (boxfst))$
Most examples of staged metaprogramming in the literature do not use more than one level of staging. This example, which pairs and unpairs two values in a rather roundabout way, illustrates that we can handle higher levels too.

Example 16: Depends on: $\textsc{h}$.
$\textbf{fun}(n)\{(\textbf{fun}(x)\{(x(x))(n)\})$
$(\textbf{fun}(x)\{\textbf{fun}(y)\{\textbf{if}(y = 0)\{\textbf{true}\}\,\textbf{else}\{(x(x))(y-1)\}\}\})$
$\}(\textsc{h} : 5)$
This program loops n times (where n is ($\textsc{h}$: 5) in this instance) before returning **true**. In this sense, the result is independent of n: if n were a high-security input and the output low, the program would satisfy noninterference, although the duration of execution may leak information about n. However, n must be examined in order to execute the program, so there is an information flow from n to the result, in the sense captured by our augmented semantics. That is, no noninterference analysis based on a sound over-approximation of the behaviour of such a semantics could ever show the program to be noninterfering [17].

Example 17: Depends on: $\textsc{l}$.
$\textbf{let } fst = \textbf{fun}(x)\{\textbf{fun}(y)\{x\}\}\textbf{ in}$
$\textbf{let } a = \textbf{box } x\textbf{ in}$
$\textbf{let } b = \textbf{box } (\textbf{fun}(x)\{\textbf{fun}(y)\{fst(\textbf{unbox } a)(y)\}\})\textbf{ in}$
$(\textbf{run } b)(\textsc{l} : 1)(\textsc{h} : 2)$
This program, based on an example from Choi et al. [9], splices a variable name into a code template to produce code that takes two arguments and returns the first. Our analysis correctly determines that the result depends only on the first.

Example 18: Depends on: $\textsc{l}, \textsc{h}$.
$\textbf{let } fst = \textbf{fun}(x)\{\textbf{fun}(y)\{x\}\}\textbf{ in}$
$\textbf{let } a = \textbf{fun}(p)\{p[\texttt{"x"}]\}\textbf{ in}$
$\textbf{let } b = (\textbf{fun}(h)\{\textbf{fun}(p)\{\textbf{fun}(x)\{\textbf{fun}(y)\{$
$fst(h((p[\texttt{"x"}] = x)[\texttt{"y"}] = y))(y)\}\}\}\})(a)\textbf{ in}$
$b(\{\texttt{"\_\_proto\_\_"} : \textbf{null}\})(\textsc{l} : 1)(\textsc{h} : 2)$
By applying Choi et al.'s unstaging translation to the core of the previous example, we obtain this unstaged one. Note that while the result of the program is the same, we lose precision by analysing this version instead of working directly on the staged version.

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.

Example 19: Depends on: L, H. *Depends (improved):* L.
let $blank = $ **fun**$(get)\{get($**null**$)($**null**$)\}$ **in**
let $getx = $ **fun**$(x)\{$**fun**$(y)\{x\}\}$ **in**
let $gety = $ **fun**$(x)\{$**fun**$(y)\{y\}\}$ **in**
let $setx = $ **fun**$(env)\{$**fun**$(newx)\{$
fun$(get)\{get(newx)(env(gety))\}\}\}$ **in**
let $sety = $ **fun**$(env)\{$**fun**$(newy)\{$
fun$(get)\{get(env(getx))(newy)\}\}\}$ **in**
let $fst = $ **fun**$(x)\{$**fun**$(y)\{x\}\}$ **in**
let $a = $ **fun**$(p)\{p(getx)\}$ **in**
let $b = ($**fun**$(h)\{$**fun**$(p)\{$**fun**$(x)\{$**fun**$(y)\{$
$fst(h(sety(setx(p)(x))(y)))(y)\}\}\}\})(a)$ **in**
$b(blank)(\text{L}:1)(\text{H}:2)$
Here we have applied the unstaging translation, as in the previous example, but using higher order functions to encode environments instead of records. In this case, we can recover the lost precision, but at the cost of an $\mathcal{O}(n^2)$ increase in the size of the source program, making the combined analysis $\mathcal{O}(n^6)$ instead of $\mathcal{O}(n^3)$.

V. Related Work

A. From SLamJS to JavaScript Applications

The application that guided our work is information flow analysis for JavaScript in Web applications. We now consider some of the features of this scenario that we have not addressed and how they have been handled by others. We claim that most of the problems have been addressed, although combining them into a single analysis system would require further effort.

Handling of Primitive Datatypes As demonstrated in some of our examples, our analysis models its primitive datatypes (such as strings and booleans) very coarsely; our abstract domains are too simple. Fortunately, more refined abstractions for these datatypes have been well-studied [18].

Imperative Control Flow and Exceptions JavaScript has several features not found in SLamJS, including typical imperative control flow features (such as **for** loops) and exceptions, but there are CFA-style analyses for JavaScript. Perhaps most notable is the recent CFA2 analysis [5], which was developed for JavaScript and features significantly better analysis of higher order flow control.

JavaScript Semantics A bigger problem in producing a sound analysis of JavaScript is the complexity and quaintness of its semantics [7]. Guha et al. attempt to simplify this problem by producing a much simpler "core calculus" for JavaScript called λ_{JS} and a transformation from JavaScript into λ_{JS} [3]. They have mechanised various proofs about their language in Coq. As Web applications execute in the context of a webpage in a browser, an analysis must also model how a webpage interacts with code via the DOM.

Code Strings vs Staged Code Perhaps the most relevant difference between JavaScript and SLamJS is our metaprogramming constructs: JavaScript **eval** runs on strings, while, in an effort to develop a more principled analysis, our staged metaprogramming follows the tradition of Lisp quotations. To analyse uses of **eval** with our techniques, we would need a sound transformation into staged metaprogramming. Jensen et al. use the result of a string analysis produced by the tool TAJS to replace certain uses of **eval** with unstaged code where it is safe to do so [10]; the transformed program is then fed back into the analysis tool. We propose to handle a wider range of use cases with the more general approach of transforming **eval** on strings into staged code and then analysing the staged code [19].

Reactive Systems A practical Web application is not simply a program that takes inputs, runs once, then gives output: it may interleave input and output throughout its execution, which might not terminate. Bohannon et al. consider the consequences of this for information security in their work on reactive noninterference [20].

Infrastructural Issues In applying an information flow analysis to a Web application, several infrastructural issues need to be addressed. Would the code be analysed before being published by on a webserver, in the browser running it or by some proxy in between? Will the entire code be available in advance, or must it be analysed in fragments [21]? Who would set the security policies that the analysis should enforce? Li and Zdancewic argue that noninterference alone is too strict a policy to enforce and that a practical policy must allow for limited declassification [22].

B. Information Flow Analysis

Early work on information flow security focused on monitoring program execution, dynamically marking variables to indicate their level of confidentiality [23]. However, the study of static analysis for information flow security can essentially be traced back to Denning, who introduced a lattice model for secure information flow and critically considered both direct and indirect flows [24]. Denning and Denning developed a simple static information flow analysis that rejected programs with flows violating a security policy [25].

Noninterference Goguen and Meseguer introduced the idea of noninterference [1] (the inability of the actions of one party, or equivalently data at one level, to influence those of another) as a way of specifying security policies, including enforcement of information flow security. Noninterference and information flow security became almost synonymous, although Pottier and Conchon were careful to emphasise the distinction between the two [12].

Security Type Systems Security type systems became a common way of enforcing noninterference policies and proving the correctness of noninterference analyses, progressing from a reformulation of Denning and Denning's analysis [26] to Simonet and Pottier's type system for ML [6]. Unfortunately, the requirement that the program analysed follow a strict type discipline makes it impractical to apply these ideas to dynamically typed languages such as JavaScript. Perhaps as a consequence, information flow

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.

in untyped and dynamically typed languages is relatively poorly understood.

Dynamic Analyses Dynamic information flow analysis circumvents the need for a type system or other static analysis by tracking information flow during program execution, and enforcing security policies by aborting program execution if an undesired flow is detected; examples of such analyses for JavaScript are presented by Just et al. [27] and Hedin and Sabelfeld [28]. Indeed, the problems they address and their motivations are very similar to ours, but our methods are very different.

Dynamic vs Static A dynamic analysis only observes one program run at a time, so dynamic code generation is easy to handle. However, care has to be taken to track indirect information flow due to code that was *not* executed in the observed run. Strategies to achieve this include, for instance, the *no-sensitive upgrade* check [29], which aborts execution if a public variable is assigned in code that is control dependent on private data. As a rule, however, such strategies are fairly coarse and could potentially abort many innocuous executions; thus it is commonly held that static analyses are superior to dynamic ones in their treatment of indirect flows [30], although there has been a resurgence of interest in dynamic analyses [31].

Hybrid Approaches As a compromise, Chugh et al. [21] propose extending a static information flow analysis with a dynamic component that performs additional checks at runtime when dynamically generated code becomes available. The static part of their analysis is similar to ours (minus staging), although they do not formally state or prove its soundness. Their study of JavaScript on popular websites suggests the static part is precise enough to be useful. Because the additional checks on dynamically generated code occur at runtime, they must necessarily be quick and simple to avoid performance degradation. Consequently, these checks are limited to purely syntactic isolation properties, with a corresponding loss of precision. Our fully static analysis does not suffer from these limitations.

Going in the other direction, Austin and Flanagan [32] have proposed *faceted execution*, a form of dynamic analysis that explores different execution paths and can thus recover some of the advantages of a static analysis.

C. Static Analysis of Staged Metaprogramming

Many different approaches to staged metaprogramming have been proposed. Our language's staging constructs are modelled after the language λ_S of Choi et al. [9]. However, our semantics of variable capture are different. For example, we allow the program $(\textbf{fun}(x)\{\textbf{run}\,(\textbf{box}\ x)\}(1))$, which behaves much like this JavaScript program: `(function (x) {return eval("x")})(1);`

Control flow analysis for a two-staged language has been investigated by Kim et al. [33]. Their approach is based on abstract interpretation, putting particular emphasis on inferring an over-approximation of all possible pieces of code to which a code quotation may evaluate. This information is not explicitly computed by our analysis, so it is quite possible that their analysis is more precise than ours. However it does not seem to have been implemented yet.

Choi et al. [9] propose a more general framework for static analysis of multi-staged programs, which is based on an unstaging translation that replaces staging constructs with function abstractions and applications. Under certain conditions, analysis results for the unstaged program can then be translated back to its staged version.

There are some limitations to their work. Most significantly, many interesting programs, such as the one mentioned earlier, are not valid in λ_S and hence cannot be unstaged using their translation; this limits its applicability to JavaScript. Furthermore, as shown in Examples 17–19, the precision of the resulting combined analysis is highly sensitive to the target language encoding used in the translation and the behaviour of the target language analysis. While their approach is useful as a quick way of adding staging to an existing language and analysis, we argue that staging constructs are sufficiently important and complex that we should aim to analyse them directly.

Inoue and Taha [34] consider the problem of reasoning about staged programs; in particular, they identify equivalences that fail to hold in the presence of staging, and develop a notion of bisimulation that can be used to prove extensionality of function abstractions, and work around some of the failing equivalences. Their language differs from ours in that it avoids name capture.

Some work on analysing metaprogramming focuses on its application to optimising compilation of programs with metaprogramming. For example, Smith et al. [35] consider using static analysis to optimise compilation in a cut-down version of Cyclone, a type-safe, C-style language with runtime code generation. Their analysis is based around a relatively coarse over-approximation of control flow between code blocks in a program, but this suits their application because their language does not have first-class functions.

VI. Conclusions

We have presented a fully static information flow analysis based on 0CFA for a dynamically typed language with staged metaprogramming, implemented it and formally proved its soundness. We believe our approach is transferrable to other CFA-style analyses and applicable to JavaScript.

Progressing from here, there are three obvious lines of work. The first is to improve the precision of the analysis by applying its ideas to CFA2 or using results from abstract interpretation. The second is to extend the language to handle more features, such as imperative control flow and exceptions. The third and most important, which we are

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.

already pursuing [19], is to apply string analysis techniques to produce a sound transformation from a language with **eval** on code strings to a language with staged code values.

All the pieces are now in place for an interesting, sound and principled analysis of JavaScript with **eval**, but it will take significant effort to bring them together.

Acknowledgments: We thank our anonymous reviewers for their comments and suggestions.

References

[1] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[2] I.-S. Kim, K. Yi, and C. Calcagno, "A polymorphic modal type system for lisp-like multi-staged languages," in *POPL*, 2006, pp. 257–268.

[3] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The Essence of JavaScript," in *ECOOP*, 2010, pp. 126–150.

[4] O. Shivers, "Control-Flow Analysis in Scheme," in *PLDI*, 1988, pp. 164–174.

[5] D. Vardoulakis and O. Shivers, "CFA2: a Context-Free Approach to Control-Flow Analysis," *Logical Methods in Computer Science*, vol. 7, no. 2, 2011.

[6] F. Pottier and V. Simonet, "Information Flow Inference for ML," *TOPLAS*, vol. 25, no. 1, pp. 117–158, 2003.

[7] S. Maffeis, J. C. Mitchell, and A. Taly, "An Operational Semantics for JavaScript," in *APLAS*, 2008, pp. 307–325.

[8] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The Eval That Men Do — A Large-Scale Study of the Use of Eval in JavaScript Applications," in *ECOOP*, 2011.

[9] W. Choi, B. Aktemur, K. Yi, and M. Tatsuta, "Static Analysis of Multi-staged Programs via Unstaging Translation," in *POPL*, 2011, pp. 81–92.

[10] S. H. Jensen, P. A. Jonsson, and A. Møller, "Remedying the Eval that Men Do," in *ISSTA*, 2012, pp. 34–44.

[11] D. V. Horn and M. Might, "An analytic framework for javascript," *CoRR*, vol. abs/1109.4467, 2011.

[12] F. Pottier and S. Conchon, "Information Flow Inference for Free," in *ICFP*, 2000.

[13] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A Core Calculus of Dependency," in *POPL*, 1999, pp. 147–160.

[14] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 1999.

[15] M. Lester, C.-H. L. Ong, and M. Schäfer, "Information flow analysis for a dynamically typed functional language with staged metaprogramming," *CoRR*, vol. abs/1302.3178, 2013.

[16] N. Heintze and D. A. McAllester, "On the cubic bottleneck in subtyping and flow analysis," in *LICS*. IEEE Computer Society, 1997, pp. 342–351.

[17] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *CSF*, 2010, pp. 186–199.

[18] T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh, "A Practical String Analyzer by the Widening Approach," in *APLAS*, 2006, pp. 374–388.

[19] M. Lester, "Position paper: The science of boxing — analysing eval using staged metaprogramming," in *PLAS*, 2013, in press.

[20] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, "Reactive Noninterference," in *Computer and Communications Security*, 2009, pp. 79–90.

[21] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged Information Flow for JavaScript," in *PLDI*, 2009, pp. 50–62.

[22] P. Li and S. Zdancewic, "Downgrading policies and relaxed noninterference," in *POPL*, 2005, pp. 158–170.

[23] J. S. Fenton, "Memoryless subsystems," *Comput. J.*, vol. 17, no. 2, pp. 143–147, 1974.

[24] D. E. Denning, "A Lattice Model of Secure Information Flow," *CACM*, vol. 19, no. 5, pp. 236–243, 1976.

[25] D. E. Denning and P. J. Denning, "Certification of Programs for Secure Information Flow," *CACM*, vol. 20, no. 7, pp. 504–513, 1977.

[26] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2/3, pp. 167–188, 1996.

[27] S. Just, A. Cleary, B. Shirley, and C. Hammer, "Information Flow Analysis for JavaScript," in *PLASTIC*, 2011.

[28] D. Hedin and A. Sabelfeld, "Information-flow security for a core of javascript," in *CSF*, S. Chong, Ed. IEEE, 2012, pp. 3–18.

[29] S. Zdancewic, "Programming Languages for Information Security," Ph.D. dissertation, Cornell University, 2002.

[30] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[31] A. Sabelfeld and A. Russo, "From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research," in *Ershov Memorial Conf.*, 2009.

[32] T. H. Austin and C. Flanagan, "Multiple Facets for Dynamic Information Flow," in *POPL*, 2012, pp. 165–178.

[33] T. Kim, C. Lee, K. Lee, S. Baik, and K. Yi, "A Control Flow Analysis for 2-staged Programming Languages," ROSAEC, Techreport ROSAEC-2009-005, 2009.

[34] J. Inoue and W. Taha, "Reasoning About Multi-stage Programs," in *ESOP*, 2012.

[35] F. Smith, D. Grossman, J. G. Morrisett, L. Hornof, and T. Jim, "Compiling for template-based run-time code generation," *J. Funct. Program.*, vol. 13, no. 3, pp. 677–708, 2003.

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.

Appendix
SLamJS Semantics Definitions

$$
\begin{array}{rcl}
(k,\rho) & \overset{n}{\dashrightarrow} & k \\
(\{\overline{s:e}\},\rho) & \overset{n}{\dashrightarrow} & \{\overline{s:(e,\rho)}\} \\
(x,\rho) & \overset{n+1}{\dashrightarrow} & x \\
(\textbf{fun}(x)\{e\},\rho) & \overset{n+1}{\dashrightarrow} & (\textbf{fun}(x)\{(e,\rho)\}) \\
(e_1(e_2),\rho) & \overset{n}{\dashrightarrow} & ((e_1,\rho)((e_2,\rho))) \\
(\textbf{box}\ e,\rho) & \overset{n}{\dashrightarrow} & (\textbf{box}\ (e,\rho)) \\
(\textbf{unbox}\ e,\rho) & \overset{n}{\dashrightarrow} & (\textbf{unbox}\ (e,\rho)) \\
(\textbf{run}\ e,\rho) & \overset{0}{\dashrightarrow} & (\textbf{run}\ (e,\rho)\ \textbf{in}\ \rho) \\
(\textbf{run}\ e,\rho) & \overset{n+1}{\dashrightarrow} & (\textbf{run}\ (e,\rho)) \\
(\textbf{if}(e_1)\{e_2\}\,\textbf{else}\{e_3\},\rho) & \overset{n}{\dashrightarrow} & (\textbf{if}((e_1,\rho))\{(e_2,\rho)\}\,\textbf{else}\{(e_3,\rho)\}) \\
(e_1[e_2],\rho) & \overset{n}{\dashrightarrow} & ((e_1,\rho)[(e_2,\rho)]) \\
(e_1[e_2]=e_3,\rho) & \overset{n}{\dashrightarrow} & ((e_1,\rho)[(e_2,\rho)]=(e_3,\rho)) \\
(\textbf{del}\ e_1[e_2],\rho) & \overset{n}{\dashrightarrow} & (\textbf{del}\ (e_1,\rho)[(e_2,\rho)])
\end{array}
$$

Figure 11. Environment propagation rules

$$
\begin{array}{rlll}
C_n^m & ::= & [\,] & \in \mathcal{C}_n^n \\
 & \mid & (\{\overline{s:v^m}, s:C_n^m, \overline{s:e}\}) & \in \mathcal{C}_n^m \\
 & \mid & (\textbf{fun}(x)\{C_n^{m+1}\}) & \in \mathcal{C}_n^{m+1} \\
 & \mid & (C_n^m(e)) & \in \mathcal{C}_n^m \\
 & \mid & (v^m(C_n^m)) & \in \mathcal{C}_n^m \\
 & \mid & (\textbf{box}\ C_n^{m+1}) & \in \mathcal{C}_n^m \\
 & \mid & (\textbf{unbox}\ C_n^m) & \in \mathcal{C}_n^{m+1} \\
 & \mid & (\textbf{run}\ C_n^m) & \in \mathcal{C}_n^m \\
 & \mid & (\textbf{if}(C_n^m)\{e\}\,\textbf{else}\{e\}) & \in \mathcal{C}_n^m \\
 & \mid & (\textbf{if}(v^{m+1})\{C_n^{m+1}\}\,\textbf{else}\{e\}) & \in \mathcal{C}_n^{m+1} \\
 & \mid & (\textbf{if}(v^{m+1})\{v^{m+1}\}\,\textbf{else}\{C_n^{m+1}\}) & \in \mathcal{C}_n^{m+1} \\
 & \mid & (C_n^m[e]) & \in \mathcal{C}_n^m \\
 & \mid & (v^m[C_n^m]) & \in \mathcal{C}_n^m \\
 & \mid & (C_n^m[e]=e) & \in \mathcal{C}_n^m \\
 & \mid & (v^m[C_n^m]=e) & \in \mathcal{C}_n^m \\
 & \mid & (v^m[v^m]=C_n^m) & \in \mathcal{C}_n^m \\
 & \mid & (\textbf{del}\ C_n^m[e]) & \in \mathcal{C}_n^m \\
 & \mid & (\textbf{del}\ v^m[C_n^m]) & \in \mathcal{C}_n^m \\
 & \mid & (\textbf{run}\ C_n^m\ \textbf{in}\ \rho) & \in \mathcal{C}_n^m
\end{array}
$$

Figure 12. Evaluation contexts

Appendix
0CFA for SLamJS

A. Labelled Semantics

We extend the syntax of SLamJS with labels to indicate program points. The labels have no effect on the result of computation, but are used to track which values may occur at which points. Consequently, it is important for the soundness of the corresponding analysis that the semantics correctly tracks labels.

$$
\begin{array}{rcll}
\lfloor \_ \rfloor_M & = & \_ & \\
\lfloor k \rfloor_M & = & k & \\
\lfloor \{\overline{s:e}\} \rfloor_M & = & \{\overline{s:\lfloor e \rfloor_M}\} & \\
\lfloor x \rfloor_M & = & x & \\
\lfloor \textbf{fun}(x)\{e\} \rfloor_M & = & \textbf{fun}(x)\{\lfloor e \rfloor_M\} & \\
\lfloor e_1(e_2) \rfloor_M & = & \lfloor e_1 \rfloor_M(\lfloor e_2 \rfloor_M) & \\
\lfloor \textbf{box}\ e \rfloor_M & = & \textbf{box}\ \lfloor e \rfloor_M & \\
\lfloor \textbf{unbox}\ e \rfloor_M & = & \textbf{unbox}\ \lfloor e \rfloor_M & \\
\lfloor \textbf{run}\ e \rfloor_M & = & \textbf{run}\ \lfloor e \rfloor_M & \\
\lfloor \textbf{if}(e_1)\{e_2\}\,\textbf{else}\{e_3\} \rfloor_M & = & \textbf{if}(\lfloor e_1 \rfloor_M)\{\lfloor e_2 \rfloor_M\}\,\textbf{else}\{\lfloor e_3 \rfloor_M\} & \\
\lfloor e_1[e_2] \rfloor_M & = & \lfloor e_1 \rfloor_M[\lfloor e_2 \rfloor_M] & \\
\lfloor e_1[e_2]=e_3 \rfloor_M & = & \lfloor e_1 \rfloor_M[\lfloor e_2 \rfloor_M]=\lfloor e_3 \rfloor_M & \\
\lfloor \textbf{del}\ e_1[e_2] \rfloor_M & = & \textbf{del}\ \lfloor e_1 \rfloor_M[\lfloor e_2 \rfloor_M] & \\
\lfloor (e,\rho) \rfloor_M & = & (\lfloor e \rfloor_M, \lfloor \rho \rfloor_M) & \\
\lfloor \textbf{run}\ e\,\textbf{in}\ \rho \rfloor_M & = & \textbf{run}\ \lfloor e \rfloor_M\ \textbf{in}\ \lfloor \rho \rfloor_M & \\
\lfloor \rho \rfloor_M(x) & = & \lfloor \rho(x) \rfloor_M & \\
\lfloor \mathfrak{m}:e \rfloor_M & = & \mathfrak{m}:\lfloor e \rfloor_M & \text{if } \mathfrak{m} \in M \\
\lfloor \mathfrak{m}:e \rfloor_M & = & \_ & \text{if } \mathfrak{m} \notin M
\end{array}
$$

Figure 13. Definition of $\lfloor e \rfloor_M$, the M-erasure of e

We reformulate the syntax of SLamJS to distinguish between terms (expressions in the unlabelled semantics) and expressions (which are labelled terms):

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & t^\ell \\
\text{Terms} & t & ::= & k \mid \{\overline{s:e}\} \mid x \mid \textbf{fun}(x)\{e\} \mid e(e) \\
 & & \mid & \textbf{box}\ e \mid \textbf{unbox}\ e \mid \textbf{run}\ e \\
 & & \mid & \textbf{if}(e)\{e\}\,\textbf{else}\{e\} \mid e[e] \mid e[e]=e \\
 & & \mid & \textbf{del}\ e[e] \mid (t,\rho) \mid \textbf{run}\ e\ \textbf{in}\ \rho
\end{array}
$$

Values remain expressions, so they include labels at the outer level. For example, k^ℓ is a value, rather than k. Contexts other than the empty context also gain labels at the outer level, so we have $(C_n^m\langle e\rangle)^\ell$ rather than $(C_n^m\langle e\rangle)$.

The labelling of the reduction rules is a little more complicated, so we list them in full in Fig. 14, 15 and 16. For an expression $e = t^\ell$, we write $e^{\ell'}$ as a shorthand for $t^{\ell'}$ and $(e,\rho)^{\ell'}$ for $(t,\rho)^{\ell'}$. Note that we use this in the rules (Lookup), (Unbox), (Run) and (Read1).

B. Analysis

The abstract domains of the analysis are defined in Fig. 6. Abstract variables of the form x represent function parameters; abstract variables of the form $\ell.p$ represent record fields. Note that e, ℓ, x and p only range over expressions, labels and names occurring in the program to be analysed, hence the abstract domains are finite.

In the extended version of the paper [15], we give full definitions of three acceptability judgements $\Gamma, \varrho \models e$; $\Gamma, \varrho \models \rho$ and $\Gamma, \varrho \models \nu \approx t$ by mutual induction.

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.

$$
\begin{array}{rcl}
(k,\rho)^{\ell} & \overset{n}{\dashrightarrow} & k^{\ell} \\
(\{\overline{s : t^{\ell}}\},\rho)^{\ell'} & \overset{n}{\dashrightarrow} & \{\overline{s : (t,\rho)^{\ell}}\}^{\ell'} \\
(x,\rho)^{\ell} & \overset{n+1}{\dashrightarrow} & x^{\ell} \\
(\textbf{fun}(x)\{t^{\ell}\},\rho)^{\ell'} & \overset{n+1}{\dashrightarrow} & (\textbf{fun}(x)\{(t,\rho)^{\ell}\})^{\ell'} \\
(t_1^{\ell_1}(t_2^{\ell_2}),\rho)^{\ell} & \overset{n}{\dashrightarrow} & ((t_1,\rho)^{\ell_1}((t_2,\rho)^{\ell_2}))^{\ell} \\
(\textbf{box}\ t^{\ell},\rho)^{\ell'} & \overset{n}{\dashrightarrow} & (\textbf{box}\ (t,\rho)^{\ell})^{\ell'} \\
(\textbf{unbox}\ t^{\ell},\rho)^{\ell'} & \overset{n}{\dashrightarrow} & (\textbf{unbox}\ (t,\rho)^{\ell})^{\ell'} \\
(\textbf{run}\ t^{\ell},\rho)^{\ell'} & \overset{0}{\dashrightarrow} & (\textbf{run}\ (t,\rho)^{\ell}\ \textbf{in}\ \rho)^{\ell'} \\
(\textbf{run}\ t^{\ell},\rho)^{\ell'} & \overset{n+1}{\dashrightarrow} & (\textbf{run}\ (t,\rho)^{\ell})^{\ell'} \\
(\textbf{if}(t_1^{\ell_1})\{t_2^{\ell_2}\}\,\textbf{else}\{t_3^{\ell_3}\},\rho)^{\ell_0} & \overset{n}{\dashrightarrow} & (\textbf{if}((t_1,\rho)^{\ell_1})\{(t_2,\rho)^{\ell_2}\}\,\textbf{else}\{(t_3,\rho)^{\ell_3}\})^{\ell_0} \\
(t_1^{\ell_1}[t_2^{\ell_2}],\rho)^{\ell_0} & \overset{n}{\dashrightarrow} & ((t_1,\rho)^{\ell_1}[(t_2,\rho)^{\ell_2}])^{\ell_0} \\
(t_1^{\ell_1}[t_2^{\ell_2}] = t_3^{\ell_3},\rho)^{\ell_0} & \overset{n}{\dashrightarrow} & ((t_1,\rho)^{\ell_1}[(t_2,\rho)^{\ell_2}] = (t_3,\rho)^{\ell_3})^{\ell_0} \\
(\textbf{del}\ t_1^{\ell_1}[t_2^{\ell_2}],\rho)^{\ell_0} & \overset{n}{\dashrightarrow} & (\textbf{del}\ (t_1,\rho)^{\ell_1}[(t_2,\rho)^{\ell_2}])^{\ell_0} \\
(\mathtt{m} : t_1^{\ell_1},\rho)^{\ell} & \overset{n}{\dashrightarrow} & (\mathtt{m} : (t_1,\rho)^{\ell_1})^{\ell}
\end{array}
$$

Figure 14. Labelled environment propagation rules

$$
\begin{array}{lrcll}
(\textsc{Lookup}) & (x,\rho)^{\ell} & \overset{0}{\dashrightarrow} & v^{\ell} & \text{where } \rho(x) = v \\
(\textsc{Apply}) & ((\textbf{fun}(x)\{t^{\ell_1}\},\rho)^{\ell_2}(v))^{\ell_3} & \overset{0}{\dashrightarrow} & (t,\rho[x \mapsto v])^{\ell_1} & \\
(\textsc{Unbox}) & (\textbf{unbox}\ (\textbf{box}\ v^1)^{\ell_1})^{\ell_2} & \overset{1}{\dashrightarrow} & (v^1)^{\ell_2} & \\
(\textsc{Run}) & (\textbf{run}\ (\textbf{box}\ v^1)^{\ell_1}\ \textbf{in}\ \rho)^{\ell_2} & \overset{0}{\dashrightarrow} & (v^1,\rho)^{\ell_2} & \\
(\textsc{IfTrue}) & (\textbf{if}(\textbf{true})\{t_1^{\ell_1}\}\,\textbf{else}\{t_2^{\ell_2}\})^{\ell} & \overset{0}{\dashrightarrow} & t_1^{\ell_1} & \\
(\textsc{IfFalse}) & (\textbf{if}(\textbf{false})\{t_1^{\ell_1}\}\,\textbf{else}\{t_2^{\ell_2}\})^{\ell} & \overset{0}{\dashrightarrow} & t_2^{\ell_2} & \\
(\textsc{Read1}) & (\{\overline{s:v}, s_i : v_i, \overline{s:v}'\}^{\ell_1}[s_i^{\ell_2}])^{\ell_3} & \overset{0}{\dashrightarrow} & v_i^{\ell_3} & \\
(\textsc{Read2}) & (\{\overline{s:v}, \texttt{"\_\_proto\_\_"} : \{\overline{s:v}'\}^{\ell_1'}, \overline{s:v}''\}^{\ell_1}[s_x^{\ell_2}])^{\ell_3} & \overset{0}{\dashrightarrow} & (\{\overline{s:v}'\}^{\ell_1'}[s_x^{\ell_2}])^{\ell_3} & \text{if } s_x \notin \overline{s} \cup \overline{s}'' \\
(\textsc{Read3}) & (\{\overline{s:v}, \texttt{"\_\_proto\_\_"} : \textbf{null}^{\ell_1'}, \overline{s:v}''\}^{\ell_1}[s_x^{\ell_2}])^{\ell_3} & \overset{0}{\dashrightarrow} & \textbf{undef}^{\ell_3} & \text{if } s_x \notin \overline{s} \cup \overline{s}'' \\
(\textsc{Write1}) & (\{\overline{s:v}, s_i : v_i, \overline{s:v}'\}^{\ell_1}[s_i^{\ell_2}] = v_i')^{\ell_3} & \overset{0}{\dashrightarrow} & \{\overline{s:v}, s_i : v_i', \overline{s:v}'\}^{\ell_3} & \\
(\textsc{Write2}) & (\{\overline{s:v}\}^{\ell_1}[s_x^{\ell_2}] = v_x)^{\ell_3} & \overset{0}{\dashrightarrow} & \{\overline{s:v}, s_x : v_x\}^{\ell_3} & \text{if } s_x \notin \overline{s} \\
(\textsc{Del1}) & (\textbf{del}\ \{\overline{s:v}, s_i : v_i, \overline{s:v}'\}^{\ell_1}[s_i^{\ell_2}])^{\ell_3} & \overset{0}{\dashrightarrow} & \{\overline{s:v}, \overline{s:v}'\}^{\ell_3} & \\
(\textsc{Del2}) & (\textbf{del}\ \{\overline{s:v}\}^{\ell_1}[s_x^{\ell_2}])^{\ell_3} & \overset{0}{\dashrightarrow} & \{\overline{s:v}\}^{\ell_3} & \text{if } s_x \notin \overline{s}
\end{array}
$$

Figure 15. Labelled proper reduction rules

$$
\begin{array}{lrcl}
(\textsc{Lift-App}) & (((\mathtt{m} : t^{\ell_1}),\rho)^{\ell_2}(v))^{\ell_3} & \overset{0}{\dashrightarrow} & (\mathtt{m} : ((t,\rho)^{\ell_1}(v))^{\ell_3})^{\ell_3} \\
(\textsc{Lift-If}) & (\textbf{if}((\mathtt{m} : v)^{\ell_0})\{t_1^{\ell_1}\}\,\textbf{else}\{t_2^{\ell_2}\})^{\ell} & \overset{0}{\dashrightarrow} & (\mathtt{m} : (\textbf{if}(v)\{t_1^{\ell_1}\}\,\textbf{else}\{t_2^{\ell_2}\})^{\ell})^{\ell} \\
(\textsc{Lift-Unbox}) & (\textbf{unbox}\ (\mathtt{m} : v)^{\ell_1})^{\ell_2} & \overset{1}{\dashrightarrow} & (\mathtt{m} : (\textbf{unbox}\ v)^{\ell_2})^{\ell_2} \\
(\textsc{Lift-RunIn}) & (\textbf{run}\ (\mathtt{m} : v)^{\ell_1}\ \textbf{in}\ \rho)^{\ell_2} & \overset{0}{\dashrightarrow} & (\mathtt{m} : (\textbf{run}\ v\ \textbf{in}\ \rho)^{\ell_2})^{\ell_2} \\
(\textsc{Lift-ReadSel}) & (v_1[(\mathtt{m} : v_2)^{\ell_1}])^{\ell_2} & \overset{0}{\dashrightarrow} & (\mathtt{m} : (v_1[v_2])^{\ell_2})^{\ell_2} \\
(\textsc{Lift-ReadRec}) & ((\mathtt{m} : v_1)^{\ell_1}[v_2])^{\ell_2} & \overset{0}{\dashrightarrow} & (\mathtt{m} : (v_1[v_2])^{\ell_2})^{\ell_2} \\
(\textsc{Lift-WriteSel}) & (v_1[(\mathtt{m} : v_2)^{\ell_1}] = v_3)^{\ell_2} & \overset{0}{\dashrightarrow} & (\mathtt{m} : (v_1[v_2] = v_3)^{\ell_2})^{\ell_2} \\
(\textsc{Lift-WriteRec}) & ((\mathtt{m} : v_1)^{\ell_1}[v_2] = v_3)^{\ell_2} & \overset{0}{\dashrightarrow} & (\mathtt{m} : (v_1[v_2] = v_3)^{\ell_2})^{\ell_2} \\
(\textsc{Lift-DelSel}) & (\textbf{del}\ v_1[(\mathtt{m} : v_2)^{\ell_1}])^{\ell_2} & \overset{0}{\dashrightarrow} & (\mathtt{m} : (\textbf{del}\ v_1[v_2])^{\ell_2})^{\ell_2} \\
(\textsc{Lift-DelRec}) & (\textbf{del}\ (\mathtt{m} : v_1)^{\ell_1}[v_2])^{\ell_2} & \overset{0}{\dashrightarrow} & (\mathtt{m} : (\textbf{del}\ v_1[v_2])^{\ell_2})^{\ell_2}
\end{array}
$$

Figure 16. Labelled lifts

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 26,2024 at 23:55:43 UTC from IEEE Xplore. Restrictions apply.