Journal of Computer Security 00 (20xx) 1–42 1
DOI 10.3233/JCS-160557
IOS Press

Information flow analysis for a dynamically typed language with staged metaprogramming

Martin Lester [a,*], Luke Ong [a] and Max Schäfer [b]
[a] *Department of Computer Science, University of Oxford, Parks Road, Oxford, OX1 3QD, UK*
[b] *Semmle Limited, Blue Boar Court, 9 Alfred Street, Oxford, OX1 4EH, UK*

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 via eval. In order to deal 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 demonstrate the applicability of the analysis, we also present a general method for transforming a program using eval into one using staged metaprogramming.

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, information flow, JavaScript, static analysis

1. 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* [15].

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. Analysis of JavaScript programs is hindered by its many dynamic features, in particular **eval**, which allows execution of a string as program code.

We have developed a static information flow analysis for a dynamically typed, pure, functional language with stage-based metaprogramming [26]; 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 [18]. The analysis is based on the idea of extending a constraint-based formulation of the analysis 0CFA [47] with constraints to track information flow. We believe that the idea could be extended

*Corresponding author. E-mail: martin.lester@cs.ox.ac.uk.

to other CFA-style analyses (such as CFA2 [50]) 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. Finally, in order to demonstrate the applicability of the analysis to programs using string-based **eval**, we have developed an automated transformation that turns a program using **eval** into an equivalent one using template-based staged metaprogramming.

Our work stems from the observation that, while programmers may pass arbitrary strings to **eval**, they are usually constructed by string concatenation; this string concatenation is used to splice together code templates, which is exactly what is captured by the more principled formalism of staged metaprogramming. As Choi et al. note [7], directly analysing strings passed to **eval** is unlikely to be fruitful, as the range of strings may be infinite, in which case there is no obvious way to analyse their behaviour finitely. They also observe that the scoping behaviour of staged metaprogramming is similar to function abstraction and application. We build on this by extending 0CFA, an analysis that handles functions well, to deal with staged metaprogramming. 0CFA gives us the control flow of the program, which essentially corresponds to the direct flows of information. On top of this, we layer some extra constraints to track indirect flows. That gives us an information flow analysis for a language with staged metaprogramming.

Next, we seek a way to transform **eval** automatically into staged metaprogramming, so that we may truly analyse **eval**-using programs. There are two key ideas here. Firstly, we note that, for certain programming language grammars, it does not matter in which order parts of a string are parsed, so we are free to parse them in any order to build up code templates. Secondly, we note that we do not need to determine fully the behaviours of **eval** in a single pass: as long as we can feed some information about its behaviours back into our analysis, we can gradually build up a transformation of a program. However, in order to do this, we need to relate information about a staged program back to the **eval**-using program from which it originated. Combining these ideas, we obtain our algorithm for transforming **eval** into staged metaprogramming, which we are now able to analyse.

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 [30]. The presentation of the analysis in this paper extends that in CSF 2013 [28]. The details of the transformation are novel, but we outlined its key ideas in previous work [29].

The structure of the remainder of the paper is as follows. In Section 2, 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 3 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 4. Then, in Section 5, we describe how to apply the analysis to **eval**-using programs via a transformation to staged metaprogramming; again we discuss its performance on various examples. In Section 6, 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 7.

2. The language SLamJS

2.1. 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 [40], there

is comparatively little tackling dynamically typed languages. The semantics of JavaScript are complex and poorly understood [33], 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.

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 [41]. 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.

We have developed a simplified language called SLamJS, which we use to present our ideas. This allows us to work reasonably formally without being distracted by the complexities of full JavaScript. The language is heavily influenced by λ_{JS}, a "core calculus" for JavaScript [18]. 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 run-time. Unlike JavaScript, this is achieved using Lisp-style code quotations rather than code strings [7]. Recent work indicates that real-world usage of **eval** is often of a form that could be expressed using code quotations [24]. Thus analysis of programs with executable code quotations is an important step towards analysis of programs with executable code strings.

2.2. *Staged metaprogramming versus eval*

The **eval** construct of JavaScript takes a string, parses it and executes the result as program code. For example, `eval("x * 2")` evaluates to double the value of x in the current scope. Note that, as strings can be stored in variables and passed around, that scope might be different from the one in which the string was first defined.

In JavaScript, as in most languages, strings can be joined together using concatenation. Thus, if a string encodes a piece of program code, string concatenation can be used to splice together code templates. Consider, for example:

```
var f = function(z) { return 3 * z };
var y = "2";
var x = "f(" + y + ")";
eval(x);
```

This program constructs the code template `f(2)`, then executes it, returning $3 * 2 = 6$.

For several years, authors of static analyses for JavaScript argued that they could ignore it because it was rarely used or used only in trivial ways [16]. Their real reason was probably that analysis of such a powerful construct seemed utterly hopeless, particularly when considering the language's lack of protection mechanisms, as it allows arbitrary behaviours to result from an unstructured data value.

In contrast, the language Lisp allows programs to construct code templates as data values, splice them together and run the resulting code. We refer to these features collectively as *staged metaprogramming*. Following Kim et al. [26], we can add these features to a programming language with three constructs:

- **box** e turns the expression e into a code value; it does not evaluate e.

- **run** e evaluates the code value e.
- **unbox** e may only occur inside a **box** expression. It forces evaluation of e; the resulting code value is spliced into the surrounding code template.

The values in a program using these constructs and the steps in its execution can be stratified into numbered *stages*. Programs that do not use these constructs execute entirely at stage 0. The contents of a **box** expression at the top level of a program exist at stage 1, but **box** can be nested arbitrarily deeply, creating expressions at any stage $n > 0$, with each nesting being one stage higher. However, when **unbox** occurs within **box**, its contents are considered to be one stage lower; that is, they are at the same stage as the expression containing the **box**.

The constructs **box**, **unbox** and **run** correspond to the backquote, comma and eval operators of Lisp. Adding them to JavaScript, the previous example could be written as:

```
var f = function(z) { return 3 * z };
var y = box(2);
var x = box(f(unbox(y)));
run(x);
```

Our goal is to produce an information flow analysis for a JavaScript-like language extended with these constructs, then show how to transform an example like the former into the latter, so that we may apply our analysis to the former.

Roughly speaking, **box** and **unbox/run** act like function abstraction and application, except that they use a dynamic (instead of static) scoping discipline. This intuition is made more precise in Choi et al.'s work on static analysis of staged programs [7], where staging constructs are translated into function abstraction and application. However, we will be working directly on the staged language.

2.3. Syntax and semantics of SLamJS

2.3.1. 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. [7]. The construct **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 [22,26] 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.

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

Fig. 1. Syntax of SLamJS.

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

Fig. 2. Evaluation contexts.

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 0. Other constructs may be values at higher stages (v^{n+1}, v^{n+2} for $n \geqslant 0$), provided that their subexpressions are values at the appropriate stage. We generally omit the stage superscript for values of stage 0 (writing v instead of v^0).

2.3.2. 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 $\overset{n}{\rightarrow}$, 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 an expression at stage m with a hole at stage n inside it. 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 evaluation contexts is given in Fig. 2. Note that the contents of **box** exist at a higher stage than the enclosing expression, while those

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

Fig. 3. Environment propagation rules.

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

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

Fig. 4. Evaluation under a context and proper reduction rules.

of **unbox** exist at a lower stage. Consequently, an expression at stage $m > 0$ can contain an **unbox** expression with contents at a stage $m' < m$ with a hole at stage $n < m$.

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). Almost all the proper reductions occur only at stage 0. The exception is (Unbox), which occurs only at stage 1; this (combined with the stratification of evaluation contexts by stage) prevents **unbox** from being reduced other than within an enclosing **box** code value. That is, **unbox** can only ever splice code into an enclosing piece of code; in order to bring code to stage 0 and execute it, **run** must be used.

The environment propagation reductions control variable scoping within the language. Note that explicit substitutions only apply at stage 0, hence (x, ρ) evaluates to x at level $n + 1$ without looking up x in ρ. Furthermore, observe that $(\mathbf{run}\, e, \rho)$ pushes its environment into e, allowing boxed code values to capture variables from outside. Only environment propagation reductions may occur at higher stages; these reductions implement dynamic variable scoping for stages above 0.

The proper reduction rules are also quite standard [7], except for the field access rules, which are designed to be similar to JavaScript semantics.

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 $\overset{m}{\rightarrow}$, namely: $C_n^m\langle e\rangle \overset{m}{\rightarrow} C_n^m\langle e'\rangle$ if $e \overset{n}{\dashrightarrow} e'$. That is, an expression at stage m can only ever be evaluated at stage m, but its evaluation may involve reductions $\overset{n}{\dashrightarrow}$ with $n > m$. In particular, a complete program is usually evaluated entirely at stage 0, but its evaluation involves reductions within the program at higher stages. However, the only reductions at stages above 0 are (UNBOX) (at stage 1 only) and environment propagation rules (at all stages). We write $\overset{\square}{\rightarrow}$ for the union over all m of $\overset{m}{\rightarrow}$, and $\overset{\square}{\rightarrow}^*$ 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.

$$
\begin{aligned}
&\big(\textbf{if}(\textbf{true})\{\textbf{false}\}\,\textbf{else}\{1\}, \epsilon\big)\\
&\overset{0}{\rightarrow} \textbf{if}\big((\textbf{true}, \epsilon)\big)\big\{(\textbf{false}, \epsilon)\big\}\,\textbf{else}\big\{(1, \epsilon)\big\}\\
&\overset{0}{\rightarrow} \textbf{if}(\textbf{true})\big\{(\textbf{false}, \epsilon)\big\}\,\textbf{else}\big\{(1, \epsilon)\big\}\\
&\overset{0}{\rightarrow} (\textbf{false}, \epsilon) \overset{0}{\rightarrow} \textbf{false}
\end{aligned}
$$

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.

$$
\begin{aligned}
&\big(\textbf{run}\,\big(\textbf{box}\,\big(\textbf{if}\big(\textbf{unbox}\,\big(\textbf{box}\,(\textbf{true})\big)\big)\{\textbf{false}\}\,\textbf{else}\{1\}\big)\big), \epsilon\big)\\
&\overset{0}{\rightarrow} \textbf{run}\,\big(\textbf{box}\,\big(\textbf{if}\big(\textbf{unbox}\,\big(\textbf{box}\,(\textbf{true})\big)\big)\{\textbf{false}\}\,\textbf{else}\{1\}\big), \epsilon\big)\,\textbf{in}\,\epsilon\\
&\overset{0}{\rightarrow}{}^* \textbf{run}\,\big(\textbf{box}\,\big(\textbf{if}\big(\textbf{unbox}\,\big(\textbf{box}\,(\textbf{true})\big)\big)\big\{(\textbf{false}, \epsilon)\big\}\,\textbf{else}\big\{(1, \epsilon)\big\}\big)\big)\,\textbf{in}\,\epsilon\\
&\overset{0}{\rightarrow} \textbf{run}\,\big(\textbf{box}\,\big(\textbf{if}(\textbf{true})\big\{(\textbf{false}, \epsilon)\big\}\,\textbf{else}\big\{(1, \epsilon)\big\}\big)\big)\,\textbf{in}\,\epsilon \quad \big(\text{as: } \textbf{unbox}\,\big(\textbf{box}\,(\textbf{true})\big) \overset{1}{\dashrightarrow} \textbf{true}\big)\\
&\overset{0}{\rightarrow}{}^* \textbf{run}\,\big(\textbf{box}\,\big(\textbf{if}(\textbf{true})\{\textbf{false}\}\,\textbf{else}\{1\}\big)\big)\,\textbf{in}\,\epsilon\\
&\overset{0}{\rightarrow} \big(\textbf{if}(\textbf{true})\{\textbf{false}\}\,\textbf{else}\{1\}, \epsilon\big)\\
&\overset{0}{\rightarrow} \textbf{if}(\textbf{true}, \epsilon)\big\{(\textbf{false}, \epsilon)\big\}\,\textbf{else}\big\{(1, \epsilon)\big\}\\
&\overset{0}{\rightarrow} \textbf{if}(\textbf{true})\big\{(\textbf{false}, \epsilon)\big\}\,\textbf{else}\big\{(1, \epsilon)\big\} \overset{0}{\rightarrow} (\textbf{false}, \epsilon) \overset{0}{\rightarrow} \textbf{false}
\end{aligned}
$$

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 y during evaluation.

$$
\begin{aligned}
&\big(\big(\big(\textbf{fun}(x)\big\{\big(\textbf{fun}(y)\{\textbf{run}\,x\}\big)\big\}\big)(\textbf{box}\,y)\big)(\textbf{true}), \epsilon\big)\\
&\overset{0}{\rightarrow}{}^*\big(\textbf{fun}(y)\{\textbf{run}\,x\}, \langle x \mapsto \textbf{box}\,y\rangle\big)(\textbf{true})\\
&\overset{0}{\rightarrow} \big(\textbf{run}\,x, \langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\,y\rangle\big)
\end{aligned}
$$

$$\xrightarrow{0} \textbf{run}\left(x, \langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\ y\rangle\right) \textbf{in}\ \langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\ y\rangle$$
$$\xrightarrow{0} \textbf{run}\ (\textbf{box}\ y)\ \textbf{in}\ \langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\ y\rangle$$
$$\xrightarrow{0} \left(y, \langle y \mapsto \textbf{true}, x \mapsto \textbf{box}\ y\rangle\right)$$
$$\xrightarrow{0} \textbf{true}$$

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 [26], which makes reasoning about programs harder.

2.4. 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 (**if**(l){h} **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 (**if**(h){**false**} **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* [1,39]. We also introduce new rules for lifting markers into their parent expressions to avoid losing information about dependencies. As an expression is executed according to the augmented semantics, these markers accumulate around the result, recording its dependencies. However, 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:

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

We extend contexts to allow evaluation within a marked expression:

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

We allow propagation of environments within marked expressions:

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

In Fig. 5 we introduce lifts to maintain a record of indirect flows. Lifts are not needed to record direct flows, as markers are part of values, so the markers will move wherever the values do. For example, in a record assignment $v_1[v_2] = v_3$, there are indirect flows from v_1 and v_2 to the resulting record, so the rules (Lift-WriteSel) and (Lift-WriteRec) are needed. However, there is no need for a lift rule to track the flow from v_3 (i.e., $v_1[v_2] = (\mathfrak{m} : v_3) \overset{0}{\dashrightarrow} (\mathfrak{m} : v_1[v_2] = v_3)$), since that flow is direct.

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

Fig. 5. Semantic rules for lifts.

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

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

Note how the markers H and L in the result indicate that it depends on the marked values (H : **true**) and (L : **false**).

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

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

Observe that the result depends on I because the function (I : (**fun**$(y)\{x\}$)) was used to compute it, but not on L, as (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 $\mathit{unmark}(e_1) = f_1$ and $f_1 \xrightarrow{n} f_2$, then for some e_2 such that $e_1 \xrightarrow{n}{}^{*} e_2$, we have $\mathit{unmark}(e_2) = f_2$:

$$
\begin{array}{ccc}
e_1 & \xrightarrow{\mathit{unmark}} & f_1 \\
\big\downarrow{\scriptstyle n}{\scriptstyle *} & & \big\downarrow{\scriptstyle n} \\
e_2 & \xrightarrow{\mathit{unmark}} & f_2
\end{array}
$$

The marked expression may require multiple evaluation steps because of the need to apply lift rules before it is possible to apply the relevant rule of the unmarked semantics.

The marked semantics introduces some nondeterminism, but only in the order of application of lift rules for records, where if both a record and its field selector are marked, then a lift rule may be applied on either first. However, this only affects the order in which markers accumulate around an expression during evaluation, not the identity of the markers or the value of the result.

3. Information flow analysis for SLamJS

3.1. Overview

Before we can define an information flow analysis, we need to define what information flow is. Following Pottier and Conchon [39], 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 3.2 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 3.3; mechanised correctness proofs in Coq are available online [30].

In Section 3.4, 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 [30].

Finally, in Section 3.5, we prove soundness of the information flow analysis. We also discuss its relationship with termination-insensitive noninterference.

3.2. Erasure and stability

3.2.1. Erasure and prefixes

We extend the language with a "hole" that behaves like an unbound variable:

Expressions	e	$::= \ldots \mid \_$
Values	v^n	$::= \ldots \mid \_$
Reductions		$(\_, \rho) \overset{n}{\dashrightarrow} \_$

That is, _ behaves like a stuck expression that cannot be evaluated.

Now for $M \subseteq \mathit{Marker}$, define the *M-retaining 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. The erasure operation captures the idea of a *low view* commonly used to reason about noninterference [39].

3.2.2. 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 $e_1 \xrightarrow{\natural}{}^* f$, where f contains no _, then $e_2 \xrightarrow{\natural}{}^* f$.

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 $e_1 \overset{n}{\dashrightarrow} e_2$ 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 $e_1 \xrightarrow{\natural}{}^* e_2$. Then for every $M \subseteq Marker$ such that $\lfloor e_2 \rfloor_M = e_2$, it follows that $\lfloor e_1 \rfloor_M \xrightarrow{\natural}{}^* \lfloor e_2 \rfloor_M$.*

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

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 $e_1 \xrightarrow{n} e \xrightarrow{\natural}{}^{k'} e_2$, aim to prove $\lfloor e_1 \rfloor_M \xrightarrow{\natural}{}^* e_2$. Assume by the induction hypothesis that $\lfloor e \rfloor_M \xrightarrow{\natural}{}^{k'} e_2$. 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 $\lfloor e \rfloor_M \xrightarrow{\natural}{}^{k'} e_2$. Now, applying Monotonicity, we get $\lfloor e_1 \rfloor_M \xrightarrow{\natural}{}^* e_2$.

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 $\lfloor e_1 \rfloor_M \xrightarrow{n} \lfloor e \rfloor_M \xrightarrow{\natural}{}^{k'} e_2$, as required. □

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

$$\big(\big(\big(\mathbf{fun}(x)\big\{\mathrm{I} : \big(\mathbf{fun}(y)\{x\}\big)\big\}\big)(\mathrm{H} : 1)\big)(\_), \epsilon\big) \xrightarrow{0}{}^* \big(\mathrm{I} : (\mathrm{H} : 1)\big)$$

That is, the result of evaluation is unchanged.

3.3. 0CFA for SLamJS

We use a context-insensitive, flow-insensitive control flow analysis (0CFA [47]) 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. 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.

3.3.1. Simple analysis

Following Nielson, Nielson and Hankin [36], 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 $Label \to \mathcal{P}(AbsVal)$ associating a set of abstract values with every program point; similarly, an abstract environment $\varrho : AbsVar \to \mathcal{P}(AbsVal)$ 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, top) 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

Abstract domains

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

Some rules for the 0CFA acceptability judgement

$\Gamma, \varrho \models k^\ell$ if $\lceil k \rceil \in \Gamma(\ell)$

$\Gamma, \varrho \models x^\ell$ if $\varrho(x) \subseteq \Gamma(\ell)$

$\Gamma, \varrho \models (\textbf{box}\, e)^\ell$ if $\Gamma, \varrho \models e$ and $\exists v \in \Gamma(\ell).\Gamma, \varrho \models v \approx \textbf{box}\, e$

$\Gamma, \varrho \models \left(\textbf{unbox}\, t^\ell\right)^{\ell_0}$ if $\Gamma, \varrho \models t^\ell$ and $\forall \texttt{BOX}\left(t'^{\ell'}\right) \in \Gamma(\ell).\Gamma\left(\ell'\right) \subseteq \Gamma(\ell_0)$

$\Gamma, \varrho \models \left(\textbf{if}\left(t_1^{\ell_1}\right)\left\{t_2^{\ell_2}\right\}\textbf{else}\left\{t_3^{\ell_3}\right\}\right)^{\ell_4}$ if $\Gamma, \varrho \models t_1^{\ell_1} \wedge \Gamma, \varrho \models t_2^{\ell_2} \wedge \Gamma, \varrho \models t_3^{\ell_3}$ and $\Gamma(\ell_2) \subseteq \Gamma(\ell_4) \wedge \Gamma(\ell_3) \subseteq \Gamma(\ell_4)$

The approximation judgement $\Gamma, \varrho \models v \approx t$ and the abstract value operation $\lceil k \rceil$

$\Gamma, \varrho \models \lceil k \rceil \approx k$ for any literal k

$\Gamma, \varrho \models \texttt{FUN}(x, e) \approx \textbf{fun}(x)\{e\}$

$\Gamma, \varrho \models \texttt{BOX}(e) \approx \textbf{box}\, e$

$\Gamma, \varrho \models \texttt{REC}\left(\ell'\right) \approx \left\{\overline{s : t^\ell}\right\}$ if $\forall i.\exists v_i \in \varrho\left(\ell'.s_i\right).\Gamma, \varrho \models v_i \approx t_i$

$\Gamma, \varrho \models v \approx t'$ if $\Gamma, \varrho \models v \approx t \wedge t^\ell \xrightarrow{n} t'^\ell$

$\Gamma, \varrho \models v \approx (t, \rho)$ if $\Gamma, \varrho \models v \approx t \wedge \Gamma, \varrho \models \rho$

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

$\lceil \textbf{null} \rceil = \texttt{NULL}$

$\lceil \textbf{undef} \rceil = \texttt{UNDEF}$

$\lceil b \rceil = \texttt{BOOL}$ for boolean b

$\lceil n \rceil = \texttt{NUM}$ for number n

$\lceil s \rceil = \texttt{STR}$ for string s

Fig. 6. Some details of the simple analysis.

$\texttt{FUN}(x, e)$, $\texttt{BOX}(e)$ and $\texttt{REC}(\ell)$ 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 $\texttt{proto}(\ell)_\varrho$ to be the smallest set $P \subseteq \mathit{Label}$ such that $\ell \in P$ and for every $p \in P$ and $\texttt{REC}(\ell') \in \varrho(p.\texttt{"\_proto\_"})$ also $\ell' \in P$.

The acceptability judgement is now defined using syntax-directed rules, some of which are shown in Fig. 6 (middle). The remaining rules, which are standard, are given in Appendix B.

We write t^ℓ to represent an expression of the syntactic form t, labelled with ℓ. (Alternatively, we write $\mathit{lbl}(e)$ to mean the label of expression e.) 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 $\Gamma, \varrho \models \nu \approx \textbf{box}\, e$. This judgement holds if $\nu = \texttt{BOX}(e)$, 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 $\texttt{BOX}(e')$ 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. 6 (bottom). Note that ρ here is the concrete environment from the concrete semantics (in Fig. 1), which maps variable names to concrete values during execution; it is not the abstract environment ϱ, which maps abstract variable names to sets of abstract values. The concrete environment plays no role in the analysis, other than its proof of correctness.

The rule for **unbox** t^ℓ is surprisingly simple: all that is required is that, for any abstract value $\texttt{BOX}(t'^{\ell'})$ that the analysis thinks can flow into ℓ, every abstract value flowing into its body $t'^{\ell'}$ 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 (Simple 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 [30]. As an overview, the first step is to prove that if ${t_1}^{\ell_1} \xrightarrow{n} {t_2}^{\ell_2}$ and $\Gamma, \varrho \models {t_1}^{\ell_1}$, then $\Gamma(\ell_2) \subseteq \Gamma(\ell_1)$. This is done by proving the corresponding result for reductions by induction over the rules for $\dashrightarrow^n$, then (as evaluation is reduction under a context) by induction over the structure of contexts. Next the main theorem can be proved, again by induction over the derivation of reductions, followed by induction over the structure of contexts. Most of the cases in the proof are straightforward, but the large number of reduction rules in the language and the need to track carefully the indices on contexts make the proof somewhat involved.

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 $\texttt{BOX}(e)$ and $\texttt{FUN}(e)$ 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.

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

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

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

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

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

3.3.2. *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 (Fig. 7, top) and various operations on it (Fig. 7, middle). 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 *abstract frames* Λ, 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 *height* of an abstract context is the level of its topmost abstract frame; that is, one less than the total number of frames in the context.

Having enhanced the analysis by recording where variables are bound, we can use this information to improve the precision of our abstract environment ϱ, which is now a binary function. For a label ℓ labelling the body of a function abstraction with parameter x, $\varrho(\ell, x)$ overapproximates all values this parameter may be bound to in any invocation of the function. Similarly, for ℓ labelling the body of a **box** expression, $\varrho(\ell, x)$ overapproximates the values x may have in any evaluation of that body. Finally, for ℓ labelling a record, $\varrho(\ell, s)$ overapproximates all values that may be stored in field s of that record.

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.

Abstract domains

Abstract values	$v \in \mathit{AbsVal} ::= \mathtt{NULL} \mid \mathtt{UNDEF} \mid \mathtt{BOOL} \mid \mathtt{NUM} \mid \mathtt{STR} \mid \mathtt{FUN}(x, e) \mid \mathtt{BOX}(e) \mid \mathtt{REC}(\ell)$
Abstract caches	$\Gamma : \mathit{Label} \to \mathcal{P}(\mathit{AbsVal})$
Abstract environments	$\varrho : \mathit{Label} \times (\mathit{Name} \uplus \mathit{String}) \to \mathcal{P}(\mathit{AbsVal})$
Abstract frames	$\Lambda : \mathit{Name} \to \mathit{Label}$
Abstract contexts	$\Xi ::= \Lambda \mid \Xi + (\ell, \Lambda)$

Operations on abstract contexts

Context height	$\mathit{height}\left(\Lambda_0 + (\ell_1, \Lambda_1) + \cdots + (\ell_n, \Lambda_n)\right) \stackrel{\text{def}}{=} n$
Context extension	$\left(\Xi + (\ell, \Lambda)\right)\left[x \mapsto \ell'\right] \stackrel{\text{def}}{=} \left(\Xi + \left(\ell, \Lambda\left[x \mapsto \ell'\right]\right)\right)$
Context base replacement	$\Lambda_0 + (\ell_1, \Lambda_1) + \cdots + (\ell_n, \Lambda_n) \restriction \Lambda' \stackrel{\text{def}}{=} \Lambda' + (\ell_1, \Lambda_1) + \cdots + (\ell_n, \Lambda_n)$
Context as partial function	$\left(\Xi + (\ell, \Lambda)\right)(x) \stackrel{\text{def}}{=} \begin{cases} \Lambda(x) & \text{if } x \in \mathrm{dom}(\Lambda) \\ \ell & \text{otherwise} \end{cases}$
Hence:	$\Xi \restriction (x) = \begin{cases} \Lambda(x) & \text{if } \mathit{height}\ (\Xi) = 0 \\ \Xi(x) & \text{otherwise} \end{cases}$

Some rules for the 0CFA acceptability judgement

$\Gamma, \varrho, \Xi \models k^\ell$	if $\lceil k \rceil \in \Gamma(\ell)$
$\Gamma, \varrho, \Xi \models x^\ell$	if $x \notin \mathrm{dom}(\Xi) \vee \varrho\left(\Xi(x), x\right) \subseteq \Gamma(\ell)$
$\Gamma, \varrho, \Xi \models \left(\mathbf{box}\, t^\ell\right)^{\ell_0}$	if $\Gamma, \varrho, \Xi + (\ell, \epsilon) \models t^\ell$ and $\exists v \in \Gamma(\ell).\Gamma, \varrho \models v \approx \mathbf{box}\, t^\ell$
$\Gamma, \varrho, \Xi \models \left(\mathbf{unbox}\, t^\ell\right)^{\ell_0}$	if $\exists \Xi', \ell_1, \Lambda.\Xi = \Xi' + (\ell_1, \Lambda)$ and $\Gamma, \varrho, \Xi' \models t^\ell$ and $\forall \mathtt{BOX}\left(t'^{\ell'}\right) \in \Gamma(\ell).\left(\forall x.\varrho\left(\Xi(x), x\right) = \varrho\left(\ell', x\right)\right) \wedge \Gamma\left(\ell'\right) \subseteq \Gamma(\ell_0)$
$\Gamma, \varrho, \Xi \models \left(\mathbf{if}\left(t_1^{\ell_1}\right)\left\{t_2^{\ell_2}\right\}\mathbf{else}\left\{t_3^{\ell_3}\right\}\right)^{\ell_4}$	if $\Gamma, \varrho, \Xi \models t_1^{\ell_1} \wedge \Gamma, \varrho, \Xi \models t_2^{\ell_2} \wedge \Gamma, \varrho, \Xi \models t_3^{\ell_3}$ and $\Gamma(\ell_2) \subseteq \Gamma(\ell_4) \wedge \Gamma(\ell_3) \subseteq \Gamma(\ell_4)$

Fig. 7. Some details of the improved analysis.

While the change is conceptually simple, the rules are now syntactically somewhat more complex. A selection are shown in Fig. 7 (bottom); the rest are in Appendix B. Observe that they are structurally very similar to those for the simple analysis. As for the simple analysis, we demonstrate correctness of the improved analysis by proving its coherence with evaluation:

Theorem 3 (Improved CFA Coherence)**.** *If* $\Gamma, \varrho, \Xi \models e$ *with* Ξ *of height* n*, and* $e \xrightarrow{n} e'$*, then* $\Gamma, \varrho, \Xi \models e'$*.*

Once again, a full formalisation in Coq is available in the online supporting material.

Example 8. Consider the following expression, in which the variable x is bound twice:

$$\left(\left(\left(\mathbf{fun}(x)\left\{\left(\mathbf{fun}(x)\left\{\left(\mathbf{run}(x)^0\right)^1\right\}\right)^2\right\}\right)^3\left((\mathbf{false})^4\right)\right)^5\left(\left(\mathbf{box}(x)^6\right)^7\right)\right)^8$$

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

$$0 \mapsto \{\texttt{BOX}(x^6)\} \qquad 1 \mapsto \{\texttt{BOX}(x^6)\} \qquad 2 \mapsto \{\texttt{FUN}(x, (\textbf{run}(x)^0)^1)\}$$
$$3 \mapsto \{\texttt{FUN}(x, (\textbf{fun}(x)(\textbf{run}(x)^0)^1)^2)\} \qquad 4 \mapsto \{\texttt{BOOL}\} \qquad 5 \mapsto \{\texttt{FUN}(x, (\textbf{run}(x)^0)^1)\}$$
$$6 \mapsto \{\texttt{BOX}(x^6)\} \qquad 7 \mapsto \{\texttt{BOX}(x^6)\} \qquad 8 \mapsto \{\texttt{BOX}(x^6)\}$$

and for ϱ:

$$(1, x) \mapsto \{\texttt{BOX}(x^6)\} \qquad (2, x) \mapsto \{\texttt{BOOL}\} \qquad (6, x) \mapsto \{\texttt{BOX}(x^6)\}$$

which correctly shows that the result of evaluation (labelled 8) is the code value **box** x.

3.4. 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 $\rightsquigarrow$:

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

Because 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. This is similar to Rushby's interference relation [42], but whereas his relation describes a security policy by specifying which parts of a system are permitted to interact, ours describes the behaviour of a system by stating which parts of a system might interact.

The constraints on $\rightsquigarrow$ between labels, variable names and markers are split into direct flows (written $x \rightsquigarrow\!\!\!\!\cdot\ y$) and indirect flows (written $x \rightsquigarrow\!\!\!\!\!\!\cdot\ y$). Both denote the same constraint on $\rightsquigarrow$, namely $x \rightsquigarrow y$, but we list them separately for clarity of exposition. There is otherwise no practical difference between them with regard to the resulting analysis. Note that if we interpret $x \rightsquigarrow\!\!\!\!\cdot\ y$ and $x \rightsquigarrow\!\!\!\!\!\!\cdot\ y$ as (elements of) relations and define $\rightsquigarrow = \rightsquigarrow\!\!\!\!\cdot \cup \rightsquigarrow\!\!\!\!\!\!\cdot$, then $\rightsquigarrow$ satisfies the constraints.

We say that $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} e$ if $\Gamma, \varrho \models e$ and the conditions in Fig. 8 hold. As Γ, ϱ and $\rightsquigarrow$ are constant throughout the definition, we abbreviate $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} e$ to $\models_{\text{IF}} e$ 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 [30].

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

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

Theorem 4 (Information Flow Coherence)**.** *If we have* $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} e_1$ *and also* $e_1 \overset{m}{\to} e_2$*, then* $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} e_2$*. Furthermore,* $\mathit{lbl}(e_2) \rightsquigarrow^* \mathit{lbl}(e_1)$*.*

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

Fig. 8. Rules for the judgment $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} e$, which generates information flow constraints on $\rightsquigarrow$.

Proof. *Sketch*: Unfolding the definition of $\xrightarrow{m}$, we let $e_1 = C_n^m\langle t_1^{\ell_1}\rangle$ 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 $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t_1^{\ell_1}$ and hence, applying Lemma 2, $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t_2^{\ell_2}$, with $\ell_2 \rightsquigarrow^* \ell_1$. 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 $\ell_2 \rightsquigarrow^* \ell_1$, they must be satisfied in the conclusion, giving $\Gamma, \varrho \rightsquigarrow \models_{\text{IF}} C_n^m\langle t_2^{\ell_2}\rangle$ as required.

The claim that $lbl(e_2) \rightsquigarrow^* lbl(e_1)$ is trivial for all non-empty contexts, as $lbl(e_2) = lbl(e_1)$. 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 9. Recall once more Example 5. Using the results of 0CFA, our implementation generates the relations $\rightsquigarrow$ and $\looparrowright$ as depicted in Fig. 9.

Setting $\hookrightarrow = \rightsquigarrow \cup \looparrowright$, we have H $\hookrightarrow^*$ 9 and I $\hookrightarrow^*$ 9 and L $\not\hookrightarrow^*$ 9. As expected, this means the result (labelled 9) has information flows from H and I, but not L.

3.5. *Information flow soundness*

The information flow relation $\rightsquigarrow$ expresses which flows might occur (locally) during a single step of execution of an expression. We now show how this relates to the flows that might occur (globally) over a sequence of execution steps that terminates in a value.

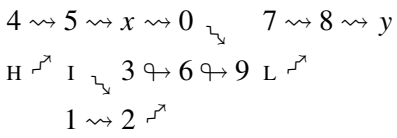

Fig. 9. Information flow constraints for Example 5.

Theorem 5 (Information Flow Soundness)**.** *Suppose* $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t^{\ell}$. *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. (We argue using the judgment for the simple analysis, but the same argument holds for the improved analysis.) First show that $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} v^{\ell'}$ with $\ell' \rightsquigarrow^{*} \ell$. Argue by a simple induction over the derivation of $t^{\ell} \xrightarrow{\square} {}^{*} v$.

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

Inductive step: Assume that $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} e_1$ and $\mathit{lbl}(e_1) \rightsquigarrow^{*} \ell$, with $e_1 \xrightarrow{\square} e_2$ the next step in the derivation. Apply Theorem 4 to show that $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} e_2$ and $\mathit{lbl}(e_2) \rightsquigarrow^{*} \mathit{lbl}(e_1)$; hence $\mathit{lbl}(e_2) \rightsquigarrow^{*} \ell$.

Now we have $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} v$ 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 $\models_{\text{IF}}$ 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 classical security property termination-insensitive 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 substitute subexpressions to model 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 $\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t^{\ell}$ and $\text{H} \not\rightsquigarrow^{*} \ell$. Further, require that $\xrightarrow{\text{low}}$ and $\xrightarrow{\text{high}}$ satisfy the following conditions:

$$\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t^{\ell} \wedge t \xrightarrow{\text{low}} t' \implies \Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t'^{\ell}$$
$$\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t^{\ell} \wedge t \xrightarrow{\text{high}} t' \implies \Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t'^{\ell}$$
$$\Gamma, \varrho, \rightsquigarrow \models_{\text{IF}} t^{\ell} \wedge t \xrightarrow{\text{high}} t' \implies \lfloor t \rfloor_{\{\text{L}\}} = \lfloor t' \rfloor_{\{\text{L}\}}$$

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 $t_2^{\ell} \overset{\Box}{\rightarrow}{}^* u^{\ell'}$. That is, the low output u is independent from the values of the high inputs for t selected using $\overset{\text{high}}{\longrightarrow}$.

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

The conditions on $\overset{\text{low}}{\longrightarrow}$ and $\overset{\text{high}}{\longrightarrow}$ seem reasonable. As an example, $\overset{\text{low}}{\longrightarrow}$ and $\overset{\text{high}}{\longrightarrow}$ 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. That is, taking $\overset{\text{low}}{\longrightarrow}$ to be the equivalence relation on expressions that differ only in the values of constants marked L satisfies the conditions; similarly for $\overset{\text{high}}{\longrightarrow}$ and constants marked H. In this sense, $\overset{\text{low}}{\longrightarrow}$ and $\overset{\text{high}}{\longrightarrow}$ play the roles of the usual low view and high view equivalence relations $\approx_{\text{L}}$ and $\approx_{\text{H}}$ (common when considering program stores in an imperative setting), at least for expressions that are yet to be executed.

Note that, as our information flow relation $\rightsquigarrow$ expresses all local information flows within a program, its applications need not be restricted to transitive noninterference. It could also be used to reason about intransitive noninterference policies [42,49], in which some flows from H to L may be allowed, but only if they occur through a specified route, which may represent a secure communication channel or declassification.

4. 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 [20]; 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 [30]. 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. [41] and discussed by Jensen et al. [24].

Example 10. *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 11. *Depends on*: H, I, L. *Depends (improved)*: H, L.

$$\begin{array}{l}\textbf{let}\ ctrue = \textbf{fun}(x)\{\textbf{fun}(y)\{x\}\}\ \textbf{in}\\ \textbf{let}\ cif = \textbf{fun}(x)\{\textbf{fun}(y)\{\textbf{fun}(z)\{(x(y))(z)\}\}\}\ \textbf{in}\\ ((cif(\text{H} : ctrue))(\text{L} : \textbf{false}))(\text{I} : 1))\end{array}$$

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*.

Example 12. *Depends on*: L.

$$\begin{array}{l}\textbf{let}\ x = \textbf{if}(\textbf{true})\{\textbf{box}\ f\}\ \textbf{else}\{\textbf{box}\ g\}\ \textbf{in}\\ \textbf{let}\ f = \textbf{fun}(y)\{1\}\ \textbf{in}\\ \textbf{let}\ g = \textbf{fun}(z)\{\text{L} : \textbf{true}\}\ \textbf{in}\\ \textbf{run}\ (\textbf{box}\ ((\textbf{unbox}\ x)(\text{H} : \textbf{undef})))\end{array}$$

This is modelled on the following JavaScript usage [24]:

```
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 (H : **undef**), so the result does not depend on H; our analysis correctly identifies this.

Example 13. *Depends on*: H, L. *Depends (improved)*: L.

$$\begin{array}{l}\textbf{let}\ c = \textbf{box}\ x\ \textbf{in}\\ \textbf{let}\ x = \text{L} : 1\ \textbf{in}\\ \textbf{let}\ eval = \textbf{fun}(b)\{\textbf{run}\ b\}\ \textbf{in}\\ \textbf{let}\ x = \text{H} : 2\ \textbf{in}\\ eval(c)\end{array}$$

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 14. *Depends on*: H, I, L.

$$\begin{array}{l}\textbf{let}\ i = \text{I} : \{\texttt{"\_proto\_"} : \textbf{null}, \texttt{"}x\texttt{"} : (\text{H} : 1), \texttt{"}y\texttt{"} : (\text{L} : 2)\}\ \textbf{in}\\ \textbf{let}\ s = \textbf{fun}(id)\{\textbf{let}\ f = \textbf{box}\ (i[\textbf{unbox}\ id])\ \textbf{in run}\ f\}\ \textbf{in}\\ s(\textbf{box}\ \texttt{"}y\texttt{"})\end{array}$$

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 H; this could be addressed refining our abstract value domain.

Example 15. *Depends on*: H.

$$\begin{array}{l}\textbf{let}\, fst = \textbf{fun}(x)\{\textbf{fun}(y)\{x\}\}\ \textbf{in}\\ \textbf{let}\ f = \textbf{if}(\textbf{false})\{fst\}\ \textbf{else}\{\textbf{box}\, fst\}\ \textbf{in}\\ \textbf{let}\, x = (\text{H} : 1)\ \textbf{in}\\ \textbf{let}\, y = (\text{L} : \textbf{true})\ \textbf{in}\\ \textbf{if}(\textbf{typeof}\ f = \texttt{"function"})\{(f(x))(y)\}\\ \textbf{else}\{\textbf{run}\,(\textbf{box}\,((((\textbf{unbox}\ f)(x))(y))))\}\end{array}$$

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 16. *Depends on*: H, L. *Depends (improved)*: L.

$$\begin{array}{l}\textbf{let}\, pair = \textbf{fun}(x)\{\textbf{fun}(y)\{\textbf{fun}(z)\{\textbf{run}\, z\}\}\}\ \textbf{in}\\ \textbf{let}\, fst = \textbf{fun}(z)\{z(\textbf{box}\, x)\}\ \textbf{in}\\ \textbf{let}\, snd = \textbf{fun}(z)\{z(\textbf{box}\, y)\}\ \textbf{in}\\ \textbf{let}\, bp = \textbf{box}\,((pair(\text{L} : (\textbf{box}\,(1))))(\text{H} : (\textbf{box}\,(\textbf{true}))))\ \textbf{in}\\ \textbf{let}\, boxfst = \textbf{box}\,((fst)(\textbf{unbox}\, bp))\ \textbf{in}\\ \textbf{run}\,(\textbf{run}\,(boxfst))\end{array}$$

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 17. *Depends on*: H.

$$\begin{array}{l}\textbf{fun}(n)\{(\textbf{fun}(x)\{(x(x))(n)\})\\ (\textbf{fun}(x)\{\textbf{fun}(y)\{\textbf{if}(y = 0)\{\textbf{true}\}\ \textbf{else}\{(x(x))(sub(y, 1))\}\}\})\\ \}(\text{H} : 5)\end{array}$$

This program loops n times (where n is $(\text{H} : 5)$ 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 [43].

Example 18. *Depends on*: L

$$\begin{array}{l}\textbf{let}\, fst = \textbf{fun}(x)\{\textbf{fun}(y)\{x\}\}\ \textbf{in}\\ \textbf{let}\, a = \textbf{box}\, x\ \textbf{in}\\ \textbf{let}\, b = \textbf{box}\,(\textbf{fun}(x)\{\textbf{fun}(y)\{fst(\textbf{unbox}\, a)(y)\}\})\ \textbf{in}\\ (\textbf{run}\, b)(\text{L} : 1)(\text{H} : 2)\end{array}$$

This program, based on an example from Choi et al. [7], 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 19. *Depends on*: L, H.

$$\begin{array}{l}\textbf{let}\, fst = \textbf{fun}(x)\{\textbf{fun}(y)\{x\}\}\ \textbf{in}\\ \textbf{let}\, a = \textbf{fun}(p)\{p[\texttt{"x"}]\}\ \textbf{in}\\ \textbf{let}\, b = (\textbf{fun}(h)\{\textbf{fun}(p)\{\textbf{fun}(x)\{\textbf{fun}(y)\{\\ fst(h((p[\texttt{"x"}] = x)[\texttt{"y"}] = y))(y)\}\}\}\})(a)\ \textbf{in}\\ b(\{\texttt{"\_proto\_"} : \textbf{null}\})(\text{L} : 1)(\text{H} : 2)\end{array}$$

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.

Example 20. *Depends on*: L, H. *Depends (improved)*: L.

$$\begin{array}{l}\textbf{let}\, blank = \textbf{fun}(get)\{get(\textbf{null})(\textbf{null})\}\ \textbf{in}\\ \textbf{let}\, getx = \textbf{fun}(x)\{\textbf{fun}(y)\{x\}\}\ \textbf{in}\\ \textbf{let}\, gety = \textbf{fun}(x)\{\textbf{fun}(y)\{y\}\}\ \textbf{in}\\ \textbf{let}\, setx = \textbf{fun}(env)\{\textbf{fun}(newx)\{\\ \textbf{fun}(get)\{get(newx)(env(gety))\}\}\}\ \textbf{in}\\ \textbf{let}\, sety = \textbf{fun}(env)\{\textbf{fun}(newy)\{\\ \textbf{fun}(get)\{get(env(getx))(newy)\}\}\}\ \textbf{in}\\ \textbf{let}\, fst = \textbf{fun}(x)\{\textbf{fun}(y)\{x\}\}\ \textbf{in}\\ \textbf{let}\, a = \textbf{fun}(p)\{p(getx)\}\ \textbf{in}\\ \textbf{let}\, b = (\textbf{fun}(h)\{\textbf{fun}(p)\{\textbf{fun}(x)\{\textbf{fun}(y)\{fst\,(h(sety(setx(p)(x))(y)))\,(y)\}\}\}\})(a)\ \textbf{in}\\ b(blank)(\text{L} : 1)(\text{H} : 2)\end{array}$$

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 $\mathcal{O}(n^3)$.

5. Transforming eval to staged metaprogramming

5.1. Overview

In order to demonstrate the applicability of the information flow analysis, we now describe the *Boxing Algorithm*, an algorithm for transforming a program that uses string-based **eval** into one that uses the **box** and **unbox** of staged metaprogramming.

Given a program e_1 that uses **eval**, the aim is to transform it into an equivalent program e_2 that instead uses staged metaprogramming. We can then meaningfully apply the information flow analysis of Section 3 to e_2. An overview of the algorithm is given in Fig. 10; a more detailed illustration of dataflow within the algorithm is shown in Fig. 11. Like most static analysis problems, the question of whether an **eval**-using program can be transformed into one using staged metaprogramming is undecidable, so we cannot hope to produce a sound algorithm that will always find a transformation when one exists.

The transformation is based on the assumption that an **eval**-using program uses string concatenation to join together well-formed code templates. Hence constant strings can be transformed into **box** expressions, uses of concatenation into **unbox** expressions, and **eval** into **run**. This assumption may be violated if, for example: the structure of concatenation in the program does not correspond closely enough to the

- *Input*: an **eval**-using program e_1.
- *Output*: a transformed staged program e_2.

While a fixed point has not been reached:

(1) Generate constraints α for e_1 ...
   - ... and (if not first iteration) any code previously generated in step 4.

(2) Solve constraints α ...
   - ... and (if not first iteration) any constraints γ previously generated in step 5.

(3) Search for a transformation.
   - If the search fails, terminate unsuccessfully.

(4) Splice **box** expressions into e_1 according to transformation, generating candidate e_2.
   - If (not first iteration and) candidate e_2 is the same as previous iteration's, a fixed point has been reached; terminate successfully and return e_2.

(5) Generate, solve and resolve staged constraints γ on spliced code.

Fig. 10. Control flow in the Boxing Algorithm.

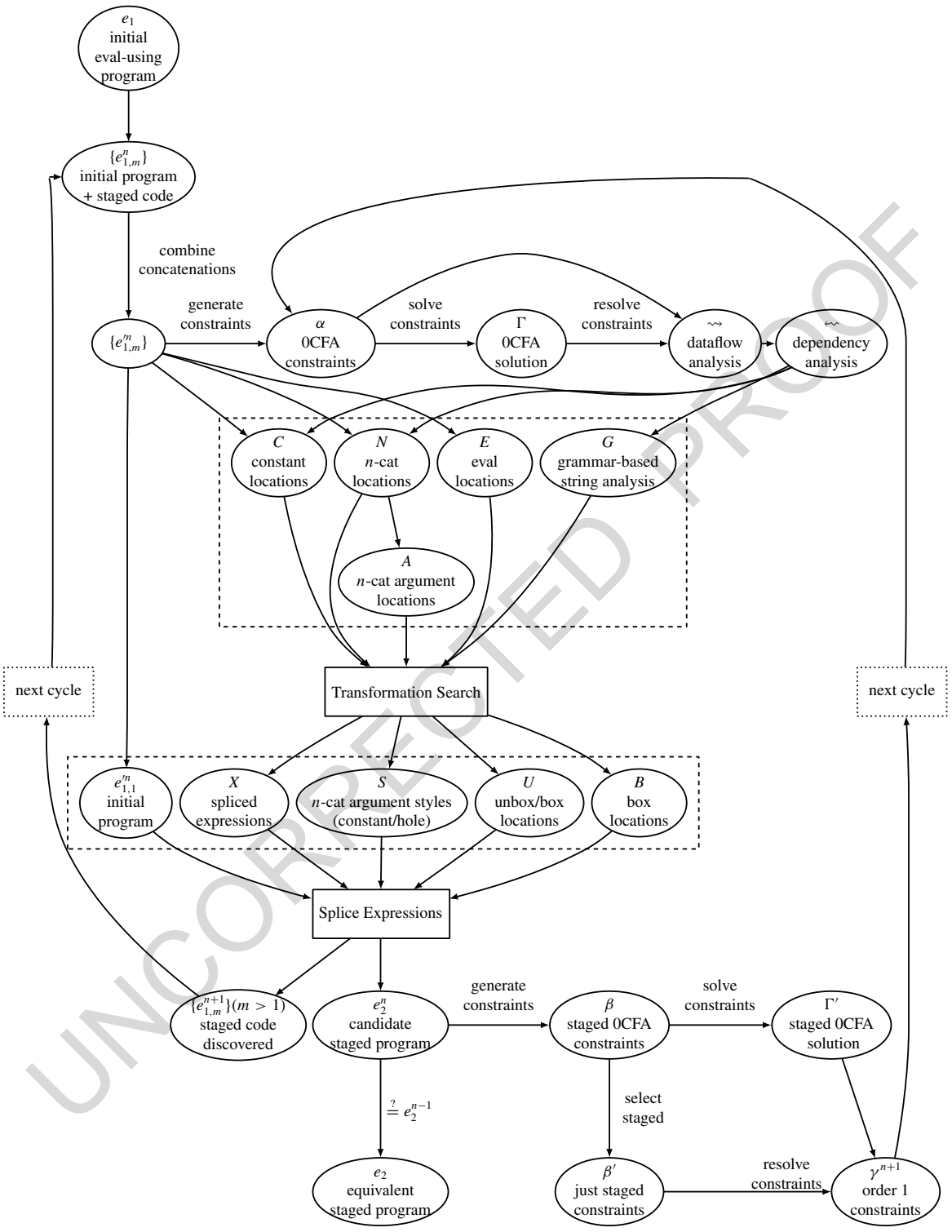

Fig. 11. Data flow in the Boxing Algorithm.

syntactic structure of expressions; the program makes decisions about flow control on the basis of the syntactic content of code strings; or the program uses substring operations to deconstruct code templates (as **unbox** only describes composition). In these cases, the transformation will fail with an error.

5.2. Key ideas

5.2.1. Prerequisites

The basic idea is that the algorithm will transform:

- code constants into **box** expressions;
- concatenation of code strings into splicing using **unbox**;
- **eval** into **run**.

For example:	becomes:
let $x =$ `"y"` **in**	**let** $x =$ **box** y **in**
eval x	**run** x
while:	becomes:
let $f =$ **fun**$(z)\{mul(3, z)\}$ **in**	**let** $f =$ **fun**$(z)\{mul(3, z)\}$ **in**
let $y =$ `"2"` **in**	**let** $y =$ **box** 2 **in**
let $x =$ `"f("` $+ y +$ `")"` **in**	**let** $x =$ **box**$\big(f($**unbox** $y)\big)$ **in**
eval x	**run** x

But in order to use it, certain conditions must hold:

(1) we need a sound dataflow analysis for the target language, including metaprogramming constructs;
(2) we need a string analysis for the target language that will produce a sound over-approximation of the string values that may occur at different program points or be bound to different variables;
(3) the language must be parseable using `lex` and `yacc` or similar tools.

We have already seen how to apply 0CFA to SLamJS, so we reuse that analysis to satisfy condition (1). As 0CFA over-approximates the flow control of a program with a regular graph, it is easy to extract a grammar-based string analysis from the results; that satisfies condition (2), although there are many techniques that could improve upon this [6]. The lexer and parser for SLamJS are implemented using OCamlLex and OCamlYacc, which satisfies condition (3). (INRIA's Prosecco has produced such an unambiguous grammar for JavaScript, so it meets the criteria for our algorithm to be applied.)

5.2.2. Building a sequence of program approximations

In most static analyses, the goal is to construct an over-approximation of the behaviours of a program. In contrast, the goal of the Boxing Algorithm is to produce a staged program that exactly captures the behaviours of the original **eval**-using program. Although we cannot immediately analyse the behaviour of a program that uses **eval**, if we under-approximate the behaviour of **eval** by supposing it does nothing, we can analyse the behaviour of other parts of the program and use this as a starting point for building a sequence of increasingly accurate approximations that will hopefully converge to a fixed point that has exactly the behaviour of the whole program.

In particular, any information supplied by the analysis about the program's behaviour prior to the first execution of **eval** will be sound. This includes information about the content of the first strings executed as code. By analysing this code, we can construct a new approximation that will either reveal new

information about the code executed, or show that we have enough information to capture the program's behaviour exactly.

We cannot simply analyse individual code strings, as there may be infinitely many possible such strings arising over all executions of the program. Fortunately, the staged metaprogramming analysis allows us to analyse the behaviour of possibly infinite sets of code values built using staged metaprogramming.

The approximation produced in the nth cycle of the fixed point process, if run, would behave the same as the original program up to the nth execution of **eval**. After that, the behaviours of the original and approximate program may diverge; in particular, the approximation may contain an odd mixture of staged metaprogramming and code string-based metaprogramming. If the algorithm reaches a fixed point, the transformed program, if executed, would behave the same as the original program.

5.2.3. Parsing expressions out-of-order

The Boxing Algorithm relies on using the existing parser for a language for part of the transformation; this makes the technique easily applicable to other languages. Typically, languages are parsed in two phases. The first phase, lexical analysis, splits the program text into a sequence of lexemes and transforms these into a sequence of tokens; the tool `lex` generates a *lexer* that does this using a deterministic finite state automaton. In the second phase, the actual parser turns the sequence of tokens into an abstract syntax tree; `yacc` generates a LALR (Look Ahead, Left-to-right, Rightmost derivation) *parser* to do this with a restricted form of pushdown automaton.

Normally, the lexer processes the characters in the program text in order from beginning to end. Similarly, the parser processes the token sequence in order from left to right (hence the second L in LALR). The Boxing Algorithm abuses these tools to process **eval**ed text out of order in fragments. Effectively, this gives us a finite way of parsing a string abstraction, which may encode infinitely many concrete strings of unbounded length. However, there is a price: we risk changing the meaning of the **eval**ed code. To avoid this (and hence keep the transformation sound), we must check that the lexing and parsing phases are unaffected by the change of order.

Parsing. In order to build an approximate analysis of an **eval**-using program, the algorithm transforms string operations on code values into uses of staged metaprogramming. (Note that the intermediate approximations may combine uses of **eval** with uses of staged metaprogramming.) The difficulty here is that, while an **eval**-using program might construct code strings using concatenation from left to right (or some other order), staged metaprogramming splices one well-formed expression into another, without respect for this order.

For example, consider a language (unlike SLamJS) with conventional arithmetic expressions. Using the usual precedence of arithmetic operators, the expression $1 + 2 * 3$ would evaluate to 7. We could use string concatenation to construct a string representation of this code and **eval** it:

$$\begin{array}{l}\textbf{let}\ x = \texttt{"1 + 2"}\ \textbf{in}\\ \textbf{let}\ y = \texttt{"3"}\ \textbf{in}\\ \textbf{eval}\big(x + \texttt{" * "} + y\big)\end{array}$$

We might be tempted to transform this into the following representation using staged metaprogramming:

$$\begin{array}{l}\textbf{let}\ x = \textbf{box}(1 + 2)\ \textbf{in}\\ \textbf{let}\ y = \textbf{box}\ 3\ \textbf{in}\\ \textbf{run}\big(\textbf{box}\big((\textbf{unbox}\ x) * (\textbf{unbox}\ y)\big)\big)\end{array}$$

But this would be incorrect, as it would change the bracketing of the expression from $1 + 2 * 3 = 1 + (2 * 3) = 7$ (according to the usual rules of precedence in arithmetic) to $(1 + 2) * 3 = 9$. (Arguably this may have been the intent of the author of the program.)

The problem arises because without the rules of precedence, the grammar of expressions in the language is ambiguous. The proposed transformation corresponds to one possible parsing of `1 + 2 * 3`, but not the one the language's parser would choose when following the usual rules of precedence.

If the language we wish to analyse is parseable using a *deterministic* context free grammar, we can parse fragments of program code out-of-order without changing the resulting expression. Hence we can safely transform a sequence of concatenations into the splicing of an expression into a template.

This is not an onerous requirement as, while `yacc` accepts ambiguous grammar specifications, it resolves the grammar ambiguity (perhaps arbitrarily) and produces a LALR parser for a more restricted, unambiguous grammar, which would itself be valid as a grammar specification.

Disambiguation in grammar specifications is often achieved by adding extra syntactic classes of expression. In the case of arithmetic expressions, there might be a class that represents only bracketed expressions or those free from addition, with multiplication only being permitted between expressions in this class. The example above might not be transformable in this case, as while x and y should clearly encode valid expressions, it is no longer permissible to multiply arbitrary expressions. In this case, replacing the final concatenation with $\texttt{"("} + x + \texttt{")} \ * \ \texttt{("} + y + \texttt{")"}$ would allow the transformation, as the multiplication would be between bracketed expressions. So we may be unable to transform some ostensibly reasonable programs because of the restriction to deterministic grammars, but it seems a reasonable sacrifice in order to ensure sound transformation of other programs.

We can easily modify the `yacc` grammar for SLamJS to support parsing of incomplete expressions containing holes by adding a new token named `HOLE`, representing a hole in an expression, and a new rule stating that `HOLE` is an expression. We write $\langle\bullet\rangle$ to mean a string that lexes to `HOLE`. In order to avoid misinterpreting text produced within a program as $\langle\bullet\rangle$, it ought to be treated as a special character that cannot occur textually within a program or be produced by the string operations within the language.

Consider again the second example at the start of Section 5.2.1. Having added $\langle\bullet\rangle$ and `HOLE` to the lexer and parser respectively, we can treat the string generated by the concatenation $\texttt{"f("} + y + \texttt{")"}$ as $\texttt{"f(}\langle\bullet\rangle\texttt{)"}$, which, as $\langle\bullet\rangle$ parses as an expression, also parses as an expression. In order to fill the hole, we also need to parse `"2"` as an expression. The algorithm's use of the parser is fairly straightforward: it still only parses complete expressions; the trick is that sometimes the parsing of their subexpressions occurs separately, those subexpressions having been replaced with holes. In contrast, other work on string analysis of code [12] tracks the parse stack produced by partially parsing code string fragments. This leads to a way of determining whether the code string must be a valid expression, but not what the content of that expression is.

Lexing. A further complication in parsing expressions out-of-order is that concatenation of code strings may change the boundaries and types of tokens produced during their lexical analysis.

This is unlikely to cause problems in SLamJS because the grammar is quite restrictive, but suppose we allowed function application to be written as juxtaposition of expressions without brackets, as in a typical functional language, and consider the following:

$$\begin{array}{l}\textbf{let}\ x = \textbf{if}(y)\{\texttt{"(g(3))"}\}\ \textbf{else}\{\texttt{"g(3)"}\}\ \textbf{in}\\ \textbf{let}\ z = \texttt{"f"} + x\ \textbf{in}\\ \textbf{eval}\ z\end{array}$$

We might be tempted to transform it to:

$$\begin{array}{l}\textbf{let}\, x = \textbf{if}(y)\{\textbf{box}\,(g(3))\}\,\textbf{else}\{\textbf{box}\,(g(3))\}\,\textbf{in}\\ \textbf{let}\, z = \textbf{box}\,(f(\textbf{unbox}\, z))\,\textbf{in}\\ \textbf{run}\, z\end{array}$$

But this would be wrong in the case where y is false. Depending on the value of y, the values of z in the original program and its tokenisations might be:

```
"f(g(3))"  ↦ VAR(f) LP VAR(g) LP INT(3) RP RP
"fg(3)"    ↦ VAR(fg) LP INT(3) RP
```

In transforming the concatenation assigned to z, we observe that x is not constant, so we tokenise it as `HOLE(x)`; the concatenation then tokenises as `VAR(f) HOLE(x)`. Treating `HOLE(x)` as a wildcard, this matches the first case, but not the second.

The problem in this case is that the use of the hole has changed the lexeme boundaries within the string and hence its tokenisation. Conceptually, we can view the lexer as a deterministic finite state transducer T that reads the string and emits tokens on lexeme boundaries. In order to avoid changing the tokenisation of code strings, we need to check that, whenever code strings x and y are concatenated, it is true that $T(x \cdot y) = T(x) \cdot T(y)$. (If it is not true, it might be deliberate on the part of the programmer, but is more likely to indicate an error or a fragile piece of code.)

This hides some details of the problem, as in practice tokens that are identical in the automaton model often carry some data that distinguishes them (as with `VAR(f)` and `VAR(fg)` in the above example). What we really need to check is that concatenation does not change the positions of lexeme boundaries in the code string.

Our solution is to look at the finite state transducer produced by the lexer generator in combination with the string analysis. Whenever we wish to treat a concatenation argument as a hole, we compute (an over-approximation to): the final states of the lexer having processed the last character of the hole string; and all possible first characters of the following argument. We then require that, for each possible final state: either it is one that emits a token (and hence the following character is irrelevant in how the hole string is lexed); or, for all possible following characters, the lexer immediately emits the same token. That is, each final state must correspond to a lexeme boundary. If the check fails, then the argument may not safely be treated as a hole.

5.2.4. *Constraint solution and resolution*

Recall that the staged metaprogramming analysis for SLamJS is formulated as a system of constraints. The constraints describe the abstract values that may arise when evaluating a particular subexpression of a program. Solution of the constraints yields a function Γ such that $\Gamma(\ell)$ is a sound over-approximation of the abstract values that may occur at program point ℓ during execution.

The majority of constraints are "order 0" constraints of the form $a \in \Gamma(\ell)$ or "order 1" constraints of the form $\Gamma(\ell_1) \subseteq \Gamma(\ell_2)$. But in order to express the behaviour of function and code values, we need "order 2" constraints of the form $\forall \texttt{BOX}(e^{\ell'}) \in \Gamma(\ell_1).\Gamma(\ell') \subseteq \Gamma(\ell_2)$.

It is possible (although not most efficient [37]) to find the smallest solution to these constraints iteratively using a fixed point computation: initialise each $\Gamma(\ell)$ to be empty; consider each constraint in turn, adding more abstract values to some relevant $\Gamma(\ell)$ until it is satisfied; repeat the process until no constraint adds new values (and hence every constraint is satisfied).

Note that, for a fixed Γ, an order 2 constraint can be expressed as a set of order 1 constraints and an order 1 constraint can be expressed as a set of order 0 constraints. Obviously, when solving the constraints iteratively, Γ is not fixed, but this observation that higher order constraints can be *resolved* to lower order ones becomes important when considering how to use staged constraints in an **eval**-based program.

5.2.5. Combining concatenations

The main premise of the Boxing Algorithm is that the syntactic structure of concatenation in reasonable **eval**-using programs will be similar to composition of templates via staged metaprogramming. However, they are unlikely to match exactly. In particular, if we consider an expression like $f(x)$ built with string concatenation by `"f(" + "x" + ")"`, the implicit bracketing of the concatenation is `("f(" + "x") + ")"`. As "$f($" is not a grammatically valid expression, we cannot represent this with staged metaprogramming.

The solution is to replace syntactically adjacent instances of binary concatenation in an expression with a single n-ary concatenation (or *n-cat*). This will not change the behaviour of the program in SLamJS, as concatenation is associative, although in other languages (such as full JavaScript) there may be subtleties arising from implicit string conversion or other peculiarities.

5.3. Cycle description

Let us now consider the steps in a single cycle of the fixed point process that produces a transformation. Each cycle operates on the original program (plus, in later cycles, fragments of staged code and associated constraints).

We begin by combining adjacent concatenations in the original program to get e_1'. Next, we run our original analysis on the program e_1' to obtain the relation $\rightsquigarrow$ that describes direct data flows between program points ℓ. If $\ell_1 \rightsquigarrow \ell_2$, then there is a direct flow from ℓ_1 to ℓ_2. We can view $\rightsquigarrow$ as an edge relation for a directed dataflow graph with program points as vertices. By reversing every edge in the graph, we obtain a dependency graph with edge relation $\leftsquigarrow$. We can use this to determine the program points from which code strings flow into **eval**s.

We also use the dataflow graph to produce a naive, grammar-based string analysis G. Essentially, we associate a non-terminal with every program point and introduce a production for every edge in the dependency graph. For string constant expressions, we add a production to the corresponding constants; concatenation operations are translated directly into concatenation in the grammar. This approach is outlined and developed further by Christensen et al. [8], considering in particular how to produce a reasonably precise but regular over-approximation to the strings generated at a program point in the presence of loops in the graph. But for the purposes of this transformation, we need only to determine whether a program point ℓ yields a constant string (and if so, what it is).

We are now able to determine which program points are candidates for transformation:

- E – all occurrences of **eval** within e_1';
- C – all constant strings on which an argument of an **eval** depends;
- N – all n-cats on which the argument of an **eval** depends;
- A – all arguments to n-cats in N.

E is determined syntactically from e_1'; the rest are determined using $\leftsquigarrow$.

For every path from an **eval** argument back to a program point that produces a code string, the goal is then to transform the expression at that program point into one that produces an equivalent staged

metaprogramming code value. If the code string produced is a constant and that constant parses correctly to an expression, then this is simply a matter of replacing it with the corresponding **box** expression. If the code string is produced by concatenation, it might not be constant. In this case, we attempt to turn it into a **box** template with **unbox** expressions to fill any holes in the template.

A code concatenation expression will consist of several subexpressions joined together. For each of these subexpressions, we must choose whether to treat it as a constant (which is obviously only possible if the subexpression is indeed constant, as determined by G), or to treat it as a hole. If we treat it as a constant, it becomes part of the code template, which we parse to produce the **box** expression. If we treat it as a hole, we introduce the obligation to transform any program point on a path from the subexpression that produces a code string.

Hence the process of choosing what and how to transform can be viewed as a depth-first search rooted at the occurrences of **eval**. If the search is successful, we produce:

- B – a set of program points where we may transform string constants into **box** expressions;
- U – a set of program points where we may transform string concatenations into **unbox** expressions;
- $X : (B \cup U) \to T$ – a mapping from transformed program points into code templates.
- $S : A \to \{h, c\}$ – a "style" specifying whether a concatenation argument should be turned into a *h*ole or treated as a *c*onstant.

A more formal description on the conditions that B, U, X and S must satisfy is given in Fig. 12. (In addition, every template must satisfy the lexing check described in Section 5.2.3.) As the conditions are described in a syntax-directed way, it is usually fairly straightforward to find values that satisfy the requirements using a search algorithm, if they exist. Some worked examples are given in Appendix D.

It is possible that we are unable to find B, U, X and S satisfying these conditions, in which case the analysis fails. Otherwise, we transform e'_1 into a candidate e_2 by splicing in all the code templates from X and turning occurrences of **eval** into **run**.

In order to make explicit the relationship between the original and transformed subexpressions, we must maintain the labels on the program points. That is, the label on a subexpression being spliced in must match the label on the subexpression it replaces. For example, in transforming $\texttt{"f(x)"}^1$ into $(\textbf{box}(f^2(x^3))^4)^1$ we preserve the outermost label 1. Similarly, in transforming $(\texttt{"f("}^1 + y^2 + \texttt{")"}^3)^4$ into $(\textbf{box}(f^5(\textbf{unbox}\ y^2)^6)^7)^4$ we preserve the label 4 on the n-ary concatenation and the label 2 on the spliced argument.

We can now analyse our approximation e_2 using our existing analysis for staged metaprogramming. This may reveal more information about the staging behaviour of the program. For example, we may find that there are new occurrences of **eval**, there are new strings used as code values, or that certain program points take on a wider range of code values than previously assumed. We need to transfer this information back to e'_1, so that we can transform it again, generating a new, more accurate approximation e'_2.

What we would like to do is augment the constraints generated in the analysis of e'_1 with the new constraints β (some of them staged) generated by the analysis of the candidate e_2. We cannot simply union the two sets of constraints, as this would introduce staged code values to the analysis of an **eval**-based program, which would be meaningless.

However, observing that the staged constraints are order 2 constraints, after solving β in our analysis Γ' of e_2, we can filter out just the staged constraints $\beta' \subseteq \beta$, then resolve them (relative to Γ') to order 1 set-inclusion constraints γ.

As the staged constraints generated in cycle n express the interaction between the unstaged parts of $e_1 = e_{1,1}^{n+1}$ and the staged code $e_{1,2}^{n+1}, e_{1,3}^{n+1}, \ldots, e_{1,k}^{n+1} = \{e_{1,m}^{n+1} \mid m > 1\}$ introduced in its transformation

Domain anti-restriction of a relation:	$x\ (A \square R)\ y \stackrel{\text{def}}{\Leftrightarrow} x\ R\ y \wedge x \notin A$
Restricted dependencies of a program point:	$\hookleftarrow \stackrel{\text{def}}{=} (N \cup C) \square \nleftrightarrow$
Cut of a program point:	$cut(\ell) \stackrel{\text{def}}{=} \{\ell' \mid \ell \hookleftarrow^* \ell'\} \cap (N \cup C)$
Cut lifted to a set of program points:	$cut(L) \stackrel{\text{def}}{=} \bigcup \{cut(\ell) \mid \ell \in L\}$
Arguments to **eval**:	$D \stackrel{\text{def}}{=} \{\ell \mid \exists \ell' \in E, e.\textbf{eval}(e^{\ell})^{\ell'} \text{ occurs in } e'_1\}$
Require that the transformed **box/unbox** locations cover the combined cuts of all arguments to **eval**:	$cut(D) \subseteq B \cup U$
Concatenated arguments treated as holes:	$V \stackrel{\text{def}}{=} \{\ell \mid S(\ell) = h\}$
Require that transformed locations cover cuts of all arguments to concatenation treated as holes:	$cut(V) \subseteq B \cup U$
n-cat arguments treated as constants:	$W \stackrel{\text{def}}{=} \{\ell \mid S(\ell) = c\}$
String analysis (for identifying constants):	$G(\ell) \stackrel{\text{def}}{=} \begin{cases} s & \text{if, according to analysis, } \ell \\ & \text{yields the constant string } s \\ \mathit{undef} & \text{otherwise} \end{cases}$
Require that locations transformed as constant parts of templates are indeed constant:	$W \subseteq \text{dom}(G)$
Parsing function:	$parse(s) \stackrel{\text{def}}{=} \begin{cases} e & \text{if } s \text{ parses to expression } e \\ \mathit{undef} & \text{otherwise} \end{cases}$
Require that locations transformed to constant **box**es agree with string analysis:	$\forall \ell \in B.X(\ell) = parse(G(\ell))$
Template fragment for a program point:	$frag(\ell) \stackrel{\text{def}}{=} \begin{cases} \langle \bullet \rangle & \text{if } S(\ell) = h \\ G(\ell) & \text{if } S(\ell) = c \end{cases}$
Template fragment lifted to n-cat arguments:	$frag(\ell)_i \stackrel{\text{def}}{=} \begin{cases} frag(\ell') \text{ if } \ell' \text{ is } i\text{th argument to } \ell \in N \\ \mathit{undef} \quad \text{otherwise} \end{cases}$
n-ary concatenation:	$\bigoplus \overline{s_i} \stackrel{\text{def}}{=} s_1 \ldots s_n$
Template for an n-cat :	$T(\ell) \stackrel{\text{def}}{=} \bigoplus \overline{frag(\ell)_i}$
Require that templates generated at n-cat locations agree with string analysis:	$\forall \ell \in U.X(\ell) = parse(T(\ell))$

Fig. 12. Conditions on B, U, X and S in a cycle of the Boxing Algorithm.

to e_2, these order 1 constraints will refer to labelled program points in both e_1 and the staged code. Consequently, in order to combine the staged constraints meaningfully with the constraints for e_1, we must also include the constraints generated by analysis of these new expressions.

So when we repeat the transformation cycle, we consider not just the original program e_1, but also the transformed code introduced in e_2, augmented with the resolved staged constraints γ' from the analysis of e_2. Note that it is now possible that we will have to transform expressions outside our original e_1.

We repeat the process until (hopefully) we reach a fixed point; that is, until the transformed expression is identical in two consecutive cycles. The rationale is that in cycle n, the transformed expression accurately models the behaviour of e_1 up to (at least) the $n+1$th use of **eval**. If we reach a fixed point at cycle k, the result of any cycle $k' > k$ will be identical. Hence, by an inductive argument, the transformed

expression in cycle k accurately models any number of uses of **eval**. Note that, in contrast to simply executing the program up to the $n+1$th use of **eval**, cycle $n+1$ may accurately model infinitely many more uses of **eval** than cycle n. For example, if in the original program an **eval** occurs in a loop with an unbounded number of iterations (or one that does not terminate at all), the transformed program in some (finitely reachable) cycle $n+1$ may accurately model the loop up to any number of iterations.

5.3.1. Termination

It is not clear that the fixed point computation will terminate. In the case of 0CFA on a purely staged program, termination is guaranteed because we need only consider finitely many program points and finitely many abstract code values (representative of infinitely many possible constructed code values). However, both of these conditions are violated when we introduce new transformed code. That said, it seems unlikely that this would ever occur; it is more likely that the increasing density of constraints in the original program would lead to a cycle where no transformation was possible, so the algorithm would fail.

It seems unlikely that a useful, realistic program will feature a pathological sequence of extracted constant code values, although it may be possible to construct one in a similar style to a Quine (self-replicating program) [21]. For example, if a program constructs and **eval**s a constant string value equal to its own source code, and the string analysis can determine this value statically, then the next cycle will involve analysis of an entire second copy of the program. Analysis of this second copy may introduce a third copy in the following cycle, and so on.

If possible non-termination is an issue, the algorithm can simply be set to terminate after a fixed number of cycles. In fact, in the vast majority of examples, where the first cycle of analysis of **eval** does not reveal any new uses of **eval** or new code strings, two cycles are sufficient.

5.4. Soundness

We now sketch a proof of the soundness of the transformation produced by the Boxing Algorithm. It would be desirable to give a detailed proof of the correctness of the algorithm, perhaps even going as far as the mechanised proof of the information flow analysis in the previous section. However, as discussed earlier, the algorithm tackles an unusually broad range of concerns. Correspondingly, a more formal and detailed proof would need to invoke many results about the behaviours of the dataflow analysis, the string analysis and lexical analysis and parsing. Furthermore, a significant amount of technical machinery would have to be introduced to track the interaction between these concerns. In particular, it would require: a reformulation of the semantics of SLamJS in terms of a graph model (rather than the current term tree model) in order to track the correspondence between the initial expression and its staged approximations; and a modified proof of the information flow analysis to take account of the meaning of the resolved staged constraints and the staged code fragments. This would be a significant undertaking, but would be unlikely to aid significantly in the understanding of the algorithm or the clarity of its exposition, so we leave it for future work.

The transformation is developed through a sequence of approximations. So in order to argue about the correctness of the final result, we must argue about the correctness of the intermediate steps.

The input to the algorithm is an expression e_1 to be transformed. But cycle n takes as input not just $e^n_{1,1} \stackrel{\text{def}}{=} e_n$, but also (for cycle $n > 1$) fragments of staged code $e^n_{1,m}$ (where $m > 1$) and 0CFA constraints γ^n; note that the fragments of staged code do not themselves include any staging constructs. The cycle then produces, in addition to a candidate transformed program e^n_2, the fragments of staged code $e^{n+1}_{1,m}$ and constraints γ^{n+1} for the following cycle. So we must argue about the relationship of all these entities.

Lemma 3 (Boxing Cycle Soundness)**.** *In cycle n of the Boxing Algorithm:*

- *the candidate* e_2^n *simulates the execution of* e_1 *up to and including at least the nth use of* **eval***;*
- *the fragments of staged code* $e_{1,m}^{n+1}$ *are those that, when spliced into* e_1*, yield* e_2^n*;*
- *the constraints* γ^{n+1}*, combined with the constraints generated by analysing* $e_{1,m}^{n+1}$*, yield all staged flows within* e_2^n*, and hence all flows in* e_1 *resulting from uses of* **eval** *up to and including its nth use.*

Proof. See Appendix C. □

Theorem 6 (Boxing Algorithm Soundness)**.** *Consider a program* e_1 *written in SLamJS extended with* **eval** *but without staging constructs. If the Boxing Algorithm transforms* e_1 *to* e_2 *and* $e_1 \xrightarrow{\square}{}^* v$*, then* $e_2 \xrightarrow{\square}{}^* v$*. Furthermore,* e_2 *is a SLamJS program without* **eval**.

Proof. If the algorithm terminates, then $e_2 = e_2^k$ for some k, with $e_2^k = e_2^{k-1}$. Hence $\forall n \geqslant k.e_2^k = e^n$. Thus by Lemma 3, e_2 simulates the execution of e_1 up to any number of uses of **eval**. There are only finitely many uses of **eval** in the execution of $e_1 \xrightarrow{\square}{}^* v$, so e_2 simulates the entirety of the execution of e_1 and $e_2 \xrightarrow{\square}{}^* v$. The fact that e_2 does not use **eval** follows from the construction of the transformation. □

5.5. Implementation and examples

We now consider some example programs on which our implementation of the transformation works and some on which it does not. As we are not immediately concerned with information flow analysis, but rather on turning **eval** into staged metaprogramming, our examples do not feature any dependency markers.

Example 21.

Original program:	Transformed program:
let $x =$ **if**(**true**){`"0"`} **else** {`"1"`} **in**	**let** $x =$ **if**(**true**){**box** 0} **else**{**box** 1} **in**
let $y =$ `"add("` $+ x +$ `", 2)"` **in**	**let** $y =$ **box**$\big(add\big((\textbf{unbox}\, x), 2\big)\big)$ **in**
eval y	**run** y

This example illustrates the basic concept that we turn constant strings (in this case, representing the numbers 0 and 1) into **box** expressions and concatenation (in this case, inside a template that adds 2) into **box** and **unbox** expressions. Constant strings and concatenation into argument position covers a large proportion of **eval** use cases [41], so in the rest of the examples we look at some more esoteric examples and situations in which the algorithm might fail to produce a transformation.

Example 22.

Original program:

let $id =$ `"fun (x) { x } "` **in**
let $arg =$ **if**(**true**)$\big\{$`"(1)"`$\big\}$ **else**$\big\{$`"(2)"`$\big\}$ **in**
eval$(id + arg)$

Transformation fails.

Although this is a seemingly reasonably program, in this case, the transformation fails because the concatenation does not match the syntactic structure of the language. As *arg* is not a constant, it must be treated as a hole for a code expression. According to the grammar of SLamJS expressions, function application has the form $e_1(e_2)$. So while $\textbf{fun}(x)\{x\}$, (1) and (2) are all valid expressions, the concatenation of two expressions in the final line does not constitute an expression; there is no grammatical form $e_1 e_2$ in the language. If the final concatenation had been $id + \texttt{"("} + arg + \texttt{")"}$, it would have worked. The source of this problem in general is that parsing is not compositional, in that it is not true (for some suitably-defined operation $\cdot$) that $parse(s_1 \cdot s_2) = parse(s_1) \cdot parse(s_2)$.

Example 23.

Original program:	Transformed program:
let $id =$ `"fun (x) { x } "` **in**	**let** $id =$ `"fun (x) { x } "` **in**
let $arg =$ `"(1)"` **in**	**let** $arg =$ `"(1)"` **in**
eval$(id + arg)$	**run**$\big($**box**$\big($**fun**$(x)\{x\}(1)\big)\big)$

Where possible, the transformation prefers to use **box** and **unbox** to preserve the structure of the original program. However, where it is not possible, it will try to use constant **box** expressions instead. This example is very similar to the previous one, but because the **eval**ed string is constant, the transformation can handle it, even though the concatenation does not follow the syntactic structure of the language. Here, the transformation determines exactly what the constant string passed to **eval** is and parses it directly, rather than trying to build it from its component subexpressions.

Note how, as *id* and *arg* are not transformed, they remain in the transformed program, but are unused. If we wish to preserve information flow in the transformed program, rather than just the final result, then the **box** expression must adopt any dependency markers that were on *id* and *arg*.

A similar concern applies if we wish to extend the transformation to handle a language with side effects, especially as they may affect which code strings are constructed. Suppose our original program contains an expression e that yields a code string s, but also has some side effects, such as incrementing a mutable variable used as a loop counter elsewhere in the source program. If we wish to transform e to a code value **box** e', we must be careful to preserve these side effects. We can do this by executing e and discarding the result; that is, transforming e not to **box** e' but to **let** $x = e$ **in box** e', where x is fresh.

Example 24.

Original program:	Transformed program:
let $x =$ `"1"` **in**	**let** $x =$ **box** 1 **in**
let $y =$ `"eval x"` **in**	**let** $y =$ **box** (**run** x) **in**
eval y	**run** y

All of the examples so far reach a fixed point in a single cycle of the algorithm, although a second cycle is required in order to check that a fixed point has been reached. This program requires two cycles to reach a fixed point: the first finds that the string in y is a code string for **eval** x; the second finds that the string in x is also a code string.

Example 25.

Original program:

```
let init = "0" in
let double = 5 in
let build = fun(loop){fun(n){fun(c){
 if(n = 0){c} else{((loop(loop))
 (sub(n, 1))("add(" + c + ", 2)")}}}} in
let code = (build(build))(double)(init) in
eval code
```

Transformed program:

```
let init = box 0 in
let double = 5 in
let build = fun(loop){fun(n){fun(c){
 if(n = 0){c} else{((loop(loop))
 (sub(n, 1))(box(add((unbox c), 2)))}}}} in
let code = (build(build))(double)(init) in
run code
```

This is a functional implementation of a motivating example of Choi et al. [7]. It builds the arithmetic expression `2 + ... + 2` of unbounded size (determined here by the value of *double*) and executes it. The purpose of that example was to demonstrate the difficulty of handling string-based metaprogramming, in contrast with template-based staged metaprogramming, as methods for analysing the former often introduce imprecision (for example, in the form of an infinite expression). As we claim our analysis is applicable to JavaScript's **eval**, it is important that we can transform it exactly. As this example shows, we can.

Example 26.

Original program:

```
let gen_power =
  let f = fun(p){fun(n){
    if(n <= 0){"1"} else
    {let q = (p(p))(sub(n, 1)) in
      "mul(x, " + q + ")"}
  }} in
  f(f) in
let power = fun(y){eval(
 "fun(x) { " + (gen_power(y)) + " } ")} in
let raise5 = power(5) in
raise5(2)
```

Transformed program:

```
let gen_power =
  let f = fun(p){fun(n){
    if(n <= 0){box 1} else
    {let q = (p(p))(sub(n, 1)) in
      (box(mul(x, unbox q)))}
  }} in
  f(f) in
let power = fun(y){run(
 box(fun(x){unbox(gen_power(y))}))} in
let raise5 = power(5) in
raise5(2)
```

The staged power function is a staple of literature on metaprogramming [3]. Given a number (in this case 5), it generates code for a function that takes an argument and raises it to that power. This can be useful in performance-critical situations, as it avoids the overhead of using a loop. While it is unlikely that anyone would use JavaScript in such a situation, the ubiquity of the example demands that we should be able to handle it.

Note that, in general, any program that only uses constant code strings (with no concatenation or any attempts to perform intensional operations) and does not feature nested uses of **eval** can be transformed successfully by the algorithm. There are many more interesting programs that can also be successfully transformed, but we do not have a succinct characterisation of them.

6. Related work

6.1. 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 [6].

Precision of 0CFA 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 that handle these. But one might ask whether 0CFA is a good fit for JavaScript. In particular, its lack of context sensitivity may make it too imprecise for some programs. However, there is good evidence to suggest that CFA-style analyses are a good fit for JavaScript, and hence our work could be adapted to these. The obvious way to add context sensitivity gives k-CFA. For example, Guarnieri et al. [17] use 1-CFA (k-CFA with $k = 1$) augmented with a variety of JavaScript-focused techniques to build Actarus, a static taint analysis tool for JavaScript. They show their tool working on a variety of challenging examples. Alternatively, Might et al. [35] suggest a subtler variant called m-CFA that adds context sensitivity as good as k-CFA for most object-oriented programs, but with much better performance. Perhaps most notable is the recent CFA2 analysis [50], which was developed for JavaScript and features significantly better analysis of higher order flow control.

Associative Arrays as Objects One of the most challenging features of JavaScript from a static analysis perspective is its objects, which are really associative arrays. (In other scripting languages these are called hashes or dicts.) In particular, as any string can be used as a field name, it is difficult to determine whether two distinct reads or writes might refer to the same field. Our analysis is deliberately coarse in its handling of objects, so that we can focus on **eval**. These challenges have been considered in detail for k-CFA by Liang and Might [32]. However, as their work targets Python, they do not discuss JavaScript's prototype chains or **with** construct. Both of these features are treated by λ_{JS} as syntactic sugar (although the translation of **with** is non-compositional), so if our underlying analysis is sufficiently precise, we might be able to do the same without loss of precision. However, as **with** interacts with variable scoping, this might not be so straightforward.

JavaScript Semantics A bigger problem in producing a sound analysis of JavaScript is the complexity and quaintness of its semantics [33]. 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} [18]. 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. As we have shown, in SLamJS, an automated and sound transformation from **eval** into staged metaprogramming is often possible. However, this relies on certain assumptions that may not always hold in full JavaScript. For example, the effect of implicit type conversion on string concatenation would need to

be considered, as would the preservation of side effects [29]. The purely extensional view of metaprogramming does not allow us to manipulate variable names within code, as in the JavaScript example `eval ("f_" + n);`. This feature could be added, but to do so in a general way would complicate the analysis somewhat, as the number of variable names used in a program might no longer be finite, so an extra layer of abstraction would be needed in order to retain precision.

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 [4].

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 [9]? 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 [31].

6.2. *Information flow analysis*

Early work on information flow security focused on monitoring program execution, dynamically marking variables to indicate their level of confidentiality [13]. 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 [10]. Denning and Denning developed a simple static information flow analysis that rejected programs with flows violating a security policy [11].

Noninterference Goguen and Meseguer introduced the idea of noninterference [15] (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 [39].

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 [51] to Simonet and Pottier's type system for ML [40]. 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 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. [25] and Hedin and Sabelfeld [19]. 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 [52], 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 [44]. Note that this is in contrast to analysis of safety properties, where static analyses may generate "false positives" because they need to approximate flow control and other complex behaviours within a program.

Nonetheless, there has been a resurgence of interest in dynamic analyses [45]. From a practical perspective, dynamic analyses are usually significantly simpler than static analyses, which means that they can be developed more easily for complex languages like JavaScript. They can also be deployed in situations where there is no opportunity to analyse the code before running it, for example because it is being supplied by a third-party advertiser on a website. In terms of speed, dynamic analyses incur no computational cost during development, but typically slow down execution of a program by a roughly constant factor, so there is often a focus on making them more efficient [38]. Static analyses can be slow to run during development, but cost nothing at run-time. Thus, provided the analysis scales reasonably with the size of the program, optimisation is less important, as it has no impact on the user.

But ultimately, dynamic and static approaches are fundamentally different in that static analyses enforce information flow policies by alerting the developer before a program is deployed, allowing the program to be fixed before it causes problems for a user. Although dynamic approaches are also able to enforce information flow policies, they do so by terminating the offending program, inconveniencing its user.

Hybrid Approaches As a compromise, Chugh et al. [9] 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 [2] 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.

6.3. 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. [7]. 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. [27]. 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. [7] 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. This method allows existing static analyses for the unstaged language to be used

on the staged language, requiring only the specification of a "projection function" that describes how analysis results for the translated program relate to the original program.

There are some limitations to their work. Most significantly, many interesting programs, such as the one mentioned earlier, are not valid in $\lambda_{\mathcal{S}}$ and hence cannot be unstaged using their translation; this limits its applicability to JavaScript. While it may be possible to adapt their approach to our semantics of variable capture, we believe that this may be indicative of a more fundamental problem with their approach, namely that it relies on variable binding when splicing code being similar to variable binding of functions. Although this is the case in $\lambda_{\mathcal{S}}$, there should be no need for it to be so in other languages.

Furthermore, as shown in Examples 18–20, 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 [23] 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. [48] consider using static analysis to optimise compilation in a cut-down version of Cyclone, a type-safe, C-style language with run-time 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.

We have mainly considered homogeneous metaprogramming, in which the code manipulated and executed is written in the same language as the code that manipulates it. In heterogeneous metaprogramming, the two languages are different. This is particularly relevant for web applications that construct database queries, which are often written in SQL. Schoepe et al. [46] extend an idea from Cheney et al. [5] to produce an ML-like language in which database queries can be built using staged metaprogramming. They develop a type system for analysing information flows within the language and the database, including flows that result from the program reading information through a database query and later writing it with a different query.

6.4. Analysing eval

There has been relatively little work on analysing **eval**. Probably the most advanced is Jensen et al.'s tool Unevalizer [24], which is based around the JavaScript analysis tool TAJS. In contrast to our approach of transforming **eval** into a better-behaved form and then analysing that, they aim to analyse and remove it in a single step, replacing it with code that does not use metaprogramming. In addition to being able to transform constant strings, their tool can recognise certain fixed patterns of variable string usage, such as concatenating an argument into a function or an object access. However, unlike our approach, theirs lacks generality: new usage patterns must be manually added and justified. Their tool is more practically motivated, meaning that they can handle full JavaScript taken from popular websites. Unfortunately, the combination of JavaScript's semantic peculiarities and their ad-hoc approach seems to lead to them to expend considerable effort reasoning about the correctness of each new transformation pattern.

A different approach is taken by Meawad et al. [34]. Their tool Evalorizer uses a proxy to intercept and log uses of JavaScript **eval** that occur during Web browsing. It then advises a developer on how best

to replace them with code that does not use **eval**. In order to do this safely and sensibly, it must first categorise the dynamically gathered code strings according to their structure and content. The aim of the tool is to aid migration of a website away from **eval**, with interaction from the developer, rather than to analyse its behaviour fully and automatically.

7. 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 have shown how to apply our analysis to a language with string-based **eval** via a transformation to staged metaprogramming. We believe our approach is transferable to other CFA-style analyses and applicable to JavaScript.

Progressing from here, there are two 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.

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.

Supplementary data

Online supplement consisting of Appendices A–D is available at: http://dx.doi.org/10.3233/JCS-160557.

Acknowledgments

We thank our anonymous reviewers for their comments and suggestions.

References

[1] M. Abadi, A. Banerjee, N. Heintze and J.G. Riecke, A core calculus of dependency, in: *POPL*, 1999, pp. 147–160.
[2] T.H. Austin and C. Flanagan, Multiple facets for dynamic information flow, in: *POPL*, 2012, pp. 165–178.
[3] M. Berger and L. Tratt, Program logics for homogeneous meta-programming, in: *LPAR (Dakar)*, E.M. Clarke and A. Voronkov, eds, Lecture Notes in Computer Science, Vol. 6355, Springer, 2010, pp. 64–81.
[4] A. Bohannon, B.C. Pierce, V. Sjöberg, S. Weirich and S. Zdancewic, Reactive noninterference, in: *Computer and Communications Security*, 2009, pp. 79–90.
[5] J. Cheney, S. Lindley and P. Wadler, A practical theory of language-integrated query, in: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13*, Boston, MA, USA, September 25–27, 2013, G. Morrisett and T. Uustalu, eds, ACM, 2013, pp. 403–416. doi:10.1145/2500365.2500586.
[6] 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.
[7] W. Choi, B. Aktemur, K. Yi and M. Tatsuta, Static analysis of multi-staged programs via unstaging translation, in: *POPL*, 2011, pp. 81–92.
[8] A.S. Christensen, A. Møller and M.I. Schwartzbach, Precise analysis of string expressions, in: *Proceedings of the Static Analysis, 10th International Symposium, SAS 2003*, San Diego, CA, USA, June 11–13, 2003, R. Cousot, ed., Lecture Notes in Computer Science, Vol. 2694, Springer, 2003, pp. 1–18.
[9] R. Chugh, J.A. Meister, R. Jhala and S. Lerner, Staged information flow for JavaScript, in: *PLDI*, 2009, pp. 50–62. doi:10.1145/1542476.1542483.

[10] D.E. Denning, A lattice model of secure information flow, *CACM* **19**(5) (1976), 236–243. doi:10.1145/360051.360056.
[11] D.E. Denning and P.J. Denning, Certification of programs for secure information flow, *CACM* **20**(7) (1977), 504–513. doi:10.1145/359636.359712.
[12] K. Doh, H. Kim and D.A. Schmidt, Abstract LR-parsing, in: *Formal Modeling: Actors, Open Systems, Biological Systems – Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, G. Agha, O. Danvy and J. Meseguer, eds, Lecture Notes in Computer Science, Vol. 7000, Springer, 2011, pp. 90–109. doi:10.1007/978-3-642-24933-4_6.
[13] J.S. Fenton, Memoryless subsystems, *Comput. J.* **17**(2) (1974), 143–147. doi:10.1093/comjnl/17.2.143.
<uncited>[14] J. Field and T. Teitelbaum, Incremental reduction in the lambda calculus, in: *LISP and Functional Programming*, 1990, pp. 307–322.
[15] J.A. Goguen and J. Meseguer, Security policies and security models, in: *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.
[16] S. Guarnieri and V.B. Livshits, GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code, in: *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 10–14, 2009, F. Monrose, ed., USENIX Association, 2009, pp. 151–168.
[17] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet and R. Berg, Saving the world wide web from vulnerable JavaScript, in: *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011*, Toronto, ON, Canada, July 17–21, 2011, M.B. Dwyer and F. Tip, eds, ACM, 2011, pp. 177–187. doi:10.1145/2001420.2001442.
[18] A. Guha, C. Saftoiu and S. Krishnamurthi, The essence of JavaScript, in: *ECOOP*, 2010, pp. 126–150.
[19] D. Hedin and A. Sabelfeld, Information-flow security for a core of JavaScript, in: *CSF*, S. Chong, ed., IEEE, 2012, pp. 3–18.
[20] N. Heintze and D.A. McAllester, On the cubic bottleneck in subtyping and flow analysis, in: *LICS*, IEEE Computer Society, 1997, pp. 342–351.
[21] D.R. Hofstadter, *Godel, Escher, Bach: An Eternal Golden Braid*, Basic Books, Inc., New York, NY, USA, 1979.
[22] D.V. Horn and M. Might, An analytic framework for JavaScript, *CoRR*, abs/1109.4467 (2011).
[23] J. Inoue and W. Taha, Reasoning about multi-stage programs, in: *ESOP*, 2012.
[24] S.H. Jensen, P.A. Jonsson and A. Møller, Remedying the eval that men do, in: *ISSTA*, 2012, pp. 34–44.
[25] S. Just, A. Cleary, B. Shirley and C. Hammer, Information flow analysis for JavaScript, in: *PLASTIC*, 2011.
[26] 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.
[27] T. Kim, C. Lee, K. Lee, S. Baik and K. Yi, A control flow analysis for 2-staged programming languages, Techreport ROSAEC-2009-005, ROSAEC, 2009.
[28] M. Lester, L. Ong and M. Schäfer, Information flow analysis for a dynamically typed language with staged metaprogramming, in: *CSF*, IEEE, 2013, pp. 209–223.
[29] M.M. Lester, Position paper: The science of boxing, in: *PLAS*, P. Naldurg and N. Swamy, eds, ACM, 2013, pp. 83–88.
[30] M.M. Lester, Verifying information flow and metaprogramming in dynamically typed languages: Mechanised Coq proofs and analysis source code supporting thesis, Oxford University Research Archive, 2015. doi:10.5287/bodleian:wxdB0NV6k.
[31] P. Li and S. Zdancewic, Downgrading policies and relaxed noninterference, in: *POPL*, 2005, pp. 158–170.
[32] S. Liang and M. Might, Hash-flow taint analysis of higher-order programs, in: *Proceedings of the 2012 Workshop on Programming Languages and Analysis for Security, PLAS 2012*, Beijing, China, June 15, 2012, S. Maffeis and T. Rezk, eds, ACM, 2012, Article No. 8.
[33] S. Maffeis, J.C. Mitchell and A. Taly, An operational semantics for JavaScript, in: *APLAS*, 2008, pp. 307–325.
[34] F. Meawad, G. Richards, F. Morandat and J. Vitek, Eval begone!: Semi-automated removal of eval from javascript programs, in: *OOPSLA*, G.T. Leavens and M.B. Dwyer, eds, ACM, 2012, pp. 607–620.
[35] M. Might, Y. Smaragdakis and D.V. Horn, Resolving and exploiting the k-CFA paradox, *CoRR*, abs/1311.4231 (2013).
[36] F. Nielson, H.R. Nielson and C. Hankin, *Principles of Program Analysis*, Springer, 1999.
[37] J. Palsberg and M.I. Schwartzbach, Safety analysis versus type inference, *Inf. Comput.* **118**(1) (1995), 128–141. doi:10.1006/inco.1995.1058.
[38] P.H. Phung, D. Sands and A. Chudnov, Lightweight self-protecting JavaScript, in: *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2009*, Sydney, Australia, March 10–12, 2009, W. Li, W. Susilo, U.K. Tupakula, R. Safavi-Naini and V. Varadharajan, eds, ACM, 2009, pp. 47–60.
[39] F. Pottier and S. Conchon, Information flow inference for free, in: *ICFP*, 2000.
[40] F. Pottier and V. Simonet, Information flow inference for ML, *TOPLAS* **25**(1) (2003), 117–158. doi:10.1145/596980.596983.
[41] 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.
[42] J. Rushby, Noninterference, transitivity, and channel-control security policies, Technical report, December 1992.
[43] A. Russo and A. Sabelfeld, Dynamic vs. static flow-sensitive security analysis, in: *CSF*, 2010, pp. 186–199.

[44] A. Sabelfeld and A.C. Myers, Language-based information-flow security, *IEEE Journal on Selected Areas in Communications* **21**(1) (2003), 5–19. doi:10.1109/JSAC.2002.806121.
[45] 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.
[46] D. Schoepe, D. Hedin and A. Sabelfeld, SeLINQ: Tracking information across application-database boundaries, in: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, Gothenburg, Sweden, September 1–3, 2014, J. Jeuring and M.M.T. Chakravarty, eds, ACM, 2014, pp. 25–38.
[47] O. Shivers, Control-flow analysis in scheme, in: *PLDI*, 1988, pp. 164–174.
[48] F. Smith, D. Grossman, J.G. Morrisett, L. Hornof and T. Jim, Compiling for template-based run-time code generation, *J. Funct. Program.* **13**(3) (2003), 677–708. doi:10.1017/S095679680200463X.
[49] R. van der Meyden, What, indeed, is intransitive noninterference?, *Journal of Computer Security* **23**(2) (2015), 197–228. doi:10.3233/JCS-140516.
[50] D. Vardoulakis and O. Shivers, CFA2: A context-free approach to control-flow analysis, *Logical Methods in Computer Science* **7**(2) (2011). doi:10.2168/LMCS-7(2:3)2011.
[51] D.M. Volpano, C.E. Irvine and G. Smith, A sound type system for secure flow analysis, *Journal of Computer Security* **4**(2/3) (1996), 167–188. doi:10.3233/JCS-1996-42-304.
[52] S. Zdancewic, Programming languages for information security, PhD thesis, Cornell University, 2002.