Pushdown Abstractions of JavaScript

David Van Horn[1] and Matthew Might[2]

[1] Northeastern University, Boston, Massachusetts, USA
[2] University of Utah, Salt Lake City, Utah, USA

Abstract. We design a family of program analyses for JavaScript that make *no approximation* in matching calls with returns, exceptions with handlers, and breaks with labels. We do so by starting from an established reduction semantics for JavaScript and systematically deriving its intensional abstract interpretation. Our first step is to transform the semantics into an equivalent low-level abstract machine: the JavaScript Abstract Machine (JAM). We then give an infinite-state yet decidable pushdown machine whose stack precisely models the structure of the concrete program stack. The precise model of stack structure in turn confers *precise* control-flow analysis even in the presence of control effects, such as exceptions and finally blocks. We give pushdown generalizations of traditional forms of analysis such as k-CFA, and prove the pushdown framework for abstract interpretation is sound and computable.

1 Introduction

JavaScript is the dominant language of the web, making it the most ubiquitous programming language in use today. Beyond the browser, it is increasingly important as a general-purpose language, as a server-side scripting language, and as an embedded scripting language—notably, Java 6 includes support for scripting applications via the `javax.script` package, and the JDK ships with the Mozilla Rhino JavaScript engine. Due to its ubiquity, JavaScript has become the target language for an array of compilers for languages such as C#, Java, Ruby, and others, making JavaScript a widely used "assembly language." As JavaScript cements its foundational role, the importance of robust static reasoning tools for that foundation grows.

Motivated by the desire to handle non-local control effects such as exceptions and `finally` precisely, we will depart from standard practice in higher-order program analysis to derive an *infinite*-state yet decidable pushdown abstraction from our original abstract machine. The stack of the pushdown abstract interpreter *exactly* models the stack of the original abstract machine with no loss of structure—approximation is inflicted on only the control states. This pushdown framework offers a degree of precision in reasoning about control inaccessible to previous analyzers.

Pushdown analysis is an alternative paradigm for the analysis of higher-order programs in which the run-time program stack is precisely modeled with the stack of a pushdown system [40, 14]. Consequently, a pushdown analysis can

exactly match control flow transfers from calls to returns, from throws to handlers, and from breaks to labels. This in contrast with the traditional approaches of finite-state abstractions which necessarily model the control stack with finite bounds.

As an example demonstrating the basic difference between traditional approaches, such as 0CFA, and our pushdown approach, consider the following JavaScript program:

```
// (ℝ → ℝ) → (ℝ → ℝ)
// Compute an approximate derivative of f.
function deriv(f) {
  var ϵ = 0.0001;
  return function (x) {
    return (f(x+ϵ) - f(x-ϵ)) / (2*ϵ);
  };
};
deriv(function (y) { return y*y; });
```

The `deriv` program computes an approximation to the derivative of its argument. In this example, it is being applied the square function, so it returns an approximation to the double function.

It is important to take note of the two distinct calls to `f`. Basic program analyses, such as 0CFA, will determine that the square function is the target of the call at `f(x+ϵ)`. *However, they cannot determine whether the call to* `f(x+ϵ)` *should return to* `f(x+ϵ)` *or to* `f(x-ϵ)`. Context-sensitive analysis, such as 1CFA, can reason more precisely by distinguishing the analysis of each call to `f`, however such methods come with a prohibitive computational cost [38] and, more fundamentally, k-CFA will only suffice for precisely reasoning about the control stack up to a fixed calling context depth.

This is the fundamental shortcoming of traditional approaches to higher-order program analysis, both in functional and object-oriented languages. This is an unfortunate situation, since the dominant control mechanism is calls and returns. To make matters worse, in addition to higher-order functions, JavaScript includes sophisticated control mechanisms further complicating and confounding analytic approaches.

To overcome this shortcoming we use a *pushdown* approach to abstraction that exactly captures the behavior of the control stack. We derive the pushdown analysis as an abstract interpretation of an abstract machine for JavaScript. The crucial difference between our approach and previous approaches is that we will leave the stack unabstracted. As this abstract interpretation ranges over an infinite state-space, the main technical difficulty will be recovering decidability of reachable states.

Challenges from JavaScript

JavaScript is an expressive, aggressively dynamic, high-level programming language. It is a higher-order, imperative, untyped language that is both functional

and object-oriented, with prototype-based inheritance, constructors, non-local control, and a number of semantic quirks. Most quirks simply demand attention to detail, e.g.:

```
if (false) { var x ; }
... x ... // x is defined
```

Other quirks, such as the much-maligned `with` construct end up succumbing to an unremarkable desugaring. Yet other features, like non-local control effects and prototypical inheritance, require attention in the mechanics of the analysis itself; for a hint of what is possible, consider:

```
out: while (true)
 try {
   break out ;
 } finally {
   try {
     return 10 ;
   } finally {
     console.log("this runs; 10 returns") ;
   }
 }
```

It has become customary when reasoning about JavaScript to assume well-behavedness—that some subset of its features are never (or rarely) used for many programs. Richards, Lebresne, Burg and Vitek's thorough study [34] has cast empirical doubt on these well-behavedness assumptions, finding almost every language feature used in almost every program in a large corpus of widely deployed JavaScript code.

Our goal is a principled approach for reasoning about *all* of JavaScript, including its unusual semantic peculiarities and its complex control mechanisms. To make this possible, the first step is the calculation of an *abstractable* abstract machine from an established semantics for JavaScript. From there, a pushdown abstract interpretation of that machine yields a sound, robust framework for static analysis of JavaScript with precise reasoning about the control stack.

Contributions

The primary contribution of this work is a provably sound and computable framework for infinite-state pushdown abstract interpretations of all of JavaScript, sans `eval`, that *exactly* models the program stack including complex local and non-local control-flow relationships, such as proper matching between calls and returns, throws and handlers, and breaks and labels.[3]

[3] One might wonder why `break` to a label requires non-local reasoning. In fact, it should not require it, but Guha *et al.*'s desugaring into λ_{JS}, handles `break` using powerful escape continuations. Constraints in the desugaring process prevent these

In support of our primary contribution, our secondary contributions include the development of a variant of a known formal model for JavaScript as a calculus of explicit substitutions; a correct abstract machine for this model obtained via a detailed derivation, carried out in SML, going from the calculus to the machine via small, meaning-preserving program transformations; and executable semantic models for the reduction semantics, the abstract machine and its pushdown-abstractions, written in PLT Redex [15].

Outline

Section 2 gives the syntax and semantics of a core calculus of explicit substitutions based on the λ_{JS}-calculus. This new calculus, $\lambda\rho_{JS}$, is shown to correspond with λ_{JS}. Section 3 derives an abstract machine, the JavaScript Abstract Machine (JAM), from the calculus of explicit substitutions, which is a correct machine for evaluating λ_{JS} programs. The machine has been crafted in such a way that it is suitable for approximation by a pushdown automaton. Section 4 yields a family of pushdown abstract interpreters by a simple store-abstraction of the JAM. The framework is proved to be sound and computable. Specific program analyses are obtained by instantiating the allocation strategy for the machine and examples of strategies corresponding to pushdown generalization of known analyses are given. Section 5 relates this work to the research literature and section 6 concludes.

Background and notation: We assume a basic familiarity with reduction semantics and abstract machines. For background on concepts, terminology, and notation employed in this paper, we refer the reader to *Semantics Engineering with PLT Redex* [15]. Our construction of machines from reduction semantics follows Danvy, *et al.*'s refocusing-based approach [13, 4, 12]. Finally, for background on systematic abstract interpretation of abstract machines, see our recent work on the approach [39].

2 A calculus of explicit substitutions: $\lambda\rho_{JS}$

Our semantics-based approach to analysis is founded on abstract machines, which give an idealized characterization of a low-level language implementation. As such, we need a correct abstract machine for JavaScript. Rather than design one from scratch and manually verify its correctness after the fact, we rely on the syntactic correspondence between calculi and machines and adopt the refocusing-based approach of Danvy, *et al.*, to construct an abstract machine systematically from an established semantics for JavaScript.

Guha, Saftoiu, and Krishnamurthi [17] give a small core calculus, λ_{JS}, with a small-step reduction semantics using evaluation contexts and demonstrate that

escape continuations from crossing interprocedural boundaries, but unrestricted—or optimized λ_{JS}—may violate these constraints. For completeness, we handle full, unrestricted λ_{JS}, which means we must model these general escape continuations.

full JavaScript can be desugared into λ_{JS}. The semantics accounts for all of JavaScript's features with the exception of `eval`. Only some of JavaScript quirks are modeled directly, while other aspects are treated traditionally. For example, lexical scope is modeled with substitution. The desugarer is modeled formally and also available as a standalone Haskell program.

We choose to adopt the λ_{JS} model since its small size results in a tractably sized abstract machine.

The remainder of this paper focuses on machines and abstract interpretation for λ_{JS}. We refer the reader to Guha, *et al.*, for details on desugaring JavaScript to λ_{JS} and rational for the design decisions made.

2.1 Syntax

The syntax of $\lambda\rho_{JS}$ is given in figure 1. Syntactic constants include strings, numbers, addresses, booleans, the undefined value, and the null value. Addresses are first-class values used to model mutable references. Heap allocation and dereference is made explicit through desugaring to λ_{JS}. Syntactic values include constants, function terms, and records. Records are keyed by strings and operations on records are modeled by functional update, extension, and deletion. Expressions include variables, syntactic values, and syntax for let binding, function application, record dereference, record update, record deletion, assignment, allocation, dereference, conditionals, sequencing, while loops, labels, breaks, exception handlers, finalizers, exception raising, and application of primitive operations. A program is a closed expression.

$$
\begin{array}{rcl}
s & \in & \mathit{String} \\
n & \in & \mathit{Number} \\
a & \in & \mathit{Address} \\
x & \in & \mathit{Variable} \\
e, f, g & ::= & x \mid s \mid n \mid a \mid \textsf{true} \mid \textsf{false} \mid \textsf{undef} \mid \textsf{null} \\
& \mid & \textsf{fun}(\overline{x})\ \{\ e\ \} \mid \{\overline{s:e}\} \mid \textsf{let}\ (x = e)\ e \mid e(\overline{e}) \mid e[e] \\
& \mid & e[e] = e \mid \textsf{del}\ e[e] \mid e = e \mid \textsf{ref}\ e \mid \textsf{deref}\ e \\
& \mid & \textsf{if}(e)\{e\}\{e\} \mid e;e \mid \textsf{while}(e)\{e\} \\
& \mid & \ell:\{\ e\ \} \mid \textsf{break}\ \ell\ e \mid \textsf{try}\ \{e\}\ \textsf{catch}\ (x)\{e\} \\
& \mid & \textsf{try}\ \{e\}\ \textsf{finally}\ \{e\} \mid \textsf{throw}\ e \mid \mathit{op}(\overline{e}) \\
t, u, v & ::= & s \mid n \mid a \mid \textsf{true} \mid \textsf{false} \mid \textsf{undef} \mid \textsf{null} \mid (\textsf{fun}(\overline{x})\ \{\ e\ \}, \rho) \mid \{\overline{s:v}\} \\
c, d & ::= & (e, \rho) \mid \{\overline{s:c}\} \mid \textsf{let}\ (x = c)\ c \mid c(\overline{c}) \mid c[c] \\
& \mid & c[c] = c \mid \textsf{del}\ c[c] \mid c = c \mid \textsf{ref}\ c \mid \textsf{deref}\ c \\
& \mid & \textsf{if}(c)\{c\}\{c\} \mid c;c \mid \textsf{while}(c)\{c\} \\
& \mid & \ell:\{\ c\ \} \mid \textsf{break}\ \ell\ c \mid \textsf{try}\ \{c\}\ \textsf{catch}\ (x)\{c\} \\
& \mid & \textsf{try}\ \{c\}\ \textsf{finally}\ \{c\} \mid \textsf{throw}\ c \mid \mathit{op}(\overline{c})
\end{array}
$$

Fig. 1: Syntax of λ_{JS}

2.2 Semantics

Guha, *et al.*, give a substitution-based reduction semantics formulated in terms of Felleisen-Hieb-style evaluation contexts [15]. The use of substitution in λ_{JS} is traditional from a theoretical point of view, and is motivated in part by want of conventional reasoning techniques such as subject reduction. On the other hand, environments are traditional from an implementation point of view. To mediate the gap, we first develop a variant of λ_{JS} that models the meta-theoretic notion of substitution with explicit substitutions.

Substitutions are represented explicitly with *environments*, which are finite maps from variables to values.

Substitution $[v/x]e$, which is a meta-theoretic notation denoting e with all free-occurrences of x replaced by v, is represented at the syntactic level in $\lambda\rho_{JS}$, a calculus of explicit substitutions, as a pair consisting of e and an environment representing the substitution: $(e, \{(x, v)\})$. Such a pair is known as a *closure*.

The heap is modeled as a top-level *store*, a finite map from addresses to values.

The complete syntax of values and closures in $\lambda\rho_{JS}$ is given in figure 1. The semantics of $\lambda\rho_{JS}$ is given in terms of a small-step reduction relation defined in figure 2. There are four classes of reductions:

1. context-insensitive, store-insensitive reductions operating over closures to implement computations that have no effect on the context or store,
2. context-sensitive or store-sensitive reductions operating over pairs of stores and programs to implement memory- and control-effects,
3. (omitted) reductions propagating environments from closures to inner expressions, and
4. (omitted) reductions raising exceptions that represent run-time errors such as applying a non-function, branching on a non-boolean[4], indexing into a non-record or with a non-string key, etc. As a result, $\lambda\rho_{JS}$ programs do not get stuck: either they diverge or result in a value or an uncaught exception.

Reduction proceeds by a program being decomposed into a redex and evaluation context, which represents a portion of program text with a single hole, written "$\bullet$". The grammar of evaluation contexts, defined in figure 3, specifies *where* in a program reduction may occur. The notation "$\mathcal{E}[c]$" denotes both the decomposition of a program into the evaluation context $\mathcal{E}$ with c in the hole and the plugging of c into $\mathcal{E}$, which replaces the single hole in $\mathcal{E}$ by c. In addition to closures, holes may also be replaced by contexts, which yields another context. This is indicated with the notation "$\mathcal{E}[\mathcal{E}']$".

There are three classes of evaluation contexts in figure 3: *local contexts* $\mathcal{C}$ range over all contexts that do not include exception handlers, finalizers, or labels; *control contexts* $\mathcal{D}$ range over contexts that are either empty or have a

[4] One of JavaScript's quirks are its broad definitions of which values act as true and false, a quirk which doesn't appear to be modeled here at first glance. The desugaring transformation eliminates this quirk by coercing the condition in an **if** expression.

Context-insensitive, store-insensitive rules:

$$
\begin{aligned}
(x, \rho) &\to \rho(x) \\
\textsf{let (}x = v\textsf{)}\ (e, \rho) &\to (e, \rho[x \mapsto v]) \\
(\textsf{fun(}\overline{x}\textsf{) \{ }e\textsf{ \}}, \rho)(\overline{v}) &\to (e, \rho[\overline{x} \mapsto \overline{v}]), \text{ if } |\overline{x}| = |\overline{v}| \\
\textsf{\{}\overline{s:v}, s_i:v, \overline{s:v'}\textsf{\}[}s_i\textsf{]} &\to v \\
\textsf{\{}\overline{s:v}\textsf{\}[}s_x\textsf{]} &\to \textsf{undef}, \text{ if } s_x \notin \overline{s} \\
\textsf{\{}\overline{s:v}, s_i:v_i, \overline{s:v'}\textsf{\}[}s_i\textsf{] = }v &\to \textsf{\{}\overline{s:v}, s_i:v, \overline{s:v'}\textsf{\}} \\
\textsf{\{}\overline{s:v}\textsf{\}[}s_x\textsf{] = }v &\to \textsf{\{}\overline{s:v}\textsf{\}}, \text{ if } s_x \notin \overline{s} \\
\textsf{del \{}\overline{s:v}, s_i:v_i, \overline{s:v'}\textsf{\}[}s_i\textsf{]} &\to \textsf{\{}\overline{s:v}, \overline{s:v'}\textsf{\}} \\
\textsf{del \{}\overline{s:v}\textsf{\}[}s_x\textsf{]} &\to \textsf{\{}\overline{s:v}\textsf{\}}, \text{ if } s_x \notin \overline{s} \\
\textsf{if(true)\{}c_1\textsf{\}\{}c_2\textsf{\}} &\to c_1 \\
\textsf{if(false)\{}c_1\textsf{\}\{}c_2\textsf{\}} &\to c_2 \\
v\textsf{;}c &\to c \\
\textsf{while(}c_1\textsf{)\{}c_2\textsf{\}} &\to \textsf{if(}c_1\textsf{)\{}c_2\textsf{;while(}c_1\textsf{)\{}c_2\textsf{\}\}\{undef\}} \\
\textsf{try \{}v\textsf{\} catch (}x\textsf{)\{}c\textsf{\}} &\to v \\
\textsf{try \{}v\textsf{\} finally \{}c\textsf{\}} &\to c\textsf{;}v \\
\ell\textsf{:\{ }v\textsf{ \}} &\to v \\
op_n\textsf{(}v_1 \ldots v_n\textsf{)} &\to \delta_n(op_n, v_1 \ldots v_n)
\end{aligned}
$$

Context-sensitive, store-sensitive rules:

$$
\begin{aligned}
\langle \sigma, \mathcal{E}[c] \rangle &\to \langle \sigma, \mathcal{E}[c'] \rangle, \text{ if } c \to c' \\
\langle \sigma, \mathcal{E}[\textsf{ref } v] \rangle &\to \langle \sigma[a \mapsto v], \mathcal{E}[a] \rangle, \text{ where } a \notin dom(\sigma) \\
\langle \sigma, \mathcal{E}[\textsf{deref } a] \rangle &\to \langle \sigma, \mathcal{E}[v] \rangle, \text{ if } \sigma(a) = v \\
\langle \sigma, \mathcal{E}[a = v] \rangle &\to \langle \sigma[a \mapsto v], \mathcal{E}[v] \rangle \\
\langle \sigma, \mathcal{C}[\textsf{throw } v] \rangle &\to \langle \sigma, \textsf{err } v \rangle \\
\langle \sigma, \mathcal{E}[\textsf{try \{}\mathcal{C}[\textsf{throw } v]\textsf{\} catch (}x\textsf{)\{}(e, \rho)\textsf{\}}] \rangle &\to \langle \sigma, \mathcal{E}[(e, \rho[x \mapsto v])] \rangle \\
\langle \sigma, \mathcal{E}[\textsf{try \{}\mathcal{C}[\textsf{throw } v]\textsf{\} finally \{}c\textsf{\}}] \rangle &\to \langle \sigma, \mathcal{E}[c\textsf{;throw } v] \rangle \\
\langle \sigma, \mathcal{E}[\ell\textsf{:\{ }\mathcal{C}[\textsf{throw } v]\textsf{ \}}] \rangle &\to \langle \sigma, \mathcal{E}[\textsf{throw } v] \rangle \\
\langle \sigma, \mathcal{E}[\textsf{try \{}\mathcal{C}[\textsf{break } \ell\ v]\textsf{\} catch (}x\textsf{)\{}c\textsf{\}}] \rangle &\to \langle \sigma, \mathcal{E}[\textsf{break } \ell\ v] \rangle \\
\langle \sigma, \mathcal{E}[\textsf{try \{}\mathcal{C}[\textsf{break } \ell\ v]\textsf{\} finally \{}c\textsf{\}}] \rangle &\to \langle \sigma, \mathcal{E}[c\textsf{;break } \ell\ v] \rangle \\
\langle \sigma, \mathcal{E}[\ell\textsf{:\{ }\mathcal{C}[\textsf{break } \ell\ v]\textsf{ \}}] \rangle &\to \langle \sigma, \mathcal{E}[v] \rangle \\
\langle \sigma, \mathcal{E}[\ell'\textsf{:\{ }\mathcal{C}[\textsf{break } \ell\ v]\textsf{ \}}] \rangle &\to \langle \sigma, \mathcal{E}[\textsf{break } \ell\ v] \rangle, \text{ if } \ell' \neq \ell
\end{aligned}
$$

Fig. 2: Reduction semantics for $\lambda\rho_{JS}$

$$
\begin{aligned}
\mathcal{C} ::= {} & [\,] \mid \textsf{let (}x = \mathcal{C}\textsf{) } c \mid \mathcal{C}\textsf{(}\overline{c}\textsf{)} \mid v\textsf{(}\overline{v}, \mathcal{C}, \overline{c}\textsf{)} \mid \textsf{\{}\overline{s:v}, s:\mathcal{C}, \overline{s:c}\textsf{\}} \mid op\textsf{(}\overline{v}, \mathcal{C}, \overline{c}\textsf{)} \\
& \mid \mathcal{C}\textsf{[}c\textsf{]} \mid v\textsf{[}\mathcal{C}\textsf{]} \mid \mathcal{C}\textsf{[}c\textsf{] = }c \mid v\textsf{[}\mathcal{C}\textsf{] = }c \mid v\textsf{[}v\textsf{] = }\mathcal{C} \mid \textsf{del } \mathcal{C}\textsf{[}c\textsf{]} \mid \textsf{del } v\textsf{[}\mathcal{C}\textsf{]} \\
& \mid \textsf{ref } \mathcal{C} \mid \textsf{deref } \mathcal{C} \mid \mathcal{C} = c \mid v = \mathcal{C} \mid \textsf{if(}\mathcal{C}\textsf{)\{}c\textsf{\}\{}c\textsf{\}} \mid \mathcal{C}\textsf{;}c \mid \textsf{throw } \mathcal{C} \mid \textsf{break } \ell\ \mathcal{C} \\
\mathcal{D} ::= {} & [\,] \mid \textsf{try \{}\mathcal{C}[\mathcal{D}]\textsf{\} finally \{}c\textsf{\}} \mid \ell\textsf{:\{ }\mathcal{C}[\mathcal{D}]\textsf{ \}} \mid \textsf{try \{}\mathcal{C}[\mathcal{D}]\textsf{\} catch (}x\textsf{)\{}c\textsf{\}} \\
\mathcal{E} ::= {} & \mathcal{C}[\mathcal{D}]
\end{aligned}
$$

Fig. 3: Evaluation contexts for $\lambda\rho_{JS}$

outermost exception handler, finalizer, or label; and *general contexts* $\mathcal{E}$ range over all evaluation contexts.

The distinction is made to describe the behavior of $\lambda\rho_{JS}$'s control constructs: breaks, finalizers, and exceptions. When an exception is thrown, the enclosing local context is discarded and if the nearest enclosing control context is an exception handler, the thrown value is given to the handler:[5]

$$\mathcal{E}[\textsf{try } \{\mathcal{C}[\textsf{throw } v]\} \textsf{ catch } (x)\{(e, \rho)\}] \longmapsto \mathcal{E}[(e, \rho[x \mapsto v])].$$

If the nearest enclosing control context is a finalizer, the finalization clause is evaluated and the exception is rethrown:

$$\mathcal{E}[\textsf{try } \{\mathcal{C}[\textsf{throw } v]\} \textsf{ finally } \{c\}] \longmapsto \mathcal{E}[c; \textsf{throw } v].$$

If the nearest enclosing control context is a label, the context up to and including the label are discarded and the thrown value continues outward toward the next control context:

$$\mathcal{E}[\ell : \{\ \mathcal{C}[\textsf{throw } v]\ \}] \longmapsto \mathcal{E}[\textsf{throw } v].$$

Finally, if there is no enclosing control context, the exception was not handled and the program has resulted in an error:

$$\mathcal{C}[\textsf{throw } v] \longmapsto \textsf{err } v.$$

Breaks are handled in a similar way, except local contexts are discarded until the matching label are found. In the case of finalizers, the finalization clause is run, followed by reinvoking the break.

The result of a computation is an *answer*, which consists of a store and either a value or error, indicating an uncaught exception:

$$A ::= \langle \sigma, v \rangle \mid \langle \sigma, \textsf{err } v \rangle$$

The *evaluation* of a program is defined by a partial function relating programs to answers:

$$\mathit{eval}(e) = A \text{ if } \mathit{inj}_{JS}(e) \longmapsto\!\!\!\!\!\rightarrow A, \text{ for some } A,$$

where $\longmapsto\!\!\!\!\!\rightarrow$ denotes the reflexive, transitive closure of the reduction relation defined in figure 2 and

$$\mathit{inj}_{JS}(e) = \langle \varnothing, (e, \varnothing) \rangle.$$

Having established the syntax and semantics of the $\lambda\rho_{JS}$-calculus, we now relate it to Guha *et al.*'s λ_{JS}-calculus.

[5] We omit the store from these examples since they have no effect upon it.

2.3 Correspondence with λ_{JS}

We have developed an explicit substitution variant of λ_{JS} in order to derive an environment-based abstract machine, which as we will see, is important for the subsequent abstract interpretation. However, let us briefly discuss this new calculus's relation to λ_{JS} and establish their correspondence so that we can rest assured that our analytic framework is really reasoning about λ_{JS} programs.

Our presentation of evaluation contexts for $\lambda\rho_{JS}$ closely follows Guha, *et al.* There are two important differences.

1. The grammar of evaluation contexts for λ_{JS} makes a distinction between local contexts including labels, and local contexts including exception handlers. Let $\mathcal{F}$ and $\mathcal{G}$ denote such contexts, respectively:

$$\begin{aligned}\mathcal{F} &::= \mathcal{C} \mid \mathcal{C}[\ell\texttt{:\{}\ \mathcal{F}\ \texttt{\}}] \\ \mathcal{G} &::= \mathcal{C} \mid \mathcal{C}[\textsf{try}\ \texttt{\{}\mathcal{G}\texttt{\}}\ \textsf{catch}\ \texttt{(}x\texttt{)\{}c\texttt{\}}].\end{aligned}$$

   The distinction allows for exceptions to effectively jump over enclosing labels and for breaks to jump over handlers in one step of reduction:

$$\mathcal{E}[\textsf{try}\ \texttt{\{}\mathcal{F}[\textsf{throw}\ v]\texttt{\}}\ \textsf{catch}\ \texttt{(}x\texttt{)\{}(e,\rho)\texttt{\}}] \longmapsto \mathcal{E}[(e,\rho[x \mapsto v])],$$

   and

$$\begin{aligned}\mathcal{E}[\ell'\texttt{:\{}\ \mathcal{G}[\textsf{break}\ \ell\ v]\ \texttt{\}}] &\longmapsto \mathcal{E}[v], \text{ if } \ell' = \ell \\ &\longmapsto \mathcal{E}[\textsf{break}\ \ell\ v], \text{ otherwise.}\end{aligned}$$

   It should be clear that our notion of reduction can simulate the above one-step reductions in one or more steps corresponding to the number of labels (exception handlers) in $\mathcal{F}$ (in $\mathcal{G}$). We adopt our single notion of label and handler free local contexts in order to simplify the abstract machine in the subsequent section.
2. The grammar of evaluation of contexts for λ_{JS} mistakenly does not include break contexts in the set of local contexts, causing break expressions within break expression to get stuck, falsifying the soundness theorem. The mistake is minor and easily fixed. When relating $\lambda\rho_{JS}$ to λ_{JS} we assume this correction has been made.

We write λ_{JS} and $\lambda\rho_{JS}$ over a reduction relation to denote the (omitted) one-step reduction relation as given by Guha, *et al.*, corrected as described above, and the one-step reduction as defined in figure 2, respectively.

The results of λ_{JS} and $\lambda\rho_{JS}$ evaluation are related by a function $\mathcal{U}$ that recursively forces the all of the delayed substitutions represented by an environment [4, §2.5], thus mapping a value to a syntactic value. It is the identity function on syntactic values; for answers, functions, and records it is defined as:

$$\begin{aligned}\mathcal{U}(\langle\sigma, v\rangle) &= \langle\sigma, \mathcal{U}(v)\rangle \\ \mathcal{U}(\langle\sigma, \textsf{err}\ v\rangle) &= \langle\sigma, \textsf{err}\ \mathcal{U}(v)\rangle \\ \mathcal{U}(\textsf{fun}\texttt{(}\overline{x}\texttt{)\ \{}\ e\ \texttt{\}}, \{(x_0,v_0),\ldots,(x_n,v_n)\}) &= \textsf{fun}\texttt{(}x\texttt{)\ \{}\ [\mathcal{U}(v_0)/x_0,\ldots,\mathcal{U}(v_n)/x_n]e\ \texttt{\}} \\ \mathcal{U}(\texttt{\{}\overline{s\texttt{:}v}\texttt{\}}, \{(x_0,v_0),\ldots,(x_n,v_n)\}) &= \texttt{\{}\overline{s\texttt{:}[\mathcal{U}(v_0)/x_0,\ldots,\mathcal{U}(v_n)/x_n]v}\texttt{\}}.\end{aligned}$$

We can now formally state the calculi's correspondence:

Lemma 1 (Correspondence). *For all programs e,*

$$\langle \varnothing, e \rangle \overset{\lambda_{JS}}{\longmapsto\!\!\!\!\!\rightarrow} A \iff inj_{JS}(e) \overset{\lambda\rho_{JS}}{\longmapsto\!\!\!\!\!\rightarrow} A',$$

where $A = \mathcal{U}(A')$.

Proof. (Sketch.) The proof follows the structure of Biernacka and Danvy's [4] proof of correspondence for the λ-calculus and Curien's $\lambda\rho$-calculus of explicit substitutions [11], straightforwardly extended to λ_{JS} and $\lambda\rho_{JS}$.

We have now established our semantic basis: a calculus of explicit substitutions corresponding to λ_{JS}, which is a model adequate for all of JavaScript minus `eval`. In the following section, we apply the syntactic correspondence to derive a correct-by-construction environment-based abstract machine.

3 The JavaScript Abstract Machine (JAM)

In the section, we present the JAM: the JavaScript Abstract Machine. The abstract machine takes the form of a first-order state transition system that operates over triples consisting of a store, a closure, and a control stack, represented by a list of evaluation contexts. There are three classes of transitions for the machine: those that evaluate, those that continue, and those that apply.

evaluate: Evaluation transitions operate over triples dispatching on the closure component. The eval transitions implement a search for a redex or a value. If the closure is a value, then the search for a value is complete; the machine transitions to a continue state to plug the value into its context. Alternatively, if the closure is a redex, then the search is also complete and the machine transitions to an apply state to contract the redex. Finally, if the closure is a neither a redex nor a value, the search continues; the machine selects the next closure to search and pushes a single evaluation context on to the control stack.

continue: Continuation transitions operate over triples where the closure component is always a value, dispatching on the top evaluation context. The value is being plugged into the context represented by the stack. If plugging the value into the context results in a redex, the machine transitions to an apply state. If plugging the value reveals the next closure that needs to be evaluated, the machine transitions to an evaluate state. If plugging the value in turn results in *another* value, the machine transitions to a continue state to plug that value. Finally, if both the control stack is empty, the result of the program has been reached and the machine halts with the answer.

apply: Application transitions operate over triples where the closure component is always a redex. These transitions dispatch (mostly) on the redex and serve to contract it, thus implementing the reduction relation of figure 2. Since

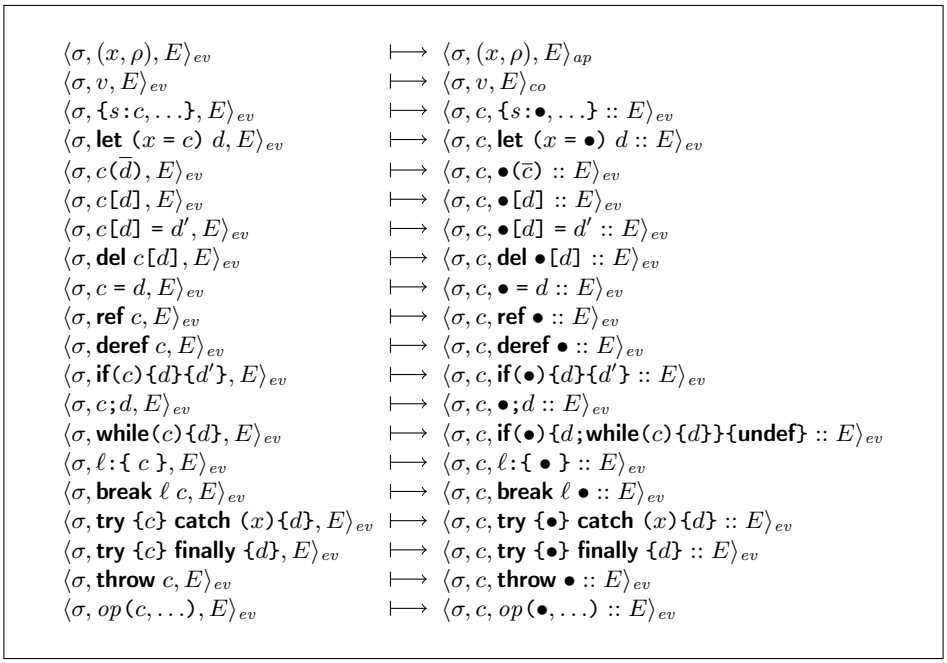


Fig. 4: Evaluation transitions

reductions are potentially store- and context-sensitive, the transitions may also dispatch on the control continuation in order to implement the control operators.

The machine relies on three functions for interacting with the store:

$$\begin{aligned} \mathit{alloc} &: \mathit{State} \to \mathit{Address}^n \\ \mathit{put} &: \mathit{Store} \times \mathit{Address} \times \mathit{Value} \to \mathit{Store} \\ \mathit{get} &: \mathit{Store} \times \mathit{Address} \to \mathcal{P}(\mathit{Value}) \end{aligned}$$

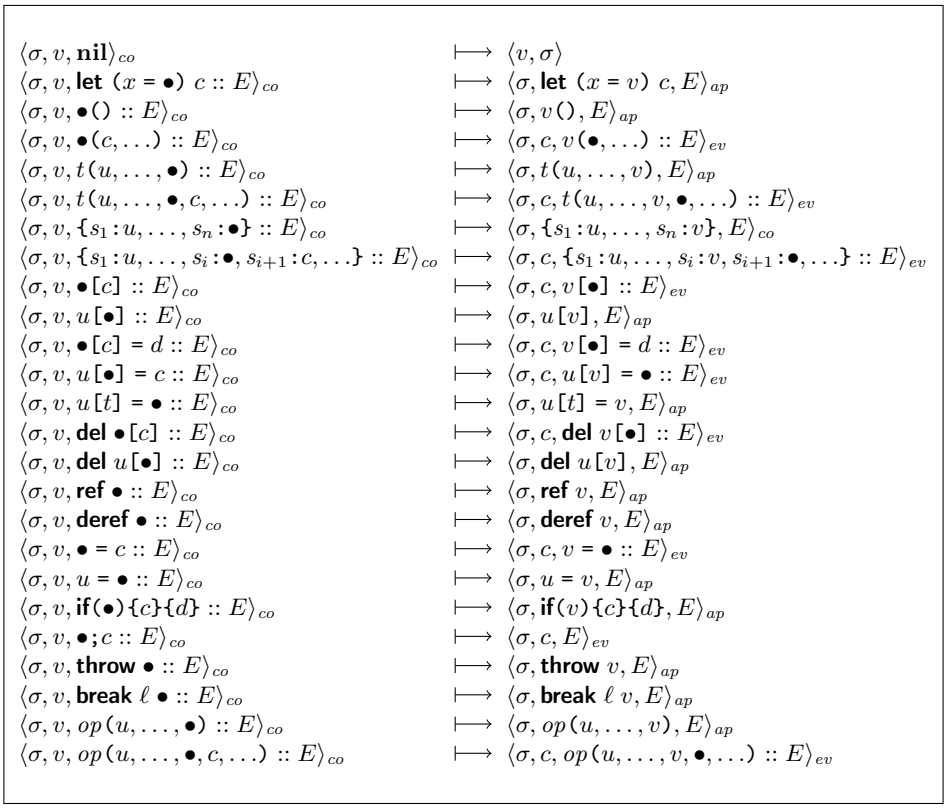
The *alloc* function makes explicit the non-deterministic choice of addresses when allocating space in the store. It returns a vector of addresses, often a singleton, based on the current state of the machine. For the moment, all that we require of *alloc* is that it return a suitable number of addresses and that none are in use in the store. The *put* function updates a store location and is defined as:

$$\mathit{put}(\sigma, \overline{a}, \overline{v}) = \sigma[\overline{a} \mapsto \overline{v}],$$

and the *get* function retrieves a value from a store location as a singleton set:

$$\mathit{get}(\sigma, a) = \{\sigma(a)\}.$$

We make explicit the use of these three functions because they will form the essential mechanism of abstracting the machine in the subsequent section; the



$\langle \sigma, v, \mathbf{nil} \rangle_{co}$	$\longmapsto$	$\langle v, \sigma \rangle$
$\langle \sigma, v, \mathsf{let}\ (x = \bullet)\ c :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, \mathsf{let}\ (x = v)\ c, E \rangle_{ap}$
$\langle \sigma, v, \bullet() :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, v(), E \rangle_{ap}$
$\langle \sigma, v, \bullet(c, \ldots) :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, c, v(\bullet, \ldots) :: E \rangle_{ev}$
$\langle \sigma, v, t(u, \ldots, \bullet) :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, t(u, \ldots, v), E \rangle_{ap}$
$\langle \sigma, v, t(u, \ldots, \bullet, c, \ldots) :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, c, t(u, \ldots, v, \bullet, \ldots) :: E \rangle_{ev}$
$\langle \sigma, v, \{s_1 : u, \ldots, s_n : \bullet\} :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, \{s_1 : u, \ldots, s_n : v\}, E \rangle_{co}$
$\langle \sigma, v, \{s_1 : u, \ldots, s_i : \bullet, s_{i+1} : c, \ldots\} :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, c, \{s_1 : u, \ldots, s_i : v, s_{i+1} : \bullet, \ldots\} :: E \rangle_{ev}$
$\langle \sigma, v, \bullet[c] :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, c, v[\bullet] :: E \rangle_{ev}$
$\langle \sigma, v, u[\bullet] :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, u[v], E \rangle_{ap}$
$\langle \sigma, v, \bullet[c] = d :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, c, v[\bullet] = d :: E \rangle_{ev}$
$\langle \sigma, v, u[\bullet] = c :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, c, u[v] = \bullet :: E \rangle_{ev}$
$\langle \sigma, v, u[t] = \bullet :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, u[t] = v, E \rangle_{ap}$
$\langle \sigma, v, \mathsf{del}\ \bullet[c] :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, c, \mathsf{del}\ v[\bullet] :: E \rangle_{ev}$
$\langle \sigma, v, \mathsf{del}\ u[\bullet] :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, \mathsf{del}\ u[v], E \rangle_{ap}$
$\langle \sigma, v, \mathsf{ref}\ \bullet :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, \mathsf{ref}\ v, E \rangle_{ap}$
$\langle \sigma, v, \mathsf{deref}\ \bullet :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, \mathsf{deref}\ v, E \rangle_{ap}$
$\langle \sigma, v, \bullet = c :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, c, v = \bullet :: E \rangle_{ev}$
$\langle \sigma, v, u = \bullet :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, u = v, E \rangle_{ap}$
$\langle \sigma, v, \mathsf{if}(\bullet)\{c\}\{d\} :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, \mathsf{if}(v)\{c\}\{d\}, E \rangle_{ap}$
$\langle \sigma, v, \bullet; c :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, c, E \rangle_{ev}$
$\langle \sigma, v, \mathsf{throw}\ \bullet :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, \mathsf{throw}\ v, E \rangle_{ap}$
$\langle \sigma, v, \mathsf{break}\ \ell\ \bullet :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, \mathsf{break}\ \ell\ v, E \rangle_{ap}$
$\langle \sigma, v, op(u, \ldots, \bullet) :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, op(u, \ldots, v), E \rangle_{ap}$
$\langle \sigma, v, op(u, \ldots, \bullet, c, \ldots) :: E \rangle_{co}$	$\longmapsto$	$\langle \sigma, c, op(u, \ldots, v, \bullet, \ldots) :: E \rangle_{ev}$

Fig. 5: Continuation transitions

slightly strange definition for *get* is to facilitate approximation where there may be multiple values residing at a store location.

The initial machine configuration is an evaluation state consisting of the empty store, the program, the empty environment, and the empty control and local continuation:

$$inj_{JAM}(e) = \langle \varnothing, (e, \varnothing), \mathbf{nil} \rangle_{ev}.$$

Final configurations are answers, just as in the reduction semantics.

3.1 Reformulation of reduction semantics

Unfortunately, there is an immediate problem with the described approach when applied to the JavaScript abstract machine. The problem stems from the JAM having two control stacks. Consequently, when abstracting we arrive at a two-stack pushdown machine, which in general has the power to simulate a Turing-machine. However this problem can be overcome: the JAM can be reformulated into a single stack machine in such a way that preserves correctness and enables a pushdown abstraction that is decidable.

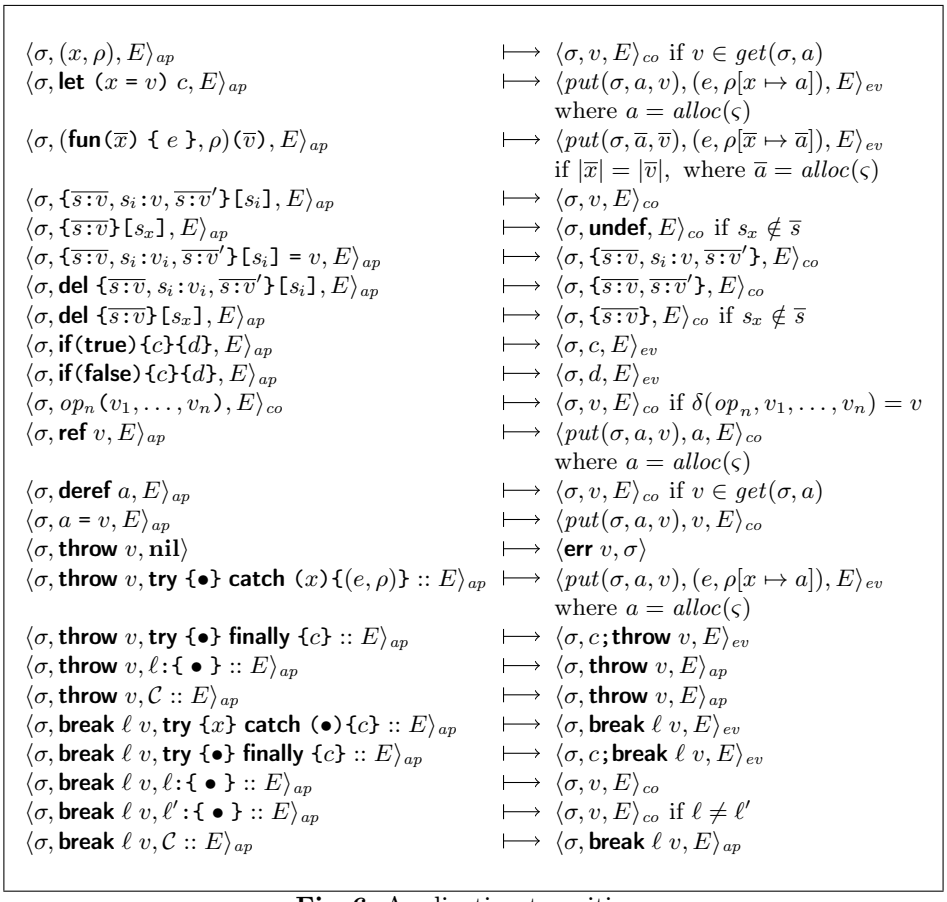


Fig. 6: Application transitions

One of the lessons of our abstract machine-based approach to analysis is that many problems in program analysis can be solved at the semantic level and then imported systematically to the analytic side. So for example, abstract garbage collection [27] can be expressed as concrete garbage collection with the pointer refinement and store abstraction applied [39]. Similarly, the exponential complexity of k-CFA can be avoided by concretely changing the representation of closures and then abstracting in an unremarkable way [29].

We likewise solve our two-stack problem by a reformulation at the level of the reduction semantics for $\lambda\rho_{JS}$ and then repeat the refocusing construction to derive a one-stack variant of the JAM.

The basic reason for maintaining the control and local stack is to allow jumps over the local context whenever a control operator is invoked. This is seen in the

$$
\begin{aligned}
\langle \sigma, \textsf{throw}\ v\rangle &\to \langle \sigma, \textsf{err}\ v\rangle \\
\langle \sigma, \mathcal{E}[\mathcal{S}[\textsf{throw}\ v]]\rangle &\to \langle \sigma, \mathcal{E}[\textsf{throw}\ v]\rangle \\
\langle \sigma, \mathcal{E}[\textsf{try \{throw}\ v\textsf{\} catch}\ (x)\{(e,\rho)\}]\rangle &\to \langle \sigma, \mathcal{E}[(e, \rho[x \mapsto v])]\rangle \\
\langle \sigma, \mathcal{E}[\textsf{try \{throw}\ v\textsf{\} finally}\ \{c\}]\rangle &\to \langle \sigma, \mathcal{E}[c\textsf{;throw}\ v]\rangle \\
\langle \sigma, \mathcal{E}[\ell\textsf{:\{ throw}\ v\ \textsf{\}}]\rangle &\to \langle \sigma, \mathcal{E}[\textsf{throw}\ v]\rangle \\
\langle \sigma, \mathcal{E}[\mathcal{S}[\textsf{break}\ \ell\ v]]\rangle &\to \langle \sigma, \mathcal{E}[\textsf{break}\ \ell\ v]\rangle \\
\langle \sigma, \mathcal{E}[\textsf{try \{break}\ \ell\ v\textsf{\} catch}\ (x)\{c\}]\rangle &\to \langle \sigma, \mathcal{E}[\textsf{break}\ \ell\ v]\rangle \\
\langle \sigma, \mathcal{E}[\textsf{try \{break}\ \ell\ v\textsf{\} finally}\ \{c\}]\rangle &\to \langle \sigma, \mathcal{E}[c\textsf{;break}\ \ell\ v]\rangle \\
\langle \sigma, \mathcal{E}[\ell\textsf{:\{ break}\ \ell\ v\ \textsf{\}}]\rangle &\to \langle \sigma, \mathcal{E}[v]\rangle \\
\langle \sigma, \mathcal{E}[\ell'\textsf{:\{ break}\ \ell\ v\ \textsf{\}}]\rangle &\to \langle \sigma, \mathcal{E}[\textsf{break}\ \ell\ v]\rangle, \text{ if } \ell' \neq \ell
\end{aligned}
$$

Fig. 7: Reformulated reduction semantics

reduction semantics with reductions such as this:

$$
\begin{aligned}
\mathcal{E}[\ell'\textsf{:\{}\ \mathcal{C}[\textsf{break}\ \ell\ v]\ \textsf{\}}] &\longmapsto \mathcal{E}[v] \text{ if } \ell' = \ell \\
&\longmapsto \mathcal{E}[\textsf{break}\ \ell\ v] \text{ if } \ell' \neq \ell
\end{aligned}
$$

To enable a single stack, we simulate this jump over the local context by "bubbling" over it in a piecemeal fashion. This is accomplished by defining a notion of single, non-empty local context frames $\mathcal{S}$, *i.e.*,

$$
\mathcal{S} ::= \textsf{let}\ (x = \bullet)\ c \mid \bullet(\overline{c}) \mid c(\overline{v}, \bullet, \overline{c}) \mid \cdots \mid \textsf{break}\ \ell\ \bullet \mid op(\overline{v}, \bullet, \overline{c})
$$

then the reduction relation for control operators remains context sensitive, but does not operate over whole contexts, but just the enclosing frame, which can then be implemented with a stack. The rules for simulating the above reduction are then:

$$
\begin{aligned}
\mathcal{E}[\mathcal{S}[\textsf{break}\ \ell\ v]] &\longmapsto \mathcal{E}[\textsf{break}\ \ell\ v] \\
\mathcal{E}[\ell'\textsf{:\{ break}\ \ell\ v\ \textsf{\}}] &\longmapsto \mathcal{E}[v] \text{ if } \ell' = \ell \\
&\longmapsto \mathcal{E}[\textsf{break}\ \ell\ v] \text{ if } \ell' \neq \ell
\end{aligned}
$$

Clearly, these reductions simulate the original single reduction by a number of steps that corresponds to the number of local frames between the label and the break.

The complete replacement of the context-sensitive reductions is given in figure 7. We refer to this alternative reduction semantics as $\lambda\rho'_{JS}$.

Lemma 2. *For all programs e,*

$$
inj_{JS}(e) \overset{\lambda\rho_{JS}}{\longmapsto\!\!\!\!\!\rightarrow} A \iff inj_{JS}(e) \overset{\lambda\rho'_{JS}}{\longmapsto\!\!\!\!\!\rightarrow} A.
$$

Guha *et al.* handle primitive operations in λ_{JS} in standard fashion by delegating to a δ-function. For the sake of analysis, we can delegate to any sound, finite abstraction of the δ-function. The simplest such abstraction maps values

to their types, which makes the abstract δ function isomorphic to its intensional signature. For in-depth discussion of richer abstract domains over basic values for use in JavaScript, we refer the reader to Jensen *et al.* [21]; they provide abstract domains for JavaScript which could be plugged directly into the $\widehat{JAM}$.

3.2 Correctness of the JAM

The JAM is a correct evaluator for $\lambda\rho_{JS}$, and hence for λ_{JS} as well.

Lemma 3 (Correctness). *For all programs e,*

$$inj_{JS}(e) \stackrel{\lambda_{JS}}{\longmapsto\!\!\!\!\!\rightarrow} A \iff inj_{JAM}(e) \stackrel{JAM}{\longmapsto\!\!\!\!\!\rightarrow} A.$$

Proof. (Sketch.) The correctness of the machine follows from the correctness of refocusing [13] and the (trivial) meaning preservation of subsequent transformations.

The detailed step-by-step transformation from the reduction semantics to the abstract machine has been carried out in the meta-language of SML.

For the purposes of program analysis, we rely on the following definition of a program's reachable machine states, where ς ranges over states:

$$JAM(e) = \{\varsigma \mid inj_{JAM}(e) \stackrel{JAM}{\longmapsto\!\!\!\!\!\rightarrow} \varsigma\}.$$

The set $JAM(e)$ is potentially infinite and membership is clearly undecidable. In the next section, we devise a sound and computable approximation to the set $JAM(e)$ by a family of pushdown automata.

4 Pushdown abstractions of JavaScript

To model non-local control precisely, the analysis must model the program stack precisely. Yet, the program stack can grow without bound—a substantial obstacle for the finite-state framework. Pushdown abstraction maps that program stack onto the unbounded stack of a pushdown system. Because the analysis inflicts a finite-state abstraction on the *control states* of the pushdown system, the analysis remains decidable.

The idea is that by bounding the store, the control stack, while unbounded, will consist of a finite stack alphabet. Since the remaining components of the machine are finite, the abstract machine is equivalent in computational power to a pushdown automaton, and thus reachability questions are casts naturally in terms of decidable PDA reachability properties.

4.1 Bounding the store, not the stack

$$\widehat{JAM}(e) = \{\hat{\varsigma} \mid inj_{JAM}(e) \overset{\widehat{JAM}}{\longmapsto\!\!\!\!\!\rightarrow} \hat{\varsigma}\}.$$

The abstracted JAM provides a sound simulation of the JAM and, by lemmas 1 and 3, a sound simulation of $\lambda\rho_{JS}$, and λ_{JS}, as well.

The machine's state-space is bounded simply by restricting the set of allocatable addresses to a fixed set of finite size, $\widehat{Address}$. This necessitates a change in the machine transition system and the representation of states. The machine can no longer restrict allocated addresses to be fresh with respect to the domain of the store as is the case when bindings are allocated, ref-expression are evaluated, and continuations are pushed. Instead, the machine calls an allocation function that returns a member of the finite set of addresses. Since the allocation function may return an address already in use, the behavior of the store must change to accommodate multiple values residing in a given location. We let $\hat{\sigma}$ range over such stores:

$$\hat{\sigma} \in \widehat{Store} = \widehat{Address} \rightarrow_{\text{fin}} \mathcal{P}(Value).$$

We let $\widehat{JAM}$ denote the abstract machine that results from replacing all occurrences of the functions *alloc*, *put*, and *get*, with the following counterparts:

$$\begin{aligned}
\widehat{alloc} &: State \rightarrow \widehat{Address}^n \\
\widehat{put} &: \widehat{Store} \times \widehat{Address}^n \times Value^n \rightarrow \widehat{Store} \\
\widehat{get} &: \widehat{Store} \times \widehat{Address} \rightarrow \mathcal{P}(Value)
\end{aligned}$$

The $\widehat{alloc}$ function works like *alloc*, but produces addresses from the finite set $\widehat{Address}$. The $\widehat{put}$ function updates a store location by *joining* the given value to any existing values that reside at that address:

$$\widehat{put}(\hat{\sigma}, a, v) = \hat{\sigma}[a \mapsto \{v\} \cup \hat{\sigma}(a)].$$

Joining rather than updating is critical for maintaining soundness.

In essence, the finiteness of the address space implies collisions may occur in the store. By joining, we ensure these collisions are modelled safely. The $\widehat{get}$ function returns the set of values at a store location, allowing a non-deterministic choice of values at that location.

We can formally relate the JAM to its abstracted counterpart through the natural structural abstraction map α on their state-spaces. This map recurs over the state-space of the JAM to inflict a finitizing abstraction at the leaves its state-space—addresses and primitive values—and structures that cannot soundly absorb that finitization, which in this case, is only the store. The range of the store expands into a power set, so that when an abstract address is *re*-allocated, it can hold both the existing values and the newly added value; formally:

$$\alpha(\sigma) = \lambda\hat{a}. \bigsqcup_{\alpha(a)=\hat{a}} \alpha(\sigma(a)).$$

Theorem 1 (Soundness). *If* $\varsigma \overset{JAM}{\longmapsto} \varsigma'$ *and* $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$*, then there exists an abstract state* $\hat{\varsigma}'$*, such that* $\hat{\varsigma} \overset{\widehat{JAM}}{\longmapsto} \hat{\varsigma}'$ *and* $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$*.*

Proof. We reason by case analysis on the transition. In the cases where the transition is deterministic, the result follows by calculation. For the the remaining non-deterministic cases, we must show an abstract state exists such that the simulation is preserved. By examining the rules for these cases, we see that all hinge on the abstract store in $\hat{\varsigma}$ soundly approximating the concrete store in ς, which follows from the assumption that $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$.

The more interesting aspect of the pushdown abstraction is decidability. Notice that since the stack has a recursive, unbounded structure, the state-space of the machine is potentially infinite so deciding reachability by enumerating the reachable states will no longer suffice.

Theorem 2 (Decidability). $\hat{\varsigma} \in \widehat{JAM}(e)$ *is decidable.*

Proof. States of the abstracted JAM consist of a store, a closure, and a list of single evaluation contexts representing the control stack. Observe that with the exception of the stack, each of these sets is finite: for a given program, there are a fixed set of expressions; environments are finite since they map variables to addresses and the address space is bounded; since expressions and environments are finite, so too are the set of values; stores are finite since addresses and values are finite.

For the machine transitions that dispatch on the control stack, only the topmost element is used and stack operations always either push or pop a single context frame on at a time, *i.e.*, the machine obeys a stack discipline. The stack alphabet consists of single evaluation contexts, which include a number of expressions, a value, or an environment, all of which are finite sets. Thus the stack alphabet is finite. Consequently the machine is a pushdown automaton and decidability follows from known results on pushdown automata.

4.2 Instantiations

We have now described the design of a sound and decidable framework for the pushdown analysis of JavaScript. The framework has a single point of control for governing the precision of an analysis, namely the $\widehat{alloc}$ function. The restrictions on acceptable $\widehat{alloc}$ functions are fairly liberal: *any* allocation policy is sound, and so long as the policy draws from a finite set of addresses for a given program, the analysis will be decidable.

At its simplest, the $\widehat{alloc}$ function could produce a constant address:

$$\widehat{alloc}(\varsigma) = \mathtt{a}.$$

A more refined analysis can be obtained by a more refined allocation policy. 0CFA for example, distinguishes bindings of differently named variables, but

merges all bindings for a given variable name. The allocation function corresponding to this strategy is:

$$\widehat{alloc}(\langle \sigma, \textsf{let (}x\textsf{ = }v\textsf{) } c, E\rangle_{ap}) = x$$
$$\widehat{alloc}(\langle \sigma, (\textsf{fun(}\overline{x}\textsf{) \{ }e\textsf{ \}}, \rho)(\overline{v}), E\rangle_{ap}) = \overline{x}$$
$$\widehat{alloc}(\langle \sigma, \textsf{throw } v, \textsf{try \{•\} catch (}x\textsf{)\{}c\textsf{\}} :: E\rangle_{ap}) = x$$

This strategy uses variables names as addresses and always allocates the variable names being bound. The strategy is finite for a given program and produces a pushdown generalization of classical 0CFA. Moreover, the state-space simplifies greatly since it can be observed that under this strategy, every environment is the identity function on variable names. Thus environments could be eliminated from the semantics.

There is still the need to designin a heap abstraction, *i.e.*, what should the allocation function produce for:

$$\widehat{alloc}(\langle \sigma, \textsf{ref } v, E\rangle_{ap})?$$

Shivers' original formulation of 0CFA had a very simple heap abstraction corresponding to the constant allocation function above [36]. More refined heap abstractions are obtained by simply designing better strategies for this case of $\widehat{alloc}$.

The k-CFA hierarchy, of which 0CFA is the base, refines the above allocation policy by pairing variables together with bounded history of the calling context at the binding site of a variable. Such an abstraction is easily expressible in our framework as follows:

$$\widehat{alloc}(\langle \sigma, \textsf{let (}x\textsf{ = }v\textsf{) } c, E\rangle_{ap}) = \langle x, \lfloor E \rfloor_k \rangle$$
$$\widehat{alloc}(\langle \sigma, (\textsf{fun(}\overline{x}\textsf{) \{ }e\textsf{ \}}, \rho)(\overline{v}), E\rangle_{ap}) = \langle \overline{x}, \lfloor E \rfloor_k \rangle$$
$$\widehat{alloc}(\langle \sigma, \textsf{throw } v, \textsf{try \{•\} catch (}x\textsf{)\{}c\textsf{\}} :: E\rangle_{ap}) = \langle x, \lfloor E \rfloor_k \rangle,$$

where

$$\lfloor \textbf{nil} \rfloor_k = \textbf{nil}$$
$$\lfloor E \rfloor_0 = \textbf{nil}$$
$$\lfloor \mathcal{E} :: E \rfloor_{k+1} = \mathcal{E} :: \lfloor E \rfloor_k.$$

This strategy uses variable names paired together with a fixed depth view of the control stack to allocate bindings. It is easy to vary this strategy for various k-CFA like abstraction, *e.g.*, taking the top k *different* stack frames, or taking the top k application frames by filtering out non-application frames.

By giving alternative definitions of $\widehat{alloc}$ it is straightforward to design pushdown versions of other known analyses such as CPA [1], sub-0CFA [2], and m-CFA [28].

4.3 Implementation

To empirically substantiate our formal claims, we have developed executable models and test beds. We have developed the reduction semantics of $\lambda\rho_{JS}$ in Standard ML (SML) and carried out the refocusing construction and subsequent program transformations in a step-by-step manner closely following the lecture notes of Danvy [12]. We found using SML as a metalanguage helpful due to its type system and non-exhaustive and redundant pattern matching warnings. For example, we were able to encode Guha *et al.*'s soundness theorem, which is false without the modification to the semantics as described in section 2.3, in SML in such a way that the type of the one-step reduction relation, coupled with exhaustive pattern matching, implies a program is either a value or can make progress.

We ported our semantics and concrete machines to PLT Redex [15] and then built their abstractions. This was done because PLT Redex supports programming with relations and includes a property-based random testing mechanism. The support for programming with relations is an important aspect for building the non-deterministic transition systems of the abstracted JAM machines since, unlike their concrete counterparts, the transition system cannot be encoded as a function in a straightforward way. Using the random testing framework [23], we tested the correspondence, correctness, and soundness theorems. As an added benefit, we were able to visualize our test programs' state-spaces using the included graphical tools.

Finally, we used Guha *et al.*'s code for desugaring in order to test our framework on real JavaScript code. We tested against the same test bed as Guha *et al.*: a significant portion of the Mozilla JavaScript test suite; about 5,000 lines of unmodified code. We tested the closure-based semantics of $\lambda\rho_{JS}$ for correspondence against the substitution-based semantics of λ_{JS} and tested the machines for correctness with respect to the $\lambda\rho_{JS}$ semantics. Finally, we tested the instantiations of our analytic framework for soundness with respect to the machines. Since the semantics of λ_{JS} have been validated against the output of Rhino, V8, and SpiderMonkey, and all of semantic artifacts simulate or approximate λ_{JS}, these tests substantiate our framework's correctness.

5 Related work

Our approach fits cleanly within the progression of work in abstract interpretation [9, 10] and is inspired by the pioneering work on higher-order program analysis by Jones [22]. Like Jones, our work centers around machine-based abstractions of higher-order languages; and like Jones [35], we have obtained our machines by program transformations of high-level semantic descriptions in the tradition of Reynolds [33]. We have been able to leverage the refocusing approach of Danvy, *et al.*, to systematically derive such machines [13, 4, 12], and our main technical insight has been that threading bindings—but not continuations—through the store results in straightforward and clearly sound framework that

precisely reasons about control flow in face of higher-order functions and sophisticated control operators.

5.1 Pushdown analyses

The most closely related work to ours is Vardoulakis and Shivers recent work on CFA2 [40]. CFA2 is a table-driven summarization algorithm that exploits the balanced nature of calls and returns to improve return-flow precision in a control-flow analysis for CPS programs. Though CFA2 alludes to exploiting context-free languages, context-free languages are not explicit in its formulation in the same way that pushdown systems are in pushdown control-flow analysis [14]. With respect to CFA2, the pushdown analysis presented here is potentially polyvariant and in direct-style.

On the other hand, CFA2 distinguishes stack-allocated and store-allocated variable bindings, whereas our formulation of pushdown control-flow analysis does not and allocates all bindings in the store. If CFA2 determines a binding can be allocated on the stack, that binding will enjoy added precision during the analysis and is not subject to merging like store-allocated bindings.

Recently, Vardoulakis and Shivers have extended CFA2 to analyze programs containing the control operator `call-with-current-continuation` [41]. The operator, abbreviated `call/cc`, works as follows: at the point it is applied, it reifies the continuation as procedure; when that procedure is applied it aborts the call's continuation and installs the reified continuation. CFA2 is able to analyze this powerful control operator, which is able to encode all of the control operators considered here, but without the same guarantees of precision as this work is able to provide for the weaker notion of exceptions and breaks. So while CFA2 can analyze `call/cc`, it does so with the potential for loss of precision about the control stack; indeed, this appears to be inherently necessary for any computable analysis of `call/cc` as the operator does not obey a stack discipline. Vardoulakis has implemented CFA2 for JavaScript as the "Doctor JS" tool.[6]

The current work also draws on CFL- and pushdown-reachability analysis [5, 24, 31, 32]. CFL-reachability techniques have also been used to compute classical finite-state abstraction CFAs [25] and type-based polymorphic control-flow analysis [30]. *These analyses should not be confused with pushdown control-flow analysis*: our results demonstrate how to compute a fundamentally more precise kind of CFA, while the work on CFL-reachability has shown how to cast classical analyses, such as 0CFA, as a reachability problem for a context-free language.

5.2 JavaScript analyses

Thiemann [37] develops a type system for Core JavaScript, a restricted subset of JavaScript. The type system rules out the application of non-functions, applying primitive operations to values of incorrect base type, dereferencing fields of the `undefined` or `null` value, and referring to unbound variables. Jensen,

[6] `http://doctorjs.org/`

Møller, and Thiemann, [21] develop an abstract interpretation computing type inference. It builds on the type system of Thiemann [37], using it as inspiration for their abstract domains. Richards *et al.*'s landmark empirical survey of JavaScript code [34] made it clear that for JavaScript analyses to work in the wild, it is not sufficient to handle only a well-behaved core of the language. Capturing ill-behaved parts of JavaScript soundly and precisely was a major motivation for our research.

Subsequently, Heidegger and Thiemann have extended the type system with a notion of *recency* to improve precision [18] and Jensen *et al.*, have developed a technique of *lazy propagation* to increase the feasibility of the analysis [20]. Balakrishnan and Reps pioneered the idea of recency [3]: objects are considered recent when created and given a singleton type and treated flow-sensitively until "demoted" to a summary type that is treated flow-insensitively. Recency enables strong update in analyses [19], which is important for reasoning precisely about initialization patterns in JavaScript programs. Recency and lazy propagation are orthogonal to our analytic framework: in our recent work, we show how to incorporate a generalization of recency into a machine-based static analysis [26] through the concept of anodization.

Guha, Krishnamurthi and Jim [16] developed an analysis for intrusion-detection of JavaScript, driven in part by an adaptation of k-CFA to a large subset of JavaScript. Our work differs from their work in that we are formally guaranteeing soundness with respect to a concrete semantics, we provide fine-grained control over precision for our finite-state analysis and we also provide a push-down analysis for handling the complex non-local control features which pervade JavaScript code. (Guha *et al.* make a best-effort attempt at soundness and dynamically detect violations of soundness in empirical trials, violations which they use to refine their analysis.)

Chugh *et al.* [6] present a staged information-flow analysis of JavaScript. In effect, their algorithm partially evaluates the analysis with respect to the available JavaScript to produce a residual analysis. When more code becomes available, the residual analysis resumes. Our own framework is directly amenable to such partial evaluation for handling constructs like `eval`: explore the state-space aggressively, but do not explore past `eval` states. The resulting partial abstract transition graph is sound until the program encounters `eval`. At this point, the analysis may be resumed with the code supplied to `eval`.

6 Conclusions and perspective

We present a principled systematic derivation of machine-based analysis for JavaScript. By starting with an established formal semantics and transforming it into an abstract machine, we soundly capture JavaScript in full, quirks and all. The abstraction of this machine yields a robust finite-state framework for the static analysis of JavaScript, capable of instantiating the equivalent of traditional techniques such as k-CFA and CPA. Finding the traditional finite-state approach wanting in precision for JavaScript's extensive use of non-local

control, we extend the theory of systematic abstraction of abstract machines from finite-state to pushdown. These decidable pushdown machines precisely model the structure of the program stack, and so do not lose precision in the presence of control constructs that depend on it, such as recursion or complex exceptional control-flow.

`https://github.com/dvanhorn/jam/`

Acknowledgments We thank Arjun Guha for answering questions about λ_{JS}. Sam Tobin-Hochstadt and Mitchell Wand provided fruitful discussions and feedback on early versions of this work. We thank Olivier Danvy and Jan Midtgaard for their hospitality and the rich intellectual environment they provided during the authors' visit to Aarhus University.

References

1. Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 2–26. Springer-Verlag, 1995.
2. J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM TOPLAS*, 20(4):845–868, 1998.
3. Gogul Balakrishnan and Thomas Reps. Recency-Abstraction for Heap-Allocated storage. In Kwangkeun Yi, editor, *SAS '06: Static Analysis Symposium*, volume 4134 of *Lecture Notes in Computer Science*, pages 221–239. Springer-Verlag, 2006.
4. Malgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Trans. Comput. Logic*, 9(1):1–30, 2007.
5. Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to Model-Checking. In *CONCUR '97: Proceedings of the 8th International Conference on Concurrency Theory*, pages 135–150. Springer-Verlag, 1997.
6. Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. pages 50–62. ACM, June 2009.
7. John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Trans. Program. Lang. Syst.*, 26(6):1029–1052, November 2004.
8. William D. Clinger. Proper tail recursion and space efficiency. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 174–185. ACM, May 1998.
9. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.
10. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282. ACM Press, 1979.
11. P. L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82(2):389–402, May 1991.

12. Olivier Danvy. From Reduction-Based to Reduction-Free normalization. In Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra, editors, *Advanced Functional Programming, Sixth International School*, number 5382, pages 66–164. Springer, May 2008. Lecture notes including 70+ exercises.
13. Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Research Report BRICS RS-04-26, Department of Computer Science, Aarhus University, November 2004. A preliminary version appeared in the informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001), Electronic Notes in Theoretical Computer Science, Vol.˜59.4.
14. Christopher Earl, Matthew Might, and David Van Horn. Pushdown Control-Flow analysis of Higher-Order programs. In *Workshop on Scheme and Functional Programming*, 2010.
15. Matthias Felleisen, Robert B. Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, August 2009.
16. Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for ajax intrusion detection. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 561–570. ACM, 2009.
17. Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 126–150. Springer-Verlag, 2010.
18. Phillip Heidegger and Peter Thiemann. Recency types for analyzing scripting languages. In Theo D'Hondt, editor, *ECOOP 2010 Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, chapter 10, pages 200–224. Springer Berlin / Heidelberg, 2010.
19. Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: must-alias analysis for higher-order languages. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 329–341. ACM, 1998.
20. Simon Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis*, volume 6337 of *Lecture Notes in Computer Science*, chapter 20, pages 320–339. Springer Berlin / Heidelberg, 2011.
21. Simon H. Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*, volume 5673 of *SAS '09*, pages 238–255. Springer-Verlag, 2009.
22. Neil D. Jones. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, pages 114–128. Springer-Verlag, 1981.
23. Casey Klein, Matthew Flatt, and Robert B. Findler. Random testing for Higher-Order, stateful programs. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2010*, 2010.
24. John Kodumal and Alex Aiken. The set constraint/CFL reachability connection in practice. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 207–218. ACM, June 2004.
25. David Melski and Thomas W. Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248(1-2):29–98, October 2000.
26. Matthew Might. Shape analysis in the absence of pointers and structure. In *VMCAI 2010: International Conference on Verification, Model-Checking and Abstract Interpretation*, pages 263–278, January 2010.

27. Matthew Might and Olin Shivers. Exploiting reachability and cardinality in higher-order flow analysis. *Journal of Functional Programming*, 18(Special Double Issue 5-6):821–864, 2008.
28. Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 305–315. ACM Press, 2010.
29. Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 305–315. ACM, 2010.
30. Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66. ACM, 2001.
31. Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, December 1998.
32. Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1-2):206–263, 2005.
33. John C. Reynolds. Definitional interpreters for Higher-Order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Originally published in *ACM'72: Proceedings of the ACM Annual Conference*.
34. Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 1–12. ACM, 2010.
35. David A. Schmidt. State-transition machines, revisited. *Higher Order Symbol. Comput.*, 20:333–335, September 2007.
36. Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.
37. Peter Thiemann. Towards a type system for analyzing JavaScript programs. In Mooly Sagiv, editor, *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, chapter 28, page 140. Springer Berlin / Heidelberg, 2005.
38. David Van Horn and Harry G. Mairson. Deciding kCFA is complete for EXPTIME. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 275–282. ACM, 2008.
39. David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 51–62. ACM, 2010. ICFP'10.
40. Dimitrios Vardoulakis and Olin Shivers. CFA2: a Context-Free approach to Control-Flow analysis. *Logical Methods in Computer Science*, 7(2), May 2011.
41. Dimitrios Vardoulakis and Olin Shivers. Pushdown flow analysis of first-class control. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 69–80. ACM, 2011.
42. Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Trans. Softw. Eng. Methodol.*, 9(4):341–378, October 2000.