University of Wisconsin Milwaukee
UWM Digital Commons

Theses and Dissertations

August 2016

H-CFA: a Simplified Approach for Pushdown Control Flow Analysis

Fei Peng
University of Wisconsin-Milwaukee

Follow this and additional works at: https://dc.uwm.edu/etd

 Part of the Computer Sciences Commons

Recommended Citation
Peng, Fei, "H-CFA: a Simplified Approach for Pushdown Control Flow Analysis" (2016). *Theses and Dissertations*. 1300.
https://dc.uwm.edu/etd/1300

This Thesis is brought to you for free and open access by UWM Digital Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of UWM Digital Commons. For more information, please contact open-access@uwm.edu.

H-CFA: A SIMPLIFIED APPROACH FOR PUSHDOWN CONTROL FLOW ANALYSIS

by

Fei Peng

A Thesis Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Master of Science

in Computer Science

at

The University of Wisconsin-Milwaukee

August 2016

ABSTRACT

H-CFA: A SIMPLIFIED APPROACH FOR PUSHDOWN CONTROL FLOW ANALYSIS

by

Fei Peng

The University of Wisconsin-Milwaukee, 2016
Under the Supervision of Professor Tian Zhao

In control flow analysis (CFA), call/return mismatch is a problem that reduces analysis precision. So-called k-CFA uses bounded call-strings to obtain limited call/return matching, but it has a serious performance problem due to its coupling of call/return matching with context-sensitivity of values. CFA2 and PDCFA are the first two algorithms that bring pushdown (context-free reachability) approach to the CFA area, which provide perfect call/return mathcing. However, CFA2 and PDCFA both need significant engineering effort to implement. The abstracting abstract machine (AAM), a configurable framework for constructing abstract interpreters, introduces store-allocated continuations that make the soundness of abstract interpreters easily obtainable. Recently, two related approaches (AAC and

P4F) provide call/return matching using AAM by modeling the call-stack as a pushdown system. However, AAC incurs high overhead and is hard to understand, while P4F cannot compute monovariant analysis. To overcome the above shortcomings, we developed a new method, h-CFA, to address the call/return mismatch problem. h-CFA records the program execution history during abstract interpretation and uses it to avoid control flow merging that causes call/return mismatch. Our method uses AAM and is very easy to implement for ANF style program. ANF is a popular intermediate representation of programs that converts all complex intra-procedural control flows to linear let-bindings and sets a syntactic variable to each sub-expression. In addition, our method reveals an essential property of any pushdown CFA, which we exploited in the development of a static analyzer for JavaScript, named JsCFA. This application of the essential property avoids recording the program execution history, so source programs are no long required being the ANF form. Meanwhile, JsCFA adopts a technique to solve the environment problem or fake rebinding, which eliminates more defects of monovariant analysis. This, in cooperation with exact call/return matching, yield more precise analysis and better performance. Moreover, JsCFA supports a configurable interface to add context-sensitivity to selected areas of programs. JsCFA applies the interface to improve the analysis precision for runtime object extensions. Finally, we quantitatively evaluated the performance of JsCFA.

© Copyright by Fei Peng, 2016
All Rights Reserved

TABLE OF CONTENTS

Abstract **ii**

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**

2 Related Work **9**

3 Pushdown CFA in AAM **13**

3.1 Abstracting Abstract Machine 13

3.2 Store-widening 20

3.3 A Defect of AAM 21

4 Pushdown CFA based on Program Execution History **24**

4.1 Program Execution History 24

4.2 Polyvariant Continuation 28
4.3 Complexity and Precision of h-CFA 32

5 Design of JsCFA **37**

5.1 Syntax Interface 38
5.2 Transition Rules 39
5.3 Store and Stack 48
5.4 Functions, Methods, and Constructors 51
5.5 Configurable Context-Sensitivity and Adaptive Object-Sensitivity 54
5.6 Abstract Garbage Collection as Stack Filtering 56
5.7 Pushdown CFA without Program Execution History 60
5.7.1 The Essence of h-CFA 62
5.7.2 Abstract Garbage Collection as Popping Call Stack Frames 65
5.8 Evaluation 66

6 Future Work **68**

7 Conclusion **71**

Bibliography **72**

LIST OF FIGURES

1.1 An example program showing imprecision of 0-CFA 2

1.2 An example program showing imprecision of 1-CFA 3

4.1 A recursive program example in ANF 29

4.2 Monovariant analysis performance comparison 33

4.3 1-call-site sensitive analysis performance comparison 33

4.4 Monovariant analysis precision comparison 35

4.5 1-call-site sensitive analysis percision comparison 35

4.6 State transition graphs 36

5.1 Abstract syntax tree data types of statements 40

5.2 Abstract syntax tree data types of expressions 41

5.3 Abstract syntax tree data types of lvalue expressions 42

5.4 Parts of evaluation transition rules for dispatching and statements 44

5.5 Parts of evaluation transition rules for expressions 45

5.6 Parts of continuation transition rules for statements 46

5.7 Parts of continuation transition rules for expressions 47

5.8 Parts of application transition rules for statements 49

5.9 Parts of application transition rules for expressions 50

5.10 Application transition rule for global function calls 53

5.11 JsCFA benchmarks 67

LIST OF TABLES

4.1 Comparison of pushdown CFA algorithms 32

1 Introduction

Dynamic programming languages, such as JavaScript, Python, and Ruby, play a significant role in computing areas, such as system management, web development, and scientific computing. Therefore, developers who are using these languages increasingly demand tools for improving code quality, such as security auditing, error- checking, debugging, refactoring, and more. However, certain features of dynamic languages (e.g. duck-typing, first-class functions, and highly dynamic object models) make achieving these requirements difficult. For example, static programming languages are able to report certain semantic errors before executing programs, but in dynamic languages all the semantic errors just can be found during runtime, which is too risky for large-scale commercial software. To this end, control flow analysis [18] (CFA) has been used to detect deep semantic information before the actual running of programs written in dynamic languages.

CFA is a class of algorithms that give conservative approximation to inter-procedural information of programs before running them. Statically detecting the precise target of a function call is difficult for programs written in higher-order (functional) languages. To illustrate this problem, consider the following example in Scheme.

```
(let* ((id (lambda (x) x))
       (a (id 1)[1])
       (b (id #t)[2]))
    ...)
```

Figure 1.1: An example program showing imprecision of 0-CFA

```
(let* ((f (lambda (x) (x 1)))
       (g (lambda (y) (+ y 2)))
       (h (lambda (z) (+ z 3))))
    (+ (f g) (f h)))
```

In the body of function `f`, the call site `(x 1)` will transfer control to function bodies that variable `x` potentially refers to. However, the next step in the control flow is not obvious because `x` is the formal parameter of function `f` and will be bound to unknown values. Shivers invented k-CFA [26] as the first popular solution to the control flow problem. k-CFA applies an abstract interpretation [2] approach to simulate program execution statically and provides conservative approximations with a configurable hierarchy of precision. Shivers chose finite call-strings [25] to represent runtime contexts for the abstract interpretation. Call-strings with length of k record latest k call sites, which make the state space of k-CFA finite, and longer call-strings yield more precise analysis with higher overhead. 0-CFA is a special case of k-CFA that uses empty call-strings (k is zero).

```
(let* ((id (lambda (x) x))
       (f (lambda (y) (id y)[1]))
       (a (f 1)[2])
       (b (f #t)[3]))
    ...)
```

Figure 1.2: An example program showing imprecision of 1-CFA

Problems of CFA The 0-CFA and k-CFA without enough context information are imprecise for realistic programs. For example, call/return mismatch is always a problem in k-CFA that dramatically reduces the precision of analysis. Consider the trivial example in Figure 1.1, where, in 0-CFA, the `id` function is called twice and `#t` eventually flows into variable `a` because there is a spurious flow from call site `(id #t)` to `(id 1)`.

In 1-CFA ($k = 1$), the values of the local variable `x` are distinguished by different call site environments. In this example, the two calls to the `id` function are labeled with 1 and 2 respectively. Different versions of the variable x in different calls are separated by the call site labels. For example, $(x, [2]) \mapsto \{\#t\}$ because the value of `x` is `#t` at call site 2. Original k-CFA also uses the variables' environment to filter inter-procedural control flows, which means that the value of `x` from call site 2 only can be returned to `(id #t)`. In this case (a non-recursive program), call-string with size 1 is enough to provide precise call/return flow (both data and control flow). However, longer call sequences or recursive calls propagate spurious information to the whole program.

Consider the example in Figure 1.2, the `id` function is called by `f`, and there are two calls to function `f`. 1-CFA is no longer precise for this program because $(x, [1])$ can be generated by call site 2 and 3 both. Then abstract value `{1, #t}` eventually flows into variable `a` and `b`. In this example, 2-CFA can distinguish the two call sites, which $(x, [2, 1]) \mapsto \{1\}$ and $(x, [3, 1]) \mapsto \{\#t\}$ indicate correct data and control flows. Therefore, we can achieve precise call/return matching with a large enough k on non-recursive programs. However, recursive function invocations can make any call-string "overflow", which call-strings will be filled by duplicated recursive call sites and lose earlier context information. Particularly, the recursion is ubiquitously existing in functional programs. Meanwhile, the performance of k-CFA is unacceptable even when $k = 1$ [29].

In addition, k-CFA is tightly bound with call-site sensitivity, other context-sensitivity strategies [1, 21, 27, 17, 32] is hard to be applied.

Existing techniques There is a family of algorithms that attempt to perfectly match return flows with their true call site entries in static analysis, which is referred to as pushdown or context-free approach [23, 24]. CFA2 [31] is the first attempt that brings precise call/return matching to monovariant analysis in exponential time complexity. Because monovariant analysis still

merges too many data flows even if it has accurate inter-procedural control flows, CFA2 introduces *stack filtering* to eliminate the imprecision of local variables. Additionally, there are other three approaches (PDCFA [6], AAC [15], and P4F [11]) that provide accurate call/return matching by modeling the call-stack as a pushdown system. However, PDCFA and CFA2 need significant engineering effort to implement [11]. AAC and P4F is easy to implement in the abstracting abstract machine (AAM) [30] framework, but AAC incurs high overhead (see Section 2 and Section 4.3) and is difficult to understand while P4F cannot compute monovariant analysis and just has limited call/return matching strength.

A simplified approach In this paper, we introduce a new method to address the call/return mismatch problem. In terms of implementation, this method is as simple as writing concrete interpreters in CESK machine style [9]. It provides perfect call/return matching for monovariant and polyvariant control flow analysis. Since this method records *program execution histories* through the abstract interpretation process and uses it to encode continuation addresses, we name it h-CFA. The program execution history can be regarded as call-strings with automatically determined length. For non-recursive calls, the execution history always provides enough context information, no matter how deep the call sequence is. Moreover, the history

automatically stops growth for recursive calls while the worklist iteration (Section 3.1) is responsible for finding the fixed-point of recursive computation.

Application To verify the practicability of our theory, we implemented a static analyzer for a subset of JavaScript (ECMAScript 3) in Scala, and we call it JsCFA. JsCFA not only uses pushdown CFA but also adopts other techniques to improve analysis precision for real-world programs. JsCFA usually computes monovariant control-flow facts that incurs critical imprecision for realistic programs and libraries that are written in dynamic higher-order languages. For example, even the abstract interpreter can perfectly match call/return flows, monovariant or polyvariant analysis without enough context information may also generate spurious data flows from false environments, which is referred to as *environment problem* [26, 19]. To illustrate this problem, consider the analyzing process of Figure 1.1 again: assuming the analyzer always matches return flows with correct call sites, which function `id` called by call site 1 only returns to `a` and call site 2 only returns to `b`. Variable `a` will get abstract value `{1}`, but `{1, #t}` flows into variable `b` because the local variable `x` retains the value from call site 1 during abstractly interpreting call site 2. This spurious data flow injures the practicability of pushdown CFA and causes other control flow problems (see details

in Section 5.6). We solved this problem by introducing abstract garbage collection [20] on the value store into JsCFA. JsCFA works with abstract GC to remove those local bindings that pollute subsequent data flows, but abstract GC collaborating with k-CFA is not safe [20]. Meanwhile, we also applied abstract GC on the continuation store, which indirectly implements h-CFA without recording program execution histories. Finally, a benchmark test is provided to show the practicability of h-CFA.

Outline In the rest of the thesis, Section 2 describes the state-of-art techniques for CFA, which tend to improve the precision and reduce the overhead of k-CFA. It also discusses existing pushdown approaches. Section 3 presents the abstracting abstract machine (AAM) technique in detail, including abstract syntax, semantics of the abstract machine, and store widening. This section provides necessary preliminary knowledge to help readers understand our techniques because h-CFA is also developed in the AAM framework. Moreover, it summarizes advantages and disadvantages of AAM and reveals an essential drawback that introduces spurious return flows. Section 4 formalizes h-CFA and explains how it works with a simple example. Meanwhile, we compare our technique with other related works in several dimensions and give a performance evaluation via benchmark results. Section 5 details the design and implementation of JsCFA, which applies our techniques in

a JavaScript static analyzer. This implementation not only uses pushdown CFA, but also adopts techniques such as abstract garbage collection. Then we describe an approach for implementing h-CFA without recording the program execution history, which simplifies the intermediate representation and semantics of JsCFA. At the end of this section, a benchmark test of JsCFA is provided. Finally, we list several potential approaches for improving h-CFA and JsCFA in Section 6, and Section 7 concludes.

2 Related Work

In order to address the precision problem of original k-CFA, many techniques are introduced from different perspectives. Some algorithms tend to find better contexts for context-sensitive (polyvariant) analysis. For example, call-site sensitivity [26], argument sensitivity [1], object sensitivity [21, 27], and field sensitivity [17] contribute different benefits to precision or performance for different situations. Other techniques attempt to improve both monovariant and polyvariant in alternative ways. One of the most popular method of this group is pushdown-based CFA (a.k.a. context-free language reachability), which introduces pushdown system into abstract interpretation. The original k-CFA algorithm abstracts each program as a finite-state machine so that the abstract interpreter is guaranteed to terminate. The abstraction of k-CFA is only precise for programs with bounded call stacks. However, many language constructs (i.e. function invocation, exception handling, and first-class continuation, etc.) can generate *recursive* control flows. Since the abstraction of k-CFA is not precise for recursive structures, pushdown-based CFA is a better choice. The first contribution of this paper is a new method for implementing pushdown-based CFA, referred to as h-CFA, that provides perfect call/return matching. Before describing our technique, we discuss

the existing algorithms for the pushdown CFA and preliminary knowledge of h-CFA.

Pushdown CFA Algorithms The core idea of pushdown CFA is to mimic function call/return as an unbounded call stack for ordinary calls and summarizing call stacks to finite height for recursive calls because an unbounded call stack is not computable in static analysis. CFA2 [31] is the first algorithm that employs a pushdown system for CFA. CFA2 models the call stack as an implicit pushdown system, and summarizes the call stack with a tabulation algorithm for recursive functions. PDCFA (pushdown control flow analysis [6]) is another strategy that approximates unbounded stack model to be computable. PDCFA analyzes programs using a *Dyck state graph* [6], and tracks all of the reachable states in the graph. Meanwhile, edges of the Dyck state graph that connect program states are annotated with stack actions (push, pop, and no action). These stack actions explicitly represent a pushdown system and summarize recursive structures of the graph. Both CFA2 and PDCFA introduce extra semantics for target languages, which makes the abstract interpreter hard to implement. For this drawback, Van Horn and Might invented the Abstracting Abstract Machine (AAM) [23, 30] as a configurable framework for constructing abstract interpreters in the CESK abstract machine [9] style. Since AAM not only allocates values in the store

(as the original k-CFA does), but also represents control flow using store-allocated continuations. In AAM, each CESK state does not directly carry any continuation, but a continuation address that refers to a set of concrete continuations. Merging several continuations in one continuation address achieves the effect of approximating control flows. Meanwhile, AAM brings two benefits to control flow analysis. On the one hand, it makes the soundness of abstract interpreters easily prove because values and continuations are both in the store and the store size is fixed. Hence, the number of machine states that abstract interpreters generate is always finite. On the other hand, store-allocated continuations separate the context-sensitivity (polyvariance) strategy from the call/return matching technique. Additionally, implementing a static analyzer in AAM style is as easy as writing concrete interpreters. AAC (Abstracting Abstract Control [15]) and P4F (pushdown control flow analysis for free [11]) are both pushdown CFA techniques based on AAM, which convert the call/return matching problem to a continuation-address allocation problem. In other words, AAC and P4F just modify the continuation-allocation function of AAM to acquire call/return matching. However, AAC has high asymptotic upper bound $O(n^9)$ in monovariance (this complexity is claimed in [11] that cites to an unpublished article) and converges slowly in practice (see Section 4.3). P4F has better performance in polyvariant analysis but it has limited call/return matching strength and

is not useful for monovariant analysis.

3 Pushdown CFA in AAM

The AAM methodology considerably simplifies the implementation of abstract interpreters by introducing store-allocated values and continuations. At the same time, the soundness of AAM is relatively easy to prove. Therefore, we also use this theory as the foundation to develop h-CFA and a JavaScript analyzer, JsCFA. In this section we will review abstract interpretation in the setting of AAM to help readers to understand our techniques.

3.1 Abstracting Abstract Machine

In this section, we describe pushdown CFA algorithms using lambda calculus in the style of Administrative Normal Form (ANF) [10].

$$
\begin{aligned}
e \in Exp ::= & \ (let\ ((x\ (f\ æ)))\ e) \\
& |(let\ ((y\ æ))\ e) && \text{[expressions]} \\
& | \ æ \\
f, æ \in AExp ::= & \ x \mid lambda && \text{[atomic expressions]} \\
lambda \in Lambda ::= & \ (\lambda\ (x)\ e) && \text{[lambda abstractions]} \\
x, y \in Var & \text{ is a set of identifiers} && \text{[variables]}
\end{aligned}
$$

Above syntax definition just focuses on three kinds of expressions, calls, declarations, and returns. Other syntactic components, such as tail calls, conditional branching, do not complicate our semantics, so we leave them out. ANF sets a unique label for every intermediate expression, and these unique labels help we to implement and express h-CFA easily. Moreover, all of the intra-procedural control flows (the order of operations) are already compiled into `let` forms, which simplifies the semantics and accelerates our implementation.

Abstracting abstract machine (AAM) describes abstract interpreters that run and approximate a language on CESK abstract machine style. The abstract interpreter operates over CESK machine states $\tilde{\varsigma}$.

$$\tilde{\varsigma} \in \tilde{\sigma} \triangleq Exp \times \widetilde{Env} \times \widetilde{Store} \times \widetilde{KStore} \times \widetilde{KAddr} \qquad \text{[states]}$$

$$\tilde{\rho} \in \widetilde{Env} \triangleq Var \to \widetilde{Addr} \qquad \text{[environments]}$$

$$\tilde{\sigma} \in \widetilde{Store} \triangleq \widetilde{Addr} \to \widetilde{Value} \qquad \text{[stores]}$$

$$\tilde{v} \in \widetilde{Value} \triangleq \mathcal{P}(\widetilde{Closure}) \qquad \text{[abstract values]}$$

$$\widetilde{clo} \in \widetilde{Closure} \triangleq Lambda \times \widetilde{Env} \qquad \text{[closures]}$$

$$\tilde{\sigma}_k \in \widetilde{KStore} \triangleq \widetilde{KAddr} \to \widetilde{Kont} \qquad \text{[continuation stores]}$$

$$\widetilde{k} \in \widetilde{Kont} \triangleq \mathcal{P}(\widetilde{Frame}) \qquad \text{[abstract continuations]}$$

$$\widetilde{\phi} \in \widetilde{Frame} \triangleq Var \times Exp \times \widetilde{Env} \times \widetilde{KAddr} \qquad \text{[stack frames]}$$

$$\tilde{a} \in \widetilde{Addr} \text{ is a finite set} \qquad \text{[value addresses]}$$

$$\tilde{a}_k \in \widetilde{KAddr} \text{ is a finite set} \qquad \text{[continuation addresses]}$$

Environments ($\tilde{\rho}$) map variables to their binding address ($\tilde{a}$) in the scope. The original AAM paper uses just one store in a state to contain values and continuations both, but we prefer to separate it to value store ($\tilde{\sigma}$) and continuation store ($\tilde{\sigma}_k$) to clarify our algorithm. Value stores save every value ($\tilde{v}$) into a slot encoded by an address. Environments cooperate with values implementing the semantics of variable access. Closure ($\widetilde{clo}$) is the only value form of pure lambda calculus, which pairs a lambda abstraction with the environment form its defining point to implement static scoping. In our semantics, continuations ($\widetilde{k}$) just represent call stack frames because intra-procedural continuations are already converted to `let` sequences. Each frame ($\widetilde{\phi}$) includes: (1) a return point that is a variable to accept and bind the result of current application, (2) an expression the control flow returns to, (3) an environment to restore, (4) a continuation address that points to "next" continuations and builds up the linked stack structure. Therefore, each state carries a continuation address ($\tilde{a}_k$) to replace the continuation component of concrete CESK machine state, which the a continuation address point to the actual continuations (frames) inhabiting in the continuation store. This technique is referred to as *store-allocated continuation*.

Transition rules of CESK abstract machine operate over an input state

and generate a success state. However, an abstracting abstract machine has to output a set of states due to the non-deterministic semantics of abstract interpretation. Function application transition rule is defined below.

$$\overbrace{((let\ ((y\ (f\ æ))\ e)), \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_k, \tilde{a}_k)}^{\tilde{\varsigma}} \rightsquigarrow (e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}_k{}', \tilde{a}_k{}'), \text{ where}$$

$$((\lambda\ (x)\ e'), \tilde{\rho}_\lambda) \in \widetilde{eval}(f, \tilde{\rho}, \tilde{\sigma})$$

$$\tilde{\rho}' = \tilde{\rho}_\lambda[x \mapsto \tilde{a}]$$

$$\tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a} \mapsto \widetilde{eval}(æ, \tilde{\rho}, \tilde{\sigma})]$$

$$\tilde{a} = \widetilde{alloc}(x, \tilde{\varsigma})$$

$$\widetilde{\phi} = (y, e, \tilde{\rho}, \tilde{a}_k)$$

$$\tilde{\sigma}_k{}' = \tilde{\sigma}_k \sqcup [\tilde{a}_k{}' \mapsto \widetilde{\phi}]$$

$$\tilde{a}_k{}' = \widetilde{kalloc}(\tilde{\varsigma}, e', \tilde{\rho}', \tilde{\sigma}')$$

When we start to analyze call sites, $\widetilde{eval}$ firstly extracts closures from f that is always an atomic expression in ANF programs. The helper $\widetilde{eval}$ directly computes values of atomic expressions that is either a variable access point or lambda abstraction in pure lambda calculus.

$$\widetilde{eval} : AExp \times \widetilde{Env} \times \widetilde{Store} \to \widetilde{Value}$$

$$\widetilde{eval}(x, \tilde{\rho}, \tilde{\sigma}) \triangleq \tilde{\sigma}(\tilde{\rho}(x))$$

$$\widetilde{eval}(lambda, \tilde{\rho}, \tilde{\sigma}) \triangleq \{(lambda, \tilde{\rho})\}$$

Then argument is also evaluated and stored in a corresponding address. Environments restored from closures are extended by the formal parameter and actual parameter's address. In monovariant analysis, the address is only determined by expression's syntactic label, so the value addresses are always context-insensitive. Furthermore, we can use certain context information of program execution to separate values into different dimensions of addresses. For example, following definition of $\widetilde{alloc_1}$ encodes the closest call site into value addresses to implement 1-call-site sensitive analysis (1-CFA).

$$\widetilde{alloc} : Var \times \tilde{\Sigma} \to \widetilde{Addr}$$

$$\widetilde{alloc_0}(x, \tilde{\varsigma}) = x$$

$$\widetilde{alloc_1}(x, \tilde{\varsigma}) = (x, \tilde{\varsigma})$$

Following the semantics of call-by-value lambda calculus, after achieving values of callees and arguments, a call stack frame ($\widetilde{\phi}$) is pushed on the top ($\tilde{a}_k$) of stack (continuation store, $\tilde{\sigma}_k$). Meanwhile, a new stack top ($\tilde{a}_k{}'$) is allocated by $\widetilde{kalloc}$. The standard method of allocating continuation addresses in AAM is shown below, which represents the function entry point by its own syntactic label. Then, the entry point representation will be propagated to return states of the application.

$$\widetilde{kalloc} : \tilde{\Sigma} \times Exp \times \widetilde{Env} \times \widetilde{Store} \to \widetilde{KAddr}$$

$$\widetilde{kalloc}((e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_k, \tilde{a}_k), e', \tilde{\rho}', \tilde{\varsigma}') = e'$$

Additionally, AAM implements over-approximation of abstract interpretation by a join operation over value and continuation stores. The join is defined as follows.

$$\tilde{\sigma} \sqcup \tilde{\sigma}' = \lambda \tilde{a}.\ \tilde{\sigma}(\tilde{a}) \cup \tilde{\sigma}'(\tilde{a})$$

$$\tilde{\sigma_k} \sqcup \tilde{\sigma_k}' = \lambda \tilde{a}_k.\ \tilde{\sigma_k}(\tilde{a}_k) \cup \tilde{\sigma_k}'(\tilde{a}_k)$$

The declaration transition rule is very simple, which just spreads context information of abstract interpretation along `let` forms.

$$\overbrace{((let\ ((y\ æ)\ e)), \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_k, \tilde{a}_k)}^{\tilde{\varsigma}} \rightsquigarrow (e, \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}_k, \tilde{a}_k),\ \text{where}$$

$$\tilde{\rho}' = \tilde{\rho}[y \mapsto \tilde{a}]$$

$$\tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a} \mapsto \widetilde{eval}(æ, \tilde{\rho}, \tilde{\sigma})]$$

$$\tilde{a} = \widetilde{alloc}(y, \tilde{\varsigma})$$

The transition of return point is another crucial rule.

$$\overbrace{(æ, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_k, \tilde{a}_k)}^{\tilde{\varsigma}} \rightsquigarrow (e, \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}_k, \tilde{a}_k{}')$$

$$(x, e, \tilde{\rho}_k, \tilde{a}_k{}') \in \tilde{\sigma}_k(\tilde{a}_k)$$

$$\tilde{\rho}' = \tilde{\rho}_k[x \mapsto \tilde{a}]$$

$$\tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a} \mapsto \widetilde{eval}(æ, \tilde{\rho}, \tilde{\sigma})]$$

$$\tilde{a} = \widetilde{alloc}(x, \tilde{\varsigma})$$

The top frame is retrieved in continuation store with the current continuation address ($\tilde{a}_k$). Firstly, we acquire a return point variable x to refer the return value of current application, and extend environment $\tilde{\rho}_k$ with the return point to $\tilde{\rho}'$. Then, computation keeps going on expression e with environment $\tilde{\rho}'$, store $\tilde{\sigma}'$, stack $\tilde{\sigma}_k$, and "next" continuation address $\tilde{a}_k{}'$.

When we launch AAM on a program, $inject$ takes the program to create an initial state.

$$inject : Exp \to \tilde{\Sigma}$$

$$inject(e) = (e, \varnothing, \bot, \bot, \tilde{a}_{kinit})$$

The abstract interpreter starts to analyze a program from the initial state with empty environment, bottom stores, and a special continuation address. The address $\tilde{a}_{kinit}$ represents the bottom of call stack.

The transition relation we defined above is a monotonic function that is used by a worklist algorithm. Because AAM saves everything (values and continuations) in store, the number of $\tilde{\Sigma}$ is finite if store size is limited. Therefore, the worklist algorithm is always able to terminate even though the input program cannot terminate in concrete semantics.

Algorithm 1 Worklist Algorithm

```
initState ← inject(program)
todo ← initState :: Nil
seen ← initState :: Nil
while todo ≠ Nil do
    state ← head(todo)
    todo ← tail(todo)
    nexts ← transition_AAM(state)
    for n ∈ nexts do
        if n ∉ seen then
            seen ← n :: seen
            todo ← n :: todo
```

3.2 Store-widening

Theoretically, naive implementations of AAM take exponential time in the input program size. The time complexity of worklist algorithm is determined by the number of reachable machine states.

$$O(\overbrace{n}^{|Exp|} \times \overbrace{n}^{|\widetilde{Env}|} \times \overbrace{n^n}^{|\widetilde{Store}|} \times \overbrace{n^n}^{|\widetilde{KStore}|} \times \overbrace{n}^{|\widetilde{KAddr}|})$$

In monovariant analysis, values are always stored in locations that are only determined by syntactic positions of expressions. Meanwhile, environments map each variable to only one corresponding address. Because monovariant analysis does not carry any execution context during abstract interpretation, each expression always take only one environment. Likewise, continuation addresses are also allocated on syntactic positions. Thus, a tighter bound is:

$$O((\overbrace{n}^{|Exp|} + \overbrace{n}^{|\widetilde{Env}|} + \overbrace{n}^{|\widetilde{KAddr}|}) \times \overbrace{n^n}^{|\widetilde{Store}|} \times \overbrace{n^n}^{|\widetilde{KStore}|})$$

This complexity bound is still obviously exponential. Consequently, AAM implementations usually adopt widening on stores. Store widening uses a global store (single-threaded store, Shivers [26]) rather than per-state stores for values and continuations respectively. Global-store widening reduces the number of combinations of possible bindings in a store to $O(n^2)$, which is proved in [30, 11].

$$O((\overbrace{n}^{|Exp|}+\overbrace{n}^{|\widetilde{Env}|}+\overbrace{n}^{|\widetilde{KAddr}|}) \times (\overbrace{n^2}^{|\widetilde{Store}|} + \overbrace{n^2}^{|\widetilde{KStore}|}))$$

Eventually, the time complexity of AAM is $O(n^3)$ in monovariance.

3.3 A Defect of AAM

Although, AAM imports store-allocated values and store-allocated continuations that make call/return matching orthogonal from context-sensitivity, $\widetilde{kalloc}$ (continuation address allocating strategy) cannot depend upon context information to implement limited call/return matching (like k-CFA does). P4F attempts to narrow the gap between original k-CFA and AAM, so it defines the very simple $\widetilde{kalloc_{P4F}}$:

$$\widetilde{kalloc_{P4F}}((e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_k, \tilde{a}_k), e', \tilde{\rho}', \tilde{\sigma}') = (e', \tilde{\rho}')$$

Continuation addresses are represented by $(e', \tilde{\rho}')$ that the most obvious change is it packing callee function with "target environment" $(\tilde{\rho}')$ of the current application. Firstly, environments $(Var \rightarrow \widetilde{Addr})$ map variable names to value addresses in CESK abstract machines, and AAM encodes polyvariant strategy (e.g. call-site sensitive, object-sensitive, argument-sensitive, etc.) into value's addresses. Thus, P4F can be regarded as an adaptive pushdown control flow analysis algorithm that automatically achieves finite call/return matching support from values' polyvariant strategy. Secondly, P4F also reveals a significant fact why original AAM misses call/return flow matching. One of the most important contributions of AAM is that separates analysis context requirements from termination of abstract interpreters. All things (values and continuations) allocated in the store make termination of abstract interpreters easily reached because the fixed size of stores lead finite number of abstract machine states, so any implementation of $\widetilde{alloc}$ and $\widetilde{kalloc}$ is sound. However, the original $\widetilde{kalloc}$ function of AAM that mimics generating call stack frames of concrete interpreters does not acquire any benefit from values' polyvariance for getting more precise call/return flows. P4F fixed the problem by introducing polyvariance into continuation store, which brings context information in target environment to distinguish continuations under different contexts. Although P4F cannot infinitely match

call/return flows, it still discovers the essence of pushdown control flow analysis in AAM: *continuations also need to be polyvariant (context-sensitive) to achieve more precise static analysis results.*

4 Pushdown CFA based on Program Execution History

Inspired by P4F, we deem that pushdown analysis (polyvariant continuation store) is orthogonal from polyvariant store. In other words, control flow analysis can achieve call/return matching without polyvariant values. At the same time, we try to find the proper contexts for polyvariant continuations.

This section describes CESK^H machines that record "program execution history" into each abstract machine state. The program execution history records and summarizes execution path from the beginning of program to the current state. During the evaluation of function calls, the program execution history can be used to uniquely represent current call site in the continuation store.

4.1 Program Execution History

First, we modify the CESK machine defined in Section 3.1 to CESK^H machine. Data types and notations of CESK^H are defined below. We changed

parts of CESK definitions and indicate them with superscript H.

$$\widetilde{\varsigma^H} \in \widetilde{\Sigma^H} \triangleq Exp \times \widetilde{Env} \times \widetilde{Store} \times \widetilde{KStore^H} \times \widetilde{KAddr^H} \times \widetilde{History} \quad \text{[states]}$$

$$\tilde{\rho} \in \widetilde{Env} \triangleq Var \to \widetilde{Addr} \quad \text{[environments]}$$

$$\tilde{\sigma} \in \widetilde{Store} \triangleq \widetilde{Addr} \to \widetilde{Value} \quad \text{[stores]}$$

$$\tilde{v} \in \widetilde{Value} \triangleq \mathcal{P}(\widetilde{Closure}) \quad \text{[abstract values]}$$

$$\widetilde{clo} \in \widetilde{Closure} \triangleq Lambda \times \widetilde{Env} \quad \text{[closures]}$$

$$\widetilde{\sigma_k^H} \in \widetilde{KStore^H} \triangleq \widetilde{KAddr^H} \to \widetilde{Kont^H} \quad \text{[continuation stores]}$$

$$\widetilde{k^H} \in \widetilde{Kont^H} \triangleq \mathcal{P}(\widetilde{Frame^H}) \quad \text{[abstract continuations]}$$

$$\widetilde{\phi^H} \in \widetilde{Frame^H} \triangleq Var \times Exp \times \widetilde{Env} \times \widetilde{History} \times \widetilde{KAddr^H} \quad \text{[stack frames]}$$

$$\tilde{h} \in \widetilde{History} \triangleq Var \to \widetilde{Addr} \quad \text{[histories]}$$

$$\tilde{a} \in \widetilde{Addr} \; is \; a \; finite \; set \quad \text{[value addresses]}$$

$$\widetilde{a_k^H} \in \widetilde{KAddr} \; is \; a \; finite \; set \quad \text{[continuation addresses]}$$

In ANF programs, environment naturally maintains intra-procedural execution history because ANF explicitly extracts intra-procedural control flows in let-bindings and saves each intermediate result in a local variable. Consequently, the program execution histories can be implemented as propagating environments by $\widetilde{History}$ field of CESK^H machine states. We consider execution histories as call-strings with automatically determined length. For

non-recursive calls, execution history always provides enough precise context information, no matter how deep the call sequences. On the other hand, program execution histories can automatically stop growing for recursive calls, and the worklist algorithm will be responsible for finding the fixed-point of recursive computation.

The following definitions describe the abstract semantics of CESK^H machine.

Calls

$$\overbrace{((let\ ((y\ (f\ æ))\ e)), \tilde{\rho}, \tilde{\sigma}, \widetilde{\sigma_k^H}, \widetilde{a_k^H}, \tilde{h})}^{\widetilde{\varsigma^H}} \rightsquigarrow (e', \tilde{\rho}', \tilde{\sigma}', \widetilde{\sigma_k^H}', \widetilde{a_k^H}', \tilde{h}'), where$$

$$((\lambda\ (x)\ e'), \tilde{\rho}_\lambda) \in \widetilde{eval}(f, \tilde{\rho}, \tilde{\sigma})$$

$$\tilde{\rho}' = \tilde{\rho}_\lambda[x \mapsto \tilde{a}]$$

$$\tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a} \mapsto \widetilde{eval}(æ, \tilde{\rho}, \tilde{\sigma})]$$

$$\tilde{a} = \widetilde{alloc}(x, \widetilde{\varsigma^H})$$

$$\widetilde{\phi^H} = (y, e, \tilde{\rho}, \tilde{h}, \widetilde{a_k^H})$$

$$\widetilde{\sigma_k^H}' = \widetilde{\sigma_k^H} \sqcup [\widetilde{a_k^H}' \mapsto \widetilde{\phi^H}]$$

$$\widetilde{a_k^H}' = \widetilde{kalloc_h}(\tilde{\varsigma}, e', \tilde{\rho}', \tilde{\sigma}')$$

$$\tilde{h}' = \tilde{h}[x \mapsto \tilde{a}]$$

The semantics of function calls propagate execution history by adding current "intermediate variable" to $\tilde{h}$, but execution history extension is different from environment extension, which recovers the base environment from function definition point.

Declarations

$$\overbrace{((let\ ((y\ æ)\ e)), \tilde{\rho}, \tilde{\sigma}, \widetilde{\sigma_k^H}, \widetilde{a_k^H}, \tilde{h})}^{\widetilde{\varsigma^H}} \rightsquigarrow (e, \tilde{\rho}', \tilde{\sigma}', \widetilde{\sigma_k^H}, \widetilde{a_k^H}, \tilde{h}'), \text{ where}$$

$$\tilde{\rho}' = \tilde{\rho}[y \mapsto \tilde{a}]$$

$$\tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a} \mapsto \widetilde{eval}(æ, \tilde{\rho}, \tilde{\sigma})]$$

$$\tilde{a} = \widetilde{alloc}(y, \tilde{\varsigma})$$

$$\tilde{h}' = \tilde{h}[y \mapsto \tilde{a}]$$

Declarations are `let` forms that just binds atomic expressions to variables. Its semantics is very straightforward that propagates environments ($\tilde{\rho}$) and histories ($\tilde{h}$) through the linear control flow.

Returns

$$\overbrace{(æ, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma_k}, \widetilde{a_k^H}, \tilde{h})}^{\widetilde{\varsigma^H}} \rightsquigarrow (e, \tilde{\rho}', \tilde{\sigma}', \widetilde{\sigma_k^H}, \widetilde{a_k^{H\prime}}, \tilde{h}')$$

$$(x, e, \tilde{\rho_k}, \tilde{h_k}, \widetilde{a_k^{H\prime}}) \in \widetilde{\sigma_k^H}(\widetilde{a_k^H})$$

$$\tilde{\rho}' = \tilde{\rho_k}[x \mapsto \tilde{a}]$$

$$\tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a} \mapsto \widetilde{eval}(æ, \tilde{\rho}, \tilde{\sigma})]$$

$$\tilde{a} = \widetilde{alloc}(x, \widetilde{\varsigma^H})$$

$$\tilde{h}' = \tilde{h_k}[x \mapsto \tilde{a}]$$

In *return*'s definition, an abstract interpreter restores up-level's history from the frames referred by the current continuation address, which is similar to restoring the environment. The $\widetilde{kalloc_h}$ takes the execution history to compute the unique continuation address for corresponding call site.

$$\widetilde{kalloc_h}((e, \tilde{\rho}, \tilde{\sigma}, \widetilde{\sigma_k^H}, \widetilde{a_k^H}, \tilde{h}), e', \tilde{\rho}', \tilde{\sigma}') = (e, e', \tilde{h})$$

$\widetilde{KAddr^H}$ in CESK^H machines is encoded by: (1) the call site e, (2) the callee function e', (3) and current execution history $\tilde{h}$. 0-CFA-like analysis in AAM just adopts e' to refer abstract continuations, so all the potential call sites that may invoke e' will merge with each others. Therefore, the $\widetilde{KAddr^H}$ definition distinguishes as many as possible call sites of e' via the very last call site e and the rests encoded by $\tilde{h}$.

4.2 Polyvariant Continuation

In this section, we use a simple example in Figure 4.1 to explain the analysis process of h-CFA.

For simplicity, the execution histories are represented as variable sequences, and the called function (the second part of $\widetilde{KAddr^H}$) is replaced

```
(letrec ((fib (lambda (n)
                (let ((res1 (< n 3)))
                  (if res1
                      1
                      (let* ((res2 (- n 1))
                             (res3 (fib res2))
                             (res4 (- n 2))
                             (res5 (fib res4))
                             (res6 (+ res3 res5)))
                        res6))))))
  (let ((a (fib 10))
        (b (fib 20)))
    (fib 30)))
```

Figure 4.1: An example written in ANF style defines a recursive function and calls it multiple times. For convenient demonstrating, we use complete Scheme language with numbers and booleans instead of pure lambda calculus.

by its function name wearing a hat. This simplification improves readability without modifying the abstract semantics of CESK^H machine.

Through steps of the abstract interpretation, the first call site $(a\ (fib\ 10))$ carries the history $\{fib\}$, which means that, at this program point, we have only finished computing the declaration of the function fib—. Thus, the continuation (call stack frame) of the call site is allocated at $((fib\ 10), \widehat{fib}, \{fib\})$, and the stack frame looks like $(a, (let\ (b\ (fib\ 20))\ \ldots), \widetilde{env_1}, \{fib\}, \widetilde{a^H_{k\ init}})$, which is the only element in the continuation store so far.

The stack frame expresses that after completing this invocation, (1) the return value will be stored in variable a, (2) the computation will shift to $(let\ (b\ (fib\ 20))\ \ldots)$ with environment $\widetilde{env_1}$, (3) and the continuation address $\widetilde{a^H_{k\ init}}$, a fake one for the top-level continuation, will be recovered.

After diving into the callee function, the second call appears at $(res3\ (fib\ res2))$. At this point, the execution history $\{fib, res1, res2\}$ is different from the history of last call site, so the continuation store contains two abstract continuations with distinct addresses.

$$\widetilde{a_k^H}_1 = ((fib\ 10), \widehat{fib}, \{fib\})$$

$$\widetilde{a_k^H}_2 = ((fib\ res2), \widehat{fib}, \{fib, res1, res2\})$$

$$\widetilde{\sigma_k^H} = \{\widetilde{a_k^H}_1 \mapsto \{(a, (let\ (b\ (fib\ 20))\ \dots), \widetilde{env}_1, \{fib\}, \widetilde{a_{k\ init}^H})\}$$

$$\widetilde{a_k^H}_2 \mapsto \{(res3, (let\ (res4\ (-\ n\ 2))\dots), \widetilde{env}_2, \{fib, res1, res2\}, \widetilde{a_k^H}_1)\}\}$$

As above illustration shows, the continuation store is a stack with linked-list structure. Each frame has a $\widetilde{a_k^H}$ that points to the next frame in the stack. This stack-like structure perfectly mimics call stacks of concrete interpreters. Certainly, the call site $(fib\ res4)$ will also gets its own execution history after computation of $(fib\ res2)$ completes (i.e. after reaching its fixed-point).

$$\widetilde{a_k^H}_3 = ((fib\ res4), \widehat{fib}, \{fib, res1, res2, res3, res4\})$$

However, $(res3\ (fib\ res2))$ is a recursive call site. So the execution history at this point will not add new element to distinguish $(res3\ (fib\ res2))$ from its variations at different recursive levels. Thus, control flows from multiple recursive levels of a call site are merged into one continuation address. Eventually, there are three frames merged into $\widetilde{a_k^H}_2$, but this merging does

not lead "static" call/return mismatch. All of the frames merged into $\widetilde{a_k^H}_2$ can bring control flow back to set `res3`.

$$\widetilde{a_k^H}_2 \mapsto \{(res3, (let\ (res4\ (-\ n\ 2))\ldots), \widetilde{env}_2, \{fib, res1, res2\}, \widetilde{a_k^H}_1),$$
$$(res3, (let\ (res4\ (-\ n\ 2))\ldots), \widetilde{env}_3, \{fib, res1, res2\}, \widetilde{a_k^H}_2)\}$$

The merging expresses a fact that the invocation of fib— at point $(fib\, res2)$ may be made by $(a\ (fib\ 10))$ or $(res3\ (fib\, res2))$. Moreover, the second frame in the above illustration has the "next" pointer $\widetilde{a_k^H}_2$ that refers to itself. This cycle makes the continuation store no longer stack-like, but a graph.

After computing $(a\ (fib\ 10))$, the function fib is called again by $(b\ (fib\ 20))$. At this point, $\widetilde{kalloc}_h$ generates a new continuation address.

$$\widetilde{a_k^H}_4 = ((fib\ 20), \widehat{fib}, \{fib, a\})$$

The execution history of this point becomes $\{fib, a\}$ that summarizes the execution path of computing $(a\ (fib\ 10))$ to a. In other words, the program execution history just cares about which portions of the program we have done, but it ignores how we got them. This summarization limits the length of the execution histories under $O(n)$ (the size of input program) in the worst case.

Then the abstract interpreter restarts to execute the function and encounters call site $(res3\ (fib\ res2))$ again. At this time, the continuation address

Algorithm	Match Strength	Mono -variance	Poly -variance	Implementation	Complexity on Monovariance
CFA2	Infinite	✓		Difficult	Exponential
P4F	Limited		✓	Easy	$O(n^3)$
PDCFA	Infinite	✓	✓	Difficult	$O(n^6)$
AAC	Infinite	✓	✓	Easy	$O(n^9)$
h-CFA	Infinite	✓	✓	Easy	Exponential

Table 4.1: Comparison of pushdown CFA algorithms in terms of analysis precision, time complexity, and ease of implementation.

$(\widetilde{a_k^H}_5)$ allocated for call site $(res3\ (fib\ res2))$ differs from last time, which makes sure that there are two distinct "call stacks". Consequently, function fib called from $(b\ (fib\ 20))$ will never return to $(a\ (fib\ 10))$ and vice versa.

$$\widetilde{a_k^H}_5 = ((fib\ res2), \widehat{fib}, \{fib, a, res1, res2\})$$

4.3 Complexity and Precision of *h*-CFA

We have applied store-widening to h-CFA for both the value store and the continuation store. However, we have not obtained a polynomial time complexity for h-CFA. A comparison of the related pushdown CFA algorithms with h-CFA is shown in Figure 4.1. According to this table, our technique seems be worse than AAC in asymptotic upper bounds. However, in practice the performance of h-CFA is better than AAC for most cases. We have run both the h-CFA implementation and AAC on test cases from *Larceny*

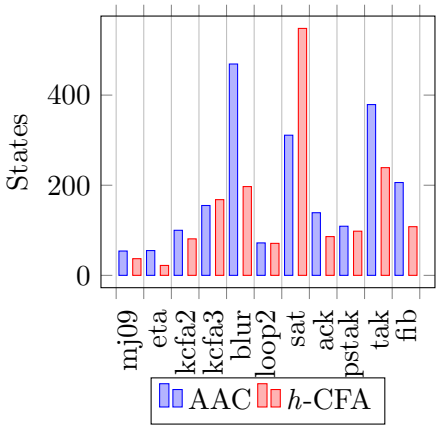


Figure 4.2: Monovariant analysis performance comparison

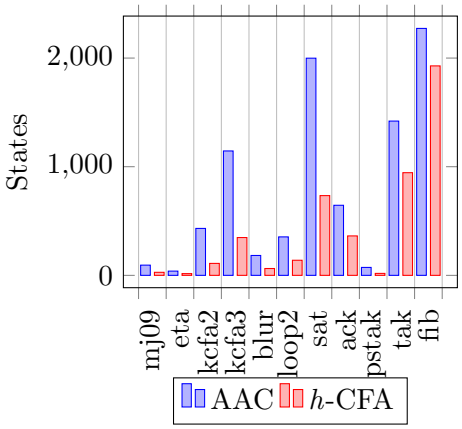


Figure 4.3: 1-call-site sensitive analysis performance comparison

R6RS benchmark suite and some other examples. Figure 4.2 compares the number of states that h-CFA and AAC explored in monovariant analysis and Figure 4.3 shows the test with 1-call-site sensitivity.

Moreover, AAC has a major drawback, which is its space complexity in real world applications. The essential strategy of AAC is defined below [11].

$$\widetilde{kalloc}_{AAC}((e, \tilde{\rho}, \tilde{\sigma}, \tilde{a}_k), e', \tilde{\rho}', \tilde{\sigma}') = (e', \tilde{\rho}', e, \tilde{\rho}, \tilde{\sigma})$$

The function $\widetilde{kalloc}_{AAC}$ encodes an unique continuation address for the call site with a target closure $(e', \tilde{\rho}')$, a source closure $(e, \tilde{\rho})$, and store $\tilde{\sigma}$. This strategy would work well if we use purely functional data structures to implement stores. However, in realistic analyzers, functional data structures usually incurs considerable performance cost, and imperative stores will significantly increase the space complexity of AAC.

Furthermore, we compared the call/return matching precision of several CFA algorithms in AAM, including k-CFA-like, P4F, AAC, and h-CFA. Figure 4.4 shows percentages of mismatching returns in monovariant analysis, where mismatching return states retrieve several different return points from their continuation address. These different return points immediately produce spurious return flows, so the statistic data can represent the precision of call/return matching. This figure indicate that h-CFA does not have any mismatch return flows on all the programs (mismatching return percentages are 0%), and P4F does not benefit to monovariant analysis due to its precision always same with 0-CFA-like analysis. Figure 4.5 provides the comparison on 1-call-site sensitive analysis, and h-CFA is also the most accurate solution.

To visually illustrate the call/return matching strength of P4F, PDCFA, PDCFA with abstract Garbage Collection (GC), and h-CFA, we have implemented them for Scheme language. We ran the four algorithms on a small program that is similar to the program showed in Figure 4.1. The resulting state-transition graphs are shown in Figure 4.6. As the h-CFA graph shows, there are three similar subgraphs in the state transition process, which obviously illustrates no call/return flow merged in h-CFA due to three subgraphs connected by single transition edges. PDCFA graph also illustrates the similar pattern. To compare, P4F that just supplies limited call/return matching merges too many control flows in this recursive program.

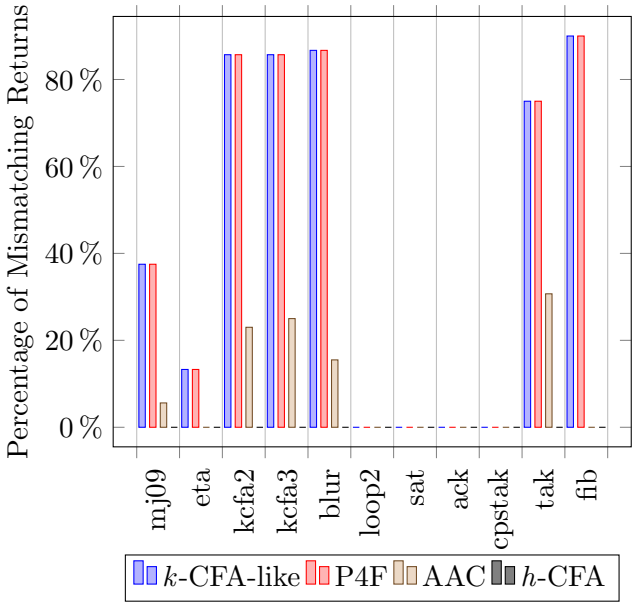


Figure 4.4: Monovariant analysis precision comparison

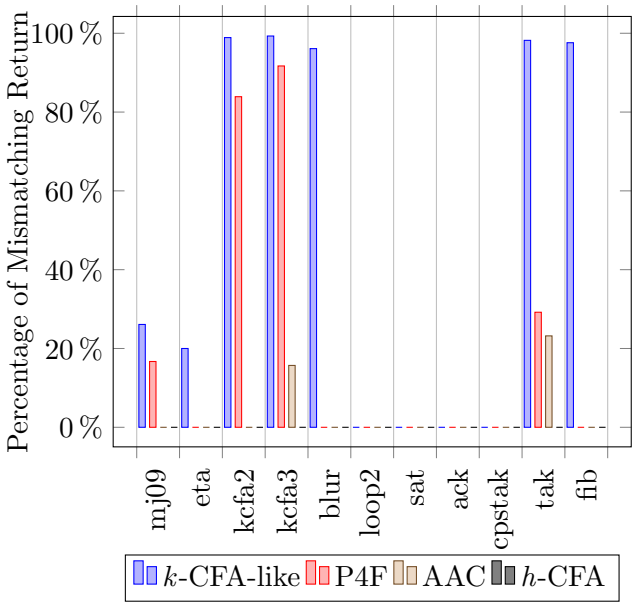
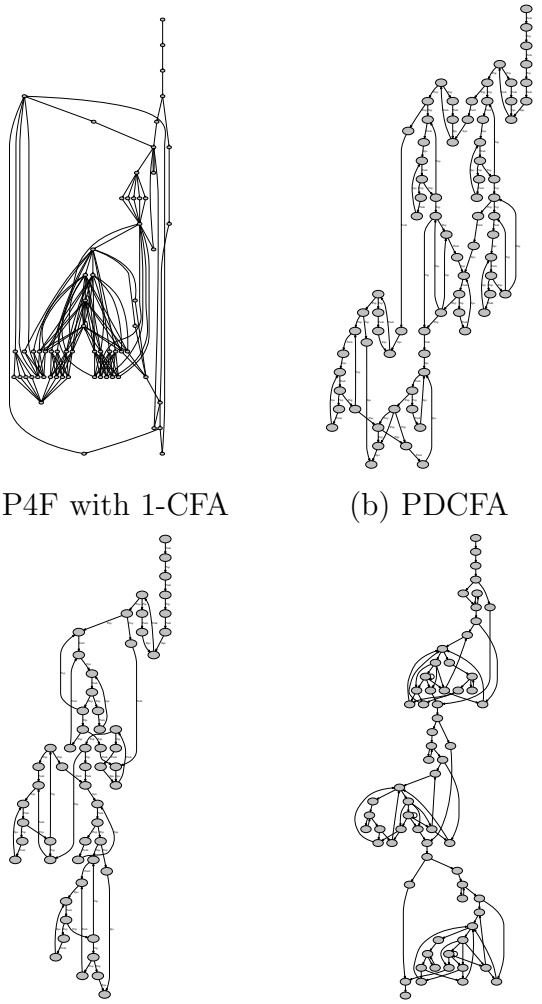


Figure 4.5: 1-call-site sensitive analysis percision comparison



(a) P4F with 1-CFA

(b) PDCFA

(c) PDCFA with Abstract Garbage Collection

(d) h-CFA

Figure 4.6: State transition graphs of: (1) P4F (pushdown CFA for free) with 1-CFA; (2) PDCFA (pushdown CFA); (3) PDCFA with abstract GC; (4) h-CFA. (2–4) are run with 0-CFA.

5 Design of JsCFA

JavaScript has become a ubiquitous computing environment in browsers, severs, desktops, even mobile devices. Developers are attracted by its effective and convenient features, such as duck typing, first-class functions, and runtime changeable objects, etc. However, these flexible features also makes large JavaScript programs to be increasingly unreliable. As one of the software engineering tools, static analysis has become an effective choice to help detect deep semantic information and defects, but the static analysis algorithms in JavaScript is still not comparable to those of the static languages such as Java.

One of the most difficult challenges is that JavaScript is a higher-order programming language that treats functions as first-class values. First-class functions can be referred by variables, passed in function arguments, and emitted as return values of other functions. In static analysis of higher-order programming languages, control flow analysis plays a significant role because we often cannot determine which function is called at a specific call site. At the same time, JavaScript heavily relies on first-class functions to implement certain high-level semantics, such as methods, block scoping, and module import/export. Consequently, we developed JsCFA, an abstract interpreter

for a subset of JavaScript (ECMAScript 3) based on h-CFA to perform a more precise control flow analysis. Although JsCFA computes monovariant and context-insensitive results by default, its AAM allows users to obtain context-sensitivity easily. To demonstrate this, we also implemented context-sensitive analysis for selected situations.

This section describes the essential pieces of JsCFA design, including abstract syntax, abstract semantic rules, context-sensitivity, analysis improvement, and usage of h-CFA.

5.1 Syntax Interface

In JsCFA, we convert the standard semantics of JavaScript to small-step abstract machine with an unbounded stack in CESK style. The CESK machine operates directly over the abstract syntax tree (AST) yielded from the parser. The AST interface is shown in Figure 5.1, Figure 5.2, and Figure 5.3 in Scala code. Most of the data structure is separated to two categories that inherit from abstract class `Statement` and `Expression` respectively. JsCFA distinguishes *left values* from *right values* at the syntactic level to simplify the implementation of CESK machine so that the expressions that are subclasses of `LValue` are eventually reduced to left values, The trait `AbstractSyntaxTree` defines the field `id` to hold an unique label for each

AST node. It also implements the method `generateFrom` that spreads the static information from the AST nodes to local continuations and values. The top-level program is a sequence of statements wrapped in the statement `Script`.

5.2 Transition Rules

The core data structure of the AAM of JsCFA is the class `State` (denoted by $\tilde{\varsigma}$ in formal definitions), which has six components `e` (control string), `env` (environment), `localStack` (intra-procedural continuation stack), `a` (inter-procedural continuation address, or called stack frame pointer), `store` (value store), and `stack` (continuation store). Among them, "store" and "stack" are packed into the `memory` object that encapsulates certain methods to manipulate the value and continuation store.

```
case class State(e: AbstractSyntaxTree,
                 env: Environment,
                 localStack: LocalStack,
                 a: StackAddress,
                 memory: Memory)

case class Memory(store: mutable.Map[JSReference, Set[JSValue]],
                  stack: mutable.Map[StackAddress, Set[Frame]]) {
  ...
```

```
sealed abstract class Statement extends AbstractSyntaxTree

case class Script(stmts: List[Statement]) extends  Statement
case class BlockStmt(stmts: List[Statement]) extends Statement
case class VarDeclListStmt(decls: List[Statement]) extends
    ↪ Statement
case class EmptyStmt() extends Statement
case class ExprStmt(expr: Expression) extends Statement
case class VarDeclStmt(name: IntroduceVar, expr: Expression)
    ↪ extends Statement
case class FunctionDecl(name: IntroduceVar, fun: Expression)
    ↪ extends Statement
case class ReturnStmt(expr: Expression) extends Statement
case class IfStmt(cond: Expression, thenPart: Statement,
    ↪ elsePart: Statement) extends Statement
case class SwitchStmt(cond: Expression, cases: List[CaseStmt],
    ↪  defaultCase: Option[CaseStmt]) extends Statement
case class CaseStmt(expr: Expression, body: Statement) extends
    ↪  Statement
case class ContinueStmt(continueLabel: String) extends
    ↪ Statement
case class DoWhileStmt(cond: Expression, body: Statement)
    ↪ extends Statement
case class WhileStmt(cond: Expression, body: Statement)
    ↪ extends Statement
case class ForStmt(init: ForInit, cond: Option[Expression],
    ↪ increment: Option[Expression], body: Statement) extends
    ↪ Statement
case class ForInStmt(init: ForInInit, expr: Expression, body:
    ↪ Statement) extends Statement
```

Figure 5.1: Abstract syntax tree data types of statements

```
sealed abstract class Expression extends AbstractSyntaxTree

case class EmptyExpr() extends Expression
case class FunctionExpr(name: Option[IntroduceVar], ps: List[
    ↪ IntroduceVar], body: Statement) extends Expression with
    ↪ ObjectGeneratePoint
case class VarRef(name: String) extends Expression with
    ↪ VariableAccess
case class ThisRef() extends Expression
case class DotRef(obj: Expression, prop: String) extends
    ↪ Expression
case class BracketRef(obj: Expression, prop: Expression)
    ↪ extends Expression
case class MethodCall(receiver: Expression, method: Expression
    ↪ , args: List[Expression]) extends Expression
case class FuncCall(func: Expression, args: List[Expression])
    ↪ extends Expression
case class NewCall(constructor: Expression, args: List[
    ↪ Expression]) extends Expression with ObjectGeneratePoint
case class AssignExpr(op: AssignOp, lv: LValue, expr:
    ↪ Expression) extends Expression
case class NullLit() extends Expression
case class BoolLit(value: Boolean) extends Expression
case class NumberLit(value: Double) extends Expression
case class StringLit(value: String) extends Expression
case class RegExp(regexp: String, global: Boolean,
    ↪ case_insensitive: Boolean) extends Expression with
    ↪ ObjectGeneratePoint
case class ObjectLit(obj: List[ObjectPair]) extends Expression
    ↪  with ObjectGeneratePoint
case class ArrayLit(vs: List[Expression]) extends Expression
    ↪ with ObjectGeneratePoint
case class UnaryAssignExpr(op: UnaryAssignOp, lv: LValue)
    ↪ extends Expression
case class PrefixExpr(op: PrefixOp, expr: Expression) extends
    ↪ Expression
case class InfixExpr(op: InfixOp, expr1: Expression, expr2:
    ↪ Expression) extends Expression
case class CondExpr(cond: Expression, thenPart: Expression,
    ↪ elsePart: Expression) extends Expression
case class ListExpr(exprs: List[Expression]) extends
    ↪ Expression
```

Figure 5.2: Abstract syntax tree data types of expressions

```
sealed abstract class LValue extends AbstractSyntaxTree

case class LVarRef(name: String) extends LValue with
    ↪ VariableAccess
case class LDot(obj: Expression, field: String) extends LValue
case class LBracket(obj: Expression, prop: Expression) extends
    ↪ LValue
```

Figure 5.3: Abstract syntax tree data types of lvalue expressions

```
}
```

The above definition shows two differences from the original AAM and h-CFA. The class `State` does not contain "History" field because we implement h-CFA indirectly for JsCFA, and this modification will be discussed in Section 5.7.2. Meanwhile, there is an extra field `localStack` that plays the role of intra-procedural continuation stack, but does not exist in AAM and h-CFA. In h-CFA, before the actual analysis, ANF transformation already flattens all the intra-procedural control flows to the let-bindings, so it just requires inter-procedural continuations in stores. Besides, the original AAM saves all of the continuations (inter and intra-procedural) into continuation stores, but actually only inter-procedural control flows have to be retrieved non-deterministically while the intra-procedural continuations are always deterministic. Therefore, we separate the inter-procedural continuations from the intra-procedural ones, which clarifies the semantics and improves the performance.

JsCFA distinguishes three types of transition states: evaluation, continuation, and application.

Evaluation transition accepts a state and matches its control string component to generate successors. If the control string is a value (instance of class `JSValue`), analysis would be dispatched to a continuation transition that depends upon the following control flow step retrieved from the top of local stack. Then the continuation transition determines how to use the value. If the control string is an expression/statement with no reducible sub-components, the abstract machine applies one of the *application transitions*. If the control string is an expression/statement with reducible sub-components, the abstract machine picks a reducible sub-component according to the evaluation order of JavaScript and generates a new continuation for the rest of the expression/statement. The new continuation is pushed onto the local stack and the abstract machine proceeds to evaluate the selected sub-component. Finally, there is a special case in dispatching to the continuation transitions. If the local stack is empty (no valid `cont` in Figure 5.2), then there is no "next step" in current execution context and the function does not have a return statement along the current execution path. In this case, we return `undefined` value to the *return* point restored from the stack frames.

```
def transitEvaluation(state: State): Set[State] = state match {
  case completeState if isComplete(completeState) =>
      transitApplication(state)
      //dispatch to application transition
  case State(v, env, localStack, a, memory) if isJSValue(v) =>
      if (localStack.nonEmpty) {
        val cont = topOfLocalStack(localStack)
        val newStack = popLocalStack(localStack)
        transitContinuation(cont, v, env, newStack, a, memory)
        //dispatch to continuation transition
      } else {
        ... \\return
      }

  //statements
  case State(Script(Nil), env, localStack, a, memory) =>
      Set(State(Halt, env, localStack, a, memory))
  case State(Script(stmt :: ss), env, localStack, a, memory) =>
      val k = KScript(ss)
      k.generateFrom(state.e)
      val newStack = pushLocalStack(localStack, k)
      Set(State(stmt, env, newStack, a, memory))

  case State(ReturnStmt(e), env, localStack, a, memory) =>
      val k = KReturn()
      k.generateFrom(state.e)
      val newStack = pushLocalStack(localStack, k)
      Set(State(e, env, newStack, a, memory))

  case State(IfStmt(cond, t, e), env, localStack, a, memory) =>
      val k = KIfCond(t, e)
      k.generateFrom(state.e)
      val newStack = pushLocalStack(localStack, k)
      Set(State(cond, env, newStack, a, memory))
  ...
}
```

Figure 5.4: Parts of evaluation transition rules for dispatching and statements

```
//expressions
case State(FuncCall(func, args), env, localStack, a, memory) =>
    val k = KFuncCallF(args)
    k.generateFrom(state.e)
    val newStack = pushLocalStack(localStack, k)
    Set(State(func, env, newStack, a, memory))

case State(AssignExpr(op, lv, expr), env, localStack, a, memory) =>
    val k = KAssignR(op, lv)
    k.generateFrom(state.e)
    val newStack = pushLocalStack(localStack, k)
    Set(State(expr, env, newStack, a, memory))

case State(InfixExpr(op, e1, e2), env, localStack, a, memory) =>
    val k = KInfixL(op, e2)
    k.generateFrom(state.e)
    val newStack = pushLocalStack(localStack, k)
    Set(State(e1, env, newStack, a, memory))
...
```

Figure 5.5: Parts of evaluation transition rules for expressions

Continuation transition works on the six components of the state object and an extra *next continuation* (referred to as `cont` in Figure 5.6 and Figure 5.7). Control strings in these transition states are always values, so the abstract machine dispatches transitions via matching `cont` and plug the value into the next continuation. If the next continuation is an expression/statement with no reducible sub-component, the next machine states will move to application transitions. In this case, the new continuation is placed on the control-string position of next state. If the next continuation contains reducible sub-components, then the abstract machine takes the same actions as it did for the *evaluation transitions*.

```
def transitContinuation(cont: Continuation,
                        value: JSValue,
                        env: Environment,
                        localStack: LocalStack,
                        a: StackAddress,
                        memory: Memory): Set[State] = cont match {
  case KScript(Nil) =>
      Set(State(Halt, env, localStack, a, memory))
  case KScript(s :: ss) =>
      val k = KScript(ss)
      k.generateFrom(cont)
      val newStack = pushLocalStack(localStack, k)
      Set(State(s, env, newStack, a, memory))

  case KReturn() =>
      val k = KReturnComplete(value)
      k.generateFrom(cont)
      Set(State(k, env, localStack, a, memory))

  case KIfCond(t, e) =>
      val k = KIfComplete(value, t, e)
      k.generateFrom(cont)
      Set(State(k, env, localStack, a, memory))
  ...
```

Figure 5.6: Parts of continuation transition rules for statements

```
  case KFuncCallF(Nil) =>
      val k = KFuncCallA(value, Nil, Nil)
      k.generateFrom(cont)
      val newStack = pushLocalStack(localStack, k)
      Set(State(cachedUndefined, env, newStack, a, memory))

  case KFuncCallF(arg :: args) =>
      val k = KFuncCallA(value, Nil, args)
      k.generateFrom(cont)
      val newStack = pushLocalStack(localStack, k)
      Set(State(arg, env, newStack, a, memory))

  case KFuncCallA(func, before, Nil) =>
      val k = KFuncCallComplete(func, before ++ List(value))
      k.generateFrom(cont)
      Set(State(k, env, localStack, a, memory))

  case KFuncCallA(func, before, arg :: args) =>
      val k = KFuncCallA(func, before ++ List(value), args)
      k.generateFrom(cont)
      val newStack = pushLocalStack(localStack, k)
      Set(State(arg, env, newStack, a, memory))

  case KAssignR(op, lv) =>
      val k = KAssignL(op, value)
      k.generateFrom(cont)
      val newStack = pushLocalStack(localStack, k)
      Set(State(lv, env, newStack, a, memory))

  case KAssignL(op, rv) =>
      val k = KAssignExprComplete(op, value, rv)
      k.generateFrom(cont)
      Set(State(k, env, localStack, a, memory))

  case KInfixL(op, e2) =>
      val k = KInfixR(op, value)
      k.generateFrom(cont)
      val newStack = pushLocalStack(localStack, k)
      Set(State(e2, env, newStack, a, memory))

  case KInfixR(op, e1) =>
      val k = KInfixExprComplete(op, e1, value)
      k.generateFrom(cont)
      Set(State(k, env, localStack, a, memory))
  ...
}
```

Figure 5.7: Parts of continuation transition rules for expressions

Application transition just accepts its state to generate the subsequent states. Function `isComplete` exams whether the passed-in state is complete by matching its control string. A complete control string represents an expressions/statement with no reducible sub-components. It is either an AST node from the parser or a continuation object generated by the continuation transitions. The application transitions described in Figure 5.8 and Figure 5.9.

Once the abstract interpreter of JsCFA launches, the parser reads input program and converts it to an AST wrapped in a `Script` object at the top level. Then an `inject` function takes the AST to generate the initial machine state that contains all JavaScript built-in variables, objects, and functions into $\widetilde{initEnv}$ and $\widetilde{initMemory}$. Lastly, the worklist algorithm starts an evaluation transition from the initial state.

$$inject_{JsCFA} : AbstractSyntaxTree \rightarrow State$$

$$inject_{JsCFA}(script) = State(script, \widetilde{initEnv}, \varnothing, \widetilde{a_{kinit}}, \widetilde{initMemory})$$

5.3 Store and Stack

The class `Memory` packs the value store and the continuation store in one object and provides four main methods for interacting with the abstract

```
def transitApplication(state: State): Set[State] = state match {
  case State(KReturnComplete(v), env, localStack, a, memory) =>
      val newMemory = memory.copy(state)
      for {
        Frame(returnPoint, oldStack, savedEnv, newGlobalAddress)
            <- newMemory.getFrames(a) //get the stack frame
      } yield {
        if(oldStack.isEmpty ||
          !oldStack.head.isInstanceOf[KUseValue]) {
          //make sure the current invocation is not
          //a "new call"
          for(vs <- newMemory.getValues(v)) {
            newMemory.putValue(returnPoint, vs)
          }
        }
        State(returnPoint, savedEnv,
              oldStack, newGlobalAddress, newMemory)
      }

  case State(KIfComplete(cond, t, e), env, localStack, a, memory) =>
      for {
        obj <- memory.getValues(cond)
        boolValue = ToBoolean(obj)
        res <- boolValue match {
          case JSBoolean(ConstantBoolean(true)) =>
              Set(State(t, env, localStack, a, memory))
          case JSBoolean(ConstantBoolean(false)) =>
              Set(State(e, env, localStack, a, memory))
          case JSBoolean(VariableBoolean) =>
              Set(State(t, env, localStack, a, memory),
              State(e, env, localStack, a, memory))
        }
      } yield res
  ...
```

Figure 5.8: Parts of application transition rules for statements

```
  case State(VarRef(x), env, localStack, a, memory) =>
      val newMemory = memory.copy(state)
      val xRef = lookup(env, x)
      for(vs <- newMemory.getValues(xRef)){
        newMemory.putValue(JSReference(state.e.id), vs)
      }
      Set(State(JSReference(state.e.id), env, localStack, a, newMemory))

  case State(LVarRef(x), env, localStack, a, memory) =>
      val xRef = lookup(env, x)
      // lvalues are addresses
      Set(State(xRef, env, localStack, a, memory))

  case State(f@FunctionExpr(name, ps, body), env, localStack, a, memory) =>
      val newMemory = memory.copy(state)
      val functionObject = createFunctionObject(f, env, newMemory)
      val value = newMemory.save(functionObject)
      Set(State(value, env, localStack, a, newMemory))

  case State(NumberLit(num), env, localStack, a, memory) =>
      val newMemory = memory.copy(state)
      val number = JSNumber(ConstantNumber(num))
      number.generateFrom(state.e)
      val value = newMemory.save(number)
      Set(State(value, env, localStack, a, newMemory))

  case State(KAssignExprComplete(op, lv, rv), env, localStack, a, memory) =>
      val newMemory = memory.copy(state)
      for(value <- newMemory.getValues(rv)) {
          newMemory.putValue(lv.asInstanceOf[JSReference], value)
      }
      Set(State(rv, env, localStack, a, newMemory))

  case State(KInfixExprComplete(op, rv1, rv2), env, localStack, a, memory) =>
      val newMemory = memory.copy(state)
      for {
          v1 <- newMemory.getValues(rv1)
          v2 <- newMemory.getValues(rv2)
      } yield {
          val res = infixFunc(op, v1, v2, newMemory)
          res.generateFrom(state.e)
          val address = newMemory.save(res)
          State(address, env, localStack, a, newMemory)
      }
  ...
}
```

Figure 5.9: Parts of application transition rules for expressions

machine.

$$putValue : Memory \times JSReference \times JSValue \to Unit$$

$$getValues : Memory \times JSReference \to \mathcal{P}(JSValue)$$

$$pushFrame : Memory \times StackAddress \times Frame \to Unit$$

$$getFrames : Memory \times StackAddress \to \mathcal{P}(Frame)$$

The method `putValue` and `pushFrame` imperatively updates value store and continuation store respectively. They will join any given value or frame to the existing values or frames that inhabit the same address.

$$m.putValue(a, v) = this.store[a] := this.store(a) \cup \{v\}$$

$$m.pushFrame(a, f) = this.stack[a] := this.stack(a) \cup \{f\}$$

The method `getValues` and `getFrames` retrieve values and stack frames non-deterministically.

$$m.getValues(a) = this.store(a)$$

$$m.getFrames(a) = this.stack(a)$$

5.4 Functions, Methods, and Constructors

The semantics of function invocation in JavaScript is more complex than many other programming languages. There are at least three patterns of

invocations, function call, method call, and new call.

Application transition rule of function call is shown in Figure 5.10, which extracts called functions (closures in function objects) and arguments by `getValues`, and extends the closure environment with the arguments. The variable `this` is regarded as an implicit parameter and we map `this` to the address of the global object (`window` in browser environment and `global` in "node.js"). Next, we save the current computation context (return point, local stack, evaluation environment, and stack pointer) to a frame and push it onto the stack (continuation store). The address of the top of stack is generated by the function `allocStackAddress`, which implements the call/return strategy of JsCFA and we will describe it in Section 5.7.2. Finally, the abstract interpreter will evaluate the function bodies with extended environments and an empty local stack.

Because JavaScript is a also prototype-based object-oriented language, it uses first-class functions to implement methods and constructors of objects. Therefore, the only difference between function calls and method calls is that a method invocation extracts function object from the receiver and explicitly brings `this` arguments. So we need to set `this` to the "receiver" in the method environment. New call (to functions named as constructors) is another type of function invocation form in JavaScript, which generates an

```
case State(KFuncCallComplete(funcRef, args),
           env, localStack, a, memory) =>
    val newMemory = memory.copy(state)
    for {
        f <- newMemory.getValues(funcRef)
        //get function objects
        if isCallable(f)
        JSClosure(func@FunctionExpr(name, ps, funcBody),
                  savedEnv) = f.code
        //get closures from the function object
    } yield {
        val psAddress = ps.map(alloc(_))
        //allocate addresses for formal parameters
        val thisAddress = biGlobalObjectRef
        // address of global object
        var newEnvPart =
          ("this" -> thisAddress) ::
            ps.map(x => x.str).zip(psAddress)
          //extend environment with "this" and other parameters
        name match {
          case Some(x) =>
            newEnvPart = (x.str -> alloc(x)) :: newEnvPart
            //extend environment with the function name
          case None =>
        }
        val newEnv = savedEnv ++ Map(newEnvPart: _*)

        for(p <- psAddress.zip(args)) {
          for(vs <- newMemory.getValues(p._2)) {
            newMemory.putValue(p._1, vs)
              //pass actual parameters to formal parameters
          }
        }
        val nextAddr = allocStackAddress(state, funcBody, newEnv)
        //allocate the stack frame

        newMemory.pushFrame(nextAddr,
            Frame(alloc(state.e),localStack, env, a))

        State(funcBody, newEnv, emptyLocalStack,
              nextAddr, newMemory)
    }
```

Figure 5.10: Application transition rule for global function calls

empty object at the "new call site", and passes the object as `this` parameter. Finally abstract interpreter returns the generated object as the result of the new call. In JsCFA, we implement the "return the generated object" by a trick, which puts an extra local continuation object `KUseValue(obj)` into the new `localStack`. `KUseValue(obj)` represents a low-level instruction that indicates: if the constructor function returns primitive values (not objects), the abstract interpreter should throw away constructor's original return values and use the specific object `obj` that is created at the "new call site" to replace.

5.5 Configurable Context-Sensitivity and Adaptive Object-Sensitivity

Context sensitivity is an effective approach to improve the precision and the performance of static analysis. Different context choices yield different analysis results and performance. On the one hand, each kind of context sensitivity strategy (e.g. call-site sensitive [26], argument sensitive [1], object sensitive [21, 27], field sensitive [17]) contributes considerable precision influence on specific problems. On the other hand, certain contexts can select different precision levels with different performance costs, such as call-site sensitivity in k-CFA. One of the most crucial contributions of AAM is that

it makes context sensitivity configurable. The contexts of polyvariant values are only determined by $\widetilde{alloc}$ and the data type of $\widetilde{Addr}$. We can even mix several context sensitivity strategies in one analyzer. For example, JsCFA allocates context insensitive addresses for most of the values using $\widetilde{alloc_{JsCFA}}$ as shown below.

$$\widetilde{alloc}_{JsCFA}(expression) = JSReference(expression.id)$$

Each expression stores its result (values) in the slot indicated by its syntactic label (`expression.id`). However, the context insensitive addresses will dramatically reduce the precision due to the dynamic features of JavaScript.

The object model of JsCFA is similar to that of λ_{JS} [12], which regards each JavaScript object as a map.

$$JSObject : JSString \rightarrow JSReference$$

The keys of an object are strings mapping to addresses that point to actual values in the store. Additionally, each object contains two special key/value pairs, `"__proto__"` and `"constructor"`, which are used to implement prototype-based inheritance of JavaScript.

Consider the following example, which dynamically adds an field to an object.

```
obj["p"] = e^l //"p" is not in obj
```

In monovariant analysis, if several objects flow into `obj` from branch or sequential paths, after the expression is evaluated, all of the objects would have a field `"p"` that points to the values allocated in `JSReference(l)`. However, if another expression assigns to field `"p"` for one of these objects, this modification will propagate to any other objects that flowed into `obj`.

To address this problem, we apply object-sensitive allocation for the dynamic object extension.

$$\widetilde{alloc}^{o}_{JsCFA}(expression, object) = JSReference(expression.id, object.id)$$

The object-sensitive allocation function separates dynamically added fields in different dimensions for each object. Only the application transition rules for `LDot` and `LBracket` use $\widetilde{alloc}^{o}_{JsCFA}$ for adding fields, while other transitions are context insensitive. Ultimately, JsCFA achieves more precision dynamic objects without significant overhead.

5.6 Abstract Garbage Collection as Stack Filtering

In practice, perfect call/return matching with monovariant analysis is not too useful. Let's revisit the simple example in Section 1. If our abstract interpreter can match call/return flows perfectly, the variable `a` would get

value `1` after executing call site 1. Then, when call site 2 invokes function `id` again, the new argument `#t` merges with `1` that was passed into `x` by call site 1. Finally, the merged abstract value `{1, #t}` returns to variable `b`. This kind of spurious result may flow into the rest of abstract interpretation and its accumulative effect will dramatically impact the analysis precision of higher-order programs.

For example, in the following function `compose-same`, the local variable `f` is called twice.

```
function compose-same(f, x) {
  return f(f(x));
}
```

In runtime, these two call sites always invoke the same function. However 0-CFA or k-CFA without enough context length may compose different closures for these two call sites when `compose-same` is called multiple times and several different functions flow into `f`. This spurious control flow problem (a.k.a. fake rebinding [31]) not only yields bad analysis result, but also increases the running time of the analysis. This is also known as the environment problem [26]. Traditional k-CFA attempts to resolve this problem via introducing context-sensitive (polyvariant) analysis. However, polyvariance is neither efficient nor a sufficient solution to this problem.

The reason why different actual parameters merge into the same formal

parameters is that the monovariant abstract interpreters breaks the concrete semantics. Concrete interpreters never merge parameters from different call sites because local variables will be deleted when the interpreter exits from the function. To solve this problem, CFA2 invented an approach named "stack filtering", which simulates the semantics of popping the stack frames to remove useless values of local variables. However, stack filtering has two limitations so that it cannot be ported to AAM. On one hand, AAM adopts reference model for all of the values (all things in stores), but CFA2 has stack allocated values. On the other hand, we cannot always pop stack frames after a function call returns because continuation stores may become graphs that contain cycles rather than stacks (see Section 4.2).

Earl et al. described introspective pushdown control flow analysis in [7] that integrates abstract garbage collection [20] in PDCFA to implement stack filtering. JsCFA also adopts this strategy to improve analysis precision and makes call/return matching more useful. The semantics of abstract garbage collection is the same as its counterparts in concrete interpreters. We first scan the current state to acquire the root set and trace from the root set to reach all the objects' fields and closures' non-local variables. Each address reached in the last phase (computing root set and tracing) is recorded in a "mark set", and values referred by the addresses that do not appear in the mark set are regarded as garbage.

However, the effect of abstract garbage collection is relatively weak in global store because the values referred by the unexecuted paths have to stay in the store even though the current state cannot reach them. Therefore, Earl et al. implemented PDCFA with abstract GC with per-state stores to achieve the full power of abstract GC. Although AAM with per-state stores theoretically has exponential time complexity, in practice, its performance is much better than AAM with global store but without abstract GC. Consequently, we also implement JsCFA with per-state stores. Moreover, we used two techniques to optimize the performance of abstract GC with per-state stores. Firstly, JsCFA implements the stores with copy-on-write since only the application states may change the store, so evaluation and continuation states interpreted between two application states share one store. Secondly, abstract GC never directly deletes values even if they are detected as garbage. When the abstract interpreter requires a new copied store, we just copy values that are referred by mark set (reached values) to eliminate the overhead of imperative deleting elements in stores. The class `Memory` provides the method `copy` that launches GC and returns a new memory instance only containing reachable elements.

$$copy : Memory \times State \to Memory$$

5.7 Pushdown CFA without Program Execution History

h-CFA is an effective and easy-to-implement method for perfect call/return matching in monovariant analysis on ANF-styled programs. However, ANF transformation limits the improvement of precision of highly dynamic languages such as JavaScript. Although, static analysis techniques already obtain acceptable perform for higher-order languages in theory, they are still not good enough for actual dynamic languages. For example, because JavaScript is a prototype-based language that has no native "inheritance" semantics, programmers usually implement their own "inherit" or "extend" functions to simulate inheritance.

```
function extend(target, source) {
  for(var propName in source) {
    if(source.hasOwnProperty(propName)) {
      target[propName] = source[propName];
    }
  }
  return target;
}
```

The function `extend` accepts two objects as parameters, `target` and `source`.

Then all fields of object `source` are copied into `target`. If we apply traditional static analysis (points-to analysis) techniques to programs that extensively use this function, the precision would be dramatically decreased. In a monovariant analysis on function `extend`, all field names (strings) of `source` flow into `propName` and merges to a "variable string" (top value of string in JsCFA), and all the field values of `source` are merged in this field of `target`.

Traditional monovariant analysis does not handle the high-level semantics that copy the fields from one object to another. Fortunately, there are techniques that attempt to recognize this kind of high-level semantics. Correlation tracking [28] is one of these techniques that matches *correlated dynamic property access* patterns that are often used to implement `extend`. That approach injects local context using the values of the *propName* in the body of the for-loop so that the field values of `source` is never merged in `target`.

To implement this strategy in JsCFA, we have to retain the code patterns of the input programs in the intermediate representations (IR) for the abstract interpreter. This is the most significant reason why we adopted AST as the IR of JsCFA rather than some low-level forms such as ANF. However, if we revise h-CFA to work with AST in JsCFA, it would require extra effort to record the program execution history. For this reason, JsCFA implements h-CFA using an indirect approach that can still achieve perfect call/return

matching without recording execution histories.

5.7.1 The Essence of *h*-CFA

Before adjusting h-CFA for JsCFA, we should discuss the most significant reason why abstract interpreters require program execution history for push-down CFA. Consider the example in Figure 4.1 again: after analyzing `(a (fib 10))`, function `fib` is invoked again and we have to recompute the recursive call site `(fib res2)` in a new analysis environment. At this point, the continuation store $\widetilde{\sigma_k^H}$ looks like below.

$$\widetilde{a_k^H}_{11} = ((fib\ 10), \widehat{fib}, \{fib\})$$

$$\widetilde{a_k^H}_{12} = ((fib\ res2), \widehat{fib}, \{fib, res1, res2\})$$

$$\widetilde{a_k^H}_{21} = ((fib\ 20), \widehat{fib}, \{fib, a\})$$

$$\widetilde{a_k^H}_{22} = ((fib\ res2), \widehat{fib}, \{fib, a, res1, res2\})$$

$$\widetilde{\sigma_k^H} = \{\widetilde{a_k^H{}_{11}} \mapsto \{(a, (let\ (b\ (fib\ 20))\ \dots), \widetilde{env}_1, \{fib\}, \widetilde{a_{k\ init}^H})\}$$

$$\widetilde{a_k^H{}_{12}} \mapsto \{(res3, (let\ (res4\ (-\ n\ 2))\dots), \widetilde{env}_2, \{fib, res1, res2\}, \widetilde{a_k^H{}_{11}})$$

$$(res3, (let\ (res4\ (-\ n\ 2))\dots), \widetilde{env}_3, \{fib, res1, res2\}, \widetilde{a_k^H{}_{12}})$$

$$\dots\}$$

$$\widetilde{a_k^H{}_{21}} \mapsto \{(b, (\dots), \{fib, a\}, \widetilde{a_{k\ init}^H})\}$$

$$\widetilde{a_k^H{}_{22}} \mapsto \{(res3, (let\ (res4\ (-\ n\ 2))\dots), \widetilde{env}_2, \{fib, a, res1, res2\}, \widetilde{a_k^H{}_{21}})$$

$$(res3, (let\ (res4\ (-\ n\ 2))\dots), \widetilde{env}_3, \{fib, a, res1, res2\}, \widetilde{a_k^H{}_{22}})$$

$$\dots\}$$

$$\dots$$

$$\}$$

As seen above, when the CESK^H machine finishes analyzing the function call from `(fib res2)` at certain recursive level, control flow will back to the return point `b` along the stack $\dots \to \widetilde{a_k^H{}_{22}} \to \widetilde{a_k^H{}_{21}} \to \widetilde{a_{k\ init}^H}$. There is no mismatch between returns and corresponding calls.

Then, we show what would happen if we remove program execution histories from $\widetilde{KAddr}$ and the new continuation address has just the current call site and called function.

$$\widetilde{kalloc_{non-h}}((e, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_k, \tilde{a}_k), e', \tilde{\rho}', \tilde{\sigma}') = (e, e')$$

This continuation allocation is similar to 1-CFA that characterizes function

entries by the last call site. Unfortunately, in the 1-CFA-like analysis, we cannot distinguish $\widetilde{a_k^H{}_{22}}$ from $\widetilde{a_k^H{}_{12}}$, which is the stack frame pointer of the states that go into `(fib res2)`. In this case, the frames referred by $\widetilde{a_k^H{}_{22}}$ and $\widetilde{a_k^H{}_{12}}$ are merged into the slot in the continuation store.

$$\widetilde{a_{k11}} = ((fib\ 10), \widehat{fib})$$

$$\widetilde{a_{k21}} = ((fib\ 20), \widehat{fib})$$

$$\widetilde{a_{k2}} = ((fib\ res2), \widehat{fib})$$

$$\begin{aligned}
\tilde{\sigma}_k = \{&\widetilde{a_{k11}} \mapsto \{(a, (let\ (b\ (fib\ 20))\ \dots), \widetilde{env_1}, \widetilde{a_{kinit}})\} \\
&\widetilde{a_{k21}} \mapsto \{(b, (\dots), \widetilde{a_{kinit}})\} \\
&\widetilde{a_{k2}} \mapsto \{(res3, (let\ (res4\ (-\ n\ 2))\dots), \widetilde{env_{21}}, \widetilde{a_{k21}}) \\
&\qquad\qquad (res3, (let\ (res4\ (-\ n\ 2))\dots), \widetilde{env_{11}}, \widetilde{a_{k11}}) \\
&\qquad\qquad (res3, (let\ (res4\ (-\ n\ 2))\dots), \widetilde{env_2}, \widetilde{a_{k2}}) \\
&\qquad\qquad \dots\} \\
&\dots \\
&\}
\end{aligned}$$

After `(fib res2)` is analyzed, the abstract interpreter will be confused by merged stack, which returns through either $\dots \to \widetilde{a_{k2}} \to \widetilde{a_{k21}}$ or $\dots \to \widetilde{a_{k2}} \to \widetilde{a_{k11}}$. In other words, the analysis result of `(fib 20)` also can flow into `a`.

The reason why removing program execution history causes mismatching call/return flows is that old call stacks of finished computation can impact

later call stacks. This call stack merging is also due to abstract interpreters violating concrete semantics. As shown in Section 5.6, call stack frames ought to be reclaimed after a function call completes, but the cycles in the stack graph do not allow us to simply pop the stack frames.

5.7.2 Abstract Garbage Collection as Popping Call Stack Frames

Inspired by the abstract garbage collection on value stores (see Section 5.6), we implemented JsCFA using abstract GC to achieve call/return matching without program execution history. The continuation address in JsCFA removes the $\tilde{h}$ field from $\widetilde{KAddr^H}$ and the function entry points are only encoded by the call site's label ($e.id$) and callee's label ($e'.id$).

$$\widetilde{kalloc_{JsCFA}}(State(e, env, localStack, a, memory), e') = StackAddress(e.id, e'.id)$$

Before copying *memory*, JsCFA starts abstract GC that eliminates unreachable elements in the stack (continuation store). When JsCFA begins to analyze the call site `(fib 20)` in the example shown in Figure 4.1, old stack frames generated by previous computation are already reclaimed.

$$\widetilde{a_{k21}} = ((fib\ 20), \widehat{fib})$$

$$\tilde{\sigma}_k = \{\widetilde{a_{k21}} \mapsto \{(b, (\dots), \widetilde{a_{kinit}})\}\}$$

Then the abstract interpreter dives into this call site. When the call completes, there is no garbage data in the stack to confuse the return flows.

$$\widetilde{a_{k21}} = ((fib\ 20), \widehat{fib})$$

$$\widetilde{a_{k2}} = ((fib\ res2), \widehat{fib})$$

$$\begin{aligned}
\tilde{\sigma}_k = \{&\widetilde{a_{k21}} \mapsto \{(b, (\dots), \widetilde{a_{kinit}})\} \\
&\widetilde{a_{k2}} \mapsto \{(res3, (let\ (res4\ (-\ n\ 2))\dots), \widetilde{env_{21}}, \widetilde{a_{k21}}) \\
&\qquad (res3, (let\ (res4\ (-\ n\ 2))\dots), \widetilde{env_2}, \widetilde{a_{k2}}) \\
&\qquad \dots\} \\
&\dots \\
&\}
\end{aligned}$$

Therefore, the control flow can only go back to the return point ɒ through the stack $\dots \to \widetilde{a_{k2}} \to \widetilde{a_{k21}} \to \widetilde{a_{kinit}}$.

Using abstract garbage collection, JsCFA can realize perfect call-return matching by analyzing programs without ANF transformation.

5.8 Evaluation

JsCFA is written in Scala and executed with Scala 2.11. We tested its performance on a personal computer that is equipped with Intel Core i7 (2.3 GHz), 16GB RAM with OSX operating system. This performance evaluation was based on SunSpider benchmark suit [22] and the result is shown

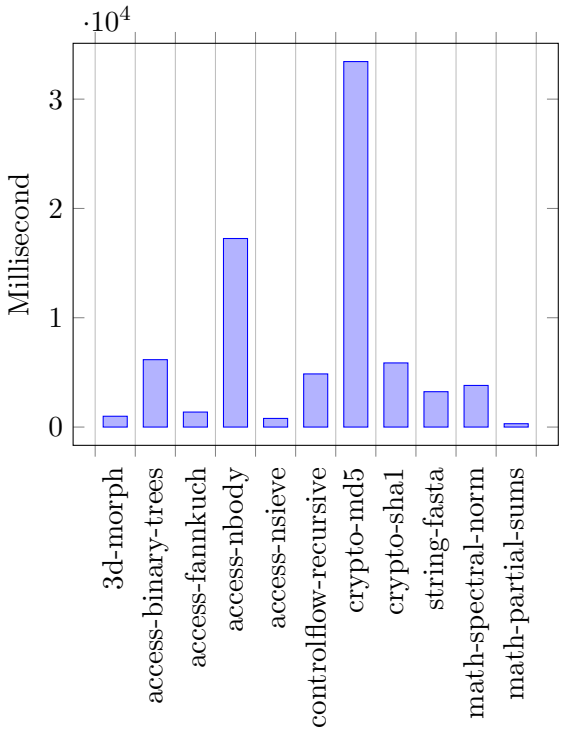


Figure 5.11: JsCFA benchmarks

in Figure 5.11. We also collected the statistics of mismatching returns (see Section 4.3), which shows JsCFA has no any spurious return flows on these test programs.

6 Future Work

Performance The most serious problem of JsCFA is that it uses a naive implementation of AAM framework, which causes the abstract interpreter to be relatively inefficient. Although AAM provides a systematic approach for constructing correct abstract interpreters, the performance is much slower than traditional hand-optimized analyzers. Therefore, our implementation of JsCFA spends too much time analyzing large-scale programs. Fortunately, there are several existing optimizations for accelerating the computation in AAM. Johnson, Labich, Nicholas, Might, and Van Horn introduced OAAM (optimizing abstracting abstract machine [14]), which is a series of techniques to refine the performance of AAM. This includes timestamped frontier, log-based store deltas, laziness, and abstract compilation that all can dramatically promote AAM in practice or theory. Thus, in the further research we are going to apply these optimizing techniques for JsCFA.

Interface Although, h-CFA and JsCFA are much easier to implement and understand than traditional abstract interpreters, they still have a common disadvantage that complicates development of static analyzers. The described details show that implementing JsCFA is very similar to writing a concrete interpreter so that programmers can directly convert their compiler

or interpreter knowledge to static analysis. However, h-CFA and JsCFA are both based on CESK abstract machine that is a small-step semantic model. For realistic programming languages that have relatively complex syntax, the implementation of small-step semantics has to introduce many "continuation" components. Manually operating various continuations is tedious and the code of abstract interpreter is also not as intuitive as the code that implements big-step operation semantics. Consequently, in future work, we plan to design a domain specific language (DSL) that describes abstract semantics in the big-step style, and abstract interpreters written in the DSL can be compiled to small-step CESK or CESK^H machines. Olivier [4] discovered that there is an essential connection between big-step operational semantics and small-step CESK machine [8]. A big-step interpreter can be translated to an equivalent small-step interpreter though CPS transformation [3], defunctionalization [5], and fusion. Therefore, we will continue to design the DSL and the compiler to make abstract interpreters closer to concrete ones.

Precision Because JavaScript is a prototype-based and highly dynamic language that heavily relies upon objects, traditional control flow analysis is not enough for realistic JavaScript programs and libraries. Various previous works applied different techniques to analyze JavaScript or subsets

of it, such as pointer analysis, string analysis, and numeric range analysis. JSAI [16] is a static analyzer for JavaScript based on AAM that uses more sophisticated models for abstract objects, abstract strings, and constant propagation. TAJS [13] is another theory providing a well-designed type system to JavaScript static analysis. Since objects in JavaScript are maps from strings to values, precise string analysis benefits dynamically extended objects. Then, object analysis also impacts control flow analysis because methods are fields of objects. Additionally, arrays are just a kind of special object in JavaScript, a better string/number analysis also improves analysis for arrays. Consequently, we plan to mix these existing techniques with our control flow analysis solution to improve JsCFA.

7 Conclusion

We have described h-CFA, a simplified approach for implementing pushdown control flow analysis. This algorithm is based on abstracting abstract machine and it precisely matches returns with corresponding calls. It achieves this effect by adding program execution histories as context to continuations. We also showed the advantages of h-CFA compared with the existing pushdown CFA algorithms (i.e. PDCFA, AAC, P4F). We designed and implemented JsCFA, a control flow analyzer for JavaScript, which demonstrated that h-CFA is a practical approach for realistic programs. In addition, we discussed the reason why pushdown CFA requires polyvariant continuations to achieve call/return matching, and we used this essential property of pushdown CFA to implement h-CFA for JsCFA without recording program execution histories. Moreover, JsCFA adopted other techniques (i.e. abstract garbage collection) in collaboration with perfect call/return matching to improve the precision of the static analysis for JavaScript. In conclusion, we believe that h-CFA is a simple and precise technique of control flow analysis for real-world programming languages.

Bibliography

[1] Ole Agesen. The Cartesian Product Algorithm. In *European Conference on Object-Oriented Programming*, pages 2–26. Springer, 1995.

[2] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[3] Oliver Danvy and Andrzex Filinski. Representing Control: A Study of the CPS Transformation. *Mathematical structures in computer science*, 2(04):361–391, 1992.

[4] Olivier Danvy. Defunctionalized Interpreters for Programming Languages. In *ACM Sigplan Notices*, volume 43, pages 131–142. ACM, 2008.

[5] Olivier Danvy and Lasse R Nielsen. Defunctionalization at Work. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 162–174. ACM, 2001.

[6] Christopher Earl, Matthew Might, and David Van Horn. Pushdown Control-flow Analysis of Higher-order Programs. *arXiv preprint arXiv:1007.4268*, 2010.

[7] Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. Introspective Pushdown Analysis of Higher-Order Programs. In *ACM SIGPLAN Notices*, volume 47, pages 177–188. ACM, 2012.

[8] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. Mit Press, 2009.

[9] Mattias Felleisen and Daniel P Friedman. A Calculus for Assignments in Higher-order Languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, page 314. ACM, 1987.

[10] Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *ACM Sigplan Notices*, volume 28, pages 237–247. ACM, 1993.

[11] Thomas Gilray, Steven Lyde, Michael D Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *ACM SIGPLAN Notices*, volume 51, pages 691–704. ACM, 2016.

[12] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *European Conference on Object-Oriented Programming*, pages 126–150. Springer, 2010.

[13] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type Analysis for JavaScript. In *International Static Analysis Symposium*, pages 238–255. Springer, 2009.

[14] J Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. Optimizing Abstract Abstract Machines. *ACM SIGPLAN Notices*, 48(9):443–454, 2013.

[15] James Ian Johnson and David Van Horn. Abstracting Abstract Control. *ACM SIGPLAN Notices*, 50(2):11–22, 2015.

[16] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 121–132. ACM, 2014.

[17] Ondřej Lhoták and Laurie Hendren. Scaling Java Points-to Analysis Using Spark. In *International Conference on Compiler Construction*, pages 153–169. Springer, 2003.

[18] Jan Midtgaard. Control-Flow Analysis of Functional Programs. *ACM Computing Surveys (CSUR)*, 44(3):10, 2012.

[19] Matthew Might. *Environment Analysis of Higher-order Languages*. PhD thesis, Georgia Institute of Technology, 2007.

[20] Matthew Might and Olin Shivers. Improving Flow Analyses via ΓCFA: Abstract Garbage Collection and Counting. In *ACM SIGPLAN Notices*, volume 41, pages 13–25. ACM, 2006.

[21] Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005.

[22] WebKit Open Source Project. SunSpider JavaScript Benchmark, 2011.

[23] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[24] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theoretical Computer Science*, 167(1):131–170, 1996.

[25] Micha Sharir and Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*. New York University. Courant Institute of Mathematical Sciences. ComputerScience Department, 1978.

[26] Olin Shivers. *Control-Flow Analysis of Higher-order Languages*. PhD thesis, Citeseer, 1991.

[27] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick Your Contexts Well: Understanding Object-Sensitivity. In *ACM SIGPLAN Notices*, volume 46, pages 17–30. ACM, 2011.

[28] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation Tracking for Points-to Analysis of JavaScript. In *European Conference on Object-Oriented Programming*, pages 435–458. Springer, 2012.

[29] David Van Horn and Harry G Mairson. Deciding k-CFA is Complete for EXPTIME. *ACM Sigplan Notices*, 43(9):275–282, 2008.

[30] David Van Horn and Matthew Might. Abstracting Abstract Machines. In *ACM Sigplan Notices*, volume 45, pages 51–62. ACM, 2010.

[31] Dimitrios Vardoulakis and Olin Shivers. CFA2: A Context-Free Approach to Control-Flow Analysis. In *European Symposium on Programming*, pages 570–589. Springer, 2010.

[32] Andrew K Wright and Suresh Jagannathan. Polymorphic Splitting: An Effective Polyvariant Flow Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):166–207, 1998.