The Essence of Pushdown Control Flow Analysis and Efficient Application for JavaScript Analysis

Fei Peng

June 11, 2016

Abstract

In traditional control flow analysis area, call/return mismatch is always a problem that dramatically reduces precision of analysis. Although, original k-CFA uses bounded call strings to acquire limited call/return exact match, this technique causes serious performance cost due to it coupling call/return match strategies with context-sensitivity of values. Meanwhile, abstracting abstract machine (AAM) a configurable framework for constructing abstract interpreters introduces store-allocated continuation that makes soundness of abstract interpreters easily acquired. Recently, there are three different approaches (PDCFA, AAC, and P4F) published, which provide perfect call/return match for AAM via modeling call stacks as pushdown systems. However, PDCFA needs extra annotated edges to represent stack actions that leads implementing become difficult. AAC requires expensive overhead, and P4F cannot work with monovariance (0-CFA). Consequently, we developed a new method to address the call/return mismatch problem that is extremely easy to implement for ANF style program in abstracting abstract machine. Simultaneously, this method reveals the essence of pushdown control flow analysis, and we exploit the essence to develop a static analyzer for JavaScript with direct abstract syntax tree.

1 Introduction

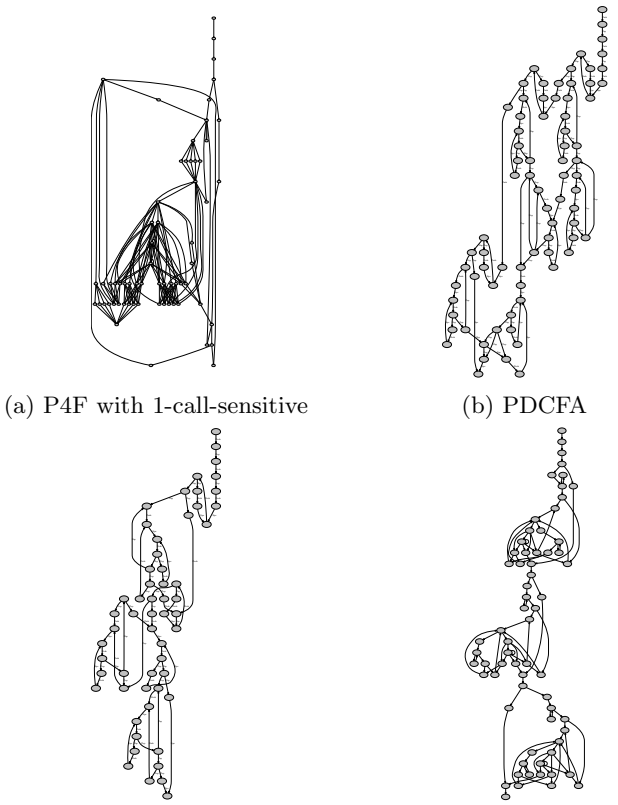
Dynamic programming languages (e.g. JavaScript, Python, and Ruby, etc.) play a significant role in a lot of computing areas, such as system management, web development, and scientific computing. Especially, in the past decade, JavaScript becomes a ubiquitous computing environment. However, their certain features (e.g. duck-typing, first-class function, highly dynamic object model) make bug detection difficult. Control flow analysis becomes a good approach to detect deeply semantic defect before actual running programs, but original control flow analysis (k-CFA) is too imprecise to apply in realized programs. For example, call/return mismatch is always a problem in k-CFA that dramatically reduces precision of analysis.

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

Let's consider about the trivial example, in traditional 0-CFA the id function is called twice and #t finally flows into variable a because there is a spurious flow from call site (id #t) return to (a (id 1)). In k-CFA (k $\geq$ 1), the values of local variable x are distinguished by different call site environments. To illustrate, $(x, 2) \longrightarrow \#t$ means that the value of x is #t in call site 2, and original k-CFA also can use the values' environment to filter inter-procedure control flows, which value of x from call site 2 only can be returned to (b (id #t)). In this case (non-recursive program), call string with finite length is enough for providing precise call/return flow (value and control flow both). However, any recursive function call will break the rule and propagate spurious information to the whole program. Meanwhile, the performance is unacceptable even when k is 1.

The core idea of pushdown control flow analysis is to mimic call/return flow as a unbounded call stack for ordinary calls, and summarizes the call stack for recursive calls because unbounded call stack is uncomputable in static analysis. CFA2 implements the summarization with a tabulate algorithm, and PDCFA annotates state transition edges with stack actions (push, pop, and no action).Both of CFA2 and PDCFA introduce extra semantics for target languages that makes the abstract interpreters hard to implement. Fortunately, XXX invented abstracting abstract machine (AAM) as a configurable framework for constructing abstract interpreters in the CESK abstract machine style. AAM not only allocate values in the store (like original k-CFA does), but also represents control flow as store-allocated continuations. In AAM, each CESK state does not directly carry continuation, but a continuation address refers to a set of continuations in the store. Merging serval continuations in one continuation address achieves approximation of control flows. Meanwhile, AAM brings two benefits to control flow analysis. On the one hand, it makes soundness of abstract interpreters easily acquired because values and continuations are both in the store and the store size is fixed. On the other hand, store-allocated traditional separates original context-sensitivity (polyvariant values) strategies from call/return match techniques. AAC and P4F both based on AAM convert call/return match problem to continuation address allocation strategy. AAC requires expensive overhead, $O(n^9)$, which puts too much information in the continuation addresses. Then P4F tries to reduce the complexity of AAC, but it is not useful for monovariant analysis.

In this paper, we introduce a new method to address the call/return mismatch problem that is as simple as P4F and AAC to implement, and it can provide perfect call/return match for monovariant and polyvariant control flow analysis. The new method puts a *program execution history* into continuation address with callee's function boy, so we name it h-CFA. The program execution history can be regarded as call strings with automatically determined length. For non-recursive calls, execution history always provides enough precise con-



(a) P4F with 1-call-sensitive (b) PDCFA

(c) PDCFA with Abstract Garbage Collection (d) h-CFA

Figure 1: State transition graphs of: (1) P4F (pushdown control flow analysis for free) with 1-CFA (1-call-site sensitive); (2) PDCFA (pushdown control flow analysis); (3) PDCFA with abstract garbage collection; (4) h-CFA.

text information, no matter how deep the call sequences. On the other hand, execution history can automatically stop growth for recursive calls, and the work list algorithm will be responsible for finding the fixed-point of recursive computation. Furthermore, we claim that for call/return match (polyvariant continuations) finite length of k is enough, which can reaches n (size of input program) in worst cases.

We implementeded P4F, PDCFA, PDCFA with GC and h-CFA for Scheme, and generated state transition graphs (Figure 1) of these four algorithms for a text program in Figure 2.

As the h-CFA graph shows, there are three similar subgraphs in the state transition process, which obviously illuminate no call/return flow merged in h-CFA due to three subgraphs connected by single transition edges.

```
(define (fib n)
  (if (< n 3)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))

(define a (fib 10))
(define b (fib 20))
(define c (fib 100))
```

Figure 2: The simple example invokes recursive fibonacci function three times on its nontrivial condition. We use the example to compare call/return match strengths of four pushdown control flow analysis algorithms.

Table 1: Pushdown Control Flow Analysis Comparison

Algorithm	Match Strength	Mono -variance	Poly -variance	Implementing	Complexity on Monovariance
CFA2	Infinite	✓		Difficult	Exponential
P4F	Limited		✓	Easy	N/A
PDCFA	Infinite	✓	✓	Difficult	$O(n^6)$
AAC	Infinite	✓	✓	Easy	$O(n^9)$
h-CFA	Infinite	✓	✓	Easy	$O(n^5)$

1.1 Contributions

2 Pushdown Control Flow Analysis in Abstracting Abstract Machines

2.1 Abstracting Abstract Machine

Before section 4, we will describe algorithms on ANF style lambda calculus.

$$\begin{aligned} e \in Exp ::= & \ (let\ ((x\ (f\ ae)))\ e) \\ & \mid ae \end{aligned}$$

$$f, ae \in AExp ::= x \mid lambda \qquad \text{[atomic expressions]}$$

$$lambda \in Lambda ::= (\lambda\ (x)\ e) \qquad \text{[lambda abstractions]}$$

$$x, y \in Var\ is\ a\ set\ of\ identifiers \qquad \text{[variables]}$$

This section reviews AAM.
Definitions:

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

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

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

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

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

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

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

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

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

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

Abstract Semantics:
alloc:

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

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

calls:

$$\overbrace{((let\ ([y\ (f\ ae)]\ e)), \widetilde{\rho}, \widetilde{\sigma}, \widetilde{\sigma_k}, \widetilde{a_k})}^{\widetilde{\varsigma}} \rightsquigarrow (e', \widetilde{\rho}', \widetilde{\sigma}', \widetilde{\sigma_k}', \widetilde{a_k}'), where$$

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

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

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

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

$$\widetilde{\sigma_k}' = \widetilde{\sigma_k} \sqcup [\widetilde{a_k}' \mapsto (y, e, \widetilde{\rho}, \widetilde{a_k})]$$

$$\widetilde{a_k}' = \widetilde{kalloc}(\widetilde{\varsigma}, e', \widetilde{\rho}', \widetilde{\sigma}')$$

returns:

$$\overbrace{(ae, \widetilde{\rho}, \widetilde{\sigma}, \widetilde{\sigma_k}, \widetilde{a_k})}^{\widetilde{\varsigma}} \rightsquigarrow (e, \widetilde{\rho}', \widetilde{\sigma}', \widetilde{\sigma_k}, \widetilde{a_k}', \widetilde{h}')$$

$$(x, e, \widetilde{\rho_k}, \widetilde{a_k}') \in \widetilde{\sigma_k}(\widetilde{a_k})$$

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

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

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

2.2 Defect of AAM

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

$$\widetilde{kalloc_{P4F}}((e, \widetilde{\rho}, \widetilde{\sigma}, \widetilde{\sigma_k}, \widetilde{a_k}), e', \widetilde{\rho}', \widetilde{\sigma}') = (e', \widetilde{\rho}')$$

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

3 The Essence of Pushdown Control Flow Analysis

Inspired by P4F, we deem that pushdown analysis (polyvariant continuation store) is orthogonal from polyvariant store, in other words, control flow analysis can get call/return match that does not depend on polyvariant values. Simultaneously, 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 cur-

rent state. During evaluating function calls, the program execution history can be used to uniquely represent current call site in continuation store.

3.1 Program Execution History

First, we modify the $CESK$ machine defined in Section 2.1 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.

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

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

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

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

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

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

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

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

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

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

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

In ANF programs, environment naturally maintains intro-procedural execution history because ANF explicitly extracts intro-procedural control flows in let-bindings and saves every 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, execution history can automatically stop growth for recursive calls, and the work list algorithm will be responsible for finding the fixed-point of recursive computation.

Following definitions describe the abstract semantics of $CESK^H$ machine.

Calls:

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

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

$$\tilde{\rho}' = \tilde{\rho}_\lambda[x \mapsto \tilde{a}]$$
$$\tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a} \mapsto \widetilde{eval}(ae, \tilde{\rho}, \tilde{\sigma})]$$
$$\tilde{a} = \widetilde{alloc}(x, \widetilde{\varsigma^H})$$
$$\widetilde{\sigma_k^H}' = \widetilde{\sigma_k^H} \sqcup [\widetilde{a_k^H}' \mapsto (y, e, \tilde{\rho}, \tilde{h}, \widetilde{a_k^H})]$$
$$\widetilde{a_k^H}' = \widetilde{kalloc_h}(\tilde{\varsigma}, e', \tilde{\rho}', \tilde{\sigma}')$$
$$\tilde{h}' = \tilde{h}[x \mapsto \tilde{a}]$$

Returns:

$$\overbrace{(ae, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma_k}, \widetilde{a_k^H}, \tilde{h})}^{\widetilde{\varsigma^H}} \rightsquigarrow (e, \tilde{\rho}', \tilde{\sigma}', \widetilde{\sigma_k^H}, \widetilde{a_k^H}', \tilde{h}')$$
$$(x, e, \tilde{\rho_k}, \tilde{h_k}, \widetilde{a_k^H}') \in \widetilde{\sigma_k^H}(\widetilde{a_k^H})$$
$$\tilde{\rho}' = \tilde{\rho_k}[x \mapsto \tilde{a}]$$
$$\tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a} \mapsto \widetilde{eval}(ae, \tilde{\rho}, \tilde{\sigma})]$$
$$\tilde{a} = \widetilde{alloc}(x, \widetilde{\varsigma^H})$$
$$\tilde{h}' = \tilde{h_k}[x \mapsto \tilde{a}]$$

The semantics of function calls propagate execution history by adding current "intermediate variable" to $\tilde{h}$, but the updating of execution history is different from environment extension that recoveries the environments from function definition points. In returns' definition, abstract interpreters restore up-level's history from the frames refereed by current continuation address, and this behavior is similar with environment restoring. Then $\widetilde{kalloc_h}$ function 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 stie 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 distinguishs as many as possible call sites of e' via the very last call site e and the rests encodeed by $\tilde{h}$.

```
(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)))
    ...))
```

Figure 3: We rewrite the former example in ANF style. For convenient demonstrating, we use complete Scheme language with numbers and booleans instead of pure lambda calculus.

3.2 Polyvariant Continuation

Let's exam the analysis process of a simple example to understand h-CFA, the example is similar with Figure 2 but written in ANF style showed by Figure 3.

In this subsection, execution histories are briefly represented as variable sequences, and the called function (the second part of $\widetilde{KAddr^H}$) is replaced by its function name wearing hat. This temporary convince just makes the explanation more readable but not actually modify the abstract semantics of $CESK^H$ machines.

Through steps of the abstract interpreter, the first call site (a (fib 10)) carries history $\{fib\}$ that means in this program point we only finished computing the declaration of 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 $(a, (let\ (b\ (fib\ 20))\ \ldots), \widetilde{env}_1, \{fib\}, \widetilde{a^H_{k\ init}})$ that is the one and only element of the continuation store so far. The stack frame expresses that after complete this invocation, (1) 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 recover continuation address $\widetilde{a^H_{k\ init}}$ a fake continuation address representing top level's continuation.

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.

$$\widetilde{a^H_{k\ 1}} = ((fib\ 10), \widehat{fib}, \{fib\})$$

$$\widetilde{a^H_{k\ 2}} = ((fib\ res2), \widehat{fib}, \{fib, res1, res2\})$$

$$\widetilde{\sigma_k^H} = \{\widetilde{a_k^H}_1 \mapsto \{(a, (let\ (b\ (fib\ 20))\ \ldots), \widetilde{env}_1, \{fib\}, \widetilde{a_{k\ init}^H})\}$$
$$\widetilde{a_k^H}_2 \mapsto \{(res3, (let\ (res4\ (-\ n\ 2))\ldots), \widetilde{env}_2, \{fib, res1, res2\}, \widetilde{a_k^H}_1)\}\}$$

As above illustration shows, the continuation store is a linked stack structure. Each frame has a $\widetilde{a_k^H}$ that points to the next frame in the stack. This stack-like structure perfect mimics call stacks of concrete interpreters. Certainly, the call stie (fib res4) will also gets its own execution history after computation of (fib res2) becomes complete (acquired 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 be added new element to distinguish (res3 (fib res2)) from its variances of 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 merge 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".

$$\widetilde{a_k^H}_2 \mapsto \{(res3, (let\ (res4\ (-\ n\ 2))\ldots), \widetilde{env}_2, \tilde{h}_2, \widetilde{a_k^H}_1),$$
$$(res3, (let\ (res4\ (-\ n\ 2))\ldots), \widetilde{env}_a, \tilde{h}_a, \widetilde{a_k^H}_2)\}$$

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 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 compute (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 summarize 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 length of execution histories under $O(n)$ (the size of input program) in worst cases.

Then the abstract interpreter restarts to execute the function and encounters call site (res3 (fib res2)) again. At this time, the continuation address allocated for call site (res3 (fib res2)) differs from last time, which make sure there are two distinct "call stacks". Consequently, function "fib" called from (b (fib 20)) will never return to (a (fib 10)), vice versa.

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

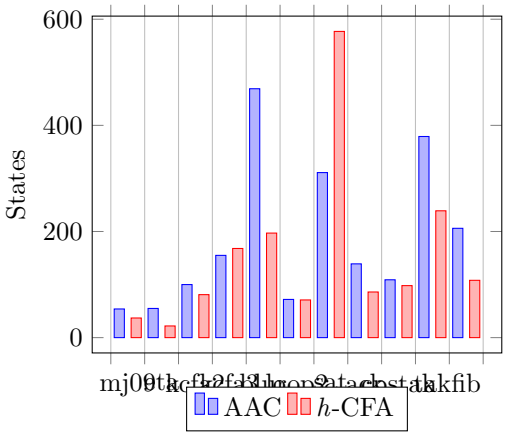


Figure 4: monovariant analysis comparison

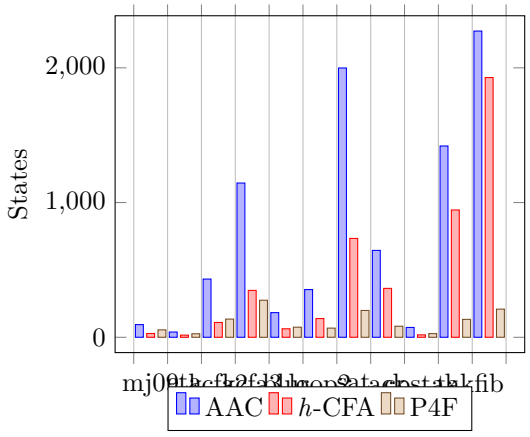


Figure 5: 1-call-site sensitive analysis comparison

3.3 Store-widenning

Theoretically, native implementations of above definitions take exponential time respecting the input program size.

Abstracting abstract machines usually adopt widening on stores and continuation stores. Store widening uses a global store (single-threaded store, Shivers []) rather than per-state stores for values and continuations respectively.

3.4 Defect of *h*-CFA

Although h-CFA provides precise call/return flows to monovariance and polyvariance both without extra effort, it just holds on ANF programs. Next section, we will show that certain situations and analysis strategies limit us not being able to compile input programs to ANF before analysis. Meanwhile, during exploring h-CFA, we become clear with why original k-CFA cannot perfect match infinite call/return flows. According to this discovery, we develop another pushdown analysis technique that is friendly for direct AST and on longer requires the execution history.

4 Static Analysis of JavaScript

4.1 Semantics of JsCFA

The abstract semantics of JsCFA is derived from JAM a unbounded stack semantics for JavaScript analysis. JAM describes a pushdown reduction semantics in λ_{JS}-calculus, and we modify it to work with direct high level AST of JavaScript in CESK abstract machine. The abstract syntax tree interface is given in figure 6 and figure 7. All the data structures are separated to two

categories that inherit from abstract class “Statement” and “Expression” respectively. Trait “AbstractSyntaxTree” defines field “id” to encode the statically and uniquely syntactic label for each AST node. Meanwhile, it decelerates and implements method “generateFrome” that spreads the statically syntactic information from AST nodes to local continuations and values. The top level of a paogram is a sequence of statements warped in “Script”.

```
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 6: Abstract syntax tree data types of statements

The core data structure of the AAM of JsCFA is class “State” (corresponding to $\tilde{\varsigma}$), which has six components “e” (control string), “env” (environment), “localStack” (intro-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[GlobalFrame]]) {
```

```
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 7: Abstract syntax tree data types of expressions

```
    ...
}
```

Above definition shows two differences from original AAM and h-CFA. Class "State" does not contain "History" field because we implement h-CFA indirectly for JsCFA, and this modification will be discuss at later section. Meanwhile, there is an extra fild "localStack" playing the role of intro-procedural continuation stack, but AAM and h-CFA not. In h-CFA, before analysis, ANF transformation already flats all intro-procedural control flows to let-binding, so it just need to tackle inter-procedural continuations in stores. Besides, original AAM saves all of the continuations (inter and intro-procedural) into continuation stores, but actually only inter-procedural control flows have to be retrieved nondeterministically and intro-procedural continuations are always deterministic. Consequently, we partition continuations in two places that make semantics clearer and performance better.

Following JAM, the abstracting abstract machine of JsCFA also distinguish three types of transition states: evaluation, continuation, and application.

Evaluation transition rules accept a state and match its control string component to generate next states. Evaluation transitions firstly search values and reducible expressions/statements (we refer states carry reducible expressions/statements to complete states in JsCFA) in the control strings. 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 current state is a complete state that its every recursive component is already computed to a value, the abstract machine transits to application to execute main semantics of the AST node. Eventually, when the control string is neither value nor reducible expression/statement, the machine generate the new continuation according to semantics of standard JavaScript. Then the new continuation is pushed on the local stack and the machine proceed to evaluate the program point whose computing result is required firstly.

Continuation transitions work on components that state objects have and a extra *next continuation* (referred to "cont" in figure 9 and figure 10). Control strings in these transition states are alway values, so abstract machines dispatch transitions via matching "cont" and plug the value into a new continuation. If the new continuation object is a reducible expression/statement, the next machine states will transition to application states. Otherwise, the new continuation that contains the value of control string is pushed into the local stack. Then, the following computating will be dispatched by next evaluation states. Finally, there is a special continuation state in JavaScript semantics. If no next continuation exists in the local stack (local stack is empty), that means the function does not have return statement through the current execution path. Therefore, we also implement "return" semantics at this point that the executing function returns "undefined" to its next return point restored from stack frames.

```
def transitEvaluation(state: State): Set[State] = state match {
  case completeState if isComplete(completeState) =>
      transitApplication(state)
  case State(v, env, localStack, a, memory) if isJSValue(v) =>
      val cont = topOfLocalStack(localStack)
      val newStack = popLocalStack(localStack)
      transitContinuation(cont, v, env, newStack, a, memory)

  //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))
  ...

  //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 8: Parts of Evaluation Transition Rules

```
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 9: Parts of Continuation Transition Rules for Statements

4.2 Abstract Garbage Collection as Stack Filtering

In practice, call/return perfect match only working with monovariant analysis is useless. Please consider the first simple example in section 1 again, if our abstract interpreter can match call/return flows perfect, 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" passed into "x" by call site 1. Then, the merged abstract value $\{1, \#t\}$ returns to variable "b". Meanwhile, this spurious analysis result will flow into following interpretation, and accumulated spurious values and flows will dramatically decrease precision of analysis. Traditional k-CFA attempts to resolve this problem via introducing context-sensitive (polyvariant) analysis. However, polyvariance is not a efficient and powerful solution to this problem.

The essential reason why different actual parameters merge into same formal parameters is monovariant abstract interpreters breaking 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. Consequently, CFA2 invents an approach named "stack filtering" making it more useful, which just returns #t to variable "b". Stack filtering simulates the semantics of popping call stack frames to remove useless values of local variables. However, there are two limitations that lead stack filtering not being able to port to AAM. On the one hand, like Java, AAM adopts reference model for all of the values (all things in stores), but CFA2 has stack allocated

```
  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 10: Parts of Continuation Transition Rules for Expressions

```
def transitApplication(state: State): Set[State] = state match {
  case State(KReturnComplete(v), env, localStack, a, memory) =>
      val newMemory = memory.copy(state)
      for {
        GlobalFrame(returnPoint, oldStack, savedEnv, newGlobalAddress)
            <- newMemory.globalFrames(a)
      } yield {
        if(oldStack.isEmpty || !oldStack.head.isInstanceOf[KUseValue]) {
          newMemory.getValue(v).foreach(newMemory.putValue(returnPoint, _))
        }
        State(returnPoint, savedEnv, oldStack, newGlobalAddress, newMemory)
      }

  case State(KIfComplete(cond, t, e), env, localStack, a, memory) =>
      for {
        obj <- memory.getValue(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 11: 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)
    newMemory.getValue(xRef).foreach(newMemory.putValue(JSReference(state.e.id), _))
    Set(State(JSReference(state.e.id), env, localStack, a, newMemory))

case State(LVarRef(x), env, localStack, a, memory) =>
    val xRef = lookup(env, x)
    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(NullLit(), env, localStack, a, memory) =>
    val newMemory = memory.copy(state)
    val newAddress = alloc(state.e)
    newMemory.putValue(newAddress, JSNull)
    Set(State(newAddress, 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(KFuncCallComplete(funcRef, args), env, localStack, a, memory) =>
    val newMemory = memory.copy(state)
    for {
        f <- newMemory.getValue(funcRef)
        if isCallable(f)
        JSClosure(func@FunctionExpr(name, ps, funcBody), savedEnv) = f.asInstanceOf[JSObject].code
    } yield {
        val psAddress = ps.map(alloc(_))
        val thisAddress = biGlobalObjectRef
        var newEnvPart = ("this" -> thisAddress) :: ps.map(x => x.str).zip(psAddress)
        name match {
          case Some(x) => newEnvPart = (x.str -> alloc(x)) :: newEnvPart
          case None =>
        }
        val newEnv = savedEnv ++ Map(newEnvPart: _*)

        psAddress.zip(args).foreach[Unit]((p: (JSReference, JSValue)) =>
          newMemory.getValue(p._2).foreach(newMemory.putValue(p._1, _)))
        val nextAddress = allocStackAddress(state, funcBody, newEnv)
        newMemory.pushGlobalStack(nextAddress, GlobalFrame(alloc(state.e), localStack, env, a))
        State(funcBody, newEnv, emptyLocalStack, nextAddress, newMemory)
    }

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

values. On the other hand, we cannot always pop call stack frames after function returns because section 3.2 mentions that continuation stores may becomes graphs rather than stacks.

Introspective pushdown control flow analysis described by XXX integrates abstract garbage collection (Γ-CFA) in PDCFA to implement the stack filtering. JsCFA also adopts this strategy to improve analysis precision and make call/return match more useful. The semantics of abstract garbage collection is same with its counterparts in concrete interpreters/runtimes. We first scan current state to acquire the root set and trace from the root set to reach all the objects' fields and closures' caught variables. Every address reached in last phase (computing root set and tracing) is recorded in a "mark set", and values referred by addresses that do not appear in mark set are regarded as garbage.

However, the effect of abstract garbage collection is relatively weaken in global store because the values referred by other unexecuted paths have to stay in the store even current state cannot reach them. Therefore, XXX implemented PDCFA with abstract GC with per-state stores to achieve the whole power of abstract GC. Although AAM with per-state stores theoretically take experiential complexity, in practice its performance much better than global store without abstract GC. Consequently, We also implement JsCFA with per-state stores. Moreover, there are two techniques to optimize performance of abstract GC with per-state stores. Firstly, JsCFA just copies stores on write due to only application states may change the store, so evaluation and continuation states interpreted between two application states share one store. Secondly, abstract GC running never directly deletes values even 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.

4.3 Abstract Garbage Collection as Popping Call Stack Frames

In this section, we show how to give up execution history and apply our theory for direct-style AST.