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

Fei Peng

June 2, 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 defeats 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, and it can provide perfect call/return match for monovariant and polyvariant control flow analysis. The new method puts a *execution history environment* into continuation address with callee's function boy, so we name it *h*-CFA . The execution history environments can be regarded 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.

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.

$$e \in Exp ::= (let((x(fe)))e) \\ |e$$

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

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

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

This section reviews AAM .

2.2 A defeat of AAM

Although, abstracting abstract machines import store-allocated values and store-allocated continuations that make call/return match orthogonal from context-sensitivity, original “kalloc” 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 “kalloc” function:

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

P4F adepts

3 The Essence of Pushdown Control Flow Analysis

3.1 Program Execution History

Definitions:

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

$\widetilde{\rho} \in \widetilde{Env} \triangleq Var \rightharpoonup \widetilde{Addr}$	[environments]
$\widetilde{\sigma} \in \widetilde{Store} \triangleq \widetilde{Addr} \to \widetilde{Value}$	[srores]
$\widetilde{v} \in \widetilde{Value} \triangleq \mathcal{P}(\widetilde{Closure})$	[abstract values]
$\widetilde{clo} \in \widetilde{Closure} \triangleq Lambda \times \widetilde{Env}$	[closures]
$\widetilde{\sigma_k} \in \widetilde{KStore} \triangleq \widetilde{KAddr} \to \widetilde{Kont}$	[continuation stores]
$\widetilde{k} \in \widetilde{Kont} \triangleq \mathcal{P}(\widetilde{Frame})$	[abstract continuations]
$\widetilde{\phi} \in \widetilde{Frame} \triangleq Var \times Exp \times \widetilde{Env} \times \widetilde{History} \times \widetilde{KAddr}$	[stack frames]
$\widetilde{h} \in \widetilde{History} \triangleq Var \rightharpoonup \widetilde{Addr}$	[histories]
$\widetilde{a} \in \widetilde{Addr}\ is\ a\ finite\ set$	[value addresses]
$\widetilde{a_k} \in \widetilde{KAddr}\ is\ a\ finite\ set$	[continuation addresses]

Abstract Semantics:
kalloc:

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

calls:

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

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

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

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

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

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

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

$$\widetilde{h}' = \widetilde{h} \cup \widetilde{\rho} \cup \widetilde{\rho}'$$

returns:

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

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

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

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

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

$$\widetilde{h}' = \widetilde{h_k} \cup \widetilde{\rho}'$$

3.2 Polyvariant Continuation

In this section, we use illustrations to show the principles of h-CFA.

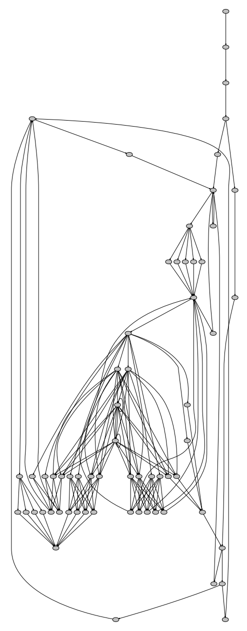
4 Static Analysis of JavaScript

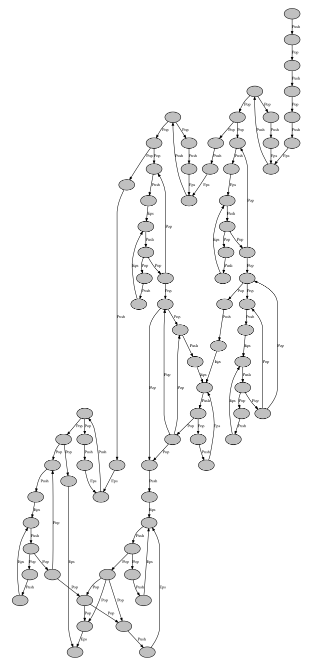
4.1 Abstract Garbage Collection as Stack Filtering

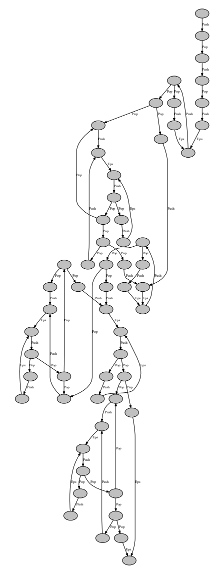
In practice, call/return perfect match working with monovariant analysis is useless.

4.2 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.


(a) P4F with 1-call-sensitive


(b) PDCFA


(c) PDCFA with Abstract Garbage Collection

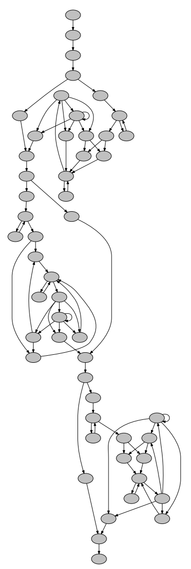

(d) h-CFA

Figure 1: (a) shows