# A Linear Encoding of Pushdown Control-Flow Analysis

Steven Lyde Thomas Gilray Matthew Might
University of Utah
{lyde,tgilray,might}@cs.utah.edu

## Abstract

We describe a linear-algebraic encoding for pushdown control-flow analysis of higher-order programs. Pushdown control-flow analyses obtain a greater precision in matching calls with returns by encoding stack-actions on the edges of a Dyck state graph. This kind of analysis requires a number of distinct transitions and was not amenable to parallelization using the approach of EigenCFA. Recent work has extended EigenCFA, making it possible to encode more complex analyses as linear-algebra for efficient implementation on SIMD architectures. We apply this approach to an encoding of a monovariant pushdown control-flow analysis and present a prototype implementation of our encoding written in Octave. Our prototype has been used to test our encoding against a traditional worklist implementation of a pushdown control-flow analysis.

***Keywords*** abstract interpretation, program analysis, flow analysis, GPU

## 1. Introduction

The goal of static analysis is to produce a bound for program behavior before run-time. This is desirable for proving the soundness of code transformations, the absence or programming errors, or the absence of malware.

However, static analysis of higher-order languages such as Scheme is nontrivial. Due to the nature of first-class functions, data-flow affects control-flow and control-flow affects data-flow, resulting in the higher-order control-flow problem. This vicious cycle has resulted in even the simplest of formulations being nearly cubic [6, 7]. However, a trade-off exists in any analysis between precision and scalability, and finding the right balance for a particular application requires special attention and effort [8].

One way to increase the scalability of an analysis is to parallelize its execution. To this end we provide a linear encoding of a pushdown control-flow analysis, giving potential speedups on many-core or SIMD architectures such as the GPU.

Prabhu et al. demonstrated the possibility of running a higher-order control flow analysis on the GPU [9]. However, their encoding has the major drawback that it only supports binary continuation-passing-style (CPS). It was restricted to a simple language which could be implemented as a single transition rule as not to introduce thread-divergence in SIMD implementations. Currying all function calls and being forced to encode all language forms and program values in the lambda calculus is not ideal for real applications because it distorts the code under analysis.

Gilray et al. addressed this issue with a demonstration that richer language forms and values can be used within this style of encoding by *partitioning* transfer functions and more precisely encoding analysis components [5]. We build on this work, demonstrating that it is not only possible to encode richer language forms, but a fundamentally richer analysis. Specifically, we demonstrate that a pushdown analysis may also be encoded using this transfer-function partitioning. A pushdown analysis has the benefit that it precisely matches function calls with function returns [10].

In this paper, we review the concrete semantics of ANF $\lambda$-calculus within a CESK machine. We then provide a direct abstraction of the pushdown-machine semantics to a monovariant pushdown control-flow analysis (0-PDCFA). We then partition the transfer function and show a linear encoding of that analysis which is faithful to its original precision.

We have also implemented an Octave prototype of our encoding. Octave allowed us to quickly implement all the matrix operations from the encoding and compare the output of this implementation with an implementation of the traditional worklist algorithm for 0-PDCFA. We were able to verify that on a range of examples their precision was identical. Our hope is that this linear encoding can be used for a GPU implementation and attain similar speedups as EigenCFA [9].

## 2. Concrete Semantics

We give semantics for a pure $\lambda$-calculus in Administrative Normal Form (ANF). ANF is a core direct-style language which strictly let-binds all intermediate-expressions [1]. This structurally enforces an order of evaluation and greatly simplifies a formal semantics. ANF is at the heart of common intermediate-representations for Scheme and other higher-order programming languages.

For simplicity we permit only call-sites, let-forms, and atomic-expressions (variables and $\lambda$-abstractions):

$$\begin{aligned} e \in \mathsf{E} &::= (\text{let } (x\ e)\ e)^l \\ &\mid (ae\ ae\ \ldots)^l \\ &\mid ae^l \\ ae \in \mathsf{AE} &::= x \mid lam \\ lam \in \mathsf{Lam} &::= (\lambda\ (x\ \ldots)\ e) \\ x \in \mathsf{Var} &::= \langle \textit{set of program variables} \rangle \\ l \in \mathsf{Label} &::= \langle \textit{set of unique labels} \rangle \end{aligned}$$

The concrete semantics for this machine will be given using a CESK machine [4], which has the following state space:

$$\begin{aligned} \varsigma \in \Sigma &= \mathsf{E} \times Env \times Store \times Time \times Kont \\ \rho \in Env &= \mathsf{Var} \rightharpoonup Addr \\ \sigma \in Store &= Addr \rightharpoonup Value \\ t \in Time &= \mathsf{Label}^* \\ \kappa \in Kont &= Frame^* \\ \phi \in Frame &= \mathsf{E} \times Env \times \mathsf{Var} \\ a \in Addr &= \mathsf{Var} \times Time \\ v \in Value &= \mathsf{Lam} \times Env \end{aligned}$$

Each state in the abstract-machine represents control at a particular expression-context $e$, with a binding environment $\rho$ encoding visible bindings of variables to addresses and a value-store (a model of the heap) mapping addresses to values. Each state is also specific to a timestamp $t$ encoding a perfect program-trace and a current continuation $\kappa$ encoding a stack of continuation frames.

The only values for this language are closures. To generate values given an atomic-expression, we will use an atomic-evaluator. Given a variable, it looks up the address of the value in the environment and then the value in the store. Given a $\lambda$-abstraction, we simply close it over the current environment.

$$\begin{aligned} \mathcal{A}\colon \mathsf{AE} \times \Sigma &\rightharpoonup Value \\ \mathcal{A}(x,\ (e,\ \rho,\ \sigma,\ t,\ \kappa)) &= \sigma(\rho(x)) \\ \mathcal{A}(lam,\ (e,\ \rho,\ \sigma,\ t,\ \kappa)) &= (lam,\ \rho) \end{aligned}$$

Looking at the grammar for our language, we can see that there are three expression forms: let bindings, applications, and atomic expressions. To fully present the semantics, we will provide a transition relation that has a rule for each form.

The first form we will describe is for let bindings. A let expression pushes a frame on the stack that captures the expression to evaluate when we return, the environment to be used, what variable we will bind, along with the stack as it exists when we push the new frame.

$$\underbrace{((\text{let } (x\ e)\ e_\kappa)^l,\ \rho,\ \sigma,\ t,\ \kappa)}_{\varsigma} \Rightarrow (e,\ \rho,\ \sigma,\ t',\ \kappa')$$

$$\begin{aligned} \textit{where} \quad \kappa' &= (e_\kappa,\ \rho,\ x) : \kappa \\ t' &= l : t \end{aligned}$$

Function calls are a little bit more involved but not too complicated. We evaluate the function we are applying, as well as all the arguments. We create new address and set the values in the store. Note that since these are tail calls the stack is unchanged.

$$\frac{((\lambda\ (x_1\ \ldots\ x_j)\ e),\ \rho_\lambda) = \mathcal{A}(ae_f,\ \varsigma)}{\underbrace{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \rho,\ \sigma,\ t,\ \kappa)}_{\varsigma} \Rightarrow (e,\ \rho',\ \sigma',\ t',\ \kappa)}$$

$$\begin{aligned} \textit{where} \quad \rho' &= \rho_\lambda[x_i \mapsto (x_i,\ t')] \\ \sigma' &= \sigma[(x_i,\ t') \mapsto \mathcal{A}(ae_i,\ \varsigma)] \\ t' &= l : t \end{aligned}$$

Finally, when we come across an atomic expression, we need to return. We do this by extracting the needed information from the top frame, extend and update the environment and return to using the previous stack.

$$\frac{\kappa = (e,\ \rho_\kappa,\ x_\kappa) : \kappa'}{\underbrace{(ae^l,\ \rho,\ \sigma,\ t,\ \kappa)}_{\varsigma} \Rightarrow (e,\ \rho',\ \sigma',\ t',\ \kappa')}$$

$$\begin{aligned} \textit{where} \quad \rho' &= \rho_\kappa[x_\kappa \mapsto (x_\kappa,\ t')] \\ \sigma' &= \sigma[(x_\kappa,\ t') \mapsto \mathcal{A}(ae,\ \varsigma)] \\ t' &= l : t \end{aligned}$$

These semantics may be used to evaluate a program $e$ by producing an initial state $\varsigma_0 = (e, \emptyset, \bot, (), ())$ and computing the transitive closure of $(\Rightarrow)$ from this state. Naturally, concrete executions may take an unbounded amount of time to compute in the general case. This manifests itself in the above semantices as an unbounded set of timestamps leading to an unbounded address-space, and as an unbounded stack used to represent the current continuation.

## 3. Abstract Semantics

We will now provide the abstract semantics of the analysis. Because our analysis is monovariant and only maintains one approximation for each variable, there is only one environment for a given expression-context. Thus it is elided from

the state space. The stack is now the only source of unboundedness in these semantics:

$$\begin{aligned}
\hat{\varsigma} \in \widehat{\Sigma} &= \mathsf{E} \times \widehat{Store} \times \widehat{Kont} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Var} \to \widehat{Values} \\
\hat{\kappa} \in \widehat{Kont} &= \widehat{Frame}^* \\
\hat{\phi} \in \widehat{Frame} &= \mathsf{E} \times \mathsf{Var} \\
\hat{v} \in \widehat{Values} &= \mathcal{P}(\widehat{Value}) \\
\hat{d} \in \widehat{Value} &= \mathsf{Lam}
\end{aligned}$$

In providing the abstract semantics, we will once again need a way to evaluate atomic expressions. The atomic evaluator is very similar to its concrete counterpart. However, since there is only one environment, we look up the value of a variable using it directly. Also, we don't need to close lambdas over an environment as their expression-body is already specific to a particular monovariant environment.

$$\begin{aligned}
\hat{\mathcal{A}} &\colon \mathsf{AE} \times \widehat{\Sigma} \rightharpoonup \widehat{Values} \\
\hat{\mathcal{A}}(x,\ (e,\ \hat{\sigma},\ \hat{\kappa})) &= \hat{\sigma}(x) \\
\hat{\mathcal{A}}(lam,\ (e,\ \hat{\sigma},\ \hat{\kappa})) &= \{lam\}
\end{aligned}$$

The abstract transition relation is also very similar to its concrete counterpart. Note that the frames no longer store environments.

$$\underbrace{((\text{let } (x\ e)\ e_\kappa)^l,\ \hat{\sigma},\ \hat{\kappa})}_{\hat{\varsigma}} \twoheadrightarrow (e,\ \hat{\sigma},\ \hat{\kappa}')$$

$$where \quad \hat{\kappa}' = (e_\kappa,\ x) : \hat{\kappa}$$

Also note that when updating the store we use the least-upper-bound to remain sound. This permits values to merge within flow-sets: $(\sigma_1 \sqcup \sigma_2)(\hat{a}) = \sigma_1(\hat{a}) \cup \sigma_2(\hat{a})$.

$$\frac{(\lambda\ (x_1\ \ldots\ x_j)\ e) \in \hat{\mathcal{A}}(ae_f,\ \hat{\varsigma})}{\underbrace{((ae_f\ ae_1\ \ldots\ ae_j),\ \hat{\sigma},\ \hat{\kappa})}_{\hat{\varsigma}} \twoheadrightarrow (e,\ \hat{\sigma}',\ \hat{\kappa})}$$

$$where \quad \hat{\sigma}' = \hat{\sigma} \sqcup [x_i \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma})]$$

Finally, when we return, we update the variable found in a stack-frame.

$$\frac{\hat{\kappa} = (e,\ x) : \hat{\kappa}'}{\underbrace{(ae,\ \hat{\sigma},\ \hat{\kappa})}_{\hat{\varsigma}} \twoheadrightarrow (e,\ \hat{\sigma}',\ \hat{\kappa}')}$$

$$where \quad \hat{\sigma}' = \hat{\sigma} \sqcup [x \mapsto \hat{\mathcal{A}}(ae,\ \hat{\varsigma})]$$

Simply enumerating all the states possible given this abstract transition relation is not guaranteed to terminate. However, there is a finite representation of the infinite state space of the stacks. If we use this transition relation to generate a Dyck state graph, our analysis will terminate. This is accomplished by taking the infinite stacks and encoding them into a finite graph, where the stack frames are labels on edges of that graph. Intuitively, we are making the explicit result of cycles in control-flow (unbounded stacks) implicit as cycles in a control-flow graph.

A Dyck state graph is a set of edges.

$$G \in \mathcal{P}(Q \times \Gamma \times Q)$$

The nodes in the graph $Q$ are the parts of an abstract state $\hat{\varsigma} \in \hat{\Sigma}$ sans the stack $\hat{\kappa} \in \widehat{Kont}$.

$$q \in Q = \mathsf{E} \times \widehat{Store}$$

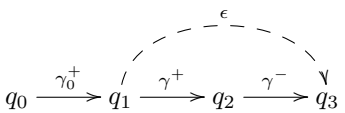
The edges describe transition between nodes and contain the stack-action that exists between these nodes. There are three different stack actions: pushing a frame $\hat{\phi}^+$, leaving the stack unchanged $\epsilon$, and popping a frame $\hat{\phi}^-$.

$$\gamma \in \Gamma = \hat{\phi}^+ \mid \epsilon \mid \hat{\phi}^-$$

Whether an edge exists in the graph can be taken directly from the abstract transition relation. We introduce the relation $(\xrightarrow{\gamma}) \subseteq Q \times \Gamma \times Q$ for edges in the Dyck state graph, defined in terms of the abstract transition relation.

$$\begin{aligned}
q \xrightarrow{\hat{\phi}^+} q' &\iff (q,\ \hat{\kappa}) \twoheadrightarrow (q',\ \hat{\phi} : \hat{\kappa}) \\
q \xrightarrow{\epsilon} q' &\iff (q,\ \hat{\kappa}) \twoheadrightarrow (q',\ \kappa) \\
q \xrightarrow{\hat{\phi}^-} q' &\iff (q,\ \hat{\phi} : \hat{\kappa}) \twoheadrightarrow (q',\ \hat{\kappa})
\end{aligned}$$

To efficiently compute the Dyck state graph, an epsilon closure graph is needed. An epsilon closure graph has edges between all nodes that have no net stack change between them. For instance, if we push a frame and then pop a frame, there should be an epsilon edge between the source node of the push edge and the target node of the pop edge. This is the epsilon edge between $q_1$ and $q_3$ below.




This allows us to immediately see that $\gamma_0$ is a possible top frame for $q_3$ when generating successor edges and nodes for $q_3$.

### 3.1 Transfer Function

When computing the analysis, we use a transfer function $\hat{f} : (Q \times \Gamma \times Q) \to (Q \times \Gamma \times Q)$ that takes a Dyck state

graph and computes new edges at the frontier of the graph, generating a new Dyck state graph. We continually apply this transfer function until a fix-point is reached.

$$\hat{f}(G) = G \cup \left\{(q, \gamma, q') : q \in Q,\ q \xrightarrow{\gamma} q'\right\}, \text{where}$$
$$Q = \{q' : (q, \gamma, q') \in G\} \cup \{q_0\}$$

### 3.2 Global Store Widening

In the given abstract semantics, each state had its own store. However, to ensure the analysis will converge more quickly, global store-widening is usually employed. This form of widening is equivalent to using a global-store for all states which is computed as the least-upper-bound of all stores visited at any individual state. To accomplish this we will remove the store from the nodes of the Dyck state graph and define the store-widened Dyck state graph as follows:

$$G_\triangledown \in \mathcal{P}(\mathsf{E} \times \Gamma \times \mathsf{E})$$

The globally store-widened transfer function then individually computes a new graph of expressions and stack actions, and a new global store.

$$\hat{f}_\triangledown(G_\triangledown, \hat{\sigma}) = (G'_\triangledown, \hat{\sigma}'), \text{where}$$
$$G'_\triangledown = G_\triangledown \cup \left\{(e, \gamma, e') : e \in Q_e,\ (e, \hat{\sigma}) \xrightarrow{\gamma} (e', \_)\right\}$$
$$\hat{\sigma}' = \bigsqcup \left\{\hat{\sigma}' : e \in Q_e,\ (e, \hat{\sigma}) \xrightarrow{\gamma} (\_, \hat{\sigma}')\right\}$$
$$Q_e = \{e : (\_, \_, e) \in G_\triangledown\} \cup \{e_0\}$$

An underscore represents a wildcard, i.e. any value.

### 3.3 Partitioning the Transfer Function

We can partition this monolithic transfer function, defining an individualized transfer function for each expression form in our language: $\hat{f}_{let}$, $\hat{f}_{call_i}$ and $\hat{f}_{ae}$. These function are defined in precisely the same manner, but only use the rule applying to their specific language form. After each iteration, we merge the resulting Dyck state graphs and stores, taking their least-upper-bound. It has been shown that partitioning a system-space transfer function by rule in this manner is sound as the least-upper-bound of the system-spaces resulting from an application of each, is always equal to the system-space resulting from a single application of the combined $\hat{f}_\triangledown$ [5].

## 4. Linear Encoding

We will construct a transfer function for each abstract transition relation. This transfer function will update the store and will also be responsible for creating a Dyck state graph. We will define these functions using matrix multiplication ($\times$), outer product ($\otimes$), and boolean or ($+$). The style of encoding we use is taken directly from the original approach of EigenCFA [9].

The abstract state space, because it is finite, is easy to represent in vector and matrix form. If the elements in the domain are given a canonical order, we can represent a set of those elements using a bit vector. If an element from the domain is present in the set, the vector representing that set should have its bit set at the index corresponding to the offset of that element in the ordering. In our encoding we will represent the set of states using a vector $\vec{s} \in \vec{S}$. We will represent atomic expressions, either variables or values, with $\vec{a} \in \vec{A}$. And we will use $\vec{v} \in \vec{V}$ to represent flow sets of abstract values.

$$\vec{s} \in \vec{S} = \{0,1\}^{|\mathsf{E}|}$$
$$\vec{a} \in \vec{A} = \{0,1\}^{|\widehat{Var}|+|\widehat{Value}|}$$
$$\vec{v} \in \vec{V} = \{0,1\}^{|\widehat{Value}|}$$

We can also encode the abstract syntax tree as matrices. We can extract the body of a closure using $\mathbf{Body}$ or the variables it binds using $\mathbf{Var_i}$. We can also deconstruct the components of a let expression using $\mathbf{Arg_1}$, $\mathbf{LetCall}$ and $\mathbf{LetBody}$.

$$\mathbf{Body}\colon \vec{V} \to \vec{S} \qquad \mathbf{Arg_i}\colon \vec{S} \to \vec{A}$$
$$\mathbf{Fun}\colon \vec{S} \to \vec{A} \qquad \mathbf{LetCall}\colon \vec{S} \to \vec{S}$$
$$\mathbf{Var_i}\colon \vec{V} \to \vec{A} \qquad \mathbf{LetBody}\colon \vec{S} \to \vec{S}$$

The store is a matrix that maps atomic expressions to abstract values.

$$\boldsymbol{\sigma}\colon \vec{A} \to \vec{V}$$

We also represent the Dyck state graph using three matrices. These three matrices map states to states, which in the case of our linear encoding, are expressions in our program. We use three different matrices to represent the three types of edges that can be found in the Dyck state graph.

$$\boldsymbol{\gamma_+}\colon \vec{S} \to \vec{S}$$
$$\boldsymbol{\gamma_\epsilon}\colon \vec{S} \to \vec{S}$$
$$\boldsymbol{\gamma_-}\colon \vec{S} \to \vec{S}$$

We also use a matrix to represent the epsilon closure graph which aids in the construction of the matrices encoding the Dyck state graph.

$$\boldsymbol{\epsilon}\colon \vec{S} \to \vec{S}$$

We now define the transfer function for the three types of expressions our language supports, let bindings, applications and atomic expressions.

For let expressions, we first extract the sub-expression whose value will be bound to the variable of the let expression, $\vec{s}_{let} \times \mathbf{LetCall}$. We then record the push edge in the Dyck state graph, $\boldsymbol{\gamma_+} + (\vec{s}_{let} \otimes \vec{s}_{next})$.

$$f_{\vec{s}_{let}}(\boldsymbol{\gamma_+}) = (\boldsymbol{\gamma_+}')$$
$$where \quad \vec{s}_{next} = \vec{s}_{let} \times \mathbf{LetCall}$$
$$\boldsymbol{\gamma_+}' = \boldsymbol{\gamma_+} + (\vec{s}_{let} \otimes \vec{s}_{next})$$

Applications are somewhat more involved. We first pull out of the store the abstract values that we are applying for the given call site. We then extract the values of the arguments. We then get variables that we are binding from the closures we are applying. We then record the updated values in the store. We must also record that we made a tail-call in the Dyck state graph. We do this by updating $\boldsymbol{\gamma_\epsilon}$. We then must also update any epsilon edges.

$$
\begin{aligned}
f_{\vec{s}_{call_j}}(\boldsymbol{\sigma}, \boldsymbol{\gamma_\epsilon}, \boldsymbol{\epsilon}) &= (\boldsymbol{\sigma}', \boldsymbol{\gamma_\epsilon}', \boldsymbol{\epsilon}') \\
\textit{where} \quad & \vec{v}_f = \vec{s}_{call_j} \times \mathbf{Fun} \times \boldsymbol{\sigma} \\
& \vec{v}_i = \vec{s}_{call_j} \times \mathbf{Arg_i} \times \boldsymbol{\sigma} \\
& \vec{a}_i = \vec{v}_f \times \mathbf{Var_i} \\
& \boldsymbol{\sigma}' = \boldsymbol{\sigma} + (\vec{a}_1 \otimes \vec{v}_1) + \ldots + (\vec{a}_j \otimes \vec{v}_j) \\
& \vec{s}_{next} = \vec{v}_f \times \mathbf{Body} \\
& \boldsymbol{\gamma_\epsilon}' = \boldsymbol{\gamma_\epsilon} + (\vec{s}_{call_j} \otimes \vec{s}_{next}) \\
& \boldsymbol{\epsilon}' = f_\epsilon(\boldsymbol{\epsilon},\ \vec{s}_{call_j},\ \vec{s}_{next})
\end{aligned}
$$

Finally, we come to the last case where we have an atomic expression and must return. We first must compute the flow set of the atomic expression. We then look up the top frames of our stack. We then update the environment by binding the variable found at the top stack frame. We also extract the expression that we will be executing next. Finally, we record the pop edge and update the epsilon closure graph accordingly.

$$
\begin{aligned}
f_{\vec{s}_{æ}}(\boldsymbol{\sigma},\ \boldsymbol{\gamma_+},\ \boldsymbol{\gamma_-},\ \boldsymbol{\epsilon}) &= (\boldsymbol{\sigma}',\ \boldsymbol{\gamma_+}',\ \boldsymbol{\gamma_-}',\ \boldsymbol{\epsilon}') \\
\textit{where} \quad & \vec{v} = \vec{s}_{æ} \times \mathbf{Arg_1} \times \boldsymbol{\sigma} \\
& \vec{s}_{push} = \vec{s}_{æ} \times \boldsymbol{\epsilon}^\top \times \boldsymbol{\gamma_+}^\top \\
& \vec{a} = \vec{s}_{push} \times \mathbf{Arg_1} \\
& \boldsymbol{\sigma}' = \boldsymbol{\sigma} + (\vec{a} \otimes \vec{v}) \\
& \vec{s}_{next} = \vec{s}_{push} \times \mathbf{LetBody} \\
& \boldsymbol{\gamma_-}' = \boldsymbol{\gamma_-} + (\vec{s}_{æ} \otimes \vec{s}_{next}) \\
& \boldsymbol{\epsilon}' = f_\epsilon(\boldsymbol{\epsilon},\ \vec{s}_{æ},\ \vec{s}_{push})
\end{aligned}
$$

The epsilon closure graph aids in the construction of the Dyck state graph. It contains edges between states that have no net stack change. This allows us to quickly find the top frames when we need to return. When updating the epsilon closure graph, we not only need to record the new edges, but take all existing predecessors and successors into account.

$$
\begin{aligned}
f_\epsilon(\boldsymbol{\epsilon},\ \vec{s}_s,\ \vec{s}_t) &= \boldsymbol{\epsilon}' \\
\textit{where} \quad & \vec{s}_n = \vec{s}_t \times \boldsymbol{\epsilon} \\
& \vec{s}_p = \vec{s}_s \times \boldsymbol{\epsilon}^\top \\
& \boldsymbol{\epsilon}' = \boldsymbol{\epsilon} + (\vec{s}_s \otimes \vec{s}_t) \\
& \qquad + (\vec{s}_s \otimes \vec{s}_n) \\
& \qquad + (\vec{s}_p \otimes \vec{s}_t) \\
& \qquad + (\vec{s}_p \otimes \vec{s}_n)
\end{aligned}
$$

## 5. Example

To help give a better understanding of how the encoding works, we provide a short example.

```
(let (id^x0 (lambda (v^x1) v^l2)^l1 d̂0)
  (id (lambda (w^x2)
        (let (a^x3 (w id)^l5)
          (a a)^l6)^l4)^d̂1)^l3)^l0
```

For this program there are only two denotable values, the two lambda terms. There are two let expressions, three call sites, and one atomic reference as the body of a lambda. There are also four variables in this program.

We will first discuss how you would encode the abstract syntax tree using matrices. Recall that there are six matrices that are needed.

First, given a flow set, we want to be able to extract which expressions are the body of a lambda term. Below we can see that $l_2$ is the body of the first lambda and $l_4$ is the body of the second lambda.

$$
\mathbf{Body} = \begin{array}{c} \\ d_0 \\ d_1 \end{array}
\begin{array}{c}
\begin{array}{ccccccc} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{array} \\
\left[\begin{array}{ccccccc}
0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}\right]
\end{array}
$$

We also need a way to extract the function being applied at a call site, whether it be a lambda term or a variable reference. Because there are only three call sites in the program, only three rows in the matrix have entries with non-zero values. In our example, every call site has a variable reference in function position.

$$
\mathbf{Fun} = \begin{array}{c}
\begin{array}{ccccccc} x_0 & x_1 & x_2 & x_3 & \hat{d}_0 & \hat{d}_1 \end{array} \\
\left[\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0
\end{array}\right]
\end{array}
\begin{array}{c} l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array}
$$

There must also be a way to extract the arguments of a call site. This matrix can also be used to determine what atomic expression we are evaluating when our control state is at an atomic expression.

$$
\mathbf{Arg_1} = \begin{array}{c}
\begin{array}{ccccccc} x_0 & x_1 & x_2 & x_3 & \hat{d}_0 & \hat{d}_1 \end{array} \\
\left[\begin{array}{cccccc}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0
\end{array}\right]
\end{array}
\begin{array}{c} l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array}
$$

Once we have a flow set, we want to be able to extract the variable that we are binding when we apply the functions in

our flow set.

$$\mathbf{Var_1} = \begin{array}{c} \\ \hat{d}_0 \\ \hat{d}_1 \end{array} \begin{array}{c} \begin{array}{cccccc} x_0 & x_1 & x_2 & x_3 & \hat{d}_0 & \hat{d}_1 \end{array} \\ \left[\begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right] \end{array}$$

We also need to be able to know what the expression is whose value we will bind to a variable when we have a let expression. This lets us know what our successor state will be. This is used when we push a frame onto our stack.

$$\mathbf{LetCall} = \begin{array}{c} \\ l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{array} \\ \left[\begin{array}{ccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

Given a let expression, we also need to know where we should return to once we have evaluated the expression which will provide the value we are binding.

$$\mathbf{LetBody} = \begin{array}{c} \\ l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{array} \\ \left[\begin{array}{ccccccc} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

The store for this program is actually rather small. We are interested in finding out which lambda terms flow to which variables. With four variables and two lambda terms there are only eight entries that can be set. Note that we have an identity matrix at the bottom of the store.

$$\boldsymbol{\sigma} = \begin{array}{c} \\ x_0 \\ x_1 \\ x_2 \\ x_3 \\ \\ \hat{d}_0 \\ \hat{d}_1 \end{array} \begin{array}{c} \begin{array}{cc} \hat{d}_0 & \hat{d}_1 \end{array} \\ \left[\begin{array}{cc} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \hline 1 & 0 \\ 0 & 1 \end{array}\right] \end{array}$$

To encode a Dyck state graph we actually need three separate matrices. A value of one represents that there exists an edge between two states. The contents of the frame (the variable to bind and the expression to execute next) are both available using $\mathbf{Arg_1}$ and $\mathbf{LetBody}$. After running the analysis on the above program, the results of the three matrices would be as follows.

$$\boldsymbol{\gamma}_+ = \begin{array}{c} \\ l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{array} \\ \left[\begin{array}{ccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

$$\boldsymbol{\gamma}_- = \begin{array}{c} \\ l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{array} \\ \left[\begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

$$\boldsymbol{\gamma}_{\boldsymbol{\epsilon}} = \begin{array}{c} \\ l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{array} \\ \left[\begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

We also need the epsilon closure graph. Initially it is an identity matrix because every state has an implicit epsilon edge to itself.

$$\boldsymbol{\epsilon} = \begin{array}{c} \\ l_0 \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{ccccccc} l_0 & l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{array} \\ \left[\begin{array}{ccccccc} 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

## 6. Prototype Implementation

To help verify our encodings we produced a prototype implementation in Octave. Octave allowed us to quickly implement the encoding as a sanity check, without having to worry about all the intricacies of coding for the GPU. Octave is a programming language for numerical analysis and as such has strong support for various matrix operations.

We wrote a Scheme front end that would parse the programs and write the abstract syntax tree in matrix form to be consumed by our Octave implementation. We also wrote utility code that would consume the output of our Octave implementation and produce output in a more human consumable format. This allowed us to easily view the store and Dyck state graph generated from the analysis.

We ran our prototype implementation on a small suite of simple benchmarks. We then compared the results of our

Octave implementation to the output of a traditional work list based pushdown control-flow analysis implementation. We took the implementation of Sergey [3] and modified it to use a single store so it would perform the same analysis as the analysis using our linear encoding. The output from both implementations were identical.

## 7. Conclusion

We have described a linear encoding for a pushdown control-flow analysis as originally formulated by Earl et al. [3] building upon the general framework of abstract interpretation [2]. By precisely matching calls and returns a pushdown control-flow analysis gives even more precision than a traditional finite state control-flow analysis. By demonstrating the feasibility of a linear encoding, we have demonstrated that it is at least possible to run a pushdown control-flow analysis on a SIMD architecture. Though a direct translation would likely be inefficient as the matrices are very sparse. Novel techniques such as those used in EigenCFA would need to be employed [9]. In the future we hope to demonstrate that this encoding is not only feasible, but practical and useful as well.

## Acknowledgments

This material is partially based on research sponsored by DARPA under agreement number FA8750-12-2-0106 and by NSF under CAREER grant 1350344. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon.

## References

[1] C. Cormac Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 237–247, 1993.

[2] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, TX, 1979. ACM Press, New York.

[3] C. Earl, I. Sergey, M. Might, and D. V. Horn. Introspective pushdown analysis of higher-order programs. In *Proceedings of the International Conference on Functional Programming*, pages 177–188, September 2012.

[4] M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *Proceedings of the Symposium on Principles of Programming Languages*, page 314, New York, NY, 1987. ACM.

[5] T. Gilray, J. King, and M. Might. Partitioning 0-cfa for the gpu. In *Proceedings of the 23rd International Workshop on Functional and (Constraint) Logic Programming*, 2014.

[6] D. V. Horn. *The Complexity of Flow Analysis in Higher-Order Languages*. PhD thesis, Mitchom School of Computer Science, Brandeis University, Boston, MA, August 2009.

[7] D. V. Horn and H. G. Mairson. Deciding k-CFA is complete for EXPTIME. *ACM Sigplan Notices*, 43(9):275–282, 2008.

[8] M. Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 2007.

[9] T. Prabhu, S. Ramalingam, M. Might, and M. Hall. EigenCFA: Accelerating flow analysis with GPUs. In *Proceedings of the Symposium on the Principals of Programming Languages*, pages 511–522, January 2010.

[10] D. Vardoulakis and O. Shivers. CFA2: a context-free approach to control-flow analysis. In *European Symposium on Programming*, pages 570–589, 2010.