A Unified Approach to Polyvariance in Abstract Interpretations

Thomas Gilray
University of Utah
tgilray@cs.utah.edu

Matthew Might
University of Utah
might@cs.utah.edu

Abstract

We describe an approach to exploring polyvariance in abstract interpretations by exposing the allocation of abstract bindings as an analysis parameter. This allocation policy is a method for selecting abstract addresses based on the current state of execution. As addresses are chosen from a finite set, the allocation policy is responsible for determining the exact degree of merging which occurs between different values at a given point in the analysis. This approach allows us to select any kind of polyvariance desired through the selection of an abstract allocation function. We show how this can be done for an intermediate representation of Scheme, implementing a sound parametric framework for such analyses. We distinguish three disparate interpretations of the k-CFA hierarchy and compare them, instantiating each within our framework and motivating our approach.

Keywords Abstract Interpretation, Allocation, Polyvariance

1. Introduction

The goal of static analysis is to make guarantees about the behavior of a program based only on its source text. Abstract interpretation is a general framework for static analysis which draws a formal correspondance between a language's concrete semantics and an abstract semantics. This correspondance ensures the latter is a computable approximation of the former. Analyses can be made which are sound and time-bounded by following a standard process for abstracting a given concrete semantics [5] [6] [16]. Abstracting Abstract Machines by Van Horn and Might presents this process especially well and would serve as a good introduction to the change in our semantics between Section 2 and Section 3 [30].

An abstract interpretation produces an upper bound for the behavior of its target program by conservatively tracking all possible values that flow to any particular variable. These flow-sets propagate through a bounded set of abstract machine addresses until a fixed-point is reached over the abstract transition relation. Polyvariant analyses are those which allocate multiple abstract addresses for each syntactic point of interest, differentiating their possible values based on a previously selected notion of program context. Take for example the following snippet of Scheme code:

```
(let ((f (lambda (x) (g x))))
     (f 5)
     (f #t))
```

The polymorphic function f is called twice with very different inputs. In a monovariant analysis, a single address would be used to store all possible values bound to x in all contexts. A polyvariant call-sensitive analysis on the other hand, would use a limited amount of call-history to differentiate multiple addresses for x and thus multiple flow-sets. In this case we produce an address "x after callsite $(f\ 5)$" and an address "x after callsite $(f\ \#t)$" which improves the precision of our results at the cost of juggling these additional individualized flow-sets.

Recent work by Might and Manolios provides a proof that any conceivable function producing abstract addresses for a given machine state (i.e. analysis context) can be justified *a posteriori* as having a sound mapping back to concrete addresses [21]. As a result, any notion of context, reasonable or unreasonable, can be used to structure the polyvariance of an abstract interpretation. This insight motivated our work to factor out the allocation policy from the rest of a working analysis framework. We then show how trivial it becomes to parameterize our framework with an allocation policy and tune its polyvariance arbitrarily with minimal modifications to our code.

1.1 Three flavors of call-sensitivity

In Olin Shivers' seminal paper on control-flow analysis of Scheme, he presents a hierarchy of increasingly polyvariant analyses called k-CFA. Shivers' hierarchy distinguishes all abstract bindings by a string of k call-sites called a contour, a timestamp, a context, or a history. These are described as being the last k call-sites which execution has passed through. As a complication, since the analysis was originally presented for CPS-converted Scheme, all return-points have been reified as syntactic lambdas and are thus analyzed as call-sites. This means the set of program points which might be included in an abstract timestamp depends on whether you follow the original description or implementation of k-CFA [25] [26] [17].

An analysis sensitive to calling context could be implemented as using the last k call-sites (as k-CFA was originally described), using the last k call-sites or return-points (as originally implemented), or using the top k stack frames. We distinguish these three kinds of call-sensitivity and, in doing so, motivate our approach to a unified framework for polyvariance by instantiating our analysis to implement each of the three.

1.2 Contributions

We introduce a simple higher-order language and its concrete semantics. We perform a minimal abstraction of these semantics to obtain a computable approximation. By taking care to define our analysis mechanics in terms of an arbitrary allocation function, we produce a framework for this language where different kinds of polyvariance, in all their subtlety, can be expressed as a simple configuration of the allocation policy. We then precisely distinguish the three flavors of call-sensitivity described and give a number of polyvariant allocation functions corresponding to novel or previously known static analyses.

2. Concrete Semantics

For simplicity, we target the λ-calculus in Administrative Normal Form extended with primitive operations and set!. As this is a good intermediate representation for compilers, our implementation naturally supports partially compiled Scheme benchmarks. This choice makes our analyses immediately applicable to a variety of real languages.

ANF centers expression-nesting around the let-form so that the order of operations becomes explicit. Where otherwise two expressions might be arbitrarily nested one within the other, in ANF this is only achieved by let-binding the result of the intermediate expression to a variable. Because all arguments to call-sites and primitive operations must first be let-bound, they can be evaluated atomically, reducing the complexity of our semantics.

We develop a concrete small-step interpreter based on the CESK machine of Felleisen and Friedman [8]. There are two kinds of machine states in our concrete state-space (defined in Figure 2). Most states contain a control-expression e, binding environment ρ, value-store σ, current continuation κ, and a timestamp t.

$$i.e.\ (e,\ \rho,\ \sigma,\ \kappa,\ t)$$

The binding environment maps variables to addresses, while the store maps addresses to values. The current continuation contains a variable which receives the return-value of the control-expression once evaluated, a binding environment to use once a value is returned, a control-expression to transition to, and a parent continuation for this new expression. The current timestamp is incremented between bindings so multiple bindings of the same variable are unique.

A special kind of state is differentiated from the normal "Eval" states. These "Apply" states are produced when a function is invoked. It succeeds an eval state with a call-site as its control-expression. In place of a call-site, these states contain a closure to apply, a list of values to bind in its environment, and a label for the generating call-site. Apply states are themselves always succeeded by an eval state for the body of the invoked closure. They are factored out to ensure the generality of the framework.

$$\begin{aligned}
e \in \mathsf{E}^l \ &::= (\text{let}\ (x\ e)\ e) \\
&\mid (ae\ ae\ \ldots) \\
&\mid (\text{if}\ ae\ e\ e) \\
&\mid (\text{prim}\ op\ ae\ \ldots) \\
&\mid (\text{set!}\ x\ ae) \\
&\mid ae \\
ae \in \mathsf{AE}^l \ &::= lam \\
&\mid x \\
&\mid c \\
lam \in \mathsf{Lam} \ &::= (\lambda\ (x\ \ldots)\ e) \\
c \in \mathsf{Const} \ &::= \textit{set of program constants} \\
x \in \mathsf{Var} \ &::= \textit{set of program variables} \\
op \in \mathsf{OP} \ &::= \textit{set of primitive operations} \\
l \in \mathsf{Label} \ &::= \textit{set of unique labels}
\end{aligned}$$

Figure 1. Our target language: An ANF Scheme

Before introducing our concrete transition relation, there are two auxilliary helper functions which its rules depend upon: an atomic-expression evaluator and a primitive-operation evaluator.

$\mathcal{A}$ transforms an atomic-expression, in the context of a state, into a value.

$$\mathcal{A}\colon \mathsf{AE} \times State \rightharpoonup Value$$

$$\begin{aligned}
\mathcal{A}(\text{false},\ (e,\ \rho,\ \sigma,\ \kappa,\ t)) &= \mathsf{FALSE} \\
\mathcal{A}(\text{void},\ (e,\ \rho,\ \sigma,\ \kappa,\ t)) &= \mathsf{VOID} \\
&\ldots \\
\mathcal{A}(x,\ (e,\ \rho,\ \sigma,\ \kappa,\ t)) &= \sigma(\rho(x)) \\
\mathcal{A}(lam,\ (e,\ \rho,\ \sigma,\ \kappa,\ t)) &= \langle lam, \rho \rangle
\end{aligned}$$

The primitive-operation evaluator takes the name of an operation and a list of values. It returns the result of a primitive operation or externally defined function.

$$\delta\colon \mathsf{OP} \times Value^{*} \rightharpoonup Value$$

For example $\delta(+,\ (2\ 3))$ would return 5 and $\delta(\text{append},\ ((1\ 2)\ (3)))$ would return $(1\ 2\ 3)$.

$$
\begin{aligned}
\varsigma \in \mathit{State} &= \mathit{Eval} + \mathit{Apply} \\
\mathit{Eval} &= \mathsf{E} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont} \times \mathit{Time} \\
\mathit{Apply} &= \mathit{Value} \times \mathit{Value}^* \times \mathit{Label} \\
&\qquad \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont} \times \mathit{Time} \\
\rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Addr} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Value} \\
\kappa \in \mathit{Kont} &= (\text{kont}\ x\ \rho\ e\ \kappa) \mid (\text{halt}) \\
t \in \mathit{Time} &= \mathbb{N} \\
a \in \mathit{Addr} &= \mathsf{Var} \times \mathit{Time} \\
v \in \mathit{Value} &= \mathsf{Lam} \times \mathit{Env} + \{\mathsf{FALSE},\ \mathsf{VOID},\ \ldots\}
\end{aligned}
$$

Figure 2. Concrete state-space

We proceed by defining a small-step operational semantics; specifically, a transition relation ($\Rightarrow$) which maps each possible machine state to a single successor, or none if computation has ended.

Let A new continuation is pushed onto the stack and the old continuation is nested inside it. When a prim-op, set! or atomic-expression is eventually encountered in e_1, the stack is popped and its value is returned to the variable x placed inside the continuation. Execution will then continue at e_2 within the saved environment extended with a binding for x.

$$((\text{let}\ (x\ e_1)\ e_2),\ \rho,\ \sigma,\ \kappa,\ t) \Rightarrow (e_1,\ \rho,\ \sigma,\ \kappa',\ t)$$

$$\text{where}\quad \kappa' = (\text{kont}\ x\ \rho\ e_2\ \kappa)$$

Call: Eval → Apply Invocation has been broken into two distinct kinds of states, Eval and Apply. This first half of the call-site transition evaluates the function to be called and the arguments to be sent. These values are placed inside a special Apply state, which then completes the process and produces a new Eval state. The Apply state's label is included and will be used once we abstract.

$$\underbrace{((ae_f\ ae_1 \ldots)^l,\ \rho,\ \sigma,\ \kappa,\ t)}_{\varsigma} \Rightarrow (v_f,\ (v_1 \ldots),\ l,\ \rho,\ \sigma,\ \kappa,\ t)$$

$$
\begin{aligned}
\text{where}\quad v_f &= \mathcal{A}(ae_f,\ \varsigma) \\
v_i &= \mathcal{A}(ae_i,\ \varsigma)
\end{aligned}
$$

Call: Apply → Eval With a specific closure to invoke, and the argument values evaluated, the call can be completed by transitioning to the lambda's body and performing the required argument bindings. An incremented timestamp is created for these bindings so each new binding is unique to each new call.

$$(v_f,\ (v_1 \ldots v_j),\ l,\ \rho,\ \sigma,\ \kappa,\ t) \Rightarrow (e,\ \rho',\ \sigma',\ \kappa,\ t')$$

$$
\begin{aligned}
\text{where}\quad v_f &= \langle(\lambda\ (x_1\ \ldots\ x_j)\ e),\ \rho_\lambda\rangle \\
\rho' &= \rho_\lambda[x_i \mapsto a_i] \\
\sigma' &= \sigma[a_i \mapsto v_i] \\
a_i &= (x_i,\ t') \\
t' &= t + 1
\end{aligned}
$$

If Control flows to one of two sub-expressions based on the value of the conditional expression.

$$\frac{v = \mathcal{A}(ae,\ \varsigma) \qquad v \neq \mathsf{FALSE}}{\underbrace{((\text{if}\ ae\ e_t\ e_f),\ \rho,\ \sigma,\ \kappa,\ t)}_{\varsigma} \Rightarrow (e_t,\ \rho,\ \sigma,\ \kappa,\ t)}$$

$$\frac{v = \mathcal{A}(ae,\ \varsigma) \qquad v = \mathsf{FALSE}}{\underbrace{((\text{if}\ ae\ e_t\ e_f),\ \rho,\ \sigma,\ \kappa,\ t)}_{\varsigma} \Rightarrow (e_f,\ \rho,\ \sigma,\ \kappa,\ t)}$$

AE These final three cases (ae, prim op, and set!) contain no sub-expressions. Each pops the stack and returns a value to the continuation's variable x_κ. In this case: the atomic-expression is evaluated, its value placed in the store at an address bound in the continuation's binding environment ρ_κ. A new timestamp is produced as it was for lambda-bindings and a transition is made to the continuation-expression e_κ within the nested continuation κ_κ.

$$\overbrace{(ae,\ \rho,\ \sigma,\ \kappa,\ t)}^{\varsigma} \Rightarrow (e_\kappa,\ \rho',\ \sigma',\ \kappa_\kappa,\ t')$$

$$
\begin{aligned}
\text{where}\quad \kappa &= (\text{kont}\ x_\kappa\ \rho_\kappa\ e_\kappa\ \kappa_\kappa) \\
\rho' &= \rho_\kappa[x_\kappa \mapsto a_x] \\
\sigma' &= \sigma[a_x \mapsto \mathcal{A}(ae,\ \varsigma)] \\
a_x &= (x_\kappa,\ t') \\
t' &= t + 1
\end{aligned}
$$

Prim A return-value is obtained by evaluating the arguments to the primitive-operation and passing them to δ.

$$\overbrace{((\text{prim}\ \mathit{op}\ ae_1\ \ldots\ ae_j),\ \rho,\ \sigma,\ \kappa,\ t)}^{\varsigma} \Rightarrow (e_\kappa,\ \rho',\ \sigma',\ \kappa_\kappa,\ t')$$

$$\begin{aligned}\text{where}\ \ \kappa &= (\text{kont}\ x_\kappa\ \rho_\kappa\ e_\kappa\ \kappa_\kappa)\\ \rho' &= \rho_\kappa[x_\kappa \mapsto a_x]\\ \sigma' &= \sigma[a_x \mapsto v]\\ a_x &= (x_\kappa,\ t')\\ v &= \delta(op, (\mathcal{A}(ae_1,\ \varsigma)\ \ldots\ \mathcal{A}(ae_j,\ \varsigma)))\\ t' &= t+1\end{aligned}$$

Set! An additional change is made to the store in this case. The current address of the variable x_{set} is obtained from the environment ρ and set to the value of ae. The return-value for set! is always VOID.

$$\overbrace{((\text{set!}\ x_{set}\ ae),\ \rho,\ \sigma,\ \kappa,\ t)}^{\varsigma} \Rightarrow (e_\kappa,\ \rho',\ \sigma',\ \kappa_\kappa,\ t')$$

$$\begin{aligned}\text{where}\ \ \kappa &= (\text{kont}\ x_\kappa\ \rho_\kappa\ e_\kappa\ \kappa_\kappa)\\ \rho' &= \rho_\kappa[x_\kappa \mapsto a_x]\\ \sigma' &= \sigma[\rho(x_{set}) \mapsto \mathcal{A}(ae,\ \varsigma)][a_x \mapsto \mathsf{VOID}]\\ a_x &= (x_\kappa,\ t')\\ t' &= t+1\end{aligned}$$

A concrete interpretation can now be performed by computing the transitive closure over the transition relation $(\Rightarrow)$ starting with a ς_0, obtained by injecting a program into a starting state which contains an appropriate final continuation. I is a function for producing this starting state given a program e:

$$I(e) = (e, [], [], (\text{kont r [] r (halt)}), 0)$$

The return value of the program is captured in a variable (e.g. r) and the continuation (halt) cannot be transitioned from, so execution may terminate.

3. Abstract Semantics

We store-allocate nested continuations, allowing us to perform a structural abstraction bounding the machine's address-space to create a computable approximation of our concrete semantics [16] [30]. Notice that our abstract semanics in Figure 3 contain three global changes from their concrete counterparts. Continuations now contain an address for their parent continuation and have been extended with a timestamp. Timestamps themselves have been redefined as lists of labels. The purpose of this transformation is fundamentally to achieve computability by making our state-space finite.

By store-allocating continuations, we have introduced merging between different continuations for the same let, and bounded the stack. This is the first crucial change we've made as it threads the last source of recursion in the state-space through the store, and puts the complexity of our analyses entirely in the hands of the machine's address-space. The next crucial change is to replace timestamps with lists of labels which can be truncated to a fixed length to ensure computability. Instead of incrementing a number that ensures each new binding we create is unique, we use a summary of the program's history which can be shortened to any approximation we would like. In the case of 0-CFA, we always use the empty list, which merges all values for like-named variables together.

In order to support this merging, we replace concrete values with flow-sets of abstract values $\hat{D}$. Using a finite number of abstract values is a simple way to ensure these flow-sets will eventually reach $\top$ if repeatedly extended.

Our framework is parameterized by an abstract atomic-expression evaluator, a timestamp allocation function, and a primitive-operation evaluator.

$\hat{\mathcal{A}}$ transforms an atomic-expression, in the context of an abstract state, into a flow-set.

$$\hat{\mathcal{A}}\colon \mathsf{AE} \times \widehat{State} \rightharpoonup \hat{D}$$

For most of the analyses we might consider, this function looks up variables and combines lambdas with the current environment:

$$\begin{aligned}\hat{\mathcal{A}}(\text{false},\ (e,\ \hat\rho,\ \hat\sigma,\ \hat\kappa,\ \hat t)) &= \{\mathsf{FALSE}\}\\ \hat{\mathcal{A}}(\text{void},\ (e,\ \hat\rho,\ \hat\sigma,\ \hat\kappa,\ \hat t)) &= \{\mathsf{VOID}\}\\ &\ldots\\ \hat{\mathcal{A}}(x,\ (e,\ \hat\rho,\ \hat\sigma,\ \hat\kappa,\ \hat t)) &= \hat\sigma(\hat\rho(x))\\ \hat{\mathcal{A}}(lam,\ (e,\ \hat\rho,\ \hat\sigma,\ \hat\kappa,\ \hat t)) &= \{\langle lam, \hat\rho\rangle\}\end{aligned}$$

Closures are now shown with ellipsis to indicate that these functions may by modified to extend closures with addtional fields if necessary.

The abstract allocation function is the sum of all timestamp manipulations which have been factored out of the main abstract semantics. It provides a simple interface for controling the method of polyvariance throughout the analysis.

$$\widehat{alloc}\colon \widehat{State} \rightharpoonup \widehat{Time}$$

For a monovariant analysis like 0-CFA, this function always returns the empty timestamp:

$$\widehat{alloc}(\hat\varsigma) = ()$$

The abstract prim-op evaluator takes an operation and a list of flow-sets to produce a flow-set of possible output values for these inputs.

$$\hat\delta\colon \mathsf{OP} \times \hat{D}^* \rightharpoonup \hat{D}$$

$$
\begin{aligned}
\hat{\varsigma} \in \widehat{State} &= \widehat{Eval} + \widehat{Apply} \\
\widehat{Eval} &= \mathsf{E} \times \widehat{Env} \times \widehat{Store} \times \widehat{Kont} \times \widehat{Time} \\
\widehat{Apply} &= \widehat{Value} \times \hat{D}^{*} \times \mathsf{Label} \\
&\quad \times \widehat{Env} \times \widehat{Store} \times \widehat{Kont} \times \widehat{Time} \\
\hat{\rho} \in \widehat{Env} &= \mathsf{Var} \rightharpoonup \widehat{Addr} \\
\hat{\sigma} \in \widehat{Store} &= (\widehat{Addr} \rightharpoonup \hat{D}) + (\widehat{KAddr} \rightharpoonup \widehat{Kont}) \\
\hat{\kappa} \in \widehat{Kont} &= (\text{kont } x\ \hat{\rho}\ e\ \hat{a}_{\kappa}\ \hat{t}) \mid (\text{halt}) \\
\hat{t} \in \widehat{Time} &= \mathsf{Label}^{*} \\
\hat{a} \in \widehat{Addr} &= \mathsf{Var} \times \widehat{Time} \\
\hat{a}_{\kappa} \in \widehat{KAddr} &= \mathsf{Var} \\
\hat{d} \in \hat{D} &= \mathcal{P}(\widehat{Value}) \\
\hat{v} \in \widehat{Value} &= \mathsf{Lam} \times \widehat{Env} + \{\mathsf{FALSE},\ \mathsf{VOID},\ \ldots\}
\end{aligned}
$$

Figure 3. Abstract state-space for 0-CFA

Let The current continuation is store-allocated, and a new continuation is produced which references it. A return timestamp is placed in the continuation which gives an analysis the ability to exploit knowledge of both the current history and this previous history when returning. The allocation function is used to factor out the work of producing the saved timestamp.

$$\overbrace{((\text{let } (x\ e_1)\ e_2),\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})}^{\hat{\varsigma}} \approx> (e_1,\ \hat{\rho},\ \hat{\sigma}',\ \hat{\kappa}',\ \hat{t})$$

$$
\begin{aligned}
\text{where}\quad \hat{\kappa} &= (\text{kont } x_{\kappa}\ \hat{\rho}_{\kappa}\ e_{\kappa}\ \hat{a}_{\kappa}\ \hat{t}_{\kappa}) \\
\hat{\kappa}' &= (\text{kont } x\ \hat{\rho}\ e_2\ x_{\kappa}\ \hat{t}'_{\kappa}) \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [x_{\kappa} \mapsto \hat{\kappa}] \\
\hat{t}'_{\kappa} &= \widehat{alloc}(\hat{\varsigma})
\end{aligned}
$$

Call: Eval → Apply This first half of the call-site transition evaluates the function to be called non-deterministically and transitions to an apply state for each appropriate closure. The allocation function is given the option of modifying or replacing the current continuation's saved timestamp.

$$\frac{\langle(\lambda\ (x_1\ \ldots\ x_j)\ e),\ \ldots\rangle \in \hat{\mathcal{A}}(ae_f,\ \hat{\varsigma})}{\underbrace{((ae_f\ ae_1 \ldots)^{l},\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})}_{\hat{\varsigma}} \approx> (\hat{v},\ (\hat{d}_1 \ldots),\ l,\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa}',\ \hat{t})}$$

$$
\begin{aligned}
\text{where}\quad \hat{v} &= \langle(\lambda\ (x_1\ \ldots\ x_j)\ e),\ \ldots\rangle \\
\hat{d}_i &= \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma}) \\
\hat{\kappa} &= (\text{kont } x_{\kappa}\ \hat{\rho}_{\kappa}\ e_{\kappa}\ \hat{a}_{\kappa}\ \hat{t}_{\kappa}) \\
\hat{\kappa}' &= (\text{kont } x_{\kappa}\ \hat{\rho}_{\kappa}\ e_{\kappa}\ \hat{a}_{\kappa}\ \hat{t}'_{\kappa}) \\
\hat{t}'_{\kappa} &= \widehat{alloc}(\hat{\varsigma})
\end{aligned}
$$

Call: Apply → Eval A new timestamp is produced for this call and our argument flow-sets are bound to their variables under this new history.

$$\overbrace{(\hat{v},\ (\hat{d}_1 \ldots \hat{d}_j),\ l,\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})}^{\hat{\varsigma}} \approx> (e,\ \hat{\rho}',\ \hat{\sigma}',\ \hat{\kappa},\ \hat{t}')$$

$$
\begin{aligned}
\text{where}\quad \hat{v} &= \langle(\lambda\ (x_1\ \ldots\ x_j)\ e),\ \hat{\rho}_{\lambda},\ \ldots\rangle \\
\hat{\rho}' &= \hat{\rho}_{\lambda}[x_i \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{d}_i] \\
\hat{a}_i &= (x_i,\ \hat{t}') \\
\hat{t}' &= \widehat{alloc}(\hat{\varsigma})
\end{aligned}
$$

If Conditionals transition to their nested expressions as appropriate.

$$\frac{\hat{v} \in \hat{\mathcal{A}}(ae,\ \hat{\varsigma}) \qquad \hat{v} \neq \mathsf{FALSE}}{\underbrace{((\text{if } ae\ e_t\ e_f),\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})}_{\hat{\varsigma}} \approx> (e_t,\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})}$$

$$\frac{\hat{v} \in \hat{\mathcal{A}}(ae,\ \hat{\varsigma}) \qquad \hat{v} = \mathsf{FALSE}}{\underbrace{((\text{if } ae\ e_t\ e_f),\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})}_{\hat{\varsigma}} \approx> (e_f,\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})}$$

AE The top of the continuation stack is popped and each of the referenced store-allocated continuations are restored. The expression in the current continuation is transitioned to at a new timestamp selected by the allocation function. The return flow-set for ae is placed at an address for x_{κ} under the new timestamp in the continuation's binding environment $\hat{\rho}_{\kappa}$.

$$\frac{\hat{\kappa}' \in \hat{\sigma}(\hat{a}_{\kappa})}{\underbrace{(ae,\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})}_{\hat{\varsigma}} \approx> (e_{\kappa},\ \hat{\rho}',\ \hat{\sigma}',\ \hat{\kappa}',\ \hat{t}')}$$

$$
\begin{aligned}
\text{where}\quad \hat{\kappa} &= (\text{kont } x_{\kappa}\ \hat{\rho}_{\kappa}\ e_{\kappa}\ \hat{a}_{\kappa}\ \hat{t}_{\kappa}) \\
\hat{\rho}' &= \hat{\rho}_{\kappa}[x_{\kappa} \mapsto \hat{a}_x] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_x \mapsto \hat{\mathcal{A}}(ae,\ \hat{\varsigma})] \\
\hat{a}_x &= (x_{\kappa},\ \hat{t}') \\
\hat{t}' &= \widehat{alloc}(\hat{\varsigma})
\end{aligned}
$$

Prim The return flow-set of a primitive operation is computed using the $\hat{\delta}$ function.

$$\frac{\hat{\kappa}' \in \hat{\sigma}(\hat{a}_\kappa)}{\underbrace{((\text{prim}\ \mathit{op}\ \mathit{ae}_1\ \ldots\ \mathit{ae}_j),\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})}_{\hat{\varsigma}} \approx\!> (e_\kappa,\ \hat{\rho}',\ \hat{\sigma}',\ \hat{\kappa}',\ \hat{t}')}$$

$$\begin{aligned}\text{where}\quad \hat{\kappa} &= (\text{kont}\ x_\kappa\ \hat{\rho}_\kappa\ e_\kappa\ \hat{a}_\kappa\ \hat{t}_\kappa)\\ \hat{\rho}' &= \hat{\rho}_\kappa[x_\kappa \mapsto \hat{a}_x]\\ \hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_x \mapsto \hat{d}]\\ \hat{a}_x &= (x_\kappa,\ \hat{t}')\\ \hat{d} &= \hat{\delta}(\mathit{op}, (\hat{\mathcal{A}}(\mathit{ae}_1,\ \hat{\varsigma})\ \ldots\ \hat{\mathcal{A}}(\mathit{ae}_j,\ \hat{\varsigma})))\\ \hat{t}' &= \widehat{\mathit{alloc}}(\hat{\varsigma})\end{aligned}$$

Set! The flow-set for x_{set} is extended in the current environment before returning $\{\mathsf{VOID}\}$ to x_κ.

$$\frac{\hat{\kappa}' \in \hat{\sigma}(\hat{a}_\kappa)}{\underbrace{((\text{set!}\ x_{set}\ \mathit{ae}),\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})}_{\hat{\varsigma}} \approx\!> (e_\kappa,\ \hat{\rho}',\ \hat{\sigma}',\ \hat{\kappa}',\ \hat{t}')}$$

$$\begin{aligned}\text{where}\quad \hat{\kappa} &= (\text{kont}\ x_\kappa\ \hat{\rho}_\kappa\ e_\kappa\ \hat{a}_\kappa\ \hat{t}_\kappa)\\ \hat{\rho}' &= \hat{\rho}_\kappa[x_\kappa \mapsto \hat{a}_x]\\ \hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_x \mapsto \{\mathsf{VOID}\}]\\ &\qquad \sqcup [\hat{\rho}(x_{set}) \mapsto \hat{\mathcal{A}}(\mathit{ae},\ \hat{\varsigma})]\\ \hat{a}_x &= (x_\kappa,\ \hat{t}')\\ \hat{t}' &= \widehat{\mathit{alloc}}(\hat{\varsigma})\end{aligned}$$

An analysis in this framework is computed by finding the transitive closure over the abstract transition relation ($\approx\!>$) starting with a $\hat{\varsigma}_0$, obtained by injecting a program into a starting state which contains an appropriate final continuation. A function for producing this starting state given a program e could look like:

$$\hat{I}(e) = (e, [], [\text{halt} \mapsto (\text{halt})], (\text{kont r } []\text{ r halt } ()), ())$$

The return values of the program are captured in a variable (e.g. r) and the continuation (halt) cannot be transitioned from.

4. Tuning

The power of this framework, and our approach in general, lies in distingushing our auxilliary functions $\hat{\mathcal{A}}$ and $\widehat{\mathit{alloc}}$, describing the constraints under which they can be soundly reimagined, and giving them the maximum power these constraints allow to tune the analysis for precisely the polyvariance desired.

We are now able to modify these functions in isolation in order to instantiate various desired analyses with different polyvariant behaviors.

4.1 Call sensitivity

This first analysis is our implementation of k-CFA *as described* in Shivers' seminal paper. For a fixed natural number k, this analysis differentiates bound variables by an abstract history comprised of the last k call-sites. We can define these first three analyses, using the atomic-expression evaluator as defined for 0-CFA. The allocation function alone is sufficient for implementing this kind of polyvariance.

During function application, we prefix our current timestamp with the latest call and keep at most k labels:

$$\widehat{\mathit{alloc}}((\hat{v},\ (\hat{d}\ldots),\ l,\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) = \text{take–left}(k,\ l : \hat{t})$$

We define take–left as returning the maximum left-most sublist which is at-most length k. This restriction enforces the bound on history length which makes the analysis computable.

When we return, we leave the current timestamp as-is, continuing its life into the continuation. The next two analyses will take return-points into consideration, but for now the current timestamp is carried through:

$$\begin{aligned}\widehat{\mathit{alloc}}(((\text{prim}\ \mathit{op}\ \mathit{ae}\ \ldots),\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) &= \hat{t}\\ \widehat{\mathit{alloc}}(((\text{set!}\ x\ \mathit{ae}),\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) &= \hat{t}\\ \widehat{\mathit{alloc}}((\mathit{ae},\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) &= \hat{t}\end{aligned}$$

Because we are not changing our time-stamp when we return, there is no need to store or manipulate the continuation's timestamp:

$$\begin{aligned}\widehat{\mathit{alloc}}(((\text{let}\ (x\ e_1)\ e_2),\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) &= ()\\ \widehat{\mathit{alloc}}(((\mathit{ae}\ \ldots),\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) &= ()\end{aligned}$$

4.2 Call+Return sensitivity

This analysis is k-CFA *as implemented* in Shivers' seminal paper. For a fixed natural number k, this analysis differentiates bound variables by an abstract history comprised of the last k call-sites or return-points. While k is described as being the number of call-sites stored, once CPS-converted, return-points in the original direct-style program become reified as syntactic lambdas in CPS. Thus, return-points are effectively considered in the abstract histories of such an analysis.

At function application, we extend our current timestamp as we did in the previous analysis:

$$\widehat{\mathit{alloc}}((\hat{v},\ (\hat{d}\ldots),\ l,\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) = \text{take–left}(k,\ l : \hat{t})$$

When a value is returned however, we also extend the current timestamp with the return-point:

$$\widehat{alloc}(((\text{prim}\ op\ ae\ \ldots)^l,\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_\kappa),\ \hat{t})) = \text{take–left}(k,\ l : \hat{t})$$

$$\widehat{alloc}(((\text{set!}\ x\ ae)^l,\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_\kappa),\ \hat{t})) = \text{take–left}(k,\ l : \hat{t})$$

$$\widehat{alloc}((ae^l,\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_\kappa),\ \hat{t})) = \text{take–left}(k,\ l : \hat{t})$$

4.3 Stack sensitivity

A quite different interpretation of k is as the top-k stack frames. This analysis differentiates bound variables by an abstract history comprised of the call-site labels for the top-k stack frames.

At function application, we extend our current timestamp as we did in both previous analyses:

$$\widehat{alloc}((\hat{v},\ (\hat{d}\ldots),\ l,\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) = \text{take–left}(k,\ l : \hat{t})$$

Then we can simply reinstate it when a value is returned:

$$\widehat{alloc}(((\text{prim}\ op\ ae\ \ldots),\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_\kappa),\ \hat{t})) = \hat{t}_\kappa$$

$$\widehat{alloc}(((\text{set!}\ x\ ae),\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_\kappa),\ \hat{t})) = \hat{t}_\kappa$$

$$\widehat{alloc}((ae,\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_\kappa),\ \hat{t})) = \hat{t}_\kappa$$

4.4 Cartesian Product Algorithm

The Cartesian Product Algorithm (CPA), is an analysis for performing type inference which deals in whole tuples of arguments as opposed to accumulating types (abstract values in our case) within flow-sets individually. It assigns each function a flow-set of type-tuples, as opposed to a tuple of flow-sets containing individual types. [1]

The authors' motivating example is a polymorphic max function:

$$max(a, b) = \text{if}\ a > b\ \text{then}\ a\ \text{else}\ b$$

Here, the only constraint for an input to max is that it support comparison, so a call $max(``abc'',\ ``xyz'')$ makes as much sense as a call $max(3,\ 5)$. However, if both these calls are made with a sufficient amount of obfuscating call-history behind them, merging will cause the flow sets for a and b to each include both *int* and *string* regardless of the length of call-histories used. This is imprecise because it implies that max("hello", 9) is possible when it is not.

The solution proposed with CPA is to store entire tuples of types, preserving inter-argument patterns, and avoiding spurious concrete variants. In essense, this differentiates each argument binding with the entire tuple of types sent in that call. This suggests a format for abstract histories as a list of types:

$$\widehat{Time} = \widehat{Type}^*$$

$$\widehat{Type} = \mathsf{Lam} + \{\mathsf{FALSE},\ \mathsf{VOID}, \ldots\}$$

In addition to the innate advantages of defining these types separately from abstract values, it is necessary in the case of closures. An abstract closure includes a binding environment, which would introduce recursion into the state-space should these bindings themselves include closures. A syntactic lambda is precise enough to represent a type, and removing environments from their closures, we reduce each abstract value in a given flow-set to its type using a new function $\hat{\mathcal{T}}$:

$$\hat{\mathcal{T}} \colon \hat{D} \to \mathcal{P}(\widehat{Type})$$

Function application produces a timestamp for its bindings comprising the full list of evaluated arguments, each reduced to their types:

$$\widehat{alloc}((\hat{v},\ (\hat{d}_1 \ldots \hat{d}_j),\ l,\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) = (\hat{\mathcal{T}}(\hat{d}_1) \ldots \hat{\mathcal{T}}(\hat{d}_j))$$

When a flow-set is returned its binding is differentiated by a timestamp containing the types of the returned flow-set:

$$\widehat{alloc}(((\text{prim}\ op\ ae_1\ \ldots\ ae_j),\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_\kappa),\ \hat{t})) = (\hat{\mathcal{T}}(\hat{\delta}(op, (\hat{\mathcal{A}}(ae_1,\ \hat{\rho},\ \hat{\sigma})\ \ldots\ \hat{\mathcal{A}}(ae_j,\ \hat{\rho},\ \hat{\sigma})))))$$

$$\widehat{alloc}(((\text{set!}\ x\ ae),\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_\kappa),\ \hat{t})) = (\{\mathsf{VOID}\})$$

$$\widehat{alloc}((ae,\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_\kappa),\ \hat{t})) = (\hat{\mathcal{T}}(\hat{\mathcal{A}}(ae,\ \hat{\rho},\ \hat{\sigma})))$$

There is no need for continuations to contain a saved timestamp:

$$\widehat{alloc}(((\text{let}\ (x\ e_1)\ e_2),\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) = ()$$

$$\widehat{alloc}(((ae\ \ldots),\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_\kappa),\ \hat{t})) = ()$$

4.5 Polymorphic Splitting

Polymorphic splitting is a kind of compromise between 0-CFA and k-CFA where the length of each timestamp varies on a per-function basis. Lambdas which have been let-bound are analyzed with a history length 1-greater than that of their parent expression. Because the number of nested let-forms possible is bounded by the length of the program, k is also bounded. We again define timestamps as lists of labels:

$$\hat{t} \in \widehat{Time} = \mathsf{Label}^*$$

In polymorphic-splitting, let-bound expressions are analyzed with the current timestamp, appended with a label unique to the let-expression. When a closure is produced, the current timestamp is included, so a let-bound lambda will carry with it the timestamp of the let-form appended with a label referencing that point in the program. When a let-bound variable is referenced, any timestamps returned have this let-form label changed for a label specific to the exact syntactic variable reference. In this way, a let-bound lambda referenced 3 times will be analyzed 3 times with values contours specific to each of these points of reference. By contrast, an inline or lambda-bound lambda will be analyzed at the timestamp in its closure, without any labels being replaced. This causes it to inherit the timestamp at its closure creation. [31]

To achieve this heuristic for timestamp-length in our semantics we modify both the atomic-expression evaluator and the allocation function.

$$\hat{\mathcal{A}}(x,\ (e,\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) = \hat{\sigma}(\hat{\rho}(x))$$
$$\hat{\mathcal{A}}(lam_{[let]},\ (e^{l},\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) = \{\langle lam, \hat{\rho}, \hat{t}\colon l\rangle\}$$
$$\hat{\mathcal{A}}(lam,\ (e,\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) = \{\langle lam, \hat{\rho}, \hat{t}\rangle\}$$

Here, $lam_{[let]}$ indicates that lam is let-bound in the program. Closures are modified upon access to differentiate based on the lexical point of reference.

$$\begin{aligned}&\hat{\mathcal{A}}(x_{[let]},\ (e^{l},\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t}))\\ &\quad = \{\langle e_{\lambda}^{l_{\lambda}},\ \hat{\rho}_{\lambda},\ \hat{t}_{\lambda}[l_{\lambda}/l]\rangle \mid \langle e_{\lambda}^{l_{\lambda}},\ \hat{\rho}_{\lambda},\ \hat{t}_{\lambda}\rangle \in \hat{\sigma}(\hat{\rho}(x))\}\\ &\quad\quad \cup\ \{\widehat{basic} \mid \widehat{basic} \in \hat{\sigma}(\hat{\rho}(x))\}\\ &\widehat{basic} \in \widehat{Basic} =\ \{\mathsf{TRUE},\ \mathsf{FALSE},\ \mathsf{VOID}, \ldots\}\end{aligned}$$

Functions are analyzed using the context stored in their closure.

$$\widehat{alloc}((\langle(\lambda\ (x_1\ \ldots)\ e),\ \hat{\rho}_{\lambda},\ \hat{t}_{\lambda}\rangle,\ (\hat{d}_1\ \ldots),\ l,\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t}))\\ =\ \hat{t}_{\lambda}$$

The previous timestamp can be resurrected when we return.

$$\widehat{alloc}(((\text{let}\ (x\ e_1)\ e_2),\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) = \hat{t}$$
$$\widehat{alloc}(((ae\ \ldots),\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_{\kappa}),\ \hat{t})) = \hat{t}_{\kappa}$$
$$\widehat{alloc}(((\text{prim}\ op\ ae\ \ldots),\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_{\kappa}),\ \hat{t})) = \hat{t}_{\kappa}$$
$$\widehat{alloc}(((\text{set!}\ x\ ae),\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_{\kappa}),\ \hat{t})) = \hat{t}_{\kappa}$$
$$\widehat{alloc}((ae,\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_{\kappa}),\ \hat{t})) = \hat{t}_{\kappa}$$

4.6 Continuation sensitivity

This analysis is an oddity we implemented while trying to model call-sensitivity in CPS. While our Call+Return senstitive analysis uses the single return-point, which reflects the location of the continuation's callsite in CPS, this analysis uses the call-sites a value would return *through*. We present this analysis as a further demonstration of the ease with which an analysis designer can capture an idea within this framework. It remains reflective of the broad applicability of an allocation-based approach, even if it does not reflect the behavior of k-CFA on a CPS language.

At function application, we extend our current timestamp as we do for call+return sensitivity:

$$\widehat{alloc}((\hat{v},\ (\hat{d}\ldots),\ l,\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) = \text{take–left}(k,\ l : \hat{t})$$

The difference with this analysis is in the maintainance of return-points within the continuation's timestamp. When a value is returned, we extend the current timestamp with the stored timestamp and remove all but the first k labels:

$$\widehat{alloc}(((\text{prim}\ op\ ae\ \ldots),\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_{\kappa}),\ \hat{t}))\\ = \text{take–left}(k,\ \text{append}(\hat{t}_{\kappa},\ \hat{t}))$$
$$\widehat{alloc}(((\text{set!}\ x\ ae),\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_{\kappa}),\ \hat{t}))\\ = \text{take–left}(k,\ \text{append}(\hat{t}_{\kappa},\ \hat{t}))$$
$$\widehat{alloc}((ae,\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_{\kappa}),\ \hat{t}))\\ = \text{take–left}(k,\ \text{append}(\hat{t}_{\kappa},\ \hat{t}))$$

When a new continuation is created, there are not yet any return-points we need to cross through before it can be restored, so we give fresh continuations the empty timestamp:

$$\widehat{alloc}(((\text{let}\ (x\ e_1)\ e_2),\ \hat{\rho},\ \hat{\sigma},\ \hat{\kappa},\ \hat{t})) = ()$$

Each time a function invokation is reached, the timestamp in the current continuation is extended with a label l_{-r} created by appending a $-\mathrm{r}$ onto the end of the label l:

$$\widehat{alloc}(((ae\ \ldots),\ \hat{\rho},\ \hat{\sigma},\ (\text{kont}\ \ldots\ \hat{t}_{\kappa}),\ \hat{t}))\\ =\ \text{take–left}(k,\ \text{append}(\hat{t}_{\kappa},\ l_{-r}))$$

This is so the new label is denoted as a return through callsite l and is not confused as a call from l. We right-append and keep only the first k labels we pass through because any further callsites are guaranteed to be off the end of our maximum history once a value is returned.

5. Comparison

Our Racket implementation is a fairly concise module which accepts already desugared benchmarks for analysis. It is implemented as a simple worklist algorithm expanding its search through a program's abstract state space until a fixed-point is reached. In addition to what we've described the analysis supports global and per-point widening along with abstract garbage collection [17] [20]. These additions are straightforward ways of improving speed and precision which are entirely orthogonal to the changes we discuss in this paper.

As it turns out, there are programs which will differentiate our implementations of call-sensitivity to favor any of the three. For example, given the following program, tracking only calls performs better than tracking both calls and returns. This is because the last callsite (id 123) as opposed to (id "abc'') determines the value returned to v. The last return-point x is the same in both cases and so does not.

```
(let (id (lambda (x) x))
     (let (f (lambda (g)
                       (let (v (g))
                            v)))
```

```
        (let (_ (f (lambda () (id 123))))
             (f (lambda () (id "abc"))))))
```

The following program performs best under the stack-based model of program context. This is because an intermediate call to $(addhist)$ obfuscates our timestamp unnecessarily while tracking calls or returns. By contrast, the stack-based method resurrects the timestamp used just before the call to $(addhist)$ upon return.

```
(let (addhist (lambda () (prim void)))
     (let (f (lambda (a) (let (_ (addhist))
                               (let (v a) v))))
        (let (_ (f 123))
             (f "abc"))))
```

6. Conclusion

We have presented an approach to producing a factored abstract interpretation that can be tuned for a specific kind of polyvariance by an abstract allocation policy. We have futher presented three different interpretations of k-CFA and show how easily they can be expressed exclusively as a change to this policy. In general, our approach allows different forms of polyvariance to be compared and casts light on their respective behaviors. It also exposes a space of sound behaviors, ready for exploration.

The ease with which new allocation policies can be expressed, and new polyvariant analyses implemented, makes our approach especially appealing. It allows for rapid prototyping of novel strategies within an existing framework. New ideas can be quickly implemented and tested. Widely differing policies can be used for the same benchmarks and more directly compared. In the future, our approach may spur innovation in this research space by making it easy to examine novel allocation strategies.

This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0106 (Automated Program Analysis for Cyber Security). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] Agesen, O. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In Proceedings of ECOOP 1995 (1995), pp. 226.

[2] Besson, F. CPA beats ∞-CFA. Formal Techniques for Java-like Programs, July 2009. p. 7.

[3] Cousot P. The calculational design of a generic abstract interpreter. NATO ASI Series F. Broy, M. and Steinbrüggen, R. (eds.): *Calculational System Design*. 1999. pp. 421-506.

[4] Cousot P. Types as Abstract Interpretations. Symposium on Principals of Programming Languages. 1997. pp. 316-331.

[5] Cousot P. and Cousot R. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Symposium on Principals of Programming Languages. 1977. pp. 238-252.

[6] Cousot P. and Cousot R. Systematic design of program analysis frameworks. Symposium on Principals of Programming Languages. 1979. pp. 269-282.

[7] Felleisen, M., Findler R. and Flatt, M. Semantics Engineering with PLT Redex. August 2009.

[8] Felleisen, M. and Friedman, D. P. A calculus for assignments in higher-order languages. Proceedings of the Symposium on Principles of Programming Languages, page 314. 1987.

[9] Jagganathan, S, and Weeks, S. A Unified Treatment of Flow Analysis in Higher-Order Languages. ACM Symposium on Principles of Programming Languages, January 1995. ACM Press. pp.393-407.

[10] Jones, N.D. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. Symposium on Principles of Programming Languages. 1982. pp. 66-74.

[11] Jones, N.D. and Muchnick, S. Flow analysis of lambda expressions (preliminary version). Proceedings of the 8th Colloquium on Automata Languages and Programming. 1981. pp. 114-128.

[12] Midtgaard, J. Control-Flow Analysis of Functional Programs. ACM Computing Surveys, Vol. 44. June 2012.

[13] Midtgaard, J. and Jensen, T. A Calculational Approach to Control-flow Analysis by Abstract Interpretation. SAS, volume 5079 of Lecture notes in Computer Science. 2008. pp. 347-362.

[14] Midtgaard, J. and Van Horn, D. Subcubic Control Flow analysis Algorithms. Higher-Order and Symbolic Computation. May 2009.

[15] Midtgaard, J. and Jensen, T. Control-ow analysis of function calls and returns by abstract interpretation. International Conference on Functional Programming. 2009.

[16] Might, M. Abstract interpreters for free. Static Analysis Symposium. 2010. pp. 407-421.

[17] Might, M. Environment Analysis of Higher-Order Languages. Ph.D. Dissertation. Georgia Institute of Technology. 2007.

[18] Might, M. Logic-Flow Analysis of Higher-Order Programs. Principals of Programming Langauges. January 2007. pp. 185-198.

[19] Might, M. and Shivers, O. Environment analysis via ΔCFA. Symposium on the Principals of Programming Languages. January 2006. pp. 127-140.

[20] Might, M. and Shivers, O. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. International Conference on Functional Programming. September 2006. pp. 13-25.

[21] Might, M. and Manolios, P. A posteriori soundness for non-deterministic abstract interpretations. 10th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2009). Savannah, Georgia, USA. January, 2009. pp. 260-274.

[22] Milanova, A., Rountev A., and Ryder, B.G. Parameterized ob-

ject sensitivity for points-to analysis for Java. ACM Transaction on Software Engineering and Methodology. 2005. pp. 1-41.

[23] Nielson, F., Nielson, H.R. and Hankin, C. Principals of Program Analysis. Springer-Verlang. 1999.

[24] Palsberg, J. and Pavlopoulou, C. From Polyvariant Flow Information to Intersection and Union Types. Principals of Programming Languages. 1998. pp. 197-208.

[25] Shivers, O. Control-flow analysis in Scheme. Programming Language Design and Implementation. June 1988. pp. 164-174.

[26] Shivers, O. Control-Flow Analysis of Higher-Order Languages. PhD dissertation. School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMUCS-91-145.

[27] Smaragdakis, Y., Bravenboer M., and Lhotak, O. Pick your contexts well: understanding object-sensitivity. Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: ACM, 2011, pp. 1730.

[28] Van Horn, D. and Mairson, G.H. Deciding k-CFA is complete for EXPTIME. International Conference on Functional Programming. September, 2008. pp. 275-282.

[29] Van Horn, D. and Mairson, G.H. Flow Analysis, Linearity, and PTIME. Static Analysis Symposium 2008. pp. 255-269.

[30] Van Horn, D. and Might, M. Abstracting Abstract Machines. International Conference on Functional Programming 2010. Baltimore, Maryland. September, 2010. pp. 51-62.

[31] Wright, A. K. and Jagannathan, S. Polymorphic splitting: An effective polyvariant flow analysis. ACM Transactions on Programming Languages and Systems. January 1998, pages 166-207.