Thesis Proposal

Thomas Gilray

University of Utah
tgilray@cs.utah.edu

Polyvariance may be soundly, efficiently, and arbitrarily refined, permitting static analyses to adapt a style more efficient for their target than existing non-introspective heuristics.

Abstract. We present a unified theory of polyvariance in abstract interpretations, and apply it to the construction of a framework for sound refinement in the presence of store-widening. Polyvariance is a method for increasing the precision of a static analysis by maintaining multiple abstractions for each syntactic point of interest. Many styles exist in the literature, each encoding a heuristic for the trade-off between complexity and precision which varies in its efficiency between targets of analysis. In order to adapt polyvariance for a specific program under investigation, we describe a lattice of analyses ranging over *all conceivable* known and unknown flavors of polyvariance and give a method for incrementally refining a sound intermediate result to an improved position on this lattice. This permits an analysis to bootstrap off previously discovered facts when deciding in what manner to tune polyvariance. Though store-widening eliminates information necessary for refinement, we show how to soundly recover this ability for widened analyses by simultaneously modeling a property of the analysis itself that we term *contribution*. We describe the proposed thesis statement in detail and discuss future work.

1 Introduction

The *polyvariance* of an abstract interpretation, in general terms, is the degree to which program locations at runtime are broken into multiple separately maintained approximations of their concrete behavior. Polyvariance has been explored in many different forms including sensitivity to the dynamic type of arguments (CPA), sensitivity to intra-procedural paths (path-sensitivity), sensitivity to the allocation-point of objects (object-sensitivity), and sensitivity to calling context (call-sensitivity), among others [1, 9, 10, 22, 27, 28]. *Call-sensitivity*, an extensively explored form of context-sensitivity, differentiates values passed to a function by an approximate history of k calls leading up to the binding. Informally, it treats functions as globally inlined k iterations for the benefit of precision.

Consider as an illustration of this, a function `f` with two call-sites:

```
... (let ([f (lambda (x) (g x y))])
      (f 0)
      (f #t))
```

A 0-call-sensitive (*monovariant*) analysis of this code will merge all values for `x` together so it appears in any context to be either `0` or `#t`. By contrast, a 1-call-sensitive analysis will maintain a separate abstraction for `x` specific to each of the call-sites for `f`. This is enough to obtain perfect precision for `x`, however a 2-call-sensitive analysis or better will be required to avoid merging for the corresponding parameter of `g`. Only an unbounded degree of polyvariance is enough to guarantee perfect precision in the general case.

Different styles of polyvariance encode a particular fixed heuristic for the trade-off between complexity and precision in an analysis. The efficiency of this trade-off is highly sensitive to the specific program under analysis; a given style of polyvariance will have programs for which it is well-suited, and programs which expose its worst-case behavior. Crucially, for control-flow analyses (CFA) in particular, where a style is most precise, it tends also to be least expensive. This is because imprecision in a CFA requires analyses to explore an even greater range of executions and compounds itself in a vicious cycle [37]. For example, *object-sensitivity* uses the allocation-point of an object for differentiating bindings to variables of its methods and has been shown in practice to be a more effective style of context-sensitivity for object-oriented programs [34].

1.1 Motivating *adaptive* polyvariance: JavaScript

Dynamic languages are notoriously difficult to model precisely. Reduction to a core calculus is an appealing approach, however this increases the need for a significant degree of polyvariance and a more adaptive approach. Perhaps the most compelling effort to date at giving a complete and tractable semantics for JavaScript has been Guha et al.'s λ_{JS} [12] and its successor, Politz et al.'s λ_{S5} [31]. This approach reduces programs to a simple core language consisting of fewer than 35 syntactic forms, reifying the hidden and implicit complexity of full JavaScript as explicit complexity written in the core language.

Desugaring is appealing for analysis designers as it gives a simple and precise semantics to abstract; however, it also presents one of the major obstacles to precise analysis as it adds a significant runtime environment and layers of indirection through it. Consider an example the authors of λ_{S5} use to motivate the need for their carefully constructed semantics: `[] + {}` yields the string `"[object Object]"`. Strangely enough, this behavior is correct as defined by the ECMAScript specification for addition – a complex algorithm encompassing a number of special cases which can interact in unexpected ways [8]. The desugaring process for λ_{S5} replaces addition with a function call to `%PrimAdd` from the runtime environment. `%PrimAdd` in-turn calls `%ToPrimitive` on both its arguments before breaking into cases. This means that for any uses of addition to return precise results, or likely anything other than $\top$, a k-call-sensitive analysis requires $k \geq 2$. Unfortunately, call-sensitivity, and any other context-sensitive analysis which allows the number of variants for a binding to be in $\Omega(n)$ will be exponential in the worst-case due to the structure of environments in higher-order languages [14, 27]. As the desugarer for λ_{S5} bloats even small programs to more than ten thousand lines, applying context-sensitivity to every function is sure to result in an absurdly time-consuming analysis.

Instead, an adaptive style of polyvariance is required, one which attempts an increased precision only where it will be safe, effective, and efficient. This requires a simultaneous modeling of both program behavior and analysis behavior for determining where polyvariance may be best exploited; however, because the model produced by such an analysis is not sound and meaningful until analysis is complete, adjusting polyvariance before this is less likely to be effective. A method for incrementally refining the polyvariance of a sound fix-point is therefore a more plausible way of fitting the application of polyvariance to suit a program.

2 Background

There are no existing methods for performing arbitrary refinement of polyvariance in an abstract interpretation, however our framework does build on a long history of pre-requisite improvements and insights.

2.1 Myriad styles of polyvariance

Originally devised as a technique for data-flow analyses by Sharir and Pnueli [32], polyvariance was utilized in the ‘80s by Jones, Muchnick, and Harrison [19, 13] and generalized to a k-call-sensitive control-flow analysis (k-CFA) of higher-order programs by Shivers [33]. In nearly 25 years since the seminal work on k-CFA, numerous alternative styles of polyvariance for CFA have been explored in the literature. The ‘90s in particular saw an extensive exploration of this design-space. Like k-CFA, many frameworks form a well-ordered hierarchy of analyses encompassing a range of granularities between monovariance and unbounded polyvariance (corresponding to a concrete semantics). Each presents the inexorable trade-off between precision and complexity in a unique way.

Milanova et al. introduced a style of polyvariance which uses a history of the allocation-points for objects [28]. Smaragdakis et al. refined this concept, distinguishing multiple hierarchies encoding different flavors of object-sensitivity along with a new approach called *type-sensitivity* which approximates these [34].

Agesen defined a polyvariant framework for type-recovery which gains the best possible precision for its application by directly using a tuple of types to differentiate functions for analysis [1]. His *Cartesian product algorithm* (CPA) may also be formalized as a style of polyvariance for general flow-analyses [9].

Wright and Jagannathan's *polymorphic-splitting* uses let-bindings as a heuristic for varying call-sensitivity on a per-function basis [37].

Palsberg and Pavlopoulou, among many others, have applied polyvariance to enhancing the precision of type-systems [29].

In their unified treatment of flow-analysis, Jagannathan and Weeks give a polynomial-time widening for call-sensitivity: poly-k-CFA [17]. While this is possible for any style of polyvariance, in an unwidened form all these styles of polyvariance are necessarily exponential in their worst-case behavior.

2.2 The AAM methodology

The Abstracting Abstract-Machines (AAM) approach of Van Horn and Might is a general method for automatically abstracting an arbitrary small-step abstract-machine semantics to obtain an approximation in a variety of styles [15, 24]. Importantly, one such style aims to focus all unboundedness in a semantics on the machine's address-space. We exploit this style of abstraction to produce a framework parameterized by an allocation function mapping variables to abstract addresses as they are bound. Combined with a meta-analysis suitable for tracking a particular flavor of context (e.g. allocation-histories), we show that this function is powerful enough to encode an arbitrary style of polyvariance.

2.3 *A posteriori* soundness

The usual process for proving the soundness of an abstract abstract-machine is *a priori* in the sense that it may be performed entirely before an analysis is executed. The approximation used, formally known as a *Galois connection*, defines an abstraction/concretization relationship between the concrete and abstract semantics. A family of functions α map entities in the concrete semantics to their most precise representative in the abstract semantics. A corresponding family of functions γ map entities in the abstract machine to a set of concrete entities such that $Id \sqsubseteq \gamma \circ \alpha$, placing a strict bound on the concrete executions represented by an analysis result. Proving that an abstract transition relation is sound with respect to a concrete one reduces to an inductive step showing that simulation, (i.e. the defined notion of approximation, γ) is preserved across every transition [23].

By contrast, Might and Manolios describe an *a posteriori* soundness proof where the abstraction map cannot be fully constructed until after analysis [25].

This approach factors α to separate the abstraction of addresses α_{Addr}, producing a parametric map β such that $\beta(\alpha_{Addr}) = \alpha$. The authors show that regardless of the allocation strategy taken during analysis, a consistent abstraction map may be constructed after-the-fact which justifies these choices.

Intuitively, because each allocation-step in a concrete machine is ephemeral, and each address unique, it is impossible to induce an inconsistent abstraction for addresses by any abstract allocation policy. If a concrete machine and its abstract machine are simulated in lock-step, each concrete address a and corresponding abstract address $\hat{a}$ represent a point $[a \mapsto \hat{a}]$ in the abstraction map α_{Addr}. For the abstraction induced by the pairing of a concrete allocator and abstract allocator to be inconsistent, the same concrete address would need to be abstracted to two different abstract addresses. Because a concrete allocator must, by definition, produce a fresh address for every invocation, no such inconsistency is possible, regardless of the abstract allocator chosen. All allocation policies represent a partitioning of the concrete address-space and vice versa.

3 Abstracting Abstract-Machine Semantics

In this section we review a process for abstracting an operational abstract-machine semantics to obtain a static analysis, specifically k-CFA. We discuss polyvariance in the context of k-CFA and distinguish two subtly different variants of call-sensitivity, one corresponding to the style of polyvariance Shivers originally described, and one corresponding to the style he formalized [10].

For simplicity, we describe how to abstract a semantics for the λ-calculus in *continuation-passing-style* (CPS). CPS constrains call-sites to tail-position so functions may never return; instead, callers explicitly pass a continuation forward to be invoked on the result [2, 30]. CPS-conversion is a common transformation for compilers and program analyses [2]. If this transformation records which lambdas are actually continuations, a program may again, along with any optimizations and analysis results, be reconstituted in direct-style form. This means the advantages of CPS can be utilized without loss of information [20]. To keep track of which call-sites represent return-points, we will assume an auxilliary predicate $\mathcal{R} : \mathsf{Lab} \to \mathit{Bool}$ defined by CPS-conversion. The form $(ae_f\ ae_1\ \ldots\ ae_j)^l$ represents a return-point in the original program iff $\mathcal{R}(l)$.

The grammar of CPS λ-calculus structurally distinguishes between complex-expressions e and atomic-expressions ae:

$$e \in \mathsf{E} ::= (ae\ ae\ \ldots)^l \mid (\mathtt{halt})^l \qquad ae \in \mathsf{AE} ::= lam^l \mid x^l$$
$$lam \in \mathsf{Lam} ::= (\lambda\ (x\ \ldots)\ e)^l \qquad x \in \mathsf{Var} = \langle \textit{program variables} \rangle$$
$$l \in \mathsf{Lab} = \langle \textit{unique program labels} \rangle$$

The textbook small-step abstract-machine semantics is straightforward.

$$\varsigma \in \Sigma = \mathsf{E} \times \mathit{Env}$$
$$\rho \in \mathit{Env} = \mathsf{Var} \rightharpoonup \mathit{Clo}$$
$$\mathit{clo} \in \mathit{Clo} = \mathsf{Lam} \times \mathit{Env}$$

We require only a single rule to define state-transition $(\Rightarrow) \subseteq \Sigma \times \Sigma$:

$$\begin{aligned} \overbrace{((ae_f\ ae_1\ \ldots\ ae_j),\ \rho)}^{\varsigma} &\Rightarrow (e_\lambda,\ \rho'),\ \text{where} \\ ((\lambda\ (x_1\ \ldots\ x_j)\ e_\lambda),\ \rho_\lambda) &= \mathcal{A}(ae_f,\ \varsigma) \\ \rho' &= \rho_\lambda[x_i \mapsto clo_i] \\ clo_i &= \mathcal{A}(ae_i,\ \varsigma) \end{aligned}$$

And an atomic-expression evaluator:

$$\begin{aligned} \mathcal{A} &: \mathsf{AE} \times \Sigma \rightharpoonup Clo \\ \mathcal{A}(x,\ (e,\ \rho)) &= \rho(x) \\ \mathcal{A}(lam,\ (e,\ \rho)) &= (lam,\ \rho) \end{aligned}$$

Evaluating a program e using this semantics requires finding the transitive closure of $(\Rightarrow)$ starting from an initial state $\varsigma_0 = (e,\ \varnothing)$. As programs may recur ad infinitum, concrete executions are naturally incomputable.

Producing a computable approximation for a semantics like this is traditionally done in one of two ways: either using a Galois-connection, or with widening/narrowing. In the first approach, imprecision is introduced in the structure of the machine to ensure a finite abstract state-space in which individual entities may each simulate an infinite number of their concrete counterparts [4, 24]. In the second, a widening operator is applied to accelerate convergence within the infinite state-space [3]. The widening approach handles otherwise unbounded semantics because infinite swathes of state-space may be collected at a time. While these approaches are typically combined, in isolation the latter is fundamentally more dynamic as the style of widening may be adapated to an analysis while live and is unrestricted by needing to settle on a particular representation for abstract entities [5].

We will exploit both approaches, producing a finite abstract semantics to enforce computability and using additional widening to further ensure efficiency. We will see how store-passing simultaneously yields both a uniform way to make a semantics finite and the range of all sound polyvariant strategies. In §6 we will allow arbitrary refinement of a given polyvariant strategy for finitizing the semantics, again permitting the dynamicity of the pure widening/narrowing approach, except in a way which allows both live adaptation to an analysis and, crucially, offline refinement based on arbitrary introspection on sound intermediate analysis results.

3.1 Store-passing semantics

Store-passing is an essential technique for implementing a stateful language (e.g. Scheme) within a stateless one (e.g. mathematics). In addition, if all recursive structures are threaded through the store, the address-space becomes the only source of unboundedness in the semantics. A computable approximation is then

obtained by bounding the number of addresses which are used, introducing imprecision in the form of merging between bindings. In this section, we introduce a small-step store-passing semantics which is more amenible to abstraction. We will demonstrate this by abstracting to two distinct variants of k-CFA.

In our new semantics, configurations $\varsigma \in \Sigma$ are factored into *Eval* and *Apply* states. *Eval* states range over expression-context e, binding-environment ρ, value-store σ, and timestamp t components. A machine in this state is ready to evaluate a call-site e by transitioning to an *Apply* state. *Apply* states range over closure clo, evaluated argument list $(clo\ \ldots)$, store σ, and timestamp t components. A machine in this state is ready to apply its closure and make new bindings in the store, transitioning back to an *Eval* configuration. This tick-tock factoring of *Eval* and *Apply* is not essential, but will simplify presentation.

$$\varsigma \in \Sigma = Eval + Apply$$

$$Eval = \mathsf{E} \times Env \times Store \times Time$$
$$Apply = Clo \times Clo^* \times Store \times Time$$

$$\rho \in Env = \mathsf{Var} \rightharpoonup Addr$$
$$\sigma \in Store = Addr \rightharpoonup Clo$$
$$a \in Addr = \mathsf{Var} \times Time$$
$$clo \in Clo = \mathsf{Lam} \times Env$$
$$t \in Time = \mathsf{Lab}^*$$

Expression-contexts e represent the current expression under evaluation; these may be thought of as akin to program-counters in machine code. Environments ρ map variables in scope to their visible address. Stores σ map addresses to a specific value (it happens that all values are closures clo); these are essentially models of the heap. A timestamp t is a perfect program trace listing all the call-sites execution has passed through.

We define a transition function $(\Rightarrow) \subseteq \Sigma \times \Sigma$ for evaluating a machine configuration by one step. Evaluation may become stuck when a program is malformed, or if the special `(halt)` form is reached. Two pattern-matching rules of inference are used to define $(\Rightarrow)$ separately for *Eval* and *Apply* states.

$$\overbrace{(ae_f\ ae_1\ \ldots\ ae_j)^l,\ \rho,\ \sigma,\ t)}^{\varsigma} \Rightarrow (clo_f,\ (clo_1\ \ldots\ clo_j),\ \sigma,\ t')$$

$$\begin{aligned} \text{where} \quad & clo_f = \mathcal{A}(ae_f,\ \varsigma) \\ & clo_i = \mathcal{A}(ae_i,\ \varsigma) \\ & t' = l : t \end{aligned}$$

The atomic-expression in call-position ae_f is evaluated to a closure clo_f using the atomic-expression evaluator $\mathcal{A} : \mathsf{AE} \times Eval \rightharpoonup Clo$. Each argument ae_i is also evaluated to a value clo_i. The machine transitions to an *Apply* state where the closure clo_f is ready to be applied on values $clo_1 \ldots clo_j$. The timestamp t is extended with a label representing the current call-site to produce t'.

$$\overbrace{((\lambda\ (x_1\ \ldots\ x_j)\ e_\lambda)\ \rho_\lambda),\ (clo_1\ \ldots\ clo_j),\ \sigma,\ t)}^{\varsigma} \Rightarrow\ (e_\lambda,\ \rho',\ \sigma',\ t)$$
$$\begin{aligned} \text{where} \quad & \rho' = \rho_\lambda[x_i \mapsto a_i] \\ & \sigma' = \sigma[a_i \mapsto clo_i] \\ & a_i = (x_i,\ t) \end{aligned}$$

An *Apply* state is evaluated by performing the necessary bindings and transitioning execution back to an *Eval* state whose expression-context is the body of the applied closure e_λ. Each formal parameter x_i is allocated a fresh address a_i by pairing x_i with the current timestamp t. Because a_i is therefore unique, it may precisely indicate a single closure in the store. Each parameter x_i is mapped to this location a_i within the updated environment ρ'. Each location a_i is mapped to the newly bound value clo_i within the updated store σ'.

As before, since multiple cases may be required for evaluating an atomic-expression, we distingush an auxiliary function $\mathcal{A}$. In a language containing syntactic literals, these would be translated into equivalent semantic values.

$$\mathcal{A}(x,\ (e,\ \rho,\ \sigma,\ m)) = \sigma(\rho(x))$$
$$\mathcal{A}(lam,\ (e,\ \rho,\ \sigma,\ m)) = (lam,\ \rho)$$

We inject a program e into this state-space by producing an initial state $\varsigma_0 = (e,\ \varnothing,\ \varnothing,\ ())$. A system-space transfer function $f : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ may be iterated to convergence on an initial input $\varnothing$ to compute e.

$$f(s) = \{\varsigma' : \varsigma \in s \text{ and } \varsigma \Rightarrow \varsigma'\} \cup \{\varsigma_0\}$$

Naturally, iteration of f may or may not converge on a fix-point.

3.2 *k*-CFA (as originally implemented)

A k-CFA analysis simply approximates all timestamps as an abridged history of the last k call-sites execution passed through. Because t no longer grows at every *Eval* state, allocated addresses are no longer guaranteed to be unique and a set of values may inhabit abstract program locations. Two different values bound following the same k calls will both be indicated at the same address in the store. This in-turn introduces non-determinism in the transition relation.

Ignoring typographical changes, the only substantive difference from our concrete state-space is that $\widehat{Time}$ and $\widehat{Addr}$ are of bounded size, and at a given address the store may denote a *flow-set* of abstract values $\hat{d}$.

$$\hat{\varsigma} \in \hat{\Sigma} = \widehat{Eval} + \widehat{Apply} \qquad \hat{\rho} \in \widehat{Env} = \mathsf{Var} \rightharpoonup \widehat{Addr}$$
$$\widehat{Eval} = \mathsf{E} \times \widehat{Env} \times \widehat{Store} \times \widehat{Time} \qquad \hat{\sigma} \in \widehat{Store} = \widehat{Addr} \to \hat{D}$$
$$\widehat{Apply} = \widehat{Clo} \times \hat{D}^* \times \widehat{Store} \times \widehat{Time} \qquad \hat{a} \in \widehat{Addr} = \mathsf{Var} \times \widehat{Time}$$
$$\hat{d} \in \hat{D} = \mathcal{P}(\widehat{Clo})$$
$$\hat{t} \in \widehat{Time} = \mathsf{Lab}^k \qquad \widehat{clo} \in \widehat{Clo} = \mathsf{Lam} \times \widehat{Env}$$

A transition relation $(\hat{\leadsto}) \subseteq \hat{\Sigma} \times \hat{\Sigma}$ matches up predecessors and successors in the state-space.

$$\overbrace{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \hat{\rho},\ \hat{\sigma},\ (l_1\ \ldots\ l_k))}^{\hat{\varsigma}} \hat{\leadsto}\ (\widehat{clo}_f,\ (\hat{d}_1\ \ldots\ \hat{d}_j),\ \hat{\sigma},\ \hat{t}')$$

$$\begin{aligned} \text{where} \quad & \widehat{clo}_f \in \hat{\mathcal{A}}(ae_f,\ \hat{\varsigma}) \\ & \hat{d}_i = \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma}) \\ & \hat{t}' = (l\ l_1\ \ldots\ l_{k-1}) \end{aligned}$$

$\widehat{Eval} \hat{\leadsto} \widehat{Apply}$ transitions are non-deterministic when multiple closures are indicated for the expression ae_f in call-position. An $\widehat{Apply}$ state results for each such closure. The one further change to this rule is that only an approximate timestamp of k call-sites is being maintained.

$$\overbrace{(((\lambda\ (x_1\ \ldots\ x_j)\ e_\lambda),\ \hat{\rho}_\lambda),\ (\hat{d}_1\ \ldots\ \hat{d}_j),\ \hat{\sigma},\ \hat{t})}^{\hat{\varsigma}} \hat{\leadsto}\ (e_\lambda,\ \hat{\rho}',\ \hat{\sigma}',\ \hat{t})$$

$$\begin{aligned} \text{where} \quad & \hat{\rho}' = \hat{\rho}_\lambda[x_i \mapsto \hat{a}_i] \\ & \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{d}_i] \\ & \hat{a}_i = (x_i,\ \hat{t}) \end{aligned}$$

$\widehat{Apply}$ states transition deterministically to an $\widehat{Eval}$ state by making the necessary bindings in $\hat{\rho}'$ and $\hat{\sigma}'$ and moving control inside the body e_λ of the applied function. Because a state may not be solely responsible for binding these addresses, weak-update ($\sqcup$) is used on the store. Join may be defined as the natural point-wise lifting, i.e. $(\hat{\sigma}_1 \sqcup \hat{\sigma}_2)(\hat{a}) = \hat{\sigma}_1(\hat{a}) \cup \hat{\sigma}_2(\hat{a})$.

Atomic-expression evaluation returns a flow-set. Closure-creation always produces a singleton.

$$\begin{aligned} \hat{\mathcal{A}} &: \mathsf{AE} \times \widehat{Eval} \rightharpoonup \hat{D} \\ \hat{\mathcal{A}}(x,\ (e,\ \hat{\rho},\ \hat{\sigma},\ \hat{t})) &= \hat{\sigma}(\hat{\rho}(x)) \\ \hat{\mathcal{A}}(lam,\ (e,\ \hat{\rho},\ \hat{\sigma},\ \hat{t})) &= \{(lam,\ \hat{\rho})\} \end{aligned}$$

To compute the analysis we define an abstract system-space transfer function $\hat{f} : \mathcal{P}(\hat{\Sigma}) \to \mathcal{P}(\hat{\Sigma})$ which explores reachable states for a program e by extending a starting state $\hat{\varsigma}_0$ with all states immediately reachable from those in its input.

$$\hat{f}(\hat{s}) = \{\hat{\varsigma}' : \hat{\varsigma} \in \hat{s} \text{ and } \hat{\varsigma} \hat{\leadsto} \hat{\varsigma}'\} \cup \{\hat{\varsigma}_0\}$$

We define the starting state $\hat{\varsigma}_0 = (e,\ \varnothing,\ \bot,\ (l_0\ \overset{k}{\ldots}\ l_0))$ where l_0 is an otherwise unused label. Because $\hat{f}$ is monotonic, continuous, and operates over a finite lattice $(\mathcal{P}(\hat{\Sigma}), \sqsubseteq)$, we know $\hat{f}^n(\bot)$ for some n will be a fix-point of $\hat{f}$ [4, 35].

Galois-connection. To formalize the abstraction connecting our concrete semantics with k-CFA, we define a Galois-connection $\mathcal{P}(\Sigma) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}(\hat{\Sigma})$. The abstraction map for states α_Σ defines, for a given state ς, a most precise abstract state $\alpha_\Sigma(\varsigma)$. An abstract state $\hat{\varsigma}$ may be said to soundly simulate a concrete one ς if and only if $\alpha_\Sigma(\varsigma) \sqsubseteq \hat{\varsigma}$. This is also precisely the definition of $\varsigma \in \gamma_\Sigma(\hat{\varsigma})$.

$$\begin{aligned}
\alpha_\Sigma(e,\ \rho,\ \sigma,\ t) &= (e,\ \alpha_{Env}(\rho),\ \alpha_{Store}(\sigma),\ take_k(t)) \\
\alpha_\Sigma(clo_f,\ (clo_1 \ldots),\ \sigma,\ t) &= (\alpha_{Clo}(clo_f),\ (\{\alpha_{Clo}(clo_1)\} \ldots),\ \alpha_{Store}(\sigma),\ take_k(t)) \\
\alpha_{Env}(\rho) &= \{(x,\ \alpha_{Addr}(a)) : (x,\ a) \in \rho\} \\
\alpha_{Store}(\sigma) &= \bigsqcup_{(a,clo)\in\sigma} [\alpha_{Addr}(a) \mapsto \{\alpha_{Clo}(clo)\}] \\
\alpha_{Clo}(lam,\ \rho) &= (lam,\ \alpha_{Env}(\rho)) \\
\alpha_{Addr}(x,\ t) &= (x,\ take_k(t))
\end{aligned}$$

The function $take_k$ truncates a list to include at most its first k elements, padding any remaining length with l_0 (the special label used to produce $\hat{\varsigma}_0$).

To show that $\hat{f}$ is always a sound simulation of f (i.e. $\alpha \circ f \circ \gamma \sqsubseteq \hat{f}$) we require a bi-simulation proof with a base case $\alpha(\varsigma_0) \sqsubseteq \hat{\varsigma}_0$ and an inductive step:

$$\varsigma \Rightarrow \varsigma' \text{ and } \alpha_\Sigma(\varsigma) \sqsubseteq \hat{\varsigma} \implies \exists \hat{\varsigma}'.\, \hat{\varsigma} \lessapprox \hat{\varsigma}' \text{ and } \alpha_\Sigma(\varsigma') \sqsubseteq \hat{\varsigma}'$$

The defined notion of simulation is preserved across every transition [23].

3.3 *k*-CFA (as originally described)

Shivers described the call-sensitivity of k-CFA as tracking the last k call-sites; however, due to CPS-conversion, this also tracks the history of recently traversed return-points in the original program. We formalize a variation of this k-CFA which tracks only true call-sites in the direct-style program. This demonstrates the kinds of subtle differences which can exist between styles of polyvariance, and clarifies which portions of an abstract semantics are relevant to the tuning of polyvariance.

To obtain this variant of k-CFA, we need only change the definition of $\hat{t}'$ in our $\widehat{Eval} \lessapprox \widehat{Apply}$ transition:

$$\overbrace{(\texttt{(}ae_f\ ae_1\ \ldots\ ae_j\texttt{)}^l,\ \hat{\rho},\ \hat{\sigma},\ (l_1\ \ldots\ l_k))}^{\hat{\varsigma}} \lessapprox (\widehat{clo}_f,\ (\hat{d}_1\ \ldots\ \hat{d}_j),\ \hat{\sigma},\ \hat{t}')$$

$$\begin{aligned}
\text{where} \quad & \widehat{clo}_f \in \hat{\mathcal{A}}(ae_f,\ \hat{\varsigma}) \\
& \hat{d}_i = \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma}) \\
& \hat{t}' = \begin{cases} (l\ l_1\ \ldots\ l_{k-1}) & \neg\mathcal{R}(l) \\ (l_1\ \ldots\ l_k) & \mathcal{R}(l) \end{cases}
\end{aligned}$$

We use the predicate $\mathcal{R}$ (introduced at the start of §3) to distinguish between true call-sites and those which represent return-points. Our analysis of execution

history is extended only at a true call-site. This in-turn empowers an allocation of addresses $\hat{a}_i = (x_i, \hat{t})$ which merges values in the store only when they share the last k call-sites, regardless of whether they share previous return-points.

Galois-connection. As above we may formalize this analysis as a Galois-connection, making the following changes:

$$\begin{aligned}
\alpha_\Sigma(e,\ \rho,\ \sigma,\ t) &= (e,\ \alpha_{Env}(\rho),\ \alpha_{Store}(\sigma),\ takecalls_k(t)) \\
\alpha_\Sigma(clo_f,\ (clo_1 \ldots),\ \sigma,\ t) &= (\alpha_{Clo}(clo_f),\ (\{\alpha_{Clo}(clo_1)\} \ldots), \\
&\qquad \alpha_{Store}(\sigma),\ takecalls_k(t)) \\
&\ldots \\
\alpha_{Addr}(x,\ t) &= (x,\ takecalls_k(t))
\end{aligned}$$

In this case $takecalls_k$ removes the first at most k true call-sites from its input, using $\mathcal{R}$ to distinguish them from return-points, and pads any remaining space with l_0 (the special label we used to produce $\hat{\varsigma}_0$).

3.4 Comparison

To further clarify where these analyses differ and to show that the precision of a particular style of polyvariance is dependent on the specific program being modeled and the properties of interest, we give two short examples. The first is better served by the analysis of §3.2 and the second by the analysis of §3.3:

Example 1. In a 2-call+return style analysis, bindings for `v` will be differentiated into two variants. One representing precisely `#t` and the other `#f`, based on whether the last return-point was in `g1` or `g2` respectively.

```
(let* ([push2 (lambda () ((lambda (a) ((lambda (b) b) a))
                          (void)))]
       [f (lambda (g) (let ([v (g)]) v))]
       [g1 (lambda () (push2) #t)]
       [g2 (lambda () (push2) #f)])
      (f g1)
      (f g2))
```

As the last two call-sites along either path are those in the body of `push2`, a 2-call-only style of analysis will produce a single shared address for `v`.

Example 2. Here, a 2-call-only analysis will differentiate both variants of `v` because the second to last call-site is still unique to a specific invocation of `f`.

```
(let ([id (lambda (x) x)]
      [f (lambda (g) (let ([v (g)]) v))])
     (f (lambda () (id #t)))
     (f (lambda () (id #f))))
```

In a 2-call+return style of analysis however, the intervening return from `id` causes merging between `#t` and `#f` in `v`. This is because the call-site `(g)` in `f` and the return-point `x` in `id` are passed through along both paths.

4 Unified Semantics

In each analysis from the previous section, the allocation of abstract addresses is the proximate cause of the specific polyvariance it exhibits. In both cases, allocation is performed by tracking a meta-analysis of recent call-stack behavior and pairing it with the variable being bound to produce an address. Without a meta-analysis that empowers allocation to differentiate addresses based on either recent call-sites or recent return-points, neither style of polyvariance would be possible by allocation alone. Other styles of polyvariance may require a very different kind of meta-analysis, or may require no meta-analysis at all.

Apart from typographical changes, our unified semantics is nearly equivalent to the abstract semantics in §3.2 (or §3.3). We generalize over allocation and meta-analysis by making both opaque parameters to our framework which tune its polyvariance.

$$\begin{aligned}
\tilde{\varsigma} \in \tilde{\Sigma} &= \widetilde{Eval} + \widetilde{Apply} & \tilde{\rho} \in \widetilde{Env} &= \mathsf{Var} \rightharpoonup \widetilde{Addr} \\
\widetilde{Eval} &= \mathsf{E} \times \widetilde{Env} \times \widetilde{Store} \times \widetilde{Meta} & \tilde{\sigma} \in \widetilde{Store} &= \widetilde{Addr} \to \tilde{D} \\
\widetilde{Apply} &= \widetilde{Clo} \times \tilde{D}^* \times \widetilde{Store} \times \widetilde{Meta} & \tilde{a} \in \widetilde{Addr} &= \mathsf{Var} \times \widetilde{Meta} \\
& & \tilde{d} \in \tilde{D} &= \mathcal{P}(\widetilde{Clo}) \\
\tilde{m} \in \widetilde{Meta} &= \mathsf{Lab}^* & \widetilde{clo} \in \widetilde{Clo} &= \mathsf{Lam} \times \widetilde{Env}
\end{aligned}$$

Unless otherwise stated, we assume a meta-analysis $\tilde{m}$ is still some list of syntax labels. In general, no restrictions need be placed on the definition of $\widetilde{Meta}$ to ensure correctness of the core flow-analysis. This is expanded upon in §4.4.

Evaluation of atomic-expressions is performed by an auxiliary function $\tilde{\mathcal{A}}$. Lambdas and variable references are handled as before. In a language containing syntactic literals, these would be translated into equivalent semantic values.

$$\begin{aligned}
\tilde{\mathcal{A}} &: \mathsf{AE} \times \widetilde{Eval} \rightharpoonup \tilde{D} \\
\tilde{\mathcal{A}}(x,\ (e,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{m})) &= \tilde{\sigma}(\tilde{\rho}(x)) \\
\tilde{\mathcal{A}}(lam,\ (e,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{m})) &= \{(lam,\ \tilde{\rho})\}
\end{aligned}$$

A transition relation $(\rightsquigarrow) \subseteq \tilde{\Sigma} \times \tilde{\Sigma}$ matches up predecessors and successors in the unified state-space. This is defined by two pattern-matching rules:

$$\overbrace{(\mathtt{(}ae_f\ ae_1\ \ldots\ ae_j\mathtt{)},\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{m})}^{\tilde{\varsigma}} \rightsquigarrow (\widetilde{clo}_f,\ (\tilde{d}_1\ \ldots\ \tilde{d}_j),\ \tilde{\sigma},\ \tilde{m}')$$

$$\begin{aligned}
\text{where} \quad & \widetilde{clo}_f \in \tilde{\mathcal{A}}(ae_f,\ \tilde{\varsigma}) \\
& \tilde{d}_i = \tilde{\mathcal{A}}(ae_i,\ \tilde{\varsigma}) \\
& \tilde{m}' = \tilde{\mathcal{M}}(\tilde{\varsigma})
\end{aligned}$$

An analysis parameter $\tilde{\mathcal{M}} : \tilde{\Sigma} \to \tilde{M}$ is now soley responsible for transitioning the meta-analysis component of any state.

$$\overbrace{((\texttt{(}\lambda\ \texttt{(}x_1\ \ldots\ x_j\texttt{)}\ e_\lambda\texttt{)},\ \tilde{\rho}_\lambda),\ (\tilde{d}_1\ \ldots\ \tilde{d}_j),\ \tilde{\sigma},\ \tilde{m})}^{\tilde{\varsigma}} \rightsquigarrow (e_\lambda,\ \tilde{\rho}',\ \tilde{\sigma}',\ \tilde{m}')$$

$$\begin{aligned} \text{where} \quad & \tilde{\rho}' = \tilde{\rho}_\lambda[x_i \mapsto \tilde{a}_i] \\ & \tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a}_i \mapsto \tilde{d}_i] \\ & \tilde{a}_i = \widetilde{alloc}(x_i,\ \tilde{\varsigma}) \\ & \tilde{m}' = \tilde{\mathcal{M}}(\tilde{\varsigma}) \end{aligned}$$

A second analysis parameter $\widetilde{alloc} : \mathsf{Var} \times \widetilde{Apply} \to \widetilde{Addr}$ may be tuned to determine how the semantics allocates addresses for variables as they're bound.

4.1 Tuning for concrete semantics

To obtain a concrete semantics in this framework, we use a simple meta-analysis which records every expression execution passes through. The allocator then uses this history to produce a fresh address for every binding. No two addresses will be the same because $\tilde{m}$ is a perfect program trace, extended at every expression. As every flow-set is therefore a singleton, all transitions are deterministic.

$$\widetilde{alloc}_\bot : \mathsf{Var} \times \widetilde{Apply} \to \widetilde{Addr}$$
$$\widetilde{alloc}_\bot(x,\ (\widetilde{clo}_f,\ (\ldots),\ \tilde{\sigma},\ \tilde{m})) = (x, \tilde{m})$$

$$\tilde{\mathcal{M}} : \tilde{\Sigma} \to \tilde{M}$$
$$\tilde{\mathcal{M}}(e^l,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{m}) = l : \tilde{m}$$
$$\tilde{\mathcal{M}}(\widetilde{clo}_f,\ (\ldots),\ \tilde{\sigma},\ \tilde{m}) = \tilde{m}$$

Evaluating a program. To evaluate a program e using this semantics, we first define a state-injection function $\tilde{\mathcal{I}} : \mathsf{E} \to \tilde{\Sigma}$ and an initial state $\tilde{\varsigma}_0 = \tilde{\mathcal{I}}(e)$.

$$\tilde{\mathcal{I}}(e) = (e,\ \varnothing,\ \bot,\ ())$$

It is then possible to compute the transitive closure of $(\rightsquigarrow)$ starting from $\tilde{\varsigma}_0$. We may define a system-space transfer function $\tilde{f} : \mathcal{P}(\tilde{\Sigma}) \to \mathcal{P}(\tilde{\Sigma})$ which accumulates $\tilde{\varsigma}_0$ with all states immediately reachable from states in its input.

$$\tilde{f}(\tilde{s}) = \{\tilde{\varsigma}' : \tilde{\varsigma} \in \tilde{s} \text{ and } \tilde{\varsigma} \rightsquigarrow \tilde{\varsigma}'\} \cup \{\tilde{\varsigma}_0\}$$

Naturally, concrete executions are incomputable in the general case. Attempting to iterate $\tilde{f}$ to a fix-point may or may not eventually converge.

4.2 Tuning for univariance and monovariance

As our approach to semantics focuses the unboundedness of a machine's state-space exclusively on addresses in the store, a properly defined abstract address

allocator $\widetilde{alloc}$ is the one essential component required to obtain a bounded approximation. The semantics is guaranteed to be computable so long as $\widetilde{alloc}$ is only permitted to produce a bounded number of addresses. This in turn introduces merging between values in the store and non-determinism in the transition relation. The most extreme version of this is the *univariant* allocator which only produces a single address $\top$ that over-approximates all concrete addresses.

$$\widetilde{alloc_\top} : \mathsf{Var} \times \widetilde{Apply} \to \widetilde{Addr}^\top \qquad \tilde{\mathcal{M}} : \tilde{\Sigma} \to \tilde{M}$$
$$\widetilde{alloc_\top}(x,\ \tilde{\varsigma}) = \top \qquad \tilde{\mathcal{M}}(\tilde{\varsigma}) = ()$$

The monovariant allocator is more precise and tunes our framework to perform a 0-CFA analysis. This allocator produces a single address for each variable.

$$\widetilde{alloc} : \mathsf{Var} \times \widetilde{Apply} \to \widetilde{Addr}$$
$$\widetilde{alloc}(x,\ \tilde{\varsigma}) = (x, ())$$

A flow-set for an address $\tilde{a}$ indicates a range of values which over-approximates all possible concrete values that can flow to any concrete address approximated by $\tilde{a}$. For example, if a concrete machine binds $(y, (l_{14} \ldots)) \mapsto \{\widetilde{clo}_1\}$ and $(y, (l_{38} \ldots)) \mapsto \{\widetilde{clo}_2\}$, its monovariant approximation might bind $(y, ()) \mapsto \{\widetilde{clo}_1, \widetilde{clo}_2\}$. Precision is lost for $(y, (l_{14} \ldots))$ both because its value has been merged with $\widetilde{clo}_2$, and because the environments for $\widetilde{clo}_1$ and $\widetilde{clo}_2$ in-turn generalize over many possible addresses for their free variables.

Naïvely computing 0-CFA. $\tilde{f}$ is monotonic and continuous. When the address space is finite, it operates over a finite lattice $\mathcal{P}(\tilde{\Sigma})$ and we therefore know that $\tilde{f}^n(\bot)$ for some n will be the least-fixed-point of $\tilde{f}$ [4, 35]. Unfortunately, the worst-case complexity of this calculation is exponential as the total number of stores which may need to be explored is in $O(2^{n^2})$ [27].

Efficiently computing 0-CFA. *Store-widening* is an essential technique for combating this complexity. It relaxes the relationship between machine configurations and the store in order to explore fewer stores during an analysis. In its coarsest form, *global store-widening* maintains a single store for the entire system-space. This approximation tends to produce significantly faster analyses with only slightly decreased precision in practice [23, 33]. With bit-packing implementation tricks, this approximation of 0-CFA is in $O(\frac{n^3}{log\ n})$ [22].

We may formalize this technique as a widening operator $\triangledown : \mathcal{P}(\tilde{\Sigma}) \to \mathcal{P}(\tilde{\Sigma})$, applied at each transition to speed convergence within the naïve state-space:

$$\begin{aligned}\triangledown(\tilde{s}) = \{&(e, \tilde{\rho}, \tilde{\sigma}, \tilde{m}) : (e, \tilde{\rho}, \_, \tilde{m}) \in \tilde{s} \text{ and } (\_, \_, \tilde{\sigma}, \_) \in \tilde{s}\} \\ &\cup \{(\widetilde{clo}_f, (\tilde{d}_1 \ldots), \tilde{\sigma}, \tilde{m}) : (\widetilde{clo}_f, (\tilde{d}_1 \ldots), \_, \tilde{m}) \in \tilde{s} \text{ and } (\_, \_, \tilde{\sigma}, \_) \in \tilde{s}\}\end{aligned}$$

The notation $\_$ is a wildcard used to match any value.

Equivalently, we may lift the store out of machine configurations entirely and compute an inherently widened system-space $\tilde{\xi}$ which pairs a single store with a set of reachable configurations.

$$\tilde{\xi} \in \tilde{\Xi} = \tilde{R} \times \widetilde{Store} \qquad \tilde{r} \in \tilde{R} = \mathcal{P}(\mathsf{E} \times \widetilde{Env} \times \widetilde{Meta} + \widetilde{Clo} \times \tilde{D}^* \times \widetilde{Meta})$$

Its transfer function $\tilde{f}_\triangledown$ (effectively $\triangledown \circ \tilde{f} \circ \triangledown$) uses the global store to transition each configuration and computes a new store as the least-upper-bound of all stores at this frontier.

$$\begin{aligned}
&\tilde{f}_\triangledown : \tilde{\Xi} \to \tilde{\Xi} \\
&\tilde{f}_\triangledown(\tilde{r},\ \tilde{\sigma}) = (\tilde{r}',\ \tilde{\sigma}') \sqcup \tilde{\xi}_0 \\
&\text{where} \quad \tilde{s} = \{\tilde{\varsigma}' : (e,\ \tilde{\rho},\ \tilde{m}) \in \tilde{r} \text{ and } (e,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{m}) \rightsquigarrow \tilde{\varsigma}'\} \\
&\qquad\qquad \cup \{\tilde{\varsigma}' : (\widetilde{clo},\ (\tilde{d}_1 \ldots),\ \tilde{m}) \in \tilde{r} \text{ and } (\widetilde{clo},\ (\tilde{d}_1 \ldots),\ \tilde{\sigma},\ \tilde{m}) \rightsquigarrow \tilde{\varsigma}'\} \\
&\qquad \tilde{r}' = \{(e,\ \tilde{\rho},\ \tilde{m}) : (e,\ \tilde{\rho},\ _,\ \tilde{m}) \in \tilde{s}\} \\
&\qquad\qquad \cup \{(\widetilde{clo},\ (\tilde{d}_1 \ldots),\ \tilde{m}) : (\widetilde{clo},\ (\tilde{d}_1 \ldots),\ _,\ \tilde{m}) \in \tilde{s}\} \\
&\qquad \tilde{\sigma}' = \bigsqcup_{(_,_,\tilde{\sigma},_) \in \tilde{s}} \tilde{\sigma}
\end{aligned}$$

For a program e, the starting state is defined $\tilde{\xi}_0 = (\{(e,\ \varnothing,\ ())\},\ \bot)$.

4.3 Extension to real languages

Setting up a unified semantics for real language features such as conditionals, primitive operations, direct-style recursion, or exceptions, is no more difficult, if more verbose. Supporting direct-style recursion, for example, requires a big-step semantics or an explicit stack as continuations are no longer baked into the source text by CPS-conversion. This can be done by modeling a stack within each concrete state and store-allocating continuation frames on abstraction to obtain a finite-state model [15]. A more precise option is to directly abstract a pushdown machine, labeling edges in the system with stack actions to obtain a Dyck state-graph [6, 7, 36]. Johnson and Van Horn showed that when continuations are store-allocated, pushdown-CFA itself corresponds to polyvariant stack allocation [18]. When CPS-converting, continuations can naturally inherit whatever style of polyvariance is used for all other bindings, or they may be treated specially as determined by the allocator.

Handling additional syntactic forms is often as straightforward as including an additional transition rule for each. Consider inclusion of a `set!` form enabling side-effects:

$$\begin{aligned}
e \in \mathsf{E} ::= &\ (\texttt{set!}\ x\ ae_v\ ae_\kappa) \\
\mid &\ \ldots
\end{aligned}$$

We simply extend the definition of $(\rightsquigarrow)$ to include a rule:

$$\overbrace{(\texttt{(set!}\ x\ ae_v\ ae_\kappa\texttt{)},\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{m})}^{\tilde{\varsigma}} \rightsquigarrow (\widetilde{clo}_\kappa,\ (\{\texttt{void}\}),\ \tilde{\sigma}',\ \tilde{m}')$$

$$\begin{aligned} \text{where} \quad & \widetilde{clo}_\kappa \in \tilde{\mathcal{A}}(ae_\kappa,\ \tilde{\varsigma}) \\ & \tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{\rho}(x) \mapsto \tilde{\mathcal{A}}(ae_v, \tilde{\varsigma})] \\ & \tilde{m}' = \tilde{\mathcal{M}}(\tilde{\varsigma}) \end{aligned}$$

4.4 Parametricity

Crucially, as meta-analyses $\tilde{\mathcal{M}}$ strictly increase information at any given state, they may be tuned *arbitrarily* without affecting the soundness of our core flow-analysis. Allocators do impact the core flow-analysis, but may also be tuned arbitrarily because the *a posteriori* soundness proof [25] ensures a consistent abstraction for addresses will always result (§2.3). Meta-analyses used only to benefit the range of allocators we can express are not restricted by any notion of soundness. By including this additional information, we expand the set of distinct allocators we can express, enabling arbitrary polyvariance. Other aspects of our framework may also be changed so long as they respect soundness. In §5.3 for example, we modify $\tilde{\mathcal{A}}$ in a way that strictly expands on the approximation we use for closures. Soundness is respected as no information is removed.

Composing meta-analyses. For simplicity, in no section do we implement more than a single variation of $\widetilde{Meta}$ at a time. Naturally in practice it may be desirable to combine several of these tunings in one. This can be done by using the Cartesian product of each, i.e. $\tilde{\mathcal{M}}(\ldots, (\tilde{m}_1, \tilde{m}_2)) = (\tilde{\mathcal{M}}_1(\ldots, \tilde{m}_1), \tilde{\mathcal{M}}_2(\ldots, \tilde{m}_2))$. Where these analyses are permitted to introspect on one another, their combination may be more precise than each run in isolation.

Constructing analyses from an allocator. Given definitions for $\widetilde{Meta}$ and $\tilde{\mathcal{M}}$ along with any other acceptable (sound) changes to our semantics, we may define a set of addresses $\widetilde{Addr}$ arbitrarily. This gives rise to a set of allocators:

$$\widetilde{alloc} \in \widetilde{Alloc} = \mathsf{Var} \times \widetilde{Apply} \to \widetilde{Addr}$$

As we expand the meta-analysis and address-space used, $\widetilde{Alloc}$ grows without restriction. To complete the framework, we define functions CFA and $CFA_\triangledown$ which construct transfer functions $\tilde{f}$ and $\tilde{f}_\triangledown$ given a specific choice of allocator:

$$CFA : \widetilde{Alloc} \to \mathcal{P}(\tilde{\Sigma}) \to \mathcal{P}(\tilde{\Sigma})$$
$$CFA_\triangledown : \widetilde{Alloc} \to \tilde{\Xi} \to \tilde{\Xi}$$

All possible tunings $CFA(\widetilde{alloc})$ soundly over-approximate $CFA(\widetilde{alloc}_\bot)$.

5 Strategies for Polyvariance

We may now survey a number of existing flavors of polyvariance and show how they are easily formalized within our framework. *Every* additional form of polyvariance (context-sensitivity, path-sensitivity, etc.) has an encoding as well.

5.1 Call-sensitivity

To implement call-sensitivity (k-CFA), addresses include a list of call-site labels.

$$\widetilde{Addr} = \mathsf{Var} \times \widetilde{Meta} \qquad\qquad \widetilde{Meta} = \mathsf{Lab}^k$$

As execution passes through a call-site, it is saved and the oldest callsite dropped.

$$\tilde{\mathcal{M}}(e^l,\ \tilde{\rho},\ \tilde{\sigma},\ (l_1\ \ldots\ l_k)) = (l\ l_1\ \ldots\ l_{k-1})$$
$$\tilde{\mathcal{M}}(\widetilde{clo}_f,\ (\ldots),\ \tilde{\sigma},\ \tilde{m}) = \tilde{m}$$

This running approximation of call-history is then used for allocating addresses.

$$\widetilde{alloc}_{call}(x,\ (\widetilde{clo}_f,\ (\ldots),\ \tilde{\sigma},\ \tilde{m})) = (x, \tilde{m})$$

This tuning instantiates our framework to the traditional k-CFA analysis in §3.2.

5.2 Variable call-sensitivity (polymorphic-splitting)

Wright et al.'s polymorphic-splitting is a form of adaptive call-sensitivity inspired by let-polymorphism where the degree of polyvariance may vary between functions [37]. The number of let definitions a lambda was originally defined within (before CPS-conversion) forms a simple heuristic for its call-sensitivity when invoked. To implement call-sensitivity with a per-function k, we assume an auxiliary function $\mathcal{L}$ for encoding let-depth (or any other heuristic).

$$\widetilde{Addr} = \mathsf{Var} \times \widetilde{Meta} \qquad\qquad \widetilde{Meta} = \mathsf{Lab}^* \qquad\qquad \mathcal{L} : \mathsf{Lam} \to \mathbb{N}$$

Call-histories are truncated on a per-function basis as defined by $\mathcal{L}$.

$$\tilde{\mathcal{M}}(e^l,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{m}) = l : \tilde{m}$$
$$\tilde{\mathcal{M}}((lam,\ \tilde{\rho}_\lambda),\ (\ldots),\ \tilde{\sigma},\ \tilde{m}) = \begin{cases} (l_1\ \ldots\ l_k) & \tilde{m} = (l_1\ \ldots\ l_k\ \ldots\ l_j) \\ & \text{and } \mathcal{L}(lam) = k \\ \tilde{m} & \text{otherwise} \end{cases}$$

This call-history is then used for allocating addresses.

$$\widetilde{alloc}_{let}(x,\ (\widetilde{clo}_f,\ (\ldots),\ \tilde{\sigma},\ \tilde{m})) = (x, \tilde{m})$$

All possible heuristics $\mathcal{L}$ are sound. Because no tuning of $\mathcal{L}$ can produce an infinite k (in the case of polymorphic-splitting because no program can contain an infinite nesting of let forms) the image of $\widetilde{alloc}_{let}$ must be finite.

5.3 Closure-sensitivity (object-sensitivity)

Smaragdakis et al. [34] distinguish multiple variants of object-sensitivity, first described by Milanova et al. [28]. This style of context uses a history of the allocation-points for objects to guide polyvariance. We define an equivalent to the more precise variant of object-sensitivity: k-full-object-sensitivity.

In higher-order languages this style would track the point of closure-creation for a lambda, the point of closure-creation for the lambda that allocated it, and so forth.

$$\widetilde{Clo} = \mathsf{Lam} \times \widetilde{Env} \times \widetilde{Meta} \qquad \widetilde{Meta} = \mathsf{Lab}^k$$
$$\widetilde{Addr} = \mathsf{Var} \times \widetilde{Meta}$$

Closures each extend an individual history with their allocation-point.

$$\tilde{\mathcal{A}}(lam^l,\ (e,\ \tilde{\rho},\ \tilde{\sigma},\ (l_1\ \ldots\ l_k))) = \{(lam,\ \tilde{\rho},\ (l\ l_1\ \ldots\ l_{k-1}))\}$$

At each application, we begin using the allocation history of the invoked closure.

$$\tilde{\mathcal{M}}(e,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{m}) = \tilde{m}$$
$$\tilde{\mathcal{M}}((lam,\ \tilde{\rho},\ \tilde{m}_\lambda),\ (\ldots),\ \tilde{\sigma},\ \tilde{m}) = \tilde{m}_\lambda$$

For parameters, allocation selects an address specific to their function's history.

$$\widetilde{alloc}_{obj}(x,\ ((lam,\ \tilde{\rho},\ \tilde{m}_\lambda),\ (\ldots),\ \tilde{\sigma},\ \tilde{m})) = (x, \tilde{m}_\lambda)$$

If our language included other kinds of objects, their representation would also be extended to include the current allocation history using $\tilde{\mathcal{A}}$, just as we've done for closures. Member functions of such objects would then use this object-specific history for allocating their parameters.

The authors of k-full-object-sensitivity also define a novel approximation for it called type-sensitivity which over-approximates allocation-points by the class in which they occur.

5.4 Path-sensitivity

Path-sensitivity is used to describe a broad range of different styles of flow-sensitivity which encode the branch taken through a conditional [21]. For example, an analysis of the expression (if (> a 5) e_1 e_2) could remember along the e_1 path that $a > 5$ and along the e_2 branch that $a \leq 5$. To demonstrate a tuning for a style of path-sensitivity, we must assume an extension of our language which includes at least conditionals and boolean values.

$$e \in \mathsf{E} ::= (\text{if } x\ e\ e) \mid \ldots \qquad \tilde{d} \in \tilde{D} = \mathcal{P}(\widetilde{Value})$$
$$\tilde{v} \in \widetilde{Value} = \widetilde{Clo} + \{\texttt{\#t}, \texttt{\#f}\}$$

Next, we give a concrete semantics for `if` which demonstrates a technique which may be used to change the polyvariance of a variable which is already bound.

$$\frac{\texttt{\#f} \neq \mathcal{A}(x, \varsigma)}{\underbrace{(\texttt{(if } x\ e_1\ e_2\texttt{)},\ \rho,\ \sigma,\ m)}_{\varsigma} \Rightarrow ((\texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e_1\texttt{)},\ \rho),\ (v_1),\ \sigma,\ m')}$$

$$\text{where} \qquad v_1 = \mathcal{A}(x, \varsigma) \qquad m' = \mathcal{M}(\varsigma)$$

$$\frac{\texttt{\#f} = \mathcal{A}(x, \varsigma)}{\underbrace{(\texttt{(if } x\ e_1\ e_2\texttt{)},\ \rho,\ \sigma,\ m)}_{\varsigma} \Rightarrow ((\texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e_2\texttt{)},\ \rho),\ (\texttt{\#f}),\ \sigma,\ m')}$$

$$\text{where} \qquad m' = \mathcal{M}(\varsigma)$$

The essential observation is simply that:

`(if x e1 e2)` is equivalent to

```
(if x
  (let ([x x]) e1)
  (let ([x x]) e2))
```

By rebinding the predicate we branch on in the concrete semantics, we introduce an opportunity for increasing the polyvariance at this point. This is perfectly valid as it causes no effective change to the concrete semantics and is abstracted soundly.

$$\frac{\texttt{\#f} \neq \tilde{v} \text{ and } \tilde{v} \in \tilde{\mathcal{A}}(x, \tilde{\varsigma})}{\underbrace{(\texttt{(if } x\ e_1\ e_2\texttt{)},\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{m})}_{\tilde{\varsigma}} \rightsquigarrow ((\texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e_1\texttt{)},\ \tilde{\rho}),\ (\tilde{d}_1),\ \tilde{\sigma},\ \tilde{m}')}$$

$$\text{where} \qquad \tilde{d}_1 = \tilde{\mathcal{A}}(x, \tilde{\varsigma})\ /\ \{\texttt{\#f}\} \qquad \tilde{m}' = \tilde{\mathcal{M}}(\tilde{\varsigma})$$

$$\frac{\texttt{\#f} \in \tilde{\mathcal{A}}(x, \tilde{\varsigma})}{\underbrace{(\texttt{(if } x\ e_1\ e_2\texttt{)},\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{m})}_{\tilde{\varsigma}} \rightsquigarrow ((\texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e_2\texttt{)},\ \tilde{\rho}),\ (\{\texttt{\#f}\}),\ \tilde{\sigma},\ \tilde{m}')}$$

$$\text{where} \qquad \tilde{m}' = \tilde{\mathcal{M}}(\tilde{\varsigma})$$

We may then track a running history of branches:

$$\tilde{\mathcal{M}}(\texttt{(if } x\ e_1\ e_2\texttt{)}^l,\ \tilde{\rho},\ \tilde{\sigma},\ (l_1\ \ldots\ l_k)) = (l\ l_1\ \ldots\ l_{k-1})$$
$$\tilde{\mathcal{M}}(e^l,\ \tilde{\rho},\ \tilde{\sigma},\ (l_1\ \ldots\ l_k)) = (l_1\ \ldots\ l_k)$$
$$\tilde{\mathcal{M}}(\widetilde{clo}_f,\ (\ldots),\ \tilde{\sigma},\ \tilde{m}) = \tilde{m}$$

This running approximation of path-history is then used for allocating addresses.

$$\widetilde{alloc}_{path}(x,\ (\widetilde{clo}_f,\ (\ldots),\ \tilde{\sigma},\ \tilde{m})) = (x,\ \tilde{m})$$

This would differentiate all variables by the path on which they were bound. If we wanted to only differentiate our new re-bindings of predicates, we could allocate monovariant addresses for all others. The above formalization is also an inter-procedural path-sensitivity. Recovering a traditional intra-procedural path-sensitivity requires only that we cut these histories short at a true function call. At a call-site we revert to an empty meta-analysis filled with the l_0 label.

$$\tilde{\mathcal{M}}((ae_f\ ae_1\ \ldots),\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{m}) = (l_0\ \dot{.}^{k}.\ l_0)$$

5.5 Argument-sensitivity (CPA)

Agesen's CPA directly differentiates flows to a function by the entire tuple of types being passed as arguments [1]. For a closure, we may assume its type is unique to its syntactic lambda even if unknown before analysis [9].

$$\widetilde{Addr} = \mathsf{Var} \times \mathcal{P}(\mathsf{Lam})^*$$
$$\widetilde{alloc}_{cpa}(x,\ (\widetilde{clo}_f,\ (\tilde{d}_1\ \ldots\ \tilde{d}_j),\ \tilde{\sigma},\ \tilde{m})) = (x, (\mathcal{T}(\tilde{d}_1)\ \ldots\ \mathcal{T}(\tilde{d}_j)))$$

We use a *type-of* function $\mathcal{T}(\tilde{d}) = \{lam : (lam, \tilde{\rho}) \in \tilde{d}\}$ that removes environments. If environments were permitted inside addresses, it would re-introduce recursion in the state-space as environments in-turn contain addresses.

5.6 Mixing-and-matching (call-sensitivity + argument-sensitivity)

We may easily mix and match these styles of polyvariance, or invent new ones. Consider combining call-sensitivity (using $\tilde{\mathcal{M}}$ from §5.1) with an argument-sensitivity that uses only the final continuation-argument at each call-site.

$$\widetilde{Addr} = \mathsf{Var} \times (\widetilde{Meta} \times \mathcal{P}(\mathsf{Lam}))$$
$$\widetilde{alloc}_{mix}(x,\ (\widetilde{clo}_f,\ (\tilde{d}_1\ \ldots\ \tilde{d}_j),\ \tilde{\sigma},\ \tilde{m})) = (x, (\tilde{m}, \mathcal{T}(\tilde{d}_j)))$$

In general we may use the product of allocators to obtain an allocator with the precision of each, one where two concrete addresses only abstract to the same abstract address when they do in every original allocator. Put another way, the product of two allocators represents their greatest-lower-bound in the lattice of partitionings of *Addr* [11].

6 Incremental Refinement

As the *a posteriori* soundness theorem has permitted us to tune allocation without restriction, we may use an arbitrary strategy for making an analysis directly

precision-sensitive while the fix-point computation is still live. However, because an incomplete analysis makes no strict guarantees, we don't have reliable information on which to base such precision-sensitivity. A more complete framework for arbitrary polyvariance therefore must permit the refinement of a sound fix-point based on strong guarantees made for analysis behavior.

6.1 Naïve refinement

Given a sound fixpoint $\tilde{s}$ for $CFA(\widetilde{alloc})$, we may produce an incrementally improved $\widetilde{alloc}'$ and attempt to reach a more precise fix-point by iteratively applying $CFA(\widetilde{alloc}')$ on $\tilde{s}$. Because monotonicity of the transfer function means

$$\tilde{s} \sqsubseteq \tilde{s}' \implies CFA(\widetilde{alloc})(\tilde{s}) \sqsubseteq CFA(\widetilde{alloc})(\tilde{s}')\,,$$

and not simply that

$$\tilde{s} \sqsubseteq CFA(\widetilde{alloc})(\tilde{s})\,,$$

it is possible for entire unreachable states to be removed one by one until a new fix-point is reached. It is also possible however, since $CFA(\widetilde{alloc})(\tilde{s}) \sqsubseteq \tilde{s}$ can also not be relied on to hold inductively, that additions and subtractions to the system-space may be made simultaneously such that a newly unreachable loop is explored ad infinitum. Alternatively, a fix-point could be reached where unreachable loops remain by virtue of supporting themselves. The solution to both these issues is a strictly culling transfer function $CFA_{[\text{-}]}$, which allows us to order any removal of states before any addition of new states.

$$CFA_{[\text{-}]}(\widetilde{alloc})(\tilde{s}) = CFA(\widetilde{alloc})(\tilde{s}) \cap \tilde{s}$$

We iterate the strictly culling transfer function to an intermediate *unsound* fix-point before iterating the original transfer function to an improved *sound* fix-point. We know $CFA(\widetilde{alloc})^m(CFA_{[\text{-}]}(\widetilde{alloc})^n(\tilde{s}))$ is a sound fix-point for some finite n and m because the intermediate unsound fix-point $\tilde{s}_{[\text{-}]}$ for $CFA_{[\text{-}]}$ recovers the monotonicity condition $\tilde{s}_{[\text{-}]} \sqsubseteq \tilde{s}_{[\text{-}]}$ which is then maintained inductively by $CFA(\widetilde{alloc})$ as it was when calculating the original fix-point pre-refinement.

6.2 Impact of store-widening

This algorithm for refinement of a naïve system-space is almost entirely ineffective on a store-widened system-space. It might at first appear this is because we over-approximate multiple stores using their least-upper-bound, but this is not directly the reason. More accurately, we are able to refine a naïve system-space because each transfer function is in charge of producing or culling entire stores and is therefore exclusively responsible for all the bindings it makes and all the bindings it no longer makes due to a change in the allocator. The store-widened transfer function would, in fact, allow refinement if a store were permitted to decrease across transition; however, this is not the case because a given transition is not able to ensure that it is exclusively responsible for its set of bindings.

6.3 Analysis analysis

In §5, $\widetilde{Meta}$ found many uses tracking different flavors of program context thereby enabling a greater variety of allocators. There are also many other potential extensions to an analysis, including those which aim to model and approximate properties of the analysis at analysis-time instead of properties of the program at run-time. We elucidate the potential importance of such *analysis analyses* and give an example, motivating our generalized solution for refinement.

Abstract-counting. One such analysis analysis is *abstract-counting* [23, 26]. Abstract-counts are a meta-analysis inspired by Hudak's abstract reference-counting which over-approximates the number of references to an object so compiler optimizations may be applied for updating data in-place instead of requiring a fresh allocation [16]. Abstract-counting permits essentially the same optimization to be applied at analysis-time so strong-update may be used on a flow-set (among other applications). The approximation used by a static analysis merges bindings in the store, throwing away the information necessary to permit strong-update. Stores must normally increase across transitions because no specific transition may assume it is exclusively responsible for an abstract binding except where it is known to only represent a single concrete binding.

Abstract-counting maintains a special count-store mapping addresses to an abstract count (either 0, 1, or ∞) which over-approximates the number of concrete bindings currently live for that address. When a count for an address $\tilde{a}$ is 1, it is sound for `set!` to perform a strong-update $\tilde{\sigma}[\tilde{a} \mapsto \tilde{d}]$ on the value-store at that address. In this way, abstract-counting approximates a property of the approximation, recovering precisely the information necessary for its applications.

We may formalize this technique as an extension to our analysis:

$$\tilde{m} \in \widetilde{Meta} = \widetilde{Addr} \to \tilde{\mathbb{N}} \qquad\qquad \tilde{n} \in \tilde{\mathbb{N}} = \{0, 1, \infty\}$$

Join distributes point-wise over count-stores as it does for value-stores, and over-approximates addition for counts themselves:

$$(\tilde{m}_1 \sqcup \tilde{m}_2)(\tilde{a}) = \tilde{m}_1(\tilde{a}) \sqcup \tilde{m}_2(\tilde{a})$$

$$\begin{aligned} 0 \sqcup \tilde{n} &= \tilde{n} \sqcup 0 = \tilde{n} \\ 1 \sqcup 1 &= \infty \\ \infty \sqcup \tilde{n} &= \tilde{n} \sqcup \infty = \infty \end{aligned}$$

A count of 1 is added for each binding made during an analysis.

$$\tilde{\mathcal{M}}(e,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{m}) = \tilde{m}$$

$$\tilde{\mathcal{M}}\underbrace{(((\lambda\ (x_1\ \ldots\ x_j)\ e_\lambda)\ \tilde{\rho}_\lambda),\ (\ldots),\ \tilde{\sigma},\ \tilde{m})}_{\tilde{\varsigma}} = \tilde{m}'$$

$$\text{where} \qquad \tilde{m}' = \tilde{m} \sqcup [\tilde{a}_i \mapsto 1]$$

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

The count-store will normally be widened in step with the value-store. As opposed to meta-analyses for allocation, abstract-counting must be sound.

6.4 Contribution-analysis

To soundly enable refinement in the presence of store-widening, we apply an analysis to approximate a property of analyses we term *contribution*. Intuitively, an analysis of contribution approximates the set of $\widetilde{Apply}$ configurations contributing a particular value at a particular address. Notationally, it is cleanest to express it as a mapping at each address from contributors to flow-sets.

$$\tilde{m} \in \widetilde{Meta} = \widetilde{Addr} \to (\widetilde{Clo} \times \widetilde{D}^*) \to \tilde{D}$$

With this formulation, we may permit strong-update for the current contributor because it is the only transition responsible for its portion of the contribution-store. This decreasing operation allows the least-upper-bound of all contribution stores to also be decreased across the widened transfer function $\tilde{f}_\triangledown$.

$$\tilde{\mathcal{M}}(e,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{m}) = \tilde{m}$$
$$\tilde{\mathcal{M}}\underbrace{(\overbrace{((\lambda\ (x_1\ \ldots\ x_j)\ e_\lambda)\ \tilde{\rho}_\lambda)}^{\widetilde{clo}_f},\ (\tilde{d}_1 \ldots),\ \tilde{\sigma},\ \tilde{m})}_{\tilde{\varsigma}} = \tilde{m}'$$

where
$$\tilde{m}' = \tilde{m}[\tilde{a}_i \mapsto \tilde{m}(\tilde{a}_i)[(\widetilde{clo}_f, (\tilde{d}_1 \ldots)) \mapsto \tilde{d}_i]]$$
$$\tilde{a}_i = \widetilde{alloc}(x_i, \tilde{\varsigma})$$

Where a value no longer exists for any contributor in $\tilde{m}(\tilde{a})$, it may also be soundly removed from $\tilde{\sigma}(\tilde{a})$. Like abstract-counting, contribution-analysis must be a sound over-approximation of its concrete counterpart.

Convenient implementation. Normally, the contribution-store will be widened along with the value-store and the two may be merged. A plain flow-analysis could use a hash-set to encode flow-sets, so an alternative formulation would be to extend these to hash-maps indicating a set of contributors for each value:

$$\tilde{m} \in \widetilde{Meta} = \widetilde{Addr} \to \widetilde{Clo} \to \mathcal{P}(\widetilde{Clo} \times \widetilde{D}^*)$$

Widening contribution-analysis. Adding contribution-analysis to our global-store widened flow-analysis increases the complexity by an exponential factor. This is the same complexity as a per-configuration widened analysis; in fact, this form of widening is equivalent to an analogous analysis of *receivers*. The added complexity may be addressed by further widening. Under the new formalism it is convenient to widen contributor-sets to $\top$ after a certain fixed size max.

$$\triangledown_C : \mathcal{P}(\widetilde{Clo} \times \widetilde{D}^*) \to \mathcal{P}(\widetilde{Clo} \times \widetilde{D}^*)$$
$$\triangledown_C(\tilde{c}) = \begin{cases} \tilde{c} & |\tilde{c}| \leq max \\ \top & \text{otherwise} \end{cases}$$

This restores our original complexity bound, though it may appear to again make refinement impossible. While this is true for each widened address, in practice these are the addresses most difficult to refine and therefore we haven't harmed our ability to perform a reasonable degree of refinement in practice.

7 Future Work: Thesis Breakdown

I am now able to address my specific progress, and more carefully pin down the meaning of my proposed thesis statement, mapping out future work and valid routes to success.

Polyvariance may be soundly, efficiently, and arbitrarily refined, permitting static analyses to adapt a style more efficient for their target than existing non-introspective heuristics.

First it's important to clarify what I mean by terms like efficiently, arbitrarily, introspective/non-introspective, etc:

- I defined *polyvariance* as the use of analysis techniques differentiating syntactic program locations at runtime into multiple abstract representatives each and showed how this definition fits previous uses of the term.
- We've also seen how the range of monovariant and polyvariant analyses is precisely circumscribed by the set of all possible allocation functions, by the set of all abstraction maps for addresses, and by the set of all possible partitionings of concrete addresses. We may be sure that even an *arbitrarily* selected style of polyvariance *soundly* simulates concrete execution because the *a posteriori* soundness proof ensures an allocator will always yield a consistant abstraction-map for addresses.
- It makes sense to say a refinement step is performed *efficiently* if only the difference between two analyses need be computed. To prove this, I will be required to show that no precision already in the model needs to be recomputed, and that every change made is needed in the improved model. Computing only the difference between two abstract interpretations is maximally efficient for that refinement step, regardless of whether the total sequence of refinements or adaptations to the analysis is efficient overall. Adaptations peformed while the analysis is live, are therefore trivially "efficient".
- For two allocators, one is *more efficient* than the other, if for the same number of addresses it gets strictly better precision, or if for the same exact precision, it requires fewer addresses (a smaller model). This is complicated by the fact that precision is partially ordered. That an analysis is more precise should therefore either be shown directly against the partial order (a more precise analysis cannot be less precise at any component), or by biasing precision toward a particular application. For example, this partial order can be mapped onto a total order by instead measuring the number of opportunities for flow-directed inlining, or for constant propagation, or for the elimination of runtime type checks. At minimum I will select a practical application for the analysis and measure efficiency as the ratio of opportunities provided by the analysis to addresses required in the model.
- Finally, a strategy for polyvariance is *introspective* if it is fully able to examine the analysis while running, or after reaching a sound intermediate result, and uses information inferred by the analysis to direct allocation decisions. If any restrictions are placed on what parts of the analysis may be accessed,

or which may be used for allocation, it cannot be fully introspective. All published strategies for polyvariance are *non-introspective heuristics* for allocation because none permit arbitrary examination of the analysis and in some cases because the strategy used by the allocator is identical across all inputs. Polymorphic-splitting is a good example of an existing heuristic which dips slightly into a gray area between these polar opposites. It does vary its behavior from input to input, however it does so only by introspecting on the syntactic structure of the program and its let-forms. Because it is not designed to vary behavior based on true static analysis results, I would still call this essentially non-introspective.

A survey of strategies for polyvariance and a unified approach to encoding these in CFA are published portions of my work. The unified semantics, encoding for various forms of polyvariance, and theory for offline refinement, are all unpublished but largely completed. What remains is to formally prove the soundness and efficiency of a refinement step, explore a variety of introspective heuristics and meta-analyses, and empirically verify their effectiveness. The unified semantics and fixed strategies for polyvariance are implemented for a substantial subset of Scheme. Refinement has so-far only been implemented for pure λ-calculus. My plan is to carve out all the work done on a Unified Semantics, arbitrary polyvariance, and encodings for traditional forms of polyvariance, combining it with an empirical comparison of non-introspective and live-adaptive strategies for submission to ICFP. I will then complete my work on incremental model-refinement and aim to submit this to OOPSLA or POPL.

In the best case I would like to show that for multiple substantial Scheme benchmarks and a variety of introspective heuristics, both those which adapt polyvariance while an analysis is live and those which perform offline refinement, analyses which are fundamentally more precise than existing styles of polyvariance can be found. Ideally, I'll demonstrate analyses which are more efficient both as defined by the partial order and as defined by several standard compiler optimizations.

Frequently in an abstract interpretation, a crucial loss of precision has its source far removed from many of the effects. I suspect there will be a variety of analysis analyses which approximate this aspect of analysis behavior, tracking the source to blame for each plausibly important imprecision. Once it is explicit, such information could be used to enable cheap heuristics for guiding the refinement of polyvariance, carving a more ideal path through the lattice of polyvariant analyses than those which are described by (well-ordered) call-sensitivity, object-sensitivity, or indeed *any* fixed non-introspective strategy for polyvariance. Deciding which such meta-analyses are most effective in practice will require a careful empirical study. It may be the case that some are excellent for a particular application, yet terrible for another.

At minimum I will show that for multiple substantial Scheme benchmarks, at least one important compiler optimization, and at least one introspective

heuristic which either adapts the style of polyvariance live or through offline refinement, it is possible to obtain a more efficient analysis as measured by the ratio of opportunities for optimization enabled to addresses required.

References

[1] Ole Agesen. The cartesian product algorithm. In *Proceedings of the European Conference on Object-Oriented Programming*, page 226, 1995.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, February 2007. ISBN 052103311X.

[3] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Paris, France, 1976.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977. ACM Press, New York.

[5] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming*, Leuven, Belgium, 13-17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[6] Christopher Earl, Matthew Might, and David Van Horn. Pushdown control-flow analysis of higher-order programs: Precise, polyvariant and polynomial-time. In *Scheme Workshop*, August 2010.

[7] Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. Introspective pushdown analysis of higher-order programs. In *International Conference on Functional Programming*, pages 177–188, September 2012.

[8] ECMA. *ECMA-262 (ECMAScript Specification)*. 5.1 edition, June 2011.

[9] Thomas Gilray and Matthew Might. A survey of polyvariance in abstract interpretations. In *Proceedings of the Symposium on Trends in Functional Programming*, May 2013.

[10] Thomas Gilray and Matthew Might. A unified approach to polyvariance in abstract interpretations. In *Proceedings of the Workshop on Scheme and Functional Programming*, November 2013.

[11] George Grätzer. *General lattice theory*. Springer, 2003.

[12] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *Proceedings of the European Conference on Object-oriented Programming*, pages 126–150, Berlin, Heidelberg, 2010.

[13] Williams Ludwell Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 1989.

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

[15] David Van Horn and Matthew Might. Abstracting abstract machines. In *International Conference on Functional Programming*, page 51, Sep 2010.

[16] Paul Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 351–363. ACM, 1986.

[17] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 393–407, January 1995.

[18] J. Ian Johnson and David Van Horn. Abstracting abstract control. In *Proceedings of the ACM Symposium on Dynamic Languages*, October 2014 2014.

[19] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Symposium on principles of programming languages*, pages 66–74, 1982.

[20] Andrew Kennedy. Compiling with continuations, continued. In *Proceedings of the International Conference on Functional Programming*, pages 177–190, New York, NY, 2007. ACM.

[21] Thomas J. Marlowe, Barbara G. Ryder, and Michael G. Burke. Defining flow sensitivity in data flow problems. Technical report, IBM T. J. Watson Research Center, 1995.

[22] Jan Midtgaard. Control-flow analysis of functional programs. *ACM Computing Surveys*, 44(3):10:1–10:33, Jun 2012.

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

[24] Matthew Might. Abstract interpreters for free. In *Static Analysis Symposium*, pages 407–421, September 2010.

[25] Matthew Might and Panagiotis Manolios. A posteriori soundness for non-deterministic abstract interpretations. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 260–274, January 2009.

[26] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *ACM SIGPLAN Notices*, volume 41, pages 13–25. ACM, 2006.

[27] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *Proceedings of the International Conference on Programming Language Design and Implementation*, pages 305–315, June 2010.

[28] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering Methodology*, 14(1):1–41, January 2005.

[29] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *Journal of functional programming*, 11(03):263–317, 2001.

[30] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. In *Theoretical Computer Science 1*, pages 125–159, 1975.

[31] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, and Shriram Krishnamurthi. A tested semantics for getters, setters, and eval in javascript. In *Proceedings of the Dynamic Languages Symposium*, 2012.

[32] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. *Program flow analysis: Theory and applications*, pages 189–234, 1981.

[33] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1991.

[34] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhotak. Pick your contexts well: Understanding object-sensitivity. In *Symposium on Principles of Programming Languages*, pages 17–30, January 2011.

[35] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[36] Dimitrios Vardoulakis and Olin Shivers. CFA2: a context-free approach to control-flow analysis. In *Proceedings of the European Symposium on Programming*, volume 6012, LNCS, pages 570–589, 2010.

[37] Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: An effective polyvariant flow analysis. In *Proceedings of the ACM Transactions on Programming Languages and Systems*, pages 166–207, January 1998.