Allocation Characterizes Polyvariance

A Unified Methodology for Polyvariant Control-Flow Analysis

Thomas Gilray Michael D. Adams Matthew Might
University of Utah, USA
{tgilray,adamsmd,might}@cs.utah.edu

Abstract

The polyvariance of a static analysis is the degree to which it structurally differentiates approximations of program values. Polyvariant techniques come in a number of different flavors that represent alternative heuristics for managing the trade-off an analysis strikes between precision and complexity. For example, call sensitivity supposes that values will tend to correlate with recent call sites, object sensitivity supposes that values will correlate with the allocation points of related objects, the Cartesian product algorithm supposes correlations between the values of arguments to the same function, and so forth.

In this paper, we describe a unified methodology for implementing and understanding polyvariance in a higher-order setting (i.e., for control-flow analyses). We do this by extending the method of abstracting abstract machines (AAM), a systematic approach to producing an abstract interpretation of abstract-machine semantics. AAM eliminates recursion within a language's semantics by passing around an explicit store, and thus places importance on the strategy an analysis uses for allocating *abstract* addresses within the abstract heap or store. We build on AAM by showing that the design space of possible abstract allocators exactly and uniquely corresponds to the design space of polyvariant strategies. This allows us to both unify and generalize polyvariance as tunings of a single function. Changes to the behavior of this function easily recapitulate classic styles of analysis and produce novel variations, combinations of techniques, and fundamentally new techniques.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors and Optimization

Keywords Polyvariance; Static analysis; Control-flow analysis; Abstract interpretation; Abstract allocation; Context sensitivity

1. Introduction

In the past 35 years since call-sensitive data-flow analysis was introduced by Sharir and Pnueli [51], a wide variety of both subtly and essentially distinct forms of polyvariant static analysis have been explored in the literature. The *polyvariance* of a static analysis, broadly construed, is the degree to which program values at runtime are broken into a multiplicity of distinct static approximations of their dynamic behavior. This is consistent with previous uses of the term, although the exact nature of its broad diversity of uses has not previously been well explored or formalized for any particular methodology.

For example, consider a function applied on different values across more than one call site—an identity function applied on both true and false in Racket [48]:

```
... ⁰(let ([id (lambda (x) x)])
      ¹(id #f)
      ²(id #t))
```

A *monovariant* analysis is one which maintains only a single structurally distinct approximation, a single variant, or a single flow set (in the nomenclature of flow analysis) that over-approximates the behavior of all possible values for each syntactic variable or intermediate expression. Although the variable `x` may become bound to `#f` when called from e_1 (i.e., the first call site, `¹(id #f)`) and `#t` when called from e_2 (i.e., the second call site, `²(id #t)`), a monovariant analysis will merge these values and produce only a single flow set $\{\texttt{\#t}, \texttt{\#f}\}$ for `x` (or perhaps $\{\texttt{bool}\}$, $\top$, etc., depending on the representation of abstract values and widening used).

A more *polyvariant* analysis by contrast, would allow for a larger number of distinct flow sets, a choice which has the potential to increase analysis complexity, analysis precision, or both, depending on the target of analysis. The seminal and still most widely used form of polyvariance, *k-call sensitivity*, distinguishes one context for each call trace or call history of length k that precedes a syntactic binding site. In our example above, even a 1-call sensitive analysis (e.g., Shivers' 1-CFA) would be enough to keep the values `#t` and `#f` from merging in a single flow set for `x`. A 1-call sensitive analysis produces two distinct flow sets for `x` in this program. One is unique to both `x` and the first call site e_1, the other is unique to both `x` and the second call site e_2.

A wide gamut of polyvariant techniques has been discussed in the literature [1, 2, 4, 6, 14–16, 19, 20, 22, 26, 29–33, 41–44, 51–53, 59, 60], comprised of both subtle variations and completely disparate strategies from methods and applications like type systems, abstract interpretations, and constraint-based analyses. While many of these designs and presentations share elements in common, each was designed and implemented separately with rather little work focused on unifying or connecting distinct implementations and strategies [2, 14, 53].

We present a new methodology which both unifies and generalizes the myriad strategies for polyvariance as tunings of a single function, an abstract allocator. We show that the design space of polyvariance uniquely and exactly corresponds to the design space of tunings for this function and that no possible tuning can lead to an unsound analysis. All classic flavors of polyvariance can be easily recapitulated using our methodology and we are able to derive

ICFP '16, September 18–24, 2016, Nara, Japan.

http://dx.doi.org/10.1145/2951913.2951936

novel variations by generalizing each. By proving that no tuning of allocation is unsound, and by permitting arbitrary instrumentation of a core flow analysis to guide the behavior of this function, we are able to show that *all conceivable* sound strategies for polyvariance may be implemented in a parametric abstract semantics.

1.1 Contributions

We make the following novel contributions:

1. We import the *a posteriori* soundness approach [38] to show that all possible allocators yield a sound analysis and explain why allocation uniquely allows this exceptionally liberal soundness process.
2. We formalize the meaning of the term *polyvariant* for AAM-style flow analyses in terms of possible strategies for allocation. We show how all conceivable sound strategies for polyvariance are exactly encompassed by the design space of allocators.
3. We apply this perspective to resolving an ambiguity in the original formulation of k-CFA.
4. We survey a variety of classic styles of polyvariance and show how each can be encoded as an allocator. We further illustrate how our new perspective permits each to be easily generalized or combined with other strategies.
5. We show that the design space of strategies for allocation even encompasses the store sensitivity lost through store widening and the environment sensitivity lost through closure conversion.

1.2 Outline

Section 2 introduces the wide design space of existing strategies for polyvariance and explores the central challenges in designing a suitable (effective and efficient) form of polyvariance, motivating our new approach.

Section 3 explains the main idea of our approach informally and introduces the required insights which make it possible.

Section 4 reviews AAM, formalizing a concrete semantics and an abstract semantics which approximates it. We discuss crucial concepts such as store widening and soundness.

Section 5 discusses the role of allocation within AAM, presents the *a posteriori* soundness theorem (liberalizing our previous soundness constraints to cover all tunings of allocation) and explains why allocation is uniquely suitable for this process. We generalize the framework of Section 4 to a parametric semantics that encompasses all possible allocation behaviors and explain the importance of leaving instrumentation as an open parameter.

Section 6 surveys a variety of fundamental styles of polyvariance, encoding each within our parametric semantics, and showing how each can be generalized to new styles of polyvariance or combined with other strategies. We show that no degree of precision lost from either store widening or closure conversion is fundamentally out of reach when using our method (these degrees of precision are also forms of polyvariance within our framework).

2. Myriad Styles of Polyvariance

Following Sharir and Pnueli [51], call sensitivity was used by Jones, Muchnick, and Harrison [19, 26] in the '80s and then generalized to control-flow analysis of higher-order languages (k-CFA) by Shivers [52]. The '90s saw a broader exploration of different strategies for polyvariance, including a polynomial-time approximation for call-sensitive higher-order flow analysis by Jagannathan and Weeks [21] and Agesen's *Cartesian product algorithm* (CPA) [1], an enhancement for type recovery algorithms. A variety of polyvariant type systems emerged, the preponderance of which are call sensitive [2, 4, 20, 29, 43, 44, 59]. Ideas from type systems also found their way back into flow analyses [2, 7]; for example, inspired by `let`-polymorphism, Wright and Jagannathan [60] presents *polymorphic splitting*, a style of call sensitivity that varies the degree of sensitivity on a per-function basis using the `let`-depth of each function as its heuristic. Milanova et al. [41] introduces another very different style of polyvariance, *object sensitivity*, which uses a history of the allocation points of objects to differentiate program contexts. Like call sensitivity, object sensitivity forms a well-ordered design space of increasingly precise analyses that may reach concrete (precise) evaluation only in its limit. Growing evidence (particularly for points-to analysis of Java) supports the idea that object-sensitive analyses tend to be more effective and efficient than call-sensitive ones for object-oriented targets [6, 30–33, 42]. Recently, Smaragdakis et al. [53] has generalized object sensitivity to a wider range of variations and introduced a new approximation of these called *type sensitivity*.

Different styles of polyvariance may be viewed as different heuristics for managing the trade-off between complexity and precision in a static analysis. Call sensitivity supposes that program values will tend to correlate with recent call sites (or the surrounding few stack frames) and allows for more complexity in a way which is capable of expressing these correlations. For targets where this is a good heuristic, a greater number of more precise flow sets will result. For targets where it is not, a greater number of equally imprecise flow sets may result. Object sensitivity supposes that program values will tend to correlate with the allocation point of a function's receiving object (and the allocation point of its allocating object in turn, and so forth). The Cartesian product algorithm supposes that program values for one argument to a function will correlate with program values for other arguments to the same function. Polymorphic splitting supposes that more deeply nested function definitions will benefit from a greater degree of call history than less deeply nested definitions. Each strategy for polyvariance represents a gambit on the part of an analysis designer that targets of the analysis will tend to behave in a certain way.

2.1 Toward Better Trade-offs

To further illustrate this point, consider a `max` function:

```
... 0(let ([max (lambda (a b) (if (> a b) a b))])
      1(max 0 1)
      2(max "a" "at"))
```

As before, a 1-call sensitive analysis will be precise enough to keep the values `0` and `"a"` from merging; however, if `max` is η-expanded k times, a k-call sensitive analysis will not be enough to keep the approximation for `a`'s behavior from becoming $\{\texttt{int}, \texttt{string}\}$ (in the case of a type recovery, or $\{\texttt{0}, \texttt{"a"}\}$ for a constant propagation). In a sense this is not imprecise because neither of these are spurious values for `a`. Even from this context-agnostic perspective however, spurious inter-argument patterns are being implied between the approximations for `a` and `b`. It appears that `max` could be invoked on both an integer and a string at the same time. To eliminate this kind of imprecision, the Cartesian product algorithm builds up whole tuples of arguments for each function, preserving these inter-argument patterns and eliminating the possibility of calls like `(max "a" 1)`. For the function `max`, CPA has the same complexity as k-CFA but yields significantly greater precision. For a different function, one where all such inter-argument combinations are possible, CPA will exhaustively enumerate all combinations at great expense, while k-CFA implies them at no additional cost. For different targets of analysis, or even different portions of the same target of analysis, different styles of polyvariance can exhibit very different efficacies in yielding degrees of precision (or efficiencies at yielding the same degree of precision).

Strategy	Allocator	Instrumentation
Univariance	$\widetilde{alloc}_{\top}(x, \tilde{\varsigma}) \triangleq \top$	None
Monovariance	$\widetilde{alloc}_{\text{0CFA}}(x, \tilde{\varsigma}) \triangleq x$	None
1-CFA	$\widetilde{alloc}_{\text{1CFA}}(x, (call, \_, \_)) \triangleq (x, call)$	None
Call-Only Sensitivity	$\widetilde{alloc}_{\text{callonly}}(x, (call, \tilde{\rho}, \tilde{\sigma}, \tilde{\iota})) \triangleq (x, \tilde{\iota})$	Tracks a history of only call sites
Call+Return Sensitivity	$\widetilde{alloc}_{\text{calls}}(x, (call, \tilde{\rho}, \tilde{\sigma}, \tilde{\iota})) \triangleq (x, \tilde{\iota})$	Tracks a history of call and return points
Polymorphic Splitting	$\widetilde{alloc}_{\text{Lcalls}}(x, (call, \tilde{\rho}, \tilde{\sigma}, \tilde{\iota})) \triangleq (x, \tilde{\iota})$	Tracks a variable history of call sites
Object Sensitivity	$\widetilde{alloc}_{\text{obj}}(x, (call, \tilde{\rho}, \tilde{\sigma}, (\tilde{\sigma}_O, \tilde{O}))) \triangleq (x, \tilde{O})$	Tracks a history of per-object allocation points
Closure Sensitivity	$\widetilde{alloc}_{\text{clo}}(x, (call, \tilde{\rho}, \tilde{\sigma}, (\tilde{\sigma}_O, \tilde{O}))) \triangleq (x, \tilde{O})$	Tracks a history of per-closure creation points
Zeroth-Argument Sensitivity	$\widetilde{alloc}_{\text{arg}_0}(x, ((ae_f\, ae_0 \ldots), \ldots)) \triangleq (x, \mathcal{T}(\tilde{\mathcal{A}}(ae_0, \tilde{\varsigma})))$	None
Store Sensitivity	$\widetilde{alloc}_{\text{ss}}(x, (call, \ldots, (\tilde{\iota}, \tilde{\rho}_\Sigma, \tilde{\sigma}_\Sigma))) \triangleq (x, call, \tilde{\rho}_\Sigma, \tilde{\sigma}_\Sigma)$	Rebuilds per-state stores lost by store widening
Concrete Evaluation	$\widetilde{alloc}_{\bot}(x, (call, \tilde{\rho}, \tilde{\sigma}, \tilde{\iota}))) \triangleq (x, \lvert dom(\tilde{\sigma})\rvert)$	None

Figure 1. A selection of allocators.

Similar trade-offs can be described for other forms of polyvariance and each further intersects with the well-known paradox of flow analysis that greater precision can, in practice, lead to smaller model sizes and faster runtimes [60]. While establishing better guarantees of analysis efficiency does correlate inversely with guarantees of analysis precision in absolute terms, analyses with more precise information for data flows will often have more precise control flows and explore a smaller overall model. Scaling polyvariant flow analysis to larger programs written in dynamic languages like Racket/Scheme, Python, or JavaScript, hinges on being able to reliably make good trade-offs and exploit this paradox; otherwise, the global use of polyvariance (for nearly all the varieties mentioned) yields an exponential-time analysis in the worst case due to the structure of environments in a higher-order setting [40, 56]. (The exception among the techniques mentioned is the poly-k-CFA of Jagannathan and Weeks [21] which effectively uses flat environments.)

Much work has gone into simply defining a correct semantics for Python and JavaScript [18, 49, 54], with perhaps the most compelling effort (at least, for the purposes of constructing a static analysis) being Guha et al.'s λ_{JS} [17] and its successor, Politz et al.'s λ_{S5} [47]. 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 [13]. 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 an intractable $k \geq 2$.

One potential solution might be to use the flat environments of poly-k-CFA or Might et al.'s mCFA [40], however in the case of a language like Racket or Scheme, the frequent idiomatic use of higher-order functions could make this impractical for getting needed precision in the structure of environments. What seems to be needed are increasingly nuanced, introspective, and adaptive forms of polyvariance which better suit their targets and the properties we may wish to prove or discover for them. For example, a recent development shows that the polyvariance of continuations can be adapted in a way which guarantees perfect stack precision (i.e., perfect return flows [11, 12, 24, 58]) at no asymptotic complexity overhead [16], a quite ideal trade-off between complexity and precision obtained through a subtle refinement of the polyvariance used. The direction of research in this area and the challenges of precisely modeling dynamic higher-order programming languages suggests an important development would be an easy way to adjust the polyvariance of a flow analysis (in theory and in practical implementations) that is both always safe and fully general.

3. The Big Picture

We develop a unified approach to encoding *all* and *only* sound forms of polyvariance, as tunings of an allocation function. We show that the design space of polyvariance uniquely and exactly circumscribes the design space of tunings for this function and that no possible allocation strategy can lead to an unsound analysis. This leads us to the main idea of this paper: *allocation characterizes polyvariance*. All classic flavors of polyvariance can be easily recapitulated using our methodology and we are able to derive novel variations by generalizing each. Furthermore, all possible variations on allocation yield a sound polyvariant analysis.

There are thus two directions to consider: that every allocation strategy gives rise to a sound polyvariant analysis, and that every sound polyvariant analysis can be implemented by an allocation strategy. We employ the *a posteriori* soundness process of Might and Manolios [38] to show that every allocator results in a sound analysis. This means we may instrument our core flow analysis arbitrarily to guide the allocator and so long as this extension to our analysis only impacts the store through the narrow interface of allocation, no instrumentation may lead to an unsound flow analysis. Furthermore, every form of polyvariance is expressible through this interface and we may express any allocation behavior by permitting any instrumentation. All forms of polyvariance are ways of merging and differentiating flow sets. In a store-passing-style interpreter, this is determined by the addresses we allocate.

In Figure 1, we summarize a selection of the styles of polyvariance we survey in Section 6. For each of these, classic styles of polyvariance and novel variations, there is a pair of an allocation function and an instrumentation that encodes it. For example, the instrumentation for k-call sensitivity adds tracking of k-length call histories to the analysis so that a call-sensitive allocator may produce addresses unique to both the syntactic variable being allocated for and the current approximate calling context.

4. Abstracting Abstract Machines

This section reviews the AAM methodology we build upon, developing a concrete semantics for a simple language and abstracting it to obtain a monovariant approximation. We follow this with an explanation of the traditional strategy for proving soundness, and of store widening—an essential approximation for obtaining a tractable, polynomial-time analysis.

Static analysis by abstract interpretation proves properties of a program by running code through an interpreter powered by an *abstract semantics* that approximates the behavior of an exact *concrete semantics*. This process is a general method for analyzing programs and serves applications such as program verification, malware/vulnerability detection, and compiler optimization, among others [8–10, 35]. Van Horn and Might's approach of *abstracting abstract machines* (AAM) uses abstract interpretation of abstract machines for *control-flow analysis* (CFA) of functional (higher-order) programming languages [25, 37, 57]. The AAM methodology is flexible in allowing a high degree of control over how program states are represented. AAM provides us with a general method for automatically abstracting an arbitrary small-step abstract-machine semantics to obtain an approximation in a variety of styles. Importantly, one such style aims to focus all unboundedness in a semantics on the machine's address-space. This makes the strategy used for the allocation of addresses crucial to the precision and complexity of the analysis, and as we will see in Section 6, its polyvariance.

4.1 A Concrete Operational Semantics

This section reviews the process of producing a formal operational semantics for a simple language [46], specifically, the untyped λ-calculus in *continuation-passing style* (CPS). CPS constrains call sites to tail position so functions may never return; instead, callers must explicitly pass a continuation forward to be invoked on the result [45]. This makes our semantics tail recursive (small-step) and easier to abstract while entirely eliding the challenges of manually managing a stack and its abstraction, a process previously discussed in the context of AAM [24, 57]. Using an AAM that explicitly models the stack in a precise manner (and allows for adjustable allocation) has also been recently addressed [16].

CPS is a widely used transformation for compiler optimization and program analysis [3]. If the transformation to CPS records which lambdas correspond to continuations, a program may again, along with any optimizations and analysis results, be precisely reconstituted in direct-style form. This means the advantages of CPS can be utilized without compromise or loss of information [28]. The grammar structurally distinguishes between call-sites $call$ and atomic-expressions ae:

$$\begin{aligned} call \in \mathsf{Call} &::= \texttt{(}ae\ ae\ \ldots\texttt{)} \mid \texttt{(halt)} \\ lam \in \mathsf{Lam} &::= \texttt{(}\lambda\ \texttt{(}x\ \ldots\texttt{)}\ call\texttt{)} \\ ae \in \mathsf{AE} &::= lam \mid x \\ x \in \mathsf{Var} &\ \textit{is a set of program variables} \end{aligned}$$

Instead of specifically affixing each expression with a unique label, we assume two identical expressions occurring separately in a program are not equal. While a direct-style language with a variety of continuations (e.g., argument continuations, `let`-continuations, etc.), or extensions such as recursive-binding forms, conditionals, mutation, or primitive operations, would add complexity to any semantics, they do not affect the concepts we are exploring and so are left out.

We define the evaluation of programs in this language using a relation $(\rightarrow_\Sigma)$, over states of an abstract-machine, which determines how the machine transitions from one state to another. States (ς) range over control expression (a call site), binding environment, and value store components:

$$\begin{aligned} \varsigma \in \Sigma &\triangleq \mathsf{Call} \times \mathit{Env} \times \mathit{Store} \\ \rho \in \mathit{Env} &\triangleq \mathsf{Var} \rightharpoonup \mathit{Addr} \\ \sigma \in \mathit{Store} &\triangleq \mathit{Addr} \rightharpoonup \mathit{Value} \\ a \in \mathit{Addr} &\triangleq \mathsf{Var} \times \mathbb{N} \\ v \in \mathit{Value} &\triangleq \mathit{Clo} \\ \mathit{clo} \in \mathit{Clo} &\triangleq \mathsf{Lam} \times \mathit{Env} \end{aligned}$$

Environments (ρ) map variables in scope to an address for the visible binding. Value stores (σ) map these addresses to values (in this case, closures); these may be thought of as a model of the heap. Both these functions are partial and accumulate points as execution progresses.

Evaluation of atomic expressions is handled by an auxiliary function $(\mathcal{A})$ which produces a value (clo) for an atomic expression in the context of a state (ς). This is done by a lookup in the environment and store for variable references (x), and by closure creation for λ-abstractions (lam). In a language containing syntactic literals, these would be translated into equivalent semantic values here.

$$\begin{aligned} \mathcal{A} &: \mathsf{AE} \times \Sigma \rightharpoonup \mathit{Value} \\ \mathcal{A}(x,\ (call,\ \rho,\ \sigma)) &\triangleq \sigma(\rho(x)) \\ \mathcal{A}(lam,\ (call,\ \rho,\ \sigma)) &\triangleq (lam,\ \rho) \end{aligned}$$

The transition relation $(\rightarrow_\Sigma) : \Sigma \rightharpoonup \Sigma$ yields at most one successor for a given predecessor in the state-space Σ. This is defined:

$$\overbrace{(\texttt{(}ae_f\ ae_1\ \ldots\ ae_j\texttt{)}, \rho, \sigma)}^{\varsigma} \rightarrow_\Sigma (call', \rho', \sigma')$$

$$\begin{aligned} \text{where} \quad & (\texttt{(}\lambda\ \texttt{(}x_0 \ldots x_j\texttt{)}\ \ call'\texttt{)}, \rho_\lambda) = \mathcal{A}(ae_f,\ \varsigma) \\ & v_i = \mathcal{A}(ae_i,\ \varsigma) \\ & \rho' = \rho_\lambda[x_i \mapsto a_i] \\ & \sigma' = \sigma[a_i \mapsto v_i] \\ & a_i = (x_i, |dom(\sigma)|) \end{aligned}$$

Execution steps to the call-site body of the lambda invoked (as given by the atomic-evaluation of ae_f). This closure's environment (ρ_λ) is extended with a binding for each variable x_i to a fresh address a_i (formally, an address is *fresh* if $a_i \notin dom(\sigma) \wedge (a_i = a_k \implies i = k)$). A particular strategy for allocating a fresh address is to pair the variable being allocated for with the current number of points in the store. The store is extended with the atomic evaluation of ae_i for each of these addresses a_i. A state becomes stuck if `(halt)` is reached or if the program is malformed (*e.g.*, a free variable is encountered).

To fully evaluate a program $call_0$ using these transition rules, we *inject* it into our state space using a helper $\mathcal{I} : \mathsf{Call} \to \Sigma$:

$$\mathcal{I}(call) \triangleq (call, \varnothing, \varnothing)$$

We may now perform the standard lifting of $(\rightarrow_\Sigma)$ to a collecting semantics defined over sets of states:

$$s \in S \triangleq \mathcal{P}(\Sigma)$$

Our collecting relation $(\rightarrow_s)$ is a monotonic, total function that gives a set including the trivially reachable state $\mathcal{I}(call_0)$ plus the set of all states immediately succeeding those in its input.

$$s \rightarrow_s s', \text{where}$$
$$s' = \{\varsigma' \mid \varsigma \in s \wedge \varsigma \rightarrow_\Sigma \varsigma'\} \cup \{\mathcal{I}(call_0)\}$$

If the program $call_0$ terminates, iteration of $(\rightarrow_s)$ from $\bot$ (i.e., the empty set $\varnothing$) does as well. That is, $(\rightarrow_s)^n(\bot)$ is a fixed point containing $call_0$'s full program trace for some $n \in \mathbb{N}$ whenever $call_0$ is a terminating program. No such n is guaranteed to exist in the general case (when $call_0$ is a non-terminating program) as our language (the untyped CPS λ-calculus) is Turing-equivalent, our semantics is fully precise, and the state-space we defined is infinite.

4.2 An Abstract Operational Semantics

Now that we have formalized program evaluation using our concrete semantics as iteration to a (possibly infinite) fixed point, we are ready to design a computable approximation of this fixed point (the exact program trace) using abstract interpretation. Previous work has explored a wide variety of approaches to systematically abstracting a semantics like these [25, 37, 57]. Broadly construed, the nature of these changes is to simultaneously finitize the domains of our machine while introducing non-determinism both into the transition relation (multiple successor states may immediately follow a predecessor state) and the store (multiple values may become conflated at a single address). We use a finite address space to cut the otherwise mutually recursive structure of values (closures) and environments. (Without addresses and a value store, environments map variables directly to closures and closures contain environments). A finite address space yields a finite state space overall and ensures the computability of our analysis. Typographically, components unique to this *abstract* abstract machine wear hats so we can tell them apart without confusing essential underlying roles:

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &\triangleq \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \\
\hat{\rho} \in \widehat{Env} &\triangleq Var \rightharpoonup \widehat{Addr} \\
\hat{\sigma} \in \widehat{Store} &\triangleq \widehat{Addr} \rightarrow \widehat{Value} \\
\hat{a} \in \widehat{Addr} &\triangleq \mathsf{Var} \\
\hat{v} \in \widehat{Value} &\triangleq \mathcal{P}(\widehat{Clo}) \\
\widehat{clo} \in \widehat{Clo} &\triangleq \mathsf{Lam} \times \widehat{Env}
\end{aligned}$$

Value stores are now total functions mapping abstract addresses to a *flow set* ($\hat{v}$) of zero or more abstract closures. This allows a range of values to merge and inhabit a single abstract address, introducing imprecision into our abstract semantics, but also allowing for a finite state space and a guarantee of computability. To begin, we use a monovariant address set $\widehat{Addr}$ with a single address for each syntactic variable. This choice (and its alternatives) is at the heart of our present topic and will be returned to shortly.

Evaluation of atomic expressions is handled by an auxiliary function ($\hat{\mathcal{A}}$) which produces a flow set ($\hat{v}$) for an atomic expression in the context of an abstract state ($\hat{\varsigma}$). In the case of closure creation, a singleton flow set is produced.

$$\begin{aligned}
\hat{\mathcal{A}} &: \mathsf{AE} \times \hat{\Sigma} \rightharpoonup \widehat{Value} \\
\hat{\mathcal{A}}(x,\ (call,\ \hat{\rho},\ \hat{\sigma})) &\triangleq \hat{\sigma}(\hat{\rho}(x)) \\
\hat{\mathcal{A}}(lam,\ (call,\ \hat{\rho},\ \hat{\sigma})) &\triangleq \{(lam,\ \hat{\rho})\}
\end{aligned}$$

The abstract transition relation $(\rightsquigarrow_{\hat{\Sigma}}) \subseteq \hat{\Sigma} \times \hat{\Sigma}$ yields any number of successors for a given predecessor in the state-space $\hat{\Sigma}$. As mentioned when introducing AAM, there are two fundamental changes required using this approach. Because abstract addresses can become bound to multiple closures in the store and atomic evaluation produces a flow set containing zero or more closures, one successor state results for each closure bound to the address for ae_f. Also, due to the relationality of abstract stores, we can no longer use strong update when extending the store $\hat{\sigma}'$.

$$\overbrace{((ae_f\ ae_1\ \ldots\ ae_j), \hat{\rho}, \hat{\sigma})}^{\hat{\varsigma}} \rightsquigarrow_{\hat{\Sigma}} (call', \hat{\rho}', \hat{\sigma}')$$

$$\begin{aligned}
\text{where} \quad & ((\lambda\ (x_0 \ldots x_j)\ \ call'), \hat{\rho}_\lambda) \in \hat{\mathcal{A}}(ae_f,\ \hat{\varsigma}) \\
& \hat{v}_i = \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma}) \\
& \hat{\rho}' = \hat{\rho}_\lambda[x_i \mapsto \hat{a}_i] \\
& \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{v}_i] \\
& \hat{a}_i = x_i
\end{aligned}$$

A weak update is performed on the store instead which results in the least upper bound of the existing store and each new binding. Join on abstract stores distributes point-wise:

$$\hat{\sigma} \sqcup \hat{\sigma}' \triangleq \lambda \hat{a}.\ \hat{\sigma}(\hat{a}) \cup \hat{\sigma}'(\hat{a})$$

Unless it is desirable, and provably safe to do so [39], we never remove closures already seen. Instead, we strictly accumulate every closure bound to each $\hat{a}$ (i.e., abstract closures which simulate closures bound to addresses which $\hat{a}$ simulates) over the lifetime of the program. A flow set for an address $\hat{a}$ indicates a range of values which over-approximates all possible concrete values that can flow to any concrete address approximated by $\hat{a}$. For example, if a concrete machine binds $(y, 345) \mapsto clo_1$ and $(y, 903) \mapsto clo_2$, its monovariant approximation might bind $y \mapsto \{\widehat{clo}_1, \widehat{clo}_2\}$. Precision is lost for $(y, 345)$ both because its value has been merged with $\widehat{clo}_2$, and because the environments for $\widehat{clo}_1$ and $\widehat{clo}_2$ in-turn generalize over many possible addresses for their free variables (the environment in $\widehat{clo}_1$ is less precise than that in clo_1).

To approximately evaluate a program according to these abstract semantics, we first define an abstract injection function, $\hat{\mathcal{I}}$, where the store begins as a function, $\bot$, that maps every abstract address to the empty set.

$$\begin{aligned}
\hat{\mathcal{I}} &: \mathsf{Call} \rightarrow \hat{\Sigma} \\
\hat{\mathcal{I}}(call) &\triangleq (call, \varnothing, \bot)
\end{aligned}$$

We again lift $(\rightsquigarrow_{\hat{\Sigma}})$ to obtain a collecting semantics $(\rightsquigarrow_{\hat{S}})$ defined over sets of states:

$$\hat{s} \in \hat{S} \triangleq \mathcal{P}(\hat{\Sigma})$$

Our collecting relation $(\rightsquigarrow_{\hat{S}})$ is a monotonic, total function that gives a set including the trivially reachable finite-state $\hat{\mathcal{I}}(call_0)$ plus the set of all states immediately succeeding those in its input.

$$\hat{s} \rightsquigarrow_{\hat{S}} \hat{s}', \text{where}$$
$$\hat{s}' = \{\hat{\varsigma}' \mid \hat{\varsigma} \in \hat{s} \wedge \hat{\varsigma} \rightsquigarrow_{\hat{\Sigma}} \hat{\varsigma}'\} \cup \{\hat{\mathcal{I}}(call_0)\}$$

Because $\widehat{Addr}$ (and thus $\hat{\Sigma}$) is now finite, we know the approximate evaluation of even a non-terminating $call_0$ will terminate. That is, for some $n \in \mathbb{N}$, the value $(\rightsquigarrow_{\hat{S}})^n(\bot)$ is guaranteed to be a fixed point containing an approximation of $call_0$'s full concrete program trace [55].

4.2.1 Extension to Larger Languages

Setting up a 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 an explicit stack as continuations are no longer baked into the source text by CPS conversion. Handling other forms is often as straightforward as including an additional transition rule for each. Consider inclusion of a `set!` form, en-

abling a direct modeling of effects:

$$call \in \mathsf{Call} ::= (\texttt{set!}\ x\ ae_v\ ae_\kappa) \mid \ldots$$

We simply extend the definitions of $(\rightarrow_\Sigma)$ and $(\rightsquigarrow_{\hat{\Sigma}})$ to include a new transition rule. In our abstract semantics this might look like:

$$\overbrace{((\texttt{set!}\ x\ ae_v\ ae_\kappa),\ \hat{\rho},\ \hat{\sigma})}^{\hat{\varsigma}} \rightsquigarrow_{\hat{\Sigma}} (call',\ \hat{\rho}',\ \hat{\sigma}')$$

where
$$((\lambda\ (x_0)\ \ call'), \hat{\rho}_\kappa) \in \hat{\mathcal{A}}(ae_\kappa,\ \hat{\varsigma})$$
$$\hat{v} = \hat{\mathcal{A}}(ae_v,\ \hat{\varsigma})$$
$$\hat{\rho}' = \hat{\rho}_\kappa[x_0 \mapsto \hat{a}_0][x \mapsto \hat{a}_x]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_0 \mapsto \{\texttt{void}\}] \sqcup [\hat{a}_x \mapsto \hat{v}]$$
$$\hat{a}_0 = x_0 \qquad \hat{a}_x = x$$

This assumes the Scheme-like behavior of having `set!`-forms return `void` and that `void` is permitted within flow sets.

$$\hat{v} \in \widehat{Value} \triangleq \mathcal{P}(\widehat{Clo} + \{\texttt{void}\})$$

4.3 Soundness

An analysis is *sound* if the information it provides about a program represents an accurate bound on the behavior of all possible concrete executions. The kind of control-flow information the analysis of section 4.2 obtains is a conservative over-approximation of program behavior. It places an upper bound on the propagation of closures though a program.

To establish such a relationship between a concrete and abstract semantics, we use Galois connections. A monotone *Galois connection* is a pair of monotonic functions connecting two lattices which uniquely define one another in the following way:

$$\alpha_S : S \rightarrow \hat{S} \qquad \gamma_S : \hat{S} \rightarrow S$$

$$\alpha_S(s) \subseteq \hat{s} \iff s \subseteq \gamma_S(\hat{s})$$

We use α to formalize a notion of abstraction, while γ encodes concretization. A family of functions α_{comp} map machine components in the concrete semantics to their most precise representative in the abstract semantics. A corresponding family of functions γ_{comp} 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. For the monovariant analysis of section 4.2, a suitable Galois connection may be defined:

$$\alpha_S(s) \triangleq \{\alpha_\Sigma(\varsigma) \mid \varsigma \in s\}$$
$$\alpha_\Sigma(call,\ \rho,\ \sigma) \triangleq (call,\ \alpha_{Env}(\rho),\ \alpha_{Store}(\sigma))$$
$$\alpha_{Env}(\rho) \triangleq \{(x,\ \alpha_{Addr}(a)) \mid (x,\ a) \in \rho\}$$
$$\alpha_{Store}(\sigma) \triangleq \bigsqcup_{(a,clo)\in\sigma} [\alpha_{Addr}(a) \mapsto \{\alpha_{Clo}(clo)\}]$$
$$\alpha_{Clo}(lam,\ \rho) \triangleq (lam,\ \alpha_{Env}(\rho))$$
$$\alpha_{Addr}(x,\ n) \triangleq x$$

This abstraction function (α_S) uniquely defines its corresponding concretization function (γ_S). Using this defined notion of simulation, we may show that our abstract semantics approximates the concrete semantics by proving that simulation is preserved across transition:

$$\alpha_S(s) \subseteq \hat{s} \wedge s \rightarrow_s s' \implies \hat{s} \rightsquigarrow_{\hat{s}} \hat{s}' \wedge \alpha_S(s') \subseteq \hat{s}'$$

$$O(n) \left\{ \begin{array}{c} \\ \hat{a}_0 \\ \hat{a}_1 \\ \vdots \\ \hat{a}_j \\ \vdots \end{array} \right. \overbrace{\begin{array}{ccccccc} \widehat{clo}_0 & \widehat{clo}_1 & \cdots & \widehat{clo}_i & \cdots \\ 0 & 0 & \cdots & 0 & \cdots \\ 0 & 0 & \cdots & 1 & \cdots \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 0 & \cdots & 1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}}^{O(n)}$$

Figure 2. The value space of stores.

Diagrammatically this is:

$$\begin{array}{ccc} s & \xrightarrow{\rightarrow_s} & s' \\ {\scriptstyle\alpha_S}\downarrow\subseteq & & {\scriptstyle\alpha_S}\downarrow\subseteq \\ \hat{s} & \xrightarrow{\rightsquigarrow_{\hat{s}}} & \hat{s}' \end{array}$$

Both constructing analyses using Galois connections and proving them sound using Galois connections has been extensively explored in the literature [10, 37, 50, 57]. Analyses of the style we constructed in this section have previously been proven sound using the above method [36].

4.4 Store Widening

Various forms of widening and further approximations may be layered on top of the naïve analysis ($\rightsquigarrow_{\hat{s}}$). One such approximation is store widening, which is necessary for our analysis to be polynomial-time in the size of the program. To see why store widening is so important, let us consider the complexity of an analysis using ($\rightsquigarrow_{\hat{s}}$). The height of the power-set lattice $(\hat{S}, \cup, \cap)$ is the number of elements in $\hat{\Sigma}$ which is the product of call sites, environments, and stores. A standard worklist algorithm at most does work proportional to the number of states it can discover [40]. Even in the monovariant case, analysis run-time is thus in:

$$O(\overbrace{n}^{|\mathsf{Call}|\times|\widehat{Env}|} \times \overbrace{2^{n^2}}^{|\widehat{Store}|})$$

The number of syntactic points in an input program is in $O(n)$. In the monovariant case, environments map variables to themselves and are isomorphic to the sets of free variables that may be determined for each syntactic point. The number of addresses produced by our monovariant analysis is in $O(n)$ as these are either syntactic variables or expressions. The number of value stores may be visualized as a table of possible mappings from every address to every abstract closure—each may be included in a given store or not as seen in Figure 2. The number of abstract closures is in $O(n)$ because lambdas uniquely determine a monovariant environment. (The same is true of call sites and their monovariant environments within states $\hat{\varsigma}$.) This times the number of addresses gives $O(n^2)$ possible additions to the value store.

The crux of the issue is that, in exploring a naïve state-space (where each state is specific to a whole store), we may explore both sides of every diamond in the store lattice. All combinations of possible bindings in a store may need to be explored, including every alternate path up the store lattice. For example, along one explored path we might extend an address $\hat{a}_1$ with $\widehat{clo}_1$ before extending it with $\widehat{clo}_2$, and along another path we might add these closures in the reverse order (i.e., $\widehat{clo}_2$ before $\widehat{clo}_1$). We might also extend another address $\hat{a}_2$ with $\widehat{clo}_1$ either before or after either of these cases, and so forth. This potential for exponential blow-

up is unavoidable without further widening or coarser structural abstraction.

Global-store widening is an essential technique for combating exponential blow up. This lifts the store alongside a set of reachable states instead of nesting them inside states $\hat{\varsigma}$. To formalize this, we define new *widened* state spaces that pair a set of reachable *configurations* (states *sans* stores) with a single, global value store we maintain as the least upper bound of all stores we encounter during analysis. Instead of accumulating whole stores, and thereby all possible sequences of additions within such stores, the analysis strictly accumulates new values in the store in the same way $(\rightsquigarrow_{\hat{s}})$ accumulates reachable states in a collection $\hat{s}$:

$$\hat{\xi} \in \hat{\Xi} \triangleq \hat{R} \times \widehat{Store} \qquad \text{[state-spaces]}$$
$$\hat{r} \in \hat{R} \triangleq \mathcal{P}(\hat{C}) \qquad \text{[reachable configurations]}$$
$$\hat{c} \in \hat{C} \triangleq \mathsf{Call} \times \widehat{Env} \qquad \text{[configurations]}$$

A widened transfer function $(\rightsquigarrow_{\hat{\Xi}})$ may then be defined that, like $(\rightsquigarrow_{\hat{s}})$, is a monotonic, total function we may iterate to a fixed point.

$$(\rightsquigarrow_{\hat{\Xi}}) : \hat{\Xi} \to \hat{\Xi}$$

This may be defined in terms of $(\rightsquigarrow_{\hat{\Sigma}})$, as was $(\rightsquigarrow_{\hat{s}})$, by transitioning each reachable configuration using the global store to yield a new set of reachable configurations and a set of stores whose least upper bound is the new global store:

$$(\hat{r}, \hat{\sigma}) \rightsquigarrow_{\hat{\Xi}} (\hat{r}', \hat{\sigma}''), \text{ where}$$
$$\hat{s}' = \{\hat{\varsigma}' \mid (call, \hat{\rho}) \in \hat{r} \wedge (call, \hat{\rho}, \hat{\sigma}) \rightsquigarrow_{\hat{\Sigma}} \hat{\varsigma}'\} \cup \{\hat{\mathcal{I}}(call_0)\}$$
$$\hat{r}' = \{(call', \hat{\rho}') \mid (call', \hat{\rho}', \hat{\sigma}') \in \hat{s}'\}$$
$$\hat{\sigma}'' = \bigsqcup_{(\_,\_,\hat{\sigma}') \in \hat{s}'} \hat{\sigma}'$$

In this definition, an underscore (wildcard) matches anything. The height of the $\hat{R}$ lattice is linear (as environments are monovariant) and the height of the store lattices are quadratic (as each global store is strictly extended). Each extension of the store may require $O(n)$ transitions because at any given store, we must transition every configuration to be sure to obtain any changes to the store or otherwise reach a fixed point. A traditional worklist algorithm for computing a fixed point is thus cubic:

$$O(\overbrace{n}^{|\hat{C}|} \times \overbrace{n^2}^{|\widehat{Store}|})$$

Using advanced bit-packing techniques [35], the best known algorithm for global-store-widened 0-CFA is in $O(\frac{n^3}{\log n})$.

5. Allocation as a Tunable Parameter

In the previous section, we systematically developed a global-store-widened analysis of CPS λ-calculus based on a concrete abstract-machine semantics. It is a *monovariant* analysis which means each syntactic variable or intermediate expression we track during analysis receives exactly one flow set to over-approximate all its possible values. A closely related term is *context insensitive*, which means insensitive to any form of context and is a broader term that may, for example, include analyses less precise than this as well. In Section 4.2, the crucial propositional statement (among those defining our abstract transition relation $(\rightsquigarrow_{\hat{\Sigma}})$) which made the analysis monovariant was this one:

$$\hat{a}_i = x_i$$

For each allocation, an address is produced which is unique only to the syntactic variable being allocated for.

The goal of this section will be to produce a parametric semantics which may be tuned by an allocator $\widetilde{alloc}$ that only varies this aspect of the analysis, but may do so without restriction. Although we will formalize this parametric semantics on its own, in the context of our $(\rightsquigarrow_{\hat{s}})$-analysis the primary change looks like:

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

This would allow us to define monovariance as a tuning of this function:

$$\widetilde{alloc}_{\text{0CFA}}(x,\ \tilde{\varsigma}) \triangleq x$$

An equivalence relation on these addresses may be lifted from a notion of equality for syntax, however we must either affix unique labels to every program expression or assume that two identical pieces of syntax found in the same program are syntactically unequal. For simplicity, we assume the latter.

The least polyvariant analysis has an allocator which produces even fewer distinct addresses—in fact, only a single address $\top$ which over-approximates all concrete addresses in any precise evaluation of the target:

$$\widetilde{alloc}_{\top}(x,\ \tilde{\varsigma}) \triangleq \top$$

We might call this the *univariant* allocation scheme because it produces only a single address and smashes all program values together. Even an analysis as imprecise as this could have a use. For example, univariant allocation would make for an exceptionally cheap analysis powering dead-code elimination.

Instead of defining a set of abstract addresses explicitly as done in Section 4.2, we can now allow this set to be defined implicitly by the image or codomain of the allocation function. This does mean that for an analysis to be computable, the allocator must only produce a finite number of abstract addresses. An allocator which does not produce a finite number of addresses, essentially an *infinitely polyvariant* allocation strategy, may be used to tune our analysis to concrete evaluation:

$$\widetilde{alloc}_{\bot}(x,\ (call, \tilde{\rho}, \tilde{\sigma})) \triangleq (x, |dom(\tilde{\sigma})|)$$

This is also an example of a form of polyvariance which must introspect on the current program state in order to produce an address. Without looking at the current store (or using another method), a concrete allocator is unable to ensure it always produces a fresh address (and thus avoids all merging in the store). Being able to represent concrete evaluation as a choice of allocator is also useful because it allows us to write a precise interpreter and a static analysis simultaneously as a single body of code. Along with promoting code reuse and concision, this means testing either one also aids the robustness and stability of the other [23].

Now we have seen three simple points within the design space of allocation strategies and polyvariance. Univariant allocation and concrete allocation frame this design space and represent two topmost and bottom-most strategies; monovariance lies between. Two important questions are left to be answered about the correspondence between polyvariance and allocation, however. First, we must consider whether there is any tuning of allocation which is unsafe (i.e., leads to an unsound analysis) or which is not polyvariant. Second, we must consider whether there are polyvariant strategies which may not be implemented as an allocator.

5.1 *A Posteriori* Soundness

The usual process for proving the soundness of an abstract interpretation is *a priori* in the sense that it may be performed entirely before an analysis is executed. This is the kind of soundness theorem we described in Section 4.3. By contrast, Might and Manolios [38] describes an *a posteriori* soundness process where the abstraction map cannot be fully constructed until after analysis.

This approach factors each α to separate the abstraction of addresses α_{Addr}, producing a family of parametric maps β such that $\beta(\alpha_{Addr}) = \alpha$. A non-deterministic abstract interpretation is then constructed which simultaneously attempts all possible allocation strategies. (This could also be an arbitrary allocation function without loss of generality.) After the analysis is performed, regardless of the allocation strategy taken, a consistent abstraction map may be constructed *a posteriori* which justifies each choice of abstract address whatever it may have been. It is then always possible to plug this Galois connection for addresses into the parametric Galois connection defined by β to obtain a complete connection and proof of soundness.

What is special about the allocation of abstract addresses which could make even a random number generator a sound choice of allocator? Clearly we couldn't define the operation of most other components of our abstract machine randomly and still guarantee a sound analysis. Intuitively, it is because in a concrete evaluation of any program, we may select a fresh and unique address for every new allocation. (In fact, we might justify a garbage collection scheme as safe by showing that when an address becomes unreachable it may be reclaimed and the semantics are guaranteed to remain equivalent to allocating a fresh address.) Allocating a sequence of fresh, unique addresses which are never duplicates of previous concrete addresses is thus a central characteristic of what it means to be a concrete allocator. Whatever the behavior of abstract address allocation then, no inconsistency may arise in the α_{Addr} it induces because of just this property. No concrete address may become abstracted to two different abstract addresses along the sound abstract program trace because no concrete address is allocated more than once.

To illustrate this point, consider Figure 3. It shows a program $call_0$ being injected into a starting state, ς_0, and evaluated step by step. A static analysis performed by iterating an abstract transition relation will produce a transition graph, but for the analysis to be sound, the concrete program trace must abstract to some path through this graph. Such a path is illustrated below, spurious transitions dangling from it. Dotted lines are used to illustrate "abstracts to" relationships for states and addresses (points in α_Σ and α_{Addr}). If a concrete machine and its abstract machine are simulated in lock-step, each abstract transition which allocates an address has two choices: either it can allocate an address $\hat{a}_i$ it has allocated before, or it may allocate a new address $\hat{a}_i$. In both cases, it is deciding what the corresponding (necessarily fresh) concrete address a_i must abstract to in α_{Addr}. In this way, a bisimulation incrementally builds up an abstraction map for addresses, incrementally adding each point $[a_i \mapsto \hat{a}_i]$, one at a time.

In the original presentation of the *a posteriori* soundness theorem, Might and Manolios state an assumption which says that each new abstraction map $\alpha_{Addr}[a_i \mapsto \hat{a}_i]$ must be consistent with whatever partial abstraction map α_{Addr} was built up previously. No further intuitions were given for this assumption, though it is actually the central property which allows the entire *a posteriori* soundness process to work. 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. Each $\alpha_{Addr}[a_i \mapsto \hat{a}_i]$ must be consistent with α_{Addr} because the concrete address, a_i, cannot already be present in α_{Addr}. This makes abstract allocation a tunable analysis parameter with the unique property that every possible tuning results in a sound analysis.

We may now review the *a posteriori* soundness theorem in the context of AAM.

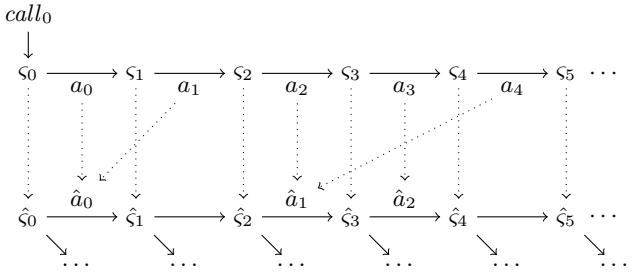

Figure 3. All strategies for allocation induce a consistent Galois connection for addresses.

Theorem 5.1 (*A posteriori* soundness)**.** *If $(\boldsymbol{\varsigma}, \mathbf{a})$ is a concrete execution where $\boldsymbol{\varsigma}$ is a sequence of states and $\mathbf{a} = \langle a_0, a_1, a_2, \ldots \rangle$ is the sequence of concrete addresses allocated, and if some $(\hat{\rightsquigarrow}_{\hat{s}})$ is a dependent simulation [38] of $(\rightarrow_\Sigma)$ under β (dependent only on finding an abstraction map for addresses), and if $\hat{s}$ is a closed abstract transition graph over states, then there must exist an abstraction map for addresses, α_{Addr}, such that $\hat{s}$ is a sound simulation of $(\boldsymbol{\varsigma}, \mathbf{a})$ under the abstraction map $\beta(\alpha_{Addr})$.*

Proof. (Summary of [38].) We proceed by performing an *a posteriori* construction of α_{Addr}. This is done by building up a simulating sequence of abstract states $\hat{\boldsymbol{\varsigma}} = \langle \hat{\varsigma}_0, \hat{\varsigma}_1, \hat{\varsigma}_2, \ldots \rangle$, their respective abstract addresses $\tilde{\mathbf{a}} = \langle \tilde{a}_0, \tilde{a}_1, \tilde{a}_2, \ldots \rangle$, and a sequence of partial address abstraction maps $\boldsymbol{\alpha} = \langle \alpha_0, \alpha_1, \ldots \rangle$. Let N be the length of $\boldsymbol{\varsigma}$ and the initial abstraction map α_0 be $\bot$. At each step a next abstract address $\hat{a}_i$ (or several for multiple-argument lambdas) and abstract state $\hat{\varsigma}_i$ may be chosen simultaneously and non-deterministically from a non-empty set of candidate transitions guaranteed to exist by the dependent simulation condition [38]. Each new point $[a_i \mapsto \hat{a}_i]$ is accumulated into an updated intermediate abstraction map $\alpha_i = \alpha_{i-1}[a_i \mapsto \hat{a}_i]$ which inductively builds up α_{Addr} in its limit:

$$\alpha_{Addr} = \lim_{i \to N} \alpha_i$$

Then $\hat{\boldsymbol{\varsigma}}$ in $\hat{s}$ is a simulation of $(\boldsymbol{\varsigma}, \mathbf{a})$ with respect to $\beta(\alpha_{Addr})$. □

5.2 Introspection and Instrumentation

Now we may consider whether or not any strategy for polyvariance can be implemented as a tuning of the allocation function. What about more precise forms of call sensitivity, 1-CFA or 2-CFA? A 1-call sensitive allocator can be defined by introspecting on the state being transitioned from and incorporating the most recent call site into the address being produced:

$$\widetilde{alloc}_{\text{1CFA}}(x,\ (call, \_, \_)) \triangleq (x_i, call)$$

This makes addresses (and their flow sets) unique to both the variable x and the call site which preceded the binding. If we were to attempt an implementation of a more precise variant of call sensitivity however, like 2-CFA, we run into a problem because our analysis simply does not include the information necessary to guide this style of polyvariance. The current abstract state contains the most recent call site passed through, but it does not include the second most recent call site.

To permit a tuning for $\widetilde{alloc}_{\text{2CFA}}$, we could instrument our core flow analysis with a new fourth component of machine states that specifically tracks the second most recent call site. If we were to extend the analysis with such information, a 2-call sensitive allocator could be defined:

$$\widetilde{alloc}_{\text{2CFA}}(x,\ (call, \_, \_, call')) \triangleq (x_i, call, call')$$

$$\tilde{\varsigma}_0 \in \tilde{\Sigma}$$
$$\widetilde{alloc} \in \mathsf{Var} \times \tilde{\Sigma} \to \widetilde{Addr}$$
$$(\overset{\text{Inst}}{\rightsquigarrow}) \in \tilde{\Sigma} \times \mathsf{Call} \times \widetilde{Env} \times \widetilde{Store} \to \mathcal{P}(\tilde{I})$$

Figure 4. Parameters to these semantics.

$\tilde{s} \in \tilde{S} \triangleq \mathcal{P}(\tilde{\Sigma})$	[analysis results]
$\tilde{\xi} \in \tilde{\Xi} \triangleq \tilde{R} \times \widetilde{Store}$	[widened results]
$\tilde{r} \in \tilde{R} \triangleq \mathcal{P}(\tilde{C})$	[reachable configs]
$\tilde{c} \in \tilde{C} \triangleq \mathsf{Call} \times \widetilde{Env} \times \tilde{I}$	[configurations]
$\tilde{\varsigma} \in \tilde{\Sigma} \triangleq \mathsf{Call} \times \widetilde{Env} \times \widetilde{Store} \times \tilde{I}$	[states]
$\tilde{\rho} \in \widetilde{Env} \triangleq \mathsf{Var} \rightharpoonup \widetilde{Addr}$	[environments]
$\tilde{\sigma} \in \widetilde{Store} \triangleq \widetilde{Addr} \to \widetilde{Value}$	[value stores]
$\tilde{\imath} \in \tilde{I}$ is defined by the parameter $(\overset{\text{Inst}}{\rightsquigarrow})$	[inst. data]
$\tilde{a} \in \widetilde{Addr}$ is defined by the parameter $\widetilde{alloc}$	[addresses]
$\tilde{v} \in \widetilde{Value} \triangleq \mathcal{P}(\widetilde{Clo})$	[flow sets]
$\widetilde{clo} \in \widetilde{Clo} \triangleq \mathsf{Lam} \times \widetilde{Env}$	[closures]

Figure 5. Abstract domains for our parametric semantics.

In this case, $call'$ is a new component of machine states that represents the second most recent call site. Naturally, $(\overset{\hat{}}{\rightsquigarrow_\Sigma})$ and $\hat{\mathcal{I}}$ would need to be extended to include this information.

Crucially, due to the *a posteriori* soundness theorem, we may add whatever instrumentation is needed to guide the behavior of an allocator. An analysis designer may wish to extend the core flow analysis in a way which is sound with respect to a dynamic analysis or instrumentation of the concrete semantics; however, even if the analysis is extended with unsound information about a program, this information can still be used to guide allocation behavior without any possibility of it causing unsoundness within the core flow analysis (e.g., within the store). This means we may leave such instrumentation open as another parameter to a semantics and place no restrictions on its behavior. Because we lose no expressivity in this instrumentation, all conceivable allocation functions can be expressed as well. This means all strategies for merging and differentiation of abstract addresses (and their flow sets) are possible, and thus all forms of polyvariance may be expressed as a combination of some allocator and some instrumentation.

5.3 A Parametric Semantics

In this section, we present a parametric semantics which may be tuned by both an allocation function and an instrumentation (an arbitrary extension of the analysis). Typographically, we switch exclusively to using tildes to keep this machine distinct from the machine of Section 4.

Our parametric semantics is encoded in a function:

$$CFA : \overbrace{\tilde{\Sigma}}^{\text{start state}} \times \overbrace{(\tilde{\Sigma} \times \mathsf{Call} \times \widetilde{Env} \times \widetilde{Store} \to \mathcal{P}(\tilde{I}))}^{\text{instrumentation}} \times \overbrace{(\mathsf{Var} \times \tilde{\Sigma} \to \widetilde{Addr})}^{\text{allocator}} \to \overbrace{(\tilde{S} \to \tilde{S})}^{\text{analysis}}$$

$$CFA(\tilde{\varsigma}_0, \overset{\text{Inst}}{\rightsquigarrow}, \widetilde{alloc}) \triangleq (\rightsquigarrow_{\tilde{s}})$$
$$CFA_\nabla(\tilde{\varsigma}_0, \overset{\text{Inst}}{\rightsquigarrow}, \widetilde{alloc}) \triangleq (\rightsquigarrow_{\tilde{\Xi}})$$

$$(\rightsquigarrow_{\tilde{s}}) : \tilde{S} \to \tilde{S}$$
$$\tilde{s} \rightsquigarrow_{\tilde{s}} \tilde{s}', \text{ where}$$
$$\tilde{s}' = \{\tilde{\varsigma}' \mid (call, \tilde{\rho}, \tilde{\sigma}, \tilde{\imath}) \in \tilde{s} \wedge (call, \tilde{\rho}, \tilde{\sigma}, \tilde{\imath}) \rightsquigarrow_{\tilde{\Sigma}} \tilde{\varsigma}'\} \cup \{\tilde{\varsigma}_0\}$$

$$(\rightsquigarrow_{\tilde{\Xi}}) : \tilde{\Xi} \to \tilde{\Xi}$$
$$(\tilde{r}, \tilde{\sigma}) \rightsquigarrow_{\tilde{\Xi}} (\tilde{r}', \tilde{\sigma}''), \text{ where}$$
$$\tilde{s}' = \{\tilde{\varsigma}' \mid (call, \tilde{\rho}, \tilde{\imath}) \in \tilde{r} \wedge (call, \tilde{\rho}, \tilde{\sigma}, \tilde{\imath}) \rightsquigarrow_{\tilde{\Sigma}} \tilde{\varsigma}'\} \cup \{\tilde{\varsigma}_0\}$$
$$\tilde{r}' = \{(call', \tilde{\rho}', \tilde{\imath}') \mid (call', \tilde{\rho}', \tilde{\sigma}', \tilde{\imath}') \in \tilde{s}'\}$$
$$\tilde{\sigma}'' = \bigsqcup_{(\_,\_,\tilde{\sigma}',\_) \in \tilde{s}'} \tilde{\sigma}'$$

$$\tilde{\mathcal{A}} : \mathsf{AE} \times \tilde{\Sigma} \rightharpoonup \widetilde{Value}$$
$$\tilde{\mathcal{A}}(x,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\imath})) = \tilde{\sigma}(\tilde{\rho}(x))$$
$$\tilde{\mathcal{A}}(lam,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\imath})) = \{(lam,\ \tilde{\rho})\}$$

$$(\rightsquigarrow_{\tilde{\Sigma}}) \subseteq \tilde{\Sigma} \times \tilde{\Sigma}$$
$$\overbrace{((ae_f\ ae_1\ \ldots\ ae_j), \tilde{\rho}, \tilde{\sigma}, \tilde{\imath})}^{\tilde{\varsigma}} \rightsquigarrow_{\tilde{\Sigma}} (call', \tilde{\rho}', \tilde{\sigma}', \tilde{\imath}')$$
where $((\lambda\ (x_0 \ldots x_j)\ \ call'), \tilde{\rho}_\lambda) \in \tilde{\mathcal{A}}(ae_f,\ \tilde{\varsigma})$
$\tilde{v}_i = \tilde{\mathcal{A}}(ae_i,\ \tilde{\varsigma})$
$\tilde{\rho}' = \tilde{\rho}_\lambda[x_i \mapsto \tilde{a}_i]$
$\tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a}_i \mapsto \tilde{v}_i]$
$\tilde{a}_i = \widetilde{alloc}(x_i, \tilde{\varsigma})$
$\tilde{\varsigma} \overset{\text{Inst}}{\rightsquigarrow} (call', \tilde{\rho}', \tilde{\sigma}', \tilde{\imath}')$ (N.B. this syntactic sugar.)

Figure 6. Transition rules for our parametric semantics.

CFA is a function of three arguments: a starting state, $\tilde{\varsigma}_0$, which specifies the program to interpret and its initial $\tilde{\imath}_0$, an instrumentation, $(\overset{\text{Inst}}{\rightsquigarrow})$, which may be used to extend the core analysis arbitrarily, and an allocator, $\widetilde{alloc}$. Given three such parameters, $CFA(\tilde{\varsigma}_0, \overset{\text{Inst}}{\rightsquigarrow}, \widetilde{alloc})$ yields a monotonic analysis function which may be iterated to a fixed point. If the image of $\widetilde{alloc}$ (the set $\widetilde{Addr}$ it can produce) and the image of $(\overset{\text{Inst}}{\rightsquigarrow})$ (the set of sets of $\tilde{I}$ it can produce) are finite sets, then there must exist an $n \in \mathbb{N}$ such that $(CFA(\tilde{\varsigma}_0, \overset{\text{Inst}}{\rightsquigarrow}, \widetilde{alloc}))^n(\bot)$ is a fixed point encoding a sound analysis of $\tilde{\varsigma}_0$ using the instrumentation and style of polyvariance specified.

Figure 4 shows the signatures of the three parameters to CFA. The allocator defines a set of addresses $\widetilde{Addr}$ for the analysis to

use. The instrumentation relation defines a set of instrumentation data $\tilde{I}$ to extend the core flow analysis and enable a greater variety of allocators. An instrumentation is a function which, taking the underlying analysis transition into account, determines the instrumentation data to be included in successor states. Although this may not constrain the core flow analysis, to emphasize that it can encode an entire analysis of its own, we use the following syntactic sugar:

$$\tilde{\varsigma} \overset{\text{Inst}}{\rightsquigarrow} (call', \tilde{\rho}', \tilde{\sigma}', \tilde{\imath}') \iff \tilde{\imath}' \in (\overset{\text{Inst}}{\rightsquigarrow})(\tilde{\varsigma}, call', \tilde{\rho}', \tilde{\sigma}')$$

Figure 5 shows the remaining domains that the machine operates over. These are similar to the domains in Section 4 except that states and configurations contain instrumentation data ($\tilde{\imath}$) and addresses are specified implicitly by the allocator chosen.

Figure 6 defines the transition relation ($\rightsquigarrow_{\tilde{s}}$) yielded by CFA when supplied with all its arguments, as well as a store-widened version ($\rightsquigarrow_{\tilde{\Xi}}$) yielded by $CFA_{\triangledown}$. There are three meaningful changes from the semantics of Section 4, one for each parameter. First, the starting state $\tilde{\varsigma}_0$ is as provided and not produced by an injection function (this allows the user to control the initial instrumentation data along with the program to be analyzed). Second, addresses $\tilde{a}_i$ are constrained only by the allocator $\widetilde{alloc}$ provided. Third, the instrumentation function ($\overset{\text{Inst}}{\rightsquigarrow}$) constrains the instrumentation data $\tilde{\imath}'$ based on all other components of a transition.

6. Allocation Characterizes Polyvariance

This section explores the design space opened up by a semantics parameterized over both an instrumentation and an abstract allocator, showing how it encompasses a variety of previously published polyvariant techniques, novel techniques, and variations on these.

6.1 Call Sensitivity (k-CFA)

Call-sensitive instrumentation tracks a history of up to k call sites for use in differentiating addresses.

$$(call,\ \_,\ \_,\ \tilde{\imath}) \overset{\text{Inst}}{\rightsquigarrow}_{call(k)} (\_,\ \_,\ \_,\ take_k(call : \tilde{\imath}))$$

The function $take_k$ returns the front at-most k elements of its input as a new list. For this instrumentation to distinguish between two syntactically equivalent call sites located in different parts of a program, we assume two pieces of syntax are only equal when they are the same piece of syntax from the same part of the same program. This allows us to safely lift an equivalence relation on syntax to an equivalence relation for addresses.

Using ($\overset{\text{Inst}}{\rightsquigarrow}_{call(k)}$), we may tune our analysis to implement k-CFA using an allocator which incorporates these k-length call histories in the addresses it produces.

$$\widetilde{alloc}_{\text{call}}(x,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\imath})) \triangleq (x, \tilde{\imath})$$

The parametric semantics of Section 5.3 can be tuned to recapitulate the k-call sensitive style of polyvariance for a program $call_0$ using the parameterization:

$$CFA((call_0, \varnothing, \bot, \epsilon), \overset{\text{Inst}}{\rightsquigarrow}_{call(k)}, \widetilde{alloc}_{\text{call}})$$

6.1.1 Ambiguity in k-CFA

The original formulation of k-CFA was described as tracking a history of the last k call sites execution passed through, however, it was applied to a CPS intermediate representation. After a CPS transformation, every return point has been encoded as a call site. This means, as implemented, k-CFA was actually tracking a history of the first k call sites *or* return points. A call sensitivity which only remembers call sites and not return points is a somewhat different form of polyvariance from what Shivers originally formalized [52].

Using a direct-style language, this difference would be easy to formalize as a tuning of our parametric framework because the difference between calls and returns is syntactically evident. Using CPS, we must assume either partitioned CPS with both `lambda` and `cont` forms which behave identically but are kept separate, or we must assume this distinction is being encoded another way. To demonstrate a tuning for call-only sensitivity, we assume a predicate $Ret : \mathsf{Call} \to Bool$ which returns true if and only if the call site given was originally a return point (before CPS conversion). We are only required to change the behavior of our instrumentation:

$$(call, \_, \_, \tilde{\imath}) \overset{\text{Inst}}{\rightsquigarrow}_{callonly(k)} \begin{cases} (\_, \_, \_, take_k(call : \tilde{\imath})) & \neg Ret(call) \\ (\_, \_, \_, \tilde{\imath}) & Ret(call) \end{cases}$$

We can then instantiate our framework to a 2-call-only sensitive analysis as follows:

$$CFA((call_0, \varnothing, \bot, \epsilon))(\overset{\text{Inst}}{\rightsquigarrow}_{callonly(2)})(\widetilde{alloc}_{\text{call}})$$

We can also produce tunings which represent an analysis that remembers only return points or an analysis sensitive to the top k stack frames. This means there are at least four reasonable interpretations of k-CFA which resolve the ambiguity between its original description and its original formalization. Each of these four styles of polyvariance are subtly different and may yield a different analysis result. Furthermore, none of these four styles of polyvariance strictly dominates the precision of any other. For each we can find examples where that specific interpretation of k-CFA produces the best result. For example, the following snippet of Racket code (before CPS conversion) differentiates call+return sensitivity and call-only sensitivity.

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

The last call before binding `v` a first time is `(id #f)`, but the second time it is `(id #t)`. This means a 1-call-only sensitive analysis will keep both addresses bound to `v` distinct. The last call *or* return before binding `v` however, is the identity function's return point `x` in both cases. This means a 1-call+return style of polyvariance will merge both `#t` and `#f` at a single address for `v`.

Different styles of polyvariance represent different heuristics for the trade-off between precision and complexity and may strike a poor balance on one program while striking an excellent balance on another. Having a safe parametric framework which can so easily instantiate any conceivable heuristic could prove an important step in understanding which styles of polyvariance work best in what situations and thereby inform us how to better adapt the polyvariance used to suite a particular target of analysis.

6.1.2 Variable Call Sensitivity

Wright and Jagannathan's polymorphic splitting is a form of adaptive call sensitivity inspired by `let`-polymorphism where the degree of polyvariance can vary between functions [60]. The number of `let`-form binding expressions (right-hand sides) a lambda was originally defined within (in the case of our language, before CPS conversion) forms a simple heuristic for its call sensitivity when invoked. To implement k-call sensitivity with a per-function k, we assume a parameter function $L : \mathsf{Call} \to \mathbb{N}$ that takes the body of a lambda and gives back a k for its `let`-depth (or any other heuristic for varying the maximum length call history).

$$(call,\ \_,\ \_,\ \tilde{\imath}) \overset{\text{Inst}}{\rightsquigarrow}_{Lcalls(L)} (call',\ \_,\ \_,\ take_{(L(call'))}(call : \tilde{\imath}))$$

This call history is then used for allocating addresses.

$$\widetilde{alloc}_{\text{Lcalls}}(x,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota})) \triangleq (x, \tilde{\iota})$$

Because all parameterizations of our semantics are sound, all possible heuristics L are too. No tuning of L can produce an infinite k, only arbitrarily large k. In the case of polymorphic-splitting, because no program can contain an infinite nesting of `let`-forms, every program has a `let`-polymorphic tuning of L.

This instrumentation and allocator generalize the behavior of polymorphic splitting and could be further generalized by adding a function like Ret from the previous subsection for selecting which call sites to include in the history to begin with. In this way, call sensitivity can be seen as a wide design space itself within the broader design space of polyvariant allocation strategies.

6.2 Object Sensitivity

Smaragdakis et al. [53] distinguishes multiple variants of *object sensitivity*, first described by Milanova et al. [41]. This style of context sensitivity is entirely different from call sensitivity and uses a history of the allocation points for objects to guide polyvariance.

We temporarily extend our language with a `vector`-form to represent simple objects and present a faithful tuning for object sensitivity.

$$ae \in \mathsf{AE} ::= lam \mid x \mid vec$$
$$vec \in \mathsf{Vec} ::= (\texttt{vector}\ x\ \ldots)$$

We define abstract-object values permitted within flow sets (tuples of pointers):

$$\tilde{v} \in \widetilde{Value} \triangleq \mathcal{P}(\widetilde{Clo} + \widetilde{Obj})$$
$$\widetilde{obj} \in \widetilde{Obj} \triangleq \widetilde{Addr}^*$$

And give vector syntax an interpretation in the atomic-expression evaluator:

$$\tilde{\mathcal{A}}(x,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota})) = \tilde{\sigma}(\tilde{\rho}(x))$$
$$\tilde{\mathcal{A}}(lam,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota})) = \{(lam,\ \tilde{\rho})\}$$
$$\tilde{\mathcal{A}}((\texttt{vector}\ x_0\ \ldots\ x_j),\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota})) = \{(\tilde{\rho}(x_0), \ldots \tilde{\rho}(x_j))\}$$

As the fields of our objects are effectively each vector's indices, and because these are strictly kept distinct instead of being merged, we can call this representation for vectors *field sensitive* [34]. Flow sets for objects look like $\{(\tilde{a}_1, \tilde{a}_2, \tilde{a}_3), \ldots\}$ and not like $\{\tilde{a}_1, \tilde{a}_2, \tilde{a}_3, \ldots\}$, preserving the relationship between keys and values. Allowing these lists of addresses within flow sets is not a new source of unboundedness in the machine because the longest possible list is the length of the longest `vector` form in the finite program text.

In Smaragdakis' framework, k-full-object sensitivity tracks the allocation point of each object, the allocation point for the object which created it, and so forth. In our extended CPS language, the zeroth parameter to a function is its receiving object.

$$\tilde{O} \in \mathsf{Call}^*$$
$$\tilde{\sigma}_O \in \widetilde{Addr} \times \widetilde{Obj} \to \mathcal{P}(\mathsf{Call}^*)$$

Each state is extended with a current allocation history, $\tilde{O}$, and an object-sensitivities store, $\tilde{\sigma}_O$, which maps an abstract object at an address to a set of possible allocation histories for that object. Each transition extends $\tilde{\sigma}_O$ with a new allocation history (produced by extending the current allocation history $\tilde{O}$ with a new allocation point, the current call site) for each ae_i that constructs a new object. Existing objects bound to a variable (some y_i) have their histories propagated along with the objects. Each transition then yields a successor for each possible allocation history associated with a receiving object. If global-store widening is used for an analysis, a similar form of widening might be used for object-sensitivities stores.

$$\overbrace{((ae_f\ \ldots\ ae_j),\ \tilde{\rho},\ \_,\ (\tilde{\sigma}_O, \tilde{O}))}^{\tilde{\varsigma}} \overset{\text{Inst}}{\rightsquigarrow}_{obj(k)} (call',\ \tilde{\rho}',\ \_,\ (\tilde{\sigma}'_O, \tilde{O}'))$$
$$\text{where}\quad ((\lambda\ (x_0 \ldots x_j)\ call'), \_) \in \tilde{A}(ae_f, \tilde{\varsigma})$$
$$\widetilde{obj}_i \in \tilde{A}(ae_i, \tilde{\varsigma})$$
$$\tilde{O}' \in \tilde{\sigma}'_O(\tilde{\rho}'(x_0), \widetilde{obj}_0)$$
$$\tilde{\sigma}'_O = \tilde{\sigma}_O \sqcup \bigsqcup_{ae_i = (\texttt{vector}\ \ldots)} [(\tilde{\rho}'(x_i), \widetilde{obj}_i) \mapsto \{take_k((ae_f\ \ldots\ ae_j) : \tilde{O})\}] \sqcup \bigsqcup_{ae_i = y_i} [(\tilde{\rho}'(x_i), \widetilde{obj}_i) \mapsto \tilde{\sigma}_O(\tilde{\rho}(y_i), \widetilde{obj}_i)]$$

An allocator for this style of polyvariance then pairs each variable with the current allocation history (ignoring the sensitivities store which only needs to be used internally).

$$\widetilde{alloc}_{\text{obj}}(x,\ (\_,\ \_,\ \_,\ (\tilde{\sigma}_O, \tilde{O}))) \triangleq (x, \tilde{O})$$

Smaragdakis, et al. [53] and Lhoták and Hendren [31] find object sensitivity to be particularly efficient for object-oriented languages in their empirical investigations using the Java DaCapo and SpecJVM benchmarks. Kastrinis and Smaragdakis [27] present combinations of object and call sensitivity. Combinations of styles of polyvariance can also be accomplished by a tuning of instrumentation and allocation. Section 6.5 presents a general method for combining multiple styles of polyvariance.

6.2.1 Closure Sensitivity

Inspired by object sensitivity, we formalize a novel analogue for functional languages called *closure sensitivity*. In this style of polyvariance, we view closures as the fundamental objects of higher-order languages (in the terminology of object-oriented languages, they are their own receivers) and associate them with closure-creation histories directly. No changes need to be made to our CPS language and its semantics.

$$\overbrace{((ae_f\ \ldots\ ae_j),\ \tilde{\rho},\ \_,\ (\tilde{\sigma}_O, \tilde{O}))}^{\tilde{\varsigma}} \overset{\text{Inst}}{\rightsquigarrow}_{clo(k)} (call',\ \tilde{\rho}',\ \_,\ (\tilde{\sigma}'_O, \tilde{O}'))$$
$$\text{where}\quad ((\lambda\ (x_0 \ldots x_j)\ call'), \tilde{\rho}_\lambda) \in \tilde{A}(ae_f, \tilde{\varsigma})$$
$$\widetilde{clo}_i \in \tilde{A}(ae_i, \tilde{\varsigma})$$
$$\tilde{\sigma}'_O = \tilde{\sigma}_O \sqcup \bigsqcup_{ae_i = (\texttt{lambda}\ \ldots)} [(\tilde{\rho}'(x_i), \widetilde{clo}_i) \mapsto \{take_k((ae_f\ \ldots\ ae_j) : \tilde{O})\}] \sqcup \bigsqcup_{ae_i = y_i} [(\tilde{\rho}'(x_i), \widetilde{clo}_i) \mapsto \tilde{\sigma}_O(\tilde{\rho}(y_i), \widetilde{clo}_i)]$$
$$\tilde{O}' \in \begin{cases} \{take_k((ae_f\ \ldots\ ae_j) : \tilde{O})\} & ae_f = (\texttt{lambda}\ \ldots) \\ \tilde{\sigma}_O(\tilde{\rho}(y_f), ((\lambda\ (x_0 \ldots)\ call'), \tilde{\rho}_\lambda)) & ae_f = y_f \end{cases}$$

Instead of vectors, closures created at the current call site become bound to the current allocation history (extended with the current call site) across each transition. Instead of the zeroth argument being used to determine successor-state allocation history, the value in call position is used.

$$\widetilde{alloc}_{\text{clo}}(x,\ (\_,\ \_,\ \_,\ (\tilde{\sigma}_O, \tilde{O}))) \triangleq (x, \tilde{O})$$

6.3 Argument Sensitivity

Agesen [1] introduces a Cartesian product algorithm (CPA) as an enhancement to a type recovery algorithm (which can be viewed as

an abstract interpretation where dynamic program types are used as abstract values). We will consider the source of imprecision that the original formulation attempts to address, generalize its solution as a form of polyvariance in our approach, and discuss CPA's complexity and precision relative to call and object sensitivity.

The basic algorithm, that CPA extends, assigns a flow set of dynamic types for each variable in the program, it establishes constraints based on the program text, and it propagates values until all these constraints have been met. The primary method for overcoming this merging, is introduced as the p-level expansion algorithm of Oxhøj et al. [43]—a polyvariant type-inference algorithm and analogue to the call-string histories of Harrison and then Shivers, where the use of p parallels that of k in k-CFA. This is shown to be insufficient however, as the authors of CPA give a case of merging which cannot be overcome by any value of p. Besson [5] further illustrates this point in the context of Java, claiming "CPA beats ∞-CFA".

The original motivating example for CPA was a polymorphic `max` function:

```
... (let ([max (lambda (a b) (if (> a b) a b))])
       ...)
```

Here, the only constraint for an input to `max` is that it support comparison, so a call `(max` "a" "at"`)` makes as much sense as a call `(max` 2 5`)`. However, if both these calls are made with a sufficient amount of obfuscating call (or object) history behind them, merging will cause the flow sets for both a and b to each include both `string` and `int` (i.e., abstract values for those types). This is imprecise as it implies that a call `(max` 2 "at"`)` is possible, even when it is not. The problem then, can be summarized as the existence of spurious inter-argument patterns which become inevitable when the flow sets for different syntactic arguments are entirely distinct.

The solution that CPA proposes is to replace flow sets of per-argument types, with flow sets of per-function tuples of types. In such an analysis, the function `max` itself would be typed $\{(\texttt{int}, \texttt{int}),\ (\texttt{string}, \texttt{string})\ \ldots\}$ preserving inter-argument patterns and eliminating spurious calls where the types don't match.

In essence, this change makes flow sets for each argument specific to the entire tuple of types received in a call. This suggests that, although no amount of call history will ensure the preservation of inter-argument correlations, a form of polyvariance which makes addresses specific to a tuple of abstract values for arguments can.

We must be careful here in extending this idea to an allocator for our CPS language. If a tuple of closures is included inside addresses, the mutual recursion of addresses, closures, and environments makes the analysis unbounded. Instead, we assume a helper function $\mathcal{T}$ which further abstracts abstract values so they cannot contain addresses. For an approach especially similar to CPA itself, we might define $\mathcal{T}$ so it yields types. For a functional language, we can define $\mathcal{T}$ so that it strips environments out of closures and leaves just a set of syntactic lambdas. For example:

$$\mathcal{T}(\tilde{d}) \triangleq \{lam \mid (lam, \tilde{\rho}) \in \tilde{d}\}$$

In a sense, syntactic lambdas are at least as specific as a type (their type signature, whatever type system is used) whether or not that type is known a priori by an analysis [14].

With this, we may define an *argument sensitive* style of polyvariance, like CPA, as an abstract allocator.

$$\widetilde{alloc}_{\text{CPA}}(x, \overbrace{((ae_f\ ae_0\ \ldots\ ae_j), \_, \_, \_)}^{\tilde{\varsigma}}) = (x, (\mathcal{T}(\tilde{\mathcal{A}}(ae_0, \tilde{\varsigma})), \ \ldots, \ \mathcal{T}(\tilde{\mathcal{A}}(ae_j, \tilde{\varsigma}))))$$

We can also observe how easy it would be to construct less precise variations of this allocator by including only some arguments within addresses. For example, including only the first argument might yield enough precision in many cases:

$$\widetilde{alloc}_{\text{arg}_0}(x, \overbrace{((ae_f\ ae_0\ \ldots\ ae_j), \_, \_, \_)}^{\tilde{\varsigma}}) = (x, \mathcal{T}(\tilde{\mathcal{A}}(ae_0, \tilde{\varsigma})))$$

We could even vary the arguments an analysis is sensitive to on a per-function basis like we did for polymorphic splitting in Section 6.1.2.

Like call sensitivity and object sensitivity, CPA can be of exponential complexity in the size of the program and is exceedingly impractical for use on sufficiently complex input programs. CPA is also, however, an excellent illustration of the principal that, in *practice*, more precision can also lead to smaller model sizes and faster analysis times. Where CPA improves precision, it is also fastest, and where CPA is unnecessary and delivers no improvement over k-CFA, it is enormously inefficient. For a function like `max`, one where the types of the arguments should match, CPA accumulates only a single value for each type that can flow to the function. For a function where all combinations of arguments are possible, CPA requires each combination to be enumerated explicitly. k-CFA *implies* all inter-argument combinations for equal precision at far greater efficiency. This would seem to support an effort to discover more adaptive variations on CPA.

6.4 Extreme-Precision Allocators

We can even further generalize the central idea of CPA to consider forms of polyvariance which preserves inter-address correlations in the store. What about an extreme case for the precision of an allocator where an analysis allocates addresses specific to entire stores (or portions of stores, or specific addresses in the store). As it turns out, we can even recover all the precision lost through structurally store widening as a form of store-sensitive polyvariance.

We assume the underlying allocator (in a store-sensitive setting) is $\widetilde{alloc}$ and its instrumentation is $\overset{\text{Inst}}{\rightsquigarrow}$. Using these, we may produce an instrumentation for recovering store sensitivity within a structurally store widened parametric semantics by rebuilding the state-specific environments and stores lost due to store widening.

$$\overbrace{((ae_f\ ae_0\ \ldots\ ae_j), \tilde{\rho}, \tilde{\sigma}, (\tilde{\iota}, \tilde{\rho}_\Sigma, \tilde{\sigma}_\Sigma))}^{\tilde{\varsigma}} \overset{\text{Inst}}{\rightsquigarrow}_{ss(\widetilde{alloc}, \overset{\text{Inst}}{\rightsquigarrow})} (call', \tilde{\rho}', \tilde{\sigma}', (\tilde{\iota}', \tilde{\rho}'_\Sigma, \tilde{\sigma}'_\Sigma))$$

where
$$((\lambda\ (x_0 \ldots x_j)\ \ call'), \tilde{\rho}_\lambda) \in \tilde{\mathcal{A}}(ae_f, \tilde{\varsigma})$$
$$\tilde{v}_i = \tilde{\mathcal{A}}(ae_i, \tilde{\varsigma})$$
$$\tilde{\rho}'_\Sigma = \tilde{\rho}_\lambda[x_i \mapsto \tilde{a}_i]$$
$$\tilde{\sigma}'_\Sigma = \tilde{\sigma}_\Sigma \sqcup [\tilde{a}_i \mapsto \tilde{v}_i]$$
$$\tilde{a}_i = \widetilde{alloc}(x_i, \tilde{\varsigma})$$
$$((ae_f\ ae_0\ \ldots\ ae_j), \tilde{\rho}, \tilde{\sigma}, \tilde{\iota}) \overset{\text{Inst}}{\rightsquigarrow} (call', \tilde{\rho}', \tilde{\sigma}', \tilde{\iota}')$$

We then use an allocator which embeds these recovered exact environments and stores to differentiate addresses.

$$\widetilde{alloc}_{\text{ss}}(x, (call, \tilde{\rho}, \tilde{\sigma}, (\tilde{\iota}, \tilde{\rho}_\Sigma, \tilde{\sigma}_\Sigma))) \triangleq (x, call, \tilde{\rho}_\Sigma, \tilde{\sigma}_\Sigma)$$

Using a similar instrumentation which rebuilds exact environments, we can also recover the full environment sensitivity lost through closure conversion, or the use of mCFA or poly-k-CFA.

$$\widetilde{alloc}_{\text{es}}(x, (call, \tilde{\rho}, \tilde{\sigma}, (\tilde{\iota}, \tilde{\rho}_\Sigma))) \triangleq (x, call, \tilde{\rho}_\Sigma)$$

In this way we can observe that some important forms of coarser structural abstraction (store widening and the use of flat environments) are encompassed by our design space for polyvariance.

6.5 Combining Forms of Polyvariance

For two forms of polyvariance, we may combine them by essentially taking the product of their instrumentations and the product of their allocators. Consider two forms of polyvariance characterized by $\widetilde{alloc}_0$ paired with $(\overset{\text{Inst}}{\rightsquigarrow}_0)$ and $\widetilde{alloc}_1$ paired with $(\overset{\text{Inst}}{\rightsquigarrow}_1)$, respectively.

We can produce a new instrumentation which compiles the information added by both $(\overset{\text{Inst}}{\rightsquigarrow}_0)$ and $(\overset{\text{Inst}}{\rightsquigarrow}_1)$:

$$(call,\ \tilde{\rho},\ \tilde{\sigma},\ (\tilde{\iota}_0, \tilde{\iota}_1)) \overset{\text{Inst}}{\rightsquigarrow}_\times (call',\ \tilde{\rho}',\ \tilde{\sigma}',\ (\tilde{\iota}'_0, \tilde{\iota}'_1))$$

$$\text{where} \quad (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota}_0) \overset{\text{Inst}}{\rightsquigarrow}_0 (call',\ \tilde{\rho}',\ \tilde{\sigma}',\ \tilde{\iota}'_0)$$

$$(call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota}_1) \overset{\text{Inst}}{\rightsquigarrow}_1 (call',\ \tilde{\rho}',\ \tilde{\sigma}',\ \tilde{\iota}'_1)$$

Likewise, we can produce a new allocator which returns an address specific to both addresses returned by $\widetilde{alloc}_0$ and $\widetilde{alloc}_1$:

$$\widetilde{alloc}_\times(x,\ (call,\ \tilde{\rho},\ \tilde{\sigma},\ (\tilde{\iota}_0, \tilde{\iota}_1))) \triangleq$$
$$(\widetilde{alloc}_0(x, (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota}_0)), \widetilde{alloc}_1(x, (call,\ \tilde{\rho},\ \tilde{\sigma},\ \tilde{\iota}_1)))$$

7. Conclusion

We have defined the polyvariance of abstract abstract machines, consistent with previous uses of the term, to mean the degree to which program values at runtime are differentiated into some number of distinct static approximations of their dynamic behavior. In a store-passing abstract interpreter, this differentiation is uniquely and entirely determined by the strategy used for the allocation of addresses.

To show both that all strategies for sound polyvariant analysis may be implemented as an allocator and that all allocators yield a sound polyvariant analysis, we employ the *a posteriori* soundness process of Might and Manolios and permit arbitrary instrumentation to guide the behavior of allocation within the core flow analysis. Our approach both allows us to easily recapitulate classic styles of polyvariance and to develop new variations and combinations of these. We are able to distinguish multiple flavors of call sensitivity, resolving an ambiguity in the original formulation of k-CFA, and encompass fundamental variations on the structure of an analysis.

Acknowledgments. The authors would like to thank the anonymous ICFP reviewers for their thorough and insightful feedback.

This material is partially based on research sponsored by DARPA under agreements number AFRL FA8750-15-2-0092 and FA8750-12-2-0106 and by NSF under CAREER grant 1350344. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

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

[2] T. Amtoft and F. Turbak. Faithful translations between polyvariant flows and polymorphic types. In *Programming Languages and Systems*, pages 26–40. Springer, 2000.

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

[4] A. Banerjee. A modular, polyvariant and type-based closure analysis. In *ACM SIGPLAN Notices*, volume 32, pages 1–10. ACM, 1997.

[5] F. Besson. CPA beats ∞-CFA. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, page 7. ACM, 2009.

[6] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *ACM SIGPLAN Notices*, volume 44, pages 243–262. ACM, 2009.

[7] P. Cousot. Types as abstract interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 316–331. ACM, 1997.

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

[9] P. Cousot and R. 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.

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

[11] C. Earl, M. Might, and D. Van Horn. Pushdown control-flow analysis of higher-order programs: Precise, polyvariant and polynomial-time. In *Scheme Workshop*, August 2010.

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

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

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

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

[16] T. Gilray, S. Lyde, M. D. Adams, M. Might, and D. V. Horn. Pushdown control-flow analysis for free. *Proceedings of the Symposium on the Principals of Programming Languages (POPL)*, January 2016.

[17] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. In *Proceedings of the European Conference on Object-oriented Programming*, pages 126–150, Berlin, Heidelberg, 2010.

[18] D. Guth. A formal semantics of python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, July 2013.

[19] W. L. Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 1989.

[20] S. Holdermans and J. Hage. Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. In *ACM Sigplan Notices*, volume 45, pages 63–74. ACM, 2010.

[21] S. Jagannathan and S. 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.

[22] S. Jagannathan, S. Weeks, and A. Wright. Type-directed flow analysis for typed intermediate languages. In *International Static Analysis Symposium*, pages 232–249. Springer, 1997.

[23] M. Jenkins, L. Andersen, T. Gilray, and M. Might. Concrete and abstract interpretation: Better together. In *Workshop on Scheme and Functional Programming*, 2015.

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

[25] J. I. Johnson, N. Labich, M. Might, and D. Van Horn. Optimizing abstract abstract machines. In *Proceedings of the International Conference on Functional Programming*, September 2013.

[26] N. D. Jones and S. 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.

[27] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Notices*, volume 48, pages 423–434. ACM, 2013.

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

[29] R. Koot and J. Hage. Type-based exception analysis for non-strict higher-order functional languages with imprecise exception semantics. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, pages 127–138. ACM, 2015.

[30] O. Lhoták. *Program analysis using binary decision diagrams*. PhD thesis, McGill University, 2006.

[31] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In *Compiler Construction*, pages 47–64. Springer, 2006.

[32] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(1):3, 2008.

[33] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the impact of context-sensitivity on andersen's algorithm for java programs. In *ACM SIGSOFT Software Engineering Notes*, volume 31, pages 6–12. ACM, 2005.

[34] S. Liang and M. Might. Hash-flow taint analysis of higher-order pograms. In *Proceedings of the Conference on Programming Language Analysis for Security*, June 2012.

[35] J. Midtgaard. Control-flow analysis of functional programs. *ACM Computing Surveys*, 44(3):10:1–10:33, Jun2012.

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

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

[38] M. Might and P. 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.

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

[40] M. Might, Y. Smaragdakis, and D. 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.

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

[42] M. Naik, A. Aiken, and J. Whaley. *Effective static race detection for Java*, volume 41. ACM, 2006.

[43] N. Oxhøj, J. Palsberg, and M. I. Schwartzbach. Making type inference practical. In *ECOOP92 European Conference on Object-Oriented Programming*, pages 329–349. Springer, 1992.

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

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

[46] G. D. Plotkin. A structural approach to operational semantics. 1981.

[47] J. G. Politz, M. J. Carroll, B. S. Lerner, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in javascript. In *Proceedings of the Dynamic Languages Symposium*, 2012.

[48] Racket Community. Racket programming language, 2015. URL `http://racket-lang.org/`.

[49] J. F. Ranson, H. J. Hamilton, and P. W. Fong. A semantics of Python in Isabelle/HOL. Technical Report CS-2008-04, Department of Computer Science, University of Regina, Regina, Saskatchewan, December 2008.

[50] I. Sergey, D. Devriese, M. Might, J. Midtgaard, D. Darais, D. Clarke, and F. Piessens. Monadic abstract interpreters. In *ACM SIGPLAN Notices*, volume 48, pages 399–410. ACM, 2013.

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

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

[53] Y. Smaragdakis, M. Bravenboer, and O. Lhotak. Pick your contexts well: Understanding object-sensitivity. In *Symposium on Principles of Programming Languages*, pages 17–30, January 2011.

[54] G. J. Smeding. An executable operational semantics for python. Master's thesis, Universiteit Utrecht, January 2009.

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

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

[57] D. Van Horn and M. Might. Abstracting abstract machines. In *International Conference on Functional Programming*, page 51, Sep 2010.

[58] D. Vardoulakis and O. 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.

[59] H. Verstoep and J. Hage. Polyvariant cardinality analysis for non-strict higher-order functional languages: Brief announcement. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, pages 139–142. ACM, 2015.

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