Under consideration for publication in J. Functional Programming

Abstract allocation as a unified approach to polyvariance in control-flow analyses

THOMAS GILRAY
University of Maryland, College Park
MICHAEL D. ADAMS MATTHEW MIGHT
University of Utah University of Alabama, Birmingham
(*e-mail:* tgilray@cs.umd.edu, adamsmd@cs.utah.edu, might@uab.edu)

Abstract

In higher-order settings, control-flow analysis aims to model the propagation of both data and control by finitely approximating program behaviors across all possible executions. The polyvariance of an analysis describes the number of distinct abstract representations, or variants, for each syntactic entity (e.g., functions, variables, or intermediate expressions). Monovariance, one of the most basic forms of polyvariance, maintains only a single abstract representation for each variable or expression. Other polyvariant strategies allow a greater number of distinct abstractions and increase analysis complexity with the aim of increasing analysis precision. For example, k-call sensitivity distinguishes flows by the most recent k call sites, k-object sensitivity by a history of allocation points, and argument sensitivity by a tuple of dynamic argument types. From this perspective, even a concrete operational semantics may be thought of as an unboundedly polyvariant analysis.

In this paper, we develop a unified methodology that fully captures this design space. It is easily tunable and guarantees soundness regardless of how tuned. We accomplish this by extending the method of abstracting abstract machines (AAM), a systematic approach to abstract interpretation of operational abstract-machine semantics. Our approach permits arbitrary instrumentation of the underlying analysis and arbitrary tuning of an abstract-allocation function. We show that the design space of abstract allocators both unifies and generalizes existing notions of polyvariance. Simple changes to the behavior of this function recapitulate classic styles of analysis and yield novel combinations and variants.

1 Introduction

In the past 40 years since call-sensitive data-flow analysis was introduced by Sharir & Pnueli (1978) (Sharir & Pnueli, 1981), a wide variety of both subtly and essentially distinct forms of polyvariant static analysis have been explored in the literature. The *polyvariance* of a finite flow analysis, broadly construed, is the degree to which program control flows and data flows are broken into multiple distinct static approximations of their dynamic behavior. This is consistent with previous uses of the term, although the exact nature of its diversity of uses has not previously been well explored or formalized.

For example, consider a function applied to different values across more than one call site such an identity function applied to both true and false in Racket (Racket Community, 2015):

```
(let ([id (λ (x) x)])
  (id #f)
  (id #t))
```

A *monovariant* analysis is one that, for each syntactic variable or intermediate expression, maintains only one approximation, variant, or flow set. For example, although `x` is bound to `#f` when called from the first call site (i.e., `(id #f)`) and `#t` when called from the second call site (i.e., `(id #t)`), a monovariant analysis merges these to produce only a single flow set $\{\texttt{\#t}, \texttt{\#f}\}$ for `x` (assuming `#t` and `#f` are not further approximated).

A more *polyvariant* analysis allows a larger number of distinct flow sets. This potentially increases analysis complexity, analysis precision, or both, depending on the analysis target. The seminal and still most widely used form of polyvariance, *k-call sensitivity*, maintains one flow set for each call history of length k that precedes a variable binding. In our example, a 1-call sensitive analysis (e.g., Shivers' 1-CFA) keeps `#t` and `#f` from merging. It maintains two distinct flow sets for `x`. One is for `x` when `id` is called from the first call site `(id #f)`. The other is for `x` when `id` is called from the second call site `(id #t)`.

A wide gamut of polyvariant techniques has been discussed in the literature (Agesen, 1995; Amtoft & Turbak, 2000; Banerjee, 1997; Bravenboer & Smaragdakis, 2009; Gilray & Might, 2013, 2014; Gilray *et al.*, 2016b; Harrison, 1989; Holdermans & Hage, 2010; Jagannathan *et al.*, 1997; Jones & Muchnick, 1982; Koot & Hage, 2015; Lhoták, 2006; Lhoták & Hendren, 2006, 2008; Liang *et al.*, 2005; Milanova *et al.*, 2005; Naik *et al.*, 2006; Oxhøj *et al.*, 1992; Palsberg & Pavlopoulou, 2001; Sharir & Pnueli, 1981; Shivers, 1991; Smaragdakis *et al.*, 2011; Verstoep & Hage, 2015; Wright & Jagannathan, 1998). These include both subtle variations and disparate strategies. They use a variety of techniques and support many different applications. While many share common elements, there is little work on connecting and unifying different implementations and strategies (Amtoft & Turbak, 2000; Gilray & Might, 2014; Smaragdakis *et al.*, 2011).

We present a new methodology that both unifies and generalizes the myriad strategies for polyvariance. We show that the design space of polyvariance uniquely and exactly corresponds to the design space of tunings of an abstract allocator and that no tuning leads to an unsound analysis. Classic flavors of polyvariance are recapitulated by our methodology, and we can derive novel variations of each. By proving that no allocation tuning is unsound and by permitting arbitrary instrumentation of a core flow analysis, we show that *all conceivable* sound strategies for polyvariance can be implemented by a parametric abstract semantics.

1.1 Contributions

We expand and clarify our proceedings paper (Gilray *et al.*, 2016a). In this version we make the following new additions:

- We expand the background material, adding examples for several styles of polyvariance in section 2 and add clarifying examples throughout.
- We present a self-contained development of AAM in section 4.
- We expand on our treatment of the *a posteriori* soundness process and its proof in section 5 for clarity and self-containment.

- We switch to a direct-style presentation both to improve clarity, accessibility and to connect the allocation approach with our solution to the call-return-matching problem for higher-order control-flow analysis (Gilray *et al.*, 2016b). This result is based entirely on the approach described in the present paper and its result is discussed in section 5.
- We detail the traditional Galois connection used in an AAM-style analysis and the parametric Galois connection used in our approach.

1.2 Outline

Section 2 introduces the design space of existing strategies for polyvariance and explores the central challenges in designing effective and efficient forms of polyvariance. This motivates our new approach.

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

Section 4 reviews the AAM methodology, formalizing both a concrete semantics and an abstract semantics that approximates it. We discuss crucial concepts such as store widening and soundness. More experienced readers may want to skim over this section as it is background for our approach.

Section 5 discusses the role of allocation within AAM. It 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 utility of leaving instrumentation as an open parameter. We present the most liberal constraint possible for the *a posteriori* soundness process to be employed. This section provides a general, tunable framework that instantiates our approach and gives a self-contained development of the *a posteriori* soundness process in its full generality.

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

2 Myriad Styles of Polyvariance

Polyvariance has a long history with many variants proposed. Following Sharir & Pnueli (1981), call sensitivity was used by Jones & Muchnick (1982) and Harrison (1989) in the ‘80s and then generalized to control-flow analysis of higher-order languages (k-CFA) by Shivers (1991). 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 & Weeks (1995) and the *Cartesian product algorithm* (CPA) by Agesen (1995), an enhancement for type recovery algorithms. A variety of polyvariant type systems emerged, the majority of which are call sensitive (Amtoft & Turbak, 2000; Banerjee, 1997; Holdermans & Hage, 2010; Koot & Hage, 2015; Oxhøj *et al.*, 1992;

Palsberg & Pavlopoulou, 2001; Verstoep & Hage, 2015). Ideas from type systems found their way into flow analyses (Amtoft & Turbak, 2000; Cousot, 1997). For example, inspired by `let`-polymorphism, Wright & Jagannathan (1998) presents *polymorphic splitting*, a style of call sensitivity that varies the degree of sensitivity per-function based on the `let`-depth. Milanova *et al.* (2005) introduces another 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 hierarchy of increasingly precise analyses that reach concrete (precise) evaluation only in the 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, as object sensitivity correlates well with dynamic dispatch behavior (Bravenboer & Smaragdakis, 2009; Lhoták, 2006; Lhoták & Hendren, 2006, 2008; Liang *et al.*, 2005; Naik *et al.*, 2006). More recently, Smaragdakis *et al.* (2011) generalizes object sensitivity to a range of variations and introduces a new approximation called *type sensitivity* that retains only the type information needed to coalesce similar dynamic dispatches.

These different styles of polyvariance can be viewed as heuristics for managing the trade-off between complexity and precision in a static analysis. Call sensitivity supposes that program values correlate with recent call sites (or the surrounding stack frames). Object sensitivity supposes that values 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 correlate with program values for other arguments to the same function. Polymorphic splitting supposes that more deeply nested function definitions benefit from more call history. For programs for which these are good heuristics, more precise flow sets result. For programs for which they are not, less precise flow sets result. Each strategy for polyvariance represents a gambit on the part of an analysis designer that the programs being analyzed behave in certain ways. In the remainder of this section, we explore a few of these and the trade-offs they make.

2.1 Call Sensitivity (k-CFA)

Consider the following uses of an identity function on two different types:

```
(let ([id (λ (x) x)])
  (id 0)
  (id "a"))
```

In a monovariant type-recovery with no context sensitivity and only one abstraction for each piece of syntax in a program, there is only one set in which to store possible types for the variable `x`. Thus, because both the `int` and `string` types flow to `x`, both are conflated as they return from `id`. This means that `string` is incorrectly reported as a possible return type for `(id 0)`, and `int` is incorrectly reported as a possible return type for `(id "a")`.

In order to distinguish between these two calls, call-sensitive flow analyses consider the two invocations of `id` as different contexts in which to analyze the function body. For example, a 1-call-sensitive analysis, like 1-CFA, tracks the most-recent call site and keeps separate approximations for each variable and most-recent call site. Thus, the set of types

reaching `x` when the most-recent call is `(id 0)` is separate from than those reaching `x` when the most-recent call is `(id "a")`. The first contains only the type `int` and the later only `string`. Thus, the set of return types reported for `(id 0)` contains just `int`, and the set of return types reported for `(id "a")` contains just `string`.

While differentiating data flows by the most-recent call site eliminates spurious flows in the preceding example, conflation can still occur if we indirect through another function call. For example, consider the following code that effectively η-expands `id`.

```
(let* ([id₁ (λ (x₁) x₁)]
       [id₀ (λ (x₀) (id₁ x₀))])
  (id₀ 1)
  (id₀ "b"))
```

A 1-CFA keeps the values of x_0 distinct for the two calls to id_0 since these are made from different call sites. However, these result in two intermediate calls to id_1 that both have the same latest call site in the body of id_0. Thus, there is only one approximation of x_1, and values are still conflated. Once again, the analysis will spuriously report `string` as a possible return type of $(id_0\ 1)$ and `int` as a possible return type of $(id_0\ \texttt{"b"})$.

This can be fixed by tracking the last *two* call sites instead of only the last call site, which is exactly what a 2-call-sensitive analysis (like 2-CFA) does. However, this can be defeated by adding yet another level of indirection. In general any k-CFA, which tracks the last k call sites, can be defeated by code of the following form:

```
(let* ([idₖ   (λ (xₖ)   xₖ)]
       [idₖ₋₁ (λ (xₖ₋₁) (idₖ xₖ₋₁))]
       ...
       [id₁   (λ (x₁)   (id₂ x₁)])
       [id₀   (λ (x₀)   (id₁ x₀))])
  (id₀ 2)
  (id₀ "c"))
```

The initial id_0 through id_{k-1} ensure that the last k call sites are the same so there is only one variant of x_k and thus values passed to id_0 are conflated.

In addition to its effect on precision, the choice of polyvariance can have a significant impact on how long an analysis takes. Consider the following change to our example where two call sites exist at every intermediate function:

```
(let* ([idₖ   (lambda (xₖ)   xₖ))]
       [idₖ₋₁ (lambda (xₖ₋₁) (idₖ xₖ₋₁) (idₖ xₖ₋₁))]
       ...
       [id₁   (lambda (x₁)   (id₂ x₁)   (id₂ x₁))])
       [id₀   (lambda (x₀)   (id₁ x₀)   (id₁ x₀))])
  (id₀ 1)
  (id₀ "at"))
```

Now each invocation of id_0 results in a number of call histories that is exponential in k—that is, all combinations of either the first call site or the second call site for all of id_0 through id_{k-1} or each of 2^k distinct call histories. Because this analysis is k-call

sensitive, it distinguishes (and thus accumulates) all these call histories for bindings to $\texttt{id}_{\texttt{k-1}}$. Then, due to the additional function $\texttt{id}_\texttt{k}$, all these values are conflated and we gain no additional precision. This illustrates how the precision and complexity of a flow analysis can vary independently across a variety of targets. A style of analysis may prove to be a good compromise in effectiveness and efficiency for some programs or idioms while neither effective nor efficient for others.

2.2 Argument sensitivity (CPA)

To further illustrate the importance of choosing a polyvariance fitting the task, consider a `max` function like the following:

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

As before, a 1-call sensitive analysis is 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). This is not unsound because neither of these are spurious values for `a`. However, spurious inter-argument patterns are being implied between the approximations for `a` and `b`. It appears that `max` could be invoked with an integer in one argument and a string in the other. To eliminate this kind of imprecision, we need to chose a different polyvariance. For example, the Cartesian product algorithm (CPA) builds tuples of arguments for each function, which preserves inter-argument patterns and thus excludes 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 handles them at no additional cost. For different programs or even components in a single program, different styles of polyvariance can exhibit very different degrees of precision or performance.

2.3 Object Sensitivity

Another style of polyvariance, closely related to argument sensitivity, is *object sensitivity*. Object sensitivity tracks the syntactic allocation point of each object in the program and differentiates bindings at function calls by the program point where the receiving object was instantiated. A k-object sensitive analysis extends this concept to include the allocation point of the object that allocated an object and so forth, forming a hierarchy of increasingly precise analyses. It can be considered a kind of argument sensitivity, but with two differences. First, it considers only the implicit `this` argument. Second, it uses allocation-point history instead of dynamic type. (Though there is a variant of object sensitivity, called *type sensitivity*, described by Smaragdakis *et al.* (2011), that does use dynamic types instead of allocation histories.)

To illustrate this behavior, consider the chain of identity functions from Section 2.1 translated into the following object-oriented code.

```
a = new I
b = new I

a.id0(0)
b.id0("a")
```

```
class I {
  def id0(x0) = this.id1(x0)
  def id1(x1) = this.id2(x1)
  ...
  def idk(xk) = xk
}
```

Object sensitivity differentiates each call to $\texttt{id}_0$ through $\texttt{id}_k$ into two allocation sites for `this`. One is for the allocation point `a = new I` and the other for `b = new I`. Because the explicit argument (i.e., `0` or `"a"`) to each of the two calls to $\texttt{id}_0$ co-varies with the implicit `this` argument, no conflation of types occurs, no matter the size of k. While a k-call-sensitive analysis merges both values in the final call to $\texttt{id}_k$, a 1-object-sensitive analysis keeps them separate. If each identity function $\texttt{id}_0$ through $\texttt{id}_{k-1}$ were to call its successor twice, k-call sensitivity would yield an exponential-time analysis while 1-object sensitivity or type sensitivity would both yield a polynomial-time analysis.

A common object-oriented idiom that object sensitivity handles particularly well is dynamic dispatch. Consider a simple example of dynamic dispatch where two different `Ape` subtypes implement different calls.

```
class Bonobo {
  def talk() = "shriek"
}

class Chimp {
  def talk() = "hoot"
}

def main() = {
  var b = new Bonobo
  var c = new Chimp
  b.printTalk()
  c.printTalk()
}
```

```
class Ape {
  def printTalk() = {
    var bonmot = this.talk()
    print(bonmot)
  }

  def virtual talk()
}
```

As in the previous example, because the two distinct calls correlate with their receiving object, both are kept distinct through the indirection of multiple method calls. In this case, dynamic dispatch is also done correctly and its control flow kept separate. The first call to `this.talk()` reaches only the bonobo's implementation and returns only `"shriek"`. The second reaches only the chimp's implementation and returns only `"hoot"`.

Generally, objects allocated at the same line of code may have similar behaviors that are reasonable to conflate and keep together. However, where two objects allocated at the same point exhibit different behaviors, important distinctions for an analysis may be lost, and where two objects allocated at different points exhibit similar behaviors, opportunities to coalesce their abstractions and reduce analysis complexity may also be missed.

2.4 Toward Better Trade-offs

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 (Wright & Jagannathan, 1998). While establishing better guarantees of analysis efficiency does correlate inversely with guarantees of analysis precision in terms of the worst case, analyses with more precision for data flows often have more precision for control flows and thus explore a smaller overall model. Scaling polyvariant flow analysis to large programs hinges on making good trade-offs and exploiting this paradox. Otherwise, for nearly all the varieties mentioned, the use of polyvariance is exponential in the worst case (Might *et al.*, 2010; Van Horn & Mairson, 2008). What seems to be needed are increasingly nuanced, introspective, and adaptive forms of polyvariance that better suit their targets and the properties we may wish to prove or discover for them. 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

In this paper, we develop a unified approach to encoding *all* and *only* sound forms of polyvariance as tunings of an allocation function. Thus the main idea is that *allocation characterizes polyvariance*. The design space of allocation strategies fully covers the design space of polyvariant strategies and leads us to a convenient implementation approach. Classic flavors of polyvariance can be recapitulated using our methodology, and we are able to derive novel variations of each. Furthermore, all possible allocation strategies 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 & Manolios (2009) to show that every allocator results in a sound analysis. To guide the allocator, we can arbitrarily instrument the analysis, and so long as the instrumentation affects only the allocator and the addresses it produces, it never leads to an unsound analysis. Furthermore, any form of polyvariance is expressible in terms of an allocator and instrumentation. This is because polyvariance concerns how flow sets are merged and differentiated. In a store-passing-style interpreter, this is determined by address allocation.

In Figure 1, we summarize the styles of polyvariance we survey in Section 6. For each, there is an allocation function and instrumentation that encode it. For example, the instrumentation for k-call sensitivity tracks k-length call histories, so the allocator can choose addresses unique to both the variable being allocated for and the current call history.

4 Abstracting Abstract Machines

Our approach builds upon the methodology of abstracting abstract machines (AAM), which we review in this section by systematically developing a concrete operational semantics

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	$\widetilde{alloc}_{\text{1CFA}}(x,(e,\_,\_,\_,\_,\_)) \triangleq (x,e)$
	None
Call-only Sensitivity	$\widetilde{alloc}_{\text{call}}(x,(\_,\_,\_,\_,\_,\tilde{\iota})) \triangleq (x,\tilde{\iota})$
	Tracks a history of (only) call sites
Call+Return Sensitivity	$\widetilde{alloc}_{\text{call}}(x,(\_,\_,\_,\_,\_,\tilde{\iota})) \triangleq (x,\tilde{\iota})$
	Tracks a history of call and return points
Polymorphic Splitting	$\widetilde{alloc}_{\text{call}}(x,(\_,\_,\_,\_,\_,\tilde{\iota})) \triangleq (x,\tilde{\iota})$
	Tracks a history of call sites
Object Sensitivity	$\widetilde{alloc}_{\text{obj}}(x,(\_,\_,\_,\_,\_,(\tilde{\sigma}_O,\tilde{o}))) \triangleq (x,\tilde{o})$
	Tracks a history of per-object allocation points
Cartesian Product Algorithm	$\widetilde{alloc}_{\text{CPA}}(x,(\texttt{(let ([y (}f\ ae_0\ \ldots\texttt{)]) e)},\ldots)) \triangleq (x,(\ldots\tilde{\mathscr{T}}(\tilde{\mathscr{A}}(ae_i,\tilde{\varsigma}))\ldots))$
	None
0^{th}-Argument Sensitivity	$\widetilde{alloc}_{arg_0}(x,(\texttt{(let ([y (}f\ ae_0\ \ldots\texttt{)]) e)},\ldots)) \triangleq (x,\tilde{\mathscr{T}}(\tilde{\mathscr{A}}(ae_0,\tilde{\varsigma})))$
	None
Store Sensitivity	$\widetilde{alloc}_{\text{ss}}(x,(\_,\_,\_,\_,\_,(\tilde{\iota},\tilde{\rho}_\Sigma,\tilde{\sigma}_\Sigma))) \triangleq (x,\tilde{\iota},\tilde{\rho}_\Sigma,\tilde{\sigma}_\Sigma)$
	Rebuilds per-state stores lost via store widening
Concrete Evaluation	$\widetilde{alloc}_{\bot}(x,(\_,\_,\tilde{\sigma},\_,\_,\_))) \triangleq (x,\lvert dom(\tilde{\sigma})\rvert)$
	None (or full execution history)

Fig. 1: A selection of allocators. (The notation used for allocation functions is explained in the course of later sections.)

for a simple functional intermediate representation into an approximating simulation of the same. Additionally we discuss the traditional strategies for proving soundness (correctness), extensions for richer languages, and a store-widening transformation (an essential approximation for obtaining a polynomial-time analysis).

Static analysis by abstract interpretation proves properties of programs by running code through an interpreter powered by an *abstract semantics* that approximates the behavior of a *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 (Cousot & Cousot, 1976, 1977, 1979; Midtgaard, 2012). 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 (Johnson *et al.*, 2013; Might, 2010; Van Horn & Might,

2010). Control-flow analysis is generally straightforward in first-order languages, but in the presence of first-class functions, it requires a simultaneous modeling of how both data and control propagate; without knowing which functions reach an application we cannot know where control jumps to or which behavioral arguments (lambdas) may flow to its parameters.

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 approximations 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 allocating 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 (Plotkin, 1981) for a simple direct-style language, specifically, the untyped, call-by-value λ-calculus in *administrative normal form* (ANF) (Flanagan *et al.*, 1993). ANF requires an administrative binding for all intermediate values so that applications (and other operations, conditionals, primitive operations, etc.) can in an atomic step lookup the value of the function to be applied and its arguments. This also reifies a particular order of operations as a stack of `let` bindings for administrative variables. For example, if we transform the code in the left column of the following into ANF, we get the code in the right column.

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

```
(define (fact n)
  (let ([i0 (= n 0)])
    (if i0
      1
      (let ([i1 (- n 1)])
        (let ([i2 (fact i0)])
          (let ([i3 (* n i2)])
            i3))))))
```

ANF is a widely used intermediate representation, for example in the Glasgow Haskell Compiler (Maurer *et al.*, 2017). Contrasted with continuation-passing style (CPS) (Appel, 2007; Kennedy, 2007), ANF has implicit continuations, extended at let forms by the language's interpretation. Having all intermediate values bound to variables, such as i_0 through i_3 in the factorial example, is convenient for analysis, and having continuations created by the abstract machine (as opposed to being reified as syntax) is convenient for discussing our approach to continuation polyvariance. The grammar structurally distin-

guishes between call sites e and atomic expressions ae:

$$
\begin{array}{rll}
e \in \mathsf{Exp} ::= & \texttt{(let ([}x\ (f\ ae)\texttt{]) }e\texttt{)} & \text{[call]} \\
\mid & ae & \text{[return]} \\
f, ae \in \mathsf{AExp} ::= & x \mid lam & \text{[atomic expressions]} \\
lam \in \mathsf{Lam} ::= & \texttt{(}\lambda\ \texttt{(}x\texttt{) }e\texttt{)} & \text{[lambda abstractions]} \\
x, y \in \mathsf{Var} \text{ is a set of identifiers} & & \text{[variables]}
\end{array}
$$

Instead of specifically affixing each expression with a unique label, we assume two identical expressions occurring separately in a program are not syntactically equal. This language only permits unary lambdas, but extrapolating our discussion to n-ary lambdas is straightforward.

We define the evaluation of programs in this language using a relation ($\rightarrow_\varsigma$), over states of an abstract-machine for a concrete semantics, which determines how the machine transitions from one state to another. (Sigma here is just a tag, part of the relation's name, to indicate the kind of transition and is not a parameter; this is the case for such subscripts throughout this paper.) As opposed to using big-step (i.e., natural) operational semantics (Kahn, 1987), we simplify our forthcoming finitization of this machine by using a small-step semantics—one where every individual step must terminate. This is accomplished by explicitly managing a stack or continuation and moves all non-termination behavior into the collecting semantics, which is the overall evaluation of a program across a series of individual steps. States (ς) range over control expression (a call site), binding environment, and continuation components:

$$
\begin{array}{rlrl}
\varsigma \in \Sigma & \triangleq \mathsf{Exp} \times \mathit{Env} \times \mathit{Kont} & \phi \in \mathit{Frame} & \triangleq \mathsf{Var} \times \mathsf{Exp} \times \mathit{Env} \\
\rho \in \mathit{Env} & \triangleq \mathsf{Var} \rightharpoonup \mathit{Value} & v \in \mathit{Value} & \triangleq \mathit{Clo} \\
\kappa \in \mathit{Kont} & \triangleq \overrightarrow{\mathit{Frame}} & \mathit{clo} \in \mathit{Clo} & \triangleq \mathsf{Lam} \times \mathit{Env}
\end{array}
$$

Binding environments (ρ) map variables in scope to values. The environment is a partial map and accumulates points as execution progresses. Continuations (κ) are sequences of stack frames (ϕ), which each contain a variable, an expression, and an environment. In this simplified language, values may only be closures (clo), which are syntactic lambda terms (lam) paired with environments.

Evaluation of atomic expressions is handled by an auxiliary function, $\mathscr{A}$, which produces a value (v) for an atomic expression in the context of a state (ς). This is done by a lookup in the environment 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 by this helper.

$$\mathscr{A} : \mathsf{AExp} \times \Sigma \rightharpoonup \mathit{Value}$$
$$\mathscr{A}(x,\ (e,\rho,\kappa)) \triangleq \rho(x)$$
$$\mathscr{A}(\mathit{lam},\ (e,\rho,\kappa)) \triangleq (\mathit{lam},\ \rho)$$

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

$$\overbrace{(\texttt{(let ([}y\ (f\ ae)\texttt{]) }e\texttt{)}, \rho, \kappa)}^{\varsigma} \rightarrow_\Sigma (e', \rho', \kappa'), \text{ where}$$

$$\begin{aligned}(\texttt{(}\lambda\ \texttt{(}x\texttt{) }e'\texttt{)}, \rho_\lambda) &= \mathscr{A}(f,\ \varsigma)\\ \rho' &= \rho_\lambda[x \mapsto \mathscr{A}(ae,\ \varsigma)]\\ \kappa' &= (y, e, \rho) : \kappa\end{aligned}$$

Execution steps to the body of the applied lambda (as determined by the atomic evaluation of f). This closure's environment (ρ_λ) is extended with a binding for the formal parameter (x) to the atomic evaluation of ae. The current continuation is extended with a stack frame containing the variable to assign (y) and the expression (e) and environment (ρ) to reinstate.

The rule governing returns is:

$$\overbrace{(ae, \rho, (x, \rho, e) : \kappa)}^{\varsigma} \rightarrow_\Sigma (e, \rho', \kappa), \text{ where}$$

$$\rho' = \rho[x \mapsto \mathscr{A}(ae,\ \varsigma)]$$

Control propagates to the expression encoded in the top continuation frame (e) and modifies its environment (ρ) by adding a binding for x to the return value. A state becomes stuck if a return point is reached under the empty continuation or if the program is malformed (e.g., a free variable is encountered).

To fully evaluate a program e_0 using these transition rules, we *inject* it into our state space using the helper $\mathscr{I} : \mathsf{Exp} \rightarrow \Sigma$ where:

$$\mathscr{I}(e) \triangleq (e, \varnothing, \varepsilon)$$

We may now perform the standard lifting of $(\rightarrow_\Sigma)$ to a collecting semantics defined over a set of states $s \in S \triangleq \mathscr{P}(\Sigma)$. Our collecting relation $(\rightarrow_s)$ is a monotonic, total function that gives a set including the trivially reachable state $\mathscr{I}(e_0)$ plus the set of all states immediately succeeding those in its input.

$$s \rightarrow_s s', \text{where}$$

$$s' = \{\varsigma' \mid \varsigma \in s \wedge \varsigma \rightarrow_\Sigma \varsigma'\} \cup \{\mathscr{I}(e_0)\}$$

If the program e_0 terminates, then 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 e_0's full program trace for some $n \in \mathbb{N}$ whenever e_0 is a terminating program. No such n is guaranteed to exist in the general case (when e_0 is non-terminating) as our language (the untyped, call-by-value, ANF λ-calculus) is Turing-equivalent, our semantics is fully precise, and the state space

we defined is infinite. Whether or not it is finite, however, this collecting semantics gives us a well-defined fixed point to approximate (Tarski, 1955). Take note that the cause of this unboundedness is the set of unbounded stacks (i.e., continuations) and the mutual recursion of environments and values (in this case, just closures).

4.2 An Abstract Operational Semantics

Now that we have formalized the concrete semantics of our language as iteration to a (possibly infinite and incomputable) fixed point, we are ready to design a computable approximation of this fixed point (the exact program trace) using the tools of abstract interpretation. Previous work on abstracting abstract machines (AAM) has explored a variety of approaches to systematically abstracting a semantics like these (Johnson *et al.*, 2013; Might, 2010; Van Horn & Might, 2010). Broadly construed, these changes simultaneously finitize the domains of our machine while introducing non-determinism into both the transition relation (multiple successor states may immediately follow a predecessor state) and value store (multiple values may become conflated). We use value stores to model the heap and a finite address space to cut the otherwise mutually recursive structure of values (closures) and environments and of continuations/stacks. (Without addresses and value stores, the stack is unbounded and environments map variables directly to closures that each contain an environment). A finite address space then yields a finite state space overall and ensures computability of the analysis. This process can be systematized as a series of three steps (Might, 2010).

Step 1: "Cut" direct recursion. Each source of direct recursion in the abstract machine's domains is "cut" by store allocation, indirecting the self-reference through an address set which can then be finitized. Below are the domains of our modified concrete semantics where (1) and (2) mark the two main sets of related changes—we will detail these two changes next.

$$\begin{aligned}
\varsigma \in \Sigma &\triangleq \mathsf{Exp} \times \mathit{Env} \\
&\quad \times \mathit{Store} \times \mathit{KStore} \times \mathit{KAddr} \\
\rho \in \mathit{Env} &\triangleq \mathsf{Var} \rightharpoonup \mathit{Addr} \quad (1) \\
\sigma \in \mathit{Store} &\triangleq \mathit{Addr} \rightharpoonup \mathit{Value} \quad (1) \\
a \in \mathit{Addr} &\triangleq \mathsf{Var} \times \mathbb{N} \quad (1) \\
a_\kappa \in \mathit{KAddr} &\triangleq \mathsf{Exp} \times \mathbb{N} \quad (2)
\end{aligned}$$

$$\begin{aligned}
s \in S &\triangleq \mathscr{P}(\Sigma) \\
\sigma_\kappa \in \mathit{KStore} &\triangleq \mathit{KAddr} \rightharpoonup \mathit{Kont} \quad (2) \\
\kappa \in \mathit{Kont} &\triangleq \mathit{Frame} \times \mathit{KAddr} \quad (2) \\
\phi \in \mathit{Frame} &\triangleq \mathsf{Var} \times \mathsf{Exp} \times \mathit{Env} \\
v \in \mathit{Value} &\triangleq \mathit{Clo} \\
\mathit{clo} \in \mathit{Clo} &\triangleq \mathsf{Lam} \times \mathit{Env}
\end{aligned}$$

We have made two principal changes. First, the mutual recursion of environments and closures has been cut using the address set *Addr*. Environments now map variables in scope to heap addresses (a), and another top-level component, value stores (σ), maps these addresses to values. Previously, binding environments mapped variables directly to values

(closures) which themselves contain environments (making the set of possible environments unbounded); changes marked (1) have introduced an unbounded set of addresses that must be used to construct all bindings. This simplifies the unboundedness of binding environments and makes it more explicit as an infinite set of addresses. As our refactoring progresses, this change will allow us to define a *finite* set of *abstract* addresses to obtain a bounded and approximate program analysis.

Second, the unboundedness of stacks (continuations) has similarly been cut; these changes are marked (2). Continuations now pair a frame with a continuation address (a_κ) which references the tail of the stack in another top-level component, continuation stores (σ_κ), that map these addresses to stored continuations. The current continuation is now simply a reference (a_κ) to a continuation stored in the current σ_κ. As elsewhere, non-numeric subscripts (in this case, kappa) should be interpreted as part of the symbol; this distinguishes continuation stores from value stores. Value-store addresses and continuation-store addresses may be chosen to be any unbounded set (such as $\mathbb{N}$). The choice of pairing a variable name with a fresh natural number (or program expression in the case of continuation addresses) is somewhat more convenient both for defining our (forthcoming) transition relation and notion of approximation.

Together these two changes requires that the set of states (Σ) be redefined. A state now contains both a value store (σ) and a continuation store (σ_κ) and the current continuation has been replaced by a current continuation address that refers to a stack encoded in σ_κ.

Atomic evaluation operates the same as before in the case of closure creation but for variable references must lookup an address in the environment before looking up that address in the store:

$$\mathscr{A} : \mathsf{AExp} \times \Sigma \rightharpoonup \mathit{Value}$$
$$\mathscr{A}(x,\ (e,\rho,\sigma,\sigma_\kappa,a_\kappa)) \triangleq \sigma(\rho(x))$$
$$\mathscr{A}(\mathit{lam},\ (e,\rho,\sigma,\sigma_\kappa,a_\kappa)) \triangleq (\mathit{lam},\ \rho)$$

The transition rule for application reflects these changes by updating the value store and continuation store:

$$\overbrace{(\texttt{((let ([}y\ (f\ ae)\texttt{]) }e\texttt{)},\rho,\sigma,\sigma_\kappa,a_\kappa)}^{\varsigma} \rightarrow_\Sigma (e',\rho',\sigma',\sigma'_\kappa,a'_\kappa), \text{ where}$$

$$\begin{aligned}
(\texttt{((}\lambda\ \texttt{(}x\texttt{) }e'\texttt{)},\rho_\lambda) &= \mathscr{A}(f,\ \varsigma)\\
\rho' &= \rho_\lambda[x \mapsto a]\\
\sigma' &= \sigma[a \mapsto \mathscr{A}(ae,\ \varsigma)]\\
\sigma'_\kappa &= \sigma_\kappa[a'_\kappa \mapsto ((y,e,\rho),a_\kappa)]\\
a &= (x, |\mathit{dom}(\sigma)|)\\
a'_\kappa &= (e', |\mathit{dom}(\sigma_\kappa)|)
\end{aligned}$$

The crucial change is that two addresses (a and a'_κ for the argument to f and its continuation respectively) are allocated and the unboundedness of environments and stacks is now encoded only by the unboundedness of the sets these addresses are drawn from. One viable strategy for allocating fresh addresses is simply to pair the variable name with the current size of the store (i.e., $a = (x, |\mathit{dom}(\sigma)|)$). There is no garbage collection in these semantics,

so each time addresses are allocated the size of the current store strictly increases and is never reduced. Even assuming the natural extension to n-ary lambdas, as no two parameters bound in a single transition may share a variable name, all generated addresses will still be allocated fresh. Likewise, the strategy used for allocating fresh continuation addresses is to combine the target function body (which "owns" its own continuations in a sense) and the current size of the continuation store. As a program makes a series of non-tail function calls, evaluation builds up a linked list of continuation frames in the continuation store, σ_κ.

For example, at a call site `(let ([z ((λ (y) e_y) g)]) e_body)`, if a function with a parameter named y is applied on a value clo_1 (here bound to g via the current environment, ρ, and store, σ), a fresh address will be generated that binds clo_1 to y using an address based on the current size of the store. If 341 addresses have previously been allocated in the value store, it will be updated with a binding $(y, 341) \mapsto clo_1$ and the current environment will be updated with a binding $y \mapsto (y, 341)$. Likewise, the continuation store will be updated with a binding like $(e_y, 239) \mapsto ((z, e_{body}, \rho), a_\kappa)$ where ρ is the current environment before `(λ (y) e_y)` is applied, 239 is the previous number of continuation addresses allocated, and a_κ points to the current stack before the state transition.

Return transitions are more straightforward:

$$\overbrace{(ae, \rho, \sigma, \sigma_\kappa, a_\kappa)}^{\varsigma} \rightarrow_\varsigma (e, \rho', \sigma', \sigma_\kappa, a'_\kappa), \text{ where}$$

$$\begin{aligned} ((x, e, \rho_\kappa), a'_\kappa) &= \sigma_\kappa(a_\kappa) \\ \rho' &= \rho_\kappa[x \mapsto a] \\ \sigma' &= \sigma[a \mapsto \mathscr{A}(ae,\ \varsigma)] \\ a &= (x, |dom(\sigma)|) \end{aligned}$$

The continuation for address a'_κ is retrieved from the continuation/stack stored at continuation address a_κ. The stack/continuation is now encoded as a linked list where a_κ points to the current continuation and a'_κ points to its tail. It may be useful here to recall that $(=)$ above refers to propositional equality and is not an assignment or definition; the first condition of this rule unifies the variable a'_κ with the continuation address stored alongside the top-most frame at location a_κ in σ_κ.

In the above example, if the body e_y of `(λ (y) e_y)` is just the variable reference y, then execution will immediately return the value clo_1 after looking it up in the current environment and store (at address $(y, 341)$) using $\mathscr{A}(y,\ \varsigma)$. The current continuation address, $(y, 341)$ given that e_y is y, maps to the continuation $((z, e_{body}, \rho), a_\kappa)$ shown above in the current σ_κ. The address newly allocated for the return value will then be $(z, 342)$ and the saved ρ (without the binding for y) will be extended with $z \mapsto (z, 342)$ and the current value store (with a now-unreachable binding for $(y, 341)$) will be extended with $(z, 342) \mapsto clo_1$.

Step 2: Finitize abstract machine domains. The next step is to approximate each unbounded domain of machine components and replace it with a finite domain of *abstract* machine components. We start by replacing the set of concrete addresses with a finite set of *abstract* addresses, $\hat{a} \in \widehat{Addr}$, relating these domains to one another using a Galois connection $((\mathscr{P}(Addr), \subseteq) \underset{\alpha_{Addr}}{\overset{\gamma_{Addr}}{\leftrightarrows}} (\mathscr{P}(\widehat{Addr}), \subseteq))$. (Typographically we use hats to dif-

ferentiate abstract domains from their respective concrete counterparts that do not wear hats.) A monotone *Galois connection* is a pair of order-preserving adjoined morphisms (α and γ, encoding abstraction and concretization) that describe a relationship between two ordered sets (in this case, $\mathscr{P}(Addr)$ and $\mathscr{P}(\widehat{Addr})$, ordered by inclusion) such that $\alpha \circ \gamma \sqsubseteq \lambda x.x$ (i.e., $\alpha \circ \gamma$ is reductive)[1] and $\gamma \circ \alpha \sqsupseteq \lambda x.x$ (i.e., $\gamma \circ \alpha$ is expansive). These two properties enforce the precision and correctness (i.e., self-consistency) of this notion of abstraction, respectively. The Galois connection formalizes corresponding notions of abstraction (α) and concretization (γ) so that an abstract semantics over abstract domains can be calculated (or justified post-hoc) as a sound simulation of a concrete semantics over concrete domains (Cousot & Cousot, 1979). Intuitively, α_{Addr} takes a set of concrete addresses and maps it to a most-precise set of abstract addresses that may be used to approximate any of the input addresses; γ_{Addr} maps a set of abstract addresses to a most-precise set of concrete addresses that must include any addresses that are encoded by input abstract addresses. A forthcoming α_S will be defined to abstract whole concrete executions.

A Galois connection may be fully defined using an extraction function that maps each concrete entity to its corresponding abstraction. In the case of addresses, this would be a function $\eta : Addr \to \widehat{Addr}$ that yields the abstract address to be used for a given concrete address. For example, $\eta((x,n)) = x$ represents each address by the name of the variable it was allocated for. If a concrete execution were to allocate an address $(y, 341)$, as in our recent example, a flow analysis using the above notion of abstraction would represent this concrete address using an abstract address y. Deriving a Galois connection over powerset domains $\mathscr{P}(X) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathscr{P}(\hat{X})$ from an extraction function $\eta : X \to \hat{X}$ is then straightforward:

$$\alpha(xs) = \{\eta(x) \mid x \in xs\} \qquad \gamma(\widehat{xs}) = \{x \mid \hat{x} \in \widehat{xs} \wedge \eta(x) = \hat{x}\}$$

We start with an abstract-address set for both values and continuations ($\widehat{Addr}$ and $\widehat{KAddr}$) along with Galois connections relating them to their concrete counterparts. We make the choice of using the syntactic domains Var and Exp for abstract value and continuation addresses, respectively, in order to construct an analysis that is *monovariant*—meaning that there is a single representation (variant) for flows to each syntactic entity (value flows to variables, continuation and control flows to functions). This choice tunes the present analysis to instantiate a 0-CFA (see §2.1). Abstract values propagate to variables that may become bound to concrete values they approximate; abstract continuations flow to function bodies that may return values to continuations they approximate. As our initial analysis is monovariant, each syntactic variable is modeled by a single conservative approximation of its possible values; likewise, each syntactic function is associated with a single overapproximation of its possible continuations. Thus our extraction map for value-store addresses maps all addresses for a variable x to the same abstract address and our abstraction map for continuation-store addresses maps all concrete continuation addresses for the same

[1] Here, $\lambda x.x$ denotes the mathematical identity function and not code in our target language. The partial order ($\sqsubseteq$) corresponds to the notion of precision used for the concrete and abstract domains. For sets, the order we use is set-inclusion ($\subseteq$); for products and maps, the order distributes pointwise in the natural way (i.e., $g \sqsubseteq f \iff \forall x.g(x) \sqsubseteq f(x)$).

function body to the same abstract continuation address (that function body).

$$\eta_{Addr}((x,\ n)) \triangleq x \qquad \eta_{KAddr}((e,\ n)) \triangleq e$$

Then, at each step, moving from $\widehat{Addr}$ up to $\hat{S}$, we systematically compose existing Galois connections with syntactic domains or with other Galois connections to derive new Galois connections. This process is detailed in Might's paper on obtaining "abstract interpreters for free" (Might, 2010). In most cases, these inferred Galois connections are straightforward.

Lifting a Galois connection for addresses to one for environments yields a set of abstract environments $\hat{\rho} \in \mathsf{Var} \rightharpoonup \widehat{Addr}$. Abstract environments remain functions because only their co-domain is abstracted. Abstract stores, however, become *relational* because the domain of stores is abstracted, so two points in the domain may become conflated. In the end, we obtain the following abstract domains:

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &\triangleq \mathsf{Exp} \times \widehat{Env} & \hat{s} \in \hat{S} &\triangleq \mathscr{P}(\hat{\Sigma}) \\
&\quad \times \widehat{Store} \times \widehat{KStore} \times \widehat{KAddr} & \hat{\sigma}_\kappa \in \widehat{KStore} &\triangleq \widehat{KAddr} \to \mathscr{P}(\widehat{Kont}) \\
\hat{\rho} \in \widehat{Env} &\triangleq \mathsf{Var} \rightharpoonup \widehat{Addr} & \hat{\kappa} \in \widehat{Kont} &\triangleq \widehat{Frame} \times \widehat{KAddr} \\
\hat{\sigma} \in \widehat{Store} &\triangleq \widehat{Addr} \to \mathscr{P}(\widehat{Value}) & \hat{\phi} \in \widehat{Frame} &\triangleq \mathsf{Var} \times \mathsf{Exp} \times \widehat{Env} \\
\hat{a} \in \widehat{Addr} &\triangleq \mathsf{Var} & \hat{v} \in \widehat{Value} &\triangleq \widehat{Clo} \\
\hat{a}_\kappa \in \widehat{KAddr} &\triangleq \mathsf{Exp} & \widehat{clo} \in \widehat{Clo} &\triangleq \mathsf{Lam} \times \widehat{Env}
\end{aligned}$$

This compaction of the address set to ensure the finiteness of our analysis leads to relationality (i.e., nondeterminism) in the store—each value address or continuation address is associated with a set of zero or more possible values or continuations, respectively. Note that any given input program will use a finite set of syntactic variables, Var, even if the set of all possible variables is unbounded. Because abstract addresses may overapproximate multiple concrete addresses, each abstract address denotes a *flow set* of possible abstract closures (closures with an environment referencing further approximate abstract addresses). Nondeterminism in the store, in turn, leads to nondeterminism in the abstract transition *relation*. It will be possible for multiple abstract states to immediately succeed a single abstract antecedent.

Now that these abstract machine domains can express different granularities of precision in approximating concrete machine components, we may associate an ordering for precision ($\sqsubseteq$) with each abstract domain. For powersets the lifted order is simply inclusion.

$$\hat{s} \sqsubseteq \hat{s}' \iff \hat{s} \subseteq \hat{s}' \iff \left(\forall \hat{\varsigma}.\, \left(\hat{\varsigma} \in \hat{s} \implies \hat{\varsigma} \in \hat{s}'\right)\right)$$

These are each defined by lifting orders for sub-components in the natural way. For products, the lifted order is conjunctive.

$$\begin{aligned}
(e, \hat{\rho}, \hat{\sigma}, \hat{\sigma}_\kappa, \hat{a}_\kappa) \sqsubseteq (e', \hat{\rho}', \hat{\sigma}', \hat{\sigma}'_\kappa, \hat{a}'_\kappa) &\iff e \sqsubseteq e' \wedge \hat{\rho} \sqsubseteq \hat{\rho}' \wedge \hat{\sigma} \sqsubseteq \hat{\sigma}' \wedge \hat{\sigma}_\kappa \sqsubseteq \hat{\sigma}'_\kappa \wedge \hat{a}_\kappa \sqsubseteq \hat{a}'_\kappa \\
(x, e, \hat{\rho}) \sqsubseteq (x', e', \hat{\rho}') &\iff x \sqsubseteq x' \wedge e \sqsubseteq e' \wedge \hat{\rho} \sqsubseteq \hat{\rho}' \\
(lam, \hat{\rho}) \sqsubseteq (lam', \hat{\rho}') &\iff lam \sqsubseteq lam' \wedge \hat{\rho} \sqsubseteq \hat{\rho}'
\end{aligned}$$

For maps, the order distributes point-wise (respecting any order on the domain):

$$\hat{\sigma} \sqsubseteq \hat{\sigma}' \iff \hat{a} \sqsubseteq \hat{a}' \wedge \hat{a} \in dom(\hat{\sigma}) \wedge \hat{a}' \in dom(\hat{\sigma}') \implies \hat{\sigma}(\hat{a}) \sqsubseteq \hat{\sigma}'(\hat{a}')$$
$$\hat{\rho} \sqsubseteq \hat{\rho}' \iff x \in dom(\hat{\sigma}) \wedge x \in dom(\hat{\sigma}') \implies \hat{\sigma}(x) \sqsubseteq \hat{\sigma}'(x)$$

The order over syntactic domains is simply equality. Thus, the order on both concrete and abstract addresses is degenerate as well:

$$x \sqsubseteq x' \iff x = x' \qquad a \sqsubseteq a' \iff a = a'$$
$$e \sqsubseteq e' \iff e = e' \qquad \hat{a} \sqsubseteq \hat{a}' \iff \hat{a} = \hat{a}'$$

A total Galois connection produced by this process may be defined:

$$\alpha_S(s) \triangleq \{\eta_\Sigma(\varsigma) \mid \varsigma \in s\}$$
$$\eta_\Sigma((e,\rho,\sigma,\sigma_\kappa,a_\kappa)) \triangleq (e, \eta_{Env}(\rho), \eta_{Store}(\sigma), \eta_{KStore}(\sigma_\kappa), \eta_{KAddr}(a_\kappa))$$
$$\eta_{Env}(\rho) \triangleq \{(x, \eta_{Addr}(a)) \mid (x,a) \in \rho\}$$
$$\eta_{Store}(\sigma) \triangleq \bigsqcup_{(a,v)\in\sigma} [\eta_{Addr}(a) \mapsto \{\eta_{Value}(v)\}]$$
$$\eta_{KStore}(\sigma_\kappa) \triangleq \bigsqcup_{(a_\kappa,\kappa)\in\sigma_\kappa} [\eta_{KAddr}(a_\kappa) \mapsto \{\eta_{Kont}(\kappa)\}]$$
$$\eta_{Kont}((\phi,a_\kappa)) \triangleq (\eta_{Frame}(\phi), \eta_{KAddr}(a_\kappa))$$
$$\eta_{Frame}((x,e,\rho)) \triangleq (x, e, \eta_{Env}(\rho))$$
$$\eta_{Value}(clo) \triangleq \eta_{Clo}(clo) \qquad \eta_{Clo}((lam,\rho)) \triangleq (lam, \eta_{Env}(\rho))$$
$$\eta_{Addr}((x,n)) \triangleq x \qquad \eta_{KAddr}((e,n)) \triangleq e$$

This specification for α_S implicitly defines a unique adjoint γ_S such that $\alpha_S \circ \gamma_S$ is reductive and $\gamma_S \circ \alpha_S$ is expansive. Here we use the interpretation of maps as sets of tuples and μ to denote injection functions that map a single abstract entity back to a set of concrete entities.

$$\gamma_S(\hat{s}) \triangleq \bigsqcup\{\mu_\Sigma(\hat{\varsigma}) \mid \hat{\varsigma} \in \hat{s}\}$$
$$\mu_\Sigma((e,\hat{\rho},\hat{\sigma},\hat{\sigma}_\kappa,\hat{a}_\kappa)) \triangleq \{(e,\rho,\sigma,\sigma_\kappa,a_\kappa) \mid \rho \in \mu_{Env}(\hat{\rho}) \wedge \sigma \in \mu_{Store}(\hat{\sigma}) \wedge \sigma_\kappa \in \mu_{KStore}(\hat{\sigma}_\kappa) \wedge a_\kappa \in \mu_{KAddr}(\hat{a}_\kappa)\}$$
$$\mu_{Env}(\hat{\rho}) \triangleq \{(x,a) \mid a \in \mu_{Addr}(\hat{a}) \wedge (x,\hat{a}) \in \hat{\rho}\}$$
$$\mu_{Store}(\hat{\sigma}) \triangleq \Pi\{\{(a,v) \mid a \in \mu_{Addr}(\hat{a}) \wedge v \in \mu_{Value}(\hat{v})\} \mid (\hat{a},\hat{v}) \in \hat{\sigma}\}$$
$$\mu_{KStore}(\hat{\sigma}_\kappa) \triangleq \Pi\{\{(a_\kappa,\kappa) \mid a_\kappa \in \mu_{KAddr}(\hat{a}_\kappa) \wedge \kappa \in \mu_{Kont}(\hat{\kappa})\} \mid (\hat{a}_\kappa,\hat{\kappa}) \in \hat{\sigma}_\kappa\}$$
$$\mu_{Kont}((\hat{\phi},\hat{a}_\kappa)) \triangleq \{(\phi,a_\kappa) \mid \phi \in \mu_{Frame}(\hat{\phi}) \wedge a_\kappa \in \mu_{KAddr}(\hat{a}_\kappa)\}$$
$$\mu_{Frame}((x,e,\hat{\rho})) \triangleq \{(x,e,\rho) \mid \rho \in \mu_{Env}(\hat{\rho})\}$$
$$\mu_{Value}(\hat{clo}) \triangleq \mu_{Clo}(\hat{clo}) \qquad \mu_{Clo}((lam,\hat{\rho})) \triangleq \{(lam,\rho) \mid \rho \in \mu_{Env}(\hat{\rho})\}$$
$$\mu_{Addr}(x) \triangleq \{(x,n) \mid n \in \mathbb{N}\} \qquad \mu_{KAddr}(e) \triangleq \{(e,n) \mid n \in \mathbb{N}\}$$

The Π operator performs a Cartesian-product-style transformation of a set of sets (into a new set of sets) where the inner output sets contain each combination of points from across

the inner input set, like so:

$$\Pi\{ps_0,\ldots,ps_m\} \triangleq \{\{p_0,\ldots,p_m\} \mid p_0 \in ps_0 \wedge \ldots \wedge p_m \in ps_m\}$$

This is used to yield a set of concrete stores (or continuation stores) that covers all combinations of concrete points (i.e., $[a \mapsto v,\ldots]$) that are approximated by a given abstract store. Note well that the abstraction map and concretization maps for components (i.e., extractors η and injectors μ) may not properly form a Galois connection (as do α_S and γ_S) unless both are lifted to operate over partial orders such as sets.

Step 3: Justify or calculate an abstract semantics. So far, this systematic process for abstracting an abstract machine (starting from address sets and moving up to domains that depend on them) yields a full set of abstract domains but not an abstract semantics (i.e., not a transition relation over these domains). To produce a relation $(\hat{\rightsquigarrow}_\Sigma) \subseteq \hat{\Sigma} \times \hat{\Sigma}$ between abstract states we have two options: justify it as sound post-hoc or directly calculate it. Note that the hatted-Sigma-subscript indicates that this relation associates abstract predecessor states with abstract successor states. By contrast, $(\rightarrow_s)$ relates whole concrete program traces (i.e., sets of states in S) to their incremental extension (also in S).

An analysis is *sound* if the information it provides for a target program is correct and represents a definite bound on possible concrete executions. A sound analysis can definitively exclude some program behaviors as provably impossible under any actual execution or can definitively include some behaviors as provably occuring under any actual execution. The kind of control-flow information the analysis of this section (§4.2) obtains is a sound conservative overapproximation of program behavior; it places a strict upper bound on the propagation of closures through a program.

Our Galois connection for states represents a notion of simulation that any acceptable $(\hat{\rightsquigarrow}_\Sigma)$ must respect to soundly approximate $(\rightarrow_\Sigma)$. It also implies many different analyses that are sound with respect to it (precise or not). For example, an abstract transition relation that yields the entire abstract state space at every step is trivial and fully imprecise but is still sound with respect to $(\rightarrow_\Sigma)$ and α_Σ (i.e., as induced by the η_Σ shown). There are two traditional approaches to showing an analysis is guaranteed to be sound: (1) we can write an analysis that is convenient to implement and then justify that it is sound after the fact, or (2) we can compute the most precise analysis allowable by the Galois connection. The first option requires showing that, in any bisimulation of the abstract semantics and concrete semantics, a sound simulation is preserved at every iteration of the fixpoint computation:

Theorem 4.1 (*Soundness of* 0-*CFA*)
Bisimulation is preserved across every transition.

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

Diagrammatically this is:

$$\begin{array}{ccc} s & \xrightarrow{\rightarrow_S} & s' \\ {\scriptstyle \alpha_S}\downarrow \subseteq & & {\scriptstyle \alpha_S}\downarrow \subseteq \\ \hat{s} & \xrightarrow{\hat{\rightsquigarrow}_S} & \hat{s}' \end{array}$$

This manually shows that our chosen $(\hat{\rightsquigarrow}_s)$ yields a successor $\hat{s}'$ that overapproximates the concrete successor s' whenever their respective predecessors are in simulation. Eventually, as the lattice $(\hat{S}, \cup, \cap)$ is finite, iteration of $(\hat{\rightsquigarrow}_s)$ yields a fixed point in a finite number of steps and thereafter simulates all further concrete steps without itself changing.

The second option is to directly compute the most precise acceptable $(\hat{\rightsquigarrow}_s)$ that this commuting diagram allows; in particular, this is the abstract transition that accomplishes the same work as concretizing, transitioning, and abstracting again:

$$(\hat{\rightsquigarrow}_s) \triangleq \alpha_S \circ (\rightarrow_s) \circ \gamma_S$$

This is what is known as the *calculational approach*, invented and championed by Cousot and Cousot. Our technique for generalizing polyvariance as abstract allocation is agnostic with regard to post-hoc justification versus direct calculation of sound analyses. The analysis presented here is the optimal, calculated, abstract transition for the Galois connection given above.

A sound abstract transition for ANF can be defined:

$$\overbrace{(\texttt{((let ([y (}f\ ae\texttt{)]) }e\texttt{)}, \hat{\rho}, \hat{\sigma}, \hat{\sigma}_\kappa, \hat{a}_\kappa)}^{\hat{\varsigma}} \hat{\rightsquigarrow}_\Sigma (e', \hat{\rho}', \hat{\sigma}', \hat{\sigma}'_\kappa, \hat{a}'_\kappa), \text{where}$$

$$\begin{aligned}
(\texttt{((}\lambda\ \texttt{(}x\texttt{) }e'\texttt{)}, \hat{\rho}_\lambda) &\in \hat{\mathscr{A}}(f, \hat{\varsigma}) \\
\hat{\rho}' &= \hat{\rho}_\lambda[x \mapsto \hat{a}] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathscr{A}}(ae, \hat{\varsigma})] \\
\hat{\sigma}'_\kappa &= \hat{\sigma}_\kappa \sqcup [\hat{a}'_\kappa \mapsto ((y, e, \hat{\rho}), \hat{a}_\kappa)] \\
\hat{a} &= x \\
\hat{a}'_\kappa &= e'
\end{aligned}$$

This has three main changes from the concrete semantics. First, we reuse the same address for every binding to a variable x (i.e., its name) and the same continuation address for every call to a function (i.e., its body). Second, values and continuations become conflated at these addresses by using a weak update on the store ($\sqcup$) that yields the least upper bound of the existing store and each new binding. Third, we nondeterministically select closures from the abstract binding for f, and an abstract successor state results for each.

Join on abstract stores distributes point-wise:

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

Soundness normally requires that we never remove values from the store, once added, however there are sound techniques for doing so under specific conditions (Might & Shivers, 2006). Instead, we accumulate every closure bound to each $\hat{a}$ (i.e., abstract closures that simulate closures bound to addresses that $\hat{a}$ simulates) over the lifetime of the program. A flow set for an address $\hat{a}$ indicates a range of values that over approximate all possible concrete values that can flow to *any* concrete address approximated by $\hat{a}$. For example, if a concrete machine binds $(y, 341) \mapsto clo_1$ and $(y, 902) \mapsto clo_2$, its approximation might bind $y \mapsto \{\widehat{clo}_1, \widehat{clo}_2\}$. Precision is lost for $(y, 341)$ 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 (i.e., the environment in $\widehat{clo}_1$ is less precise than the environment in clo_1).

The abstract atomic evaluator produces flow sets instead of individual concrete values. In the case of closure creation, a singleton flow set is returned as follows.

$$\hat{\mathscr{A}} : \mathsf{AExp} \times \hat{\Sigma} \rightharpoonup \mathscr{P}(\widehat{Value})$$
$$\hat{\mathscr{A}}(x, (e, \hat{\rho}, \hat{\sigma}, \hat{\sigma}_\kappa, \hat{a}_\kappa)) \triangleq \hat{\sigma}(\hat{\rho}(x))$$
$$\hat{\mathscr{A}}(lam, (e, \hat{\rho}, \hat{\sigma}, \hat{\sigma}_\kappa, \hat{a}_\kappa)) \triangleq \{(lam, \hat{\rho})\}$$

In the same way that a successor results for each closure in call position, at a return point, a successor results for each possible continution:

$$\overbrace{(ae, \hat{\rho}, \hat{\sigma}, \hat{\sigma}_\kappa, \hat{a}_\kappa)}^{\hat{\varsigma}} \hat{\rightsquigarrow}_\Sigma (e, \hat{\rho}', \hat{\sigma}', \hat{\sigma}_\kappa, \hat{a}'_\kappa), \text{where}$$

$$\begin{aligned} ((x, e, \hat{\rho}_\kappa), \hat{a}'_\kappa) &\in \hat{\sigma}_\kappa(\hat{a}_\kappa) \\ \hat{\rho}' &= \hat{\rho}_\kappa[x \mapsto \hat{a}] \\ \hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{\mathscr{A}}(ae, \hat{\varsigma})] \\ \hat{a} &= x \end{aligned}$$

We again lift $(\hat{\rightsquigarrow}_\Sigma)$ to obtain a collecting semantics $(\hat{\rightsquigarrow}_s)$, for a target e_0, defined over sets of states:

$$\hat{s} \hat{\rightsquigarrow}_s \hat{s}', \text{where}$$

$$\hat{s}' = \{\hat{\varsigma}' \mid \hat{\varsigma} \in \hat{s} \wedge \hat{\varsigma} \hat{\rightsquigarrow}_\Sigma \hat{\varsigma}'\} \cup \{\hat{\mathscr{I}}(e_0)\}$$

This uses a starting-state injection function:

$$\hat{\mathscr{I}}(e) \triangleq (e, \varnothing, \bot, \bot, \hat{a}_{\text{halt}})$$

where the special address $\hat{a}_{\text{halt}}$ has no continuations and so halts the machine when reached.

Because $\widehat{Addr}$ and $\widehat{KAddr}$ (and thus $\hat{\Sigma}$ and $\hat{S}$) are now finite, we know the abstract evaluation of even nonterminating e_0 will terminate. That is there is some $n \in \mathbb{N}$ such that $(\hat{\rightsquigarrow}_s)^n(\bot)$ is a fixed point. Furthermore, this value contains an approximation of the target program e_0's full concrete program trace.

4.2.1 Extension to Larger Languages

Setting up a semantics for real language features such as conditionals, primitive operations, direct-style recursion, objects, mutation, etc, is straightforward but adds complication to the transition rules and soundness proof. Handling other forms is sometimes as straightforward as including an additional transition rule (or case in atomic evaluation, $\hat{\mathscr{A}}$), for each. Consider the inclusion of booleans, conditionals, vectors (or desugared structures, objects)

and n-ary lambdas via the following changes:

$$\begin{aligned} e \in \mathsf{Exp} &::= \texttt{(let ([}y\ \texttt{(}f\ ae\ \ldots\texttt{)])}\ e\texttt{)} \mid ae \mid \texttt{(if}\ ae\ e\ e\texttt{)} \\ ae \in \mathsf{AExp} &::= lam \mid x \mid vec \mid \texttt{\#t} \mid \texttt{\#f} \\ lam \in \mathsf{Lam} &::= \texttt{(}\lambda\ \texttt{(}x\ \ldots\texttt{)}\ e\texttt{)} \\ vec \in \mathsf{Vec} &::= \texttt{(vector}\ x\ \ldots\texttt{)} \mid \texttt{(vector-ref}\ x\ n\texttt{)} \end{aligned}$$

Vectors require their fields to be administratively bound so an address is immediately available—this simplifies vector semantics and requires the same kind of code expansion/desugaring involved in conversion to ANF. Likewise, conditionals require their guard expression to be previously let-bound. Vectors can be encoded as a tuple of addresses that each reference that field's flow set:

$$\begin{aligned} \hat{v} \in \widehat{Value} &\triangleq \widehat{Clo} + \widehat{Vec} + \widehat{Bool} \\ \widehat{vec} \in \widehat{Vec} &\triangleq \widehat{Addr}^* \\ \widehat{bool} \in \widehat{Bool} &\triangleq \{\texttt{true}, \texttt{false}\} \end{aligned}$$

Atomic evaluation can be extended with a case for each new atomic expression:

$$\begin{aligned} \hat{\mathscr{A}}(x,\ (\_, \hat{\rho}, \hat{\sigma}, \_, \_)) &\triangleq \hat{\sigma}(\hat{\rho}(x)) \\ \hat{\mathscr{A}}(lam,\ (\_, \hat{\rho}, \_, \_, \_)) &\triangleq \{(lam,\ \hat{\rho})\} \\ \hat{\mathscr{A}}(\texttt{(vector}\ x_0\ \ldots\ x_j\texttt{)},\ (\_, \hat{\rho}, \_, \_, \_)) &\triangleq \{(\hat{\rho}(x_0), \ldots, \hat{\rho}(x_j))\} \\ \hat{\mathscr{A}}(\texttt{(vector-ref}\ x\ n\texttt{)},\ \overbrace{(\_, \_, \hat{\sigma}, \_, \_)}^{\hat{\varsigma}}) &\triangleq \bigsqcup_{(\hat{a}_0, \ldots, \hat{a}_j) \in \hat{\mathscr{A}}(x, \hat{\varsigma})} \hat{\sigma}(\hat{a}_n) \\ \hat{\mathscr{A}}(\texttt{\#t},\ \_) &\triangleq \{\texttt{true}\} \\ \hat{\mathscr{A}}(\texttt{\#f},\ \_) &\triangleq \{\texttt{false}\} \end{aligned}$$

A `vector` expression yields a singleton set containing a tuple of addresses to encode the object's fields. A `vector-ref` expression looks-up all possible tuples with a matching n^{th} field and returns the least-upper-bound of the value (from the store) at all those fields.

The "then" and "else" transitions for conditionals are straightforward. Scheme and Racket semantics treat all non-false values as truthy, so the first rule applies whenever any non-false value may reach the guard expression:

$$\overbrace{(\texttt{(if}\ ae\ e_{then}\ e_{else}\texttt{)}, \hat{\rho}, \hat{\sigma}, \hat{\sigma}_\kappa, \hat{a}_\kappa)}^{\hat{\varsigma}} \rightsquigarrow_\Sigma^{\hat{}} (e_{then}, \hat{\rho}, \hat{\sigma}, \hat{\sigma}_\kappa, \hat{a}_\kappa), \text{where}$$
$$\bot \sqsubset \hat{\mathscr{A}}(ae, \hat{\varsigma}) \setminus \{\texttt{false}\}$$

$$\overbrace{(\texttt{(if}\ ae\ e_{then}\ e_{else}\texttt{)}, \hat{\rho}, \hat{\sigma}, \hat{\sigma}_\kappa, \hat{a}_\kappa)}^{\hat{\varsigma}} \rightsquigarrow_\Sigma^{\hat{}} (e_{else}, \hat{\rho}, \hat{\sigma}, \hat{\sigma}_\kappa, \hat{a}_\kappa), \text{where}$$
$$\{\texttt{false}\} \sqsubseteq \hat{\mathscr{A}}(ae, \hat{\varsigma})$$

An extension to n-ary lambdas ensures that the number of arguments matches the number of parameters and generates an address, and a binding to $\hat{\rho}$ and $\hat{\sigma}$, for each:

$$\overbrace{(\texttt{(let ([}y\ (f\ ae_0 \ldots ae_j)\texttt{])}\ e),\hat{\rho},\hat{\sigma},\hat{\sigma}_\kappa,\hat{a}_\kappa)}^{\hat{\varsigma}} \rightsquigarrow_\Sigma^\wedge (e',\hat{\rho}',\hat{\sigma}',\hat{\sigma}'_\kappa,\hat{a}'_\kappa), \text{where}$$

$$\begin{aligned}
(\texttt{(}\lambda\ \texttt{(}x_0 \ldots x_j\texttt{)}\ e'\texttt{)},\hat{\rho}_\lambda) &\in \hat{\mathscr{A}}(f,\hat{\varsigma}) \\
\hat{\rho}' &= \hat{\rho}_\lambda[x_0 \mapsto \hat{a}_i] \ldots [x_j \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup \bigsqcup_{i \in \{0 \ldots j\}} [\hat{a}_i \mapsto \hat{\mathscr{A}}(ae,\hat{\varsigma})] \\
\hat{\sigma}'_\kappa &= \hat{\sigma}_\kappa \sqcup [\hat{a}'_\kappa \mapsto ((y,e,\hat{\rho}),\hat{a}_\kappa)] \\
\hat{a}_i &= x_i \\
\hat{a}'_\kappa &= e'
\end{aligned}$$

4.3 Store Widening

Various forms of widening and further approximations can be layered on top of the basic unwidened 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, 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 the number of call sites, environments, and stores. A standard worklist algorithm does work that is at most proportional to the number of states it can discover (Might *et al.*, 2010). Analysis run-time is thus in:

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

The number of syntactic points in an input program is in $O(n)$. In our monovariant analysis, environments map variables to themselves (environments only contain points like $[y \mapsto y]$) and so are isomorphic to the sets of free variables at each syntactic point. In addition, each program expression is unique to its containing function body, so there is only one continuation address per syntactic point.

Regarding the store, the number of addresses produced by our monovariant analysis is in $O(n)$ as these are just syntactic variables. Likewise, the number of abstract closures is in $O(n)$ because there are this many syntactic lambdas and each lambda is unique to a monovariant environment for the same reason as all program expressions. The number of value stores may thus 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 addresses times the number of abstract closures gives $O(n^2)$ possible additions to the value store.

The crux of the present issue is that, in exploring a basic unwidened state space (where each state contains 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

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

Fig. 2: The value space of stores.

path we might add these closures in the reverse order. We might also extend another address $\hat{a}_2$ with $\widehat{clo}_1$ before, between, or after either of these cases, and so forth. Therefore a potential for exponential blow-up is unavoidable without further widening or coarser structural abstraction.

A bound on the number of continuation stores is obtained in the same manner. The number of abstract continuations is bounded by the size of the program for essentially the same reason as the number of abstract closures.

Global-store widening is an essential technique for combating this exponential blowup. 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 that we maintain as the least upper bound of all stores we encounter during analysis. The widened state spaces include both a single, global value store and a single, global continuation store. Instead of accumulating whole stores, and thereby all possible sequences of additions to such stores, the analysis strictly accumulates new values in the global store (independent of other values being accumulated at other addresses). This is similar to the way $(\rightsquigarrow_{\hat{s}})$ accumulates reachable states in a collection $\hat{s}$, independent of the distinct paths that lead to them:

$$
\begin{array}{ll}
\hat{\xi} \in \hat{\Xi} \triangleq \hat{R} \times \widehat{Store} \times \widehat{KStore} & \text{[state spaces]} \\
\hat{r} \in \hat{R} \triangleq \mathscr{P}(\hat{C}) & \text{[reachable configurations]} \\
\hat{c} \in \hat{C} \triangleq \mathsf{Exp} \times \widehat{Env} \times \widehat{KAddr} & \text{[configurations]}
\end{array}
$$

A widened transfer function $(\rightsquigarrow_{\hat{\Xi}} : \hat{\Xi} \to \hat{\Xi})$ can then be defined that, like $(\rightsquigarrow_{\hat{s}})$, is a monotonic, total function we may iterate to a fixed point. This can be defined in terms of $(\rightsquigarrow_{\hat{\Sigma}})$ and $(\rightsquigarrow_{\hat{s}})$ 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:

$$(\hat{r}, \hat{\sigma}, \hat{\sigma}_\kappa) \rightsquigarrow_{\hat{\Xi}} (\hat{r}', \hat{\sigma}'', \hat{\sigma}''_\kappa), \text{ where}$$

$$\hat{s} = \{(e, \hat{\rho}, \hat{\sigma}, \hat{\sigma}_\kappa, \hat{a}_\kappa) \mid (e, \hat{\rho}, \hat{a}_\kappa) \in \hat{r}\}$$
$$\hat{s}' = \{\hat{\varsigma}' \mid \hat{\varsigma} \in \hat{s} \wedge \hat{\varsigma} \rightsquigarrow_\Sigma^\wedge \hat{\varsigma}'\} \cup \{\hat{\mathscr{I}}(e_0)\}$$
$$\hat{r}' = \{(e', \hat{\rho}', \hat{a}'_\kappa) \mid (e', \hat{\rho}', \hat{\sigma}', \hat{\sigma}'_\kappa, \hat{a}'_\kappa) \in \hat{s}'\}$$
$$\hat{\sigma}'' = \bigsqcup_{(\_,\_,\hat{\sigma}',\_,\_)\in\hat{s}'} \hat{\sigma}' \qquad \hat{\sigma}''_\kappa = \bigsqcup_{(\_,\_,\_,\hat{\sigma}'_\kappa,\_)\in\hat{s}'} \hat{\sigma}'_\kappa$$

In this definition, underscores match anything (i.e., are wildcards). 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 and in time:

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

Using advanced bit-packing techniques (Midtgaard, 2012; Midtgaard & Horn, 2009), the best known algorithm for global-store-widened monovariant CFA is in $O(\frac{n^3}{\log n})$.

5 Allocation as a Tunable Parameter

In the previous section, we developed a global-store-widened analysis of ANF λ-calculus based on the concrete semantics of an abstract machine. It is *monovariant*, which means each variable or intermediate expression that we track during analysis receives exactly one flow set to overapproximate all its possible values. A closely related term is *context insensitive*, which means insensitive to any form of context and is a somewhat broader term that may, for example, include analyses less precise than this as well (e.g., the univariant analysis we present shortly). In Section 4.2, the crucial propositional statement (among those defining our abstract transition relation $(\rightsquigarrow_\Sigma^\wedge)$) that makes the analysis monovariant is this one:

$$\hat{a} = x$$

For each variable in the program, exactly one address is allocated to represent it.

The goal of this section is to produce a semantics that is parametric over the allocator $\widetilde{alloc}$ and that is sound regardless of how this function is defined. Although we formalize this semantics on its own, the primary change is that $\tilde{a}$ is chosen by an arbitrary allocator and thus:

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

Instead of defining the set of abstract addresses explicitly as done in Section 4.2, we define this set implicitly by the image and codomain of the allocation function. Typographically, we also switch to using tildes instead of hats for distinguishing components in our new parametric semantics.

This allows us to define monovariance as the following tuning of this function.

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

An equivalence relation on these addresses can 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 different parts of the same program are syntactically unequal. For simplicity in our formalism, we assume the latter.

The least polyvariant allocator produces only a single address $\top$ that overapproximates all concrete addresses:

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

We call this the *univariant* allocation scheme as it produces only a single address and conflates all program values. As imprecise as this is, it has some uses. For example, univariant allocation would make for an exceptionally cheap analysis powering dead-code elimination (yielding a worst-case quadratic analysis instead of a cubic one by collapsing $O(n)$ addresses to $O(1)$ addresses).

By contrast, the most polyvariant allocator produces new, previously-unused addresses each time. The resulting analysis is infinitely polyvariant and precisely models the concrete semantics.

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

It is also an example of an allocator that introspects on the current program state in order to choose an address. It uses the size of the current store to ensure it always produces a fresh address. 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 (Jenkins *et al.*, 2014).

Having seen these examples and the two extremes in terms of allocation strategies (i.e., univariant and concrete), there are two important questions. First, is there any allocation strategy that leads to an unsound analysis? Second, is there any polyvariant strategy that is not implementable by an allocation strategy? We answer these in the remainder of this section.

5.1 A Posteriori Soundness

The usual process for proving the soundness of an abstract interpretation is *a priori* in the sense that it can be performed entirely before the analysis is executed. This is the kind of soundness theorem we described in Section 4.2. By contrast, Might & Manolios (2009) describes an *a posteriori* soundness process where the abstraction map is not fully constructed until after analysis. This approach factors each abstraction map (α_X for some component X) to separate the abstraction of addresses η_{Addr}, producing a family of parametric maps β_X such that $\beta_X(\eta_{Addr}) = \alpha_X$. A non-deterministic abstract interpretation is then constructed that attempts all possible allocation strategies to show that all lead to sound approximations. (This could also be an arbitrary allocation function without loss of generality.) After the analysis is performed, regardless of the allocation strategy used,

a consistent abstraction map can be constructed *a posteriori* that justifies each choice of abstract address whatever it was. It is then always possible to use this Galois connection for addresses in the parametric Galois connection defined by β to obtain a complete connection and proof of soundness.

For example, the parametric Galois connection for the analysis of the last section may be defined:

$$\begin{aligned}
\beta_S(\eta_{Addr})(s) &\triangleq \{\beta_\Sigma(\eta_{Addr})(\varsigma) \mid \varsigma \in s\} \\
\beta_\Sigma(\eta_{Addr})((e,\rho,\sigma,\sigma_\kappa,a_\kappa)) &\triangleq (e, \beta_{Env}(\eta_{Addr})(\rho), \beta_{Store}(\eta_{Addr})(\sigma), \\
&\qquad \beta_{KStore}(\eta_{Addr})(\sigma_\kappa), \beta_{KAddr}(\eta_{Addr})(a_\kappa)) \\
\beta_{Env}(\eta_{Addr})(\rho) &\triangleq \{(x, \eta_{Addr}(a)) \mid (x,a) \in \rho\} \\
\beta_{Store}(\eta_{Addr})(\sigma) &\triangleq \bigsqcup_{(a,v)\in\sigma} [\eta_{Addr}(a) \mapsto \{\beta_{Value}(\eta_{Addr})(v)\}] \\
\beta_{KStore}(\eta_{Addr})(\sigma_\kappa) &\triangleq \bigsqcup_{(a_\kappa,\kappa)\in\sigma_\kappa} [\eta_{KAddr}(a_\kappa) \mapsto \{\beta_{Kont}(\eta_{Addr})(\kappa)\}] \\
\beta_{Kont}(\eta_{Addr})((\phi,a_\kappa)) &\triangleq (\beta_{Frame}(\eta_{Addr})(\phi), \eta_{KAddr}(a_\kappa)) \\
\beta_{Frame}(\eta_{Addr})((x,e,\rho)) &\triangleq (x, e, \beta_{Env}(\eta_{Addr})(\rho)) \\
\beta_{Value}(\eta_{Addr})(clo) &\triangleq \beta_{Clo}(\eta_{Addr})(clo) \\
\beta_{Clo}(\eta_{Addr})((lam,\rho)) &\triangleq (lam, \beta_{Env}(\eta_{Addr})(\rho)) \\
\eta_{KAddr}((e,n)) &\triangleq e
\end{aligned}$$

Note that each abstract domain that depends transitively upon the notion of abstraction used for addresses takes η_{Addr} explicitly as a parameter.

What is special about the allocation of abstract addresses that makes even a random number generator a sound choice of allocator? Clearly we could not 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 can 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 can be reclaimed and the semantics are guaranteed to remain equivalent to allocating a fresh address.) Allocating a sequence of fresh, unique addresses that never duplicate previous concrete addresses is thus a central characteristic of what it means to be a concrete allocator. Because of this property, whatever the behavior of abstract address allocation, no inconsistency can arise within the η_{Addr} it induces. Because no concrete address is allocated more than once, no single concrete address can become abstracted to two different abstract addresses along a sound abstract program trace.

To illustrate this, consider Figure 3. It shows a program e_0 being injected into a starting state, ς_0, and evaluated step by step. A static analysis performed by iterating an abstract transition relation produces 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 in the bottom of Figure 3, with spurious transitions dangling from it. Dotted lines are used to illustrate "abstracts to" relationships for states and addresses (points in η_Σ and η_{Addr}). If a concrete machine and its abstract machine are simulated in lock-step, each abstract

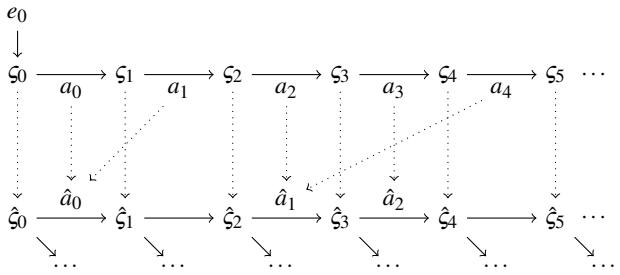

Fig. 3: All strategies for allocation induce a consistent Galois connection for addresses.

transition that allocates an address has two choices; either it can allocate an address $\hat{a}_i$ that it has allocated before, or it can 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 η_{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 a *dependent simulation* condition, which requires that each new map $\eta_{Addr}[a_i \mapsto \hat{a}_i]$ must be consistent with whatever partial abstraction map η_{Addr} was built previously. No further intuitions are given for this assumption, though it is the central property that allows the entire *a posteriori* soundness process to work—a property implied by the definition of concrete allocation. For the abstraction induced by the pairing of a concrete allocator and abstract allocator to be inconsistent, the same concrete address needs 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 $\eta_{Addr}[a_i \mapsto \hat{a}_i]$ must be consistent with η_{Addr} because the concrete address, a_i, cannot already be present in η_{Addr}. This makes abstract allocation a tunable parameter with the unique property that every possible tuning results in a sound analysis.

5.2 Introspection and Instrumentation

Now consider whether more precise forms of call sensitivity, such as 1-CFA or 2-CFA, can be implemented as a tuning of the allocation function. A 1-call sensitive allocator is straightforward. It can be defined by introspecting on the state being transitioned from and incorporating the most recent call site into the address being produced:

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

This makes addresses (and their flow sets) uniquely defined by both the variable `x` and the call site that preceded the binding. If we were to attempt an implementation of a more precise variant of call sensitivity however, such as 2-CFA, we run into a problem because the 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 $\widetilde{alloc}_{2\text{CFA}}$, we can instrument this core flow analysis with a new sixth component of machine states that specifically tracks the second most recent call site. In the original formulation of AAM, this was encoded in a function $\widehat{tick} : \hat{\Sigma} \to \widehat{Time}$, where the additional information was construed as an abstract *timestamp* encoding (in this case) a call history (Van Horn & Might, 2010). If we were to extend our analysis with such information, a 2-call sensitive allocator could be defined as:

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

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

Crucially, due to the *a posteriori* soundness theorem, we can 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 that 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 it causing unsoundness within the core flow analysis (i.e., within the store and set of reachable control-flow configurations). This means we can leave such instrumentation open, as another parameter to our semantics, and place no restrictions on its behavior. This can be thought of as an abstract timestamp, as in the original AAM framework, but it can also be an arbitrary extension to the core flow analysis, and this generality is crucial to our observation that this framework can encompass arbitrary flavors of polyvariant analysis. 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 can be expressed as a combination of some allocator and some instrumentation.

5.3 A Parametric Semantics

In this section, we present a parametric semantics that can be tuned by two allocation functions and an instrumentation relation (i.e., an arbitrary extension of the analysis). As before, we exclusively use tildes to keep this machine distinct from the machine of Section 4.

Our parametric semantics is encoded in the function:

$$\begin{aligned} CFA : \overbrace{\tilde{\Sigma}}^{\text{start state}} \times \overbrace{(\tilde{\Sigma} \times \mathsf{Call} \times \widetilde{Env} \times \widetilde{Store} \times \widetilde{KStore} \times \widetilde{KAddr} \to \mathscr{P}_{\geq 1}(\tilde{I}))}^{\text{instrumentation}} \\ \to \overbrace{(\mathsf{Var} \times \tilde{\Sigma} \to \widetilde{Addr})}^{\text{value allocator}} \to \overbrace{(\tilde{S} \to \tilde{S})}^{\text{analysis}} \end{aligned}$$

CFA is a curried function of several arguments: first, a starting state, $\tilde{\varsigma}_0$, which specifies the program to interpret and its initial $\tilde{\imath}_0$, paired with an instrumentation, $(\overset{\text{Inst}}{\rightsquigarrow})$, which may be used to extend the core analysis arbitrarily; then, a value allocator, $\widetilde{alloc}$. Figure 5 shows the signatures of these parameters to *CFA*. The allocator implicitly defines a set of addresses, $\widetilde{Addr}$, used by the analysis. The instrumentation relation defines a set of instrumentation

$$
\begin{array}{rll}
\tilde{s} \in \tilde{S} \triangleq \mathscr{P}(\tilde{\Sigma}) & & \text{[analysis results]} \\
\tilde{\xi} \in \tilde{\Xi} \triangleq \tilde{R} \times \widetilde{Store} & & \text{[widened results]} \\
\tilde{r} \in \tilde{R} \triangleq \mathscr{P}(\tilde{C}) & & \text{[reachable configs]} \\
\tilde{c} \in \tilde{C} \triangleq \mathsf{Call} \times \widetilde{Env} \times \widetilde{KAddr} \times \tilde{I} & & \text{[configurations]} \\
\tilde{\varsigma} \in \tilde{\Sigma} \triangleq \mathsf{Call} \times \widetilde{Env} \times \widetilde{Store} \times \widetilde{KStore} \times \widetilde{KAddr} \times \tilde{I} & & \text{[states]} \\
\tilde{\rho} \in \widetilde{Env} \triangleq \mathsf{Var} \rightharpoonup \widetilde{Addr} & & \text{[environments]} \\
\tilde{\sigma} \in \widetilde{Store} \triangleq \widetilde{Addr} \to \mathscr{P}(\widetilde{Value}) & & \text{[value stores]} \\
\tilde{\sigma} \in \widetilde{KStore} \triangleq \widetilde{KAddr} \to \widetilde{Kont} & & \text{[continuation stores]} \\
\tilde{\kappa} \in \widetilde{Kont} \triangleq \widetilde{Frame} \times \widetilde{KAddr} & & \text{[continuations]} \\
\tilde{\phi} \in \widetilde{Frame} \triangleq \mathsf{Var} \times \mathsf{Exp} \times \widetilde{Env} & & \text{[stack frames]} \\
\tilde{\imath} \in \tilde{I} \text{ is defined by the parameter } (\overset{\text{Inst}}{\rightsquigarrow}) & & \text{[inst. data]} \\
\tilde{a} \in \widetilde{Addr} \text{ is defined by the parameter } \widetilde{alloc} & & \text{[addresses]} \\
\tilde{a}_\kappa \in \widetilde{KAddr} \triangleq \mathsf{Exp} \times \widetilde{Env} & & \text{[cont. addresses]} \\
\tilde{v} \in \widetilde{Value} \triangleq \widetilde{Clo} & & \text{[flow sets]} \\
\widetilde{clo} \in \widetilde{Clo} \triangleq \mathsf{Lam} \times \widetilde{Env} & & \text{[closures]}
\end{array}
$$

Fig. 4: Abstract domains for our parametric semantics.

$$
\begin{array}{l}
\tilde{\varsigma}_0 \in \tilde{\Sigma} \\
\widetilde{alloc} \in \mathsf{Var} \times \tilde{\Sigma} \to \widetilde{Addr} \\
(\overset{\text{Inst}}{\rightsquigarrow}) \in \tilde{\Sigma} \times \mathsf{Exp} \times \widetilde{Env} \times \widetilde{Store} \times \widetilde{KStore} \times \widetilde{KAddr} \to \mathscr{P}(\tilde{I})
\end{array}
$$

Fig. 5: Parameters to these semantics.

data, $\tilde{I}$, to extend the core flow analysis and enable a greater variety of allocators. An instrumentation is a function that, fully 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 arbitrary analysis of its own, we use the following syntactic sugar:

$$\tilde{\varsigma} \overset{\text{Inst}}{\rightsquigarrow} (e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}'_\kappa, \tilde{\imath}') \iff \tilde{\imath}' \in (\overset{\text{Inst}}{\rightsquigarrow})(\tilde{\varsigma}, e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}'_\kappa)$$

The computed analysis is then a reduced product of the two analyses where the instrumentation analysis can only impact the core flow analysis (reachable $(e, \hat{\rho}, \hat{\sigma}, \hat{\sigma}_\kappa, \hat{a}_\kappa)$-tuples), via the allocation of addresses. The first five components of the successor state are inputs to $(\overset{\text{Inst}}{\rightsquigarrow})$, constraining the instrumentation data $\tilde{\imath}$ that is reached in the next step. The instrumentation may not yield an empty set, or it could lead to unsoundness. Figure 4 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 ($\tilde{\iota}$) and addresses are specified implicitly by the allocator chosen.

Provided a starting state and instrumentation, $\mathit{CFA}(\tilde{\varsigma}_0, \overset{\text{Inst}}{\rightsquigarrow})$ yields a parametric analysis that can be tuned further by an allocator. Provided all arguments, $\mathit{CFA}(\tilde{\varsigma}_0, \overset{\text{Inst}}{\rightsquigarrow})(\widetilde{\mathit{alloc}})$ yields a monotonic analysis function that can be iterated to a fixed point:

$$\mathit{CFA}(\tilde{\varsigma}_0, \overset{\text{Inst}}{\rightsquigarrow})(\widetilde{\mathit{alloc}}) \triangleq (\rightsquigarrow_s^{\sim}), \text{ where}$$

$$\tilde{s} \rightsquigarrow_s^{\sim} \{\tilde{\varsigma}' \mid \tilde{\varsigma} \in \tilde{s} \wedge \tilde{\varsigma} \rightsquigarrow_\Sigma^{\sim} \tilde{\varsigma}'\} \cup \{\tilde{\varsigma}_0\}$$

and the state transition relation $(\rightsquigarrow_\Sigma^{\sim})$ for applications and returns is as follows:

$$\overbrace{(\texttt{((let ([y (}f\ ae\texttt{)]) }e\texttt{)}, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa, \tilde{\iota})}^{\tilde{\varsigma}} \rightsquigarrow_\Sigma^{\sim} (e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}'_\kappa, \tilde{\iota}'), \text{ where}$$

$$\begin{aligned}
(\texttt{((}\lambda\ \texttt{(}x\texttt{)}\ e'\texttt{)}, \tilde{\rho}_\lambda) &\in \tilde{\mathscr{A}}(f,\ \tilde{\varsigma}) \\
\tilde{\rho}' &= \tilde{\rho}_\lambda[x \mapsto \tilde{a}] \\
\tilde{\sigma}' &= \tilde{\sigma} \sqcup [\tilde{a} \mapsto \tilde{\mathscr{A}}(ae_i,\ \tilde{\varsigma})] \\
\tilde{\sigma}'_\kappa &= \tilde{\sigma}_\kappa \sqcup [\tilde{a}'_\kappa \mapsto ((y, e, \tilde{\rho}), \tilde{a}_\kappa)] \\
\tilde{a} &= \widetilde{\mathit{alloc}}(x, \tilde{\varsigma}) \\
\tilde{a}'_\kappa &= \widetilde{\mathit{alloc}}_\kappa(\tilde{\varsigma}) \\
\tilde{\varsigma} &\overset{\text{Inst}}{\rightsquigarrow} (e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}'_\kappa, \tilde{\iota}')
\end{aligned}$$

$$\overbrace{(ae, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa, \tilde{\iota})}^{\tilde{\varsigma}} \rightsquigarrow_\Sigma^{\sim} (e, \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}_\kappa, \tilde{a}'_\kappa, \tilde{\iota}'), \text{where}$$

$$\begin{aligned}
((x, e, \tilde{\rho}_\kappa), \tilde{a}'_\kappa) &\in \tilde{\sigma}_\kappa(\tilde{a}_\kappa) \\
\tilde{\rho}' &= \tilde{\rho}_\kappa[x \mapsto \tilde{a}] \\
\tilde{\sigma}' &= \tilde{\sigma} \sqcup [\tilde{a} \mapsto \tilde{\mathscr{A}}(ae, \tilde{\varsigma})] \\
\tilde{a} &= \widetilde{\mathit{alloc}}(x, \tilde{\varsigma}) \\
\tilde{\varsigma} &\overset{\text{Inst}}{\rightsquigarrow} (e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}'_\kappa, \tilde{\iota}')
\end{aligned}$$

In this transition, 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 $\tilde{a}$ are constrained only by the allocator $\widetilde{\mathit{alloc}}$ provided. Third, the instrumentation function $(\overset{\text{Inst}}{\rightsquigarrow})$ constrains the instrumentation data $\tilde{\iota}'$ based on all other components of the transition. (At no point may the instrumentation relation yield an empty set of instrumentation data; this could lead to unsoundness.)

A continuation allocation is performed by the helper $\widetilde{\mathit{alloc}}_\kappa$ that is not treated as a parameter because it has a simple optimal definition (Gilray *et al.*, 2016b):

$$\widetilde{\mathit{alloc}}_\kappa((e', \tilde{\rho}', \_, \_, \_, \_)) \triangleq (e', \tilde{\rho}')$$

This could be made a parameter as easily as value allocation, but this strategy for continuation allocation yields an analysis that can be implemented with no asymptotic complexity overhead and perfect stack precision—that is, return flows that are as precise as a more expensive pushdown model. Roughly speaking, the inclusion of $\tilde{\rho}'$ ensures that continuations are allocated in a way that differentiates them as much as is necessary to adapt to the behavior of $\widetilde{alloc}$ and keep return flows distinct. Although it may appear that the same polyvariance could be used for both calls and returns to obtain precision equivalent to a pushdown model, this is not the case due to the structure of higher-order environments (i.e., a return value can depend on multiple distinct polyvariant allocations referenced in the binding environment). As it turned out, however, the additional, heavyweight machinery used in other approaches such as PDCFA (Earl *et al.*, 2012) and CFA2 (Vardoulakis & Shivers, 2010) could be captured as a tuning of continuation allocation that was simple to engineer and that provides a guarantee of *a posteriori* soundness at no asymptotic complexity cost. Discovering this optimal strategy for instantiating a continuation allocator is an example of productively applying our more general approach to polyvariance and tunable flow analyes—and the subject of its own paper. For more details on this specific technique, the reader is referred to Gilray *et al.* (2016b).

The atomic evaluator is defined in the same way as before:

$$\tilde{\mathscr{A}} : \mathsf{AExp} \times \tilde{\Sigma} \rightharpoonup \widetilde{Value}$$
$$\tilde{\mathscr{A}}(x,\ (\_, \tilde{\rho}, \tilde{\sigma}, \_, \_, \_)) = \tilde{\sigma}(\tilde{\rho}(x))$$
$$\tilde{\mathscr{A}}(lam,\ (\_, \tilde{\rho}, \_, \_, \_, \_)) = \{(lam,\ \tilde{\rho})\}$$

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. As in the previous section, a global-store-widened variant of this analysis can just as easily be defined.

5.4 A Parametric Abstraction

Now we can present the *a posteriori* soundness process in greater detail. The proof operates by factoring apart the concrete and abstract state-transition relations to separate the allocation of addresses from the rest of the semantics. In the concrete case, this recasts $(\rightarrow_\Sigma)$ as a curried function $f : \Sigma \rightarrow Addr \rightarrow \Sigma$ that takes both a predecessor state and a concrete address (to use for the current state transition) to produce a successor state. In the abstract case, this recasts $(\rightsquigarrow_{\tilde{\Sigma}})$ as a curried function $\tilde{f} : \tilde{\Sigma} \rightarrow \mathscr{P}(\widetilde{Addr} \rightarrow \tilde{\Sigma})$ that takes a predecessor state and yields a set of successors, each parameterized over the address allocated to produce them. The soundness process then requires a dependent simulation condition (Might & Manolios, 2009) that requires that two identical concrete addresses not be used in the same concrete execution. The soundness process then takes a completed analysis and directly computes the justifying abstraction map for addresses *a posteriori* by exploiting the following property:

Definition 5.1 (*Dependent simulation*)

A policy-factored abstract transition function $\tilde{f} : \tilde{\Sigma} \to \mathscr{P}(\widetilde{Addr} \to \tilde{\Sigma})$ is a **dependent simulation** of a policy-factored concrete transition function $f : \Sigma \to Addr \to \Sigma$, under a factored abstraction $\beta_\Sigma : (Addr \to \widetilde{Addr}) \to \Sigma \to \tilde{\Sigma}$, if and only if for all (partial) address extraction maps $\eta_{Addr} : Addr \to \widetilde{Addr}$ such that $\beta_\Sigma(\eta_{Addr})(\varsigma) \sqsubseteq \tilde{\varsigma}$, for any pair of concrete address a and abstract address $\tilde{a}$, there exists a parametric successor state $\tilde{h} \in \tilde{f}(\tilde{\varsigma})$ where $\beta_\Sigma(\eta_{Addr}[a \mapsto \tilde{a}])(f(\varsigma)(a)) \sqsubseteq \tilde{h}(\tilde{a})$.

This property states that the analysis must simulate the concrete semantics at non-address components, assuming a partial simulation for addresses used already, and that extending this map arbitrarily preserves simulation across a transition. This property is exactly what is needed to incrementally build up a justifying abstraction map for addresses post-hoc, regardless of what abstract addresses were used. What may be less clear at first is why this property should hold for allocation and not arbitrary analysis components. The reason is *no concrete address can be allocated more than once*. If a concrete address could be used twice in the same analysis, then it could be already present in the partial extraction map for addresses, η_{Addr}, and the final inequality in the definition of dependent simulation could not be derived from its premeses.

With this general requirement and notation intoduced, we may review the *a posteriori* soundness theorem in the context of polyvariant AAM.

Theorem 5.1 (*A posteriori soundness*)

If:

– $(\vec{\varsigma}, \vec{\mathbf{a}})$ is a concrete execution where $\vec{\varsigma}$ is a sequence of states $\langle \varsigma_0, \varsigma_1, \varsigma_2, \ldots \rangle$ and $\vec{\mathbf{a}} = \langle a_0, a_1, a_2, \ldots \rangle$ is the sequence of concrete addresses allocated at each transition,

– $\tilde{f}$ is a dependent simulation of f under the factored abstraction map β_Σ,

– $(\tilde{s}, \tilde{\rightsquigarrow_\Sigma})$ is the closed abstract transition graph yielded by $\tilde{f}$, with $\tilde{\varsigma}_0 \in \tilde{s}$, and

– for all η_{Addr}, $\beta_\Sigma(\eta_{Addr})(\varsigma_0) \sqsubseteq \tilde{\varsigma}_0$,

then there exists an η_{Addr} that makes $(\tilde{s}, \tilde{\rightsquigarrow_\Sigma})$ a sound simulation of $(\vec{\varsigma}, \vec{\mathbf{a}})$ under $\beta_\Sigma(\eta_{Addr})$.

Proof

We proceed by performing an *a posteriori* construction of η_{Addr}. This is done by building up a simulating sequence of abstract states $\vec{\tilde{\varsigma}} = \langle \tilde{\varsigma}_0, \tilde{\varsigma}_1, \tilde{\varsigma}_2, \ldots \rangle$, their respective abstract addresses $\vec{\tilde{\mathbf{a}}} = \langle \tilde{a}_0, \tilde{a}_1, \tilde{a}_2, \ldots \rangle$, and a sequence of partial extraction maps for addresses $\vec{\eta}_{Addr} = \langle \eta_0, \eta_1, \eta_2, \ldots \rangle$. We then show by induction that $\beta_\Sigma(\eta_i)(\varsigma_i) \sqsubseteq \tilde{\varsigma}_i$.

Let N be the length of $\vec{\varsigma}$ and the initial map η_0 be $\bot$. At each step a next abstract address $\tilde{a}_i$ and abstract state $\tilde{\varsigma}_{i+1}$ are chosen simultaneously from the set of candidate transitions $\{(\tilde{a}_i, \tilde{\varsigma}_{i+1}) \mid \tilde{\varsigma}_{i+1} = \tilde{h}(\tilde{a}_i), \tilde{h} \in \tilde{f}(\tilde{\varsigma}_i), \beta_\Sigma(\eta_i)(\varsigma_i) \sqsubseteq \tilde{\varsigma}_{i+1}\}$, which is guaranteed to be non-empty by the dependent simulation condition. Each new point $[a_i \mapsto \tilde{a}_i]$ is accumulated into an updated intermediate abstraction map $\eta_i = \eta_{i-1}[a_i \mapsto \tilde{a}_i]$ which inductively builds up η_{Addr} in its limit:

$$\eta_{Addr} = \lim_{i \to N} \eta_i$$

Then $(\tilde{s}, \tilde{\rightsquigarrow_\Sigma})$ is a simulation of $(\vec{\varsigma}, \vec{\mathbf{a}})$ with respect to $\beta(\eta_{Addr})$. □

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.

$$(\texttt{((let ([}y\ (f\ ae)\texttt{]) }e\texttt{)}, \_, \_, \_, \_, \tilde{\imath}) \overset{\text{Inst}}{\rightsquigarrow}_{call(k)} (\_, \_, \_, \_, \_, take_k(\texttt{(}f\ ae\texttt{)} : \tilde{\imath}))$$

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.

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

The parametric semantics of Section 5.3 can be tuned to recapitulate the k-call sensitive style of polyvariance for a program e_0 using the parameterization:

$$CFA((e_0, \varnothing, \bot, \bot, \tilde{a}_{\text{halt}}, \varepsilon), (\overset{\text{Inst}}{\rightsquigarrow}_{call(k)}), \widetilde{alloc}_{\text{call}})$$

6.1.1 Ambiguity in k-CFA

The original formulation of k-CFA by Shivers (1991) 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. Our call sensitivity, as just formalized, only remembers call sites but not return points and is a somewhat different form of polyvariance from what Shivers originally implemented.

To formalize a tuning of k-CFA that does track both call and return points, we are only required to change the behavior of our instrumentation to include both cases:

$$(\texttt{((let ([}y\ (f\ ae)\texttt{]) }e\texttt{)}, \_, \_, \_, \_, \tilde{\imath}) \overset{\text{Inst}}{\rightsquigarrow}_{\text{call+ret}(k)} (\_, \_, \_, \_, \_, take_k(\texttt{(}f\ ae\texttt{)} : \tilde{\imath}))$$

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

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

$$CFA((e_0, \varnothing, \bot, \bot, \tilde{a}_{\text{halt}}, \varepsilon), (\overset{\text{Inst}}{\rightsquigarrow}_{\text{call+ret}(2)}), \widetilde{alloc}_{\text{call}})$$

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 snippets of code differentiates 2-call+return sensitivity and 2-call-only sensitivity.

```
(let ([id (λ (x) x)])
  (let ([f (λ (y)
             (let ([v (id y)]) v))])
    (let ([r0 (f #f)])
      (let ([r1 (f #t)])
        r1))))
```

The last 2 calls before binding `v` the first time are `(id y)` and `(f #f)`, and the second time they are `(id y)` and `(f #t)`. The last 2 calls or returns before binding `v` the first time are `(id y)` and `(f #f)`, and the second time they are `(id y)` and `(f #t)`. However, when returning the values `#t` and `#f` from `f`, the last two calls are `(id y)` and `(f #f)` and then `(id y)` and `(f #t)` respectively, but the last two calls or returns in these cases are `x` and `(id y)` and then `x` and `(id y)` respectively. This means 2-call-only sensitivity will keep both addresses returned to `r1` distinct, while 2-call+return sensitivity will not.

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 may instantiate any conceivable heuristic may be an important step in understanding which styles of polyvariance work best in what situations and thereby inform us how to better adapt polyvariance to suit a particular target of analysis.

We could also define a predicate $I : \mathsf{Exp} \to \mathit{Boolean}$ that decides whether to include a particular expression (be it call or return) in the running context history:

$$(e, \_, \_, \_, \_, \tilde{\iota}) \overset{\text{Inst}}{\rightsquigarrow}_{\text{inc}(I,k)} \begin{cases} (\_, \_, \_, \_, \_, \mathit{take}_k(e : \tilde{\iota})) & I(e) \\ (\_, \_, \_, \_, \_, \tilde{\iota}) & \text{otherwise} \end{cases}$$

Then call sensitivity, return sensitivity, call+return sensitivity, and many other strategies all become expressible as tunings of the predicate I.

6.1.2 Variable Call Sensitivity

Wright & Jagannathan (1998) presents just such an adaptive heuristic for varying call sensitivity according to the syntax of a program. Their *polymorphic splitting* is a simple form of adaptive call sensitivity inspired by `let`-polymorphism where the degree of polyvariance can vary between functions. The number of `let`-form binding expressions (right-hand sides) a lambda was defined within forms a simple heuristic for its call sensitivity when invoked.

We generalize this to an arbitrary strategy for varying the depth of call sensitivity with a per-function k; to do so, 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).

$$(e, \_, \_, \_, \_, \tilde{\iota}) \overset{\text{Inst}}{\rightsquigarrow}_{\text{variable}(L)} (e', \_, \_, \_, \_, \mathit{take}_{(L(e'))}(e : \tilde{\iota}))$$

This variable call history is then used for allocating addresses.

$$\mathit{CFA}((e_0, \varnothing, \bot, \bot, \tilde{a}_{\text{halt}}, \varepsilon), (\overset{\text{Inst}}{\rightsquigarrow}_{\text{variable}(L)}), \widetilde{\mathit{alloc}}_{\text{call}})$$

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 to decide which call sites versus return points to extend.

6.2 Object Sensitivity

Smaragdakis *et al.* (2011) distinguishes multiple variants of *object sensitivity*, first described by Milanova *et al.* (2005). This style of context sensitivity is entirely different from call sensitivity and uses a history of the allocation points for objects to guide polyvariance.

For this section and the next, we temporarily extend our language with n-ary lambdas (in the straightforward manner) and a `vector` form as we did previously in section 4.2.1. We assume a desugaring such that the first argument to any function will be its receiving object.

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

$$\tilde{v} \in \widetilde{\mathit{Value}} \triangleq \widetilde{\mathit{Clo}} + \widetilde{\mathit{Vec}}$$
$$\widetilde{\mathit{vec}} \in \widetilde{\mathit{Vec}} \triangleq \widetilde{\mathit{Addr}}^*$$

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

$$\tilde{\mathscr{A}}(\texttt{(vector}\ x_0\ \ldots\ x_j\texttt{)},\ (\_, \tilde{\rho}, \_, \_, \_)) \triangleq \{(\tilde{\rho}(x_0), \ldots, \tilde{\rho}(x_j))\}$$
$$\tilde{\mathscr{A}}(\texttt{(vector-ref}\ x\ n\texttt{)},\ \overbrace{(\_, \_, \tilde{\sigma}, \_, \_)}^{\tilde{\varsigma}}) \triangleq \bigsqcup_{(\tilde{a}_0, \ldots, \tilde{a}_j) \in \tilde{\mathscr{A}}(x, \tilde{\varsigma})} \tilde{\sigma}(\tilde{a}_n)$$

As the fields of our objects are effectively each vector's indices, and because these are strictly kept distinct instead of being merged, we may call this representation for vectors *field sensitive* (Liang & Might, 2012). A flow set of objects could be $\{(\tilde{a}_1, \tilde{a}_2, \tilde{a}_3), \ldots\}$ but not $\{\tilde{a}_1, \tilde{a}_2, \tilde{a}_3, \ldots\}$, as they preserve the relationship between keys and values. Allowing these tuples of addresses within flow sets is not a new source of unboundedness in the machine because the longest possible tuple 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—up to a limited

depth of k allocation points. We instrument our analysis with an object-sensitivities store $\tilde{\sigma}_O$ and a current allocation history ($\tilde{o}$):

$$\tilde{\iota} \in \tilde{I} \triangleq \widetilde{OStore} \times \tilde{O}$$
$$\tilde{o} \in \tilde{O} \triangleq \mathsf{Exp}^*$$
$$\tilde{\sigma}_O \in \widetilde{OStore} \triangleq \widetilde{Addr} \times \widetilde{Vec} \to \mathscr{P}(\tilde{O})$$

Each state is extended with 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, and a current allocation history, $\tilde{o}$, a sequence of allocation points for the current receiver object (and its allocating object, etc). 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 expression) for each *ae* that constructs an object. Existing objects bound to a variable (some y) 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 should likely be used for object-sensitivities stores to parallel the global value store.

$$\overbrace{(\texttt{let ([}z\ (f\ ae_0 \ldots ae_j)\texttt{])}\ e), \tilde{\rho}, \_, \_, \_, (\tilde{\sigma}_O, \tilde{o})}^{\tilde{\varsigma}} \overset{\text{Inst}}{\rightsquigarrow}_{\text{vec}(k)} (e', \tilde{\rho}', \_, \_, \_, (\tilde{\sigma}'_O, \tilde{o}'))$$

$$\begin{aligned} \text{where}\quad & ((\lambda\ (x_0 \ldots x_j)\ e'), \_) \in \tilde{\mathscr{A}}(f, \tilde{\varsigma}) \\ & vec_0 \in \tilde{\mathscr{A}}(ae_0, \tilde{\varsigma}) \\ & \tilde{o}' \in \tilde{\sigma}'_O((\tilde{\rho}'(x_0), \widetilde{vec}_0)) \end{aligned}$$

$$\begin{aligned} \tilde{\sigma}'_O = \tilde{\sigma}_O \sqcup & \bigsqcup_{\substack{ae_i = \texttt{(vector ...)} \\ \widetilde{vec} \in \tilde{\mathscr{A}}(ae_i, \tilde{\varsigma})}} [(\tilde{\rho}'(x_i), \widetilde{vec}) \mapsto \{take_k((f\ \ldots\ ae_j) : \tilde{o})\}] \\ & \sqcup \bigsqcup_{\substack{ae_i = y_i \\ \widetilde{vec} \in \tilde{\mathscr{A}}(ae_i, \tilde{\varsigma})}} [(\tilde{\rho}'(x_i), \widetilde{vec}) \mapsto \tilde{\sigma}_O(\tilde{\rho}(y_i), \widetilde{vec})] \end{aligned}$$

Recall that this is syntactic sugar for defining a function that yields a set of instrumentations, in this case each $(\tilde{\sigma}'_O, \tilde{o}')$, from examining other components of the transition. The first line in the transition's constraints fixes the lambda being invoked and its list of parameters $x_0 \ldots x_j$. The next two lines non-deterministically select a new allocation history, $\tilde{o}'$, as any possible history for any possible receiver object mapped to in the new object-sensitivities store $\tilde{\sigma}'_O$. This store $\tilde{\sigma}'_o$ is produced by modifying the existing sensitivities store, $\tilde{\sigma}_O$, joining new histories into the parameter's address $\tilde{\rho}'(x_i)$, and each object at that address $\widetilde{vec}$; where the argument ae_i is a `vector`-form, a new history formed by extending the current object history with the current point, $take_k((f\ \ldots\ ae_j) : \tilde{o})$, and where the argument ae_i is a variable, pulling out any objects bound to the variable and propagating their histories.

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

histories associated with objects).

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

Smaragdakis *et al.* (2011) and Lhoták & Hendren (2006) find object sensitivity to be particularly efficient for object-oriented languages in their empirical investigations using the Java DaCapo and SpecJVM benchmarks. The intuition for this seems to be that object-sensitivity is particularly good at modeling correct flows for dynamic dispatch. Kastrinis & Smaragdakis (2013) present combinations of object and call sensitivity. Combinations of styles of polyvariance can also be accomplished in our framework by a tuning of instrumentation and allocation as we will observe in Section 6.5.

6.3 Argument Sensitivity

Agesen (1995) introduces a *Cartesian product algorithm* (CPA) as an enhancement to a type recovery algorithm. 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 monovariant inference 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.* (1992)—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 strictly overcome by any value of p. Besson (2009) 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 (λ (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 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 general framework. 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 $\mathscr{T}$ which further abstracts abstract values so they cannot contain addresses. For an approach especially similar to CPA itself, we might define $\mathscr{T}$ so it yields dynamic types. For a functional language, we may define $\mathscr{T}$ so that it strips environments out of closures and leaves just a set of syntactic lambdas. For example:

$$\mathscr{T}(\vec{v}) \triangleq \{lam \mid (lam, \tilde{\rho}) \in \vec{v}\}$$

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 (Gilray & Might, 2014).

With this, we may define an *argument sensitive* style of polyvariance, like CPA, as an abstract allocator. Each abstract address is produced specific to a variable and the exact tuple of dynamic types for the arguments passed at the same time:

$$\begin{aligned}\widetilde{alloc}_{\text{CPA}}(x,\ \overbrace{(\texttt{(let ([y (}f\ ae_0\ \ldots\ ae_j\texttt{)]) }e\texttt{)}, \_, \_, \_, \_, \_)}^{\tilde{\xi}}) \\ \triangleq (x, (\mathscr{T}(\tilde{\mathscr{A}}(ae_0, \tilde{\xi})),\ \ldots,\ \mathscr{T}(\tilde{\mathscr{A}}(ae_j, \tilde{\xi}))))\\ \widetilde{alloc}_{\text{CPA}}(x,\ (ae, \_, \_, \_, \_, \_)) \triangleq x\end{aligned}$$

We can also observe how natural 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:

$$\begin{aligned}\widetilde{alloc}_{\text{arg}_0}(x,\ \overbrace{(\texttt{(let ([y (}f\ ae_0\ \ldots\ ae_j\texttt{)]) }e\texttt{)}, \_, \_, \_, \_, \_)}^{\tilde{\xi}}) \triangleq (x, \mathscr{T}(\tilde{\mathscr{A}}(ae_0, \tilde{\xi})))\\ \widetilde{alloc}_{\text{arg}_0}(x,\ (ae, \_, \_, \_, \_, \_)) \triangleq x\end{aligned}$$

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 principle 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. Such observations would seem to support an effort to discover more adaptive nuanced 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. Consider 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). In fact, in this manner we can even recover all the precision lost through structurally store widening (as discussed in Section 4.3) 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.

$$\overbrace{(\texttt{(let ([}y\ (f\ ae)\texttt{]) }e\texttt{)}, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa, (\tilde{\iota}, \tilde{\rho}_\Sigma, \tilde{\sigma}_\Sigma))}^{\tilde{\xi}} \overset{\text{Inst}}{\rightsquigarrow}_{\text{ss}(\widetilde{alloc}, \overset{\text{Inst}}{\rightsquigarrow})} (e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}'_\kappa, (\tilde{\iota}', \tilde{\rho}'_\Sigma, \tilde{\sigma}'_\Sigma))$$

where
$$\begin{aligned}
&(\texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e'\texttt{)}, \tilde{\rho}_\lambda) \in \tilde{\mathscr{A}}(f,\ \tilde{\xi}) \\
&\tilde{\rho}'_\Sigma = \tilde{\rho}_\lambda[x \mapsto \tilde{a}] \\
&\tilde{\sigma}'_\Sigma = \tilde{\sigma}_\Sigma \sqcup [\tilde{a} \mapsto \tilde{\mathscr{A}}(ae,\ \tilde{\xi})] \\
&\tilde{a} = \widetilde{alloc}(x, \tilde{\xi}) \\
&(\texttt{(let ([}y\ (f\ ae)\texttt{]) }e\texttt{)}, \tilde{\rho}, \tilde{\sigma}, \tilde{\sigma}_\kappa, \tilde{a}_\kappa, \tilde{\iota}) \overset{\text{Inst}}{\rightsquigarrow} (e', \tilde{\rho}', \tilde{\sigma}', \tilde{\sigma}'_\kappa, \tilde{a}'_\kappa, \tilde{\iota}')
\end{aligned}$$

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

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

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 (Might *et al.*, 2010) or poly-k-CFA (Jagannathan & Weeks, 1995).

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

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 $(\overset{\text{Inst}}{\rightsquigarrow}_1)$:

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

$$\text{where} \quad (e,\tilde{\rho},\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa,\tilde{\imath}_0) \overset{\text{Inst}}{\rightsquigarrow}_0 (e',\tilde{\rho}',\tilde{\sigma}',\tilde{\sigma}'_\kappa,\tilde{a}'_\kappa,\tilde{\imath}'_0)$$
$$(e,\tilde{\rho},\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa,\tilde{\imath}_1) \overset{\text{Inst}}{\rightsquigarrow}_1 (e',\tilde{\rho}',\tilde{\sigma}',\tilde{\sigma}'_\kappa,\tilde{a}'_\kappa,\tilde{\imath}'_1)$$

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

$$\widetilde{alloc}_\times(x,\ (e,\tilde{\rho},\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa,(\tilde{\imath}_0,\tilde{\imath}_1))) \triangleq$$
$$(\widetilde{alloc}_0(x,\ (e,\tilde{\rho},\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa,\tilde{\imath}_0)),\ \widetilde{alloc}_1(x,\ (e,\tilde{\rho},\tilde{\sigma},\tilde{\sigma}_\kappa,\tilde{a}_\kappa,\tilde{\imath}_1)))$$

7 Conclusion

We have presented a general approach to encoding arbitrary strategies for polyvariance in systematically developed control-flow analyses. Different styles of polyvariance represent different ways of differentiating program flows that vary in their efficiency and effectiveness across targets of analysis. As such, having a general approach for encoding them that is simultaneously straightforward to engineer, fully general, and guaranteed to yield a sound analysis, provides a solid foundation for investigation and targeted adaptation.

A key requirement of our approach is to perform a store-passing transformation to the target language's semantics that exposes an address set. This infinite set of concrete addresses may then be finitized to yield a bounded set of abstract addresses (each overapproximating a set of concrete bindings). The manner in which these addresses are allocated and reused then becomes crucial—different designs for grouping and conflating concrete bindings encode different styles of polyvariance. Using *a posteriori soundness*, we show that the defining characteristic of concrete allocation, namely that every concrete address is allocated *fresh*, also means that no process of abstract allocation can result in an inconsistent abstraction map for addresses. This in turn means that any tuning of an abstract allocation function yields a sound analysis.

Then, in order to capture all conceivable styles of polyvariance, we further permit the flow analysis to be instrumented arbitrarily. While we are unable to make guarantees about arbitrary extensions of a core flow analysis, regardless of how this added information is used by the abstract allocator, we are guaranteed to have a sound core flow analysis. This allows us to be both fully general for our clearly defined notion of polyvariance and fully sound for control-flow and data-flow results yielded by the analysis.

Acknowledgments. The authors would like to thank the anonymous ICFP and JFP reviewers for their thorough and insightful feedback. It has been gratefully received.

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

Bibliography

Agesen, Ole. (1995). *The cartesian product algorithm*. Lecture Notes in Computer Science, vol. 952. Berlin, Heidelberg: Springer. Pages 2–26.

Amtoft, Torben, & Turbak, Franklyn. (2000). *Faithful translations between polyvariant flows and polymorphic types*. Lecture Notes in Computer Science, vol. 1782. Berlin, Heidelberg: Springer. Pages 26–40.

Appel, Andrew W. (2007). *Compiling with continuations*. Cambridge University Press.

Banerjee, Anindya. (1997). A modular, polyvariant and type-based closure analysis. *Pages 1–10 of: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*. ICFP '97. New York, NY, USA: ACM.

Besson, Frédéric. (2009). CPA beats ∞-CFA. *Pages 7:1–7:6 of: Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*. FTfJP '09. New York, NY, USA: ACM.

Bravenboer, Martin, & Smaragdakis, Yannis. (2009). Strictly declarative specification of sophisticated points-to analyses. *Pages 243–262 of: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '09. New York, NY, USA: ACM.

Cousot, Patrick. (1997). Types as abstract interpretations. *Pages 316–331 of: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. New York, NY, USA: ACM.

Cousot, Patrick, & Cousot, Radhia. (1976). Static determination of dynamic properties of programs. *Pages 106–130 of: Proceedings of the Second International Symposium on Programming*. Paris, France.

Cousot, Patrick, & Cousot, Radhia. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Pages 238–252 of: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. New York, NY, USA: ACM.

Cousot, Patrick, & Cousot, Radhia. (1979). Systematic design of program analysis frameworks. *Pages 269–282 of: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '79. New York, NY, USA: ACM.

Earl, Christopher, Sergey, Ilya, Might, Matthew, & Van Horn, David. 2012 (September). Introspective pushdown analysis of higher-order programs. *Pages 177–188 of: International Conference on Functional Programming*.

Flanagan, Cormac, Sabry, Amr, Duba, Bruce F., & Felleisen, Matthias. (1993). The essence of compiling with continuations. *Pages 237–247 of: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. PLDI '93. New York, NY, USA: ACM.

Gilray, Thomas, & Might, Matthew. 2013 (Nov.). A unified approach to polyvariance in abstract interpretations. *Proceedings of the Workshop on Scheme and Functional Programming*. Scheme '13.

Gilray, Thomas, & Might, Matthew. (2014). *A survey of polyvariance in abstract interpretations*. Theoretical Computer Science and General Issues, vol. 8322. Berlin, Heidelberg: Springer. Pages 134–148.

Gilray, Thomas, Adams, Michael D., & Might, Matthew. (2016a). Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. *Pages 407–420 of: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. New York, NY, USA: ACM.

Gilray, Thomas, Lyde, Steven, Adams, Michael D., Might, Matthew, & Van Horn, David. (2016b). Pushdown control-flow analysis for free. *Pages 691–704 of: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. New York, NY, USA: ACM.

Harrison, Williams Ludwell. (1989). The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and symbolic computation*, **2**(3–4), 179–396.

Holdermans, Stefan, & Hage, Jurriaan. (2010). Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. *Pages 63–74 of: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. New York, NY, USA: ACM.

Jagannathan, Suresh, & Weeks, Stephen. (1995). A unified treatment of flow analysis in higher-order languages. *Pages 393–407 of: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. New York, NY, USA: ACM.

Jagannathan, Suresh, Weeks, Stephen, & Wright, Andrew. (1997). *Type-directed flow analysis for typed intermediate languages*. Lecture Notes in Computer Science, vol. 1302. Berlin, Heidelberg: Springer. Pages 232–249.

Jenkins, Maria, Andersen, Leif, Gilray, Thomas, & Might, Matthew. 2014 (Nov.). Concrete and abstract interpretation: Better together. *Workshop on Scheme and Functional Programming*. Scheme '14.

Johnson, J. Ian, Labich, Nicholas, Might, Matthew, & Van Horn, David. (2013). Optimizing abstract abstract machines. *Pages 443–454 of: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. New York, NY, USA: ACM.

Jones, Neil D., & Muchnick, Steven S. (1982). A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *Pages 66–74 of: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. New York, NY, USA: ACM.

Kahn, G[illes]. (1987). *Natural semantics*. Lecture Notes in Computer Science, vol. 247. Berlin, Heidelberg: Springer. Pages 22–39.

Kastrinis, George, & Smaragdakis, Yannis. (2013). Hybrid context-sensitivity for points-to analysis. *Pages 423–434 of: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. New York, NY, USA: ACM.

Kennedy, Andrew. (2007). Compiling with continuations, continued. *Pages 177–190 of: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. ICFP '07. New York, NY, USA: ACM.

Koot, Ruud, & Hage, Jurriaan. (2015). Type-based exception analysis for non-strict higher-order functional languages with imprecise exception semantics. *Pages 127–138 of: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*. PEPM '15. New York, NY, USA: ACM.

Lhoták, Ondrej. (2006). *Program analysis using binary decision diagrams*. Ph.D. thesis, McGill University.

Lhoták, Ondřej, & Hendren, Laurie. (2006). *Context-sensitive points-to analysis: Is it worth it?* Theoretical Computer Science and General Issues, vol. 3923. Berlin, Heidelberg: Springer. Pages 47–64.

Lhoták, Ondřej, & Hendren, Laurie. (2008). Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM transactions on software engineering and methodology (TOSEM)*, **18**(1), 3:1–3:53.

Liang, Donglin, Pennings, Maikel, & Harrold, Mary Jean. (2005). Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs. *Pages 6–12 of: Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE '05, vol. 31, no. 1. New York, NY, USA: ACM.

Liang, Shuying, & Might, Matthew. (2012). Hash-flow taint analysis of higher-order programs. *Pages 8:1–8:12 of: Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*. PLAS '12. New York, NY, USA: ACM.

Maurer, Luke, Downen, Paul, Ariola, Zena M., & Peyton Jones, Simon. (2017). Compiling without continuations. *Pages 482–494 of: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. New York, NY, USA: ACM.

Midtgaard, Jan. (2012). Control-flow analysis of functional programs. *ACM computing surveys (CSUR)*, **44**(3), 10:1–10:33.

Midtgaard, Jan, & Horn, David Van. 2009 (May). *Subcubic control flow analysis algorithms*. Computer Science Research Report 125. Roskilde University, Roskilde, Denmark.

Might, Matthew. (2010). *Abstract interpreters for free*. Programming and Software Engineering, vol. 6337. Berlin, Heidelberg: Springer. Pages 407–421.

Might, Matthew, & Manolios, Panagiotis. (2009). *A posteriori soundness for non-deterministic abstract interpretations*. Theoretical Computer Science and General Issues, vol. 5403. Berlin, Heidelberg: Springer. Pages 260–274.

Might, Matthew, & Shivers, Olin. (2006). Improving flow analyses via ΓCFA: abstract garbage collection and counting. *Pages 13–25 of: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '06. New York, NY, USA: ACM.

Might, Matthew, Smaragdakis, Yannis, & Van Horn, David. (2010). Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. *Pages 305–315 of: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. New York, NY, USA: ACM.

Milanova, Ana, Rountev, Atanas, & Ryder, Barbara G. (2005). Parameterized object sensitivity for points-to analysis for Java. *ACM transactions on software engineering and methodology (TOSEM)*, **14**(1), 1–41.

Naik, Mayur, Aiken, Alex, & Whaley, John. (2006). Effective static race detection for Java. *Pages 308–319 of: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '06. New York, NY, USA: ACM.

Oxhøj, Nicholas, Palsberg, Jens, & Schwartzbach, Michael I. (1992). *Making type inference practical*. Lecture Notes in Computer Science, vol. 615. Berlin, Heidelberg: Springer. Pages 329–349.

Palsberg, Jens, & Pavlopoulou, Christina. (2001). From polyvariant flow information to intersection and union types. *Journal of functional programming*, **11**(3), 263–317.

Plotkin, Gordon D. (1981). *A structural approach to operational semantics*. Tech. rept. DAIMI Arhus, Denmark.

Racket Community. (2015). *Racket programming language*. `http://racket-lang.org/`.

Sharir, Micha, & Pnueli, Amir. (1978). Two approaches to interprocedural data flow analysis. *Tech report, new york university*, CSD TR–002.

Sharir, Micha, & Pnueli, Amir. (1981). Two approaches to interprocedural data flow analysis. *Program flow analysis: Theory and applications*, 189–234.

Shivers, Olin. 1991 (May). *Control-flow analysis of higher-order languages*. Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA.

Smaragdakis, Yannis, Bravenboer, Martin, & Lhoták, Ondrej. (2011). Pick your contexts well: understanding object-sensitivity. *Pages 17–30 of: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. New York, NY, USA: ACM.

Tarski, Alfred. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of mathematics*, **5**(2), 285–309.

Van Horn, David, & Mairson, Harry G. (2008). Deciding *k*CFA is complete for EXPTIME. *Pages 275–282 of: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*. ICFP '08. New York, NY, USA: ACM.

Van Horn, David, & Might, Matthew. (2010). Abstracting abstract machines. *Pages 51–62 of: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. New York, NY, USA: ACM.

Vardoulakis, Dimitrios, & Shivers, Olin. (2010). CFA2: a context-free approach to control-flow analysis. *Pages 570–589 of: Proceedings of the European Symposium on Programming*, vol. 6012, LNCS.

Verstoep, Hidde, & Hage, Jurriaan. (2015). Polyvariant cardinality analysis for non-strict higher-order functional languages: Brief announcement. *Pages 139–142 of: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*. PEPM '15. New York, NY, USA: ACM.

Wright, Andrew K., & Jagannathan, Suresh. (1998). Polymorphic splitting: an effective polyvariant flow analysis. *ACM transactions on programming languages and systems (TOPLAS)*, **20**(1), 166–207.