Polyvariance Refined:
A unified theory of polyvariance in abstract interpretations

Thomas Gilray and Matthew Might

University of Utah
{tgilray, might}@cs.utah.edu

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

1 Introduction

The *polyvariance* of an abstract interpretation, in general terms, is the degree to which program locations at runtime are broken into multiple separately maintained approximations of their concrete behavior. Polyvariance has been explored in many different forms including sensitivity to the dynamic type of arguments (CPA), sensitivity to intra-procedural paths (path-sensitivity), sensitivity to the allocation-point of objects (object-sensitivity), and sensitivity to calling context (call-sensitivity), among others [1, 7, 8, 17, 22, 23]. *Call-sensitivity*, an extensively explored form of context-sensitivity, differentiates values passed to a function by an approximate history of k calls leading up to the binding. Informally, it treats functions as globally inlined k iterations for the benefit of precision.

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

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

A 0-call-sensitive (*monovariant*) analysis of this code will merge all values for `x` together so it appears in any context to be either `0` or `#t`. By contrast, a 1-call-sensitive analysis will maintain a separate abstraction for `x` specific to each

of the call-sites for `f`. This is enough to obtain perfect precision for `x`, however a 2-call-sensitive analysis or better will be required to avoid merging for the corresponding parameter of `g`. Only an unbounded degree of polyvariance is enough to guarantee perfect precision in the general case.

Different styles of polyvariance encode a particular fixed heuristic for the trade-off between complexity and precision in an analysis. The efficiency of this trade-off is highly sensitive to the specific program under analysis; a given style of polyvariance will have programs for which it is well-suited, and programs which expose its worst-case behavior. Crucially, for control-flow analyses (CFA) in particular, where a style is most precise, it tends also to be least expensive. This is because imprecision in a CFA requires analyses to explore an even greater range of executions and compounds itself in a vicious cycle [32]. For example, *object-sensitivity* uses the allocation-point of an object for differentiating bindings to variables of its methods and has been shown in practice to be a more effective style of context-sensitivity for object-oriented programs [29].

1.1 Motivating *adaptive* polyvariance: JavaScript

Dynamic languages are notoriously difficult to model precisely. Much work has gone into simply defining a precise semantics of Python and JavaScript. Perhaps the most compelling effort to date at giving a complete and tractable semantics for JavaScript has been Guha et al.'s λ_{JS} [9] and its successor, Politz et al.'s λ_{S5} [26]. This approach reduces programs to a simple core language consisting of fewer than 35 syntactic forms, reifying the hidden and implicit complexity of full JavaScript as explicit complexity written in the core language.

Desugaring is appealing for analysis designers as it gives a simple and precise semantics to abstract; however, it also presents one of the major obstacles to precise analysis as it adds a significant runtime environment and layers of indirection through it. Consider an example the authors of λ_{S5} use to motivate the need for their carefully constructed semantics: `[] + {}` yields the string `"[object Object]"`. Strangely enough, this behavior is correct as defined by the ECMAScript specification for addition – a complex algorithm encompassing a number of special cases which can interact in unexpected ways [6]. The desugaring process for λ_{S5} replaces addition with a function call to `%PrimAdd` from the runtime environment. `%PrimAdd` in-turn calls `%ToPrimitive` on both its arguments before breaking into cases. This means that for any uses of addition to return precise results, or likely anything other than $\top$, a k-call-sensitive analysis requires $k \geq 2$. Unfortunately, call-sensitivity, and any other context-sensitive analysis which allows the number of variants for a binding to be in $\Omega(n)$ will be exponential in the worst-case due to the structure of environments in higher-order languages [11, 22]. As the desugarer for λ_{S5} bloats even small programs to more than ten thousand lines, applying context-sensitivity to every function is sure to result in an absurdly time-consuming analysis.

Instead, an adaptive style of polyvariance is required, one which attempts an increased precision only where it will be safe, effective, and efficient. This requires a simultaneous modeling of both program behavior and analysis behavior for

determining where polyvariance may be best exploited; however, because the model produced by such an analysis is not sound and meaningful until analysis is complete, adjusting polyvariance before this is unlikely to be effective. A method for incrementally refining the polyvariance of a sound fix-point is therefore a more plausible way of fitting the application of polyvariance to suit a program.

1.2 Contributions

Our work makes the following novel contributions:

1. We refine the definition of *polyvariance*, producing a parametric framework (over allocators; see §3 and §5) and a unified theory which generalizes all well-ordered hierarchies such as k-call-sensitivity or k-full-object-sensitivity to a lattice of analyses encompasing all conceivable styles of polyvariance.
2. We survey a number of published strategies for polyvariance and show how they may be easily formalized within our framework (§4).
3. We show how to efficiently refine a sound fix-point to an incrementally more precise position in the lattice of polyvariant analyses (§6).
4. We elucidate the unsoundness of refining store-widened analyses according to the naïve method and produce a meta-analysis we term *contribution*-analysis which enables sound refinement in the presence of store-widening (§6.4).

2 Background

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

2.1 Myriad styles of polyvariance

Originally devised as a technique for data-flow analyses by Sharir and Pnueli [27], polyvariance was utilized in the '80s by Jones, Muchnick, and Harrison [15, 10] and generalized to a k-call-sensitive control-flow analysis (k-CFA) of higher-order programs by Shivers [28]. In nearly 25 years since the seminal work on k-CFA, numerous alternative styles of polyvariance for CFA have been explored in the literature. The '90s in particular saw an extensive exploration of this design-space. Like k-CFA, many frameworks form a well-ordered hierarchy of analyses encompassing a range of granularities between monovariance and unbounded polyvariance (corresponding to a concrete semantics). Each presents the inexorable trade-off between precision and complexity in a unique way.

Milanova et al. introduced a style of polyvariance which uses a history of the allocation-points for objects [23]. Smaragdakis et al. refined this concept, distinguishing multiple hierarchies encoding different flavors of object-sensitivity along with a new approach called *type-sensitivity* which approximates these [29].

Agesen defined a polyvariant framework for type-recovery which gains the best possible precision for its application by directly using a tuple of types to

differentiate functions for analysis [1]. His *Cartesian product algorithm* (CPA) may also be formalized as a style of polyvariance for general flow-analyses [7].

Wright and Jagganathan's *polymorphic-splitting* uses let-bindings as a heuristic for varying call-sensitivity on a per-function basis [32].

Palsberg and Pavlopoulou, among many others, have applied polyvariance to enhancing the precision of type-systems [24].

In their unified treatment of flow-analysis, Jagganathan and Weeks give a polynomial-time widening for call-sensitivity: poly-k-CFA [14]. While this is possible for any style of polyvariance, in an unwidened form all these styles of polyvariance are necessarily exponential in their worst-case behavior.

2.2 The AAM methodology

The Abstracting Abstract-Machines (AAM) approach of Van Horn and Might is a general method for automatically abstracting an arbitrary small-step abstract-machine semantics to obtain an approximation in a variety of styles [12, 19]. Importantly, one such style aims to focus all unboundedness in a semantics on the machine's address-space. We exploit this style of abstraction to produce a framework parameterized by an allocation function mapping variables to abstract addresses as they are bound. Combined with a meta-analysis suitable for tracking a particular flavor of context (e.g. allocation-histories), we show that this function is powerful enough to encode an arbitrary style of polyvariance.

2.3 *A posteriori* soundness

The usual process for proving the soundness of an abstract abstract-machine is *a priori* in the sense that it may be performed entirely before an analysis is executed. The approximation used, formally known as a *Galois connection*, defines an abstraction/concretization relationship between the concrete and abstract semantics. A family of functions α map entities in the concrete semantics to their most precise representative in the abstract semantics. A corresponding family of functions γ map entities in the abstract machine to a set of concrete entities such that $Id \sqsubseteq \gamma \circ \alpha$, placing a strict bound on the concrete executions represented by an analysis result. Proving that an abstract transition relation is sound with respect to a concrete one reduces to an inductive step showing that simulation, (i.e. the defined notion of approximation, γ) is preserved across every transition [18]. Details may be found in §A.2.

By contrast, Might and Manolios describe an *a posteriori* soundness proof where the abstraction map cannot be fully constructed until after analysis [20]. This approach factors α to separate the abstraction of addresses α_{Addr}, producing a parametric map β such that $\beta(\alpha_{Addr}) = \alpha$. The authors show that regardless of the allocation strategy taken during analysis, a consistent abstraction map may be constructed after-the-fact which justifies these choices. Intuitively, because each allocation-step in a concrete machine is ephemeral, and each address unique, it is impossible to induce an inconsistent abstraction for addresses by any abstract allocation policy. Details for this approach are presented in §A.3.

3 Semantics

In this section we introduce our general semantics framework tuned both to precise evaluation and to a monovariant flow-analysis for the pure λ-calculus in *continuation-passing-style* (CPS). CPS constrains call-sites to tail-position so functions may never return; instead, callers must explicitly pass a continuation forward to be invoked on the result [25]. CPS is an excellent and widely used transformation for compiler optimization and program analysis [2]. If the transformation to CPS records which lambdas correspond to continuations, a program may again, along with any optimizations and analysis results, be precisely reconstituted in direct-style form. This means the advantages of CPS can be utilized without compromise or loss of information [16]. The grammar structurally distinguishes between complex-expressions e and atomic-expressions ae:

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

We define this language using a relation ($\rightsquigarrow$), over configurations of an abstract-machine, that determines formally how the machine transitions from one state to another. Configurations $\hat{\varsigma} \in \hat{\Sigma}$ are factored into $\widehat{Eval}$ and $\widehat{Apply}$ states. $\widehat{Eval}$ states range over expression-context e, binding-environment $\hat{\rho}$, value-store $\hat{\sigma}$, and meta-analysis $\hat{m}$ components. A machine in this state is ready to evaluate its expression e by transitioning to an $\widehat{Apply}$ state. $\widehat{Apply}$ states range over closure $\widehat{clo}$, evaluated argument list $(\hat{d}\ \ldots)$, store $\hat{\sigma}$, and meta-analysis $\hat{m}$ components. A machine in this state is ready to apply its closure and make new bindings in the store, transitioning back to an $\widehat{Eval}$ configuration.

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

Environments $\hat{\rho}$ map variables in scope to the address for the visible binding. Value-stores $\hat{\sigma}$ map these addresses to *flow-sets* of values (in this case, closures). These may be thought of as a model of the heap. Meta-analyses $\hat{m}$ are a generic hook in our semantics for extending and tuning its behavior. This will take on many forms as we explore styles of polyvariance and extensions to the analysis.

Evaluation of atomic-expressions is handled by an auxiliary function $\hat{\mathcal{A}}$ which produces a set of values for an atomic-expression in the context of an $\widehat{Eval}$ state. This is done by lookup in the environment and store for variable references, and by closure-creation for λ-abstractions. In a language containing syntactic literals,

these would be translated into equivalent semantic values.

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

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

$$\overbrace{((ae_f\ ae_1\ \ldots\ ae_j),\ \hat{\rho},\ \hat{\sigma},\ \hat{m})}^{\hat{\varsigma}} \hat{\rightsquigarrow} (\widehat{clo}_f,\ (\hat{d}_1\ \ldots\ \hat{d}_j),\ \hat{\sigma},\ \hat{m}')$$
$$\text{where} \quad \widehat{clo}_f \in \hat{\mathcal{A}}(ae_f,\ \hat{\varsigma})$$
$$\hat{d}_i = \hat{\mathcal{A}}(ae_i,\ \hat{\varsigma})$$
$$\hat{m}' = \hat{\mathcal{M}}(\hat{\varsigma})$$

$\widehat{Eval} \hat{\rightsquigarrow} \widehat{Apply}$ transitions are non-deterministic when multiple closures are indicated for the expression ae_f in call-position. An $\widehat{Apply}$ state results for each such closure. An as-yet undefined function $\hat{\mathcal{M}}$ may be tuned in isolation to transition the meta-analysis component.

$$\overbrace{((\texttt{(}\lambda\ \texttt{(}x_1\ \ldots\ x_j\texttt{)}\ e_\lambda\texttt{)},\ \hat{\rho}_\lambda),\ (\hat{d}_1\ \ldots\ \hat{d}_j),\ \hat{\sigma},\ \hat{m})}^{\hat{\varsigma}} \hat{\rightsquigarrow} (e_\lambda,\ \hat{\rho}',\ \hat{\sigma}',\ \hat{m}')$$
$$\text{where} \quad \hat{\rho}' = \hat{\rho}_\lambda[x_i \mapsto \hat{a}_i]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{d}_i]$$
$$\hat{a}_i = \widehat{alloc}(x_i,\ \hat{\varsigma})$$
$$\hat{m}' = \hat{\mathcal{M}}(\hat{\varsigma})$$

$\widehat{Apply}$ states transition deterministically to an $\widehat{Eval}$ state by making the necessary bindings in $\hat{\rho}'$ and $\hat{\sigma}'$ and moving control inside the body e_λ of the applied function. Crucially, the allocator $\widehat{alloc}$ may be tuned to determine how these semantics allocate addresses for variables when they are bound. Because a state may not be solely responsible for binding these addresses, weak-update ($\sqcup$) is used on the store. As for all functions in our semantics, join is defined as the natural point-wise lifting, i.e. $(\hat{\sigma}_1 \sqcup \hat{\sigma}_2)(\hat{a}) = \hat{\sigma}_1(\hat{a}) \sqcup \hat{\sigma}_2(\hat{a})$.

A state becomes stuck if `(halt)` is reached or if the program is malformed.

3.1 Tuning for concrete semantics

To obtain a concrete semantics in this framework, we use a simple meta-analysis which records every expression execution passes through. The allocator then uses this history to produce a fresh address for every binding. No two addresses will

be the same because $\hat{m}$ is a perfect program trace, extended at every expression. As every flow-set is therefore a singleton, all transitions are deterministic.

$$\widehat{alloc}_\bot : \mathsf{Var} \times \widehat{Apply} \to \widehat{Addr} \qquad \hat{\mathcal{M}} : \hat{\Sigma} \to \hat{M}$$
$$\widehat{alloc}_\bot(x,\ (\widehat{clo}_f,\ (\ldots),\ \hat{\sigma},\ \hat{m})) = (x, \hat{m}) \qquad \hat{\mathcal{M}}(e^l,\ \hat{\rho},\ \hat{\sigma},\ \hat{m}) = l : \hat{m}$$
$$\hat{\mathcal{M}}(\widehat{clo}_f,\ (\ldots),\ \hat{\sigma},\ \hat{m}) = \hat{m}$$

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

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

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

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

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

3.2 Tuning for univariance and monovariance

As our approach to semantics focuses the unboundedness of a machine's state-space exclusively on addresses in the store, a properly defined abstract address allocator $\widehat{alloc}$ is the one essential component required to obtain a bounded approximation. The semantics is guaranteed to be computable so long as $\widehat{alloc}$ is only permitted to produce a bounded number of addresses. This in turn introduces merging between values in the store and non-determinism in the transition relation. The most extreme version of this is the *univariant* allocator which only produces a single address $\top$ that over-approximates all concrete addresses.

$$\widehat{alloc}_\top : \mathsf{Var} \times \widehat{Apply} \to \widehat{Addr}^\top \qquad \hat{\mathcal{M}} : \hat{\Sigma} \to \hat{M}$$
$$\widehat{alloc}_\top(x,\ \hat{\varsigma}) = \top \qquad \hat{\mathcal{M}}(\hat{\varsigma}) = ()$$

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

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

A flow-set for an address $\hat{a}$ indicates a range of values which over-approximates all possible concrete values that can flow to any concrete address approximated by $\hat{a}$. For example, if a concrete machine binds $(y, (l_{14} \ldots)) \mapsto \{\widehat{clo}_1\}$ and $(y, (l_{38} \ldots)) \mapsto \{\widehat{clo}_2\}$, its monovariant approximation might bind $(y, ()) \mapsto \{\widehat{clo}_1, \widehat{clo}_2\}$. Precision is lost for $(y, (l_{14} \ldots))$ 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.

Naïvely computing 0-CFA. $\hat{f}$ is monotonic and continuous. When the address space is finite, it operates over a finite lattice $\mathcal{P}(\hat{\Sigma})$ and we therefore know that $\hat{f}^n(\bot)$ for some n will be a least-fixed-point of $\hat{f}$ [3, 30]. Unfortunately, the worst-case complexity of this calculation is exponential as the total number of stores which may need to be explored is in $O(2^{n^2})$ [22].

Efficiently computing 0-CFA. *Store-widening* is an essential technique for combating this complexity. It relaxes the relationship between machine configurations and the store in order to explore fewer stores during an analysis. Per-context widening maintains a single store for each context as the least-upper-bound of all stores reachable at that context. Per-configuration widening maintains a single store for each pair of expression-context and environment. Global store-widening maintains a single store for the entire system-space. These approximations tend to produce significantly faster analyses with only slightly decreased precision in practice [18, 28]. With global store-widening applied and bit-packing implementation tricks, 0-CFA is in $O(\frac{n^3}{log\ n})$ [17].

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

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

The notation $\_$ is for matching any value. Equivalently, we may lift the store out of machine configurations entirely and compute an inherently widened system-space $\hat{\xi}$ which pairs a single store with a set of reachable configurations.

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

Its transfer function $\hat{f}_\triangledown$ (effectively $\triangledown \circ \hat{f} \circ \triangledown$) uses the global store to transition each configuration and computes a new store as the least-upper-bound of all stores at this frontier. A more compact formulation is given in §A.4.

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

We define $\hat{\xi}_0 = (\{(e, \varnothing, ())\}, \bot)$. As we will eventually permit (and exploit) the address-space itself forming a lattice (§5), we may define an analogous *environment-widened* analysis where the join of environments $\hat{\rho}$ is computed directly. A per-context environment-widened analysis is defined in §A.5.

3.3 Extension to real languages

We will continue to explore analysis of our extremely simple CPS language; however, setting up a semantics for real language features such as conditionals, primitive operations, direct-style recursion, or exceptions, is no more difficult, if more verbose. Handling other forms is often as straightforward as including an additional transition rule for each. Consider the inclusion of a `set!` form enabling mutation/side-effects:

$$\begin{aligned} e \in \mathsf{E} ::= & \ (\text{set!}\ x\ æ_v\ ae_\kappa) \\ | & \ \ldots \end{aligned}$$

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

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

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

Supporting direct-style recursion requires a big-step semantics or an explicit stack as continuations are no longer baked into the source text by CPS-conversion. This can be done by modeling a stack within each concrete state and store-allocating continuation frames on abstraction to obtain a finite-state model [12]. A more precise option is to directly abstract a pushdown machine, labeling edges in the system with stack actions to obtain a Dyck state graph [4, 5, 31].

3.4 Parametricity

Crucially, as meta-analyses $\hat{\mathcal{M}}$ strictly increase information at any given state, they may be tuned *arbitrarily* without affecting the soundness of our core flow-analysis. Allocators do impact the core flow-analysis, but may also be tuned arbitrarily because the *a posteriori* soundness proof [20] ensures a consistent abstraction for addresses will always result (§A.3). Meta-analyses used only to benefit the range of allocators we can express are not restricted by any notion of soundness. By including this additional information, we expand the set of distinct allocators we can express, enabling arbitrary polyvariance. Other aspects of our framework may also be changed so long as they respect soundness. In §4.3 for example, we modify $\hat{\mathcal{A}}$ in a way that strictly expands on the approximation we use for closures. Soundness is respected as no information is removed.

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

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

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

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

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

All possible tunings $CFA(\widehat{alloc})$ soundly over-approximate $CFA(\widehat{alloc}_\bot)$. Self-contained definitions for CFA and $CFA_\triangledown$ are given in §A.4.

4 Strategies for Polyvariance

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

4.1 Call-sensitivity

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

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

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

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

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

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

4.2 Variable call-sensitivity (polymorphic-splitting)

Wright et al.'s polymorphic-splitting is a form of adaptive call-sensitivity where the degree of polyvariance may vary between functions [32]. The number of let definitions a lambda was originally defined within (before CPS-conversion) forms a simple heuristic for its call-sensitivity when invoked. To implement this style of polyvariance, we assume an auxiliary function $\mathcal{L}$ for encoding this let-depth.

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

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

$$\hat{\mathcal{M}}(e^l,\ \hat{\rho},\ \hat{\sigma},\ \hat{m}) = l : \hat{m}$$

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

This call-history is then used for allocating addresses.

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

Clearly, all possible heuristics $\mathcal{L}$ are sound. For this simple tuning, it is easy to produce an example which exposes an exponential worst-case running time.

4.3 Closure-sensitivity (object-sensitivity)

Smaragdakis et al. [29] distinguish multiple variants of object-sensitivity, first described by Milanova et al. [23]. This style of context uses a history of the allocation-points for objects to guide polyvariance. We define an equivalent to the more precise variant of object-sensitivity: k-full-object-sensitivity. In higher-order languages this style would track the point of closure-creation for a lambda, the point of closure-creation for the lambda that allocated it, and so forth.

$$\widehat{Clo} = \mathsf{Lam} \times \widehat{Env} \times \widehat{Meta} \qquad \widehat{Meta} = \mathsf{Lab}^k$$

$$\widehat{Addr} = \mathsf{Var} \times \widehat{Meta}$$

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

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

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

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

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

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

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

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

The authors of k-full-object-sensitivity also define a novel approximation for it called type-sensitivity which over-approximates allocation-points by the class in which they occur. We give a tuning for this style of context in §A.7.

4.4 Argument-sensitivity (CPA)

Agesen's CPA directly differentiates flows to a function by the entire tuple of types being passed as arguments [1]. For a closure, we may assume its type is unique to its syntactic lambda even if unknown before analysis [7]. We use a *type-of* function $\mathcal{T}(\hat{d}) = \{lam : (lam, \hat{\rho}) \in \hat{d}\}$ that removes environments.

$$\widehat{Addr} = \mathsf{Var} \times \mathcal{P}(\mathsf{Lam})^*$$

$$\widehat{alloc}_{cpa}(x,\ (\widehat{clo}_f,\ (\hat{d}_1\ \ldots\ \hat{d}_j),\ \hat{\sigma},\ \hat{m})) = (x, (\mathcal{T}(\hat{d}_1)\ \ldots\ \mathcal{T}(\hat{d}_j)))$$

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

We may easily mix and match these styles of polyvariance, or invent new ones. Consider combining call-sensitivity (using $\hat{\mathcal{M}}$ from §4.1) with an argument-sensitivity that uses only the final continuation-argument at each call-site.

$$\widehat{Addr} = \mathsf{Var} \times (\widehat{Meta} \times \mathcal{P}(\mathsf{Lam}))$$

$$\widehat{alloc}_{mix}(x,\ (\widehat{clo}_f,\ (\hat{d}_1\ \ldots\ \hat{d}_j),\ \hat{\sigma},\ \hat{m})) = (x, (\hat{m}, \mathcal{T}(\hat{d}_j)))$$

5 A Lattice of Analyses

In general, we may use a lattice of addresses $(\widehat{Addr},\ \sqsubseteq)$ for mixing allocators such that $\hat{a}_1 \sqcap \hat{a}_2$ approximates addresses approximated by both $\hat{a}_1$ and $\hat{a}_2$:

$$(\alpha_{Addr_1} \sqcap \alpha_{Addr_2})(a) \sqsubseteq \hat{a}_1 \sqcap \hat{a}_2 \iff \alpha_{Addr_1}(a) \sqsubseteq \hat{a}_1 \wedge \alpha_{Addr_2}(a) \sqsubseteq \hat{a}_2 \quad (\S 2.3)$$

Meet and join for allocators may then be defined naturally as a point-wise lifting.

In addition, our order on the precision of analysis results gives rise to possible definitions for their meet. Given results $\hat{\xi}_1$ and $\hat{\xi}_2$, produced by two different strategies for polyvariance $CFA_\triangledown(\widehat{alloc}_1)$ and $CFA_\triangledown(\widehat{alloc}_2)$, their greatest-lower-bound $\hat{\xi}_1 \sqcap \hat{\xi}_2$ may be defined as an analysis result proving all properties proven by either $\hat{\xi}_1$ or $\hat{\xi}_2$. Definitions for meet and join are discussed in §A.6.

As it turns out, the parametric frameworks CFA and $CFA_\triangledown$ form embeddings of the address-lattice. That is to say, performing an analysis with $\widehat{alloc}_1 \sqcap \widehat{alloc}_2$ obtains precisely the same result as if each analysis had been computed separately and their results merged, except without computing both.

Theorem 1. *CFA and $CFA_\triangledown$ are lattice-embeddings.*

$$CFA_\triangledown(alloc_1 \sqcup alloc_2) = CFA_\triangledown(alloc_1) \sqcup CFA_\triangledown(alloc_2)$$
$$CFA_\triangledown(alloc_1 \sqcap alloc_2) = CFA_\triangledown(alloc_1) \sqcap CFA_\triangledown(alloc_2)$$

As transfer functions are ordered by their results, a proof of this second property for $CFA_\triangledown$ requires a base case $\hat{\xi}_0 = \hat{\xi}_0 \sqcap \hat{\xi}_0$ and inductive step:

$$\hat{\xi} = \hat{\xi}_1 \sqcap \hat{\xi}_2 \implies$$
$$CFA_\triangledown(alloc_1 \sqcap alloc_2)(\hat{\xi}) = CFA_\triangledown(alloc_1)(\hat{\xi}_1) \sqcap CFA_\triangledown(alloc_2)(\hat{\xi}_2)$$

The proof-sketch for an environment-widened analysis is provided in §A.6.

6 Incremental Refinement

As the *a posteriori* soundness theorem has permitted us to tune allocation without restriction, we may use an arbitrary strategy for making an analysis directly *precision-sensitive* while the fix-point computation is still live. However, because an incomplete analysis makes no strict guarantees, we don't have reliable information on which to base such precision-sensitivity. A more complete framework for arbitrary polyvariance therefore must permit the refinement of a sound fixpoint based on strong guarantees made for analysis behavior.

6.1 Naïve refinement

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

$$\hat{s} \sqsubseteq \hat{s}' \implies CFA(\widehat{alloc})(\hat{s}) \sqsubseteq CFA(\widehat{alloc})(\hat{s}')\,,$$

and not simply that

$$\hat{s} \sqsubseteq CFA(\widehat{alloc})(\hat{s})\,,$$

it is possible for entire unreachable states to be removed one by one until a new fix-point is reached. It is also possible however, since $CFA(\widehat{alloc})(\hat{s}) \sqsubseteq \hat{s}$ can also not be relied on to hold inductively, that additions and subtractions to the system-space may be made simultaneously such that a newly unreachable loop is explored ad infinitum. The solution is to cull unreachable states first.

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

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

6.2 Impact of store-widening

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

6.3 Analysis analysis

In §3 and §4, $\widehat{Meta}$ found many uses tracking different flavors of program context and thereby enabling a greater variety of allocators. There are also many other potential extensions to an analysis, including those which aim to model and approximate properties of the analysis at analysis-time instead of properties of the program at run-time. We elucidate the potential importance of such *analysis analyses* and give an example, motivating our generalized solution for refinement.

Abstract-counting. One such analysis analysis is *abstract-counting* [18, 21]. Abstract-counts are a meta-analysis inspired by Hudak's abstract reference-counting which over-approximates the number of references to an object so compiler optimizations may be applied for updating data in-place instead of requiring a fresh allocation [13]. Abstract-counting permits essentially the same optimization to be applied at analysis-time so strong-update may be used on a flow-set (among other applications). The approximation used by a static analysis merges bindings in the store, throwing away the information necessary to permit strong-update. Stores must normally increase across transitions because no specific transition may assume it is exclusively responsible for an abstract binding except where it is known to only represent a single concrete binding.

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

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

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

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

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

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

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

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

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

$$\text{where} \qquad \hat{m}' = \hat{m} \sqcup [\hat{a}_i \mapsto 1]$$
$$\hat{a}_i = \widehat{alloc}(x_i, \hat{\varsigma})$$

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

6.4 Contribution-analysis

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

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

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

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

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

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

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

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

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

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

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

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

7 Future Work

Our progress in this work leads us to a number of interesting questions. Most immediate of these is how to best guide the refinement of $\widehat{alloc}$ in practice.

Analysis analyses for refinement. Frequently in an abstract interpretation, a crucial loss of precision has its source far removed from many of the effects. We believe there will be a variety of analysis analyses which approximate this aspect of analysis behavior, tracking the source to blame for each plausibly important imprecision. Once it is explicit, such information could be used to enable cheap heuristics for guiding the refinement of polyvariance, carving a path upwards through the lattice of analyses along a more ideal path than those which are described by call-sensitivity, object-sensitivity, or indeed *any* fixed (pre-determined) strategy for polyvariance. Deciding which such meta-analyses are most effective in practice will require a careful empirical study.

How complete is polyvariance for precision? Whether or not polyvariance is capable of expressing all sound granularities of precision is an interesting open question. While clearly we may express any precision required to prove a specific property, it is not clear that the least-such precision is always expressable.

Another way of phrasing this question might be: *is CFA surjective for a co-domain restricted to sound analyses*? Clearly it is not for all possible transfer functions as we cannot express an unsound analysis in our framework. Devising a lattice of all sound analyses for an arbitrary program e may itself be a difficult problem, especially if the result is greatly affected by language features.

In the likely case that there are granularities of precision which may not be expressed, this question generalizes to asking if such degrees are interesting in practice. Is there a clean way of factoring out a second lattice, for which the product of both is isomorphic to all conceivable granularities of precision?

8 Conclusion

We formalized the meaning of the term *polyvariant* as all strategies for abstracting concrete instances of a variable's data to *more than one* approximation. We developed a framework which may be tuned to soundly express an analysis using any abstraction for program locations and show that it naturally encompases previous styles of analysis described as being *polyvariant*. We produced the first algorithm for sound and arbitrary refinement of polyvariance after reaching a sound intermediate result. Finally, we used a novel meta-analysis of the states which contribute a binding to permit refinement of store-widened analyses.

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A Appendix

Details which could not be included in the paper's body due to space constraints.

A.1 A traditional concrete semantics

For completeness, we include a traditional concrete semantics for our language. These are equivalent to the semantics provided in [18].

$$\begin{aligned} \varsigma \in \Sigma &= \mathit{Eval} + \mathit{Apply} & \rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup \mathit{Addr} \\ \mathit{Eval} &= \mathsf{E} \times \mathit{Env} \times \mathit{Store} \times \mathit{Meta} & \sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Clo} \\ \mathit{Apply} &= \mathit{Clo} \times \mathit{Clo}^* \times \mathit{Store} \times \mathit{Meta} & a \in \mathit{Addr} &= \mathsf{Var} \times \mathbb{N} \\ & & \mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{Env} \\ & & m \in \mathit{Meta} &= \mathbb{N} \end{aligned}$$

Addresses precisely indicate a single closure in the store.

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

Atomic-expression evaluation likewise results in an exact closure.

$$\begin{aligned} \mathit{alloc} &: \mathsf{Var} \times \mathit{Apply} \to \mathit{Addr} \\ \mathit{alloc}(x,\ (\mathit{clo},\ (\ldots),\ \sigma,\ m)) &= (x, m) \end{aligned} \qquad \begin{aligned} \mathcal{M} &: \Sigma \to M \\ \mathcal{M}(\mathit{clo},\ (\ldots),\ \sigma,\ m) &= m + 1 \\ \mathcal{M}(e,\ \rho,\ \sigma,\ m) &= m + 1 \end{aligned}$$

We count the number of states previously visited in m to obtain a unique timestamp without implying any particular kind of history.

$$\overbrace{((\mathit{ae}_f\ \mathit{ae}_1\ \ldots\ \mathit{ae}_j),\ \rho,\ \sigma,\ m)}^{\varsigma} \Rightarrow (\mathit{clo}_f,\ (\mathit{clo}_1\ \ldots\ \mathit{clo}_j),\ \sigma,\ m')$$

$$\begin{aligned} \text{where} \quad & \mathit{clo}_f = \mathcal{A}(\mathit{ae}_f,\ \varsigma) \\ & \mathit{clo}_i = \mathcal{A}(\mathit{ae}_i,\ \varsigma) \\ & m' = \mathcal{M}(\varsigma) \end{aligned}$$

Eval $\Rightarrow$ *Apply* transitions are always deterministic.

$$\overbrace{((\lambda\ (x_1\ \ldots\ x_j)\ e_\lambda)\ \rho_\lambda),\ (\mathit{clo}_1\ \ldots\ \mathit{clo}_j),\ \sigma,\ m)}^{\varsigma} \Rightarrow (e_\lambda,\ \rho',\ \sigma',\ m')$$

$$\begin{aligned} \text{where} \quad & \rho' = \rho_\lambda[x_i \mapsto a_i] \\ & \sigma' = \sigma[a_i \mapsto \mathit{clo}_i] \\ & a_i = \mathit{alloc}(x_i, \varsigma) \\ & m' = \mathcal{M}(\varsigma) \end{aligned}$$

All addresses a_i are fresh and unique so strong-update of the store is used.

A.2 *A priori* soundness: a Galois connection for 0-CFA

To formally define the abstraction connecting this concrete machine with our framework tuned to 0-CFA (§3.2) we produce a Galois connection $\Sigma \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \hat{\Sigma}$. The abstraction map for states α_Σ defines, for a given state ς, a most precise abstract state $\alpha_\Sigma(\varsigma)$. Any state $\hat{\varsigma}$ may be said to soundly simulate ς if and only if $\alpha_\Sigma(\varsigma) \sqsubseteq \hat{\varsigma}$. This is also precisely the definition of $\varsigma \in \gamma_\Sigma(\hat{\varsigma})$.

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

To prove that the least-fix-point of $\hat{f}$ is always a sound simulation of the fix-point for f (or of its unbounded trace), we are only required to show a base case $\alpha(\varsigma_0) \sqsubseteq \hat{\varsigma}_0$ and an inductive step:

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

Which shows that simulation is preserved across every transition.

A.3 *A posteriori* soundness: a factored Galois connection

Just as we factored our semantics, we may factor our Galois connection to be parameterized over the address abstraction maps which justifying a given allocator as sound.

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

Where α_{Addr} is the abstraction map justifying a monovariant allocator

$$\alpha_{Addr}(x, m) = (x, ())$$

we may note the correspondence to α_Σ from §A.2: $\alpha_\Sigma = \beta_\Sigma(\alpha_{Addr})$, i.e.

$$\alpha_\Sigma(\varsigma) = \hat{\varsigma} \iff \beta_\Sigma(\alpha_{Addr})(\varsigma) = \hat{\varsigma}$$

In a worked bisimulation proof for *a priori* soundness, it is easy to see that the required justification of an allocator $\widehat{alloc}$ against a mapping from concrete addresses to abstract addresses α_{Addr} is modular and unaffected by other portions of the proof. This means that so long as a self-consistent abstraction-map exists for an allocator, it may be proven sound. Allocation maps are a special component of any analysis which provably *always* meets this condition. Because no concrete address is ever produced twice in a concrete semantics, no inconsistant abstraction-map for addresses mapping two equivalent concrete addresses to different abstract addresses may ever be induced by an abstract allocator. By contrast, one may observe that this is not true for other analysis components such as expression-contexts or binding-environments because arbitrary strategies for transitioning these components could imply a self-inconsistant abstraction.

This condition is given a general formalization as the *dependent simulation condition* for a non-turing-complete *Malloc* language in [20] along with a proof of the address abstraction-map's *a posteriori* construction. A formulation of this construction for our framework is straightforward and analogous.

A.4 Compact definitions for CFA, CFA_∇

We give simple self-contained definitions for CFA and CFA_∇ using no meta-analysis and a single representation for states (as opposed to factoring Eval from Apply) using a combined transition.

$$
\begin{aligned}
CFA(\widehat{alloc})(\hat{s}) = \{(e',\ \hat{\rho}',\ \hat{\sigma}') : {} & \overbrace{(\texttt{(}ae_f\ ae_1\ \ldots\ ae_j\texttt{)},\ \hat{\rho},\ \hat{\sigma})}^{\hat{\varsigma}} \in \hat{s} \\
& (\texttt{(}\lambda\ \texttt{(}x_1\ \ldots\ x_j\texttt{)}\ e_\lambda\texttt{)},\ \hat{\rho}_\lambda) \in \hat{\mathcal{A}}(ae_f, \hat{\rho}, \hat{\sigma}) \\
& \hat{a}_i = \widehat{alloc}(x_i, \hat{\varsigma}) \\
& \hat{d}_i = \hat{\mathcal{A}}(ae_i,\ \hat{\rho},\ \hat{\sigma}) \\
& \hat{\rho}' = \hat{\rho}_\lambda[x_i \mapsto \hat{a}_i] \\
& \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{d}_i]\ \} \cup \{\hat{\varsigma}_0\}
\end{aligned}
$$

$$
\begin{aligned}
& CFA_\nabla(\widehat{alloc})(\hat{r}, \hat{\sigma}) = (\hat{r}', \hat{\sigma}') \sqcup \hat{\xi}_0 \\
& \text{where} \quad \hat{r}' = \{\ (e,\ \hat{\rho}) : (\texttt{(}ae_f\ ae_1\ \ldots\texttt{)}, \hat{\rho}_r) \in \hat{r} \\
& \qquad\qquad (\texttt{(}\lambda\ \texttt{(}x_1\ \ldots\texttt{)}\ e\texttt{)}, \hat{\rho}_\lambda) \in \hat{\mathcal{A}}(ae_f, \hat{\rho}_r, \hat{\sigma}) \\
& \qquad\qquad \hat{\rho} = \hat{\rho}_\lambda[x_i \mapsto \widehat{alloc}(x_i, (\texttt{(}ae_f\ \ldots\texttt{)}, \hat{\rho}_r, \hat{\sigma}))]\ \} \\
& \qquad \hat{\sigma}' = \{\ (\hat{a},\ \hat{d}) : (\texttt{(}ae_f\ ae_1\ \ldots\texttt{)}, \hat{\rho}_r) \in \hat{r} \\
& \qquad\qquad (\texttt{(}\lambda\ \texttt{(}x_1\ \ldots\texttt{)}\ e\texttt{)}, \hat{\rho}_\lambda) \in \hat{\mathcal{A}}(ae_f, \hat{\rho}_r, \hat{\sigma}) \\
& \qquad\qquad \hat{a} = \widehat{alloc}(x_i, (\texttt{(}ae_f\ \ldots\texttt{)}, \hat{\rho}_r, \hat{\sigma})) \\
& \qquad\qquad \hat{d} = \hat{\mathcal{A}}(ae_i, \hat{\rho}_r, \hat{\sigma})\ \}
\end{aligned}
$$

An atomic-expression evaluator for both:

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

A.5 Environment-widening

A lattice of addresses $(\widehat{Addr}, \sqsubseteq)$ leads to a more elegant definition for a binding-environment widened analysis as we can directly compute the join of any two addresses indicated for a variable in order to compute the join of two binding-environments. We provide a self-contained definition for per-context environment widening:

$$CFA_{\nabla_\rho} : \widehat{Alloc} \to ((\mathsf{E} \to \widehat{Env}) \times \widehat{Store}) \to ((\mathsf{E} \to \widehat{Env}) \times \widehat{Store})$$
$$CFA_{\nabla_\rho}(\widehat{alloc})(\hat{r}, \hat{\sigma}) = (\hat{r}', \hat{\sigma}') \sqcup \hat{\xi}_0$$

where

$$\hat{r}' = \bigsqcup\{\, [e \mapsto \hat{\rho}] \;:\; (\texttt{(}ae_f\ ae_1\ \ldots\texttt{)}, \hat{\rho}_r) \in \hat{r}$$
$$(\texttt{(}\lambda\ \texttt{(}x_1\ \ldots\texttt{)}\ e\texttt{)}, \hat{\rho}_\lambda) \in \hat{\mathcal{A}}(ae_f, \hat{\rho}_r, \hat{\sigma})$$
$$\hat{\rho} = \hat{\rho}_\lambda[x_i \mapsto \hat{a}_i]$$
$$\hat{a}_i = \widehat{alloc}(x_i, (\texttt{(}ae_f\ \ldots\texttt{)}, \hat{\rho}_r, \hat{\sigma}))\ \}$$

$$\hat{\sigma}' = \bigsqcup\{\, [\hat{a} \mapsto \hat{d}] \;:\; (\texttt{(}ae_f\ ae_1\ \ldots\texttt{)}, \hat{\rho}_r) \in \hat{r}$$
$$(\texttt{(}\lambda\ \texttt{(}x_1\ \ldots\texttt{)}\ e\texttt{)}, \hat{\rho}_\lambda) \in \hat{\mathcal{A}}(ae_f, \hat{\rho}_r, \hat{\sigma})$$
$$\hat{a} = \widehat{alloc}(x_i, (\texttt{(}ae_f\ \ldots\texttt{)}, \hat{\rho}_r, \hat{\sigma}))$$
$$\hat{d} = \hat{\mathcal{A}}(ae_i, \hat{\rho}_r, \hat{\sigma})\ \}$$

Its atomic-expression evaluator:

$$\hat{\mathcal{A}}(x,\ \hat{\rho},\ \hat{\sigma}) = \hat{\sigma}(\hat{\rho}(x))$$
$$\hat{\mathcal{A}}(lam,\ \hat{\rho},\ \hat{\sigma}) = [lam \mapsto \hat{\rho}]$$

A starting state $\hat{\xi}_0 = ([e \mapsto \varnothing],\ \bot)$ for e.

A.6 A lattice of analyses

Greatest-lower-bound for environment-widened analyses may defined in the natural way:

$$\hat{\xi}_1 \sqcap \hat{\xi}_2 = (\hat{r}_1, \hat{\sigma}_1) \sqcap (\hat{r}_2, \hat{\sigma}_2) = (\hat{r}_1 \sqcap \hat{r}_2, \hat{\sigma}_1 \sqcap \hat{\sigma}_2)$$
$$\hat{r}_1 \sqcap \hat{r}_2 = \lambda e.\, \hat{r}_1(e) \sqcap \hat{r}_2(e)$$
$$\hat{d}_1 \sqcap \hat{d}_2 = \lambda lam.\, \hat{d}_1(lam) \sqcap \hat{d}_2(lam)$$
$$\hat{\rho}_1 \sqcap \hat{\rho}_2 = \lambda \hat{a}.\, \hat{\rho}_1(\hat{a}) \sqcap \hat{\rho}_2(\hat{a})$$
$$\hat{\sigma}_1 \sqcap \hat{\sigma}_2 = \lambda x.\, \hat{\sigma}_1(x) \sqcap \hat{\sigma}_2(x)$$

Join is analogous (N.B. The join of two environments adds a point where a point exists in either function).

We clarify Theorem 1 in greater detail by showing specifically how

$$CFA_{\nabla_\rho}(\widehat{alloc}_1 \sqcap \widehat{alloc}_2) = CFA_{\nabla_\rho}(\widehat{alloc}_1) \sqcap CFA_{\nabla_\rho}(\widehat{alloc}_2)\,,$$

and by explaining how a construction of join and meet is futher complicated for CFA and CFA_∇.

Proof (Sketch). This falls out from our definitions. We observe that $\hat{\xi}_0 = \hat{\xi}_0 \sqcap \hat{\xi}_0$ is trivially true and, referring to §A.5, show the inductive step:

$$\begin{aligned}&\hat{\xi} = \hat{\xi}_1 \sqcap \hat{\xi}_2 \implies \\ &\quad CFA(alloc_1 \sqcap alloc_2)(\hat{\xi}) = CFA(alloc_1)(\hat{\xi}_1) \sqcap CFA(alloc_2)(\hat{\xi}_2)\end{aligned}$$

This inductive step may be decomposed into a number of simpler inductive steps. For example, transitioning $\hat{r}$ for immediate applications of a lambda:

$$\begin{aligned}R_{lam}(\widehat{alloc})(\hat{r}) = \bigsqcup\{[e_\lambda \mapsto \hat{\rho}[x_i \mapsto \widehat{alloc}(x_i, \hat{\varsigma})]] \\ : (((\lambda\ (x_1\ \ldots)\ e_\lambda)\ ae_1\ \ldots), \hat{\rho}) \in \hat{r}\}\end{aligned}$$

And the inductive step:

$$\begin{aligned}&\hat{r} = \hat{r}_1 \sqcap \hat{r}_2 \ \wedge\ \hat{\sigma} = \hat{\sigma}_1 \sqcap \hat{\sigma}_2 \implies \\ &\quad R_{lam}(\widehat{alloc}_1 \sqcap \widehat{alloc}_2)(\hat{r}) = R_{lam}(\widehat{alloc}_1)(\hat{r}_1) \sqcap R_{lam}(\widehat{alloc}_2)(\hat{r}_2)\end{aligned}$$

Another component transitions applications of variable references:

$$\begin{aligned}R_{ref}(\widehat{alloc})(\hat{r}, \hat{\sigma}) = \bigsqcup\{[e_\lambda \mapsto \hat{\rho}_\lambda[x_i \mapsto \widehat{alloc}(x_i, \hat{\varsigma})]] \\ :((x_f\ ae_1\ \ldots),\ \hat{\rho}) \in \hat{r} \\ \ ((\lambda\ (x_1\ \ldots)\ e_\lambda),\ \hat{\rho}_\lambda) \in \hat{\sigma}(\hat{\rho}(x_f))\}\end{aligned}$$

Another transitions $\hat{\sigma}$ for the i^{th} argument of an application of a variable reference where this argument is a λ-abstraction.

$$\begin{aligned}S_{ref/lam}(\widehat{alloc})(\hat{r}) = \bigsqcup\{[\widehat{alloc}(x_i, \hat{\varsigma}) \mapsto [lam_i \mapsto \hat{\rho}]] \\ :((x_f\ \ldots\ ae_{i-1}\ lam_i\ \ldots),\ \hat{\rho}) \in \hat{r} \\ \ ((\lambda\ (x_1\ \ldots)\ e_\lambda),\ \hat{\rho}_\lambda) \in \hat{\sigma}(\hat{\rho}(x_f))\}\end{aligned}$$

Along with 3 additional cases for transitioning the store. □

The difficulty in producing such a straightforward, if verbose, proof of the same property for CFA and CFA_∇ is that no suitable definition of meet may be expressed for our simple definitions in §A.4, though it exists. When considering the difference between the above proof and one for CFA_∇, notice that it preserves

multiple possible pairs of a specific epression e with various environments $\hat{\rho}$. What might a suitable definition for $\hat{r}_1 \sqcap \hat{r}_2$ look like then?

A reasonable first try could be

$$\hat{r}_1 \sqcap \hat{r}_2 = \{(e,\ \hat{\rho}_1 \sqcap \hat{\rho}_2) \ :\ (e,\ \hat{\rho}_1) \in \hat{r}_1 \text{ and } (e,\ \hat{\rho}_2) \in \hat{r}_2\}\,,$$

producing all combinations. All reachable concrete states must abstract to one of the configurations in $\hat{r}_1$ and also to one of those in $\hat{r}_2$, so this seems like *a definition* for meet. It is not the one which corresponds to the meet of two allocators however, as this introduces additional structure in the lattice which allocators do not, breaking our precise embedding. The definition matching that for allocators more precisely matches up the $(e,\ \hat{\rho}_1)$ which corresponds to a specific $(e,\ \hat{\rho}_2)$, partitioning the meet over environments. Much like being unable to conveniently express an abstraction for 3-CFA given our concrete encoding for addresses as a variable and a natural number in §A.1, we do not have the information encoded to conveniently express a closed-form definition for the meet of CFA or CFA_∇, though it exists. A rigorous proof for this will need to track such meta-information, or show that a sound definition of meet is always induced by the definition for allocators.

A.7 Bonus example: type-sensitivity

Smaragdakis et al. define an approximation of their k-full-object-sensitivity which replaces allocation-points with class-names corresponding to each [29]. Two allocation-points within the same class definition become indistinguishable in this style. In our simple language this would approximate allocation-points by the body of the lambda containing them.

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

Closures each store the current running allocation-history.

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

Entering a lambdas body (our our case, call-sites) extends this current history. At each application, we begin using the individual history of the invoked closure.

$$\hat{\mathcal{M}}(e^l,\ \hat{\rho},\ \hat{\sigma},\ \hat{m}) = (l\ l_1\ \ldots\ l_{k-1})$$
$$\hat{\mathcal{M}}((lam,\ \hat{\rho},\ \hat{m}_\lambda),\ (\ldots),\ \hat{\sigma},\ \hat{m}) = \hat{m}_\lambda$$

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

$$\widehat{alloc}_{type}(x,\ ((lam,\ \hat{\rho},\ \hat{m}_\lambda),\ (\ldots),\ \hat{\sigma},\ \hat{m})) = (x, \hat{m}_\lambda)$$

To extend this style for languages with user-definable classes, objects would be extended with a history of class-names or labels unique to these definitions, just as we've done for closures.

A.8 Bonus example: path-sensitivity

Path-sensitivity is a broad range of different styles of polyvariance which encode which branch was taken through a conditional. An analysis of the expression $(\texttt{if}\ (>\ a\ 5)\ e_1\ e_2)$ could thereby remember along the e_1 path that $a > 5$ and along the e_2 branch that $a \leq 5$. To demonstrate a tuning for a style of path-sensitivity, we must assume an extension to our language which includes at least conditionals and boolean values.

$$\begin{array}{rl} e \in \mathsf{E} ::= & (\text{if } x\ e\ e) \\ | & \cdots \end{array} \qquad \begin{array}{c} \hat{d} \in \hat{D} = \mathcal{P}(\widehat{Value}) \\ \hat{v} \in \widehat{Value} = \widehat{Clo} + \{\texttt{True}, \texttt{False}\} \end{array}$$

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

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

$$\begin{array}{rl} \text{where} & v_1 = \mathcal{A}(x, \varsigma) \\ & m' = \mathcal{M}(\varsigma) \end{array}$$

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

$$\begin{array}{rl} \text{where} & v_1 = \mathcal{A}(x, \varsigma) \\ & m' = \mathcal{M}(\varsigma) \end{array}$$

The essential observation is simply that:

```
(if x e1 e2)        is equivalent to        (if x
                                              (let ([x x]) e1)
                                              (let ([x x]) e2))
```

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

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

$$\begin{array}{rl} \text{where} & \hat{d}_1 = \hat{\mathcal{A}}(x, \hat{\varsigma}) \\ & \hat{m}' = \hat{\mathcal{M}}(\hat{\varsigma}) \end{array}$$

$$\frac{\texttt{False} \in \hat{\mathcal{A}}(x, \hat{\varsigma})}{\underbrace{(\texttt{(if } x\ e_1\ e_2\texttt{)},\ \hat{\rho},\ \hat{\sigma},\ \hat{m})}_{\hat{\varsigma}} \rightsquigarrow ((\texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e_2\texttt{)},\ \hat{\rho}),\ (\{\hat{d}_1\}),\ \hat{\sigma},\ \hat{m}')}$$

$$\text{where} \qquad \hat{d}_1 = \hat{\mathcal{A}}(x, \hat{\varsigma})$$
$$\hat{m}' = \hat{\mathcal{M}}(\hat{\varsigma})$$

We may then track a running history of branches:

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

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

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

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