A Survey of Polyvariance in Abstract Interpretations

Thomas Gilray and Matthew Might

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

Abstract. Abstract interpretation is an efficient means for approximating program behaviors before run-time. It can be used as the basis for a number of different useful techniques in static analysis more broadly, and can thus in-turn be used to prove properties needed for security or optimization. Polyvariance represents a way of obtaining higher precision in an abstract interpretation by producing multiple abstract states for each function or lexical point of interest in the program. This paper explores the role of polyvariance in these analyses and how it is manifested, unifying the disparate presentations in the literature.

1 Abstract Interpretation

An abstract interpretation is a non-deterministic interpretation of a program that determines abstract flow-sets, each representing all possible values a given expression could refer to during any particular concrete execution. The result is a finite abstract state-space which conservatively approximates a usually infinite number of different concrete state-spaces.

All valid paths in the program are guaranteed to be represented in a sound analysis. Above and beyond these genuine executions, imprecision is manifested as spurious traces which are indicated by the analysis but which do not exist in any concrete execution.

1.1 CPS λ-Calculus

For our survey of polyvariance, we will be using a simple language with familiar abstract semantics at each step to stay consistent. Call-sites are marked with a unique label which refers to its containing lambda. Consider the CPS λ-calculus:

$$\begin{aligned} call \in \mathsf{Call} &::= (ae\ ae\ \ldots)^l \mid (\text{halt}) \\ ae \in \mathsf{AE} &::= x \mid lam \\ lam \in \mathsf{Lam} &::= (\lambda\ (x\ \ldots)\ call) \\ x \in \mathsf{Var} &::= \textit{set of program variables} \\ l \in \mathsf{Label} &::= \textit{set of unique labels} \end{aligned}$$

The grammar structurally distinguishes between atomic expressions and call-sites to permit only calls in tail position. This constrains the language to a

J. McCarthy (Ed.): TFP 2013, LNCS 8322, pp. 134–148, 2014.

continuation-passing-style (CPS) form. Abstract interpretation can be implemented for any language so long as we have a concrete (in our case, operational) semantics to abstract. CPS is used here (as it was in its original formulation) purely for the purposes of simplifying our discussion. We can compactly represent its semantics using a CES-style machine:

$$\begin{aligned}
\varsigma \in State &= \mathsf{Call} \times Env \times Store \times Time \\
\rho \in Env &= \mathsf{Var} \rightharpoonup Addr \\
\sigma \in Store &= Addr \rightharpoonup Value \\
t \in Time &= Label^* \\
a \in Addr &= \mathsf{Var} \times Time \\
v \in Value &= \mathsf{Lam} \times Env
\end{aligned}$$

and a single small-step transition:

$$\frac{\langle (\lambda\ (x_1\ \ldots\ x_j)\ call),\ \rho' \rangle\ =\ \mathcal{A}(ae_f,\ \rho,\ \sigma)}{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \rho,\ \sigma,\ t)\ \Rightarrow\ (call,\ \rho'',\ \sigma',\ t')}$$

$$\begin{aligned}
\text{where}\quad & \rho'' = \rho'[x_i \mapsto a_i] \\
& \sigma' = \sigma[a_i \mapsto \mathcal{A}(ae_i,\ \rho,\ \sigma)] \\
& a_i = (x_i,\ t') \\
& t' = l : t
\end{aligned}$$

where $\mathcal{A}$ is a concrete atomic-expression evaluator:

$$\begin{aligned}
\mathcal{A}(x,\ \rho,\ \sigma)\ &= \sigma(\rho(x)) \\
\mathcal{A}(lam,\ \rho,\ \sigma)\ &= \langle lam,\ \rho \rangle
\end{aligned}$$

Each state (machine configuration) contains a call-site, a binding environment, a value-store, and a timestamp. Each state transitions to a new state when a function can be invoked at the current call-site, or fails to transition and terminates when a (halt) is reached. The atomic-expression in call-position ae_f is evaluated to a closure and evaluation transitions to its body, another call-site. The closure's binding environment is augmented with addresses for each function-argument, and the store maps each of these to the value being bound. Each address is guaranteed to be unique because it is being paired with the new timestamp t'. t' is constructed by prefixing the current timestamp with a label for the current call-site. Because this call-history increases in length with each transition, no two values will share a binding.

1.2 0-CFA

0-CFA is the monovariant form of the k-CFA algorithm as presented in Shivers' seminal paper [25] [16]. We use an abstract version of our concrete semantics to

compute a conservative approximation of program behavior. In order to make this state-space finite, we need only to bound the size of our timestamp or call-history. k-CFA uses a k-length approximation of call-history, and 0-CFA merges all histories together.

As a repercussion of bounding $\widehat{Time}$, multiple values will now share a single address. Our abstract store maps addresses to flow-sets: sets of abstract values. All possible values for a particular variable now share the same address:

$$\begin{aligned}
\hat{\varsigma} \in \widehat{State} &= \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \times \widehat{Time} \\
\hat{\rho} \in \widehat{Env} &= \mathsf{Var} \rightharpoonup \widehat{Addr} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \rightharpoonup \mathcal{P}(\widehat{Value}) \\
\hat{t} \in \widehat{Time} &= \mathsf{Label}^0 \\
\hat{a} \in \widehat{Addr} &= \mathsf{Var} \times \widehat{Time} \\
\hat{v} \in \widehat{Value} &= \mathsf{Lam} \times \widehat{Env}
\end{aligned}$$

The abstract transition function is non-deterministic, as multiple closures can be referenced by a single variable:

$$\frac{\langle (\lambda\ (x_1\ \ldots\ x_j)\ call),\ \hat{\rho}' \rangle\ \in\ \hat{\mathcal{A}}(ae_f,\ \hat{\rho},\ \hat{\sigma})}{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \hat{\rho},\ \hat{\sigma},\ ())\ \approx\!\!>\ (call,\ \hat{\rho}'',\ \hat{\sigma}',\ ())}$$

$$\begin{aligned}
\text{where}\quad \hat{\rho}'' &= \hat{\rho}'[x_i \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{\rho},\ \hat{\sigma})] \\
\hat{a}_i &= (x_i,\ ())
\end{aligned}$$

The abstract atomic-exppression evaluator returns flow-sets:

$$\begin{aligned}
\hat{\mathcal{A}}(x,\ \hat{\rho},\ \hat{\sigma}) &= \hat{\sigma}(\hat{\rho}(x)) \\
\hat{\mathcal{A}}(lam,\ \hat{\rho},\ \hat{\sigma}) &= \{\langle lam,\ \hat{\rho} \rangle\}
\end{aligned}$$

When discarding typographical differences, the two semantics are almost identical. There are essentially only two fundamental changes we've made to achieve a finite approximation: we use a finite set of abstract addresses to bound the size of our store, and introduce merging between values at each address. If we were including other basic types, we could also replace them with a finite abstraction. An unbounded set of numbers might become just $\{num\}$ to differentiate from other basic types, or perhaps elaborated slightly to $\{+, 0, -\}$ in order to perform a sign analysis.

In our case, the only types involved are closures, which thanks to our abstraction for addresses, are now drawn from a finite set. These however, are now being merged together at bindings in our abstract store. Where before we indicated a strong-update of our concrete store, we now use function-join to indicate merging sets of values together via set-union. In this way, all values which have have

ever been bound to an address are kept. In 0-CFA there is a single address for each program-variable. If some argument z is bound to 3 different closures in our analysis, all 3 need to be represented by the same address z upon completion [29].

1.3 Soundness

Soundness of an abstract interpretation entails showing that all possible concrete executions are represented by the final analysis in general, for all inputs. Its embarrassing imprecision notwithstanding, $\lambda x.\widehat{Value}$ is an example of a trivially sound store because it does indeed represent all possible flows in any concrete execution of any program.

Showing that a more precise analysis is sound in general involves introducing a bit more machinery we won't bother with fully, and so we'll not attempt to do more than give a very rough sketch of the proof here. A proof of soundness relies on defining the relationship between the concrete and abstract domains. This relationship is a pair of functions for abstraction and concretization known as a Galois Connection. Previous work has shown the use of this model in both proving an existing analysis sound, and in producing analyses which are correct by construction. Methods have been developed for automatically constructing abstract approximations of concrete machines through the composition of these Galois Connections [29] [15] [12].

To specify the correspondence between our abstract semantics and our concrete semantics, we would need to provide at least an abstraction function α which maps concrete states to their most precise abstract representative:

$$\alpha\colon State \to \widehat{State}$$

With this specification we can prove a statement *for each concrete transition* $\varsigma \Rightarrow \varsigma'$*, there exists an abstract transition* $\hat{\varsigma} \approx\!\!> \hat{\varsigma}'$ *such that* $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$ *and* $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$ which shows that simulation is preserved across transition. The soundness proof for k-CFA has been published for both a denotational [25] and an operational [16] style of semantics.

1.4 Complexity

Termination is guaranteed because the search is being performed over a finite state-space.

0-CFA is known specifically to be of worst-case cubic complexity. To determine whether or not an abstract closure flows to a variable, requires examining at most each call site in the program $O(n)$. There are then at most $O(n) * O(n)$ of these possible flows because the number of variables is bounded by the size of the program, as is the number of lambdas [16]. The number of abstract closures in the monovariant analysis is the same as the number of lambdas since each abstract binding environment is fixed by the free variables in its function which can be determined lexically.

VanHorn and Mairson reduce the circuit value problem to an instance of the 0-CFA control flow problem, proving it to be PTIME-hard [27].

2 Polyvariance

In 0-CFA, each syntactic callsite is represented by a single abstract state. Polyvariance, in general terms, is the degree to which an analysis breaks up these syntactic points in the program and represents them with multiple differentiated abstract states.

2.1 k-CFA

k-CFA is the broader heirarchy of algorithms to which 0-CFA belongs. All forms of this algorithm where $k \geq 1$ represent increasingly polyvariant analyses. k-CFA differentiates states with the addition of an abstract history, or calling-context, referred to in its original presentation as an "abstract contour" [25].

The semantics below introduce a k-length calling-context $\hat{t}$ at each state which serves to differentiate like variables with unlike calling histories. Each calling-context is a tuple of call-site labels which represents the abstract history of calls that lead to a given state. The state's successors then get a calling-context which has lost its oldest callsite, and has been appended with the label for the most recent callsite. This new history is then included in the abstract addresses for these new states, differentiating their flow-sets and giving our binding environment a purpose for the first time.

$$\begin{aligned}
\hat{\varsigma} \in \widehat{State} &= \mathsf{Call} \times \widehat{Env} \times \widehat{Store} \times \widehat{Time} \\
\hat{\rho} \in \widehat{Env} &= \mathsf{Var} \rightharpoonup \widehat{Addr} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \rightharpoonup \mathcal{P}(\widehat{Value}) \\
\hat{a} \in \widehat{Addr} &= \mathsf{Var} \times \widehat{Time} \\
\hat{v} \in \widehat{Value} &= \mathsf{Lam} \times \widehat{Env} \\
\hat{t} \in \widehat{Time} &= \mathsf{Label}^k
\end{aligned}$$

$$\frac{\langle (\lambda\ (x_1\ \ldots\ x_j)\ call), \hat{\rho}' \rangle\ \in\ \hat{\mathcal{A}}(ae_f,\ \hat{\rho},\ \hat{\sigma}) \qquad \hat{t} = (l_1\ \ldots\ l_k)}{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \hat{\rho},\ \hat{\sigma},\ \hat{t})\ \approx\!\!>\ (call,\ \hat{\rho}'',\ \hat{\sigma}',\ \hat{t}')}$$

$$\begin{aligned}
\text{where}\quad \hat{\rho}'' &= \hat{\rho}'[x_i \mapsto \hat{a}_i] \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{\rho},\ \hat{\sigma})] \\
\hat{a}_i &= (x_i,\ \hat{t}') \\
\hat{t}' &= (l\ l_1\ \ldots\ l_{k-1})
\end{aligned}$$

A state $(call^{l_9},\ \hat{\rho},\ \hat{\sigma},\ (l_2\ l_5\ l_6))$ would mean that l_9 could be reached by a call from l_2, when reached after a call from l_5 and so-forth. A calling-context like

this if found in an address $(x,\ (l_2\ l_5\ l_6))$ would indicate that the values stored at this address were bound to x following the above history. Values in k-CFA are only merged once the fixed amount of call-history has been exceeded.

Consider an example where there are two calls of indirection in front of a function:

$$(\lambda x.\ (\lambda y.\ (\lambda z.\ \ldots)\ y)\ x)$$

Here, if x is bound to two different values in a 2-CFA analysis, by the time they reach z, the original context for the call to λx will have been lost and the values will be merged. If multiple values reach a recursive function, no matter how long a context is used, the values will eventually merge assuming the analysis cannot determine a bound on the calling depth before the context runs out. Using sufficiently precise abstract values to make this possible in the general case would tend to make the analysis impractical to compute.

2.2 Exponential Complexity for $k \geq 1$

The use of these call-string histories pays dividends where unlike call-sites provide a lambda with unlike abstract values. Where the history used is sufficient to capture these differences, they will be kept apart in the store, avoiding the usual merging and loss of precision. The major downside of k-CFA for $k \geq 1$ is that its precision against run-time trade-off comes at too great a price: polyvariant k-CFA is intractible for real world inputs.

Though long suspected, the proof that k-CFA is EXPTIME-complete came only recently in another work by Van Horn and Mairson [27].

3 Object Sensitivity

Object Sensitivity is an alternative notion of context for object-oriented languages which can be used in place of call-string histories or in conjunction with them [21]. There are various presentations of this strategy with subtle differences. The flavor which best fulfills the original intentions of the technique, and which has appeared most effective in practice is k-full-object-sensitivity by Smaragdakis, Bravenboer, and Lhotak [26]. This method differentiates argument-bindings by the allocation-history of a member-function's receiving object. This requires syntactic allocation-points to be stored inside the abstract representation of an object upon creation, so they can be retrieved later when one of its methods is invoked. When $k = 0$, object-sensitivity is equivalent to 0-CFA.

Consider a 1-full-object-sensitive analysis of Java. The abstract value for an object will store its allocation-point internally and when a method is invoked on the object, its bindings are made specific to this saved context. Take for example:

$$\mathsf{Object\ obj\ =\ new\ Object}();^{l_3}$$

The abstract value which flows into obj contains an allocation-history (l_3). When a method obj.m(...) is invoked, its bindings are unique to this program-point.

To extend this strategy to deeper levels of context sensitivity, we include allocation-history from the object which performs the allocation. If we instead wish to perform a 3-full-object-sensitive analysis on the same program, our addtional context is drawn from the variable *this* at the allocation-site. For example, if *this* contains an allocation-history $(l_8\ l_4\ l_9)$, the variable obj is represented by an object with a timestamp $(l_3\ l_8\ l_4)$.

3.1 Closure Sensitivity

We cannot easily modify our previous analysis to faithfully represent true object-sensitivity because the CPS λ-calculus does not include objects or classes. Instead we present a purely functional analog of this technique we call closure-sensitivity. Just as object instances are the building blocks of object-oriented programs, closures are the building blocks of functional programs. Objects can be implemented as a closure which accepts an additional parameter for selecting the method to invoke. Likewise, flat-closures can be implemented as an object with a single method. The allocation-point of a function is thus the syntactic position of the lambda, its point of closure-creation.

With this in view, we can produce a context-sensitve analysis of our language where abstract closures directly contain their allocation history.

$$\hat{v} \in \widehat{Value} = \ \mathsf{Lam} \times \widehat{Env} \times \widehat{Time}$$

Instead of appending a label for the current call-site to our timestamp, an abstract transition simply pulls the allocation-context out of our closure and uses this for new bindings.

$$\frac{\langle (\lambda\ (x_1\ \ldots\ x_j)\ call),\ \hat{\rho}',\ \hat{t}' \rangle\ \in\ \hat{\mathcal{A}}(ae_f,\ \hat{\rho},\ \hat{\sigma},\ \hat{t})}{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \hat{\rho},\ \hat{\sigma},\ \hat{t})\ \approx\!\!>\ (call,\ \hat{\rho}'',\ \hat{\sigma}',\ \hat{t}')}$$

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

When a lambda is atomically evaluated, this allocation-point is combined with the current context and stored inside the abstract closure. This requires a slight modification so the current context $\hat{t}$ is available to the atomic-expression evaluator:

$$\begin{aligned} & \hat{\mathcal{A}}(x,\ \hat{\rho},\ \hat{\sigma},\ \hat{t}) = \hat{\sigma}(\hat{\rho}(x)) \\ & \hat{\mathcal{A}}(lam^l,\ \hat{\rho},\ \hat{\sigma},\ (l_1\ \ldots\ l_k)) = \{\langle lam,\ \hat{\rho},\ (l\ l_1\ \ldots\ l_{k-1}) \rangle\} \end{aligned}$$

This analysis has the same fundamental complexity as k-CFA, but where call-sensitivity causes merging, closure-sensitivity might not and vice versa. As the basic technique has proved more effective than k-CFA in practice for languages like Java [26], our analog may have something to offer in the functional realm.

4 The Cartesian Product Algorithm

The Cartesian Product Algorithm (CPA) was originally introduced as an enhancement to a type inference algorithm which itself can be viewed as a specialization of the abstract interpretation concept: one where dynamic program types are used as constituents of the abstract value domain. We will present the source of imprecision that the original formulation attempts to address, generalize the solution as a form of polyvariance in abstract interpretations (as was suggested in publications which followed), and discuss CPA's complexity and precision relative to k-CFA.

4.1 The Problem / Original Formulation

In an abstract interpretation using types for values, where polymorphism is non-existent each flow-set could contain a maximum of one value each, and the algorithm reduces to a straightforward type-inference. Therefore, the authors of CPA introduce it as an enhancement to a basic flow-set based type-inference algorithm where polymorphic functions introduce merging and thus spurious concrete variants. They turn a single polymorphic call in the analysis into multiple monomorphic calls, preserving the precise values across function calls, and their inter-argument relationships.

The basic algorithm that CPA enhances works similarly to an abstract interpretation over types. It also assigns a flow-set of dynamic types for each variable in the program, but it then establishes constraints based on the program text, and 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 Palsberg and Schwartzbach – a kind of type-inference analog to call-string histories in k-CFA, where the use of p parallels that of k. This is shown to be insufficient however, as the authors of CPA give a case of merging which cannot be overcome by any sized p. Their motivating example is the polymorphic *max* function:

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

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

The solution that CPA proposes is to replace flow-sets of per-argument types, with flow-sets of per-function tuples of types. In such an analysis, the function *max* itself would be typed $\{(int, int), (string, string)\}$ preserving inter-argument patterns and eliminating spurious concrete calls like $(int, string)$ [1].

4.2 Abstract Contour Formulation

In essence, this change makes flow-sets for each argument specific to the entire tuple of types received in a call. This suggests an abstract contour representation which pairs variables with tuples of abstract values in the store, instead of pairing them with call histories as in k-CFA [17].

$$\widehat{Store} = \widehat{Addr} \rightharpoonup \widehat{Value}$$
$$\widehat{Time} = \widehat{Value}^*$$

This would seem to maintain perfect precision; exact values would be known for any given address. The problem with this approach is that it introduces recursion into our state-space making it again unbounded. Closures contain environments containing contours made of closures. Our analysis again becomes a concrete interpreter using arbitrarily precise values to differentiate one another in the store.

To faithfully extend this algorithm to a higher-order language, in the spirit of its original presentation, we reduce abstract values to their types. An abstract value like *string* could potentially remain as it is, but closures must be limited to a finite set of types. We've chosen to reduce them to only their syntactic lambda, merely dropping environments, on the assumption that this point in the program is associated with a single type signature – whether it is known pre-analysis or not.

$$\widehat{Store} = \widehat{Addr} \rightharpoonup \mathcal{P}(\widehat{Value})$$
$$\widehat{Time} = \mathcal{P}(\widehat{Type})^*$$
$$\widehat{Type} = \mathsf{Lam}$$

A helper function can be defined which performs this reduction:

$$\hat{\mathcal{T}} : \mathcal{P}(\widehat{Value}) \to \mathcal{P}(\widehat{Type})$$

At each call, a new contour is formed by reducing each of the flow-sets of the atomically-evaluated function arguments:

$$\frac{\langle(\lambda\ (x_1\ \ldots\ x_j)\ call), \hat{\rho}'\rangle \in \hat{\mathcal{A}}(ae_f,\ \hat{\rho},\ \hat{\sigma})}{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \hat{\rho},\ \hat{\sigma},\ \hat{t})\ \approx\!\!>\ (call,\ \hat{\rho}'',\ \hat{\sigma}',\ \hat{t}')}$$

$$\begin{aligned}
\text{where}\quad & \hat{\rho}'' = \hat{\rho}'[x_i \mapsto \hat{a}_i] \\
& \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{\rho},\ \hat{\sigma})] \\
& \hat{a}_i = (x_i,\ \hat{t}') \\
& \hat{t}' = (\hat{\mathcal{T}}(\hat{\mathcal{A}}(ae_1,\ \hat{\rho},\ \hat{\sigma}))\ \ldots\ \hat{\mathcal{T}}(\hat{\mathcal{A}}(ae_j,\ \hat{\rho},\ \hat{\sigma})))
\end{aligned}$$

The semantics for a CPA-like abstract interpretation are fundamentally that of k-CFA with the exception that our abstract contours in $\widehat{Time}$ are now tuples of

abstract values. This preserves the exact flows from function to function and the only merging now possible is exists in the predetermined merging inherent to our abstract values. Essentially, each abstract address is already specific to the exact abstract value it points to in the store, thus flows are no longer sets, and non-determinism can no longer occur at a call-site. Non-determinism would in practice be reintroduced by the addition of practical language constructs such as primitive operations on basic values, conditionals/if-statements, etc. Each of these would need abstract transition rules which produce multiple monomorphic abstract calls as opposed to making a single call which sends a set of abstract values.

4.3 Precision and Complexity

It is straightforward to see intuitively that CPA is more precise than k-CFA, as is discussed in the original publication. For any pre-determined value of k, a program can be constructed which nests calls passed this call depth and causes merging. Any such merging, even when completely precise at the level of a particular argument, can produce spurious inter-argument patterns. CPA on the other hand differentiates functions directly based on the full tuple of arguments they receive and obtains perfect precision for a given finite set of abstract values.

That no length call-history can match the precision of CPA has also been formally demonstrated on an object-oriented language [2]. It is important to note that k-CFA contains context information which CPA does not and which might be useful for its own sake. k-CFA may also be more general and amenable to infinitely wide value domains, while CPA may rely more directly on the finiteness of the abstract values used to ensure computability.

CPA is of-course, like k-CFA, of exponential complexity, and exceedingly impractical for use on sufficiently complex input programs. Somewhat ironically, 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 might require as few as one flow per-type; just as with k-CFA, except carries a vast improvement in precision. For a function where all combinations of arguments are possible, CPA requires each to be explicitly made, while k-CFA implies them for equal precision at far greater efficiency.

5 Practical Polyvariance

In contrast to CPA's attempt to improve on the precision of abstract call-string histories, attempts have been made to bring a degree of call-string history polyvariance to an analysis without incurring the full cost of 1-CFA.

5.1 Polymorphic Splitting

Polymorphic Splitting is a compromise between 0-CFA and k-CFA where the length of the contour used varies on a per-function basis. Lambdas which have

been let-bound are analyzed with a contour length 1-greater than that of their parent expression. In this way, let-bindings can be used as a heuristic for guiding the length of the contour used within a function during analysis. Because the number of let forms within-which an expression can be nested is bound by the program's size, the maximum length of k is likewise fixed.

In order to give a semantics for polymorphic splitting which will work on our simple language, make it quickly understandable to the reader, and comparable to the other analyses discussed here, we imagine our program has been CPS-converted from a direct-style language with a let-form, and we add a k annotation to call-sites which indicate their function's let-depth:

$$call \in CALL \ ::= \ (ae \ \ldots)^{l}_{k}$$

This annotation can then be used to direct the amount of polyvariance used in our abstract transition:

$$\frac{\langle(\lambda \ (x_1 \ \ldots \ x_j) \ call_{k'}), \hat{\rho}'\rangle \ \in \ \hat{\mathcal{A}}(ae_f, \ \hat{\rho}, \ \hat{\sigma}) \qquad \hat{t} = (l_1 \ \ldots \ l_k)}{((ae_f \ ae_1 \ \ldots \ ae_j)^{l}_{k''}, \ \hat{\rho}, \ \hat{\sigma}, \ \hat{t}) \ \approx\!\!> \ (call, \ \hat{\rho}'', \ \hat{\sigma}', \ \hat{t}')}$$

$$\begin{aligned}
\text{where} \quad & \hat{\rho}'' = \hat{\rho}'[x_i \mapsto \hat{a}_i] \\
& \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{\mathcal{A}}(ae_i, \ \hat{\rho}, \ \hat{\sigma})] \\
& \hat{a}_i = (x_i, \ \hat{t}') \\
& \hat{t}' = (\text{take} \ (l \ l_1 \ \ldots \ l_k) \ k') \\
& \hat{\mathcal{A}}(x, \ \hat{\rho}, \ \hat{\sigma}) = \hat{\sigma}(\hat{\rho}(x)) \\
& \hat{\mathcal{A}}(lam, \ \hat{\rho}, \ \hat{\sigma}) = \{\langle lam, \ \hat{\rho}\rangle\}
\end{aligned}$$

We have added a subscript k' to the call-site found in a closure for ae_f which determines the length of the contour we'll use for our new argument bindings. The function $(take \ lst \ n)$ removes all but the first n entries from lst.

This presentation of polymorphic splitting may perhaps introduce a confusion as to how we know that k' is not greater than $k+1$; that there is enough history for us to use at each transition. This concern is unlikely to arise from looking at the original semantics. We know a call into a let-bound function is unreachable above that let form's body, and since that let form's body shares the contour length of its parent expression, it can be at most one less than the contour length of the let-bound function.

The complexity of polymorphic spitting remains exponential, as it easily devolves into doing all the work of a k-CFA analysis in the worst-case, however it has been empirically shown to be practical for sizable benchmarks. The authors found its precision comparable to that of a 1-CFA, while its running times were closer to that of 0-CFA. That it even beat the running time for 0-CFA in some test-cases can be attributed to its higher precision culling spurious paths which would have otherwise been explored by the monovariant analysis [30].

5.2 Polynomial-Time 1-CFA

Polynomial-time 1-CFA differentiates each state with a single call history, as 1-CFA does, but only allows free variables in a closure's environment to remember this history for a single closure creation deep. Each time a function is called, its abstract contour is updated and all the flows for its free variables are propagated to the new history for that call. They then share a history with the latest arguments to be sent in all new closures created. An environment in this analysis boils down to the single abstract contour it maps all variables onto. We simplify this and pair lambdas directly with a single contour to form a closure:

$$\hat{\varsigma} \in \widehat{State} = \mathsf{CALL} \times \widehat{Store} \times \widehat{Time}$$
$$\hat{v} \in \widehat{Value} = \mathsf{LAM} \times \widehat{Time}$$
$$\hat{t} \in \widehat{Time} = \mathsf{Label}$$

$$\frac{\langle(\lambda\ (x_1\ \ldots\ x_j)\ call), \hat{t}'_b\rangle\ \in\ \hat{\mathcal{A}}(ae_f,\ \hat{t},\ \hat{\sigma})}{((ae_f\ ae_1\ \ldots\ ae_j)^l,\ \hat{\sigma},\ \hat{t})\ \twoheadrightarrow\ (call,\ \hat{\sigma}',\ \hat{t}')}$$

where
$$\hat{\sigma}' = \hat{\sigma} \sqcup [(x_i, \hat{t}') \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{t},\ \hat{\sigma})] \sqcup \bigsqcup\{[(y, \hat{t}') \mapsto \hat{\mathcal{A}}(y,\ \hat{t}_b,\ \hat{\sigma})] \mid y \in free(call)\}$$
$$\hat{t}' = l$$
$$\hat{\mathcal{A}}(x,\ \hat{t},\ \hat{\sigma}) = \hat{\sigma}((x, \hat{t}))$$
$$\hat{\mathcal{A}}(lam,\ \hat{t},\ \hat{\sigma}) = \{\langle lam,\ \hat{t}\rangle\}$$

Because the closure is updated at each call, the binding environment previously in the second position of our abstract state is redundant with the single call-history in the final position, so we omit it. Likewise, the creation of a new binding environment (previously called $\hat{\rho}''$) is no longer needed as it was in k-CFA since it would simply be set to $\lambda\_.\hat{t}'$ and so is subsumed here by $\hat{t}'$ itself. Our updated store is one joined with the bindings formed by the function call, along with bindings which propagate values for the free variables in the function to their new contour.

Polynomial-time 1-CFA improves on 0-CFA in many of the usual places. Function parameters given different values at different callsites are analyzed polyvariantly. Where it compromises as compared with full 1-CFA is in the addresses used for free variables. When a function is closed over its free variables, they are differentiated by the call-history of the containing lambda. Upon invocation however, these values are propagated to addresses using the most recent callsite. This means if we call a function $\lambda x.\lambda y.x$ more than once, we may obtain multiple different abstract closures, but if we invoke each of them at the same callsite l, all these variants of x will be merged together into an address $(x,\ l)$.

Polynomial-time 1-CFA has not yet been empirically investigated, but its complexity has an upper bound of $O(n^6)$ [8].

6 The Future

The potential for new explorations into this area looks bright. The recent paper *A posteriori soundness* by Might and Manolios [20] has provided an exceptionally general guarantee of soundness for abstract allocation functions which allows for nearly any form of merging or differentiation in the store which could be conceived. Even methods which tune a live analysis directly for precision are allowed for, so no fully pre-defined strategy would even be necessary.

6.1 A Posteriori Soundness

The usual process for demonstrating the soundness of an abstract interpretation is *a priori* in the sense that the concrete and abstract transition relations along with the abstraction map relating the two state-spaces have been defined in advance, and are then justified as sound before any analysis is produced. *A posteriori* soundness differs from this in that a portion of the justifying abstraction map cannot be known until after the analysis is run.

The *a posteriori* soundness proof relies on factoring apart the concrete semantics, abstract semantics, and their correspondence. A portion of the abstraction map α is isolated which represents the correspondence between concrete addresses and abstract addresses: α_L. A portion of the transition relation is also factored out which represents the process of producing bindings. The abstract transition relation can then be parameterized by an allocation-policy $\hat{\pi}$ which determines this process for a given abstract state. The crux of the argument is then that given a non-deterministic selection of $\hat{\pi}$, a justifying α_L can always be produced after the fact, which proves the prior selection sound – whatever it might have been. This means that so long as the remaining analysis follows a single liberal soundness condition: the choice of allocation policy $\hat{\pi}$ is entirely arbitrary as far as the correctness of the analysis is concerned [20].

6.2 Precision-Adaptive Analyses

The implication of this is that the allocation policy $\hat{\pi}$ of an abstract interpretation can be selected entirely with precision and complexity in view. A policy can even adapt to the source text itself to make these choices without soundness needing to be proven for each specific program. If soundness needed to be proved *a priori*, this would not be possible since the mechanics of the proof would rely upon aspects of specific programs which could not be known in advance. The work thus not only simplifies deciding that a new form of polyvariance would be sound, but makes it possible to produce polyvariant analyses which use different amounts of history for different functions, different kinds of history for different functions, and which make these decisions while the analysis is still live.

7 Conclusion

The concept of polyvariance in abstract interpretations covers a wide array of techniques which allows for an analysis to be tuned up or down along the

precision/complexity trade-off. Merging and differentiation of flow-sets in the store, beyond one address per variable, requires a value on which to base the differentiation: in the case of k-CFA this is Shivers' abstract contour. It has since been proved that any basis for differentiation which obeys a single liberal constraint will remain sound, and a number of specific variants on the traditional contour have already been discussed in the literature each offering a unique trade-off in precision.

References

1. Agesen, O.: The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In: Echtle, K., Powell, D.R., Hammer, D. (eds.) EDCC 1994. LNCS, vol. 852, pp. 2–26. Springer, Heidelberg (1994)
2. Besson, F.C.: beats ∞-CFA. Formal Techniques for Java-like Programs, p. 7 (July 2009)
3. Cousot, P.: The calculational design of a generic abstract interpreter. In: Broy, M., Steinbrüggen, R. (eds.) Calculational System Design. NATO ASI Series F, pp. 421–506 (1999)
4. Cousot, P.: Types as Abstract Interpretations. In: Symposium on Principals of Programming Languages, pp. 316–331 (1997)
5. Cousot, P., Cousot, R.: Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Symposium on Principals of Programming Languages, pp. 238–252 (1977)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Symposium on Principals of Programming Languages, pp. 269–282 (1979)
7. Felleisen, M., Findler, R., Flatt, M.: Semantics Engineering with PLT Redex (August 2009)
8. Jagganathan, S., Weeks, S.: A Unified Treatment of Flow Analysis in Higher-Order Languages. In: ACM Symposium on Principles of Programming Languages, pp. 393–407. ACM Press (January 1995)
9. Jones, N.D.: A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In: Symposium on Principles of Programming Languages, pp. 66–74 (1982)
10. Jones, N.D., Muchnick, S.: Flow analysis of lambda expressions (preliminary version). In: Even, S., Kariv, O. (eds.) ICALP 1981. LNCS, vol. 115, pp. 114–128. Springer, Heidelberg (1981)
11. Midtgaard, J.: Control-Flow Analysis of Functional Programs. ACM Computing Surveys 44 (June 2012)
12. Midtgaard, J., Jensen, T.: A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 347–362. Springer, Heidelberg (2008)
13. Midtgaard, J., Van Horn, D.: Subcubic Control Flow analysis Algorithms. Higher-Order and Symbolic Computation (May 2009)
14. Midtgaard, J., Jensen, T.: Control-ow analysis of function calls and returns by abstract interpretation. In: International Conference on Functional Programming (2009)
15. Might, M.: Abstract interpreters for free. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 407–421. Springer, Heidelberg (2010)

16. Might, M.: Environment Analysis of Higher-Order Languages. Ph.D. Dissertation. Georgia Institute of Technology (2007)
17. Might, M.: Logic-Flow Analysis of Higher-Order Programs. In: Principals of Programming Langauges, pp. 185–198 (January 2007)
18. Might, M., Shivers, O.: Environment analysis via ΔCFA. In: Symposium on the Principals of Programming Languages, pp. 127–140 (January 2006)
19. Might, M., Shivers, O.: Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In: International Conference on Functional Programming, pp. 13–25 (September 2006)
20. Might, M., Manolios, P.: *A posteriori* soundness for non-deterministic abstract interpretations. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 260–274. Springer, Heidelberg (2009)
21. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Transaction on Software Engineering and Methodology, 1–41 (2005)
22. Nielson, F., Nielson, H.R., Hankin, C.: Principals of Program Analysis. Springer (1999)
23. Palsberg, J., Pavlopoulou, C.: From Polyvariant Flow Information to Intersection and Union Types. In: Principals of Programming Languages, pp. 197–208 (1998)
24. Shivers, O.: Control-flow analysis in Scheme. In: Programming Language Design and Implementation,pp. 164–174 (June 1988)
25. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. PhD dissertation. School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, Technical Report CMUCS-91-145 (May 1991)
26. Smaragdakis, Y., Bravenboer, M., Lhotak, O.: Pick your contexts well: understanding object-sensitivity. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 17–30. ACM, New York (2011)
27. Van Horn, D., Mairson, G.H.: Deciding k-CFA is complete for EXPTIME. In: International Conference on Functional Programming, pp. 275–282 (September 2008)
28. Van Horn, D., Mairson, H.G.: Flow analysis, linearity, and PTIME. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 255–269. Springer, Heidelberg (2008)
29. Van Horn, D., Might, M.: Abstracting Abstract Machines. In: International Conference on Functional Programming 2010, Baltimore, Maryland, pp. 51–62 (September 2010)
30. Wright, A.K., Jagannathan, S.: Polymorphic splitting: An effective polyvariant flow analysis. ACM Transactions on Programming Languages and Systems, 166–207 (January 1998)