A Survey of Polyvariance in Control-Flow Analyses

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 Introduction

A control-flow analysis conservatively approximates the control-flow behavior of a program, which for higher-order languages, necessarily implies a sound supporting data-flow analysis. This paper considers these analyses from the perspective of abstract interpretation: a general method for producing sound conservative approximations of program semantics. Our process is to describe a simple language, give a concrete operational semantics for a CES-like machine, and then derive a sound abstract semantics which is made computable by bounding the size of the machine's configuration space. The result of the analysis 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 cannot exist in any concrete execution.

Polyvariant analyses are those which seek to improve precision by maintaining separate values for each context in which they may be found. Call-sensitive analyses for example, distinguish between values recieved from different call-sites. A call-sensitive analysis could determine that x in the following code is true in one context and false in another, while a monovariant analysis would only determine that it was boolean.

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

[⋆] Different parts of our effort on this work were partially supported by the DARPA programs APAC and CRASH.

This work surveys the role of polyvariance in such abstract interpretations, as well as the diversity of approaches taken in the literature. We discuss the fixed length call-string histories of Shivers' k-CFA, the variable length contours used in the polymorphic-splitting analysis of Wright and Jagannathan, and the restricted binding environments in the polynomial-time 1-CFA of Jagannathan and Weeks. We describe Agesen's Cartesian Product Algorithm and the potential pitfalls of applying it to a higher-order language. We also consider object-sensitivity and give a rendering of this technique for our language. [25] [30] [1] [8] [21]

We unify the presentation of these concepts by giving each as a small-step operational semantics for a simple language. Though originally formulated as a type-inference algorithm (in the case of CPA) or as a constraint semantics (in the case of polymorphic splitting), we show that each technique can be viewed as a different model of polyvariance within an abstract interpretation.

1.1 CPS λ-calculus

We use 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:

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

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

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

and a single small-step transition:

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

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

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

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

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 closure 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 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 guarenteed 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 need share a binding.

1.2 0-CFA

0-CFA is the monovariant form of the k-CFA algorithm as presented in Shivers' seminal paper [24] [16]. We use an abstracted 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:

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

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

The abstract atomic-exppression evaluator returns flow-sets:

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

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 would 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

An abstract interpretation is sound if all possible concrete executions will be represented by the final analysis. 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. [16]

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 hierachy 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 it's original presentation as an 'abstract contour'. [24] [25]

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

The state-space above introduces 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 lambda, and has been appended with the calling lambda. 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.

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

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

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.

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

Consider an example where there are two calls of indirection in front of a function. 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 for 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 [27].

3 The Cartesian Product Algorithm

The Cartesian Product Algorithm 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 the original formulation attempts to address, generalize the solution as a form of polyvariance in abstract interpretations (as it is suggested in publications which followed), and discuss CPA's complexity and precision relative to k-CFA.

3.1 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 a 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 off 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 $max(“abc”, “xyz”)$ 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 the 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]

3.2 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].

$$\begin{aligned}\widehat{Store} &= \widehat{Addr} \rightharpoonup \widehat{Value}\\ \widehat{Time} &= \widehat{Value}^*\end{aligned}$$

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 themselves 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.

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

A helper function can be defined which performs this reduction:

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

Merging is now possible between different binding environments. At each call, a new contour is formed by reducing each of the flow-sets of the atomically-evaluated function arguments:

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

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

3.3 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 calls 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-string history can match the precision of CPA was also formally demonstrated for an object-oriented language just recently [2]. It may be worth noting that k-CFA contains context information which CPA does not and which might be useful for its own sake.

CPA, like k-CFA, is 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; this is just as with k-CFA, except it 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.

The exponential complexity of CPA can be realized by examining the possible abstract contours. If the largest number of arguments any function takes is J, the number of contours the analysis could produce is $|\widehat{Types}|^J$. Since J is only bounded by the size of the program, CPA is exponential in the worst case. The program could be transformed soundly into a curried style where no function took more than a single argument. This could make the analysis efficient, but it would be effectively threading most arguments through binding environments and introducing merging which would lower precision and largely defeat the purpose of CPA.

4 Object Sensitivity

Object-sensitivity is an alternative to traditional call-site sensitivity for object oriented languages. When a method is invoked, the *allocation context* of its recieving object is used to differentiate new bindings. Like k-CFA, a k-full-object sensitive analysis is a heirarchy of analyses using allocation contexts of increasing length. A 1-full-object analysis uses the syntactic location of the object's allocation. A 2-full-object analysis uses the allocation point of the recieving object, along with the allocation-point of the object that created it (an allocation context of length 2). [21] [26]

Objects and closures are both fundamentally the pairing of code with a data environment over which the code operates. Either can be translated into the other. We apply the concept of object-sensitivity to the CPS λ-calculus by including an allocation context within closures:

$$\begin{aligned}\hat{v} \in \widehat{Value} &= \mathsf{Lam} \times \widehat{Env} \times \widehat{Time}\\ \hat{t} \in \widehat{Time} &= \mathsf{Label}^k\end{aligned}$$

Our transition then uses the contour of the closure invoked to differentiate new bindings, as it would a calling-context in k-CFA.

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

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

The atomic-expression evaluator is extended to accept the current label and the current contour as arguments. Then, upon closure creation, the current history is extended with the current allocation-point and placed in the new closure.

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

Object-sensitivity has been epirically evaluated in multiple publications recently, and has become a preferred method of polyvariance for object-oriented languages, but has not yet been studied for primarily functional languages.

5 Practical Call-Strings

In contrast to these attempts 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 history used varies on a per-function basis. Let-bindings are used as syntactic cues for determining which lambdas should be analyzed at increased precision. A let-bound lambda is given a timestamp one longer than the timestamp at the let-form. The overall length of countours in this analysis is bounded by the deepest nesting of let forms, and thus by the program's length.

To give a faithful rendering of this technique we introduce the ANF λ-calculus for this section alone.

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

Administrative Normal Form is an intermediate representation which supports two complications in addition to a call-site whose arguments can be atomically evaluated: An atomic expression to return, and a let-form. We model the execution of this language using a CESK*-like machine with nested continuations threaded through the store to achieve a finite configuration space. Closures are extended with an abstract contour.

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

There are three abstract transitions, one for each kind of expression. A let-form proceeds inside the bound expression and sets up a new continuation. Each continuation contains a variable to be let-bound, an environment to reinstate for this binding, an expression to return to, an address for this expression's continuation, and a timestamp to reinstate. The current continuation is placed in the store at the address used for the new continuation.

$$((\text{let } (x\ e_1)\ e_2)^l,\ \hat{\rho},\ \hat{\sigma},\ (x_\kappa,\ \hat{\rho}_\kappa,\ e_\kappa,\ a_\kappa,\ \hat{t}_\kappa),\ \hat{t}) \ \approx\!> \ (e_1,\ \hat{\rho},\ \hat{\sigma}',\ \hat{\kappa},\ \hat{t})$$

$$\begin{aligned}
\text{where}\quad \hat{\kappa} &= (x,\ \hat{\rho},\ e_2,\ x_\kappa,\ \hat{t}) \\
\hat{\sigma}' &= \hat{\sigma} \sqcup [x_\kappa \mapsto (x_\kappa,\ \hat{\rho}_\kappa,\ e_\kappa,\ a_\kappa,\ \hat{t}_\kappa)]
\end{aligned}$$

Call-sites operate as they do in k-CFA, except the timestamp used is taken from the closure to be invoked.

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

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

Values are returned according to the current continuation. A binding is made in the continuation's environment for the variable x_κ and control continues to the continuation expression e_κ. Nested continuations are selected non-deterministically from the address $\hat{a}_\kappa$ provided.

$$\frac{\hat{\kappa} \in\ \hat{\sigma}(\hat{a}_\kappa)}{(ae,\ \hat{\rho},\ \hat{\sigma},\ (x_\kappa,\ \hat{\rho}_\kappa,\ e_\kappa,\ \hat{a}_\kappa,\ \hat{t}_\kappa),\ \hat{t}) \ \approx\!> \ (e_\kappa,\ \hat{\rho}',\ \hat{\sigma}',\ \hat{\kappa},\ \hat{t}_\kappa)}$$

$$\text{where} \quad \begin{aligned} \hat{\rho}' &= \hat{\rho}_\kappa[x_\kappa \mapsto \hat{a}_x] \\ \hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}_x \mapsto \hat{\mathcal{A}}(ae,\ \hat{\rho},\ \hat{\sigma},\ \hat{t})] \\ \hat{a}_x &= (x_\kappa,\ \hat{t}_\kappa) \end{aligned}$$

From these transitions, the anaysis behaves as k-CFA would over an ANF language, with the exception that contours are stored within closures, each closure choosing it's own timestamp. The meaning of the algorithm is thus largely defined by atomic-expression evaluation, which determines how lambdas are evaluated to closures. When a lambda is encountered being called, returned, or lambda-bound, it is placed in a closure with the current timestamp and binding environment. If however, it occurs immediately inside a let-form, the current timestamp is extended with the syntactic label for the lambda.

$$\hat{\mathcal{A}}(lam^l,\ \hat{\rho},\ \hat{\sigma},\ \hat{t}) \ = \begin{cases} \{(lam,\ \hat{\rho},\ l : \hat{t})\} & \textit{if lam is let–bound} \\ \{(lam,\ \hat{\rho},\ \hat{t})\} & \textit{otherwise} \end{cases}$$

This alone only serves to make bindings unique to the syntactic lambda which creates them, and has no impact on precision. When a variable is referenced however, those bound by a let-form have their closures mutated during access. The timestamp $\hat{t}_\lambda$ in each closure is replaced with $\hat{t}_\lambda[l_\lambda/l]$, a version with all labels l_λ replaced with l. This effectively makes the contour used for let-bound lambdas specific to the exact point of access.

$$\hat{\mathcal{A}}(x^l,\ \hat{\rho},\ \hat{\sigma},\ \hat{t}) \ = \begin{cases} \{(e_\lambda^{l_\lambda}, \hat{\rho}_\lambda, \hat{t}_\lambda[l_\lambda/l]) & \\ \qquad \mid (e_\lambda^{l_\lambda}, \hat{\rho}_\lambda, \hat{t}_\lambda) \in \hat{\sigma}(\hat{\rho}(x))\} & \textit{if x is let–bound} \\ \hat{\sigma}(\hat{\rho}(x)) & \textit{otherwise} \end{cases}$$

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 it's precision comparable to that of a 1-CFA, while it's 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, it's 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:

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

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

$$\begin{aligned}\text{where}\quad \hat{\sigma}' &= \hat{\sigma} \sqcup [(x_i, l) \mapsto \hat{\mathcal{A}}(ae_i,\ \hat{t},\ \hat{\sigma})] \\ &\quad \sqcup \bigsqcup \{[(y, l) \mapsto \hat{\mathcal{A}}(y,\ \hat{t}_\lambda,\ \hat{\sigma})] \mid y \in free(call')\} \\ \hat{\mathcal{A}}(x,\ \hat{t},\ \hat{\sigma}) &= \hat{\sigma}((x, \hat{t})) \\ \hat{\mathcal{A}}(lam,\ \hat{t},\ \hat{\sigma}) &= \{(lam,\ \hat{t})\}\end{aligned}$$

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 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 in 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 a 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 In Summary

The concept of polyvariance in control-flow analyses 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 Proceedings of ECOOP 1995 (1995), pp. 226.
2. Besson, F. CPA beats ∞-CFA. Formal Techniques for Java-like Programs, July 2009.
3. Cousot P. The calculational design of a generic abstract interpreter. NATO ASI Series F. Broy, M. and Steinbrüggen, R. (eds.): *Calculational System Design*. 1999.
4. Cousot P. Types as Abstract Interpretations. Symposium on Principals of Programming Languages. 1997.
5. Cousot P. and Cousot R. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Symposium on Principals of Programming Languages. 1977.
6. Cousot P. and Cousot R. Systematic design of program analysis frameworks. Symposium on Principals of Programming Languages. 1979.
7. Felleisen, M., Findler R. and Flatt, M. Semantics Engineering with PLT Redex. August 2009.
8. Jagganathan, S, and Weeks, S. A Unified Treatment of Flow Analysis in Higher-Order Languages. ACM Symposium on Principles of Programming Languages, January 1995. ACM Press.
9. Jones, N.D. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. Symposium on Principles of Programming Languages. 1982.
10. Jones, N.D. and Muchnick, S. Flow analysis of lambda expressions (preliminary version). Proceedings of the 8th Colloquium on Automata Languages and Programming. 1981.
11. Midtgaard, J. Control-Flow Analysis of Functional Programs. ACM Computing Surveys, Vol. 44. June 2012.
12. Midtgaard, J. and Jensen, T. A Calculational Approach to Control-flow Analysis by Abstract Interpretation. SAS, volume 5079 of Lecture notes in Computer Science. 2008.
13. Midtgaard, J. and Van Horn, D. Subcubic Control Flow analysis Algorithms. Higher-Order and Symbolic Computation. May 2009.
14. Midtgaard, J. and Jensen, T. Control-ow analysis of function calls and returns by abstract interpretation. International Conference on Functional Programming. 2009.
15. Might, M. Abstract interpreters for free. Static Analysis Symposium. 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. Principals of Programming Langauges. January 2007.
18. Might, M. and Shivers, O. Environment analysis via ΔCFA. Symposium on the Principals of Programming Languages. January 2006.
19. Might, M. and Shivers, O. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. International Conference on Functional Programming. September 2006.
20. Might, M. and Manolios, P. A posteriori soundness for non-deterministic abstract interpretations. 10th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2009). Savannah, Georgia, USA. January, 2009. pages 260–274.

21. Milanova A., Rountev A., and Ryder B.G. Parameterized Object Sensitivity for Points-to Analysis for Java. ACM Transactions on Software Engineering and Methodology, Vol. 14, No. 1. January 2005.
22. Nielson, F., Nielson, H.R. and Hankin, C. Principals of Program Analysis. Springer-Verlang. 1999.
23. Palsberg, J. and Pavlopoulou, C. From Polyvariant Flow Information to Intersection and Union Types. Journal of Functional Programming. May 2001.
24. Shivers, O. Control-flow analysis in Scheme. Programming Language Design and Implementation. June 1988.
25. Shivers, O. Control-Flow Analysis of Higher-Order Languages. PhD dissertation. School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMUCS-91-145.
26. Smaragdakis, Y., Bravenboer M., and Lhotak, O. Pick your contexts well: understanding object-sensitivity. Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: ACM, 2011, pp. 1730.
27. Van Horn, D. and Mairson, G.H. Deciding k-CFA is complete for EXPTIME. International Conference on Functional Programming. September, 2008.
28. Van Horn, D. and Mairson, G.H. Flow Analysis, Linearity, and PTIME. Static Analysis Symposium 2008. pp. 255-269.
29. Van Horn, D. and Might, M. Abstracting Abstract Machines. International Conference on Functional Programming 2010. Baltimore, Maryland. September, 2010. pp. 51-62.
30. Wright, A. K. and Jagannathan, S. Polymorphic splitting: An effective polyvariant flow analysis. ACM Transactions on Programming Languages and Systems. January 1998, pages 166-207.