Polyvariant Flow Analysis with Constrained Types

Scott F. Smith and Tiejun Wang*

Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218, USA
{scott,wtj}@cs.jhu.edu

Abstract. The basic idea behind improving the quality of a monovariant control flow analysis such as 0CFA is the concept of *polyvariant* analyses such as Agesen's Cartesian Product Algorithm (CPA) and Shivers' nCFA. In this paper we develop a novel framework for polyvariant flow analysis based on Aiken-Wimmers constrained type theory. We develop instantiations of our framework to formalize various polyvariant algorithms, including nCFA and CPA. With our CPA formalization, we show the call-graph based termination condition for CPA will not always guarantee termination. We then develop a novel termination condition and prove it indeed leads to a terminating algorithm. Additionally, we show how data polymorphism can be modeled in the framework, by defining a simple extension to CPA that incorporates data polymorphism.

1 Introduction

The basic idea behind improving the precision of a simple control flow analysis such as 0CFA is the concept of *polyvariant* analysis, also known as *flow splitting*. For better analysis precision, the definition of a polymorphic function is re-analyzed multiple times with respect to different application contexts. The original polyvariant generalization of the monovariant 0CFA control flow algorithm is the nCFA algorithm, defined by Shivers [17]. This generalization however has been shown to be not so effective: the values of n needed to obtain more accurate analyses are usually beyond the realm of the easily computable, and even 1CFA can be quite slow to compute [19]. Better notions of polyvariant analysis have been developed. In particular, Agesen's CPA [1, 2] analyzes programs with parametric polymorphism in an efficient and adaptive manner.

In this paper we develop a general framework for polyvariant flow analysis with Aiken-Wimmers constrained types [3]. We represent each function definition with a polymorphic constrained type scheme of form $(\forall\ \bar{t}.\ t \rightarrow \tau \setminus C)$. The subtyping constraint set C bound in the type scheme captures the flow corresponding to the function body. Each re-analysis of the function is realized by a new instantiation of the type scheme.

There have recently been several frameworks developed for polyvariant flow analysis, in terms of union and intersection types [16], abstract interpretation [13], flow graphs [12], and more implementation-centric [10]. Our purpose in designing a new

* Partial funding provided by NSF grant CCR-9619843

framework is not primarily to give “yet another framework” for polyvariant flow analysis, but to develop a framework particularly useful for the development of new polyvariant analyses, and for improving on implementations of existing analyses. We will give an example of a new analysis developed within the framework, Data-Adaptive CPA, which extends CPA to incorporate data polymorphism. There also are implementation advantages obtained by basing analyses on polymorphic constrained types. Compared to the flow graph based approach used in other implementations of flow analyses [2, 10, 14], our framework has several advantages: using techniques described in [8, 15], constrained types can be simplified on-the-fly and garbage collection of unreachable constraints can be performed as well, leading to more efficient analyses; and, re-analysis of a function in a different polyvariant context is also realized by instantiation of the function’s constrained type scheme, and does not require re-analysis of the function body.

This paper presents the first proposal to use constrained type schemes to model polyvariance; there are several other related approaches in the literature. Palsberg and Pavlopoulou [16] develop an elegant framework for polyvariant analyses in a type system with union/intersection types and subtyping. There are also subtype-free type-based realizations of polymorphism which can be adapted to polyvariant flow analysis. Let-polymorphism is the classic form of polymorphism used in type inference for subtype-free languages, and has been adapted to constrained types in [3, 7], as well as directly in the flow analysis setting by Wright and Jagannathan [19]. Another representation of polymorphism found in subtype-free languages is via rank-2 intersection types [11], which has also been applied to polyvariant flow analysis [4]. The Church group has developed type systems of union/intersection types decorated with flow labels to indicate the flow information [18].

The form of polyvariance we use is quite general: we show how CPA, nCFA, and other analyses may be expressed in the framework. A $\forall$ type is given to each function in the program, and for every different call site and each different type of argument value the function is applied to, a new contour (re-analysis of the function via instantiation of the $\forall$ type) is possible. The framework is flexible in how contours are generated: a completely new contour can be assigned for an particular argument type applied to the function, or for that argument type it can share a pre-existing contour. For example, 0CFA is the strategy which uses exactly one contour for every function.

One difficult problem for CPA is the issue of termination: without a termination check, the analysis may loop forever on some programs, producing infinitely many contours. We develop a termination condition which detects a certain kind of self-referential flow in the constraints and prove that by merging some contours in this case, non-termination is prevented and the analysis is implementable. Our termination condition is different from the call-graph based condition commonly used in other algorithms, which we show will not guarantee termination in all cases.

We also aim here to model polyvariant algorithms capable of handling *data polymorphism*: the ability of an imperative variable to hold values of different types at run-time. Data polymorphism arises quite frequently in object-oriented programming, especially with container classes, and it poses special problems for flow analysis. The one precise algorithm for detecting data polymorphism is the iterative flow analysis (IFA) of Plevyak and Chien [14]. We present a simple non-iterative algorithm, Data-Adaptive CPA, based on an approach distinct from that of IFA.

2 A Framework for Polyvariant Flow Analysis

This section presents the framework of polyvariant constrained type inference. In the next section we instantiate the framework for particular analyses.

2.1 The Language

The language we study here is an extension to the language used in Palsberg and Pavlopoulou's union/intersection type framework for flow analysis [16], adding mutable state so we can model data polymorphism. We believe the concepts of current paper should scale with relative ease to languages with records, objects, classes, and other features, as illustrated in [7, 6].

Definition 21 (The language):

$$e = x \mid n \mid \texttt{succ}\, e \mid \texttt{if0}\, e\, e\, e \mid \lambda x.e \mid e\, e \mid \texttt{new} \mid e \texttt{ := } e \mid \texttt{!}e \mid e \texttt{ ; } e$$

This is a standard call-by-value lambda calculus extended with reference cells. Execution of a `new` expression creates a fresh, uninitialized reference cell. We use `new` because it models the memory creation mode of languages like Java and C++, where uninitialized references are routinely created. Recursive definitions may be constructed in this language via the Y-combinator.

2.2 The Types

Our basis is an Aiken-Wimmers-style constraint system [3]; in particular it is most closely derived from the system described in [7], which combines constraints and mutable state.

Definition 22 (Types): The type grammar is as follows.

$$\begin{array}{lll}
\tau & \in \mathbf{Type} & ::= t \mid \tau v \mid \mathbf{read}\ t \mid \mathbf{write}\ \tau \mid t_1 \to t_2 \\
t & \in \mathbf{TypeVar} \supset \mathbf{ImpTypeVar} & \\
u & \in \mathbf{ImpTypeVar} & \\
\overline{t} & \in \mathbf{TypeVarSet} & = \ \mathbf{P}_{\mathrm{fin}}(\mathbf{TypeVar}) \\
\tau v & \in \mathbf{ValueType} & ::= \mathbf{int} \mid (\forall\ \overline{t}.\ t \to \tau \setminus C) \mid \mathbf{ref}\ u \\
\tau_1 <: \tau_2 & \in \mathbf{Constraint} & \\
C & \in \mathbf{ConstraintSet} & = \ \mathbf{P}_{\omega}(\mathbf{Constraint})
\end{array}$$

The types for the most part are standard. Function uses (call sites) are given type $t_1 \to t_2$. $\mathbf{ValueType}$ are the types for data values. $\mathbf{ref}\ u$ is the type for a cell whose content has type u. We distinguish imperative type variables $u \in \mathbf{ImpTypeVar}$ for the presentation of data polymorphism. Read and write operations on reference cells are represented with types $\mathbf{read}\ t$ and $\mathbf{write}\ \tau$ respectively. Functions are given polymorphic types $(\forall\ \overline{t}.\ t \to \tau \setminus C)$, where t is the type variable for the formal argument, τ is the return type, C is the set of constraints bound in this type scheme, and $\overline{t}$ is the set of bound type variables. Such types are also referred as $\forall$ types or closure types in the paper.

(Var) $\dfrac{A(x) = t}{A \vdash x : t \setminus \{\}}$

(Int) $\dfrac{}{A \vdash n : \mathbf{int} \setminus \{\}}$

(Succ) $\dfrac{A \vdash e : \tau \setminus C}{A \vdash \mathtt{succ}\, e : \mathbf{int} \setminus \{\tau <: \mathbf{int}\} \cup C}$

(If0) $\dfrac{A \vdash e_1 : \tau_1 \setminus C_1,\ e_2 : \tau_2 \setminus C_2,\ A \vdash e_3 : \tau_3 \setminus C_3}{A \vdash \mathtt{if0}\ e_1\, e_2\, e_3 : t \setminus \{\tau_1 <: \mathbf{int},\ \tau_2 <: t,\ \tau_3 <: t\} \cup C_1 \cup C_2 \cup C_3}$

(Abs) $\dfrac{A, \{x : t\} \vdash e : \tau \setminus C}{A \vdash \lambda x.e : (\forall\, \overline{t}.\, t \to \tau \setminus C) \setminus \{\}}$
where $\overline{t} = \mathit{FreeTypeVar}(t \to \tau \setminus C) - \mathit{FreeTypeVar}(A)$

(Appl) $\dfrac{A \vdash e_1 : \tau_1 \setminus C_1,\ e_2 : \tau_2 \setminus C_2}{A \vdash e_1\, e_2 : t_2 \setminus \{\tau_1 <: t_1 \to t_2,\ \tau_2 <: t_1\} \cup C_1 \cup C_2}$

(New) $\dfrac{}{A \vdash \mathtt{new} : \mathbf{ref}\ u \setminus \{\}}$

(Read) $\dfrac{A \vdash e : \tau \setminus C}{A \vdash \mathtt{!}e : t \setminus \{\tau <: \mathbf{read}\ t\}}$

(Write) $\dfrac{A \vdash e_1 : \tau_1 \setminus C_1,\ e_2 : \tau_2 \setminus C_2}{A \vdash e_1 \mathtt{:=} e_2 : \tau_2 \setminus \{\tau_1 <: \mathbf{write}\ \tau_2\} \cup C_1 \cup C_2}$

(Seq) $\dfrac{A \vdash e_1 : \tau_1 \setminus C_1,\ e_2 : \tau_2 \setminus C_2}{A \vdash e_1\ ;\ e_2 : \tau_2 \setminus C_1 \cup C_2}$

Fig. 1. Type inference rules

2.3 The Type Inference Rules

We present the type inference rules in Figure 1. A type environment A is a mapping from program variables to type variables. Given a type environment A, the proof system assigns a type to expression e via the type judgment $A \vdash e : \tau \setminus C$, where τ is the type for e, and C is the set of constraints which models the flow paths in e. We abbreviate $A \vdash e : \tau \setminus C$ as $\vdash e : \tau \setminus C$ when A is empty. The rules are deterministic except that nondeterminism may arise in the choice of type variables. We restrict type derivations to be of a form where fresh type variables are used whenever it is possible. With this restriction, type inference is trivially decidable and is unique modulo choice of type variable names.

Definition 23 (Type inference algorithm): For closed expression e, its inferred type is $\tau \setminus C$ provided $\vdash e : \tau \setminus C$.

The intuition behind those inference rules is that a subtyping constraint $\tau_1 <: \tau_2$ indicates a potential flow from expressions of type τ_1 to expressions of type τ_2. The rules generally follow standard presentations of Aiken-Wimmer constrained type system, except for the (Abs) and cell typing rules. Detailed descriptions of other rules could be found in [7, 3]. The (Abs) rule assigns each function a polymorphic type $(\forall\ \overline{t}.\ t \to \tau \setminus C)$. In this rule, $\mathit{FreeTypeVar}(\cdot)$ is a function that extracts free type variables, $\overline{t}$ collects all the type variables generated when the inference is applied to the function body, and C collects all the constraints corresponding to the function body. The manner in which $\forall$ type schemes are formed is similar to standard polymorphic constrained type systems, but the significant difference here is that every function is given a $\forall$ type. By contrast, in a system based on let-polymorphism, the `let` construct dictates where $\forall$ types are introduced.

The (New) rule assigns the reference cell type $\mathbf{ref}\ u$, with u, the type of the cell content, initially unconstrained. In the (Read) rule, $\mathbf{read}\ t$ is the type for a cell whose read result is of type t. In the (Write) rule, $\mathbf{write}\ \tau_2$ is the type for a cell assigned with a value of type τ_2.

We take an intensional view of types: two types are equivalent if and only if they are syntactically identical. In particular, $\forall$ types corresponding to different functions in the program are always different, even though they might be α-variants. This is because we wish to distinguish different functions in the analysis to obtain precise flow properties. For type soundness properties, an extensional view could be taken.

We illustrate the inference rules with the example studied in [16]:

$$E_1 \equiv (\lambda f.\mathtt{succ}\,((f\,f)\,0)\,(\mathtt{if0}\ n\,(\lambda x.x)\,(\lambda y.\lambda z.z))$$

To ease presentation, each program variable is inferred with a type variable having exactly the same name. We have

$\vdash\ (\lambda f.\mathtt{succ}\,((f\,f)\,0)\ :\ \tau_f\backslash\{\}$, where $\tau_f \equiv (\forall\ \{f, t_1, t_2, t_3, t_4\}.f \to \mathbf{int} \setminus \{f <: t_1 \to t_2, f <: t_1, t_2 <: t_3 \to t_4, \mathbf{int} <: t_3, t_4 <: \mathbf{int}\})$,
$\vdash\ (\lambda x.x) : \tau_x\backslash\{\}$, where $\tau_x \equiv (\forall\ \{x\}.x \to x\backslash\{\})$,
$\vdash\ (\lambda y.\lambda z.z)\ :\ \tau_y\backslash\{\}$, where $\tau_y \equiv (\forall\ \{y\}.y \to (\forall\ \{z\}.z \to z\backslash\{\})\backslash\{\})$.
$\vdash\ E_1\ :\ t_7 \setminus \{\mathbf{int} <: \mathbf{int}, \tau_x <: t_5, \tau_y <: t_5, \tau_f <: t_6 \to t_7, t_5 <: t_6\}$

2.4 Computation of the Closure

The inference algorithm applied to program e results in a type judgment $\vdash\ e\ :\ \tau \setminus C$. For a flow analysis, we need to generate all the possible data-flow and control-flow paths and propagate value types along all the data-flow paths. This is achieved by applying the closure rules of Figure 2 to C, propagating information via deduction rules on the subtyping constraints.

The rule (Trans) is the transitivity rule which models run-time data flow by propagating value types forward along flow paths. The (Read) closure rule applies when a read operation is applied on a cell of type $\mathbf{ref}\ u$, and the reading result is of type t. By constraint $u <: t$, the cell content flows to the reading result. The (Write) closure rule applies when a write operation is applied on a cell of type $\mathbf{ref}\ u$, and a value of type τ is assigned to the cell. By constraint $\tau <: t$, the value flows to the content of the cell. With (Read) and (Write) rules together, any value assigned to a cell flows to the cell's reading result. Flanagan [9] uses a related set of rules for references and was the source of the idea for us.

The most important closure rule is (Fun), which performs $\forall$ elimination. The constraint $(\forall\ \overline{t}.\ t \to \tau \setminus C) <: t_1 \to t_2$ indicates a function flowing to a call site, where $(\forall\ \overline{t}.\ t \to \tau \setminus C)$ is the type for the function and $t_1 \to t_2$ is the type representing the call site. The constraint $\tau v <: t_1$ means that a value of type τv flows in as the actual

$$\text{(Trans)}\ \frac{\tau v <: t,\ \ t <: \tau}{\tau v <: \tau}$$

$$\text{(Fun)}\ \frac{(\forall\ \overline{t}.\ t \to \tau \setminus C) <: t_1 \to t_2,\ \ \tau v <: t_1}{\tau v <: \Theta(t),\ \ \Theta(\tau) <: t_2,\ \ \Theta(C)}$$
$$\text{where } \Theta = \texttt{Poly}((\forall\ \overline{t}.\ t \to \tau \setminus C) <: t_1 \to t_2, \tau v)$$

$$\text{(Read)}\ \frac{\mathbf{ref}\ u <: \mathbf{read}\ t}{u <: t}$$

$$\text{(Write)}\ \frac{\mathbf{ref}\ u <: \mathbf{write}\ \tau}{\tau <: u}$$

Fig. 2. Constraint Closure rules

argument. At run-time, upon each function application, all local variables of the function are allocated fresh locations on the stack. To model this behaviour in the analysis, a renaming $\Theta \in \mathbf{TypeVar} \xrightarrow{\mathrm{p}} \mathbf{TypeVar}$ is applied to type variables in $\overline{t}$. The partial function Θ is extended to types, constraints, and constraint sets in the usual manner. $\Theta(t)$ for $t \notin \overline{t}$ is defined to return t. We call $\Theta(\tau)$ an *instantiation* of τ. Following the terminology of Shivers' nCFA [17], we call a renaming Θ a *contour*. The $\forall$ is eliminated from $(\forall\ \overline{t}.\ t \to \tau \setminus C)$ by applying Θ to C. The (Fun) rule then generates additional constraints to capture the flow from the actual argument τv to the formal argument $\Theta(t)$, and from the return value $\Theta(\tau)$ to the application result t_2. The (Fun) rule is parameterized by function $\texttt{Poly} \in \mathbf{Constraint} \times \mathbf{ValueType} \to (\mathbf{TypeVar} \xrightarrow{\mathrm{p}} \mathbf{TypeVar})$, which decides for this particular function, call site, and actual argument type, which contour is to be used (*i.e.*, created or reused). Providing a concrete `Poly` instantiates the framework to give a concrete algorithm. For example, the monovariant analysis 0CFA is defined by letting `Poly` always return the identity renaming. This particular example shows how `Poly` may reuse existing contours. The differing analyses are defined by differing `Poly` which use different strategies for sharing contours. In the next section we show how this works by presenting some particular `Poly`.

Definition 24 (Closure): For a constraint set C, $\mathit{Closure}_{\texttt{Poly}}(C)$ is the least superset of C closed under the closure rules of Figure 2.

This closure is well-defined since the rules can be seen to induce a monotone function on constraint sets. By this definition, some `Poly` may produce infinite closures since infinitely many contours may be created. Such analyses are still worthy of study even though they are usually unimplementable.

Definition 25 (Flow Analysis): Define $\mathit{Analysis}_{\texttt{Poly}}(e) = \mathit{Closure}_{\texttt{Poly}}(C)$, where the inference algorithm infers $\vdash\ e\ :\ \tau \setminus C$.

The output of an analysis is a set of constraints, which is the closure of the constraint set generated by the inference rules. The closure contains complete flow information about the program, various program properties can be deduced from it.

Definition 26 (Type-Checking): A program e is well-typed iff $\mathit{Analysis}_{\texttt{Poly}}(e)$ contains no immediately type-contradictory constraints such as $\mathbf{ref}\ u <: t \to t'$.

For example, analyzing program `succ` $(\lambda x.x)$ would generate a type-contradictory constraint $(\forall\ \{x\}.x \rightarrow x\backslash\{\})$ <: **int**, which indicates an type error. A computation state is *wrong* if computation cannot continue due to a type error. Our type system does not statically check for errors due to reading uninitialized cells.

To illustrate how the results of a conventional control flow analysis can be obtained in our framework, we use the fact that by the structure of the inference rules, every $\forall$ type in the closure corresponds to a unique lambda abstraction in the program.

Definition 27 (Control Flow Analysis): For an expression e in the program, if e is assigned with type τ by the inference rules, the function corresponding to $(\forall\ \overline{t'}.\ t' \rightarrow \tau' \setminus C')$ is considered flowing to e, if either $\tau = (\forall\ \overline{t'}.\ t' \rightarrow \tau' \setminus C')$ or $(\forall\ \overline{t'}.\ t' \rightarrow \tau' \setminus C')$ <: $t \in \mathit{Analysis}_{\texttt{Poly}}(e)$, and either $\tau = t$ or t is an instantiation of τ.

The above definition includes two cases: either e is directly assigned with a $\forall$ type, in this case e is a lambda abstraction which trivially flows to itself; or e is assigned with a type variable by the inference rules, and the type variable or an instantiation of it has a $\forall$ type as lower bound.

A subject reduction property for our type system can be established, with a proof similar to the one in [7]. The subject reduction property implies the type soundness and flow soundness of the framework.

Theorem 28 (Subject Reduction, Type Soundness, Flow Soundness): 1. The type system has a subject reduction property;
2. A well-typed program e cannot go *wrong* during execution;
3. If an expression evaluates to a closure value of a function, the function is considered flowing to the expression by the the control flow analysis.

The soundness of the framework implies that any analysis defined as an instantiation of the framework is also sound.

3 Instantiating the Framework

In this section we present various polyvariant algorithms as instantiations of our framework.

3.1 nCFA Instantiation

In Shivers' nCFA analysis [17], each function application (call) is associated with a call-string of length at most n. The call-string contains the last n or fewer calls on the call-path leading to this application. Applications of the same function share the same contour (i.e., analysis of the function) if they have the same call-string. To present nCFA in our framework, type variables are defined with superscripts that denote the call-string:

$$\begin{array}{lll} \alpha \in \textbf{Identifier} & & \\ s \in \textbf{Superscript} = & \textbf{Identifier List} & \\ t \in \textbf{TypeVar} & ::= \alpha^s & \end{array}$$

We use the following list notation: The empty list is $[\,]$, $[\alpha_1, \ldots, \alpha_m]$ is a list of m elements, $l_1\ @\ l_2$ appends lists l_1 and l_2, and $l(1..n)$ is the list consisting of the first

$min(n, length(l))$ elements of list l. Each type variable α^s is tagged with a call-string s. All type variables generated by the inference rules have empty lists as superscripts. By the inference rule (Appl), a call site is inferred with a type $\alpha_1^{[\,]} \to \alpha_2^{[\,]}$, we use α_2 to identify this call site, thus a call-string is a list of such identifiers. All bound type variables of a $\forall$ type have empty list superscripts. When the $\forall$ quantifier is eliminated by the (Fun) closure rule, those bound type variables are renamed by changing the superscripts from empty lists to the appropriate call-strings.

Definition 31 (nCFA Algorithm): The nCFA algorithm is defined as the instantiation of the framework with `Poly` $= \mathbf{CFA}$, where

$$\begin{aligned}&\mathbf{CFA}((\forall\, \bar{t}.\, t \to \tau \setminus C) <: t_1 \to \alpha_2^{s_2}, \tau v) = \Theta, \text{ where for each } \alpha^{[\,]} \in \bar{t},\\ &\quad \Theta(\alpha^{[\,]}) = \alpha^{s'}, \text{where } s' = ([\alpha_2] \mathbin{@} s_2)(1..n)\end{aligned}$$

It can be shown by induction that s' is the call-string for application $(\forall\ \bar{t}.\ t \to \tau \setminus C) <: t_1 \to \alpha_2^{s_2}$. The definition of Θ ensures that applications of the same function share the same contour if and only if they have the same call-string.

Not only is nCFA inefficient, but even for large n it may be imprecise. Applying nCFA to program E_1, since $(\lambda f \ldots)$ has only one application, the (Fun) rule generates only one contour Θ for this function, resulting in $\tau_x <: \Theta(f)$ and $\tau_y <: \Theta(f)$. This means both $(\lambda x.x)$ and $(\lambda y.\lambda z.z)$ flow to f, and at the application site $f\,f$ there are four applications. One of them, $(\lambda x.x)$ applying to $(\lambda y.\lambda z.z)$ leads to a type error: $(\forall\ \{z\}.z \to z\backslash\{\}) <: \mathbf{int}$. Hence nCFA fails to type-check E_1 for arbitrary n.

3.2 Idealized CPA

The Cartesian Product Algorithm (CPA) [1, 2] is a concrete type inference algorithm for object-oriented languages. For a message sending expression, CPA computes the cartesian product of the types for the actual arguments. For each element of the cartesian product, the method body is analyzed exactly once with one contour generated. The calling-contexts of a method are partitioned by the cartesian product, rather than by call-strings as in nCFA. In our language, each function has only one argument. For each function, CPA generates exactly one contour for each distinct argument type that the function may be applied to. Without a termination check, CPA may fail to terminate for some programs. We first present an idealized CPA which may produce an infinite closure, and in Section 5 show how a terminating CPA analysis may be defined which keeps the closure finite. To present CPA, type variables are defined with structure:

$$\begin{aligned}&\alpha \in \mathbf{Identifier}\\ &t\ \in \mathbf{TypeVar}\ ::= \alpha \mid \alpha^{\tau v}\end{aligned}$$

The inference rules are constrained to generate type variables without superscripts.

Definition 32 (Idealized CPA algorithm): The Idealized CPA algorithm is the instantiation of the framework with `Poly` $= \mathbf{CPA}$, where

$$\mathbf{CPA}((\forall\, \bar{t}.\, t \to \tau \setminus C) <: t_1 \to t_2, \tau v) = \Theta, \text{ where for each } \alpha \in \bar{t}, \Theta(\alpha) = \alpha^{\tau v}$$

The contours Θ are generated based on the actual argument type τv, independent of the application site $t_1 \to t_2$. This is the opposite of $\mathbf{CFA}$, which ignores the value type

τv, and only uses the call site $t_1 \to t_2$. Given a particular function and its associated $\forall$ type in a program, this algorithm will generate a unique contour ($\forall$ elimination) for each distinct value type the function is applied to. It however may share contours across call sites. Agesen [2] presents convincing experimental evidence that the CPA approach is both more efficient and more feasible than nCFA.

We now sketch what Idealized CPA will produce when applied to program E_1. Even though there is only one application site for $(\lambda f \ldots)$, it applies to two different actual argument values. So, the (Fun) rule generates two contours Θ_1 and Θ_2 for $(\lambda f \ldots)$ with $\Theta_1(f) = f^{\tau_x}, \tau_x <: f^{\tau_x}, \Theta_2(f) = f^{\tau_y}, \tau_y <: f^{\tau_y}$. At application site $f\,f$, there would be only two applications: $(\lambda x.x)$ applying to itself and $(\lambda y.\lambda z.z)$ applying to itself. Thus the program is type-checked successfully.

4 Data Polymorphism

Data polymorphism is defined informally in [2] as the ability of an imperative program variable to hold values of different types at run-time. In our language, a more precise definition could be that cells created from a single imperative creation point (`new` expression) in the program could be assigned with run-time values of different types. CPA addresses parametric polymorphism effectively, but may lose precision in the presence of data polymorphism. For example, consider when CPA is applied to the program

$$E_2 \equiv (\lambda f.(\lambda x.\ x \mathrel{\texttt{:=}} 0; \texttt{succ}\ \texttt{!}x)(f\,1); (f\,2) \mathrel{\texttt{:=}} (\lambda y.y))\,(\lambda z.\texttt{new})$$

Function $(\lambda y.y)$ has type $(\forall\ \{y\}.y \to y\backslash\{\})$, and $(\lambda z.\texttt{new})$ has type $(\forall\ \{z,u\}.z \to \mathbf{ref}\ \ u\backslash\{\})$. The two applications of $(\lambda z.\texttt{new})$ have same actual argument type $\mathbf{int}$, so one contour Θ is shared by the two applications. At run-time the two applications return two distinct cells, but in CPA closure, the two cells share type $\mathbf{ref}\ \ u'$ (assume $\Theta(u) = u'$), since there is only one contour for $(\lambda z.\texttt{new})$. At run-time, one cell is assigned with 0, and the other is assigned with $(\lambda y.y)$. The two assignments are both reflected on u' as constraints $\mathbf{int}\ <:\ u'$ and $(\forall\ \{y\}.y \to y\backslash\{\})\ <:\ u'$, as if there were only one cell, which is assigned with values of two different types. This leads to a type error: $(\forall\ \{y\}.y \to y\backslash\{\}) <: \mathbf{int}$. But if distinct contours were used for the two applications of $(\lambda z.\texttt{new})$, the two cells would have separate cell types and the program would be type-checked.

This small example illustrates that data polymorphism is a problem that arises in a function that contains a creation point (a `new` expression). Different applications of the function may create different cells which are assigned with values of different types, a precise analysis should disambiguate these cells by letting them have separate cell types. To illustrate how data polymorphism can be modeled in our framework, we present a refinement of CPA to give better precision in the analysis of data polymorphic programs.

Consider two applications of a single function. If the applications have same actual argument type, then CPA generates a single contour for them. But, if the two applications return separate mutable data structures at run-time, and the data structures are modified differently after being returned from the two different applications, CPA would lose precision by merging the data structures together. If two separate contours were used for the two applications, the imprecision could be avoided. In the result of CPA analysis, such a function has a return type which is a mutable data structure with polymorphic contents. We call such functions *data-polymorphic*. In program E_2, $(\lambda z.\texttt{new})$ is data-polymorphic, and the other functions are data-monomorphic.

Based on the above observation, our Data-Adaptive CPA algorithm is a two-pass analysis. The first pass is just CPA. From the CPA closure, we detect a set of functions which are possibly data-polymorphic. In the second pass, for data-polymorphic functions, a distinct contour is generated for *every* function application. In this way, imprecision associated with data-polymorphic functions can be avoided; only CPA splitting is performed for data-monomorphic functions, avoiding generation of redundant contours.

Mutable data structures with polymorphic contents are detected from the CPA closure with following definition:

Definition 41 (Data Polymorphic Types): Type τ is data-polymorphic in constraint set C if any of the following cases hold:

1. $\tau = \textbf{ref}\ u$, $\tau v_1 <: u \in C$, $\tau v_2 <: u \in C$, and $\tau v_1 \neq \tau v_2$;
2. τ is type variable t, $\tau' <: t \in C$, and τ' is data-polymorphic in C;
3. $\tau = \textbf{ref}\ u$ and u is data-polymorphic in C;
4. $\tau = (\forall\ \overline{t'}.\ t' \rightarrow \tau' \setminus C')$ and τ' is data-polymorphic in C.

The above definition is inductive. The first case is the base case, detecting cell types with polymorphic contents. The second case declares a type variable as data-polymorphic when it has a data polymorphic lower bound. The remaining two cases are inductive cases based on the idea that a type is data-polymorphic if it has a data-polymorphic component. Particularly, a closure type is declared as data-polymorphic when the type of its return value is data-polymorphic. Note that, for purely functional programs with no usage of cells, no types would be detected as data-polymorphic.

Recall that CPA type variables are either of the form α or $\alpha^{\tau v}$. We define an operation $erase$ on type variables as: $erase(\alpha) = \alpha,\ \ erase(\alpha^{\tau v}) = \alpha$. And we extend it naturally to types, defining $erase(\tau)$ as the type with all superscripts erased from all type variables in τ. In particular, $erase$ maps a closure type to the type for the lambda abstraction in the program corresponding to the closure type, and it maps a cell type to the type for the corresponding creation point (`new` expression) in the program. From now on, we call τv an *instantiation* of $erase(\tau v)$.

Definition 42 (Data Polymorphic Functions): For function $\lambda x.e$ assigned with type $(\forall\ \overline{t}.\ t \rightarrow \tau \setminus C)$ by the inference rules, $\lambda x.e$ is a data-polymorphic function in constraint set C' iff there appears τ' in C' s.t. $erase(\tau') = \tau$ and τ' is data-polymorphic in C'.

In the above definition we use the fact that every distinct function in the program is given a unique type $(\forall\ \overline{t}.\ t \rightarrow \tau \setminus C)$ by the inference rules. The constraint set C' is a flow analysis result of the program. The condition $erase(\tau') = \tau$ means that the function $\lambda x.e$ may return a value of type τ'. Since τ' is data-polymorphic in C', we know that, according to analysis result C', the function may return mutable data structures with polymorphic contents, and we declare it as a data-polymorphic function.

Definition 43 (Data-Adaptive CPA): For program e, Data-Adaptive CPA is an instantiation of the framework with $\texttt{Poly} = \mathbf{DCPA}$, where

$$\mathbf{DCPA}((\forall\ \bar{t}.\ t \to \tau \setminus C) <: t_1 \to t_2, \tau v) = \Theta, \text{where for each } \alpha \in \bar{t},$$

$$\Theta(\alpha) = \begin{cases} \alpha' & \text{where } \alpha' \text{ is a fresh identifier, if } erase(\forall\ \bar{t}.\ t \to \tau \setminus C) \text{ is type for} \\ & \text{a data-polymorphic function in } Analysis_{\mathbf{CPA}}(e) \\ \alpha^{\tau v} & \text{otherwise} \end{cases}$$

The second pass of Data-Adaptive CPA differs from CPA only when the function is detected as data polymorphic in the closure obtained by the first CPA pass. In this case, a new contour is always generated for every application. We now illustrate Data-Adaptive CPA on program E_2. After the first CPA pass, we have

$$\mathbf{int} <: u', (\forall\ \{y\}.y \to y \backslash \{\}) <: u') \in Analysis_{\mathbf{CPA}}(E_2)$$

Thus u' is data-polymorphic, and so is $\mathbf{ref}\ \ u'$. Since $\lambda z.\texttt{new}$ is inferred with type $(\forall\ \{z,u\}.z \to \mathbf{ref}\ u \backslash \{\})$ and $erase(\mathbf{ref}\ u') = \mathbf{ref}\ u$, $\lambda z.\texttt{new}$ is a data-polymorphic function. In the second pass, the two applications of $\lambda z.\texttt{new}$ have separate contours, and the program type-checks.

We briefly sketch how Data-Adaptive CPA could be applied to data polymorphism in object-oriented programming. We illustrate the ideas by assuming an encoding of instance variables as cells, objects as records (which we expect can be added to our language without great difficulty), classes as class functions, and object creation as application of class functions. An example of such an encoding is presented in [7].

Consider applying such an encoding to the Java program fragment of Figure 3: The two `new Box()` expressions would be encoded as two applications of the class

```
class Box {
    public Object content;
    public void set(Object obj) {
        content=obj;
    }
    public Object get() {
        return content;
    }
}
...
Box box1=new Box(); box1.set(new Integer(0));
Box box2=new Box(); box2.set(new Boolean(true));
... box1.get() ...
```

Fig. 3. Java program exhibiting the need for data polymorphism

function for class `Box`. When CPA is applied, since the two applications always apply to arguments of same type in any object encoding, the two applications share a single contour. Thus the two `Box` instances share a same object type, and the analysis would imprecisely conclude that the result of `box1.get()` includes object type `Boolean`. When Data-Adaptive CPA is applied, from the closure of the first CPA pass, the instance variable `content` would be detected as being associated with a data-polymorphic cell type. Since the class function for class `Box` returns a object value with `content` as a component, the class function would be detected as a data-polymorphic function. During the second pass, the two applications of the class function would have separate contours, thus the two instances of `Box` would have separate types and the imprecision would be avoided.

For programs with much data polymorphism, Data-Adaptive CPA may become impractical as many functions are detected as data-polymorphic. Similar to Agesen's CPA implementation [2], a practical implementation should restrict the number of contours generated.

Plevyak and Chien's iterative flow analysis (IFA) [14] uses an iterative approach for precise analysis of data polymorphic programs. The first pass analyzes the program by letting objects of the same class share the same object contour. Every pass detects a set of confluence points (imprecise flow graph nodes where different data values merge) based on the result of the previous pass, and generates more contours with aim to resolve the imprecision at confluence points. The iteration continues until a fixed-point is reached. The advantage of IFA is that splitting is performed only when it is profitable, yet every pass is a whole-program analysis and the number of passes needed could be large. Use of declarative parametric polymorphism [5] to guide the analysis of data polymorphism could be a completely different approach that also could be considered.

5 Terminating CPA Analyses

Any instantiation of our polyvariant framework terminates when only finitely many distinct contours are generated. The nCFA algorithms we defined terminate for arbitrary programs since the number of call-strings of length no more than n is finite. Unfortunately, the Idealized CPA and Data-Adaptive CPA algorithms fail to terminate for some programs.

Agesen [2] develops various methods to detect recursion and avoid the generation of infinitely many contours over recursive functions in his CPA implementation. One approach is to construct a call-graph during analysis, and restrict the number of contours generated along a cycle in the call-graph. However, for Idealized CPA, adding call-graph cycle detection is not enough to ensure termination. Consider the program

$$E_3 \equiv (\lambda c.\ c \texttt{ := } \lambda x.x; (\lambda d.\ c \texttt{ := } (\lambda y.\ d\ y))\ \texttt{!}c)\ \texttt{new}$$

Its call-graph has only one edge: function $(\lambda c \ldots)$ calls $(\lambda d \ldots)$. There is no cycle in it. Consider running Idealized CPA on the program. For each value type lower bound of of u (assume the cell has type $\mathbf{ref}\ u$), there is a contour generated for function $(\lambda d \ldots)$. At first the type for $f_0 = (\lambda x.x)$ becomes a lower bound of u, one contour is generated for function $(\lambda d \ldots)$, and the type for closure $f_1 = (\lambda y.f_0\, y)$ becomes another lower bound of u. So another contour is generated for $(\lambda d \ldots)$, and the type for closure $f_2 = (\lambda y.f_1\, y)$ also becomes a lower bound of u. This process would repeat forever, with an infinite number of contours generated for function $(\lambda d \ldots)$. This example shows that call-graph based approach cannot ensure the termination of Idealized CPA.

Here we present a novel approach that ensures the termination of CPA for arbitrary programs. Our approach is based on the following observation: when Idealized CPA fails to terminate for a program, there must be a cyclic dependency relation among those functions having infinitely many contours. In the example, there exists such a cyclic relation: function $(\lambda y \ldots)$ is lexically enclosed by $(\lambda d \ldots)$, and $(\lambda d \ldots)$ applies to closure values corresponding to $(\lambda y \ldots)$. If we detect such cycles and restrict the number of contours generated for functions appearing in cycles, non-termination could be avoided. To be precise, the key of our method is to construct a relation among value types during closure computation. This relation is defined as:

Definition 51 (Flow Dependency, $\Rightarrow$): For constraint set C, define $\Rightarrow$ as a relation among value types such that if either

$$\tau v_1 <: t \to t', \tau v_2 <: t \in C, \tau v_1 = (\forall\, \overline{t}_1.\, t_1 \to \tau_1 \setminus C_1)$$

or

τv_1 occurs as a subterm of $(\forall\, \overline{t}_2.\, t_2 \to \tau_2 \setminus C_2), \tau v_2 = (\forall\, \overline{t}_2.\, t_2 \to \tau_2 \setminus C_2), \tau v_1 \neq \tau v_2$, and there exists a $t \in \overline{t}_2$ such that t appears in τv_1

holds, then $erase(\tau v_1) \Rightarrow erase(\tau v_2)$ in C.

The first case above defines a dependency when closure type τv_1 applies to value type τv_2. The second case defines a dependency when closure type τv_2 contains value type τv_1 as a subterm, so that when a new contour is generated for closure type τv_2, a new value type is created which is τv_1 with some of its free type variables renamed. If $\tau v_1 \Rightarrow \tau v_2$ in C_1, and $C_1 \subseteq C_2$, we have $\tau v_1 \Rightarrow \tau v_2$ in C_2. Thus relation $\Rightarrow$ could be incrementally computed along with the incremental closure computation. We abbreviate $\tau v_1 \Rightarrow \tau v_2$ in C as $\tau v_1 \Rightarrow \tau v_2$ when C refers to the current closure under computation. We call $\tau v_1 \Rightarrow \tau v_2, \ldots, \tau v_n \Rightarrow \tau v_1$ a cycle, and we write $\tau v_1 \overset{*}{\Rightarrow} \tau v_n$ if there exists a sequence $\tau v_1 \Rightarrow \tau v_2, \ldots, \tau v_{n-1} \Rightarrow \tau v_n$.

Definition 52 (Terminating CPA): Terminating CPA is the instantiation of the framework obtained by defining `Poly` as:

$$\texttt{Poly}((\forall\, \overline{t}.\, t \to \tau \setminus C) <: t_1 \to t_2, \tau v') = \Theta, \text{where for each } \alpha \in \overline{t},$$
$$\Theta(\alpha) = \begin{cases} \alpha^{erase(\tau v')} & \text{if } erase(\tau v') \overset{*}{\Rightarrow} erase((\forall\, \overline{t}.\, t \to \tau \setminus C)) \\ \alpha^{\tau v'} & \text{otherwise} \end{cases}$$

The new algorithm differs from Idealized CPA in just one case: when a closure of type $(\forall\ \overline{t}.\ t \to \tau \setminus C)$ is applied to argument type $\tau v'$ and we have $erase(\tau v') \overset{*}{\Rightarrow} erase((\forall\ \overline{t}.\ t \to \tau \setminus C))$ in the current closure, then by the definition of $\Rightarrow$, there would be a cycle: $erase(\tau v') \overset{*}{\Rightarrow} erase((\forall\ \overline{t}.\ t \to \tau \setminus C)) \Rightarrow erase(\tau v')$. In this case, instead of renaming type variables in $\overline{t}$ as in Idealized CPA, they are renamed to a form only dependent on $erase(\tau v')$. In this way, even if $(\forall\ \overline{t}.\ t \to \tau \setminus C)$ applies to different types which are different instantiations of $erase(\tau v')$, there is only one contour generated for them. We will prove shortly that this will ensure termination of the closure computation.

Applying the algorithm to example E_3, suppose that, by the inference rule (Abs), function $(\lambda d \ldots)$ has type τ_d and function $(\lambda y \ldots)$ has type τ_y. Since $(\lambda d \ldots)$ lexically encloses $(\lambda y \ldots)$, we have $\tau_y \Rightarrow \tau_d$; and, since $(\lambda d \ldots)$ applies to closures of $(\lambda y \ldots)$, we also have $\tau_d \Rightarrow \tau_y$. Thus a cycle is detected, only two contours are generated for $(\lambda d \ldots)$, and the algorithm terminates.

Theorem 53 (Termination): The Terminating CPA analysis terminates for arbitrary programs.

Proof: Suppose not, *i.e.,* for some program, its Terminating CPA closure C contains a $\forall$ type τv_0 which has an infinite number of contours. Then, there must exist at least one τv_1 s.t. τv_0 takes as arguments an infinite number of instantiations of $erase(\tau v_1)$, and an infinite number of contours are generated for those applications. To have an infinite number of instantiations of $erase(\tau v_1)$, there must exist a $\forall$ type τv_2 s.t. τv_2 contains $erase(\tau v_1)$ as a sub-term, every new contour of τv_2 causes the generation of a new instantiation of $erase(\tau v_1)$, and τv_2 has an infinite number of contours. Repeating this process gives an infinite sequence $erase(\tau v_0), erase(\tau v_1), \ldots erase(\tau v_i) \ldots$ where for each i, τv_{2*i} has infinite number of contours when applying to instantiations of $erase(\tau v_{2*i+1})$, and $erase(\tau v_i) \Rightarrow erase(\tau v_{i+1})$. Since the program is finite, there are finitely many $erase(\tau v)$ and there must be a cycle in the sequence. Thus, there exists j s.t. $erase(\tau v_{2*j}) \Rightarrow erase(\tau v_{2*j+1}) \overset{*}{\Rightarrow} erase(\tau v_{2*j})$ and τv_{2*j} has an infinite number of contours for applying to instantiations of $erase(\tau v_{2*j+1})$. But, by the definition of `Poly` for Terminating CPA, this is impossible. □

A terminating Data-Adaptive CPA analysis can be similarly defined except that, besides cycles in the Flow Dependency relation, cycles in call-graph also need to be detected.

6 Conclusions

We have defined a polymorphic constrained type-based framework for polyvariant flow analysis. Some particular contributions include: showing how a type system with parametric polymorphism may be used to model polyvariance as well as data polymorphism; modeling nCFA and CPA in our framework; a refinement of CPA in the presence of data polymorphism; and, an approach to ensure the termination of CPA-style analyses.

Acknowledgements

We would like to thank the anonymous referees for their helpful comments.

References

1. Ole Agesen. The cartesian product algorithm. In *Proceedings ECOOP'95*, volume 952 of *Lecture notes in Computer Science*, 1995.
2. Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Stanford University, 1996. Available as Sun Labs Technical Report SMLI TR-96-52.
3. A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

4. Anindya Bannerjee. A modular, polyvariant, and type-based closure analysis. In *International Conference on Functional Programming*, 1997.
5. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33, 10 of *ACM SIGPLAN Notices*, pages 183–200, New York, October 18–22 1998. ACM Press.
6. Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '95*, pages 169–184, 1995.
7. Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. http://www.elsevier.nl/locate/entcs/volume1.html.
8. Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 235–248, New York, June 15–18 1997. ACM Press.
9. Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, 1997.
10. David Grove, Greg DeFouw, Jerey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1997.
11. Trevor Jim. What are principal typings and what are they good for? In *Conference Record of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, 1996.
12. Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 393–408, 1995.
13. Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Conference Record of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 332–345, 1997.
14. John Plevyak and Andrew Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the Ninth Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–340, 1994.
15. François Pottier. A framework for type inference with subtyping. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 228–238. ACM, June 1999.
16. Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In *Proceedings of 25th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 197–208, 1998.
17. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991. Available as CMU Technical Report CMU-CS-91-145.
18. J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A typed intermediate language for flow-directed compilation. In *Theory and Practice of Software Development (TAPSOFT)*, number 1214 in Lecture notes in Computer Science. Springer-Verlag, 1997.
19. Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, January 1998.