

Fixpoint Computation for Polyvariant Static Analyses of Higher-Order Applicative Programs

J. Michael Ashley
Indiana University
and
Charles Consel
Oregon Graduate Institute

This paper presents an optimized general-purpose algorithm for polyvariant, static analyses of higher-order applicative programs. A polyvariant analysis is a very accurate form of analysis that produces many more abstract descriptions for a program than a conventional analysis. It may also compute intermediate abstract descriptions that are irrelevant to the final result of the analysis. The optimized algorithm addresses this overhead while preserving the accuracy of the analysis. The algorithm is also parameterized over both the abstract domain and degree of polyvariance. We have implemented an instance of our algorithm and evaluated its performance compared to the unoptimized algorithm. Our implementation runs significantly faster on average than the other algorithm for benchmarks reported here.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors

General Terms: Algorithms, Design

Additional Key Words and Phrases: abstract interpretation, fixpoint algorithm, program analysis

1. INTRODUCTION

Abstract interpretation [Abramsky and Hankin 1987; Cousot 1981; Jones and Nielson 1990] has been used to formulate a wide variety of static analyses aimed at optimizing programs in practical programming languages. Research in the area has been focusing on the conceptual and formal aspects of the topic. Special emphasis has been put on such issues as accuracy, termination, and relating non-standard and standard semantic definitions. However, the practical aspects involved in implementing an abstract interpreter have not received as much attention. This situation is even more pronounced for analyses of languages with higher-order functions.

A static analysis includes a critical phase that consists of finding a fixpoint to a set of (possibly) recursive abstract functions derived from the analyzed program. To be of any practical use, an analysis should include an efficient and accurate fixpoint algorithm. The efficiency depends mostly on this accuracy. For example, some analyses only determine one abstract description for each object (e.g., functions and data structures) in the program. By analogy with partial evaluation [Consel and Danvy 1993], they are said to be *monovariant*. Other analyses are more accurate. They determine multiple abstract descriptions for each object in a

This research was partially supported by NSF grant CCR-9224375.

Authors' addresses: J. Michael Ashley, Computer Science Department, Lindley Hall 215, Bloomington, Indiana 47405. email: jashley@cs.indiana.edu; Charles Consel, Oregon Graduate Institute, Department of Computer Science, PO Box 91000, Portland, Oregon 97291. email: consel@cse.ogi.edu

program depending on the context in which these objects are used. Again, by analogy with partial evaluation, these analyses are said to be *polyvariant*. They make the fixpoint computation expensive because they yield more abstract descriptions.

Unlike first-order programs, higher-order programs do not have a static control flow graph since higher-order languages treat procedures as values. As a consequence control-flow and data-flow aspects of higher-order programs are intertwined. In fact, a control-flow analysis for higher-order programs (also called closure analysis [Sestoft 1989]) includes data-flow aspects.

In a polyvariant analysis, separating control-flow and data-flow analyses is not desirable since both first-order and higher-order values are used to differentiate contexts in which an object is used [Consel 1993a]. Indeed, accuracy in this scheme is obtained at the cost of iterating control-flow and data-flow analyses [Rytz and Gengler 1992]. Alternatively, control-flow and data-flow analyses can be combined. In a polyvariant analysis, this approach yields more accurate information [Consel 1993a].

As presented in [Young 1989], analyzing a set of mutually recursive functions consists of processing each function at each fixpoint iteration until no changes occur. More precisely, the iteration process is complete when the abstract values have been propagated through every control flow path and the abstract descriptions of the functions have stabilized.

O’Keefe [O’Keefe 1987], Hall and Kennedy [Hall and Kennedy 1992], and others noticed that this strategy causes needless recomputations to be performed. Indeed, a given abstract function can be reanalyzed even though the changes that triggered a new iteration do not affect it. One solution to remedy this situation consists of maintaining dependencies so that when a change occurs, only those abstract functions affected by the change are reanalyzed.

Another problem discussed in the literature (e.g., [Charlier, Musumbu, and Hentenryck 1991]) has to do with the way the control flow of the analyzed program is traversed by the analysis. Some traversal strategies cause abstract functions to be analyzed more times than necessary. For example, traversing a control-flow graph depth-first as opposed to breadth-first can affect the rate at which the analysis reaches a fixpoint.

When the analyzed program is first order, the call graph can be determined statically in one pass. The analysis can then use this information to accelerate the convergence of the fixpoint process. For example, subsets of abstract functions derived from strongly-connected components of the control-flow graph can be solved individually. Strategies for first-order programs are presented in the literature (e.g., [Charlier, Musumbu, and Hentenryck 1991]).

However, when the analyzed program is higher order, only an approximation of the call graph can be computed. More importantly, the call-graph analysis itself requires a fixpoint process. Consequently, exploiting call-graph information to accelerate the fixpoint process requires an analysis for higher-order programs to construct call-graph information incrementally.

This paper presents a new algorithm for implementing accurate fixpoint analyses. Our algorithm applies to call-by-value programs with higher-order procedures and data structures. We introduce an optimization technique designed to deal with the overhead of polyvariance while preserving accuracy.

We have implemented our algorithm in a polyvariant binding-time analysis. A binding-time analysis is a part of the preprocessing phase in an offline partial evaluator [Jones, Sestoft, and Søndergaard 1989; Nielson and Nielson 1992]. It aims at determining invariants in the partial-evaluation process.

$\langle \text{program} \rangle$	$::=$	$\langle \text{definition} \rangle^* \langle \text{exp} \rangle$	program (defs exp)
$\langle \text{definition} \rangle$	$::=$	(define $\langle \text{id} \rangle$ $\langle \text{exp} \rangle$)	def (name exp)
$\langle \text{exp} \rangle$	$::=$	$\langle \text{const} \rangle$	lit (val)
		$\langle \text{id} \rangle$	varref (name)
		(if $\langle \text{exp} \rangle$ $\langle \text{exp} \rangle$ $\langle \text{exp} \rangle$)	cond (test then else)
		(lambda ($\langle \text{id} \rangle$) $\langle \text{exp} \rangle$)	proc (formal body)
		($\langle \text{exp} \rangle$ $\langle \text{exp} \rangle$)	app (rator rand)

Fig. 1. A simple higher-order language

We have experimentally evaluated the new algorithm and compared its performance with the unoptimized version of the algorithm. While we do not give precise bounds on the running time of our algorithm, its implementation can run significantly faster than the implementation of the other algorithm. This estimation is based on a series of benchmarks given in this paper.

Overview

In the next section we present a version of the algorithm that may be used for both monovariant and polyvariant analyses. We then illustrate by example how its accuracy and performance may be hampered in the presence of polyvariance. In Section 3 we revise our algorithm to improve both its performance and accuracy. Some preliminary performance results are given in Section 4, and our approach is put into perspective with related work in Section 5. We close, in Section 6, with conclusions.

2. A GENERIC ANALYSIS

We now present a generic static analysis for a simple higher-order programming language. The analysis is presented in Scheme [Clinger and (editors) 1991; Dybvig 1987] extended with a simple record facility. The facility allows a new record type *name* with fields $x_0 \dots x_n$ to be defined by the expression (define-record *name* ($x_0 \dots x_n$)). Fields of a record can then be accessed and updated with the procedures *name*-> x_i and set-*name*-> x_i ! respectively. For convenience, the construct variant-case allows one to dispatch on the type of a record and extract the value of record's fields by name (see Figure 2 for an example); it evaluates to an unspecified value if the record does not match any case.

The language treated by the analysis is given in the left-hand part of Figure 1. The right-hand part of this figure lists the record declarations associated with each syntactic rule.

A program is a sequence of mutually recursive definitions followed by an initial expression. This corresponds to a record *program* containing two fields: *defs* (the definitions) and *exp* (the initial expression). To simplify the presentation, procedures may take only one argument. There are also no data structures and no side-effects. We discuss returning data structures to the language in our concluding remarks. We do not address the issue of side-effects. The value of a program is the value of its expression, evaluated in the environment established by the definitions of the program.

The generic analysis is abstracted over the abstract domain and polyvariant function. An instantiation of the analysis must meet the following requirements.

- (1) The abstract domain A must form a complete lattice of finite height. Abstract closures, which are the abstract descriptions of run-time closures, are explicitly represented in the domain. The function α is an abstraction function mapping constants to elements of A .
- (2) The polyvariant function π maps a context of analysis to an element of a finite index set [Consel 1993a]. The different descriptions for an object are grouped (i.e., folded together) with respect to this index.

The finite height of the abstract domain ensures that the fixpoint computation terminates. Furthermore, explicitly representing abstract closures allows the analysis to manipulate their components (e.g., their environment and text).

Let us now give an example of both an abstract domain and a polyvariant function that could be used to instantiate the generic analysis. Suppose that we are interested in developing a form of closure analysis. Such an analysis could collect the set of possible closures that could occur at each call site in a program. To do so, we use the following domain sets.

$$\begin{aligned}
 P &= \{\eta_1, \dots, \eta_n\} && \text{Labels for lambda abstractions.} \\
 (\eta, \rho) \in CL &= (P \times Env) && \text{Closures.} \\
 \rho \in Env &= Var \rightarrow A && \text{Environments.} \\
 X \in A &= \mathcal{P}(CL) \cup \{\perp\} && \text{Abstract values.}
 \end{aligned}$$

where $\perp \sqsubseteq \emptyset$.

As it is defined, the domain of abstract values A has an infinite height; this contradicts the first requirement listed above. As is, if the domain A is used in a polyvariant analysis, it may create an infinite number of descriptions for lambda expressions. To obtain a finite domain and to control the polyvariance of the analysis, we introduce a polyvariant function (π) that is used to create two different kinds of abstract descriptions: abstract closures and abstract applications. In fact, π is a family of functions since it can be applied to elements of different domains as shown below.

First, π is used to partition the set of abstract closures created from a lambda expression. Each partition is represented by a *closure variant*, which is an abstract closure representing the abstract closures in the partition. When a lambda expression is analyzed, an abstract closure is formed from the text of the lambda expression and the current lexical environment. The result of the analysis, however, is the closure variant identifying the partition in which the abstract closure lies. Let us give an example of a polyvariant function on the domain A .

$$\begin{aligned}
 \pi_{\overline{A'}} : A &\rightarrow \overline{A'} \\
 \pi_{\overline{A'}}(\perp) &= \perp \\
 \pi_{\overline{A'}}(\emptyset) &= \emptyset \\
 \pi_{\overline{A'}}(X) &= \{\eta \mid (\eta, \rho) \in X\}
 \end{aligned}$$

where $\overline{A'} = \mathcal{P}(P) \cup \{\perp\}$

Although the domain of $\pi_{\overline{A'}}$ is infinite, its co-domain is finite since it maps closures into labels. We are now equipped to define a polyvariant function for a closure.

$$\begin{aligned}
 \pi_{\overline{CL}} : CL &\rightarrow \overline{CL} \\
 \pi_{\overline{CL}}(\eta, \rho) &= (\eta, \{[\pi_{\overline{A'}}(\rho \ x_0)/x_0], \dots, [\pi_{\overline{A'}}(\rho \ x_n)/x_n]\}) \\
 &\text{where } \{x_0, \dots, x_n\} = Dom(\rho)
 \end{aligned}$$

The domain of closure variants and the finite version of domain A , noted \overline{A} , are defined as follows.

$$\begin{aligned} \overline{CL} &= P \times \overline{Env} \\ &\text{where } \overline{Env} = \overline{Var} \rightarrow \overline{A} \\ &\text{and } \overline{A} = \mathcal{P}(\overline{CL}) \cup \{\perp\} \end{aligned}$$

As can be noticed, the domain of closure variants is finite and thus descriptions of abstract closures are finite as well.

The π function on domain \overline{A} is defined as

$$\begin{aligned} \pi_{\overline{A}} : A &\rightarrow \overline{A} \\ \pi_{\overline{A}}(\perp) &= \perp \\ \pi_{\overline{A}}(X) &= \{\pi_{\overline{CL}}(\eta, \rho) \mid (\eta, \rho) \in X\} \end{aligned}$$

π is used a second time to partition the set of values to which a closure variant is applied. It determines the partition of an application. Each of these partitions is represented by an *application variant*. An application variant represents the application of the closure variant to abstract values in the partition and records the result of the application as well. An application variant is an element of the domain

$$\overline{AV} = \overline{CL} \times \overline{A}$$

Considering the purpose of this analysis, the function α would map a first-order value to \emptyset and a higher-order value to a singleton set consisting of the corresponding label.

This example shows how infinite domains can be finitely partitioned to make abstract descriptions of objects finite. This is achieved by introducing π -functions for components of the infinite domain.

A wide spectrum of π functions can be defined. A more detailed description of this approach can be found in [Consel 1993a]. Examples of polyvariant functions (or more generally, of contexts of analysis) include the current call chain, lexical context, and the abstract values of arguments of procedure calls (e.g., see [Consel 1993a; Harrison III 1989; Shivers 1991]). Notice that a monovariant analysis can be obtained by defining π to be a constant function mapping to a unique partition.

We now turn to describing a concrete algorithm that implements the generic analysis. In essence, the algorithm is driven by the need to analyze application variants. Besides computing the control-flow and data-flow of the program, our algorithm must construct and maintain a dependency graph. This graph is used to initiate computations as application variants are updated.

Note that, for convenience, we use π , \perp , and α both in mathematical formulas and programs.

2.1 Analyzing expressions

An application variant represents the application of a closure variant to an abstract value. The algorithm to compute the result of the application is given in Figure 2. The procedure `analyze-exp` receives as arguments an expression, a lexical environment mapping identifiers to abstract values, and the application variant being analyzed. It returns an abstract value.

`analyze-exp` dispatches on the syntactic category of the expression. A literal is mapped into the abstract value domain and a variable is looked up in the current

```

(define-record cv (exp env creators avs) ; closure variant
(define-record av (cv callers arg return) ; application variant
(define-record call-si te (av exp arg))

(define (analyze-exp exp env self)
  (variant-case exp
    (lit (value) ( $\alpha$  value))
    (varref (name) (abstract-lookup env name))
    (cond (test then else)
      (abstract-cond
        (analyze-exp test env self)
        (analyze-exp then env self)
        (analyze-exp else env self)))
    (proc (formal body)
      (let ((cv (lookup-clos-variant exp ( $\pi$  env))))
        (update-creators! (cv->creators cv) self env)
        (update-environment! cv)
        (build-absval cv)))
    (app (rator rand)
      (let ((arg (analyze-exp rand env self))
            (cvs (cvs-of-absval (analyze-exp rator env self))))
        (let ((call-si te (make-call-si te self exp arg)))
          (fold (lambda (cv acc)
                  (let ((av (lookup-app-variant cv ( $\pi$  arg)))
                        (update-callers! (av->callers av) call-si te)
                        (update-argument! av)
                        (set-cv->avs! cv (union (cv->avs cv) (make-set av)))
                        (lub (av->arg av) acc)))
                    cvs  $\perp$ )))))))

(define (update-environment! cv)
  (variant-case cv
    (cv (env creators avs)
      (let ((new-env (creators->env creators)))
        (unless (env=? env new-env)
          (set-cv->env! cv new-env)
          (push-many! workstack avs))))))

(define (update-argument! av)
  (variant-case av
    (av (arg callers)
      (let ((new-arg (callers->arg callers)))
        (unless (arg=? arg new-arg)
          (set-av->arg! av new-arg)
          (push! workstack av))))))

```

Fig. 2. Algorithm to analyze expressions

lexical environment. The abstract value of a conditional is determined by the abstract values of its subexpressions.

The analysis of lambda expressions and applications causes dependency information to be recorded. Two abstract data types (*creators* and *callers*) are used to help maintain this information. The creators abstract data type (ADT) notes in what context a closure variant can enter the data flow of the program. The callers ADT notes where a closure variant may be applied. Operations on these datatypes will be introduced as needed. Note that these ADTs simply abstract bookkeeping details and are straightforward to implement.

The analysis of a lambda expression yields a closure variant. The creators of the closure variant is updated to include the application variant being analyzed and the environment in which the analysis occurs. This is done with the `update-creators!` operation on the creators ADT.

The procedure `update-environment!` is then called to update the environment of the closure variant. Another operation, `lub-creators-env` is used to obtain the least upper bound of the environments of all the creators. By recording the environment with each creator, we are effectively recording the environment of each abstract closure in the partition named by the closure variant. The update ensures that the closure variant accurately reflects the abstract closures in its partition.

The environment of a closure variant may change by adding a new application variant to the set of creators or because the environment of a creator changed. If so, applications of the closure variant must be reanalyzed. The `avs` field of a CV record notes these dependencies, and the `push-many!` operation in `update-environment!` initiates the reanalyses by pushing the application variants onto the workstack.

To compute the abstract value of an application, the closure variants collected at an application site are applied to the abstract value of the argument one at a time. Each application yields an abstract value representing the result of the application. The least upper bound of the resulting set of abstract values is the result of the application.

A closure variant is applied to an abstract value by first finding an application variant to represent the application. Similar to `update-creators!`, the callers ADT operation `update-callers!` records where a closure variant is being applied in the context represented by the application variant.

The procedure `update-argument!` is called to update the argument of the application variant. The operation `lub-callers-arg`, analogous to `lub-creators-env`, is used to obtain the least upper bound of the arguments of all the callers. Again, by recording the argument with each caller, we are effectively recording the abstract values in the partition named by the application variant, and the update ensures that the application variant accurately reflects the values in its partition.

The argument of an application variant may change for reasons similar to the environment of a closure variant changing. If it changes, the application variant must be reanalyzed.

Finally, the application variant representing the application context is added to the `avs` field of the closure variant being applied. This updates the dependency of the application variant on the closure variant's environment.

We now continue with the algorithm for analyzing complete programs.

```

(define workstack (make-stack))

(define (analyze-prog prog)
  (variant-case prog
    (prog (defs exp)
      (push! workstack (build-root-variant exp (build-initial-env defs)))
      (letrec ((loop
                (lambda ()
                  (unless (empty-stack? workstack)
                    (let ((av (top workstack)))
                      (pop! workstack)
                      (analyze-variant av))
                    (loop))))))
        (loop))))))

(define (analyze-variant av)
  (variant-case av
    (av (cv arg)
      (variant-case cv
        (cv (exp env)
          (variant-case exp
            (proc (formal body)
              (let ((env (extend-env env formal arg)))
                (update-return-value! av (analyze-exp body env av))))))))))

(define (update-return-value! av new-return)
  (variant-case av
    (av (callers return)
      (unless (absval=? return new-return)
        (set-av->return! signature new-return)
        (push-many! workstack (callers->avs callers))))))

```

Fig. 3. Algorithm for analyzing programs

2.2 Analyzing programs

The algorithm to compute the abstract value of an entire program is given in Figure 3. The procedure `analyze-prog` takes a program and computes its abstract value. Upon termination, the argument and result for each application variant in the program will have been computed.

`analyze-prog` first creates an initial environment from the definitions of the program (`build-initial-env`) and a *root* application variant (`build-root-variant`). This variant describes the program's expression. The workstack, represented as a stack, is initialized with the variant, and the procedure then iterates until the stack is empty. Since changes are monotonic and the lattice of abstract values has a finite height, the iteration terminates in a finite number of steps.

The procedure `analyze-variant` analyzes an application variant and updates the variant's return value. It extends the lexical environment of the closure variant being applied with a binding for the argument and calls `analyze-exp`. The value returned is then used to update the variant.

```

(define f (lambda (x) (k (h (g x)))))
(define g (lambda (x) x))
(define h (lambda (x) (eta: lambda (y) x)))
(define k (lambda (f) (f 1)))

(f 0)

```

Fig. 4. An example program

Step	Application Variants
1	$\langle root, [] \rangle : \emptyset \mapsto \perp, \langle f, [] \rangle : \emptyset \mapsto \perp$
2	$\langle g, [] \rangle : \emptyset \mapsto \perp, \langle h, [] \rangle : \perp \mapsto \perp, \langle k, [] \rangle : \perp \mapsto \perp$
4	$\langle h, [] \rangle : \perp \mapsto \langle \eta, [\perp/x] \rangle$
5	$\langle k, [] \rangle : \{ \langle \eta, [\perp/x] \rangle \} \mapsto \perp$
6	$\langle \eta, [\perp/x] \rangle : \emptyset \mapsto \perp$
8	$\langle g, [] \rangle : \emptyset \mapsto \emptyset$
9	$\langle h, [] \rangle : \emptyset \mapsto \perp$
10	$\langle h, [] \rangle : \emptyset \mapsto \langle \eta, [\emptyset/x] \rangle$
11	$\langle k, [] \rangle : \{ \langle \eta, [\perp/x] \rangle, \langle \eta, [\emptyset/x] \rangle \} \mapsto \perp$
12	$\langle \eta, [\emptyset/x] \rangle : \emptyset \mapsto \emptyset$
13	$\langle k, [] \rangle : \{ \langle \eta, [\perp/x] \rangle, \langle \eta, [\emptyset/x] \rangle \} \mapsto \emptyset$
14	$\langle f, [] \rangle : \emptyset \mapsto \emptyset$
15	$\langle root, [] \rangle : \emptyset \mapsto \emptyset$

Fig. 5. Fixpoint computation for the example program of Figure 4

If the variant's return value changes, then the application variants that call the modified variant must be reanalyzed. The variants that depend on the results are obtained with `callers->avs`, a third operation on the callers abstract data type.

2.3 Evaluation

The algorithm just given can perform well when instantiated for a monovariant analysis (i.e., π is a constant function). Unfortunately, it may lose accuracy and suffer performance penalties in the presence of polyvariance. We identify the problem and then turn to solving it.

Consider the program given in Figure 4 and a trace of its analysis in Figure 5. In the example program, we have labeled the anonymous lambda expression with η for convenience. To carry the example we assume the abstract domain and π function from the beginning of the section.

The trace shows the application variants that are created or changed on each step of the analysis. Recall, each step constitutes the analysis of one application variant. Steps are omitted when there is no change to any variant on that step.

The following notation is used in the table. $\langle f, [v/a] \rangle$ denotes a closure variant obtained from the label for the procedure f and the environment in which a is mapped to the abstract value v . An application variant is denoted by $c : v_0 \mapsto v_1$, where c is the closure variant whose application is represented by the application variant, v_1 is the argument of the application, and v_2 is the result.

Now let us follow the trace. In step 1 we show the root variant as it is initially created and the result of applying the procedure f to \emptyset . On the next step the application variant for f is analyzed, resulting in the creation of three new application variants for g , h , and k .

Since the argument of an application is evaluated first, the application variant for k was pushed onto the workstack last. It is analyzed in step 3, but there is no change since there is no closure variant to be applied.

The application variant for h is analyzed next, yielding a closure variant for η . Since the result of the variant has changed, the application variant for f is analyzed, and this causes the creation of a new application variant for k , since $\pi(\perp) \neq \pi(\{\langle \eta, [\perp/x] \rangle\})$.

The analysis of the new variant for k causes an application variant for $\langle \eta, [\perp/x] \rangle$ to be created and put onto the workstack. However, its analysis causes no change to any application variants.

Finally the application variant for g is analyzed, and the result is propagated yielding a new application variant for h , since $\pi(\perp) \neq \pi(\emptyset)$. The analysis of this new variant also yields a new closure variant which is passed to k .

Step 9 is crucial. $\langle h, [] \rangle : \perp \mapsto \langle \eta, [\perp/x] \rangle$ is no longer called from $\langle f, [] \rangle : \emptyset \mapsto \perp$ since more accurate information about the argument to $\langle h, [] \rangle$ has been obtained. This also implies that $\langle k, [] \rangle$ is no longer applied to $\langle \eta, [\perp/x] \rangle$.

In step 11 we see that the least upper bound of the new variant passed to $\langle k, [] \rangle$ and the old argument have been taken to obtain a new argument for $\langle k, [] \rangle$. We have now lost precision in the control/data flow, since there are two closure variants that are part of the argument when there really should be only one.

In practice, application variants may be analyzed repeatedly after they become useless. This is because the dependency information we record is not appropriately updated when the control-flow and data-flow information changes due to polyvariance.

This example illustrates how an application variant may be created at some intermediate point in the fixpoint computation but no longer used once the fixpoint is reached. It also shows how the data flow of the program may become less precise because of values returned by intermediate variants that are not part of the final set of variants.

Intermediate variants are necessary for the fixpoint to be reached in some fixpoint computations, but the fixpoint computation can be accelerated and made more accurate by detecting when variants become useless and eliminating them from the computation.

The next section presents a revision of our algorithm intended to achieve this improvement.

3. IMPROVING THE ANALYSIS

The changes to the algorithm are illustrated in Figures 6 and 7. They aim at maintaining more accurate dependency information in the presence of polyvariance by detecting and eliminating useless variants.

Some of the changes are for bookkeeping purposes. An extra field is added to application variants to note whether or not a variant is useful. We also extend the record type for call sites to include the set of closure variants applied at a call site. Finally, a lookup operation for call sites is introduced. The lookup operation creates a new record instance if a matching instance does not exist. It otherwise returns the preexisting instance. Hence, for each application variant-expression pair, there will exist exactly one call-site record to represent it.

```

(define-record av (cv signature callers useful?)
  (define-record call-site (self exp cvs arg))

(define (analyze-variant av)
  (variant-case av
    (av (cv signature useful?)
      ...
      (let ((env (extend-env env formal arg)))
        (cond
          ((still-useful? av)
            (set-av->useful? av #t)
            (update-return-value! av (analyze-exp body env av)))
          (useful?
            (set-av->useful? av #f)
            (sweep-exp body env av)))))))

(define (analyze-exp exp env self)
  (variant-case exp
    ...
    (app (rator rand)
      (let ((arg (analyze-exp rand env self))
            (cvs (cvs-of-absval (analyze-exp rator env self)))
            (call-site (lookup-call-site self exp)))
        (update-call-site! call-site cvs arg)
        ...))))

(define (update-call-site! call-site new-cvs new-arg)
  (variant-case call-site
    (call-site (cvs arg)
      (let ((dropped-cvs
              (if (pi=? ( $\pi$  arg) ( $\pi$  new-arg))
                  (set-difference cvs new-cvs)
                  cvs)))
        (cleanup dropped-cvs call-site)
        (set-call-site->arg! call-site new-arg)
        (set-call-site->cvs! call-site new-cvs))))))

```

Fig. 6. Revised algorithm for analyzing expressions

```

(define (sweep-exp exp env self)
  (variant-case exp
    (test (pred then else)
      (sweep-exp pred env self)
      (sweep-exp then env self)
      (sweep-exp else env self))
    (proc ()
      (let ((cv (lookup-clos-variant exp ( $\pi$  env))))
        (remove-creators! cv self)
        (update-env! cv)))
    (app (rator rand)
      (let ((call-site (lookup-call-site self exp)))
        (cleanup cvs call-site)
        (set-call-site->cvs! call-site '()))
      (sweep-exp rator env self) (sweep-exp rand env self))))

(define (cleanup cvs call-site)
  (let ((pi-arg ( $\pi$  (call-site->arg call-site))))
    (for-each (lambda (cv)
      (let ((av (lookup-app-variant cv pi-arg)))
        (remove-callers! av call-site)
        (update-argument! av)))
      cvs)))

(define (still-useful? av)
  (letrec ((loop
    (lambda (avs seen)
      (cond
        ((null? avs) #f)
        ((memq (car avs) seen) (loop (cdr avs) seen))
        ((or (root-variant? (car avs)) (highest? (car avs))) #t)
        (else (loop
          (append (callers->avs (av->callers (car avs))) avs)
          (cons (car avs) seen)))))))
    (loop (list av) '())))

(define (highest? av)
  (variant-case av
    (av (cv arg)
      (andmap (lambda (av) (absval=? (lub (av->arg av) arg) arg))
        (cv->avs cv))))))

```

Fig. 7. Auxiliaries for revised algorithm

Call sites are now updated with the `update-call-site!` operation. In addition to simply updating the record fields, it determines which closure variants are either no longer being applied at the call site or else being applied in a different context of analysis. The procedure `cleanup` is called to update the affected variants.

`cleanup` considers each closure variant and determines the application variant representing the application at the call site in question. It then removes the call site from the callers of the application variant using a new ADT operation `remove-callers!` and calls `update-argument!` to record the changes to the application variant. The effect is to remove the caller’s data-flow contribution to the called variant.

These changes to the analysis already cause more accurate control-flow and data-flow information to be gathered since calls from one variant to another are erased as data-flow information becomes more precise. We now turn to removing useless variants from further analysis.

The revised procedure `analyze-variant` checks to see if a variant is useful before analyzing it (`still-useful?`). A variant is useful if it is reachable from either the root application variant (`root-variant?`) or the highest application variant (`highest?`) derived from a closure variant. If it is indeed useful, the analysis proceeds as before. If it is not useful, the variant must be swept to remove its contributions from the data- and control-flow of the analysis and marked as no longer live.

Considering a variant useful if it is reachable the highest application variant of a closure variant ensures the monotonicity of the fixpoint finding process. Without it, the algorithm may not terminate as the argument and result values in a closure variant oscillate between points in the abstract domain.

The procedure `sweep-exp` traverses the abstract syntax tree of the application variant `self` to find lambda expressions and applications. For each lambda expression, the closure variant created by `self` at that point is determined. `self` is then removed from the creators of the closure variant using the new operation `remove-creators!` and `update-env!` is called to update the environment of the closure variant.

For each application, `cleanup` is called to update the application variants that are no longer called at that call site. The call site is also reset to indicate that no closure variants are called from the site.

Implementation issues

A more efficient version of the algorithm can be easily implemented. For example, the procedure `still-useful?` is quadratic in the number of application variants in the control-flow graph. A linear version can be implemented with some extra bookkeeping.

Also, we have discovered empirically that the analysis reaches a fixpoint much more quickly if variants are analyzed somewhat “greedily.” That is, instead of putting a variant onto the workstack for analysis, it is (re)analyzed immediately unless the analysis of the variant is already pending. We also limit the number of pending analyses.

We have applied both of these improvements to the implementation used to report the results in the next section.

		ds- λ	cps- λ	Prolog	MP	Match	DB	Proof ₁	Proof ₂
	size	437	483	1183	2197	454	701	1083	3151
	functions	7	7	43	68	15	27	62	171
initial	variants	36	153	962	136	30	68	336	671
	time	3.6	29.9	196.5	16.2	2.7	4.1	116.2	290.6
revised	variants	20	103	568	133	30	36	73	305
	time	1.3	16.5	111.8	17.1	2.7	2.5	8.4	97.4
	ratio	2.8	1.8	1.8	.9	1	1.6	13.8	3
	reused	0	1	74	0	0	0	1	22

Fig. 8. Benchmarks

4. RESULTS

The revised algorithm is implemented in the binding-time analysis of the partial evaluator Schism [Consel 1993a; Consel 1993b]. The general-purpose algorithm we compare to ours was also implemented in Schism’s BTA. Figure 8 tabulates the results of analyzing eight programs using the new analyzer.

Four of the eight programs are interpreters. The first program (ds- λ) is an interpreter for an extension of the call-by-value lambda calculus. The second (cps- λ) is a continuation-passing style version of that same interpreter. The third is a Prolog interpreter (Prolog) [Consel and Khoo 1991], and the fourth is an interpreter for a subset of Algol (MP) [Consel and Danvy 1991b]. The remaining four programs are a pattern matcher for patterns expressed using a regular expression language (Match) [Consel and Danvy 1991a], an implementation of a simple database system (DB) [Daniels and Vance 1993], and two encodings of theorems (Proof₁ and Proof₂) [Lawall 1993].

For each program we report its size (measured as the number of words in the program) and its number of top-level definitions and lambda expressions.

The reported number of variants is the number of application variants that were reachable when the analysis terminated. Times are reported in seconds, and the ratio shown indicates the speedup over the initial algorithm. The number of variants reused is the number of variants that became unreachable at some point in the analysis but were called again at a later point in the analysis.

The revised algorithm shows that the strategy of eliminating useless variants generally performs well. The result of analyzing the program Proof₁ shows that it can drastically reduce the number of variants created and the time spent analyzing them.

The result of analyzing the program MP is particularly interesting. It illustrates the price of detecting and discarding useless variants. Three variants were found to be useless, but it took more time to perform the analysis. Finding and handling useless variants can be expensive. In our implementation, the principal overhead comes from determining whether or not a variant is reachable.

Eliminating useless variants can also be costly if many variants are subsequently reused, since the time invested in detecting and eliminating them was wasted. We have observed that variants are not usually reused, but in some cases, the number can be significant.

These initial benchmarks indicate that it is possible to execute accurate polyvariant analyses while keeping the overhead of polyvariance to a manageable degree.

5. RELATED WORK

Martin and Hankin [Martin and Hankin 1987] discuss practical methods for performing strictness analysis of a higher-order language. They optimize the abstract interpretation by attempting to reduce the number of function-argument pairs that have to be analyzed. Their techniques however are dedicated to two-point abstract domains.

Henglein developed an efficient constraint-based binding-time analysis [Henglein 1991] for higher-order programs. Bondorf and Jørgensen [Bondorf and Jorgensen 1994] implemented this approach in the partial evaluator Similix [Bondorf 1991]. The analysis is monovariant however, and it is unclear at this time whether or not the analysis can be made polyvariant while preserving its efficiency.

O’Keefe [O’Keefe 1987] gave an efficient fixpoint algorithm that is similar to Henglein’s in that terms are rewritten to an intermediate form better suited for analysis. His algorithm is able to find solutions to sets of equations with arbitrary lattices of finite height. The algorithm is not parameterized with respect to the degree of polyvariance, however, and it does not address the specific issues that arise in the analysis of higher-order programs.

Using a pending list and eagerly traversing the control-flow graph is a technique that has been applied elsewhere (e.g., [Charlier, Musumbu, and Hentenryck 1991]). However, it has only been used in the context of first-order programs. It also appears in the computation of minimal function graphs [Jones and Mycroft 1986], which is intended for use with first-order programs as well.

Hall and Kennedy [Hall and Kennedy 1992] describe an efficient call-graph analysis for Fortran. Procedures are reanalyzed only if their use context changes. Le Charlier *et al.* describe an algorithm for analyzing Prolog programs [Charlier, Musumbu, and Hentenryck 1991]. Both of these strategies reanalyze equations by need as does ours.

Several other static analyzers for higher-order languages appear in the literature (e.g., [Harrison III 1989; Shivers 1991; Young 1989]). None of these works address parameterized polyvariance, nor do they make any attempt to optimize the analysis process (e.g., reanalyzing equations by-need).

6. CONCLUSIONS

We have presented a general-purpose algorithm for computing fixpoints in the polyvariant static analysis of higher-order applicative programs. Other algorithms have been presented previously that compute polyvariant analyses, but to date, none have attempted to optimize the efficiency of the analysis.

The language considered for analysis cannot express data structures, but in fact, data structures can be added to the language without significantly complicating the analysis. They are treated exactly like closures from the perspective of implementing the algorithm [Consel 1990; Consel 1993a]. Data variants are associated with partitions of abstract data structures. The construction of an abstract data structure corresponds to the creation of an abstract closure. The access of a data structure corresponds to the application of an abstract closure. Also analogous to closure variants, data variants must be removed from the data-flow of the program when they are no longer referenced.

Future work includes formalizing the algorithm and proving its correctness. The crucial component of the proof will be showing that the arguments and return values of application variants stabilize despite removing closure variants from them. We also plan to undertake a complexity analysis to assess the cost of polyvariance.

We hope to extend this approach to side-effects. One direction is to follow Shivers's strategy [Shivers 1991]. This strategy should not involve any major modifications to our analysis. We are studying other strategies that could be cast in our framework as well.

ACKNOWLEDGMENTS

We thank Olivier Danvy and Karoline Malmkjær for their careful reading of drafts of this paper.

REFERENCES

- Abramsky, S. and Hankin, C. (Eds.) 1987. *Abstract Interpretation of Declarative Languages*. Ellis Horwood.
- Bondorf, A. 1991. Automatic autoprojection of higher-order recursive equations. *Science of Computer Programming* 17, 3–34.
- Bondorf, A. and Jorgensen, J. 1994. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming, special issue on Partial Evaluation* 3, 3 (July).
- Charlier, B. L., Musumbu, K., and Hentenryck, P. V. 1991. A generic abstract interpretation algorithm and its complexity analysis. In K. Furukawa (Ed.), *Proceedings of the Eighth International Conference on Logic Programming, Paris, France*, pp. 64–78. MIT Press, Cambridge, Massachusetts.
- Clinger, W. and (editors), J. R. 1991. Revised⁴ report on the algorithmic language Scheme. *LISP Pointers IV*, 3 (July-September), 1–55.
- Consel, C. 1990. Binding time analysis for higher order untyped functional languages. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pp. 264–272.
- Consel, C. 1993a. Polyvariant binding-time analysis for higher-order, applicative languages. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp. 66–77.
- Consel, C. 1993b. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp. 145–154.
- Consel, C. and Danvy, O. 1991a. For a better support of static data flow. In *FPCA'91, 5th International Conference on Functional Programming Languages and Computer Architecture*, pp. 496–519. Springer-Verlag.
- Consel, C. and Danvy, O. 1991b. Static and dynamic semantics processing. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pp. 14–23.
- Consel, C. and Danvy, O. 1993. Tutorial notes on partial evaluation. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 493–501.
- Consel, C. and Khoo, S. C. 1991. Semantics-directed generation of a Prolog compiler. In *PLILP'91, 3rd International Symposium on Programming Language Implementation and Logic Programming*. Springer-Verlag.
- Cousot, P. 1981. Semantic foundations of program analysis: Theory and applications. In S. S. Muchnick and N. D. Jones (Eds.), *Program Flow Analysis: Theory and Applications*. Prentice-Hall.
- Daniels, S. D. and Vance, B. 1993. Partial evaluation of a simple database system. Technical report, Oregon Graduate Institute, Beaverton, Oregon, USA. (Student Report).
- Dybvig, R. K. 1987. *The Scheme Programming Language*. Prentice-Hall.
- Hall, M. W. and Kennedy, K. 1992. Efficient call graph analysis. *Letters on Programming Languages and Systems* 1, 3.
- Harrison III, W. L. 1989. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation* 2, 3/4, 179–396.

- Henglein, F. 1991. Efficient type inference for higher-order binding-time analysis. In *FPCA'91, 5th International Conference on Functional Programming Languages and Computer Architecture*, pp. 448–472. Springer-Verlag.
- Jones, N. D. and Mycroft, A. 1986. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*.
- Jones, N. D. and Nielson, F. 1990. Abstract interpretation: a semantics-based tool for program analysis. Technical report, University of Copenhagen and Aarhus University, Copenhagen, Denmark.
- Jones, N. D., Sestoft, P., and Søndergaard, H. 1989. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation* 2, 1, 9–50.
- Lawall, J. L. 1993. Proofs by structural induction using partial evaluation. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp. 155–166.
- Martin, C. and Hankin, C. 1987. Finding fixed points in finite lattices. In *Functional Programming Languages and Computer Architecture*, Volume 274 of *Lecture Notes in Computer Science*, pp. 426–445. Springer-Verlag.
- Nielson, F. and Nielson, H. R. 1992. *Two-Level Functional Languages*, Volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.
- O’Keefe, R. A. 1987. Finite fixed-point problems. In J.-L. Lassez (Ed.), *Proceedings of the Fourth International Conference on Logic Programming, Melbourne, Australia*, pp. 729–743. MIT Press, Cambridge, Massachusetts.
- Rytz, B. and Gengler, M. 1992. A polyvariant binding time analysis. In C. Consel (Ed.), *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 21–28. Yale University. Research Report 909.
- Sestoft, P. 1989. Replacing function parameters by global variables. In *FPCA'89, 4th International Conference on Functional Programming Languages and Computer Architecture*, pp. 39–53.
- Shivers, O. 1991. The semantics of Scheme control-flow analysis. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, New Haven, Connecticut.
- Young, J. H. 1989. The theory and practice: Semantic program analysis for higher-order programming languages. Ph. D. thesis, Yale University. YALEU/DCS/RR-669.