Fast Interprocedural Alias Analysis

Keith D. Cooper
Ken Kennedy

Department of Computer Science†
Rice University
Houston, Texas 77251-1892

Abstract

We present a new algorithm for computing interprocedural aliases due to passing parameters by reference. This algorithm runs in $\mathbf{O}(N^2 + NE)$ time and, when combined with algorithms for alias-free, flow-insensitive data-flow problems, yields algorithms for solution of the general flow-insensitive problems that also run in $\mathbf{O}(N^2 + NE)$ time.

1. Introduction

Interprocedural analysis of the side effects of subroutine invocation has been widely discussed in the literature [Spil 71, Alle 74, Bart 78, Bann 79, Rose 79, Myer 80, Burk 84, CoKe 84, BuCy 86, CaRy 86, CoKe 87, Call 88, CoKe 88, HoRB 88]. Banning first suggested the approach of decomposing interprocedural data-flow problems into two subcomponents: the problem of analyzing potential aliases and the *alias-free* data-flow analysis problem [Bann 79]. For many important data-flow analysis problems, these two subproblems can be solved independently and the results combined later to produce a general solution. Cooper and Kennedy have shown that alias-free flow-insensitive problems[1] can be solved in $\mathbf{O}(N + E)$ bit vector steps of $\mathbf{O}(N)$ length, where N and E are the number of nodes and edges of the program's call graph, respectively. Thus, the overall running time of the algorithm is $\mathbf{O}(N^2 + NE)$ [CoKe 88].

The best previous algorithm for single-variable alias analysis requires at least $\mathbf{O}(N^3 + N^2E)$ time [Coop 85]. Hence, the time to solve the alias problem dominates the computation of the complete solution. Furthermore, once aliases are determined, there still remains the problem of integrating them with the solution of the alias-free problem to produce a general result. A naive solution to this problem can easily take $\mathbf{O}(N^2E)$ steps.

In this paper we present a new algorithm for alias analysis that requires $\mathbf{O}(N^2 + NE)$ steps, along with a method for integrating the solution with the solution to the alias-free problem in $\mathbf{O}(NE)$ steps. It builds on the fundamental insight, developed in our previous work, that significant advantages can be achieved by separating the treatment of reference formal parameters from the treatment of global variables. Combining this result with our previous work yields a method for precise solution of flow-insensitive summary problems in time proportional to the size of the call graph times the number of variables in the program.

This paper divides into nine sections. Section 2 introduces the problem of interprocedural alias analysis and shows how it relates to a particular interprocedural summary problem. Section 3 introduces the assumptions we make about the relationship of certain problem parameters to the size of the call graph. Section 4 describes the graphical representation that the algorithm will use, the *binding graph*. Section 5, we present the algorithm for languages with a simple two-level nesting structure. Section 6 sketches how to extend the method to languages with general procedure nesting. Section 7 discusses practical implementation details. Section 8 proposes several areas for future work. Finally, section 9 contains a summary of the results.

---

† This work has been supported by the National Science Foundation through grants CCR 86-19893, CCR 87-06229, and ASC 85-18578 and by IBM Corporation.

[1] Some researchers use the terms ''may'' and ''must following Barth [Bart 78]. However, we prefer Banning's classification of side-effects as ''flow-insensitive'' and ''flow-sensitive'' because the terminology reflects the precise definition that he provided for this distinction [Bann 79].

2. The Problem

Whenever a program can reference a single storage location using two different variable names, those names are said to be *aliases*. In the following example, formal parameters f3 and f4 must be considered aliases on entry to q, since the call at s4 passes y to both of them.

```
global x, y, z
    ...
    procedure p(f1, f2)
    s1:   call q(f1, z)
    s2:   call q(f2, f1)
    end
    ...
    procedure q(f3, f4)
       ...
    end
    ...
s3: call p(x, z)
s4: call q(y, y)
```

The goal of alias analysis is to determine, for each variable v that is visible in procedure p, the set of variables that may be aliases of v in some invocation of p.

Throughout this paper, we will use the interprocedural MOD problem as our example. In that problem, one wishes to determine, at each call site s, the set of variables MOD(s) that may be modified as a side effect of the call. Banning observed that the treatment can be simplified by first computing DMOD(s), the set of variables that may be modified by execution of s, ignoring any aliasing effects in the procedure containing s, and factoring aliasing in later. In other words, MOD(s) can be computed by adding to DMOD(s) any variable that may be aliased to a member of DMOD(s). For the purposes of this paper, we will only consider aliases that are generated through the use of reference formal parameters to procedures.

Hence, the problem is to compute, for each variable v and procedure p in the program, a set ALIAS(v, p) of the variables visible in p that may be aliased to v within p. Then for a call site s in p, we can compute MOD(s) by the equation

$$\text{MOD}(s) = \text{DMOD}(s) \cup \bigcup_{v \in \text{DMOD}(s)} \text{ALIAS}(v, p) . \quad (1)$$

From the naive point of view, if we assume that each DMOD set is of length $\mathbf{O}(N)$ and each ALIAS set is of length $\mathbf{O}(N)$, this operation will require $\mathbf{O}(N^2)$ time at each of the E call sites in the program or at least N^2E time overall. Thus, one of the challenges of producing a faster algorithm is to reduce the time required for this operation.

We assume a simple language that supports recursion and in which parameters are passed to procedures by reference. Procedures may be declared within other procedures and scoping is static — that is, code within a procedure may reference any variable that is known in the procedure that contains its declaration.[2] Thus, the language resembles Pascal with simple variables and call-by-reference binding and the analysis is relevant to both Fortran and C.

3. Problem Parameters

It will be critical in the analysis that follows to determine which problem parameters grow as the size of the total program grows and which remain relatively constant. In a program consisting of many procedures, we will make the following assumptions:

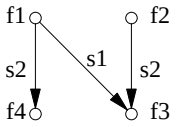
1) The number of formal parameters to any procedure is relatively constant and bounded from above by the constant c_P.
2) The maximum depth at which a procedure is nested in the program is a constant d_P.
3) The total number of variables in the program is proportional to the number of procedures N.

4. The Binding Graph

This method uses the *binding graph* introduced in our previous work on alias-free flow-insensitive summary problems [CoKe 88]. It is a natural adaptation of the scheme used in Torczon's algorithm for interprocedural constant propagation [Torc 85, CCKT 86]. The program's binding graph, $\beta = (N_\beta, E_\beta)$, represents interactions between formal parameters. Nodes in N_β uniquely represent the formal parameters of the various procedures in the program. Edges in E_β represent individual *binding events*. If p calls q from some call site s and f_1 gets bound to f_7 at s, then there is an edge $(f_1, f_7) \in E_\beta$. Since p can call q several times, binding f_1 to f_7 at each call site, β may be a multi-graph. Because β reflects the pattern of binding chains in the program, it will almost certainly consist of a number of disjoint components.

The binding graph for the program fragment shown in Section 2 looks like:

[2] We assume, without loss of generality, that each variable is mentioned in only one declaration.



β for example program

We will use this example throughout the paper.

How large is β? Since the complexity of data-flow algorithms is usually stated in terms of the size of the underlying graph, this issue is crucial to our later complexity analysis.[3] The important comparison is to the program's call graph, $C = (N, E)$. C contains a node for each procedure and an edge for each call site. Since we have assumed that no procedure in the program takes more than c_P formal parameters, where c_P is a constant independent of program size, we have $N_\beta \leq c_P\, N$ and $E_\beta \leq c_P\, E$.

The binding graph can be constructed in time linearly proportional to its size by simply visiting each of the call sites in the original call graph. The construction need not represent a node unless it is the endpoint of an edge in E_β. Thus, $2E_\beta \geq N_\beta$, everywhere.

5. Algorithm Overview

The important insight in the algorithm is that significant savings can be achieved if we treat formal parameters and global variables separately. Although the total number of formal parameters in the program can be as large as $\mathbf{O}(N)$, the number visible from any particular point in a program is bounded by $c_P d_P$, a constant. On the other hand, there may be $\mathbf{O}(N)$ global variables visible from any point in the program.

Now consider the alias sets. If v is a variable that is global to procedure p and is not a formal parameter to any procedure in which p is nested, ALIAS(v, p), its alias set within p, can consist *only of formal parameters visible from* p. Hence, its alias set can have no more than a constant ($c_P d_P$) number of members. This is because each global variable is mapped to a particular location in storage and two global variables may not be mapped to the same location unless one of them is a formal parameter to which the other is passed through a series of calls (or both are formal parameters to which the same location is passed). On the other hand, if v is a formal parameter, its alias set can contain any global variable. Hence it can be as large as $\mathbf{O}(N)$. We will take significant advantage of this observation in designing the algorithm.

[3] We use set names like N_β to name both the set and its cardinality The meaning should be clear from the context.

The algorithm has four distinct phases.

1) Solve a forward data-flow problem over β to compute A(f) for each formal f. A(f) is the set of variables v such that v can be aliased to f by virtue of a call chain that binds v to f.
2) Find all pairs of formal parameters that can be aliased to each other. This involves constructing a pair-wise analogue of β, called π, and using a marking algorithm to propagate the pairs of formals over π.
3) Use the solutions from steps 1) and 2) to compute precise ALIAS sets.
4) Combine the precise ALIAS sets with the results of an alias-free summary analysis to derive the solution for the general summary problem.

The following subsections explain the individual phases in greater detail.

5.1. Two-Level Model

For simplicity, we describe the algorithm for the case of a two-level nesting hierarchy where only global and local variables are permitted, as in Fortran or C. The extension of this method to languages that permit general nested procedure declarations will be discussed later.

Assume that some number of variables can be declared local to each routine and some number can be declared global to the whole program. A local variable is visible only in the routine where it is declared, while global variables are visible everywhere. Formal parameters are local variables that are bound to addresses dynamically by reference parameter passing.

In the discussion that follows, we will assume that all alias sets are represented by bit vectors of the appropriate length, either $\mathbf{O}(N)$ or $c_P d_P$. In this model, inserting a single element into a set takes constant time, as does testing a set to determine if it contains some specific element. Set initialization, set union and set intersection take time proportional to the size of the largest possible set. In some cases this will be a constant; in others it will be $\mathbf{O}(N)$.

5.2. Phase 1: Global Aliases

For each formal f in the program, let A(f) be the set of variables v that may be aliased to f at its instantiation by virtue of a call chain that passes v to f. Note that v must be visible at the instantiation of f — that is, v must be visible in the procedure to which f is a formal parameter.

```
                /* Reduce to a directed acyclic graph */
1               find the strongly connected components of β;
2               for each strongly connected component c do
3                   replace c with a representer node n;

                /* Initialize */
4               for each node x in the reduced binding graph do
5                   A(x) := ∅;
6               for each call site s do
7                   for each global variable v passed to formal f at s do
8                       A(f) := A(f) ∪ {v};

                /* Traverse the binding graph */
9               for each node f in β in forward topological order do
10                  A(f) := A(f) ∪ ⋃_{(g,f)∈E_β} A(g) ∩ GLOBAL;

                /* Set values for nodes in a cycle */
11              for each strongly-connected component c do
12                  for each vertex f ∈ c do begin
13                      let n be the representer node for c;
14                      A(f) := A(n);
15                  end
```

Figure 1 - Global Alias Computation

To compute A(v), we make a forward pass over the binding graph according to the algorithm shown in Figure 1. As we observed in our previous work, the binding graph can be reduced to an acyclic graph, by simply treating all the variables in a cycle as the same variable, and shrinking the cycle to a single node. Once the graph is so reduced, the algorithm initializes the globals that can be passed to each formal, then propagates these sets forward along the binding graph, taking unions when two paths merge. When the computation is complete, each node that was part of a cycle in the original binding graph inherits the A set from its representer.

The running time of this phase is dominated by the loops at lines 4, 6 and 9. The loop at line 4 performs $\mathbf{O}(N)$ bit-vector operations of length $\mathbf{O}(N)$. The nested loop at line 6 performs $\mathbf{O}(N^2)$ single element insertions, a constant-time operation. The loop at line 9 performs $\mathbf{O}(E)$ bit-vector operations, one for each edge in the binding graph, of length $\mathbf{O}(N)$. Hence, the entire phase takes $\mathbf{O}(N^2 + NE)$ time.

Since a global variable can only become an alias by being passed to a formal parameter of a procedure in which the global is visible, A(f) contains all the global variables that can be aliased to f. Thus, we can now compute the ALIAS sets for each global variable in the program by a simple inversion.

```
1   for each procedure p do begin
2     for each g ∈ GLOBAL do
3       ALIAS(g,p) := ∅;
4     for each formal parameter f of p do
5       for each g ∈ A(f) do
6         ALIAS(g,p):= ALIAS(g,p) ∪ { f };
7   end
```

The loop at line 1 is iterated N times and the loop at line 2 is iterated $\mathbf{O}(N)$ times but, since each of the alias sets for a global variable in a particular procedure can have no more than c_P elements, the overall cost of line 3 is only $\mathbf{O}(N^2)$ operations. The loop on line 4 is iterated c_P times and the one on line 5 is iterated no more than $\mathbf{O}(N)$ times, so the overall cost of this loop and the entire inversion is $\mathbf{O}(N^2)$.

Consider applying Phase 1 to the example from Section 2. Because β is acyclic, the reduced graph is just β. The loop at line 7 will find the following bindings:

call site	s1	s2	s3	s4
bindings	z→f4	—	x→f1 z→f2	y→f3 y→f4

The propagation step (the loop at line 9) computes the A sets for each formal parameter.

	f1	f2	f3	f4
A(f)	x	z	x,y,z	x,y,z

Applying the inversion procedure produces correct ALIAS sets for the global variables.

ALIAS	x	y	z
p	f1	—	f2
q	f3, f4	f3, f4	f3, f4

5.3. Phase 2: Formal Pair Analysis

Because formal parameters can become aliased by several different mechanisms, the computation of aliases for formal parameters is much more subtle. Two formal parameters can be aliased to one another if a single variable is passed to both formals at some call site. Similarly, if a call site passes a global variable to one formal parameter and one of its aliases to another, the call site creates an alias between the corresponding formal parameters of the called procedure. Furthermore, formal parameter alias pairs can be passed from routine to routine if the aliased formal parameters are both passed at the same call site.

To analyze this situation, we must do two things: determine the points at which two formals become aliased to one another and trace the propagation of alias pairs through the call chains.

To compute the points where pairs of formal parameters become aliases, we note that aliasing is introduced in two ways:

1) the same variable is passed in two different actual parameter positions at a call site, or
2) a formal parameter f_0 of routine p is passed to formal parameter f_1 of procedure q and some global $g \in \text{A}(f_0)$ is passed to formal parameter f_2 of q at the same call site.

The algorithm shown in Figure 2 finds all pairs of parameters (f_1, f_2) to the same procedure that may become aliases in either of these fashions. Since there are at most a constant number of parameters at each of the E call sites in the program and testing for membership is a constant-time operation on bit vectors, this operation takes $\mathbf{O}(E)$ time.

To analyze the propagation effects, we make use of an analogue of the binding graph called the *pair binding graph* $\pi = (N_\pi, E_\pi)$. In this new graph, each vertex is a pair of formal parameters to the same procedure and each edge represents a possible mapping of an alias pair in one procedure to an alias pair of another procedure called within it. This graph is needed to insure that we determine pairs of parameters that can be mapped to the same location along the same path through the call graph.

Since there are at most c_P parameters to each of the N procedures in a program, there are fewer than $c_P^2 N$ or $\mathbf{O}(N)$ vertices in the pair binding graph. Furthermore, since each pair of vertices can map to no more than c_P^2 different pairs at a given call site, there must be fewer than $c_P^4 E$ or $\mathbf{O}(E)$ edges in the pair binding graph. Hence, the pair binding graph is only a constant factor larger than the call graph. In practice, we expect that it will remain fairly small.

We associate with each node in the pair binding graph a mark which indicates whether or not the pair can be aliases. The marks are initially *false*, but some are set to true by the alias introduction algorithm of

```
1     for each call site s do begin;
2         if variable x is passed to two different formal parameters
3             f1 and f2
4         then mark (f1, f2) an initial alias pair;
7         else if f0 is a formal parameter of the procedure p
              containing s, and f0 is passed to formal f1
              of procedure q called at s
8         then
9             for each parameter x passed to another formal
                  parameter f2 at the same call site do
                  if x∈ALIAS(f0) then
12                    mark (f1, f2) an initial alias pair;
13    end
```

Figure 2 - Marking Parameters on π

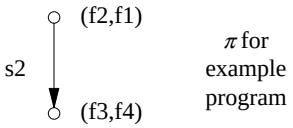
Figure 2. Now the marks are propagated by a simple bit-pushing algorithm.

```
1   let the worklist W initially contain all
      marked nodes as determined by the
      alias introduction routine above;
2   while W ≠ ∅ do begin
3     select and remove a pair p from W;
4     for each edge (p, q) ∈ E_π do
5       if q is unmarked do begin
6         mark q;
7         W := W ∪ {q};
8       end
9   end
```

When the algorithm halts, every pair of formal parameters that can be aliased to one another is marked. The pair binding graph is needed in this algorithm because two formal parameters can be aliased to one another only if they are mapped to the same location *along the same path* through the call graph. The simple binding graph doesn't contain sufficient information to trace these propagations.

Since each node can be put on the worklist at most once and each edge can be examined at most once, this phase takes **O($N + E$) steps.**

Returning to our example program fragment, its pair binding graph looks like:



π for example program

Applying the marking algorithm to this graph yields only one marked node, (f3,f4). It is discovered by the test at line 2 applied to call site s4. The bit-pushing algorithm removes the sole marked node from the worklist, discovers that it has no outward edges, and halts.

5.4. Phase 3: Formal Parameter Alias Sets

The set A(f_1) is an approximation to the set of possible aliases to a given formal parameter f_1. It contains those global variables that may be directly mapped to a formal along some path through the call graph. We claim this comprises all the global variables that can be aliased to f_1, because mapping into a formal parameter via a chain of reference parameter passing at call sites is the only way for a global to become an alias of a formal parameter.

Therefore, the only other candidates for aliases of f_1 are other formal parameters of the same procedure. To compute the correct alias sets we must add each formal f_2 that may be aliased to f_1, as determined by the marking algorithm of the previous section. This is accomplished by the following algorithm.

```
1   for each procedure p do
2     for each formal parameter f1 of p
        do begin
3       ALIAS(f1, p) := A(f1);
4       for each formal parameter f2 of p
          that may be an alias of f1
        do
5         ALIAS(f1, p) = ALIAS(f1, p) ∪ {f2};
6     end
```

Since there are N procedures, each with at most c_P parameters, and since the alias sets for formal parameters are of size **O**(N), the complexity of this phase is **O**(N^2).

Applying this algorithm to the sets for our example program, we find that it initializes each formal's ALIAS set to its A set. The loop at line 4 adds f4 to ALIAS(f3) and f3 to ALIAS(f4). (Recall that Phase 2 found only that one pair alias, between f3 and f4.) This yields the correct ALIAS sets for all the formal parameters in the program.

ALIAS	f1	f2	f3	f4
p	x	z	—	—
q	—	—	x,y,z,f4	x,y,z,f3

5.5. Phase 4: Integration

Recall that we have alias sets of two sizes. If x is a global variable, ALIAS(x, p) can only contain formal parameters of p and hence need not be larger than c_P bits. If x is a formal parameter, ALIAS(x, p) can contain any global and the bit vector representation will have length **O**(N).

We capitalize on this fact in the algorithm to integrate the ALIAS and DMOD sets, shown in Figure 3. The complexity of this phase is determined by the loops at lines 4 and 8. The loop at line 4 performs at most c_P operations of length **O**(N), while the loop at line 8 performs up to **O**(N) operations of length c_P. Hence the overall complexity of this phase is **O**(NE).

To apply this algorithm to our example, we must add a side-effect to procedure q. Assume that q modifies its formal parameter f4, so GMOD(q) is {f4}. This results in the following sets:

	s1	s2	s3	s4
DMOD	z	f1	x,z	y
MOD	z,f2	f1,x	x,z	y

The loop at line 4 adds x to MOD(s2), while the loop at line 8 adds f2 to MOD(s1).

```
1     for each call site s in the program do begin
2         let p be the procedure containing s;
3         MOD(s) := DMOD(s);
4         for each formal parameter f of p in DMOD(s) do
5             MOD(s) = MOD(s) ∪ ALIAS(f, p);
6         let tempA be a bit string of length equal to the number of
              formal parameters of p;
7         tempA := ∅;
8         for each global variable x in DMOD(s) do
9             tempA := tempA ∪ ALIAS(x, p);
10        expand tempA to a bit string of full length by inserting zeros
              in the positions not represented in the short form;
11        MOD(s) := MOD(s) ∪ tempA
12    end
```

Figure 3 - Integrating ALIAS and DMOD

Since the complexity of each phase of the algorithm is less than or equal to $\mathbf{O}(N^2 + NE)$, this is a bound for the entire algorithm.

6. General Nesting

Let us now reexamine the algorithm under the assumption of general nesting. We must show how each step of the program can be extended to handle nesting levels. There are two principal problems introduced by nesting.

1) Formal parameters of a routine defined at level k can be globals to a procedure at level $k+1$. This complicates the construction of the binding graph and the pair binding graph, makes it more difficult to identify alias introduction points and causes more work in updating the alias sets after pair propagation.

2) Variables defined at level k cannot be global to a procedure defined at level $k-1$. This will force a revision of Phase 1, which propagates global variables along the binding graph, because a variable x defined at level k cannot be aliased to any parameter of a procedure defined at level $k-1$.

In the discussion below we will make use of one important insight regarding the number of formal parameters visible from any procedure in the program. Although the total number of formal parameters in a program can be $\mathbf{O}(N)$, from a given procedure p, the only parameters that are visible are the parameters of p itself and parameters of procedures in which it is nested. The total number of procedures that can be nested is d_P (not counting the main procedure, which has no formal parameters). Since we have assumed that d_P is a constant independent of program size, the total number of formal parameters visible from any procedure is also bounded by a constant, $d_P c_P$. We will sometimes refer to the entire collection of parameters visible from p as the *extended formal parameter set* of p.

6.1. Phase 1: Global Aliases

The problem of constructing a binding graph in the presence of nesting was discussed in connection with our previous work on flow-insensitive side-effect analysis [CoKe 88]. The same construction will work for alias propagation as well. Therefore, let us concentrate on the problem of propagation.

A naive way to address the problem of nesting levels in this phase is to solve for global aliases by levels. Begin by applying the algorithm in Figure 1 of Section 5.2, taking as global only those variables declared at nesting level 0. Next, delete all edges (f, g) where g is a parameter of a procedure at level 0 from the binding graph and apply the algorithm in Figure 1, taking variables declared at level 1 as global and adding these to the A sets for formal parameters computed for the previous level. Continue in this manner until all nesting levels have been processed. Since the total number of passes will be at most d_P and each pass can take no more than $\mathbf{O}(N^2 + NE)$ time, the entire process takes $\mathbf{O}(N^2 + NE)$ time.

In practice, reconstructing and reanalyzing the binding graph d_P times can be avoided by noticing that problems only arise when cycles occur in a graph. If there are no cycles, the algorithm in Figure 1 can be slightly modified to handle the problem. If f is a formal parameter of procedure p, we define ABOVE(f) to be the set of all variables that are declared at a nesting level equal to or greater than the nesting level of p. Then we can replace line 10 of the first algorithm of section 5.2, with the statement

$$\text{A}(f) := \text{A}(f) \cup \bigcup_{(g,f)\in E_\beta} \text{A}(g) \cap \text{ABOVE}(f);$$

This prevents variables that are not visible from within the procedure p from being added to the alias set for f.

In this framework, a cycle can be handled by processing it when its representer would be processed in line 10 of the algorithm in Figure 1. Instead of applying line 10, expand the representer node to the full cycle and perform a level-by-level analysis on the subgraph to get the A sets for each formal in the cycle. Then continue with the algorithm normally. This should be significantly more efficient than reconstructing the entire binding graph on each pass.

Once the A sets are computed, the alias sets for global variables can be computed by an inversion that differs from the one in Section 5.2 in that any member of the extended formal parameter set for a procedure may be aliased to a global in that procedure. If d_P is a constant, as we have assumed, the total number of extended parameters is still a constant $c_P d_P$. Suppose we define GLOBAL(p) to be the set of variables that are global to procedure p and not formal parameters of any procedure in which p is nested. Then the following modified inversion routine can be used.

```
1   for each procedure p do begin
2     for each g ∈ GLOBAL(p) do
3       ALIAS(g,p) := ∅;
4     for each f in the extended formal
        parameter set of p do
5       for each g ∈ A(f) do
6         ALIAS(g,p):= ALIAS(g,p) ∪ {f};
7   end
```

By analogy with the analysis of this algorithm in Section 5.2, the entire inversion requires $\mathbf{O}(N^2)$ time because no more than $c_P d_P$ formals, the maximum number in the extended formal parameter set, can be aliased to a global variable in a given procedure p. Hence the same time bound holds for the modified algorithm.

6.2. Phase 2: Formal Pair Analysis

To handle formal pair analysis, the pair binding graph must be correctly constructed in the presence of procedure nesting. Using a direct analogue of the trick for constructing the binding graph with nesting, we view formal parameters of procedures in which the declaration of p is nested as an extended set of formal parameters of p. Suppose there is a procedure q that contains procedure p. If there are two parameters of q, say f_1 and f_2 and a call site in p where f_1 and f_3 are passed to f_4 and f_5, we put an edge in the binding graph between (f_1, f_2) and (f_4, f_5) if $f_2 \in \text{A}(f_3)$. This is illustrated by the following example.

```
procedure q(x,y);
  procedure p(z);
    ...
    call s(x,z);
    ...
  end
  procedure s(a,b);
    ...
  end
  ...
  call p(y);
end
```

In addition to the obvious edge from (x,z) to (a,b), we would also construct an edge from (x,y) to (a,b) because z is an alias of y at the call site to s.

With this construction of the pair binding graph, the algorithm in Figure 2 of Section 5.3 will work correctly if line 7 is modified to consider extended formal parameters. The second algorithm in Section 5.3 works without change. Both retain their time bounds.

6.3. Phase 3: Formal Parameter Alias Sets

In the algorithm of section 5.4, the only formal parameters that could be aliases of one another were the parameters of the same routine. Hence, the simple marking algorithm of section 5.4 identified all of the possibilities. With nesting, however, it is also possible that two parameters of different routines, one nested within the other, can be aliased to one another. The only way this can happen is for there to be a pair of formals, f_1 and f_2, of some procedure p aliased to one another (this would be discovered by the pair computation) and a third parameter f_3 to a procedure q nested within p such that $f_2 \in \text{A}(f_3)$. Then f_1 and f_3 are potential aliases in q. This principle is illustrated in the following example.

```
procedure p(x,y);
  procedure q(z);
    ...
  end
  ...
  call q(y);
  ...
end
```

If (x,y) is marked as an alias pair, we must add x to the alias set for z in q, and vice versa, since y is passed to z. Note that y and z would be recognized as potential aliases in Phase 1.

Thus, to correctly compute the alias sets for formal parameters, we must use the modified procedure shown in Figure 4. The success of this algorithm

```
1       for each procedure p do
        /* Copy alias sets from above */
2           for each formal parameter f global to p
                and visible within it do
3               ALIAS(f, p) := ALIAS(f, q),
                    where q is the procedure in which p is declared;

            /* Process parameters of the current procedure */
4           for each formal parameter f1 of p do begin
5               ALIAS(f1,p) = A(f1);
6               for each formal parameter f2 of p
                    or of any procedure in which p is nested
                    such that f2 may be an alias of f1 or
                    of some formal in A(f1)
7               do begin
8                   ALIAS(f1,p) := ALIAS(f1,p) ∪ {f2};
9                   ALIAS(f2,p) := ALIAS(f2,p) ∪ {f1};
10              end
11          end
```

Figure 4 - Modified ALIAS Computation

depends on visiting the procedures of the program in an outside-in order. That is, procedure p is only visited after procedures in which it is nested have been visited. This insures that the assignment in line 3 gets the right versions of alias sets for global parameters.

To see the correctness of this algorithm, recall that Phase 1 correctly computes the alias sets for global variables. In addition, if f is a formal parameter of procedure p and g is a parameter of some procedure in which p is nested and g is passed to f by a series of calls, A(f) contains g. The only aliases of f missing from A(f) are those that arise from pair binding—a single variable is passed to two different formal parameters which are passed through a sequence of formal parameter pairs to p or a routine in which p is nested. Any pair of formal parameters for which this is true will be marked by Phase 2. Now all that remains is to add to the alias set for f all formal parameters that can be aliased to f because of pair binding. This can happen if f itself is a member of a pair or some member of a pair visible in p is passed to f by a sequence of calls. In either case, the other member of the pair should be added to ALIAS(f, p). In addition, a symmetric addition should be performed, as it is in line 9.

Next, we consider the complexity of the algorithm in Figure 4. Since there are $c_P d_P$ parameters visible from any procedure, line 3 is executed at most $c_P d_P N$ times and contributes no more than $\mathbf{O}(N^2)$ to the time bound. A similar analysis bounds the cost of line 5. The test in line 6 is entered at most $c_P N$ times and can be implemented by searching through all the pairs of aliased formals that are visible within p—there are no more than $c_P^2 d_P^2$ of these—to see if one member of the pair is in A(f_1) or is equal to f_1. Thus, the test takes at most constant time and the updates in lines 8 and 9 take **O(N) time. Therefore, the overall cost of this phase is O(N^2).**

6.4. Phase 4: Integration

The integration algorithm can be run unchanged if the test in line 4 of the algorithm in Figure 3 (Section 5.5) considers extended formal parameters. That is, look for parameters of the current procedure and any procedure it is nested within. The time bounds still hold because there are at most $c_P d_P$ of these and the alias set for any global variable that is not a formal parameter can have no more than $c_P d_P$ elements in any given procedure.

7. Implementation Considerations

Phase 2 can be implemented without ever constructing the pair binding graph itself. Suppose we associate with each procedure p a set PAIRS(p). Initially, all these sets are empty. Whenever the alias introduction calculation indicates that a pair of p's formal parameters may be aliased, we add that pair to a worklist. We then extract elements from the worklist one at a time. Whenever we extract a pair (f_1, f_2) of parameters to p from the worklist, we check to see if it is already in PAIRS(p). If not, we insert it and visit every call site in the body of p, including those of procedures nested in p. If one of the parameters in the pair, say f_1, is passed to parameter f_4 of procedure q at that call site and

another parameter f_3 of some procedure within p is passed to f_5 at that call site, then we add (f_4, f_5) to the worklist if $f_2 = f_3$ or $f_2 \in A(f_3)$. Since we only visit call sites when we have a new pair, we visit each call site no more than a constant number of times, because there can never be more than $d_P c_P$ pairs that can affect any call site. As a result no pair can be put on the worklist more than $\mathbf{O}(E + N)$ times and the time bound follows.

The same trick can be used to implement Phase 1 without building the binding graph. However, since the lattice of values being propagated is not bounded, the time bound will not hold. Nevertheless, this may be effective in practice because the only problem arises in cycles, which are likely to be infrequent and small when they do occur.

8. Future Work

In the course of developing this method, a number of interesting problems have presented themselves for future work:

1) Callahan has demonstrated an algorithm for alias-free flow-sensitive summary problems that runs in time proportional to N times the size of the *program summary graph* [Call 88]. We believe that information about aliases can be used to improve the precision of the answers to some, but not all, problems solvable with his method.
2) The method we have presented computes information about variables that may be aliases. While it is unlikely that exact must-style alias information can be computed efficiently, it may be possible to compute useful approximations to such sets with efficient techniques.
3) This method treats arrays in a very naive way — as single names. A more sophisticated treatment of arrays, in the style of Callahan and Kennedy's regular section analysis, might produce useful information about the offsets and patterns of overlap in a program [CaKe 87].

We intend to pursue each of these ideas.

9. Conclusions

We have introduced a new interprocedural analysis algorithm for computing aliases due to call-by-reference parameter binding. This algorithm has a complexity of $\mathbf{O}(N^2 + NE)$ where N is the total number of procedures in the program and E is the number of call sites. It is amenable to bit-vector implementation and uses the same graphical representation as our algorithms for the alias-free summary problems [CoKe 88] and interprocedural constant propagation [Torc 85, CCKT 86].

We have demonstrated how the method can be used to produce a general solution to an interprocedural summary problem from the solution to the alias-free problem. This process takes $\mathbf{O}(NE)$ time. When combined with previous results on the flow-insensitive analysis of alias-free problems [CoKe 88], this produces an algorithm for solving these problems in time proportional to the size of the call graph times the number of variables in the program.

The time bound achieved is asymptotically the fastest known. We expect variants of this algorithm to be extremely fast in practice.

10. Acknowledgements

Throughout our work on this problem, we have benefitted from discussions with David Callahan, Linda Torczon, Barbara Ryder, Susan Horwitz, and Tom Reps. To these people go our heartfelt thanks.

References

[Alle 74] F.E. Allen, ''Interprocedural data flow analysis'', *Proc. of the 1974 IFIPS Congress*, 1974.

[Bann 79] J.P. Banning, ''An efficient way to find the side effects of procedure calls and the aliases of variables'', *Proc. Sixth POPL*, Jan., 1979.

[Bart 78] J.M. Barth, ''A practical interprocedural data flow analysis algorithm'', *CACM* 21(9), Sept., 1978.

[Burk 84] M. Burke, ''An interval analysis approach toward interprocedural data flow'', Report RC 10640, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., July, 1984.

[BuCy 86] M. Burke and R. Cytron, ''Interprocedural dependence analysis and parallelization'', *Proc. SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Notices* 21(7), July 1986.

[Call 88] D. Callahan, ''The program summary graph and flow-sensitive interprocedural data flow analysis'', *Proc. SIGPLAN 88 Conference on Programming Language Design and Implementation, SIGPLAN Notices* 23(7), July 1988.

[CCKT 86] D. Callahan, K.D. Cooper, K. Kennedy, and L. Torczon, ''Interprocedural constant propagation'', *Proc. SIGPLAN 86 Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), July 1986.

[CaKe 87] D. Callahan and K. Kennedy, ''Analysis of interprocedural side effects in a parallel programming environment'', *Proc. First Int'l Conference on Supercomputing*, Athens, Greece, June 1987.

[CaRy 86] M.D. Carroll and B.G. Ryder, ‘‘An incremental algorithm for software analysis’’, *Proc. of the* SIGSOFT/SIGPLAN *Software Engineering Symposium on Practical Software Development Environments*, *SIGPLAN Notices* 22(1), Jan. 1987.

[Coop 85] K.D. Cooper, ‘‘Analyzing aliases of reference formal parameters’’, *Proc. Twelfth POPL*, Jan. 1985.

[CoKe 84] K.D. Cooper and K. Kennedy, ‘‘Efficient computation of flow insensitive interprocedural summary information’’, *Proc. SIGPLAN 84 Symposium on Compiler Construction*, *SIGPLAN Notices* 19(6), June 1984.

[CoKe 87] K.D. Cooper and K. Kennedy, ‘‘Efficient computation of flow-insensitive interprocedural summary information — a correction’’, *SIGPLAN Notices*, 23(4), April, 1988 (also TR87-60, Department of Computer Science, Rice University, Oct. 1987).

[CoKe 88] K.D. Cooper and K. Kennedy, ‘‘Interprocedural side-effect analysis in linear time’’, *Proc. SIGPLAN 88 Conference on Programming Language Design and Implementation*, *SIGPLAN Notices* 23(7), July 1988.

[HoRB 88] S. Horwitz, T. Reps and D Binkley, ‘‘Interprocedural slicing using dependence graphs’’, *Proc. SIGPLAN 88 Conference on Programming Language Design and Implementation*, *SIGPLAN Notices* 23(7), July 1988.

[Myer 80] E. Myers, ‘‘A precise and efficient algorithm for determining existential summary data flow information’’, Technical Report CU-CS-175-80, Department of Computer Science, University of Colorado, March, 1980.

[Rose 79] B. Rosen, ‘‘Data flow analysis for procedural languages’’, *JACM* 26(2), April, 1979.

[Spil 71] T.C. Spillman, ‘‘Exposing side-effects in a PL/I optimizing compiler’’, *Proc. of the 1971 IFIPS Congress*, 1971.

[Torc 85] L. Torczon, ‘‘Compilation dependences in an ambitious optimizing compiler’’, Ph.D. dissertation, Department of Computer Science, Rice University, May, 1985.