Analyzing Aliases of Reference Formal Parameters

Keith D. Cooper[†]

Department of Computer Science
Rice University
Houston, Texas

1. Abstract

Compilers for languages with call-by-reference formal parameters must deal with aliases arising from the renaming effects at call sites. This paper presents a set of techniques for analyzing aliasing patterns. The analysis is divided into detecting the introduction of aliases and tracking their propagation. The algorithm for introduction analysis is simple enough to be performed in a structured editor or parser. A data flow analysis framework is given for the propagation problem, making it possible to solve using standard algorithms from global data flow analysis. Several optimizations are shown which can shrink the size of the problem, and extensions are given to handle Algol-style name scoping. Finally, this technique is compared to an alternative implementation strategy and an approximative technique.

2. Introduction

Any compiler for a language which has call-by-reference formal parameters must deal with potential aliases involving formal parameters. In traditional optimizing compilers, this complication is handled in one of two ways: either the compiler intentionally ignores aliasing or else it avoids many optimizations involving either global variables or call-by-reference formal parameters. This problem is of particular concern to compilers which attempt to look across procedure boundaries to gather information and perform optimizations. Several such systems have been either designed or implemented [Alle] [Spil] [HoKe].

Mechanisms for gathering information about aliases have been examined in some depth in the literature. Barth worked the aliasing computation into his relational formulas for computing flow-insensitive interprocedural summary information [Bart]. In Banning's dissertation, he gives an aliasing algorithm which follows parameter binding chains through the program in a depth first fashion to compute possible aliases [Bann]. Myers presents an algorithm for solving the aliasing problem which is similar in flavor to the work presented here [Myer].

While the techniques proposed in the literature certainly work, the papers presenting these techniques have largely dealt with aliasing as a side issue. As a result, they provide the reader and the implementor with little intuition into the underlying problem. This paper gives a tutorial treatment of the analysis of aliases arising from the use of call-by-reference formal parameters. The analysis is formulated as a data flow analysis problem, allowing the compiler writer to use any of the standard techniques from global data flow analysis to solve the problem. The techniques shown are intended for implementation in a programming environment. They work equally well in a more traditional batch compiler. Several simple optimizations are identified which can shrink the size of an instance of the problem.

The call-by-reference binding mechanism for formal parameters creates new names for storage locations in a program. Because of this, using call-by-reference parameters can lead to situations where a single procedure has multiple names which refer to a single storage location. These

[†]This research has been supported by the National Science Foundation through grants MCS 81-04006, MCS 81-21884, and MCS 83-03638.

To appear: Proceedings, 12th POPL Symposium

names are said to be *aliases*.

Aliases can also arise from other source language features, notably pointers and user specified storage overlays like the FORTRAN EQUIVALENCE statement or the PL/I DEFINED attribute. This paper does not address these issues; a discussion of pointers can be found in [Wieh] while standard techniques to deal with EQUIVALENCE style aliasing are discussed in [AhUl].

To simplify the discussion, most of this paper will assume the two-level name scoping conventions of ANSI FORTRAN. Section 6 shows how to extend the algorithms to account for the nested name scoping of Algol-like languages. Discussing aliasing and FORTRAN seems problematical; the ANSI standard [ANSI] expressly prohibits a program from storing a value into a variable which is aliased. Nonetheless, most FORTRAN compilers painstakingly avoid using register optimizations on global variables or call-by-reference formal parameters to ensure that aliased variables behave in the expected fashion.

To simplify the task of generating concise pathological examples, I will use recursion in the example FORTRAN program fragments even though recursion is not included in standard FORTRAN. Several current compilers support recursion and it is likely to appear in the next FORTRAN standard.

3. Terminology

A small amount of notation is necessary to simplify the remainder of the paper. Because aliasing is an interprocedural phenomenon, any algorithm which computes information about potential aliasing patterns requires structural information about the program. In particular, the program is represented by a graph, $\mathbf{C} = (\mathbf{N},\mathbf{E})$, which has a node $n \in \mathbf{N}$ representing each procedure in the program and an edge $e = (m,n) \in \mathbf{E}$, $m, n \in \mathbf{N}$[1], for each call site in procedure m which invokes procedure n. Since procedure m can contain several calls to procedure n, each with a unique set of actual parameters, **C** must be a multi-graph with a unique edge for each call site. To make this structural information accessible to the algorithms, each node is assumed to be annotated with two sets:

(1) *in*(n) contains each edge entering n,

(2) *out*(n) contains each edge leaving n.

Aliasing arises from the renamings that occur during parameter binding at call sites. Discussing these interactions requires a notation for specifying the various parameters at each call site. For the call site corresponding to an edge $e \in \mathbf{E}$, there are two sets associated with the call site to describe its parameters:

(1) *fp*(n) contains the formal parameters of n,

(2) *ap*(e) contains the actual parameters at the call site.

Specific parameters are referenced by simply subscripting them with an ordinal position. Thus, at an edge e, *ap*(e) is passed to *fp*(n), $1 \le i \le |fp(n)|$.

To describe the impact of name scoping rules, a set *gv*(n) is associated with each procedure n. This set contains the names of all variables which are mentioned in the body of n but are not declared as local to n. This information should be readily available as a result of the normal syntactic analysis performed by the editor or parser prior to aliasing analysis.

It is convenient to have a concise notation to describe the bindings which occur at call sites. Along edge e, x is *bound* to y, denoted $x \rightarrow_e y$, if one of the following is true:

(1) x is *ap*(e) and y is *fp*(n)

(2) x is global to n and y is x.

Thus, $ap(e) \rightarrow_e fp(n)$, $1 \le i \le |fp(n)|$, and, for every $v \in gv(n)$, $v \rightarrow_e v$. A chain of bindings

$$w \rightarrow_{e1} x,\ x \rightarrow_{e2} y,\ y \rightarrow_{e3} z$$

is denoted by $w \rightarrow^* z$.

4. The Problem

Whenever a program can reference a single storage location using two different variable names, those names are said to be *aliases*. It is

[1]Throughout the paper, an edge e is assumed to be (m,n) unless it is explicitly stated otherwise.

essential that the compiler understand which aliases can occur, since even trivial inter-statement optimizations, like retaining values in registers across statement boundaries, can yield incorrect code in the presence of aliasing. Consider the following sequence of assignments:

```
a = 10
b = 12
c = a * b.
```

In the absence of aliasing, the compiler would probably retain the values of a and b in registers, since they are computed and then referenced immediately. If, however, a and b are names for a single storage location because they are aliases, then this simple optimization leads to *incorrectly* assigning the value 120 to c. Thus, if aliases are possible, the compiler is faced with the choice of either not using register optimizations on global variables and call-by-reference formal parameters, or generating code which may produce incorrect results.

The goal of alias analysis is to annotate each procedure n in the program with a set *Alias*(n) containing all the aliases which can hold on entry to n. Each potential alias is represented by an *alias pair*, written <x,y>. The presence of the alias pair <x,y> in *Alias*(n) indicates that some sequence of call sites in the program results in n being invoked with x and y as names for a single storage location. This does not imply that x and y are aliases on every invocation of the procedure, but simply that the procedure can be invoked with x and y aliased.

```
PROGRAM MAIN
COMMON //N
INTEGER I,J,K,N
 ...
CALL S1(N, I, J, K)
 ...
END

SUBROUTINE S1(A, B, C, D)
COMMON //N
INTEGER A,B,C,D,E
 ...
CALL S1(E, A, B, C)
 ...
END
```

Figure 1

To provide a concrete example of the information derived by this analysis, consider the FORTRAN program fragment shown in Figure 1. The call site in MAIN gives rise to an alias on entry to procedure S1, since it passes the global variable N as an actual parameter. When invoked from MAIN, S1 can access a single storage location using either of the two names, N or A. Thus, the call site in MAIN gives rise to an alias pair <N,A> on entry to S1. The remaining variables passed at the call site in MAIN are purely local in their name scope and visibility, so no other aliases arise from that call site. Thus, *Alias*(S1) contains the pair <N,A>.

The recursive call in S1 gives rise to a number of aliases. Given that the alias pair <N,A> holds on entry to S1, the first execution of the recursive call site creates an alias pair <N,B>. This happens because the global variable N is visible to each incarnation of S1, and the original reference to N, through A, is now passed through to B. Similarly, the next two calls add <N,C> and <N,D> to the set of potential aliases which can hold on entry to S1. Since there are no more call sites to consider, *Alias*(MAIN) is ∅ and *Alias*(S1) is {<N,A>,<N,B>,<N,C>,<N,D>}.

In the example, all of the aliasing patterns discovered are the direct result of binding patterns at call sites. In MAIN, an alias is produced because a variable global to S1 is passed as an actual parameter. In S1, new alias pairs are created when the recursive call passes a variable already involved in an alias pair as an actual parameter.

To simplify the task of understanding the problem, consider it as a pair of separate and distinct subproblems: detecting the *introduction* of aliases, and analyzing the *propagation* of aliases through the call graph. Aliases are introduced by specific binding patterns at call sites and propagated by another type of binding patterns at call sites. Call sites which introduce no aliases may propagate aliases; likewise call sites which propagate no aliases may introduce aliases.

4.1. Alias Introduction

Alias introduction occurs only at call sites, because only at a procedure call is the mapping from names to locations modified.[2] An alias is created when the called procedure is given multiple names for a single location. This occurs in two ways:

(1) a single actual parameter is passed *in more than one parameter position*. This binds the corresponding formal parameters of the called procedure to a single storage location, making them potential aliases.

(2) a variable which is global to the called procedure is passed as an actual parameter. This introduces an alias between the corresponding formal parameter of the called procedure and the global variable.

These are the *only* mechanisms for introducing an alias. Any call site which passes distinct actual parameters, none of which is global to the called procedure, can *not* introduce aliases.

4.2. Propagation Analysis

Once an alias has been introduced at a call site, that alias can hold on entry to the called procedure. Thus, any modifications of the value of one name in the alias pair changes the value of the other name. Further, any call site inside that procedure can propagate the alias pair to another procedure simply by giving the called procedure a name for each element of the alias pair.

More precisely, an alias pair $\langle x,y\rangle \in$ *Alias*(m) can be propagated along an edge e by two different mechanisms:

(1) If $x, y \in fp(m)$, $x \rightarrow_e x'$ and $y \rightarrow_e y'$ then $\langle x,y\rangle$ is propagated into the pair $\langle x',y'\rangle$ which can hold on entry to n.

(2) If $x \in fp(m)$, y is global to n, and $x \rightarrow_e x'$, then $\langle x,y\rangle$ is propagated into the pair $\langle x',y\rangle$ which can hold on entry to n.

In either case, the procedure propagating the alias must pass one of its own formal parameters as an actual parameter, creating a situation where there exists an i, $1 \le i \le |fp(m)|$, and a j, $1 \le j \le |fp(n)|$, such that $fp(m) \rightarrow_e fp(n)$, Thus, alias propagation only occurs where call-by-reference formal parameters are used as actual parameters.[3] Call sites which do not pass formal parameters as actual parameters can *not* propagate aliases.

Because the alias propagation problem involves storage mappings, parameters bindings, and tracking the binding paths of pairs of names through the call multi-graph, there are a number of properties of the problem which may not be obvious. Aliasing is not *transitive*; knowing that both <X,Y> and <Y,Z> are contained in *Alias*(n) does not imply that <X,Z> is also in *Alias*(n). To see this, consider the program fragment of figure 1. The only alias introduced at any call site in the program is <N,A>, introduced from the call in MAIN. On entry to S1, this is the only alias arising from introduction effects.

The propagation effects of the recursive call in S1 are complex. Each execution of the call causes the following bindings to occur:

$$N \rightarrow_e N,\ A \rightarrow_e B,\ B \rightarrow_e C,\ C \rightarrow_e D.$$

Thus, when called from MAIN, the only alias which actually occurs is <N,A>. The first recursive call propagates <N,A> into <N,B>, the second <N,B> into <N,C>, and the third <N,C> into <N,D>. On consecutive calls, each of A, B, C, and D is an alias of N, but *at no time* do any two of them refer to the same location. Thus in any execution of S1, at most one of A, B, C, or D is aliased to N. The other names are bound to local

[2] Language features which dynamically manipulate the mapping of names to storage, like the IDENTIFY statement in VECTRAN [Paul], would of course introduce another source for introduction of aliases.

[3] There is one other possibility, that both x and y are global to n. This arises only with more complex name scoping rules, and is dealt with in section 6.

```
PROGRAM MAIN
COMMON //N
 ...
CALL S2(N)
 ...
END


SUBROUTINE S2(A)
INTEGER A
 ...
CALL S3(A)
 ...
END


SUBROUTINE S3(B)
COMMON //N
INTEGER B,C
 ...
N = B * C
B = C - N
 ...
END
```

Figure 2

variables of MAIN or the local variable E some previous incarnation of S1. However, all of the alias pairs <N,A>, <N,B>, <N,C>, and <N,D> are potential aliases on entry to S1.

Alias propagation is *path sensitive*. It requires that both elements of the alias pair be bound at a single call site. Propagation must involve both elements of the pair, not just one of the individual variables in the pair. Assume <x,y> $\in$ *Alias*(m) and that both x and y get propagated into some procedure p, so x $\rightarrow^{*}$ x', y $\rightarrow^{*}$ y', where x' and y' are parameters or globals of p. This is not sufficient to conclude that <x',y'> $\in$ *Alias*(p). For the alias pair to be propagated requires not only that both names in the original alias pair, <x,y>, reach p, but also that the bindings leading to x' and y' occur along the same path through the call graph.

The propagation analysis must be careful in considering the range of global names. Reference formal parameters can propagate only through parameter bindings, so the program regions over which a given binding chain merits attention is obvious. With global names, like FORTRAN COMMON or PL/I EXTERNAL, this is not the case. Such variables appear in a procedure's name space by explicit declaration; their absence from one procedure's name space implies nothing about their presence in either an invoked procedure or an internally nested name scope. In any case where a global is involved in an alias, the global must be considered to be declared in each procedure which the other element of the alias pair can reach. This does not mean that the compiler should consider such an alias pair in generating code; clearly the alias is irrelevant to the compilation of any procedure in which one of its elements is not visible.

The program fragment shown in figure 2 illustrates this point. The call site in MAIN introduces an alias <N,A> which holds on entry to procedure S2, but N is not accessible inside S2. At the call from S2 to S3, A $\rightarrow_{e}$ B. Since N is declared in S3, this propagates <N,A> into <N,B>, even though <N,A> was not meaningful as an alias in S2.

5. An Analytical Technique

The actual computation of aliasing information reflects the division of the problem into introduction and propagation analysis. The computations required to detect and record alias introductions at call sites are completely local in nature, while the analysis of propagation effects uses the machinery of data flow analysis to account for interactions between procedures.

5.1. Introduction Analysis

Introduction analysis constructs, for each call site, a set *Intro*(e) containing the alias pairs representing all of the aliases created by the call site. All of the information required to perform the computation is contained in the text of the call site and the data declarations associated with the procedure containing the call site. Thus *Intro*(e) is a local property of the call site. Because all this information is readily available, *Intro*(e) can be computed by the programming environment's editor or by the parser in a compiler.

The algorithm for performing alias introduction analysis assumes the existence of the

Computing Intro(e)

```
/* initialize the temporary slots */
for 1 ≤ i ≤ |ap(e)|
        let AliasPtr[ap(e)] ← λ

/* build the lists from actual parameters */
for 1 ≤ i ≤ |ap(e)|
        add i to the list AliasPtr[ap(e)]

/* use the lists to construct Alias(n) */
let Intro(e) ← ∅

for 1 ≤ i ≤ |ap(e)|
        /* 1 name, > 1 position */
        if | AliasPtr[ap(e)] | > 1
          for each pair of list elements j and k
                add <fp(n),fp.if 0>0 .        tm Line 507 -- I cannot skip multiple pages
(n)> to Intro(e)

        /* global passed as an actual */
        if ap(e) is globally visible in n,
          add <ap(e),fp(n)> to Intro(e)
```

Figure 3

typical environment found inside a language based editor or parser. Specifically, it expects that a symbol table has been constructed, that there is a slot of temporary space, named AliasPtr, available for each entry in the table, and that variables are represented in the intermediate form by symbol table indices. Given this information, the analysis of a call site e proceeds as shown in Figure 3. For each actual parameter of the call site, the algorithm builds a list containing the name of the corresponding formal at each parameter position in which the variable is bound. It then traverses these lists placing an alias pair in *Intro*(e) every time an actual is found which is passed in two distinct parameter positions. At the same time, it checks the scopes of the actual parameters, creating appropriate alias pairs for any variables which are global to the called procedure. Thus, constructing *Intro*(e) takes **O**(|*ap*(e)| + |*Intro*(e)|) time for each call site e.

To aid in optimizing the propagation analysis, some additional information can be constructed during the introduction analysis. As each call site is analyzed, two additional lists can be kept:

(1) call sites with non-empty *Intro*(e) sets. These are the only call sites which can introduce aliases.

(2) call sites which pass formal parameters of the calling procedure as actual parameters. These are the only call sites which can propagate aliases.

Because these lists will require some preprocessing before they are useful to the propagation analysis, it suffices to maintain the lists on a procedure by procedure basis. Since the first list is trivial to construct, there is no need to store it. The overhead entailed in keeping the latter list is minimal. The next section shows how to use these lists to speed up the propagation analysis.

5.2. Propagation Analysis

The task of propagation analysis is to annotate each node n ∈ **N** with a set *Alias*(n) containing the set of aliases which *may* hold on entry to the procedure corresponding to n. Since propagation analysis only tracks aliases which originate in the *Intro*(e) set of some e ∈ **E**, introduction analysis is a prerequisite for propagation analysis.

An alias pair belongs in *Alias*(n) only if it is in *Intro*(e) for some edge e or derives from a pair in *Alias*(m) by propagation through an edge e = (m,n). Thus, *Alias*(n) can be computed as:

$$Alias(n) = \cup_{e=(m,n)\in in(n)} \ (Intro(e) \cup b_e(Alias(m)))$$

where b_e is an edge specific function modeling the alais propagation rules. It is defined as:

$$b_e(X) = \cup_{<x,y>\in X} \{ <x',y'> \mid x \rightarrow_e x' \text{ and } y \rightarrow_e y' \}.$$

Formulating the propagation problem as a set of simultaneous equations over the nodes and edges of a graph makes it natural to solve it with standard techniques from global data flow analysis.

5.2.1. A Data Flow Framework

Much of the underlying theory of global data flow analysis has been developed using semi-lattice based models. To help understand the properties of the propagation problem, it is instructive to pose a Kam-Ullman style semi-lattice framework for the problem [KaUl].

A data flow framework for a problem is a four-tuple, (G, L, F, M), where

- G is a graph, (N,E), representing the control flow in the portion of the program being analyzed,
- L is a semi-lattice representing the facts being derived about the program,
- F is a function space of transformations on elements of L, F:L→L, and
- M is a mapping, M:E→F, which assigns a function in F to each edge in the graph G.

Casting the propagation problem as a data flow framework makes a number of theoretical results applicable to it.

The data flow framework for the propagation problem is

P = (**C**, **L**, **F**, **M**)

where

- **C** is the program's call multi-graph, (**N**,**E**), from section 3.
- **L** is a semi-lattice ($2^{\mathbf{AP}}$,), where **AP** is the set of possible names of alias pairs, or the set of pairs of variables in the program, 2^X denotes the power set of X, and is set union (∪). $2^{\mathbf{AP}}$ is finite, and is idempotent, associative, and commutative.
- **F** is the function space consisting of all functions mapping $2^{\mathbf{AP}} \rightarrow 2^{\mathbf{AP}}$ which are of the form f(X) = A g{X}, with A ≤ $2^{\mathbf{AP}}$ a constant, g:**AP**→$2^{\mathbf{AP}}$ a function, and {} denoting elemental application of a function to a set.
- **M** defines a specific function for every edge e = (m,n) ∈ **E**. This is done by assigning A the value *Intro*(e) and letting g be the function f(<x,y>) = {<w,z>| $x \rightarrow_e w$ and $y \rightarrow_e z$}.

Kam and Ullman give a set of four criteria which a framework must meet for their results to apply. **P** meets the criteria. Because **P** is an admissible Kam-Ullman framework, the solution derived by the iterative algorithm is independent of the order in which nodes of the graph are updated. This leaves the implementor free to order the node updates in an order which leads to an efficient implementation. Further, applying the iterative algorithm to an instance of the problem yields a unique least fixed point solution, which is identical to the meet over all paths solution for the instance.

Among admissible frameworks, Kam and Ullman identified a class of *rapid* frameworks which the iterative algorithm can solve in nearly linear time. A framework is rapid if and only if:

$$f(X) \geq X \ \ f(), \forall f \in \mathbf{F}, \forall X \in \mathbf{L}.^4$$

The propagation framework is *not* a rapid framework. Similarly, **P** is not Graham-Wegman *fast* [Coop]. Thus, the problem can *not* not be directly solved using one of the near linear time techniques from global data flow analysis.

5.2.2. Time Complexity

It is important to understand the time complexity of solving this problem with the iterative algorithm. The iterative algorithm halts only when the information being propagated through the graph can go no further. Aliases propagate along paths defined by the bindings of parameters, u→v, v→w, w→x, x→y. If the call multi-graph **C** is acyclic, the longest such binding path has length |**N**| - 1. If **C** contains a cycle, then the path can loop back and pass through multiple formal parameters of a single procedure. In a graph with a single cycle, the longest possible path will be bounded by k(|**N**| - 1) where k is the smallest number of formal parameters of any procedure involved in the cycle. Following this line of reasoning, the longest binding chain through a cyclic graph is bounded by $(|\mathbf{N}| - 1) \times MaxParms$ where *MaxParms* is the maximum number of formal parameters of any procedure in the program.

Consider a round-robin version of the iterative algorithm. It makes complete passes over the call graph, updating each node. If any node's value changes during a pass, another pass is made. In this scheme, any alias pair which can propagate further through the graph advances along at least one edge each pass. Thus, the round-robin iterative algorithm halts in no more than $(|\mathbf{N}| - 1) \times MaxParms + 1$ passes[5], assuming *MaxParms* is a constant independent of program size. Since each pass can visit every node in the graph, this requires $\mathbf{O}(|\mathbf{N}|^2)$ individual node updates,

The performance of the iterative algorithm on an actual instance of the alias propagation problem depends on two characteristics of the program, which I call the *density* of aliasing and the *longevity* of individual alias pairs. It is important to realize that it is unlikely that the worst case will be encountered.

---

[4][KaUl] Observation 6, p. 166.

[5]Using a worklist formulation of this algorithm, with a depth first ordering to visit nodes can improve the expected case running time of the algorithm, but the asymptotic complexity remains the same.

The density of aliasing refers to the raw number of introduced aliases. If there are few introduced aliases, opportunities for alias propagation are few. In practice, the number of alias introductions per call site is likely to be small because most call sites will introduce no aliases. Although there is no empirical evidence to support this claim today, examination of several large programs appears to support it. Most call sites pass local variables and constants as distinct actual parameters.

The longevity of an alias pair is the length of the path over which it can be propagated. Recall that an alias pair only propagates through a call site if both variables in the alias pair are visible in the called procedure. In the case of an alias pair between two formals, this requires that the procedure use both formals as actuals in the same call site. If the alias pair contains a global and a formal, the procedure need only pass the formal.

Aliases should have relatively short lifetimes, no more than two or three procedures for most alias pairs. Programmers do not normally pass variables into a procedure as formal parameters and out of it as actuals, repeatedly, over long chains of procedures. Global variables, like FORTRAN COMMON, were created precisely to allow programmers to avoid doing this! To achieve the maximum binding path length discussed above would require a procedure to pass a single value around the cycle through every formal parameter in the procedure with the smallest number of formal parameters. Such a program would be difficult to design and understand.

If the density of aliasing is low, and the longevity of alias pairs is not long, the expected running time of an iterative implementation will be much lower than the worst case time bound.

5.2.3. Optimizations

In an implementation of the iterative framework, the additional information collected during the introduction analysis can be used to decrease the size of the graph being processed. For each procedure, the introduction analysis constructed a list of call sites which can propagate aliases. Call sites which introduce aliases can be easily identified by their non-empty *Intro*(e) sets. This information is used to construct, for each procedure, a set *introduces*(n) containing those edges entering n which can introduce aliases and a set *propagates*(n) containing those edges entering n which

can propagate aliases. This can be done in **O**(|**E**|) time.

Given that each node is annotated with *introduces*(n) and *propagates*(n) sets, the formula for computing *Alias*(n) can be restructured to perform unions over the edges which are relevant to the analysis, rather than over all edges incident to a given node. This yields the formula:

$$Alias(n) = (\cup_{e \in introduces(n)} Intro(e)) \cup (\cup_{e=(m,n) \in propagates(n)} b_e(Alias(m)))$$

which avoids examining unnecessary edges. In effect, this allows the introduction analysis to trim the graph seen by propagation analysis. Further, the *Intro*(e) term is invariant throughout the analysis. Thus, it can be precomputed, restricting the actual computations performed at each node to evaluating the propagation functions for each e ∈ *propagates*(n) and unioning together the result with precomputed *Intro*(e) term. In practice, these optimizations can produce substantial savings.

6. Algol-style Name Scoping

In developing the techniques for analyzing aliases, there has been an implicit assumption that the programming language being analyzed has the two-level name scoping rules of FORTRAN. While the FORTRAN rules simplify the task of explaining the aliasing problem, these techniques will find far more application if they are readily adaptable to the more general block structured scoping rules typically found in Algol-style languages. This section shows how the techniques can be extended to cope with this more general case.

Implementation of $b_e(X)$

```
result ← ∅

for i ← 1 to |ap(e)|
   if ap(e) is involved in
     an alias pair <ap(e),x>
     then
          if x ∈ gv(n)
            then add alias pairs <x,z> to result
                 for each z ∈ bind_e(ap(e))
          if x ∈ ap(e)
            then add alias pairs <y,z> to result
                 for each y ∈ bind_e(x)
                 for each z ∈ bind_e(ap(e))
```

Figure 4

The alias introduction analysis is unchanged. Recall that the set *gv*(n) contains all of the name scoping information required by the introduction algorithms. In an analyzer for a language with more complex scoping rules, the increased complexity of name scoping is absorbed completely in the computation of *gv*(n). Since this knowledge is already needed by the editor or parser, the added complexity is of no real concern. The algorithm for computing *Intro*(e) given in section 5.1 needs no modification.

The discussion of propagation analysis relied on the assumption that the name spaces of the calling and called procedures are disjoint, except where explicit sharing exists in the form of global variables. In a block structured language, the name scopes of two procedures can be nested. This requires a re-examination of the propagation analysis to insure that aliases which are *statically inherited* from surrounding name spaces are handled properly.

In section 4.2, two conditions were given for alias propagation along an edge e = (m,n):

(1) If x, y ∈ *fp*(m), x $\rightarrow_e$ x' and y $\rightarrow_e$ y' then <x,y> is propagated into the pair <x',y'> which can hold on entry to n.

(2) If x ∈ *fp*(m), y is global to n, and x $\rightarrow_e$ x', then <x,y> is propagated into the pair <x',y> which can hold on entry to n.

Extending the name scoping rules to handle block structure requires a third rule:

(3) If x and y ∈ *gv*(n), then <x,y> is inherited by n from m.

This rule allows an alias to be propagated into any nested block in which both elements of the alias pair are visible.

Again, this rule hides much of the complexity of the issue in the construction of *gv*(n). All of the issues like redeclaration of the textual name are absorbed into *gv*(n). Of course, names must be assumed to be unique; in an implementation this would naturally lead to the use of fully qualified names in the analysis.

In an actual implementation, the impact of adding this third rule is straightforward. The interpretation of the propagation function b_e is changed slightly. In the two-level name scoping situation, the implementation might look like the fragment shown in Figure 4. Here, $bind_e(x)$ contains the set of variables to which x can be bound at the call site corresponding to e. This is identical to the information constructed in the AliasPtr lists in the introduction analysis. The code fragment examines each actual parameter, and applies the propagation rules to any actual parameter which is an element of an alias pair.

In a more general block structured language, the following loop must be added to the fragment shown in figure 4 to account for the third rule. It should be placed at the outermost nesting level.

```
for each <x,y> ∈ Alias(m)
    if x ∈ gv(n) and y ∈ gv(n)
        result ← result ∪ <x,y>
```

This extends the computation of b_e to account for more general name scoping rules.

7. Other Techniques

It is important to consider other techniques to solve the alias propagation problem. In particular, we should consider a technique similar to that presented in [CoKe] for computing flow insensitive interprocedural summary information. That algorithm runs in $\mathbf{O}(|\mathbf{E}|\alpha(|\mathbf{E}|,|\mathbf{N}|))$ time. It achieves that time bound by computing a mapping which describes all of the binding patterns among the formal parameters of procedures in the program. This information might prove useful in an aliasing analysis.

The fundamental assumption which allows Cooper and Kennedy to achieve that time bound is that the maximum number of formal parameters of any procedure in the program, *MaxParms*, is bounded by a small constant. Applying the same technique to the alias propagation problem would lead to an assumption that *MaxParms* (*MaxParms*-1) / 2 is a small constant since alias propagation must track pairs of formal parameters. This assumption is fairly tenuous; it is reasonable to claim that 20 is a small constant, but the claim is less clear for 190.

The second technique worthy of consideration is an approximative technique. If aliasing is rare, as suggested in the previous section, it may be reasonable to compute an extremely conservative approximation to the information. In section 4.2, it was pointed out that alias propagation requires that both elements of the alias pair be transmitted along the same call chain. Removing this requirement will produce an extremely conservative approximation; the resulting information includes any pair produced by using the actual propagation framework from section 5.2, along with some additional pairs propagated because of the relaxed propagation rule.

By removing the path sensitivity, the propagation algorithm need only track the manner in which formal parameters of one procedure can be bound to other formal parameters in the program. This is precisely the information computed in the Cooper-Kennedy technique for producing flow insensitive summary information. Because the algorithm need only track formal parameters, instead of pairs of formal parameters, the assumption that *MaxParms* is bounded by a small constant is still reasonable. Further, the parameter to parameter mapping required to compute the aliasing information is produced as a part of the interprocedural summary computation.

The tradeoffs between these three techniques are difficult to assess. The iterative approach is certainly the simplest to implement, but has the worst case asymptotic behavior. A method based on [CoKe] will have trouble claiming the $\mathbf{O}(|\mathbf{E}|\alpha(|\mathbf{E}|,|\mathbf{N}|))$ time bound, but it is important to remember that the expensive operations only arise in response to cycles in the call graph, and that the actual size of the operations performed is not based on *MaxParms*, but on the actual number of parameters at each call site. Finally, the approximative technique will achieve the $\mathbf{O}(|\mathbf{E}|\alpha(|\mathbf{E}|,|\mathbf{N}|))$ time bound, but with a potentially disastrous loss of precision. The cases where the approximation is close to precise are also cases where the iterative approach should converge quickly.

8. Conclusions

The problem of analyzing the aliasing relationships introduced as artifacts of call-by-reference parameter passing is a complex and subtle problem. The problem divides nicely into analysis of alias introduction and alias propagation. The former problem has a simple solution, suitable for implementation in either a traditional parser or a structured editor. The latter problem can be solved with standard data flow analysis techniques.

The alias propagation problem is not amenable to direct solution using one of the nearly linear time methods from global data flow analysis. An iterative framework for the problem has a worst case complexity of $\mathbf{O}(|\mathbf{N}|^2)$, but the actual behavior of the algorithm is a function of the frequency with which the programmer introduces aliases and passes formal parameters through chains of calls.

To determine whether the iterative algorithm is an appropriate implementation strategy will require quantitative study of aliasing patterns in real programs. Only then will the trade-offs between iterative, Cooper-Kennedy style, and approximative schemes become clear.

References

[AhUl] A. Aho, and J. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Ma., 1977.

[Alle] F. Allen, J. Carter, J. Fabri, J. Ferrante, W. Harrison, P. Loewner, L. Trevillyan, "The Experimental Compiling System", *IBM Journal of Research and Development*, Vol. 24, No. 6, Nov. 1980.

[ANSI] ANSI, American National Standard FORTRAN-66 X3.9-1966.

[Bann] J. Banning, "A Method for Determining the Side Effects of Procedure Calls", PhD Dissertation, Stanford University, Aug. 1978.

[Bart] J. Barth, "A Practical Interprocedural Data Flow Analysis Algorithm". *CACM* Vol. 21, No. 9, Sept. 1978, pp. 724-736.

[CoKe] K. Cooper, and K. Kennedy, "Efficient Computation of Flow Insensitive Interprocedural Summary Information", Proceedings SIGPLAN '84 Symposium on Compiler Construction, June, 1984.

[Coop] K. Cooper, "Interprocedural Data Flow Analysis in a Programming Environment", PhD Dissertation, Rice University, May 1983.

[HoKe] R. Hood, and K. Kennedy, "A Programming Environment for Fortran", Rice University, Department of Mathematical Sciences TR84-1, Rice University, 1984, (to appear in the proceedings of the Eighteenth Annual Hawaii International Conference on Systems Sciences, January, 1985).

[KaUl] J. Kam and J. Ullman, "Global Data Flow Analysis and Iterative Algorithms", *JACM*, Vol. 23, No. 1, Jan. 1976, pp. 158-171.

[Myer] E. Myers, "A Precise and Efficient Algorithm for Determining Existential Summary Data Flow Information", University of Colorado at Boulder, Department of Computer Science Technical Report #CU-CS-175-80.

[Paul] G. Paul, "VECTRAN and the Proposed Vector/Array Extensions to ANSI FORTRAN for Scientific and Engineering Computation", IBM Conference on Scientific and Parallel Computing, Rome, Italy, March 1982.

[Spil] T. Spillman, "Exposing Side-Effects in a PL/I Optimizing Compiler", IFIPS Proceedings, 1971, pp 376-381.

[Wieh] W. Weihl, "Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables", Proceedings 7th POPL, Jan. 1980.