D. B. Lomet

Data Flow Analysis in the Presence of Procedure Calls

Abstract: The aliasing that results in a variable being known by more than one name has greatly complicated efforts to derive data flow information. The approach we take involves the use of a series of claims that, after we compute the data flow for some of the aliasing possibilities, allows us to produce good approximations for the remaining cases. The method can thus limit the potential combinatorial explosion of aliasing computations while providing results that are frequently exact and almost always very good. The method is illustrated in the context of data flow analysis involving multiple procedures and their calling interactions. It is applicable also in the treatment of recursive procedures.

Introduction

Data flow analysis of one form or another has been performed in many compilers for nearly a decade, the first widely known instance being the work of Lowry and Medlock on FORTRAN H [1]. Considerable new work has been done resulting in new or improved algorithms [2–6] and new applications of the information [7–10]. An area that has been generally overlooked, however, is the determination of data flow information in the presence of procedure calls. Historically, most practical realizations of data flow algorithms have simply assumed that nothing is known after another procedure has been called. Recently, both Allen [11] and Rosen [12] have attacked this facet of the problem.

There are, in fact, two aspects of the data flow problem:

1. What is the effect of a procedure call on the data flow of the calling program? We call this the summary data flow problem as the idea is to summarize the effect of the procedure call, given the particular argument-parameter matching at the point of call.
2. What is the effect on the data flow within the called procedure when it is called with a particular combination of arguments? Further, what data flow information is safe to assume for multiple calls of a procedure involving several different arguments for any given parameter? We call this the local data flow problem. An approximate solution to the summary data flow problem can be derived in the process.

For algorithms to perform data flow analysis in the presence of procedure calls, they must cope with the so-called aliasing relationships, cases in which several names all identify the same variable. The aliasing problem and its implications when procedures are involved, and in other cases such as pointer (reference) variables, is treated directly by Spillman [7]; however, the method he uses does not involve knowledge of the local control flow of procedures and hence provides only a very approximate bound on what the procedure might do.

We introduce in this paper a set of inequalities that can be used to approximate the effects of argument-parameter aliasing without requiring a separate analysis for each such aliasing situation. We explore the implications of these approximations for both summary data flow analysis and local data flow analysis. Used in conjunction with Rosen's summary data flow methodology, the effect is to limit the potential combinatorial explosion while producing only slightly less precise results. In the case of Allen's local data flow methodology, the effect is to produce considerably more precise information. This latter approach has much to commend it. Not only are the results in a form that is directly useful for optimizing transformations but they lead to very good results for the summary data flow problem as well. It should be emphasized that this paper is not concerned with the detection of aliasing but rather with the problem of coping with it. The detection of aliasing conditions has been treated by both Spillman [7] and Rosen [12].

Blocks and control flow graphs

To characterize the data flow analysis more precisely, we make use of the notion of a control flow graph and the program it represents. We assume that we are dealing with a typical procedural language, e.g., FORTRAN, ALGOL, PL/I. We do not treat, however, programs that use procedure parameters or variables, ON units, label variables, or tasks. The use of these constructs implies that the control flow graph of a procedure can change dynamically. We hasten to add, however, that techniques exist that make it possible to produce a "worst-

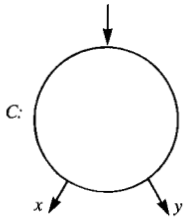

Figure 1 Schematic of a block as seen from the outside.

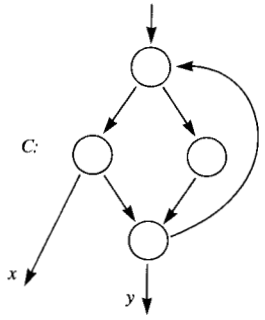

Figure 2 A possible control flow graph for the block of Fig. 1.

case" control flow graph [7]. Further, no attempt is made to distinguish between different generations of variables, as can occur, for example, in the *heap* storage of ALGOL 68 or the BASED and CONTROLLED storage of PL/I. The control flow graph merely links, via arcs, the nodes that correspond to the executable elements of the procedure. These executable elements may be primitive operations, procedure calls, blocks, or even constructs introduced solely to facilitate analysis, i.e., intervals [2]. In treating procedures, one must acquire knowledge concerning all possible abnormal exits (see [7]). Once this is done, the method can readily be applied to such procedures. In particular, executable elements may have multiple exits. Because of this, the local data flow information is most conveniently associated with arcs of the control flow graph.

We call the executable elements blocks, though these should not be confused with BEGIN blocks. Rather, a "block" is a generic name derived from the term basic block (extended basic block) and is simply the unit of code that is of interest. Schematically, any block may be represented as in Fig. 1. It may have only a single entry point but may have multiple exit points.

There are two kinds of blocks, primitive and nonprimitive. Nonprimitive blocks have an inner structure that consists of a control flow graph. Thus, if the block C of Fig. 1 were nonprimitive, peeling away its "shell" might reveal the control flow graph of Fig. 2.

The primitive blocks do not possess a control flow graph, and they are not subject to the form of analysis we present. These primitives are such executable elements as *add* or *move* operators. The data flow characteristics of primitives must be assumed a priori. We discuss the nature of these assertions in the next section. A call to a procedure does not constitute a primitive block. Rather, one should conceive of a copy of the control flow graph of the procedure residing within each block (node) that represents a procedure call. Analysis must be performed on this control flow graph in order to produce the summary data flow information that characterizes these call blocks.

Summary data flow

For the summary data flow computation, we are interested in determining whether a variable may be *defined* (updated), whether the value of a variable may be *used*, and whether there is a path through the unit of interest on which the variable is not updated, i.e., the value of a variable may be *preserved*. (*Preserved* information is essential in determining which definitions can reach a use and whether a definition has subsequent uses.) Data flow information must be asserted in some way for primitive blocks. This information for a nonprimitive block must be derived from the information acquired during an analysis of the control flow graph of the block.

In particular, the data flow information derived for each exit of a block becomes the basis for the data flow information associated with the corresponding exit arc in any other control flow graph containing an instance of the block. Thus, the results of data flow analysis for the exits of the control flow graph of C in Fig. 2 become the basis for the information associated with arcs x and y of all nonprimitive blocks C as in Fig. 1. The definitions for this data flow information given below make this interaction precise.

Aliasing the names of variables can change the results of data flow analysis. Consider the example

$$A \leftarrow C * 2,$$

$$C \leftarrow B + 1.$$

If there is no aliasing, A and C are defined, B and C are used. However, if A and B are aliased, then (A,B), the common variable, is defined but it is not used in the sense that previous assignments to B do not affect the computation, which is now only affected by the assignment

$$(A,B) \leftarrow C * 2,$$

Further, in the unaliased case, B is preserved (since it is not defined), while if A and B are aliased, (A,B) is not preserved.

This simple example is merely intended to indicate how aliasing can affect data flow information and clearly does not reveal all the implications of the aliasing possibilities. We attempt to make these implications clear as we proceed.

Because aliasing can alter the results of data flow analysis, it is necessary to discuss the conditions under which summary data flow analysis is performed. For this purpose we introduce the equivalence condition E:

$E(J)$ When data flow for a variable is analyzed under the equivalence condition $E(J)$, the results describe the situation when the variable and all members of the set J *are* aliased (name the same variable). (S1)

The conditions that we deal with change over the course of our exposition. One change that we have to deal with results from the relative globalness of variables. Only variables that are in the lexical scope of a block can have aliases within the block. It is important to drop from the conditions that we consider all those variables that are not global with respect to the entity we are dealing with. This is an important means of reducing the explosion in the number of conditions. However, it is semantically significant as well because it keeps us from confusing the multiple generations of local (AUTOMATIC) variables that can occur with recursive procedures. For example, we define a function $G(C,P)$ that eliminates all variables in condition P that are not global to block C. Thus, if $P = E(\{a, b, c, d\})$ and variables b and c are not global to C, $G(C,P) = E(\{a, d\})$.

• *Summary data flow definitions*

Whereas the notation for expressing the data flow quantities of interest is the same in both cases, we need to treat separately blocks with control graphs and primitive blocks. For blocks with control flow graphs, the data flow quantities of interest are defined in terms of equations that depend on the analysis of contained blocks. For primitive blocks, i.e., operators such as *add, multiply, move,* etc., it is necessary to make some assertions concerning the nature of their internal flow. It should be emphasized again, however, that calls to procedures are treated as nonprimitive blocks since the assumptions concerning primitives do not generally apply to procedures.

• *Blocks with control flow graphs*

In the case of blocks with control flow graphs, we make use of the results of the data flow analysis on the blocks that represent the nodes of the control flow graph. In particular, we must take the results of these analyses of blocks and tailor them so as to characterize the arcs of the control flow graph. By using the information associated with arcs of a control flow graph, the results for the entire control flow graph (and hence the containing block) can be computed.

In the definitions that follow we designate program variables by lower case letters running $i, j, \cdots$, sets of variables by upper case letters running $I, J, \cdots$, blocks by upper case letters running $C, D, \cdots$, exit points with lower case letters running $u, v, \cdots$, conditions with upper case letters running $P, Q, \cdots$, and arcs with lower case letters running $l, m, \cdots$. The data flow quantities pertaining to blocks are subscripted with the letter B, whereas those pertaining to arcs are subscripted with an A. Thus we have the following definitions, which represent a reformulation of the definitions presented by Rosen [12]:

$Def_B(i, C, x, P) = 1$ iff for some arc l on a path in C from C's entry to exit x, $Def_A(i, l, P) = 1$; $Def_A(i, l, P) = 1$ iff $Def_B(i, D, y, G(D, P \wedge E(J)) = 1$, where block D, exit y, serves to define arc l and the set of variables J are the parameters in D to which variable i is passed (bound) as an argument. (S2)

$Pre_B(i, C, x, P) = 1$ iff there is some path through block C from C's entry to exit x such that for all arcs l on that path $Pre_A(i, l, P) = 1$; $Pre_A(i, l, P) = 1$ iff $Pre_B(i, D, y, G(D, P \wedge E(J))) = 1$, where block D, exit y, serves to define arc l and the set of variables J are the parameters in D to which variable i is passed as an argument. (S3)

$Use_B(i, C, x, P) = 1$ iff there is some path through block C from C's entry to exit x on which there is an arc l such that $Use_A(i, l, P) = 1$ and on each arc k on this path between C's entry and l it is the case that $Pre_A(i, k, P) = 1$; $Use_A(i, l, P) = 1$ iff $Use_B(i, D, y, G(D, P \wedge E(J))) = 1$, where block D, exit y, serves to define arc l and the set of variables J are the parameters in D to which variable i is passed as an argument. (S4)

The quantities Def_B, Pre_B, and Use_B, and the corresponding quantities for arcs, represent "may be" information. Thus, if $Def_B(i, C, x, P) = 1$, then variable i *may be* updated (but perhaps not) when control leaves block C at exit x under condition P. A similar situation exists for the other quantities. So-called "must be" information is available, however. In particular, if $Def_B(i, C, x, P) = 0$, then variable i must be preserved; if $Pre_B(i, C, x, P) = 0$, then variable i must be defined. "Must be used" information is not available. There is one qualification to the above. If both Def_B and Pre_B are zero, then control does not pass through the block to the given exit.

• *Primitives*

We have given definitions for the data flow information in the case where a block has a control flow graph. Primitive blocks have no such graphs, and the question arises as to what we assume concerning the primitives.

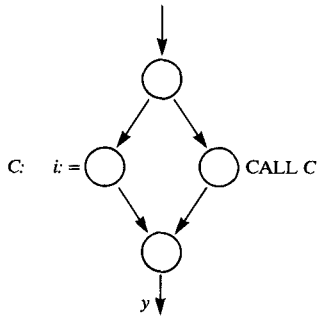

Figure 3 Control flow graph for a recursive procedure C, used to illustrate that Eqs. (S2), (S3), and (S4) have more than one solution.

The information assumed given for each variable i that is referenced by a primitive block C is the following:

$Def_B(i, C, x, \emptyset)$ and $Use_B(i, C, x, \emptyset)$.

We further assume that all variables not given such information are, in fact, not referenced, and hence, if j is such a variable, then

$$Def_B(j, C, x, \emptyset) = Use_B(j, C, x, \emptyset) = 0.$$

Also, we assume that

$$Pre_B(j, C, x, \emptyset) = \neg Def_B(j, C, x, \emptyset).$$

Thus, any definitions that occur on some "path" leaving C at x occur on all such paths. Finally, we assume that if $Use_B(j, C, x, \emptyset) = 1$, then there is always a use of j that occurs before all definitions of any variable in C. This prevents a definition of one variable from interfering with the use of a second variable if the two should be aliased. Since primitives typically acquire values (*use* variables), perform a computation, and then store a result (*definition*), these assumptions are reasonable. (The alternative to making assumptions such as these is to insist on much more information for each primitive. In particular, one must have precise knowledge as to the effects of parameter aliasing.)

• *Multiple solutions*

The equations of (S2), (S3), and (S4) may have more than one solution when recursive procedures are involved. The problem arises because some assumption must be made concerning the effect of a recursive procedure call prior to the analysis of this procedure. Different assumptions can result in different results. Consider the control flow graph for procedure C in Fig. 3.

If, in C, we make the most pessimistic initial assumptions for the call of C, i.e., that

$$Def_B(i, C, y, \emptyset) = Pre_B(i, C, y, \emptyset) = Use_B(i, C, y, \emptyset) = 1,$$

then this assignment is a solution to the equations despite the fact that i is neither preserved nor used. Since it is a solution, attempts to iterate by re-evaluating the equations so as to produce a "better" solution will fail. The solution is already a fixed point, so additional iteration does not produce further changes. The difficulty is that by assuming $Pre_B(i, C, y, \emptyset) = Use_B(i, C, y, \emptyset) = 1$ and using these to describe the call to C results in a self-fulfilling prophecy in that assumptions for this call become the result for C. This solution is, however, safe for the purposes of optimizing transforms, though excessively pessimistic. It will result in more definitions and uses being considered than actually are necessary, thus reducing the scope for optimization. However, invalid optimizations will not be performed. Allen's strategy [11] of performing analysis on called procedures prior to analyzing their callers requires these kinds of pessimistic assumptions but is safe without further iteration.

The most precise solution for C in Fig. 3 is

$$Def_B(i, C, y, \emptyset) = 1,$$

$$Pre_B(i, C, y, \emptyset) = Use_B(i, C, y, \emptyset) = 0.$$

This solution can be achieved by initially assuming that Def_B, Use_B, and Pre_B are all zero. This represents a best-case initial assignment. This assignment is not safe, however, since it may permit an optimizing transformation that should not be performed. Thus, this best-case initial assignment must be improved by iteration until a fixed point is achieved. Rosen [12] uses this strategy and shows that such a fixed point is reachable and is the minimal (most precise) solution.

Thus we have two fundamentally different strategies. 1) Make a pessimistic assignment for recursive calls. Then analyze procedures in reverse invocation order (analyze called procedures prior to their callers). Iteration to find a solution is optional since such assignments are safe. The minimal solution might not be achieved because of "self-fulfilling prophecies." 2) Make an optimistic assignment for recursive calls. Iteration is now required since such assignments are not safe. The fixed point solution found will be the minimal (most precise) solution.

• *Argument-parameter aliasing*

Whichever of the foregoing strategies is chosen, the effects of argument-parameter aliasing must be computed and such effects must be propagated systematically. One might compute a worst case, which assumes that parameters may be aliased to all of the arguments that are passed to them from all of the calls. This is done in the context of local data flow computation and is further elaborated in a later section. On the other hand, one might carefully distinguish the separate argument-pa-

rameter aliasing cases and compute a separate result for each such equivalence. Thus, one is assured of computing a precise result at the expense of a potentially large number of computations.

The approach we pursue here for coping with aliasing can be used with either Allen's reverse invocation order strategy or Rosen's iteration strategy. What we present, via a set of claims which are justified below, is a method for approximating the separate argument-parameter aliasing cases using computations that assume there is no aliasing. We list the approximations as claims first and then describe briefly how they can be used.

Claim DE

$$Def_B(i, C, x, P \wedge E(J)) = Def_B(i, C, x, P) \vee \bigvee_{j \in J} Def_B(j, C, x, P).$$

Justification
If i names the same variable as does each j in J, then a definition for one is a definition for all of them. That is exactly what the claim DE states.

Claim PE

$$Pre_B(i, C, x, P \wedge E(J)) \leq Pre_B(i, C, x, P) \wedge \bigwedge_{j \in J} Pre_B(j, C, x, P).$$

Justification
1. If C is primitive, then

$$Pre_B(i, C, x, P \wedge E(J)) = \neg Def_B(i, C, x, P \wedge E(J))$$
$$= \neg (Def_B(i, C, x, P) \vee \bigvee_{j \in J} Def_B(j, C, x, P))$$
$$= \neg Def_B(i, C, x, P) \wedge \bigwedge_{j \in J} \neg Def_B(j, C, x, P)$$
$$= Pre_B(i, C, x, P) \wedge \bigwedge_{j \in J} Pre_B(j, C, x, P).$$

2. If C is not primitive, then we have the following cases:
   a. $Pre_B(k, C, x, P) = 0$ for some $k \in (J \cup \{i\})$. On every path from C's entry to exit x, there is an arc on which k is not preserved, i.e., all paths are blocked by some definition of the variable named by k. Hence, since k names the same variable as the variables in J, it must be the case that

$$Pre_B(i, C, x, P \wedge E(J)) = 0 = Pre_B(i, C, x, P) \wedge \bigwedge_{j \in J} Pre_B(j, C, x, P).$$

   b. $Pre_B(j, C, x, P) = 1$ for all $j \in (J \cup \{i\})$; then

$$Pre_B(i, C, x, P \wedge E(J)) \leq 1 = Pre_B(i, C, x, P) \wedge \bigwedge_{j \in J} Pre_B(j, C, x, P).$$

   Equality cannot be assured since, if p and p' are distinct paths, we have the following possibility: variable i is preserved on p, variable j is not preserved on p, while variable i is not preserved on p' but variable j is preserved on p'. When i is aliased to j, the combined variable should not be preserved even though i and j are separately preserved.

Given only two variables and considering all their combinations of values for *Def* and *Pre*, of which there are 16, nine combinations represent valid possibilities and in eight out of nine equality holds. For one case out of nine, when both are preserved, the inequality represents the most precise information obtainable from the separate summary information. Even in this case, however, equality may hold.

Claim UE

$$Use_B(i, C, x, P \wedge E(J)) \leq Use_B(i, C, x, P) \vee \bigvee_{j \in J} Use_B(j, C, x, P).$$

Justification
As in the Def_B case, a reference to any j in J must be considered as a reference to all. Hence, $Use_B(i, C, x, P \wedge E(J))$ can be no more than the above disjunction. Equality cannot be established, however, because a definition of one of the variables may block all paths to a use of another variable where such a use was not blocked by definitions of that variable itself. (This is the same argument used for Pre_B.)

Given only two variables, all combinations of Def_B and Use_B are possible, and equality holds in eleven out of the sixteen combinations. Thus, the disjunction is a reasonable approximation.

The strategy suggested by the above claims is to compute the "no aliasing" case, i.e., for any variable i in a block C, exit x, one computes $Def_B(i, C, x, \varnothing)$, etc. These "no aliasing" computations require the computation of quantities involving aliasing, of course. Whenever one of these cases arises, however, one of the preceding approximation schemes can be substituted. This strategy produces a result which is almost as precise as that produced by multiple aliasing computations while not suffering from the potential combinatorial explosion. The result will be much better than a single worst-case computation that is used for every call. It has the advantage of being no worse computationally while permitting the analysis to be tailored to each call point's particular aliasing situation.

Local data flow

Summary data flow analysis, while interesting for program diagnosis and documentation, does not provide the

information required for many of the optimizing transformations performed by compilers, such as constant propagation, common expression elimination, dead variable elimination. Local data flow analysis is needed here as it may be necessary to identify precisely the set of definitions (updates) that can reach a computation and the set of uses that might still be affected by previous computations. In the exposition that follows, only the "reaching definitions" computation is treated. The computation for "exposed uses" proceeds in an analogous fashion (see [2]).

We wish to know precisely what set of definitions at the various arcs of a control flow graph can, in fact, affect a computation at some other arc. This set is called the set of *reaching* definitions. Before defining it, however, we must describe where the elements of the set, i.e., the definitions, originate. Each element is represented as $\mathbf{d}(i, l)$, where

$DEF_A(i, l, P) = \{\mathbf{d}(i, l)\}$ if $Def_A(i, l, P) = 1$; otherwise $DEF_A(i, l, P) = \varnothing$. (L1)

Then the set of reaching definitions for a variable i, at an arc l, under condition P, i.e., those that reach the "end" of arc l, is defined as

$REACH_A(i, l, P) = \cup \{DEF_A(i, m, P) \mid$ there is a path originating from arc m and terminating with arc l, all arcs n of which have $Pre_A(i, n, P) = 1.\}$ (L2)

We hasten to add that the computation of $REACH_A$ does not use the above definition. Rather, a definition modeled on Allen's [2] is used, i.e.,

$$REACH_A(i, l, P) = \bigcup \{REACH_A(i, m, P) \,\&\, Pre_A(i, l, P) \mid m \in pred(l)\} \cup DEF_A(i, l, P) \quad (L2')$$

where SET & $0 = \varnothing$ and SET & 1 = SET; $pred(l)$ = the set of predecessors of arc l, i.e., the arcs of the node of which l is an out arc.

The set $REACH_A$, since it is drawn from "may be" information, i.e., $Def_A(i, l, P)$ and $Pre_A(i, l, P)$, represents the set of definitions that *may* reach an arc.

The problem faced in computing $REACH_A$ is as follows. We want the local data flow information, e.g., $REACH_A$, to be the best that we can compute in the light of multiple calls. Thus we must compute $REACH_A$ information under conditions that are safe to assume for all the calls of interest.

While a solution to the above problem is essential, it is also highly desirable to be able to provide both local and summary data flow information that is specialized to a particular call point. This serves three purposes:

1. It is valuable as documentation.
2. It provides more precise results in the analysis of the calling program.
3. It permits the optimization of a tailored version of the called procedure which is suitable for a particular call or subset of calls.

Thus, what we develop is a variant of the Cocke-Allen type of local data analysis flow [2] in which worst-case data flow information is computed but which, simultaneously and with trivial additional computation, permits this information to be tailored to any particular call point aliasing situation.

In the absence of precise knowledge concerning argument-parameter aliasing, we must be able to compute data flow information under the "may be aliased" condition M:

$M(J)$ When data flow for a variable is analyzed under the "may be aliased" condition $M(J)$, the results must be safe both for the $E(J)$ condition (i.e., the condition under which the variable aliases any or all variables in set J) and for the $\varnothing$ condition (i.e., the variable is not aliased to any other variable). (L3)

When analyzing a procedure in isolation from its callers, we must use worst-case assumptions, i.e., that any parameter "may be aliased" to any other parameter and to any global, assuming compatible types. Any global may be aliased to any parameter, again assuming compatible types. Thus, we wish to compute, for each parameter i, on each arc l, and with the sets of all type-compatible parameters K and globals J, $REACH_A(i, l, M(K \cup J))$. For each global variable i under the same conditions we compute $REACH_A(i, l, M(K))$. As with the summary data flow we can choose to approximate these quantities in terms of the corresponding quantities under the null condition. Thus, it is the case that

$$REACH_A(i, l, P \wedge M(J)) \subseteq REACH_A(i, l, P) \cup \bigcup_{j \in J} REACH_A(j, l, P).$$

Unfortunately, this is not a good approximation as the simple example of Fig. 4 shows. In Fig. 4(a), the definition of i at arc m should make all previous definitions of i (and j) unavailable after m, but this is not captured by the approximation.

The strategy we pursue is to precisely characterize the $REACH_A$ definitions under each of the "may be aliased" conditions and use the information developed as the basis for deriving good approximations for other conditions.

To accomplish our goal, we need yet a third condition under which data flow analysis is performed. This is the "interferes" condition $I(J)$ defined as follows:

$I(J)$ When data flow for a variable is analyzed under condition $I(J)$, the *preserved* information used in the computation is the same as that produced under the $E(J)$ condition. Thus, all definitions of variables in J "interfere with" the propagation of the definitions or uses of the variable of interest. (L4)

For the $REACH_A$ computation, we then have

$$REACH_A(i, m, P \wedge M(J)) = REACH_A(i, m, P) \cup \bigcup_{j \in J} REACH(j, m, P \bigwedge I(\{i\})). \quad (L5)$$

This simply asserts that the $REACH_A$ set of definitions of i for the "may be aliased" case consists of all the definitions for i in the unaliased case, together with those definitions of j in J that are not killed, i.e., are preserved, by definitions of both i and j. In Fig. 4(a), $REACH_A(i, m, \emptyset) = \{\mathbf{d}(i, m)\}$ and $REACH_A(j, m, I(\{i\})) = \emptyset$ so that $REACH_A(i, m, M(\{j\})) = \{\mathbf{d}(i, m)\}$. However, in Fig. 4(b), $REACH_A(i, m, \emptyset) = \{\mathbf{d}(i, l)\}$ and $REACH_A(j, m, I(\{i\})) = \{\mathbf{d}(j, m)\}$ so that $REACH_A(i, m, M(\{j\})) = \{\mathbf{d}(i, l), \mathbf{d}(j, m)\}$.

• *Tailoring may be aliased information*

While information which can be relied on to be safe for all calls of a procedure is essential, we have argued that it is frequently useful to have information which is specific to a given call of the procedure so that a tailored form of the procedure can be used. Information specific to a call can make use of known aliasing conditions, i.e., one can compute the $REACH_A$ information using the precise $E(J)$ conditions that relate arguments to parameters. A separate analysis of a procedure in order to generate $REACH_A$ information for each $E(J)$ condition is not necessary, however, as a good approximation to this information is computable from the "may be aliased" $M(J)$ computation by making separate use of the part of this computation done using the $I(J)$ condition. This strategy relies on the following three claims.

Claim RE1

$$REACH_A(i, l, P \wedge E(J)) = REACH_A(i, l, P \bigwedge I(J)) \cup \bigcup_{j \in J} REACH_A(j, l, P \bigwedge I(\{i\} \cup (J - \{j\}))).$$

Justification

The definitions for all the aliased variables must be included, subject to the *preserved* information of the aliased variables. Each $REACH_A(k, l, P \wedge I(L))$ represents the set of definitions contributed by a variable k subject to the $Pre_A(k, m, P \wedge E(L))$ condition. The union of these sets then constitutes all the reaching definitions of i under the $E(J)$ condition.

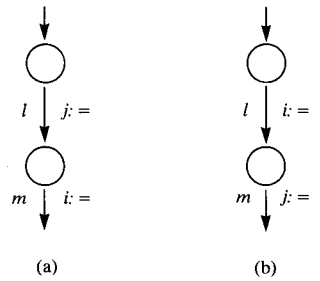

Figure 4 In (a), $REACH_A(i, m, M(\{j\})) = \{\mathbf{d}(i, m)\}$ since the definition of i "kills" all previous definitions. However, in (b), $REACH_A(i, m, M(\{j\})) = \{\mathbf{d}(i, l), \mathbf{d}(j, m)\}$ since j is only a possible alias of i and cannot be used to kill previous definitions of i.

Claim RE2

If $\mathbf{d}(i, l) \in REACH_A(i, m, P \wedge E(J))$, then for all $j \in J$, if $\mathbf{d}(j, l) \in DEF_A(j, l, P)$, then $\mathbf{d}(j, l) \in REACH_A(j, m, P)$.

Justification

Since

$$Pre_A(i, n, P \wedge E(J)) \leq Pre_A(i, n, P) \wedge \bigwedge_{j \in J} Pre_A(j, n, P)$$

by Claim PE, if definitions survive $Pre_A(i, n, P \wedge E(J))$, they must surely survive each of the terms on the right above. Hence a path which permits definitions from l to reach node m under the preserved function $Pre_A(i, n, P \wedge E(J))$ will surely permit definitions to survive under any one of the $Pre_A(j, n, P)$ functions.

Claim RI

$$REACH_A(i, l, P \wedge I(J)) \subseteq \bigcap_{j \in J} REACH_A(i, l, P \wedge I(\{j\})).$$

Justification

The preserved function used to compute $REACH_A(i, l, P \wedge I(J))$ is $Pre_A(i, l, P \wedge E(J))$, which is less than or equal to

$$\bigwedge_{j \in J} Pre_A(i, l, P \wedge E(\{j\}))$$

by Claim PE, each term of which represents the preserved function in the computation of the $REACH_A(i, l, P \wedge I(j\}))$ terms. The DEF_A set for $REACH_A(i, l, P \wedge I\{j\})$ is $DEF_A(i, l, P \wedge I(J))$, which is a subset of $DEF_A(i, l, P \wedge I(\{j\}))$, the DEF_A set used for the $REACH_A(i, l, P \wedge I\{j\})$ terms.

The $REACH_A(i, l, P \wedge M(J))$ computation involves a separate $REACH_A(i, l, P \wedge I(\{j\}))$ computation for all $j \in J$. The RI approximation above must be used to ap-

proximate the $I(J)$ computation whenever J has more than one member. Thus, precise $REACH_A(i, l, P \wedge E(J))$ information is computable using RE1 above only if pairwise aliasing occurs. If, for instance, repeated arguments result in three or more variables being aliases of each other, then RI must be used to produce an approximation to the $REACH_A(i, l, P \wedge I(J))$ computation before RE1 can be applied. Claim RE2 can then be used to improve the result by eliminating some of the extra definitions. Using RE1, RE2, and RI thus permits the $REACH_A$ information for the $E(J)$ condition to be approximated rather precisely without a complete recalculation by deriving the result from the $M(J)$ (and hence $I(\{j\})$) computations.

• *Summary data flow information*

If certain provisions are taken [11, 12], it becomes possible to compute both Def_B and Pre_B information using the results of the $REACH_A$ computation. The Use_B information can also be computed from the local "exposed uses" information in an analogous fashion. The special provisions have to be made for the Pre_B computation, but we must be aware of these provisions in order that Def_B be properly done.

In order to compute Pre_B for each variable i in block C, we create a hypothetical definition for i on the input arc of the control flow graph for C. We denote this arc as arc 0 and the definition as $\mathbf{d}(i, 0)$. We then perform the $REACH_A$ computation as before. The important property of $\mathbf{d}(i, 0)$ is that all definitions of i from outside of C that reach the beginning of C will also reach the arcs reached by $\mathbf{d}(i, 0)$. Thus, we have

Claim RP

$Pre_B(i, C, x, P) = 1$ iff $\mathbf{d}(i, 0) \in REACH_A(i, x, P)$.

Justification

A definition at the entry point of the block is available at an exit point, if and only if there is a path from the entry point to the exit, each arc of which preserves the definition.

Finally, we can determine Def_B from the $REACH_A$ results by using the following approach.

Claim RD

$Def_B(i, C, x, P) = 1$ iff $REACH_A(i, x, P) - \{\mathbf{d}(j, 0) \mid \text{for all } j\} \neq \emptyset$.

Justification

If any definition is available on exit, other than the hypothetical ones at arc 0, then it is a locally generated definition and $Def_B(i, C, x, P) = 1$. If no such definition exists, then no path to exit x contains a definition of i and $Def_B(i, C, x, P) = 0$.

• *Aliases and the REACH computation*

By using the preceding claims, we are now in a position to cope with the aliasing conditions that arise during the $REACH_A$ computation. First let us describe the $REACH_A$ results that we desire. Ultimately, we wish to compute the $REACH_A$ information for the "may be aliased" conditions so that optimizing transformations can be applied that will be valid no matter how a given procedure is called. We also wish, however, to use the component parts of the preceding in order to be able to tailor the $REACH_A$ results so that accurate summary data flow information can be derived for each call point.

As indicated previously, the "may be aliased" computation can be decomposed into an "unaliased" computation and an "interferes" computation. We make use of (L5) where P is taken as the null condition. Thus we need to compute the following in a given block or procedure.

a. For each variable or parameter i, $REACH_A(i, l, \emptyset)$ for every arc l.
b. For each variable i that is global to the block/procedure and that might be used as an argument, $REACH_A(j, l, I(\{i\}))$ for all parameters j to which it might be passed.
c. For each parameter i that might be passed the same argument as is passed to some other parameter j, $REACH_A(j, l, I(\{i\}))$.
d. For each parameter i, and for all global variables j that might be passed as arguments to it, $REACH_A(j, l, I(\{i\}))$.

We now concentrate on establishing the DEF_A and Pre_A values required for the computations in (a) through (d) above.

1. DEF_A: Recall from (L1) that $DEF_A(i, l, P) = \{\mathbf{d}(i, l)\}$ iff $Def_A(i, l, P) = 1$ and is null otherwise. Further $Def_A(i, l, P) = Def_B(i, C, x, G(C, P \wedge E(J)))$ from (S2), where block C, exit arc x, serves to define arc l of our procedure. The function G eliminates all conditions involving variables not global to C, and J is the set of parameters in C to which i is passed.

   In the worst case, $P \wedge E(J)$ will survive as the condition of interest, i.e., $G(C, P \wedge E(J)) = P \wedge E(J)$, other cases being simpler. But

$$Def_B(i, C, x, P \wedge E(J)) = Def_B(i, C, x, P) \cup \bigcup_{j \in J} Def_B(j, C, x, P).$$

   Since P is either null ($\emptyset$) or $I(\{k\})$ for some variable k, $Def_B(i, C, x, P)$ is either $Def_B(i, C, x, \emptyset)$ or $Def_B(i, C, x, I(\{k\}))$ and similarly for each of the j's. For those arcs realized by blocks that themselves have

control flow graphs, we use Claim RD, which yields $Def_B(i, C, x, P) = 1$ iff $REACH_A(i, x, P) - \{\mathbf{d}(j, 0) \mid \text{for all } j\} \neq \emptyset$.

The result of this chain of formulas is to relate the required DEF_A information for an arc to the $REACH_A$ information concerning the block and exit that serves to define this arc. But this reach information is of exactly the form computed in (a) through (d). Further, none of the formulas used was an inequality and hence the computation above is precise.

In the case where the block C realizing a given arc is primitive, we need some other way to compute $Def_B(i, C, x, \emptyset)$ and $Def_B(i, C, x, I(\{k\}))$. But $Def_B(i, C, x, \emptyset)$ is given for primitives. Further $Def_B(i, C, x, I(\{k\})) \leq Def_A(i, C, x, \emptyset)$, equality holding when either $Def_B(i, C, x, \emptyset) = 0$ or when $Def_B(k, C, x, \emptyset) = 0$, i.e., when there are either no definitions of i or no definitions of k to interfere with those of i. The strict inequality may apply if both of the preceding quantities equal one. Normally, however, primitives only update a single variable and, under these circumstances, equality always holds. Thus we can use $Def_B(i, C, x, \emptyset)$ to approximate $Def_B(i, C, x, I(\{k\}))$.

2. Pre_A: Using (S3), we have $Pre_A(i, l, P) = Pre_B(i, C, x, G(C, P \wedge E(J)))$ again where block C, exit x, serves to define arc l. Again, in the worst case, $P \wedge E(J)$ will survive as the condition of interest, other cases being simpler. But from Claim RP, $Pre_B(i, C, x, P \wedge E(J)) = 1$ iff $\mathbf{d}(i, 0) \in REACH_A(i, x, P \wedge E(J))$, while from RE1,

$$REACH_A(i, x, P \wedge E(J)) = REACH_A(i, x, P \wedge I(J)) \cup \bigcup_{j \in J} REACH_A(j, x, P \wedge I(J - \{j\}) \cup \{i\}).$$

Claim RE2 then requires that $\mathbf{d}(k, 0)$ for all k in $(J \cup \{i\})$ be in $REACH_A(k, x, P)$ in order for $\mathbf{d}(i, 0)$ to be truly in $REACH_A(i, x, P \wedge E(J))$.

The condition P is either $I(\{k\})$ for some variable k or it is null. Thus, the problem reduces to one of computing $REACH_A(i, x, I(J'))$. When J' is the unit set or null, it is computed precisely. With interference conditions involving more than one variable, Claim RI must be used, i.e., $REACH_A(i, l, I(J'))$ is approximated by

$$\bigcap_{j \in J'} REACH_A(i, l, I(\{j\})).$$

The preceding formulation for producing DEF_A and Pre_A can be used in a number of ways to compute $REACH_A$ information. One can use Allen's inverse invocation order strategy to compute results for called blocks prior to computing results for their callers, or one can use Rosen's strategy, which involves solving the set of equations for calling and called procedures simultaneously by iteration to find a fixed point solution. Further, the local computation of $REACH_A$, if one employs the inverse invocation order strategy, can be performed in several ways (see [2–4]).

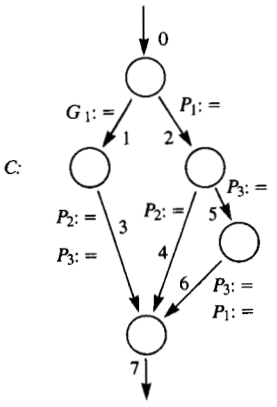

Figure 5 Control flow graph of block C indicating the definitions that appear on the various arcs.

• *Redundant computations*

The computational demands of local data flow can be reduced if all redundant computations are eliminated. In this context, we wish to find definitions that will be present in a $REACH_A$ set at any arc under a given condition if and only if another definition is present in some other $REACH_A$ set. Then, the computation involving only one of the definitions need be performed. The following claims identify such redundancies.

Claim RR1

If $\mathbf{d}(i, l) \in DEF_A(i, l, P \wedge I(\{j\}))$ and $\mathbf{d}(j, l) \in DEF_A(j, l, P \wedge I(\{i\}))$, then, for all arcs x, $\mathbf{d}(i, l) \in REACH_A(i, x, P \wedge I(\{j\}))$ iff $\mathbf{d}(j, l) \in REACH_A(j, x, P \wedge I(\{i\}))$.

Justification

Both $\mathbf{d}(i, l)$ and $\mathbf{d}(j, l)$ originate at the same arc l and are subjected to interference (are either preserved or not) by all other definitions of both i and j.

Claim RR2

If there are no definitions of variable j, then $\mathbf{d}(i, l) \in REACH_A(i, l, P \wedge I(\{j\}))$ iff $\mathbf{d}(i, l) \in REACH_A(i, l, P)$.

Justification

Since there are no definitions for variable j, they cannot affect the propagation of the $REACH_A$ information and hence the $I(\{j\})$ condition has no effect.

When bit vector methods are used to compute the $REACH_A$ information, the preceding claims can be used to permit a single bit to serve multiple purposes. This will be illustrated using a concrete example in the next section.

Table 1 Correspondence table relating definitions of variables to bit positions for the local data flow computation of Fig. 5.

Variable	*Condition*	*Definition on arc represented by bit?*							
		0	1	2	3	4	5	6	7
G_1	$\varnothing$	1	2						
	$I(P_1)$	3	4						
	$I(P_2)$	5	6						
	$I(P_3)$	7	8						
G_2	$\varnothing$	9							
	$I(P_1)$	10							
	$I(P_2)$	11							
	$I(P_3)$	12							
P_1	$\varnothing$	10*+		13				14	
	$I(G_1)$	3+		15				16	
	$I(G_2)$	10+		13*				14*	
	$I(P_2)$	17		18				19	
	$I(P_3)$	20		21				22	
P_2	$\varnothing$	11*+			23	24			
	$I(G_1)$	5+			25	26			
	$I(G_2)$	11+			23*	24*			
	$I(P_1)$	17+			27	28			
	$I(P_3)$	29			30	31			
P_3	$\varnothing$	12*+			32		33	34	
	$I(G_1)$	7+			35		36	37	
	$I(G_2)$	12+			32*		33*	34*	
	$I(P_1)$	20+			38		39	22+	
	$I(P_2)$	29+			30+		40	41	

+Claim RR1 is used to eliminate redundancy.
*Claim RR2 is used to eliminate redundancy.

Example

In order to illustrate how the preceding treatment of aliasing might be used in practice, we provide an example and use bit vectors to represent the sets that we require. We do not present a real program in any particular language but merely a control flow graph on whose arcs we indicate the variables that are defined and those that are preserved. The nodes themselves can be any of the blocks we have previously discussed. We assume that the processing has proceeded in inverse invocation order and that the Def_A and Pre_A information is thus available. Figure 5 is the annotated control flow graph for our example called block C.

The variables G_i, $i = 1, 2$, denote global variables, whereas P_j, $j = 1, 2, 3$, denote parameters. We assume that all G_i and P_j are type-compatible and are potential aliases of each other. For simplicity, the presence of a definition for a variable on an arc implies that the variable is not preserved on that arc. The $REACH_A$ sets that will interest us at each arc involve each variable under both the null ($\varnothing$) condition and the interferes conditions, with its possible aliases taken singly. Claim R1 is used to approximate the higher order interference conditions.

Since these sets will be represented by bit vectors, the first task is to lay out the format of these bit vectors and indicate what each bit is to represent. Table 1 relates each $REACH_A$ set computation to a position in the bit vectors. Note that definitions on arc 0 are included for all variables in order to perform the "preserved" computation. Table 2 provides the results of the $REACH_A$ computation. Bit vector $P(l)$ represents the set of definitions preserved at arc l. And $D(l)$ represents the set of definitions originating at arc l, i.e., those in $DEF_A(i, l, P)$ for some variable i and condition P. The set of definitions that reach arc l (survive and are available after arc l has been traversed) is represented by $R(l)$. We provide $D(l)$ and $P(l)$ to permit the reader to easily verify the correctness of the $R(l)$ results by using (L2′) in the obvious fashion.

Using the $REACH_A$ results of Table 2, we are in a position to apply our claims in order to compute quantities that may be of interest to us. We illustrate a few of these below.

1. $REACH_A(G_1, m, M(\{P_1, P_2, P_3\}))$
$= REACH_A(G_1, m, \varnothing)$
$\cup\, REACH_A(P_1, m, I(\{G_1\}))$

Table 2 Results of local data flow computation for the example of Fig. 5. The meaning of each bit is given in Table 1.

Set	*Bit Positions*								
	1-5	6-10	11-16	16-20	21-25	26-30	31-35	36-40	41
$P(0)$	-----	-----	-----	-----	-----	-----	-----	-----	-
$D(0)$	10101	01011	11000	01001	00000	00010	00000	00000	0
$R(0)$	10101	01011	11000	01001	00000	00010	00000	00000	0
$P(1)$	00000	00011	11110	01111	11110	01111	11110	00111	1
$D(1)$	01010	10100	00000	00000	00000	00000	00000	00000	0
$R(1)$	01010	10111	11000	01001	00000	00010	00000	00000	0
$P(2)$	11001	11110	11000	00000	00111	10011	11111	11001	1
$D(2)$	00000	00000	00101	00100	10000	00000	00000	00000	0
$R(2)$	10001	01010	11101	00100	10000	00010	00000	00000	0
$P(3)$	11110	00011	00111	10000	00000	00000	00000	00000	0
$D(3)$	00000	00000	00000	00000	00101	01001	01001	00100	0
$R(3)$	01010	00011	00000	00000	00101	01001	01001	00100	0
$P(4)$	11110	01111	01111	10001	11000	00000	01111	11110	0
$D(4)$	00000	00000	00000	00000	00010	10100	10000	00000	0
$R(4)$	10000	01010	01101	00000	10010	10100	10000	00000	0
$P(5)$	11111	10011	10111	11110	00111	11100	00000	00000	0
$D(5)$	00000	00000	00000	00000	00000	00000	00100	10011	0
$R(5)$	10001	00010	10101	00100	00000	00000	00100	10011	0
$P(6)$	11001	10010	10000	00000	00111	10000	00000	00000	0
$D(6)$	00000	00000	00010	10010	01000	00000	00010	01000	1
$R(6)$	10001	00010	10010	10010	01000	00000	00010	01000	1
$P(7)$	11111	11111	11111	11111	11111	11111	11111	11111	1
$D(7)$	00000	00000	00000	00000	00000	00000	00000	00000	0
$R(7)$	11011	01011	11111	10010	11111	11101	11011	01100	1

$\cup\, REACH_A(P_2, m, I(\{G_1\}))$
$\cup\, REACH_A(P_3, m, I(\{G_1\}))$
by (L5).

We can create a mask which, when "anded" to a reach computation, selects the definitions of interest. This mask has "one" bits as follows.

a. In positions 1 and 2 for $\langle G_1, \varnothing\rangle$
b. In positions 3, 15, and 16 for $\langle P_1, I(G_1)\rangle$
c. In positions 5, 25, and 26 for $\langle P_2, I(G_1)\rangle$
d. In positions 7, 35, 36, and 37 for $\langle P_3, I(G_1)\rangle$.

Zeros occur everywhere else.
Examining these positions, we ascertain that, for example,

$REACH_A(G_1, 7\, M(\{P_1, P_2, P_3\})) = \{\mathbf{d}(G_1, 0), \mathbf{d}(G_1, 1), \mathbf{d}(P_1, 2), \mathbf{d}(P_1, 6), \mathbf{d}(P_2, 0), \mathbf{d}(P_2, 3), \mathbf{d}(P_2, 4), \mathbf{d}(P_3, 0), \mathbf{d}(P_3, 3), \mathbf{d}(P_3, 6)\}$.

The arc 0 definitions are only present for the preserved computation. From them we conclude that

$Pre_B(G_1, C, 7, M(\{P_1, P_2, P_3\})) = 1$,

and that

$Def_B(G_1, C, 7, M(\{P_1, P_2, P_3\})) = 1$.

2. $REACH_A(G_1, m, E(\{P_1\}))$
$= REACH_A(G_1, m, I(\{P_1\}))$
$\cup\, REACH_A(P_1, m, I(\{G_1\}))$
by Claim RE1.

We create a mask for this as in 1 above with "one" bits as follows:

a. In positions 3 and 4 for $\langle G_1, I(P_1)\rangle$,
b. In positions 3, 15, and 16 for $\langle P_1, I(G_1)\rangle$.

Then

$REACH_A(G_1, 7, E(\{P_1\})) = \{\mathbf{d}(G_1, 1), \mathbf{d}(P_1, 2), \mathbf{d}(P_1, 6))\}$, from which we conclude that

$Pre_B(G_1, C, 7, E(\{P_1\})) = 0$ by Claim RP,

and that

$Def_B(G_1, C, 7, E(\{P_1\})) = 1$ by Claim RD.

3. $REACH_A(G_1, m, E(\{P_2, P_3\}))$
$= REACH_A(G_1, m, I(\{P_2, P_3\}))$
$\cup REACH_A(P_2, m, I(\{G_1, P_3\}))$
$\cup REACH_A(P_3, m, I(\{G_1, P_2\}))$

by Claim RE1. Using Claim RI, this is included in

$(REACH_A(G_1, m, I(\{P_2\})) \cap REACH_A(G_1, m, I(\{P_3\}))) \cup$
$(REACH_A(P_2, m, I(\{G_1\})) \cap REACH_A(P_2, m, I(\{P_3\}))) \cup$
$(REACH_A(P_3, m, I(\{G_1\})) \cap REACH_A(P_3, m, I(\{P_2\}))).$

Unfortunately, when an intersection is required, a mask cannot be constructed that will identify the definitions desired. Rather, the members of each of the $REACH_A(i, m, I(\{j\}))$ sets must be determined and the intersections performed by examining the individual definitions. When this is done, we obtain, for arc 7,

a. $REACH_A(G_1, 7, I(\{P_2\})) = \{\mathbf{d}(G_1, 0)\}$ and $REACH_A(G_1, 7, I(\{P_3\})) = \{\mathbf{d}(G_1, 0)\}$ so that $REACH_A(G_1, 7, I(\{P_2, P_3\})) \subseteq \{\mathbf{d}(G_1, 0)\}$.

b. $REACH_A(P_2, 7, I(\{G_1\})) = \{\mathbf{d}(P_2, 0), \mathbf{d}(P_2, 3), \mathbf{d}(P_2, 4)\}$ and $REACH_A(P_2, 7, I(\{P_3\})) = \{\mathbf{d}(P_2, 3), \mathbf{d}(P_2, 4)\}$, so that $REACH_A(P_2, 7, I(\{G_1, P_3\})) \subseteq \{\mathbf{d}(P_2, 3), \mathbf{d}(P_2, 4)\}$.

c. $REACH_A(P_3, 7, I(\{G_1\})) = \{\mathbf{d}(P_3, 0), \mathbf{d}(P_3, 3), \mathbf{d}(P_3, 6)\}$ and $REACH_A(P_3, 7, I(\{P_2\})) = \{\mathbf{d}(P_3, 3), \mathbf{d}(P_3, 6)\}$, so that $REACH_A(P_3, 7, I(\{G_1, P_2\})) \subseteq \{\mathbf{d}(P_3, 3), \mathbf{d}(P_3, 6)\}$.

d. From the union of a, b, and c, we conclude that $REACH_A(G_1, 7, E(\{P_2, P_3\})) \subseteq \{\mathbf{d}(G_1, 0), \mathbf{d}(P_2, 3), \mathbf{d}(P_2, 4), \mathbf{d}(P_3, 3), \mathbf{d}(P_3, 6)\}$.

e. Finally, we can use Claim RE2 to eliminate some excess definitions in d. Note that $\mathbf{d}(P_2, 0)$ and $\mathbf{d}(P_3, 0)$ have not survived so that $REACH_A(G_1, 7, E(\{P_2, P_3\})$ cannot contain any definitions from arc 0. Thus we have

$REACH_A(G_1, 7, E(\{P_2, P_3\})) \subseteq \{\mathbf{d}(P_2, 3), \mathbf{d}(P_2, 4), \mathbf{d}(P_3, 3), \mathbf{d}(P_3, 6)\}$.

Equality, in fact, holds in this case.

The computation of intersections discussed under 3 can be made comparatively simple if none of the redundant computations are eliminated via Claims RR1 and RR2. Then contiguous segments of each bit vector represent the definitions reaching an arc under a given condition. Bit vector "anding" of these segments can then be used to compute the intersections of the corresponding sets. Claim RE2 would still be needed to improve the approximation produced by the union operation used in RE1.

Discussion

Once one understands the nature of the conditions imposed on the data flow computation, the claims we have made can be justified by means of straightforward reasoning about graphs and sets. Despite this simplicity the "aliasing problem" has previously been a troublesome one and the methodology we have suggested copes with it quite well. Not only does the computation remain feasible in a pragmatic sense, but the results provided are very good.

Basically, another way of describing the methodology is that we compute both unaliased information and the pair-wise aliasing information precisely. Then we use good approximations to compute higher order aliasing. The interesting fact is that the general "may be aliased" case requires only the two previous computations and not the higher order aliasing computations.

The approach we have described can be used for more than just the summary and $REACH_A$ computations. We mention two.

1. If the purpose of the local information is only to compute the summary information, none of the local definitions need be distinguished, i.e., they can all be folded into a single bit, distinct from the bit used for the arc 0 definition used for the "preserved" computation (which represents global definitions). The arc 0 bit will then continue to indicate, directly, whether a variable is preserved, while the local bit will indicate whether it is defined. Claim RR2 can continue to be used to eliminate redundant computations. Claim RR1 can only be used if, for two variables, whenever one has a definition on an arc, then so does the other. This is possible but not likely.
2. Computing "exposed uses" (also called the "live variable" computation) is treated in an analogous fashion. a) If we are interested in retaining knowledge concerning particular uses, then we must, as in the case for definitions, provide a computation for each such use under the unaliased and the single variable interference conditions. Claims RR1 and RR2 can be applied directly to this problem by substituting "*USES*" for "*REACH*" and $\mathbf{u}(i, l)$ (a use) for $\mathbf{d}(i, l)$ (a definition) and then doing the ordinary "exposed use" computation. The Claims RE and RI can be transformed into the analogous UE and UI claims. The same considerations concerning parameters and globals also apply. If the summary "preserved" information is desired, hypothetical uses of variables must be provided on *each* exit arc. If one of these uses

remains exposed at the block entry, then that variable is preserved from entry to the exit arc of the use. (This requires more "uses" to be introduced than analogous "definitions" which must simply appear on the *single* block entry arc.) b) If only live variable information is desired, i.e., whether there is an exposed use somewhere (we do not care which use it is), then all local uses of a variable can be folded into a single "local use" bit, though again it must be distinguished from the hypothetical exit arc uses. The previous claims have their analogs for this computation as well.

We have been discussing the aliasing problem introduced by parameters and hence have been dealing only with parameters and globals. The local variables cannot be aliased by the parameters and hence need only be computed under the $\emptyset$ or no aliasing condition with respect to parameters and globals. There are, however, purely local aliasing problems that we alluded to in the introduction. We have in mind address arithmetic as occurs when one uses constructs such as $A(I)$ and $P \rightarrow X$. These do not name variables but specify computations whose results name variables. Definitions or uses involving these computations must always be preserved, independently of the presence of definitions. Why? Because on each traversal of a path involving a definition for such an address computation, the variable named by the computation may be different. With this adjustment, the techniques relevant to argument-parameter aliasing can be applied also to this local aliasing.

Acknowledgments

We are indebted to B. Rosen and F. Allen for reading and commenting on an earlier draft of this paper. F. Allen also read this version and provided several suggestions for improvements as well as correcting a number of errors. Her encouragement of this work is also acknowledged. Finally, the referees provided several useful suggestions that improved the clarity of the presentation.

References

1. E. Lowry and D. Medlock, "Object Code Optimization," *Commun. ACM* **12,** 13 (1969).
2. F. E. Allen and J. Cocke, "A Program Data Flow Analysis Procedure," *Commun. ACM* **19,** 137 (1976).
3. S. L. Graham and M. Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis," *J. ACM* **23,** 1 (1976).
4. M. S. Hecht and J. D. Ullman, "A Simple Algorithm for Global Data Flow Analysis Problems," *SIAM J. Comput.* **4,** 519 (1975).
5. G. A. Kildall, "A Unified Approach to Global Program Optimization," *Proceedings of the ACM Symposium on Principles of Programming Languages*, October 1973, p. 194.
6. J. M. Barth, "An Interprocedural Data Flow Analysis Algorithm," *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, January, 1977, p. 119.
7. T. C. Spillman, "Exposing Side Effects in a PL/I Optimizing Compiler," *Information Processing 71*, North Holland Publishing Co., Amsterdam, 1972, p. 376.
8. T. C. Spillman, "Analysis and Documentation System," *Research Report RC 3706*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1972.
9. R. P. Tapscott, "ADS: the Source Listing Annotator," *Research Report RC 5065*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1974.
10. L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability," *Computing Surveys* **8,** 305 (1976).
11. F. E. Allen, "Interprocedural Data Flow Analysis," *Information Processing 74*, North Holland Publishing Co., Amsterdam, 1974, p. 398.
12. B. K. Rosen, "Data Flow Analysis for Recursive PL/I Programs," *Research Report RC 5211*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1975.

The author is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

Received March 9, 1977