Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables
(extended abstract)

William E. Weihl*

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

ABSTRACT

Interprocedural data flow analysis is complicated by the use of procedure and label variables in programs and by the presence of aliasing among variables. In this paper we present an algorithm for computing possible values for procedure and label variables, thus providing a call graph and a control flow graph. The algorithm also computes the possible aliasing relationships in the program being analyzed.

We assume that control flow information is not available to the algorithm; hence, this type of analysis may be termed "flow-free analysis." Given this assumption, we demonstrate the correctness of the algorithm, in the sense that the information it produces is conservative, and show that it is as precise as possible in certain cases. We also show that the problem of determining possible values for procedure variables is P-space hard. This fact indicates that any algorithm which is precise in all cases must also run very slowly for some programs.

* Author's current address: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139. This work was performed to fulfill the S.M. degree thesis requirements under the VI-A program, a cooperative program of the MIT Department of Electrical Engineering and Computer Science. The work was supported in part by a graduate fellowship from the Fannie and John Hertz Foundation.

1. INTRODUCTION

Program optimization in the presence of procedure calls is increasingly important because of the current emphasis on modularity and abstraction in program design. Since these design methodologies imply a heavy use of procedures and procedure calls, it is essential that analysis techniques used to compile these programs perform well when calls are used. This is important for several reasons. Among these are the facts that procedures are often fairly general, so that better code can be produced for them given the contextual information implied by their use in the program, that procedure calls imply some overhead which it might be desirable to avoid, and that lack of knowledge about a called procedure makes it difficult to determine information about the procedure which contains the call.

Standard data flow analysis techniques [2, 4, 9] have many problems when applied to programs containing procedure calls. The reason for this is that a procedure call is essentially a statement whose effects cannot be determined upon examining only the statement itself. Rather, the body of the procedure must be examined as well. Failure to examine the body of the procedure may result either in no optimization being performed, or in optimization being performed in small sections of the program which contain no procedure calls, with very pessimistic assumptions being made about the use of parameters and global variables. If the latter is the case then many valuable optimizations may be lost; for example, the presence of a procedure call within a loop might prevent code which is invariant inside the loop from being moved outside it.

These deficiencies have led to the development of a number of methods for interprocedural data flow analysis [3, 6, 7, 10] which produce summaries of the effects of each procedure (e.g., an indication of which variables are used, modified, etc.). These summaries

are used when analyzing invoking procedures, avoiding the problems mentioned above.

The methods proposed by Barth and Banning have a number of advantages over that proposed by Allen, including greater speed, fewer passes required over the code, and the handling of recursion. Rosen's method, though slower than the others mentioned here, can produce more precise information. Unfortunately these methods are not sufficiently general to be used for many languages. Specifically, all of these methods require a call graph. However, if procedure variables are included in the language, the call graph cannot be obtained through a simple scan of the text of the program being compiled. Further complications occur when aliasing [12] among variables in the program is possible. This can result from mechanisms such as pointers and call-by-reference parameter passing. These two mechanisms are the ones considered in this paper. As an example of the problems which aliasing can cause, a call on a procedure variable using call-by-reference could have the effect, depending on the value of the procedure variable at the time of the call, of assigning a procedure value to one of the parameters of the call. This fact must be taken into account in constructing the call graph, for if procedure A contains a call on procedure variable X, the call graph must contain arcs from the node for procedure A to the nodes for each procedure which X can have as its value.

Another language feature which complicates the situation is the use of label variables. Such a feature prevents the construction of a control flow graph until the possible values of all label variables in the program have been determined. Since a control flow graph is required for standard data flow analysis, it is necessary for some part of the analysis phase of the compiler to compute this range information for label variables before standard data flow analysis is performed.

2. RELATED WORK

Very little work has been done on the problem of handling procedure variables and pointers in performing interprocedural data flow analysis. Spillman [12] is the only one who addresses the problems associated with these language features. However, there are a number of limiting aspects to Spillman's work. First, the algorithm requires iteration in the presence of recursion. This can lead to large time requirements. Second, the algorithm is presented at a low level, making it difficult both to understand and to verify. Finally, the algorithm as presented is specific to certain features of PL/1. This, combined with the low level of presentation, makes it difficult to adapt the algorithm for use in compiling other languages.

Ryder [11] presents an algorithm which determines the call graph for a program with procedure parameters. The algorithm is designed to be used with Fortran and is meant to be portable across a wide range of machines. The portability constraint leads to limitations on the use of in-core storage; these limitations influenced the design of the algorithm. The intent to use the algorithm for Fortran leads to the assumption that there is no aliasing and no recursion. These factors all limit the general applicability of the algorithm.

The methods proposed by Allen, Barth, Banning, and Rosen solve the problems associated with procedure variables and pointers to varying degrees, usually by placing restrictions on the use of these features in the programs to be compiled. In the extreme case these features are not allowed at all.

3. PROPOSED SOLUTION

We propose to deal with the problems introduced by pointers, procedure variables, and label variables by first computing range information (i.e., lists of possible values) for procedure variables, thus providing a call graph, and then using one of the known methods, such as Barth's, for generating summaries. It is necessary to compute aliasing patterns as well as range information for pointers while computing range information for procedure variables, simply because procedure variables can acquire values as a result of aliasing with other variables. All of this information will be computed without taking control flow information into account. It will subsequently be used to compute range information for label variables and to generate summaries for procedures, prior to generating a control flow graph.

The context in which this work has been done is that of the Experimental Compiling System (ECS) project of the Computer Sciences Department at IBM's Watson Research Center. An overview of this project is given in [5]. Briefly, the goal of the project is to build a general purpose optimizing compiler which, given the appropriate source language definition and target machine description, can compile programs for that source language into code for that target machine. Clearly it will not be possible to compile all languages for all machines. However, it is hoped that it will be possible to compile a large class of languages for a large class of machines. For this reason the analysis and optimization phases should be formulated in a manner which is as independent as possible of language and machine.

The ECS compiler attempts to treat primitives of the intermediate language and previously analyzed procedures as uniformly as possible, at least as far as

analyzing a program which uses them. This is accomplished by associating with each primitive and each analyzed procedure a summary of its effects. Information contained in such a summary includes lists of variables which are used, modified, and preserved, as well as an indication of the copies which may be performed when the primitive or procedure is invoked. For example, the summary for a primitive which moves data from its second argument to its first argument would specify that the primitive uses and preserves its second argument, modifies its first argument, and performs a copy of its second argument into its first argument. The information about copies is used in propagating procedure, label, and pointer values.

This means that in order to analyze a program, we need either the code or a summary for every procedure which it calls. This might seem to force a compilation order for separately compiled programs, and to achieve reasonable optimization of the programs, it does. However, it is desirable to be able to compile programs separately, and to allow this to be done in any order. This creates two problems in analyzing a procedure. First, the effects of the caller, such as aliasing of parameters, may be unknown. Second, there may be calls on procedures for which no summary exists. In either case, the worst case must be assumed. In ECS the second case is treated by creating a summary for the unknown procedure, and ensuring that this summary specifies all of the possible effects of the procedure. The first case is somewhat more complicated; in effect, the program is treated as if there were a call on the procedure which causes all possible parameter aliasing, including with external variables. If any of the parameters are procedure variables the problems are compounded. This is because a call on the parameter could be a call to a procedure which was passed as an actual parameter, and the effects of this procedure are unknown. We omit the details of the handling of such situations.

We assume that we are given a collection of procedures, each of which consists of a set of instructions. Each instruction consists of an opcode, which indicates a call on either a primitive, a previously analyzed procedure, a procedure in the collection, or a variable, and a list of operands, each of which is either a variable or a constant. Expressions are not allowed as operands; rather, we assume that the computation of expressions has been expanded into sequences of instructions. Operands are assumed to be passed by reference.

For each opcode which is a primitive or a previously analyzed procedure we assume that there exists a summary for that opcode. This summary must specify all of the possible copies among parameters and globals in which a call on the opcode could result. We make no assumptions about the possible flow of control between instructions in a single procedure.

Given such a collection of procedures, the problem is to compute sets of possible values for procedure variables, sets of possible values for label variables, sets of variables which may be addressable through pointer variables, and sets of possible aliases for all variables. In the next section we show that this problem is inherently very difficult. In succeeding sections we present our solution in stages, demonstrating its correctness and evaluating its precision and time requirements. Following this we discuss briefly some characteristics of the alias relationships as computed by our method, and then summarize our results.

4. COMPLEXITY

In this section we show that the problem of determining possible values for procedure variables is P-space hard. We assume some familiarity with the term "P-space hard". A definition of this term and a discussion of its significance can be found in [1].

In fact, we prove the following theorem, which makes a stronger statement.

Theorem: Determining possible values for procedure parameters for programs in which there is no aliasing among variables, no nesting of procedure declarations, and no significant flow of control, and in which every procedure is formally reachable, is a P-space hard problem.

This theorem is significant for a number of reasons. First, it indicates that it is extremely unlikely that there is any efficient method for computing possible values for procedure variables. Second, and perhaps more important, it shows that even when assignment statements, aliasing, and nesting of procedure declarations are not allowed, it is *still* unlikely that an efficient method exists. Furthermore, this is still true when there is no significant control flow in the program being examined, other than that implied by the call graph. These results lead one to search for solutions to the problem which are approximate and reasonably efficient, such as the one presented in this paper. These solutions may not produce exact information, but they must produce *safe* information; i.e., information which is conservative in that the possible values determined should be a superset of the exact possible values. We make the restriction that all procedures be formally reachable since this is an assumption which is often made in compilers.

Proof: We make use of a theorem proved by Winklmann [16], in which he shows that deciding the property of formal reachability in programs without nested procedure declarations is a P-space hard problem. Formal reachability is defined in terms of a formal execution tree, which is a tree of calls where the nodes of the tree are pairs consisting of procedures and their environments. (This is not the same as reachability defined in terms of a call graph as it is usually used in compilers.)

Winklmann shows this by constructing a program **P** for a given Turing machine **M**, polynomial *s*, and input *w*, such that a procedure **HALT** in **P** is formally reachable if and only if **M**, when started in its initial state with *w* written on its tape and its head scanning the leftmost symbol of *w*, halts without its head ever moving outside the *s(n)* squares to the right of, and including, the tape square scanned at the start, where *n* is the length of *w*. **P** satisfies all of the requirements given in the statement of the theorem except for the restriction that all procedures be formally reachable, and can be constructed in polynomial time from a description of the Turing machine **M**, the input *w*, and the polynomial *s*. In addition, **P** has the property that there is a procedure $\mathbf{Q_f}$ such that $\mathbf{Q_f}$ is formally reachable in **P** if and only if **M**, when executed with input *w*, enters its final state without its head ever leaving the boundaries stated above. $\mathbf{Q_f}$ contains a single call to the procedure **HALT**.

Deciding whether **HALT** is formally reachable is therefore as difficult as deciding whether **M** halts in the required manner. Furthermore, **HALT** is formally reachable if and only if $\mathbf{Q_f}$ is formally reachable. Since the first parameter of $\mathbf{Q_f}$ has no possible values if $\mathbf{Q_f}$ is not formally reachable, and has exactly one possible value if $\mathbf{Q_f}$ is formally reachable, it follows that determining whether a given procedure parameter has a non-empty set of possible values is a P-space hard problem. This problem can be easily solved given the set of possible values for the procedure parameter, so determining sets of possible values for procedure parameters must also be P-space hard.

We have not yet proved the theorem, since **P** does not satisfy the restriction given in the statement of the theorem that all procedures in **P** must be formally reachable. However, it is possible to transform **P** into a program **P'** in which all procedures are formally reachable, and about which we can ask the same kind of question which we asked about **P**. Furthermore, the construction of **P'** from **P** can be done in time linear in the length of **P**. Details of this construction can be found in [15].

The basic idea in constructing **P'** is to introduce calls to every procedure in **P**, but to do so in such a way that it is still possible to talk about the formal execution of **P'** simulating the execution of the Turing machine **M** on input *w*. Since it will then be the case that $\mathbf{Q_f}$ is always called at least once, the question which we will ask about **P'** is whether the first parameter to $\mathbf{Q_f}$ has more than one possible value. This will be true if and only if $\mathbf{Q_f}$ is called more than once, which will be true if and only if **M**, when executed on input *w*, halts in the prescribed manner. Therefore, answering this question is P-space hard, and since this question can be easily answered given sets of possible values for procedure parameters, the theorem follows.

5. THE METHOD

As stated earlier, we will present the method in stages. We begin with the simplest case, a single procedure with no aliasing, and gradually allow more complexity in the program being analyzed until we have included reference parameters, pointers, and calls on procedure variables.

5.1. NO ALIASING

We will first consider propagating values within a single procedure. Given that there is a summary for every instruction in the procedure, create a relation named PVAL and initialize it to all pairs (A,B) such that B is copied into A. B may be an constant or a variable; A must be a variable. A pair (X,A) in PVAL means that X has possible value A. PVAL ranges over the variables in the program for which we wish to determine values and over the values which we are interested in propagating. We determine which copies are possible by examining the instructions in the procedure. For each instruction, consider each copy in the summary for the opcode of the instruction. If one of the elements of the copy is a formal parameter of the opcode, substitute the corresponding operand of the instruction. The resulting copy gives a pair which should be placed in PVAL. To propagate values, replace PVAL by its transitive closure PVAL^+. For variable X and constant A, the resulting relation gives an answer to the question of whether A is a possible value of X.

We claim that propagating values in this manner, for this limited case, is both correct and as precise as possible. To show that it is correct, suppose that a variable X has value A at some point during the execution of the program. For X to have value A, the execution of the program must include a sequence of assignments $X_{i+1} := X_i$, with X_0 being A and X_n being X. If this is the case, then each of these assignments must appear as a copy in the summary of some instruction in the program. Therefore each appears as a pair in the initial PVAL relation. From this it follows that the transitive closure of PVAL must

include the pair (X,A). Therefore the information computed is correct.

To show that it is as precise as possible, suppose that there is a pair (X,A) in the transitive closure of PVAL. There must exist a sequence of pairs (X_{i+1},X_i) in the initial PVAL relation, with X_0 being A and X_n being X. Each of these pairs corresponds to a copy specified by the summary for some instruction in the procedure. Since we are making no assumptions about the possible flow of control between instructions in the program, any sequence of instructions must be considered possible. In particular, the sequence of instructions which corresponds to the given sequence of copies must be considered possible. This means that, ignoring control flow information, X may have A as value. Therefore the information computed is precise.

The complexity of this algorithm is bounded by the complexity of computing the non-reflexive transitive closure of an $n \times n$ matrix, with n being the total number of variables and values involved in the propagation. Under the assumptions made up to this point, this is asymptotically the best possible algorithm for computing this information. We can show this by demonstrating that computing this information is of the same complexity as computing the transitive closure of a matrix.

We consider a single procedure and assume that we have no control flow information. If $P(n)$ is the time to propagate values for a program containing n variables and constants, and $T(n)$ is the time to compute the transitive closure of an $n \times n$ matrix, we must show that there exists a constant c such that $T(n) \leq cP(n)$. Suppose that we have an $n \times n$ binary matrix M and we wish to compute its transitive closure. We first create variables X_i and constants A_i, for $1 \leq i \leq n$. For each X_i we create an instruction whose summary indicates a copy to X_i from A_i. For each 1 in the matrix, say at position (i,j), we create an instruction whose summary indicates a copy to X_i from X_j. These instructions constitute the procedure for which we wish to propagate values. We claim that X_i has possible value A_j if and only if there is a 1 in position (i,j) in the transitive closure of M. This implies that $T(n) \leq P(2n)$. Since P(n) is bounded by the time to compute transitive closure, and this is $O(n^3)$, we can assume that $P(2n) \leq 8P(n)$. From this we conclude that $T(n) \leq 8P(n)$.

Now consider the situation in which the program to be analyzed consists of multiple procedures, and in which instructions may be calls on primitives, previously analyzed procedures, or procedures in the given collection. Operands to primitives and previously analyzed procedures are passed by reference while operands to procedures in the collection are passed by value. Propagating values in this situation is almost identical to propagating values in the case of a single procedure. The only difference is that we must account for the transmission of values from actual parameters to formal parameters. This can be done in initializing the relation PVAL. For each call on a procedure P in the collection, where P is declared with formal parameters X_i, and the call has corresponding operands Y_i, add the pairs (X_i,Y_i) to PVAL. For each other instruction initialize PVAL as before. Then form the transitive closure $PVAL^+$. We claim that, as in the case of a single procedure, this computes correct and completely precise information, and does so asymptotically as quickly as possible. The proof is quite similar to the previous proof; we omit the details.

5.2. CALL BY REFERENCE

We now wish to allow parameters to procedures in the collection to be passed by reference. This means that when a value Y is copied into a variable X, there is an implied copy of Y into each alias of X. There are now two different effects to consider. The first is the modification of a variable by assignment to it. The second is the association of a formal parameter with an actual parameter by a call to the procedure which owns the formal. To keep track of this information, we create two relations called MODVAL and AFFECT. A pair (X,A) in MODVAL means that X is assigned value A. A pair (X,A) in AFFECT means that X may be aliased to A and to every other variable which may be aliased to A. However, it can be the case that there is some variable which may be aliased to X but not to A. The characterization of parameter aliasing with AFFECT was first suggested by Barth [7]. MODVAL is initialized to all copies which are specified in the summaries of instructions in the program. AFFECT is initialized to all formal-actual parameter pairs which result from calls to procedures in the collection. Assume for the moment that constants are never used as actual parameters for calls to procedures in the collection. We will relax this restriction in the final version of the algorithm and will explain the reason for it at that time. Barth shows that the ALIAS relation, which indicates the possible aliasing relationships among variables, may be computed by the expression $AFFECT^* \circ (AFFECT^*)^T$.

We claim that the following computation results in PVAL specifying correct possible values:

$$PVAL := (AFFECT \vee ((AFFECT^*)^T \circ MODVAL))^+,$$

where R^* denotes the reflexive transitive closure of the relation R, R^T denotes the transpose of R, and R^+ denotes the non-reflexive transitive closure of R. We first note a theorem by Barth [7], which states that a modification of a variable can affect any actual parameters, including the variable itself, which

correspond to the variable, as well as any formal parameters which correspond to any of those actuals. Observe that if the variable is not a formal parameter then the set of corresponding actuals will include only the variable itself. Furthermore, a modification of a variable can affect only these variables. He then goes on to show that AFFECT* gives, for each formal parameter, all of the possible corresponding actuals.

Now suppose that variable X can have possible value A at some point in the execution of the program. There must be a sequence of calls and assignments which resulted in the assignment of A to X. For each call in the sequence which matches formal Y with actual Z, the pair (Y,Z) is in AFFECT and therefore in PVAL. For each assignment of Z to Y in the sequence, the pair (Y,Z) is in MODVAL and therefore in PVAL. Furthermore, if W is aliased to Y, there exists a U such that W AFFECT* U and Y AFFECT* U, as shown by Barth. The pair (U,Z) is therefore in PVAL, since we know that U $(\text{AFFECT*})^{T} \circ \text{MODVAL}$ Z. The pair (W,U) is also in PVAL, since each pair in AFFECT is in PVAL. Since PVAL is closed, the pair (W,Z) must be in PVAL. Therefore, each pair corresponding to the values transmitted by each action in the execution sequence of the program is in PVAL. This implies each pair corresponding to the values transmitted by the sequence as a whole must be in PVAL. In particular, the pair (X,A) is in PVAL. Therefore the information computed is correct. Since the complexity of boolean matrix multiplication is the same as that of transitive closure, the complexity of this algorithm is, like the previous versions of the algorithm, bounded by the time to compute the transitive closure of an $n \times n$ matrix. Furthermore, since this algorithm computes the same information for a single procedure as the previous algorithm, it can be argued that this algorithm is asymptotically the fastest possible algorithm for computing the information which it produces.

This algorithm does not compute completely precise information. The imprecision stems from the use of AFFECT to characterize parameter aliasing, and is discussed by Barth in [7]. As a simple example of the imprecision, consider the skeleton program in figure 1. In this example, A should be assigned Q by the first call on P, and B should be assigned R by the second call on P. However, the method determines

```
procedure P(S,T); ... S:=T; ... ;
procedure Q;
procedure R;
procedure variable A,B;
P(A,Q);
P(B,R);
```

Figure 1

that both A and B have possible values Q and R. This results from the fact that, based on the assignment of T to S, we propagate all values of T to S and then to all actual parameters for S. The basic reason for this is that separate calls on a procedure are not treated separately.

5.3. POINTER VARIABLES

We now introduce pointer variables into the programs being analyzed. We do so in two steps, first considering a single procedure and then allowing multiple procedures with parameters passed by reference. As mentioned earlier, it is necessary to augment the summary information kept for primitives and previously analyzed procedures to give some information about indirect accesses through variables. To simplify the summaries no information will be kept about the number of levels of indirection involved in accessing storage through a pointer, e.g., in accessing an element of a list. Instead, a distinction is made only between a direct access of a variable and an access of storage via some positive number of indirections on a variable.

The summaries must now distinguish between four types of copies. Letting P and Q denote variables and *P and *Q denote storage accessible through the variables, these four types are as follows:

1) P := Q
2) P := *Q
3) *P := Q
4) *P := *Q

Two other types of copies are also allowed. These involve the assignment of the address of a variable to another variable or to storage accessible through another variable. These are as follows:

5) P := addr(Q)
6) *P := addr(Q)

We will utilize the relations AFFECT and MODVAL. Previously these relations dealt with the variables and values which were interesting in terms of propagating values. To handle pointers and the transmission of values via assignments to storage accessible through pointers, we introduce dummy "variables" for each pointer variable. For a pointer variable P this dummy variable is meant to represent the storage accessible through P and will be denoted by *P. To propagate aliasing information correctly based on the assignment of the address of a variable to another variable, we introduce dummy literals for each variable whose address is copied. For a variable Q whose address is copied (cases 5 and 6 above), this literal will be denoted by AQ and represents the address of Q.

1) P := Q
   Add the pair (P,Q) to MODVAL.
   Add the pair (*P,*Q) to AFFECT.

2) P := *Q
   Add the pair (P,*Q) to MODVAL.
   Add the pair (*P,*Q) to AFFECT.

3) *P := Q
   Add the pair (*P,Q) to MODVAL.
   Add the pair (*P,*Q) to AFFECT.

4) *P := *Q
   Add the pair (*P,*Q) to MODVAL.
   Add the pair (*P,*Q) to AFFECT.

5) P := addr(Q)
   Add the pair (P,AQ) to MODVAL.
   Add the pair (*P,Q) to AFFECT.
   Add the pair (*P,*Q) to AFFECT.

6) *P := addr(Q)
   Add the pair (*P,AQ) to MODVAL.
   Add the pair (*P,Q) to AFFECT.
   Add the pair (*P,*Q) to AFFECT.

Figure 2

Figure 2 describes the initialization of AFFECT and MODVAL for each type of copy. If one element of a copy, say X, is not a pointer variable, then all pairs involving *X should be ignored. To give some intuition about the reasons for initializing the relations in this manner, consider the simple assignment P := Q. If P and Q are pointer variables, this has two effects. First, it results in P (and any alias of P) acquiring the value contained in Q. It also causes any storage accessible through Q to be accessible through P, from which it follows that every alias of *Q is an alias of *P.

There are many similarities between the effects which occur due to parameter aliasing and those occuring due to the use of pointers. If there are several assignments to a pointer P, say from Q and R, then *P is aliased to both *Q and *R, but *Q and *R are not necessarily aliased. On the other hand, if P is assigned to several pointers, say Q and R, then *Q and *R are both aliased to *P and, since Q and R may be assigned the same value, *Q and *R are aliased to each other. These two situations are very similar to two situations which can occur with reference parameters, the first being when two different actual parameters are passed to the same formal parameter, and the second being when a single actual is passed to two formals. In fact, parameter aliasing behaves much like pointer aliasing, something which makes more sense when we consider the fact that a call binds the formal parameters to the locations occupied by the corresponding actual parameters for the duration of the call. In effect, for formal X and actual Y, there is an assignment of the form addr(X) := addr(Y). Taking addr(X) to be a variable, so that the storage accessible through it is simply X, we see that the initialization for such a copy is exactly that used in initializing the relations for a call using call-by-reference; i.e., add the pair (X,Y) to AFFECT.

However, the algorithm which we used to propagate values for reference parameters is not sufficiently general to handle pointers. Although the effects are very similar in the two cases, there is one crucial difference. We mentioned that the binding of a formal parameter to an actual parameter is in effect an assignment of the address of the actual to the address of the formal. Considering these two addresses to be variables, this almost models the situation which occurs with pointers. The difference with pointers is that the variables which contain addresses can be aliased as well, and so assignments to a pointer variable must be propagated to all of the aliases of the variable. This includes assignments of the address of a variable (cases 5 and 6). Furthermore, for any variable which is assigned the address of another variable, it is necessary to ensure that the appropriate aliasing is computed between the second variable and the storage accessible through the first variable.

The method which we choose to solve this problem is to iterate. For each modification to a variable which we discover, we will add the aliasing relationships implied by that modification and then iterate to see if this produces any more modifications. This produces the algorithm in figure 3. The function *ind* returns the object which denotes storage accessible via one level of indirection on X. If X is of the form AY, Y is returned. If X is of the form Y and Y is a pointer variable, *Y is returned; if Y is not a pointer variable then the pair should be ignored. Finally, if X is of the form *Y, *Y itself is returned.

For each modification X := Y, this explicitly propagates the implied aliasing information to all Z such that X AFFECT* Z. The propagation to other aliases of X, e.g., to those Z such that Z AFFECT* X, is already done by virtue of the fact that whenever we have Z AFFECT* X we also have

Initialize AFFECT and MODVAL as indicated above.
repeat
   M := $(\text{AFFECT}^*)^T \circ \text{MODVAL}$
   for each (X,Y) **in** M
      Add $(\text{ind}(X),\text{ind}(Y))$ to AFFECT
      Add $(\text{ind}^2(X),\text{ind}^2(Y))$ to AFFECT
until there is no change in AFFECT
PVAL := $(\text{AFFECT} \vee ((\text{AFFECT}^*)^T \circ \text{MODVAL}))^+$

Figure 3

Z AFFECT *X. Adding (*X,*Y) to AFFECT and then recomputing the closure of AFFECT will give *Z AFFECT* *Y, as desired.

The key to demonstrating the correctness of this algorithm lies in the definition of AFFECT. Remember that a pair (X,Y) in AFFECT means that every alias of Y is also an alias of X. Now suppose that variable X has value A at some point in the execution of the program. There must be a sequence of assignments which results in the assignment of A to X. Assume that for the *i*th assignment in the sequence, all of the possible aliasing which can result from previous assignments is embodied in AFFECT. We will show that the same is true for the aliasing which results from the *i*th assignment. The proof of the correctness of the computation of PVAL is then identical to the proof used for reference parameters.

We first note that for each possible assignment which appears in the program, AFFECT is initialized such that if an assignment is the first in the sequence, the aliasing computed from AFFECT is correct after considering that assignment. The *i*th assignment in the sequence, however, could assign a value not just to the explicit target of the assignment, but also to any aliases of that target. Assuming that AFFECT contains at least the aliasing information resulting from the previous *i-1* assignments, and that the target of the *i*th assignment is W, the computation of M finds all possible modifications of those Y such that W AFFECT* Y. The aliasing implied by these modifications is then entered into AFFECT. We must show that forming the closure of AFFECT computes all aliasing which could result from the *i*th assignment. Since W may alias any Z for which there exists a Y such that W AFFECT* Y and Z AFFECT* Y, we must show that the pairs entered in AFFECT by the loop over the pairs in M cause the aliasing for each such Z to be correct. We have shown that this is true for each Y such that W AFFECT* Y. Since, as may be easily verified, *Z AFFECT* *Y is true if Z AFFECT* Y is true, the aliasing which was entered for Y is transferred to Z when the closure of AFFECT is recomputed. This means that the aliasing is correct after considering the *i*th assignment, from which we can deduce that the aliasing is correct after considering the sequence of assignments. Therefore the computation of PVAL is correct, and so A is determined as a possible value for X.

We claim that this algorithm is precise as well as correct, given the assumption that no information about control flow is available. Observe that a pair (X,Y) in AFFECT means that every alias of Y, as computed by the expression AFFECT* $\circ$ (AFFECT*)$^{\top}$, is also an alias of X. Now observe that this is actually the case for every pair which is placed into AFFECT because of a modification. From this it follows that the aliasing is precise, which implies that the computation of PVAL produces precise information.

The reason why the aliasing information computed is precise for pointers but not for reference parameters is that the call structure of the program contains information about the relative lifetimes of the alias relationships for parameters. Unless control flow information is considered, no such information is available for aliasing due to pointers within a single procedure.

Allowing multiple procedures in the collection with parameters passed by reference requires a change only in the initialization of AFFECT. No change in the propagation algorithm itself is required. For each call with operands Y_i to a procedure with formal parameters X_i, the pairs (X_i,Y_i) and $(*X_i,*Y_i)$ should be added to AFFECT. The initialization for all other statements is as above. We omit the details of the proof of correctness for this version of the algorithm. The algorithm has the same imprecision as it did for programs with reference parameters and without pointers.

Before discussing the time requirements of this algorithm, we make an observation about the algorithm itself. This is that it is not necessary to recompute the transitive closure of AFFECT each time through the loop, nor is it necessary to consider the effects on M of a pair in AFFECT whose effects have already been considered. In other words, we can propagate the effects of modifications incrementally. This leads to the equivalent version of the algorithm given in figure 4. In this algorithm, we keep track of all recently discovered aliasing relationships and determine any modifications implied by these relationships. We then compute the aliasing relationships implied by these

Initialize AFFECT and MODVAL as indicated above.
AFFECT := AFFECT^{+}
NEWA := AFFECT
do while NEWA $\neq \phi$
  M := NEWA$^{\top}$ $\circ$ MODVAL
  NEWA := ϕ
  for each (X,Y) **in** M
    Add (ind(X),ind(Y)) to NEWA
    Add (ind^2(X),ind^2(Y)) to NEWA
  Remove those pairs from NEWA that are already in AFFECT.
  Add each pair in NEWA to AFFECT and reform the closure of AFFECT.
  Let NEWA be all those pairs which were added to AFFECT by the previous statement.
end
PVAL := (AFFECT $\vee$ ((AFFECT*)$^{\top}$ $\circ$ MODVAL))$^{+}$

Figure 4

modifications, and continue this process until no new aliasing is discovered.

Let n be the size of the domain of the relations. Let e be the total number of pairs in AFFECT* when the algorithm finishes. The initial closure of AFFECT can be done in time $T(n)$. The computation of the contribution of a single pair in NEWA to M can be done in time n. Every pair in AFFECT appears in NEWA at this point in the program at most once. Therefore the total time spent in the computation of M for all iterations of the outermost loop is at most ne. The loop over the elements of M can be done as M is computed, and so the total time spent in this loop is at most ne. The time spent deleting those pairs in NEWA which are already in AFFECT is proportional to the number of such pairs. There are at most $2ne$ such pairs for all iterations of the outermost loop, since the total number of pairs placed in M for all iterations of the outermost loop is at most ne. Finally, the forming of the closure of AFFECT can be done in time at most n for each pair which is added to AFFECT, whether it is in NEWA or is added in forming the closure after adding a pair in NEWA. There are at most e such pairs, so the total time spent forming the closure of AFFECT for new pairs is at most ne. The computation of PVAL can be done in time $O(T(n))$. The total time for the algorithm as a whole is therefore $O(T(n)+ne)$.

5.4. CALLS ON PROCEDURE VARIABLES

The final step is to consider propagating values through calls on procedure variables. The basic problem with a call on a procedure variable is that at the time the call is encountered in scanning the program, the possible values for the variable, and hence the actual procedures which might be called by the statement, are unknown. Therefore it is not possible to immediately associate the actual parameters of the call with the formal parameters of the procedure being called. To avoid rescanning the program several times, we need a mechanism to keep track of the actual parameters of calls on procedures variables. When a value is determined for a procedure variable, we can then associate the actual parameters of the calls on the variable with the formal parameters of the value.

The mechanism which we choose to accomplish this is to create, for each procedure variable, dummy formal parameters. For a given procedure variable X which is called with m actual parameters, we create m dummy formal parameters XF_i, for $1 \le i \le m$. We also create dummy variables $*XF_i$ for each dummy formal parameter, representing the storage accessible through the dummy formal. The number of dummy formal parameters which need to be created can be determined by an initial scan of the program which keeps track of the number of actual parameters passed to each procedure variable. If the source language requires complete type specifications of procedure variables, i.e., that the types of the parameters be specified as well, then the number of dummy formal parameters which are needed for each procedure variable can be determined from the declaration of the variable. Also, entries $*XF_i$ only need to be created for those parameter positions which have pointer types. Having created dummy formal parameters for each procedure variable, the initialization required for a call on a procedure variable is exactly that for a call on a procedure in the collection. If the call has actual parameters Y_i and is to procedure variable X with dummy formal parameters XF_i, the pairs (XF_i,Y_i) and $(*XF_i,*Y_i)$ should be added to AFFECT.

If we consider a procedure variable X to be a procedure with formal parameters XF_i which contains a single statement, that statement being a call on the current value of X with actual parameters XF_i, it should be clear that each time a value A is determined for X we should associate the formal parameters of A with the dummy formal parameters of X as formal-actual pairs. One way in which this can be done, as suggested by Kenneth Walter [13], is to create relations $FPARM_i$, one for each parameter position. A pair (X,Y) in $FPARM_i$ means that X has ith formal parameter Y. For each procedure A in the collection with formal parameters Y_i, the pair (A,Y_i) is placed in $FPARM_i$ for each parameter position i. For each procedure variable X with dummy formal parameters XF_i, the pair (X,XF_i) is placed in $FPARM_i$ for each parameter position i. Now suppose that A is determined as a possible value for X. If Y_i is the ith formal parameter of A, and A is a possible value for X, and X has ith formal parameter XF_i, then the pairs (Y_i,XF_i) and $(*Y_i,*XF_i)$ should be added to AFFECT. The expression $FPARM_i^T \circ PVAL^T \circ FPARM_i$ computes the pair (Y_i,XF_i). This leads to the algorithm in figure 5. This algorithm, like the one developed for pointers,

```
Initialize AFFECT and MODVAL as indicated above.
repeat
    M := (AFFECT*)^T ∘ MODVAL
    for each (X,Y) in M
        Add (ind(X),ind(Y)) to AFFECT
        Add (ind^2(X),ind^2(Y)) to AFFECT
    PVAL := (AFFECT ∨ ((AFFECT*)^T ∘ MODVAL))^+
    for each parameter position i
        P := FPARM_i^T ∘ PVAL^T ∘ FPARM_i
        for each (X,Y) in P
            Add (X,Y) to AFFECT
            Add (ind(X),ind(Y)) to AFFECT
until there is no change in AFFECT
PVAL := (AFFECT ∨ ((AFFECT*)^T ∘ MODVAL))^+
```

Figure 5

can be transformed into an equivalent algorithm which propagates information incrementally. A similar time bound can also be derived for it. Its correctness should be fairly clear given the correctness of the algorithm for programs without calls on procedure variables, and we omit the proof.

We mentioned earlier that actual parameters to calls on procedures in the collection or on procedure variables should be restricted to be variables and not constants. The reason for this was to avoid unnecessarily complicating the discussion of aliasing, since constants are passed by value under call-by-reference. The solution to this is to initialize PVAL with all pairs (X,A) such that there is a call to a procedure (or procedure variable) with formal parameter X and corresponding actual parameter A. No entry is made in AFFECT for such pairs. The computation of

$$\text{PVAL} := (\text{AFFECT} \vee ((\text{AFFECT}^*)^T \circ \text{MODVAL}))^+$$

is then changed to assign

$$(\text{PVAL} \vee \text{AFFECT} \vee ((\text{AFFECT}^*)^T \circ \text{MODVAL}))^+$$

to PVAL. In this way constant actual parameters are propagated but no values may be attributed to them due to modification of the corresponding formals.

6. THE ALIAS RELATION

The ALIAS relation, as mentioned earlier, can be computed by the expression

$$(\text{AFFECT}^*) \circ (\text{AFFECT}^*)^T \text{ [7].}$$

This relation gives an answer to the question "Is it possible at some point in the program for variable A to be aliased with variable B?" The obvious ways to store this relation, e.g., as a boolean matrix, or as a list for each variable of the variables to which it might be aliased, take space which is roughly proportional to the square of the number of variables. In many situations, however, it is the case that there are sets of variables which are equivalent under this relation. We define equivalence of two variables to mean that they may be aliased to each other and that the sets of variables to which they may be aliased are identical. Each such class could potentially be stored in space linearly proportional to the number of variables in it, rather than to the square of that number. The amount of storage required for the ALIAS relation is then c^2 rather than v^2, where c is the number of classes (which may be of unit size) and v is the number of variables. This is especially useful in ECS because of the large number of temporaries which are generated for constructs such as array indexing, and which fall into fairly large classes of equivalent variables.

We prove the following theorem, which gives a necessary and sufficient condition for two variables to be equivalent as defined above.

Theorem: Given the relation AFFECT, consider it as a graph and find its maximal strongly connected components. Replace each such component with a new node identified with the component. This leaves a directed acyclic graph (DAG). Define a *sink* in the DAG to be a node which has no edges coming out of it. A node X in the original graph is a sink if the node identified with the strongly connected component containing X is a sink in the DAG. We say that node A reaches node B if there is a path, possibly of length zero, in the graph from A to B. Two nodes in the original graph are equivalent if and only if they reach the same set of sinks [8].

Proof: We will consider AFFECT and ALIAS as graphs, derived in the obvious way from the relations previously discussed, and will give the proof in terms of nodes and edges of these graphs. AFFECT is a directed graph, while ALIAS may be considered as an undirected graph, since the ALIAS relation is symmetric. This is easily seen from the definition of ALIAS. When we speak of an edge in ALIAS, we henceforth mean an undirected edge. Also, when we speak of a node X reaching a node Y, we mean that there is a path from X to Y in AFFECT, unless stated otherwise. A path is defined as a possibly empty sequence of edges.

We note that there is an edge between node X and node Y in ALIAS if and only if there exists a node Z such that X reaches Z and Y reaches Z. This follows immediately from the definition of ALIAS in terms of AFFECT. Two nodes X and Y are equivalent if and only if the following three conditions hold: there is an edge between X and Y in ALIAS; for each edge between X and some node Z in ALIAS there is an edge between Y and Z; for each edge between Y and some node Z in ALIAS there is an edge between X and Z. In other words, two nodes are equivalent if and only if they alias each other and the sets of nodes which they alias are identical.

Lemma: X ALIAS Y is true if and only if there exists a sink Z such that X reaches Z and Y reaches Z.

Proof of Lemma: From the definition of ALIAS, it is clear that X ALIAS Y is true if and only if there exists a node W such that X reaches W and Y reaches W. Therefore, if there exists a sink Z such that X reaches Z and Y reaches Z, it follows that X ALIAS Y is true. We now show that such a Z exists if X ALIAS Y is true. Let W be such that X reaches W and Y reaches W. Consider the DAG derived from AFFECT in the statement of the theorem. Let U be the node in the DAG which is

identified with the strongly connected component containing W. U must reach some sink V in the DAG. Let Z be a node in the strongly connected component identified with V. Since U reaches V in the DAG, it follows that W reaches Z. This means that X reaches Z and Y reaches Z. V is a sink, implying that Z is a sink, and so Z is the desired node.

We now prove the theorem, first showing that if X and Y reach the same set of sinks they are equivalent, and then showing that if they reach different sets of sinks, they are not equivalent.

Suppose that X and Y reach the same set of sinks. Since this set is necessarily non-empty, they are aliased to each other. Suppose that X ALIAS W is true. Let Z be a sink such that X reaches Z and W reaches Z, as in the lemma. Since X reaches Z, it follows by hypothesis that Y also reaches Z. By the lemma, it follows that Y ALIAS W must be true. Similarly, if Y ALIAS W is true it follows that X ALIAS W is true. Therefore X and Y are equivalent.

Now suppose that X and Y are equivalent. Furthermore, suppose that there exists a sink Z which one of them, say X, reaches, and which the other one, say Y, does not reach. By the lemma, it follows that X ALIAS Z is true, since a sink reaches itself. Furthermore, there is no W such that Y reaches W and Z reaches W, since Z reaches only itself, being a sink, and Y does not reach Z. Therefore Y ALIAS Z is not true. This gives a contradiction, since we have found a node Z such that X ALIAS Z is true and Y ALIAS Z is not true, implying that X and Y are not equivalent. This completes the proof of the theorem.

This theorem leads naturally to a reasonably efficient method for computing the sets of equivalent variables. These sets can then be used for storing the ALIAS relation. Strongly connected components can be computed in time O(max(n,e)), where n is the number of nodes in AFFECT and e is the number of edges [1]. Deciding which nodes are sinks can be done by first forming the reflexive transitive closure of AFFECT, and then checking each strongly connected component to see if there is an edge from any node in the component to a node in another component. If there is no such edge then the component, and each node in it, is a sink.

A technique described by Wegman and Carter [14] can then be used to partition the nodes into classes based on the sets of sinks which they reach. This technique involves hashing the sinks which a given node reaches, exclusive-oring the results of the hash together to get a new representation of the set. Having found the new representation of the set of sinks reached by each node, the nodes can be partitioned very quickly based on the equality of these representations by using a hash table. For those nodes whose sets of sinks have the same such representation, the actual sets should be compared. This is because the representation is guaranteed to be unique only within a specified probability [14].

7. SUMMARY

In this paper we have presented an approach to interprocedural data flow analysis for programs which use pointers, label variables, and procedure variables. We have stated as the major obstacle to such analysis the problems of determining the call graph, the control flow graph, and the alias relationships in the program, and have presented an algorithm for determining these program characteristics. Subject to the basic assumption that information about control flow is not available, we have shown that the algorithm is precise for programs containing simple assignments and multiple procedures, with parameters passed by value. Assuming that information about the number of levels of indirection involved in accessing storage through a pointer is not available, we have shown that the algorithm is precise for programs containing pointer variables, as long as the program consists of a single procedure. The algorithm is in fact precise for programs containing multiple procedures and pointer variables as long as pointers are not passed as parameters. When pointers may be passed as parameters, or parameters are passed by reference, the information produced by the algorithm is no longer as precise as possible. Similarly, when the program may contain calls to procedure variables the information produced lacks some precision.

We have also shown that the problem of determining possible values for procedure variables is P-space hard. This fact makes it unlikely that a method exists which is both precise and reasonably efficient. In certain cases we have shown that the algorithm is asymptotically as efficient as possible. We have also discussed some characteristics of the aliasing information which is produced, and have shown how these characteristics can be used to store the information more efficiently.

It is still an open question whether it is possible to produce better information without any loss of efficiency. It is also open as to whether iteration is required in cases involving aliasing and calls on procedure variables. It may be that the information produced can be computed in time bounded by the time to compute the transitive closure of an $n \times n$ matrix. It is also possible that certain restrictions on the use of procedure variables and reference parameters may make it possible to compute the desired information both precisely and efficiently.

8. ACKNOWLEDGEMENTS

The author would like to thank Bill Harrison for suggesting this work. He, Louise Trevillyan, and Larry Carter deserve thanks for the many helpful suggestions which they made as these algorithms were being developed. Steve Ward also provided many suggestions on the work. Jeanne Ferrante, Janet Fabri, Fran Allen, John Guttag, and the others mentioned above also deserve thanks for their comments on previous drafts of this manuscript which greatly improved the presentation. Mark Wegman provided some key insights in analyzing the time requirements of the algorithm. Finally, IBM itself should be thanked for providing the opportunity to do this work through the MIT VI-A cooperative program.

9. REFERENCES

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

2. Aho, A.V. and Ullman, J.D. *Principles of Compiler Design*, Addison-Wesley, 1977.

3. Allen, F.E. Interprocedural Data Flow Analysis. *Proceedings IFIP Congress 74*, North Holland Publishing Company, Amsterdam, 398-402.

4. Allen, F.E. and Cocke, J. A Program Data Flow Analysis Procedure. *CACM 19*, 3 (March 1976), 137-147.

5. Allen, F.E., et. al. The Experimental Compiling Systems Project. *IBM Research Report* RC6718, T.J. Watson Research Center, Yorktown Heights, N.Y. September 1977.

6. Banning, J.P. A Method for Determining the Side Effects of Procedure Calls. *Ph.D. Thesis, Stanford University*. Report No. 213, Stanford Linear Accelerator Center (August 1978).

7. Barth, J. Interprocedural Data Flow Analysis Based on Transitive Closure. Univ. of California at Berkeley, Computer Science Dept., Tech. Rep. UCB-CS-76-44, September 1976.

8. Carter, J.L. Private Communication.

9. Graham, S.L. and Wegman, M. A Fast and Usually Linear Algorithm for Global Flow Analysis. *JACM 23*, 1 (January 1976), 172-202.

10. Rosen, B.K. Data Flow Analysis for Procedural Languages. *JACM 26*, 2 (April 1979), 322-344.

11. Ryder, B.G. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering SE-5*, 3 (May 1979), 216-226.

12. Spillman, T.C. Exposing Side-Effects in a PL/I Optimizing Compiler. *Proceedings IFIP Conference 1971*, North Holland Publishing Company, Amsterdam, 376-381.

13. Walter, K.G. Recursion Analysis for Compiler Optimization. *CACM 19*, 9 (September 1976), 514-516.

14. Wegman, M.N., and Carter, J.L. New Classes and Applications of Hash Functions. *Proceedings 20th Annual Symposium on Foundations of Computer Science* (October 1979), 175-182.

15. Weihl, W.E. Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables. *S.M. Thesis, Massachusetts Institute of Technology* (to be published).

16. Winklmann, K.A. A Theoretical Study of Some Aspects of Parameter Passing in ALGOL60 and in Similar Programming Languages. *Ph.D. Thesis, Purdue University* (August 1977).