Copyright © 1976, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

INTERPROCEDURAL DATA FLOW ANALYSIS
BASED ON TRANSITIVE CLOSURE

by

Jeffrey M. Barth

Memorandum No. ERL-M600

13 September 1976

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract:

A new interprocedural data flow analysis algorithm is presented and analyzed. The algorithm associates with each procedure in a program information about which variables may be modified, which may be used, and which are possibly preserved by a call on the procedure, and all of it subcalls. The algorithm is sufficiently powerful to be used on recursive programs and to deal with the sharing of variables which arises through reference parameters. The algorithm is unique in that it is strictly single pass, not requiring a prepass to compute calling relationships or sharing patterns. A lower bound for the computational complexity of gathering interprocedural data flow information is derived and the algorithm is shown to be asymptotically optimal. An implementation is described which utilizes bit vector operations for the computation of the data flow information, and indicates various space saving tricks which are possible because of characteristics of the algorithm.

Research sponsored by the National Science Foundation Grant MCS74-07644-A02.

Introduction:

A great deal of recent research has been devoted to developing algorithms for intraprocedural global flow analysis. [6,9,15,16,19] There is a limit to the usefulness of these methods in the absence of interprocedural data flow information since, in practice, subroutine calls are interspersed among simple program statements. Consider this example code sequence:

```
x := (y/z) + w;
u := SOMEFUNCTION(x);
v := y/z;
```

Whether y/z can be considered a common subexpression depends on the information that SOMEFUNCTION can not modify the values of y and z.

Interprocedural information that is used at the point of call of a subroutine has been called "summary" data flow information. [11] With each function call, a summary of the variables that may be modified, that may be read, and that are possibly preserved will be useful for intraprocedural analysis. [14] Precise definitions for these three data flow problems will appear in the section on notation and problem definition which follows.

There are three fundamental difficulties associated with gathering summary information. In order to summarize a procedure, P, its body is examined. If P calls another procedure, Q, then the flow analysis of P requires a summary of Q. In the case of mutually recursive procedures, there is no order for analyzing procedure bodies that avoids this difficulty.

The second fundamental problem that an interprocedural data flow analysis algorithm must cope with arises in programming languages which allow name or reference parameters. [10,12] Since these mechanisms introduce storage sharing among different variables, called aliases, the determination of which variables may be modified by a statement is nontrivial. Suppose that r is a reference parameter in a procedure P which contains

```
r := u + v;
```

as a statement. The summary information for P must reflect the fact that any variable potentially shared with r may be modified.

Another difficulty associated with gathering summary information for recursive programs is in the correct treatment of variables. It is understood that separately declared variables are distinct even if their spellings coincide, but there is a more intrinsic problem. Recursive invocations of a procedure, P, have separate copies, called incarnations, of P's local variables. High quality summary data flow information for a procedure, Q, will not include the possibility that Q may modify a variable local to P if the incarnation which may be modified is not the same as the incarnation addressable at the point of call.

The algorithm to be presented in this paper is sufficiently powerful to collect summary data flow information for recursive programs. It can be used on programs with reference parameters and will deal with different incarnations of variables local to recursive procedures.

The data flow analysis technique described is strictly one pass in nature and can be implemented in the first (parsing) pass of a compiler. An implementation is described that utilizes bit vector operations, which are available as word operations on most hardware. The running time of the algorithm is approximately the same as the running time for transitive closure, which is shown to be a lower bound for the problem under reasonable assumptions. The algorithm simplifies in a natural way for use in languages with naming structures less general than Algol or for use on programs that do not use recursion.

Before embarking on any further study of interprocedural data flow analysis, it must be made clear at what stage of an optimizing compiler this information is gathered. Interprocedural data flow information is used to determine that particular optimizing transformations do not affect the correctness of the program. Thus, it is necessary to first compute the data flow information and then to apply specific optimizations. In the case of recursive programs, an entire pass over the program may be necessary to expose the data flow information. For these reasons, an optimizing compiler should first compute summary data flow information in a pass, and then attempt to find optimizing transformations in subsequent passes.

Summary of previous work:

The oldest interprocedural data flow analysis technique of which the author is aware was developed by Spillman. [18] His method is primarily designed to determine side effects of assigning to a variable (due to aliasing), but can be used to determine the side effects of calling a procedure. He codifies information in a boolean matrix and then iteratively merges rows to account for various ways in which side effects are propagated in running programs. In particular, for procedures calls, he merges rows which represent effects due to procedures iteratively until the bits of the matrix stabilize. His technique does not distinguish the incarnations of variables nor does it generalize to data flow problems other than "modifies".

Allen developed the most widely used algorithm for interprocedural data flow analysis. [2] Originally, the algorithm was limited to nonrecursive programs. A first pass over the program is used to discover the calling relationships of the procedures. Interprocedural information is then gathered by processing procedures in the "reverse invocation order". This ordering has the property that all procedures called by some procedure, P, are analyzed before P. The existence of some ordering with this property is guaranteed by the nonrecursive character of the program.

Allen and Schwartz have extended this basic algorithm to handle recursive programs. [3] The essential idea is that a procedure, P, which calls another procedure, Q, can be analyzed before Q if one is willing to make worst case assumptions about the data flow impact of Q. The information can be refined by

reanalyzing P after a better approximation for Q has been computed.

Allen does not discuss the problem of different incarnations of variables in recursive programs. Her techniques do not naturally handle sharing of variables because the order in which procedures are examined is an order in which bindings at calling sites are always discovered after the called procedure has been analyzed. The prepass which gathers the calling information can be used to collect sharing information [18], which allows Allen's method to base its actions on the most general sharing pattern for each given variable.

Other major work in interprocedural data flow analysis has been done by Rosen. [14,17] His method works on recursive programs and produces very precise information even in cases in which the sharing patterns of variables vary depending on the context. Unfortunately, his method is probably impractically slow and complicated. The basic idea exploited by Rosen is that when analyzing a procedure, P, with a call on a subprocedure, Q, the information impact of Q can be expressed in a formula which summarizes P. That is, the summary of P is an equation with unknowns for subprocedures called in P. Analyzing a program yields a set of equations which can be solved simultaneously to obtain useful data flow information.

Sharing is handled by allowing formulas to be parametric, in some sense, in a sharing pattern. When an equation is used to summarize information at a point of call, the particular bindings at that point of call are used to refine the formula.

Incarnations of variables can be kept apart by renaming variables in equations as they are passed back into the procedure which created them, but Rosen never explicitly states how this is to be done.

The technique used by Rosen to solve the simultaneous equations is a bit reduction strategy, in which maximal information is at first assumed. The equations are substituted into each other until the information, which is monotonically decreasing, stabilizes. He proves that the information thus obtained is correct. A characteristic of this method is that the iteration must be fully carried out, because partial solutions are incorrect until complete stabilization. Rosen proposes that the equations may be simplified in advance by using various symbolic execution strategies on the program.

Hecht and Shaffer are in the process of implementing an interprocedural analyzer for a language called SIMPL-T. [8] They are employing a multipass scheme which uses transitive closure. Because of the limited scoping structure of SIMPL-T, the problem of variable incarnations does not enter into their analysis. Sharing is handled by a technique which is based on preanalyzing sharing configurations. Hecht and Shaffer only plan to calculate information about which global variables may be modified, which greatly simplifies the problem that they are attacking. Their work appears to be the most similar to the algorithm described in this paper, but when the details of the new algorithm have been presented, it will be possible to see that the fundamental strategies for collecting summary data flow information are totally

different. Their techniques are successful largely as a result of the simplifications that were possible because of the limited design goals of the SIMPL-T project.

Notation and Problem Definition:

It will be convenient to introduce some notational conventions which will be used throughout this paper. A formal definition of the summary data flow information that we wish to compute will then be presented.

We are considering the summary data flow problem for programs in Algol like (static naming structure) languages. A <u>program</u> is understood be reasonably well formed, i.e. there is no inaccessible code. Subroutines are called explicitly at <u>calling sites</u> which are textually visible, rather than being activated by the existence of some condition. A <u>variable</u> is a declared entity whose storage may be associated with subroutine entry. That is, if a procedure is called recursively, a particular variable may be associated with several storage locations. Each such location is called an <u>incarnation</u> of the variable.

Def: Let <u>PP</u> be the set of procedures in a program to be analyzed. For the examples, assume that P, Q, R, and S are members of PP.

Def: Let <u>VV</u> be the set of variables in the program. For the examples, assume that r, u, v, w, x, y, and z are members of VV. Separately declared variables are distinct even if their spellings coincide.

Def: A (binary) <u>relation</u> is a set of ordered pairs. In accordance with standard notation, for relations A and B:

- A* is the reflexive transitive closure of A,
- A+ is the transitive closure of A,
- A B is the composition of A and B,
- ~A is the set of pairs not in A,
- A and B is the set of pairs in both A and B,
- A or B is the set of pairs in either A or B,
- TRANS(A) is the transpose of A, that is
  (a,a') ∈ A iff (a',a) ∈ TRANS(A), and
- (a,a') ∈ A is written interchangeably with a A a'
  or (a,a') in A.

Def: Let <u>CALL</u> be a relation defined on PP x PP. A pair (P,Q) is in CALL if procedure P contains a call on procedure Q.

Def: Let <u>MUSTCALL</u> be a relation defined on PP x PP. A pair (P,Q) is in MUSTCALL if procedure P contains a call on procedure Q on all paths of execution for which P terminates normally. That is, a call on P must always be followed by a call on Q before P returns to its calling site.

The notion of "must" in contrast to "may" which distinguishes MUSTCALL from CALL is sufficiently important to warrant careful exposition. A relation which contains "may" information only expresses the possibility that the action suggested by the relation name will occur in the executing program. For example, (P,Q) ∈ CALL states the possibility that P may call Q. In contrast to this, "must" information is appropriate when a certainty about the executing program is intended. If (P,Q) ∈ MUSTCALL then the call on Q must be executed as a subcall of P (in the case of normal termination of P for which summary data flow information is meaningful).

May information is usually used in the negative sense. The fact that (P,Q) ∈ ~CALL constitutes definite information that P will not directly call Q. The fact that (P,Q) ∈ CALL does not imply that P really will call Q, and so is of limited use for extracting important information.

Def: Let <u>DIRECTMOD</u> be a relation defined on PP x VV. A pair (P,x) is in DIRECTMOD if procedure P contains a statement which modifies the value of variable x. (may information)

Def: Let <u>DIRECTUSE</u> be a relation defined on PP x VV. A pair (P,x) is in DIRECTUSE if procedure P contains a use of the value of variable x. (may information)

Def: Let <u>DIRECTNOTPRE</u> be a relation defined on PP x VV. A pair (P,x) is in DIRECTNOTPRE if procedure P does not preserve the value of variable x on any path of execution for which P terminates normally. That is, x must be set by an invocation of P. (must information)

All of the above relations are direct in the sense that they contain information about effects of procedures ignoring indirect effects due to subcalls. The remaining relations to be defined contain summary information about the effects of procedures and associated subcalls.

Def: Let <u>MOD</u>, <u>USE</u>, and <u>PRE</u> be relations defined on PP x VV which are the summary information that we wish to calculate:

If (P,x) ∈ ~MOD then procedure P, and any subcalls of P, will not modify the value of x.

If (P,x) ∈ ~USE then procedure P, and any subcalls of P, will not use the value of x.

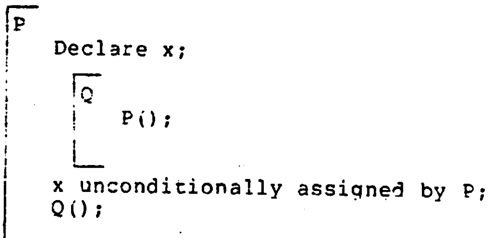
If (P,x) ∈ ~PRE then before P returns, the variable x will have been assigned.

MOD, USE, and PRE are relations which contain may information. It will be more convenient to calculate preserves information as must information, so we define an additional summary data flow relation:

Def: Let <u>NOTPRE</u> be a relation defined on PP x VV. A pair (P,x) ∈ NOTPRE implies that before P returns, the variable x will have been assigned. (NOTPRE is simply ~PRE.)

It is understood that summary information must be correct with respect to any particular point of call of a procedure. The summary data flow relations are defined on PP x VV, where VV is the set of addressable incarnations of variables at the point of

call. The following example illustrates why this interpretation of summary information is critical:



```
P
   Declare x;
   Q
      P();

   x unconditionally assigned by P;
   Q();
```

It is wrong to conclude that P NOTPRE x at its call within Q, since the incarnation of x which is addressable at this site is different from the incarnation assigned by the call on P.

The relations defined above, and several to be introduced in later sections, are summarized in Table 1.

Interprocedural data flow analysis is essentially the process of computing the summary data flow relations. We must be somewhat careful in specifying what is expected of an interprocedural data flow analysis technique because the definitions of the relations do not exclude trivial (and useless) solutions. Several evaluation criteria of data flow analysis techniques will be necessary.

Def: We say that a summary data flow relation, A, is <u>correct at a particular calling site</u> for a procedure, P, if:

For may information - There is no instance of the following three conditions occurring simultaneously for any variable, x:

i. x is addressable at the calling site
ii. (P,x) ∈ ~A
iii. The call on P may have the summary effect A on the incarnation of x addressable at the time of call.

For must information - There is no in-

Table 1

Relation	Domain	Direct/Summary/ Program Independent	May/Must
CALL	PP x PP	Direct	May
MUSTCALL	PP x PP	Direct	Must
DIRECTMOD	PP x VV	Direct	May
DIRECTUSE	PP x VV	Direct	May
DIRECTNOTPRE	PP x VV	Direct	Must
MOD	PP x VV	Summary	May
USE	PP x VV	Summary	May
PRE	PP x VV	Summary	May
NOTPRE	PP x VV	Summary	Must
SCOPE	PP x VV	Program Independent	
GENSCOPE	PP x VV	Program Independent	
GLOBAL	PP x VV	Program Independent	
NONRECURSIVE	PP x VV	Program Independent	
UPCALL	PP x PP	Program Independent	
AFFECT	VV x VV	Direct	May

stance of the following three conditions occurring simultaneously for any variable, x:

i. x is addressable at the calling site
ii. $(P,x) \in A$
iii. the call on P may fail to have the summary effect A on the incarnation of x which was addressable at the time of the call.

Def: We say that a summary data flow relation, A, is correct if it is correct at every calling site in the program.

We wish to define the precision of a calculated relation to capture the concept of the amount of definite information available. In particular, for may information, the sparser the relation, the more effects are known not to be possible. Relations have a natural partial ordering by set inclusion. We define precision to be a meaningful comparator between related elements in the partial order.

Def: We say that a correct summary data flow relation is more precise than another correctly calculated version of that relation if:

For may information - the more precise relation is a subset of the pairs of the less precise relation.

For must information - The more precise relation is a superset of the pairs of the less precise relation.

The notion of correct does not exclude the trivial solutions of all pairs for may information and no pairs for must information. These solutions are correct, but are usually too imprecise to be useful. It is too stiff a criterion to ask for the most precise solution for a summary data flow relation, since the determination of this will be undecidable in general.

Def: We say that a summary relation is precise up to symbolic execution if

For may information - it is the most pre-

> cise information possible assuming that all conditionally executed code is executable and that all the variables in the program are spelled distinctly.
>
> For must information - it is the most precise information possible assuming that any path of execution through procedures is possible and that all variables in the program are spelled distinctly.

The condition pertaining to pairwise distinct variable spellings removes a degenerate case in several proofs, and will be explained in the first such instance.

The part of the definition pertaining to conditional execution is somewhat different for may and must relations. Clearly, if all paths through a procedure are executable (the must condition), then all conditionally executed code is executable (the may condition). The converse is not true. Consider this procedure:

```
P
   Declare w;
   Comment: w is local so that the assignments can't
            affect its value;

   w := someexpression;
   IF w = 0 THEN x := u+1 ELSE y := v+2;
   IF w = 0 THEN v := u+3 ELSE x := v+4;
```

Even assuming that all conditionally executed code is executable, it would still be possible to conclude that neither x nor y is preserved by P (by symbolically merging the THEN and ELSE parts of the conditional statements). We wish, however, to consider must information precise up to symbolic execution without requiring this kind of analysis.

Ultimately what is required of an interprocedural data flow analysis algorithm is that it produce provable correct relations which are empirically sufficiently precise. When a technique is precise up to symbolic execution we may be confident that the information produced is of very high quality. Specific results of testing the techniques developed here will be included in [5].

Properties of MOD, USE, and PRE:

The algorithm which will be presented computes MOD and USE as may information, but will compute NOTPRE as must information. To obtain PRE, the complement of NOTPRE is calculated. This section will partially justify the use of a different technique for computing PRE. There is a means of computing PRE directly using a variation of the algorithm presented here. An account of this appears in [5].

Although MOD and PRE are both may information, the manner in which they are collected intraprocedurally differs. Consider straightline code:

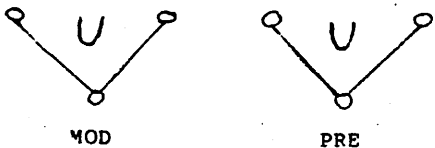
```
x := u + 1;
y := v + 1;
```

The first statement modifies x and the second statement modifies y. The MOD information for the two statements together is the union of the information for each statement: they modify both x and y. The first statement may preserve all variables except x. The second may preserve all variables except y. The PRE information for the two statements together is the intersection of the preserves information for each statement: they may preserve all variables except x and y. In pictures, this is summarized:

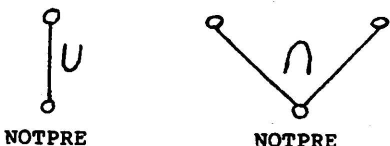
Now consider code which is conditionally executed:

```
IF booleanexpression THEN x := u + 1 ELSE y := v + 1;
```


In this case the union of the may information for the internal statements is the information for the entire statement for both MOD and PRE. In pictures this is summarized:



For purposes of this paper, it must suffice to say that the algorithm presented has a hidden assumption that the information composition function for straightline code is union. NOTPRE satisfies this requirement, with the disadvantage of being must information. All forms of must information require intersection among conditionally executed statements:



Calculating MOD and USE, no sharing:

For simplicity in this section, we assume that there is no mechanism, such as call by reference parameters, for introducing sharing among variables. Formulas which use CALL, DIRECTMOD, and DIRECTUSE to calculate MOD and USE are presented. A series of formulas will be presented which calculate summary data flow relations to differing levels of precision. In order to distinguish the computed relations, we will write, for example, MOD/1.1 to be the MOD relation as calculated by formula 1.1. Only the formulas for MOD are justified since the arguments in both cases are essentially the same.

Correct formulas for MOD and USE are easy to obtain:

MOD := CALL* DIRECTMOD (1.1)
USE := CALL* DIRECTUSE (1.2)

Claim: MOD/1.1 is correct.

> Justification: Since MOD is may information, we must justify the absence of any pair, (P,x), missing from the computed relation. Suppose that P ~ (CALL* DIRECTMOD) x. This says that P, and all procedures callable from P (directly or indirectly), do not contain a statement which modifies x. From this we conclude that P, in summary, does not modify x.

Although the above formulas are correct, they are extremely conservative in their treatment of variables. If any incarnation of x may be modified by a subcall of P, then P MOD/1.1 x is calculated. If the incarnation of x which is modified by the subcall must be different from the one addressable at the point of call of P, then the pair (P,x) could have been eliminated from MOD, increasing the precision of the information.

Refined formulas will be obtained for MOD and USE by including Algol scoping rules in the calculations. The techniques presented can be modified to accommodate languages with less general static naming structure.

> Def: The <u>level of a procedure</u> is the static depth at which the procedure is defined. Declarations which occur anywhere within a procedure will be associated with the procedure invocation, rather than with BEGIN block entry. The <u>level of a variable</u> is the same as the level of the procedure to which it is local. (This differs somewhat from standard Algol terminology.) Global variables are at level 0, the lowest naming level.

When comparing levels, it will usually be clearer to refer to lower levels as <u>outer</u> and higher levels as <u>inner</u>.

> Def: Let <u>SCOPE</u> be a relation defined on PP x VV. A pair (P,x) is in SCOPE iff the level of x is strictly lower than the level of P. That is, x is declared at a outer level from P.

SCOPE is weaker than an addressability relation, since it is possible that P SCOPE x, even though x is not addressable within P. It is shown in [5] that using SCOPE rather than an addressability relation does not degrade the calculated summary data flow information.

The following lemma will be used repeatedly in justifying formulas involving scoping rules:

<u>Scoping Lemma</u>: When computing summary data flow information, a call (direct or indirect) on a level n procedure, P, and all of its subcalls, can affect addressable variables at the original point of call only at levels 0 thru n-1.

> Proof: P, and its subcalls, may be able to address variables at levels which exceed n-1, but the variables at these levels will be new incarnations and

are not addressable at the original calling site. Under the rules of static scoping, when a procedure is called, the variables which are addressable in the called procedure are a subset (not necessarily proper) of those addressable at the calling site, plus new incarnations of local variables. In the body of P, the levels 0 thru n-1 contain variables addressable at the original calling site, and level n contains new incarnations of local variables for P. By applying the previous observation inductively, subcalls of P (direct or indirect) can only affect a subset of the variables addressable in P plus new incarnations of local variables which were not addressable at the original calling site.

Note that the above lemma is false in the presence of reference parameters.

Formulas 2.1 and 2.2 produce summary data flow information which is more refined than that produced by formulas 1.1 and 1.2:

```
MOD := CALL* (DIRECTMOD and SCOPE)                (2.1)
USE := CALL* (DIRECTUSE and SCOPE)                (2.2)
```

Claim: MOD/2.1 is correct.

Justification: If P ~MOD/1.1 x then by the correctness of formula 1.1 it follows that the absence of (P,x) from MOD/2.1 will not make the relation incorrect. Suppose that P MOD/1.1 x but that P ~MOD/2.1 x. Since P MOD/1.1 x, it must be true that P CALL* Q and Q DIRECTMOD x (for at least one procedure Q). Since P ~MOD/2.1 x it must be true that for any such Q, Q ~SCOPE x. That is, the level of x >= level of Q. (In short, x is local to Q since the only variables addressable from Q at a level which exceeds or equals Q's are locals.) By the scoping lemma, the incarnation of x which Q modifies can not be addressable at the calling site of P. Thus, all pairs which are eliminated by using formula 2.1 over 1.1 were superfluous. (Note: The fact that MOD/2.1 is a subset of MOD/1.1 follows immediately from the semantics of "and". It is this fact which implies that formula 2.1 is at least as precise as formula 1.1)

Formulas 2.1 and 2.2 reflect the observation that actions on local variables never affect summary data flow information. These equations are sufficiently powerful to process programs in

languages which do not allow the nesting of naming levels except to distinguish locals and globals. For such languages the relations computed are precise up to symbolic execution, a fact that will follow as a corollary to a more general statement which is proven later in this section. SIMPL-T, C, and BLISS are languages which enforce this naming limitation. [8,13,20]

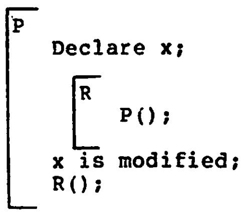
A series of examples which illustrate program skeletons will be useful in developing intuition as to the precision of various formulas used in calculating summary data flow relations. We begin with two examples for which formulas 2.1 and 2.2 are more precise than formulas 1.1 and 1.2. For all the examples, the reader is to assume that subroutine calls are executed conditionally so that nonterminating recursion is avoided.

```
P
   Declare x;

   x is modified;
   P();
```

Example 1

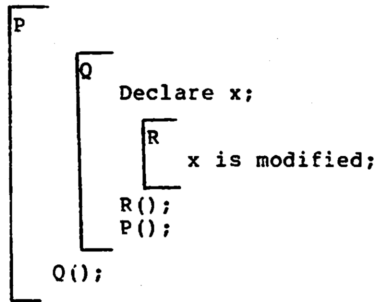
The call on P does not modify the addressable incarnation of x at the point of call. This is the important case of direct recursion.



Example 2

Here too, since x is local to P, P ~(DIRECTMOD and SCOPE) x. This example will figure in a later discussion.

Formulas 2.1 and 2.2 fail to produce data flow information which is precise up to symbolic execution for the following example:



Example 3

The call on P within Q can not modify the currently addressable incarnation of x as a consequence of the scoping lemma. Plugging into formula 2.1,

```
P CALL Q
Q CALL R
R (DIRECTMOD and SCOPE) x
```

shows that P MOD/2.1 x. We generalize from this example to produce these formulas:


```
MOD := (CALL* DIRECTMOD) and SCOPE        (3.1)
USE := (CALL* DIRECTUSE) and SCOPE        (3.2)
```

Formula 3.1 produces information for example 3 which is completely precise since P ~SCOPE x. It produces the same information as formula 2.1 on example 1. Unfortunately, it fails to be as precise as formula 2.1 on example 2. Before developing formulas which are completely precise on all of these examples, we pause to prove the correctness of the third set of formulas.

Claim: MOD/3.1 is correct.

> Justification: Using the scoping lemma, it follows that a procedure can have summary effects only on variables declared at outer levels. This is precisely what distinguishes formula 3.1 from formula 1.1. That is, P ~MOD/3.1 x implies either P ~MOD/1.1 x (in which case the correctness of formula 1.1 insures that the absence of (P,x) does not make MOD/3.1 incorrect) or P ~SCOPE x (in which case the scoping lemma insures that the absence of (P,x) does not make MOD/3.1 incorrect).

The desirable properties of all of these formulas can be combined:

```
MOD := (CALL* (DIRECTMOD and SCOPE)) and SCOPE        (4.1)
USE := (CALL* (DIRECTUSE and SCOPE)) and SCOPE        (4.2)
```

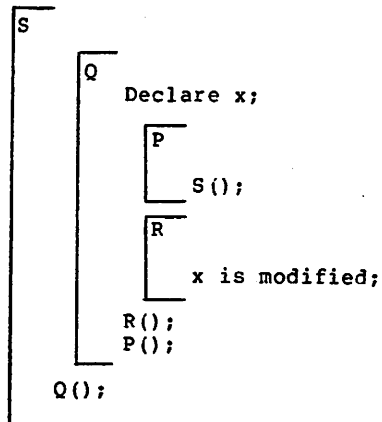
These formulas produce precise data flow information for all of the above examples.

Claim: MOD/4.1 is correct.

> Justification: The combined arguments for the first three sets of equations implies the correctness of formula 4.1. Suppose that P ~MOD/4.1 x. We will show that (P,x) is also absent from either MOD/2.1 or MOD/3.1 which will imply that the absence of the pair (and the previous arguments) can not make MOD/4.1 incorrect. If P MOD/2.1 x then P ~SCOPE x and P ~MOD/3.1 x. (Note that the above argument also shows that MOD/4.1 is at least as precise as MOD/2.1 and MOD/3.1).

Although it is possible to construct examples for which formulas

4.1 and 4.2 are not precise up to symbolic execution, these formulas are the ones recommended for use in practice. Formula 4.1 fails to be completely precise on this complicated example:



The call on P within Q can not modify the addressable x thru the call on S but:

> P CALL* R,
> R (DIRECTMOD and SCOPE) x,
> and P SCOPE x

so P MOD/4.1 x.

Completely characterizing the effects of scoping rules on MOD and USE information will result in formulas for them which are computationally less efficient. It will be convenient to introduce some notation for this characterization.

> Def: A <u>call chain</u> is an ordered sequence of procedures which are pairwise in the CALL relation. Thus, P CALL* S as a result of P CALL Q, Q CALL R, and R CALL S results in P,Q,R,S as a call chain.

> Def: Let the <u>call chain level</u> be the level of the outermost procedure in the call chain. In the example above, the call chain level is min(level P, level Q, level R, level S).

> Def: Let <u>MAXCHAINLEVEL</u> be a |PP| x |PP| matrix of integers where rows and columns are selected by supplying procedure names. MAXCHAINLEVEL[P,Q] is the maximum call chain level for all call chains from P to Q.

Calculating MOD and USE can be accomplished with these formulas:

$$\text{MOD} := \{ (P,x) \mid \text{For some } Q \in PP,\ P \text{ CALL* } Q,\ Q \text{ DIRECTMOD } x, \text{ and level } x < \text{MAXCHAINLEVEL}[P,Q] \} \quad (5.1)$$

$$\text{USE} := \{ (P,x) \mid \text{For some } Q \in PP,\ P \text{ CALL* } Q,\ Q \text{ DIRECTUSE } x, \text{ and level } x < \text{MAXCHAINLEVEL}[P,Q] \} \quad (5.2)$$

The intuition which justifies the use of the maximum chain level in the formulas is that one may only be certain that some pair, (P,x), is absent from the computed MOD if the call chains which result in the modification of x all involve the call of a procedure at a level at least as low as x. If a call chain of maximum level contains a procedure at a level as low as x, then all other call chains must also.

Claim: MOD/5.1 is correct.

> Proof: Suppose that P ~MOD/5.1 x. For a contradiction, assume that P does modify the addressable x from some point of call. Since P can modify x, there is some call chain, C, beginning with P, and ending with the procedure, Q, which directly modifies x. The call chain level of C <= MAXCHAINLEVEL[P,Q] by the definition of MAXCHAINLEVEL. The modification by Q of the same incarnation of x which is addressable at the call of P is only possible if level x < call chain level of C (by the scoping lemma). Thus, the level of x < MAXCHAINLEVEL[P,Q] and all three conditions of formula 5.1 are satisfied, a contradiction.

Claim: Formula 5.1 calculates MOD precisely up to symbolic execution.

> Proof: We must show that the elimination of any pair, (P,x), from MOD/5.1 results in an incorrect summary relation (assuming that all paths of condi-

tional execution are executable). Suppose that P MOD/5.1 x. Look at any calling site for P. We know that there is some call chain beginning with P and ending with a procedure Q which modifies some incarnation of x. Since all paths of conditional execution are executable, we may assume that P calls Q through a call chain of maximal level. Since the level of this call chain exceeds the level of x, we know that no incarnation of x is created between the time P is called and the time Q modifies some incarnation of x. We also know that the variables addressable at the level of x from the calling site are the same as the variables addressable at that level from Q (or in particular, not only is x addressable at the calling site of P, but it is the same incarnation of x which is modified by Q). We have proven that eliminating (P,x) from MOD/5.1 produces incorrect information. (1)

Corollary: For languages like BCPL which do not allow the nesting of procedures, formula 2.1 is precise up to symbolic execution.

Proof: All call chains in such languages have a call chain level of 1. SCOPE selects effects on global (level 0) variables. It follows that all calculated pairs satisfy the three conditions of formula 5.1.

Note that when analogous formulas for NOTPRE are derived, it will not be true that the information is precise up to symbolic execution. This is one of the places in which the theory of may information differs from that of must information.

Having studied formulas for MOD and USE which are precise up to symbolic execution, we are in a position to argue convincingly that formulas 4.1 and 4.2 should empirically be good heuristic methods. Here once again are those formulas:

----

(1) If the spelling of x were not distinct from all other variables, it might be possible that at the calling site x "is on the run time stack" but is not addressable because a more local variable has the same spelling.


```
MOD := (CALL* (DIRECTMOD and SCOPE)) and SCOPE        (4.1)
USE := (CALL* (DIRECTUSE and SCOPE)) and SCOPE        (4.2)
```

We will claim, without proof that formula 4.1 is the same as

```
MOD := { (P,x) | For some Q ∈ PP,
                 P CALL* Q, Q DIRECTMOD x, and
                 level x < min(level P, level Q)  }
```

Thus, formula 4.1 differs from the chain level calculation only in cases in which MAXCHAINLEVEL[P,Q] < min(level P, level Q). What this equation says is that if the "highest" level chain (innermost) from P to Q must go through some procedure less deeply nested than either P or Q, then formula 4.1 fails to be as precise as formula 5.1. This is a somewhat pathological condition which one can expect to arise rarely in practice. This matter has been studied empirically and the results are presented in [5].

Calculating NOTPRE, no sharing:

In this section, formulas for computing NOTPRE from MUSTCALL and DIRECTNOTPRE will be presented. It will be shown that the calculation of this summary information involves very different considerations than those which applied to the calculation of MOD and USE. As in the previous section, the assumption that there is no aliasing of variables will be used here.

In a language without recursion, the calculation of NOTPRE could be accomplished using the formula:

NOTPRE := MUSTCALL* DIRECTNOTPRE (6)

Formula 6 is not correct for programs with recursion as the following example proves:

```
P
   Declare x;

   P();
   x is unconditionally assigned;
```

The call on P does preserve the value of the addressable incarnation of x, although

P MUSTCALL* P and
P DIRECTNOTPRE x.

Using SCOPE in NOTPRE formulas, as it was used in calculating MOD and USE, would have the effect of complicating the examples for which the formulas are incorrect, but they would remain incorrect. At the end of this section, the call chain argument is modified to produce a correct, but computationally inefficient, method for calculating NOTPRE.

Before presenting correct formulas for recursive programs which will be based on formula 6, its correctness for nonrecursive programs must be established.

Claim: Formula 6 is correct for calculating NOTPRE for nonrecursive programs.

Justification: We assume that (P,x) $\in$ NOTPRE as calculated by formula 6 for some procedure P and some variable x. We can assume that x is addressable at some calling site of P (or the relation is trivially correct). From the formula, there is some procedure Q, such that P MUSTCALL* Q and Q DIRECTNOTPRE x. By the definitions of MUSTCALL and DIRECTNOTPRE, Q must be called and it must assign x before P returns. Therefore, the call on P must have the summary effect of assigning x. Since the program is nonrecursive, the incarnation of x which is assigned by Q is the same one as the incarnation addressable at the point of call of P.

Extending formula 6 for use on recursive programs can be accomplished in several ways.

Def: Let <u>GLOBAL</u> be a relation defined on PP x VV. For every <u>P $\in$ PP</u>, a pair (P,x) is in GLOBAL iff x is a global variable (i.e. is defined at the outermost level).

A superset of GLOBAL can be defined which will have all of the desirable properties of GLOBAL and which will result in the calculation of somewhat more precise summary data flow information.

Def: Let <u>NONRECURSIVE</u> be a relation defined on PP x VV. For <u>every P $\in$ PP</u>, a pair (P,x) is in NONRECURSIVE iff x is either a global variable or a local variable to a procedure, Q, which can not call itself (directly or indirectly). That is, Q $\sim$CALL$^+$ Q.

These relations can be used to calculate NOTPRE for recursive programs:

NOTPRE := (MUSTCALL* DIRECTNOTPRE) and GLOBAL (7)

NOTPRE := (MUSTCALL* DIRECTNOTPRE) and NONRECURSIVE (8)

Claim: NOTPRE/8 is correct.

Justification: Building on the arguments which applied to formula 6, if x is a global variable, then the addressable incarnation of x at the point of call of P is the same as the incarnation which Q assigns. If x is local to a nonrecursive procedure, and x is addressable at the point of call of P, then no new incarnation of x could have been created before P returns. If x is not addressable at the point of call, then the relation is automatically correct at that calling site.

Claim: NOTPRE/7 is correct.

Justification: Follows immediately from above.

Formula 7 is sufficiently powerful to be used in languages which do not allow the nesting of procedures. In this case, the variables local to the procedure in which a call is processed are always preserved, and the only other addressable variables are globals. Formula 8 will be part of the formula that is recommended for use in practice for languages with Algol naming conventions. It will be possible to obtain more precise information at the expense of additional computation.

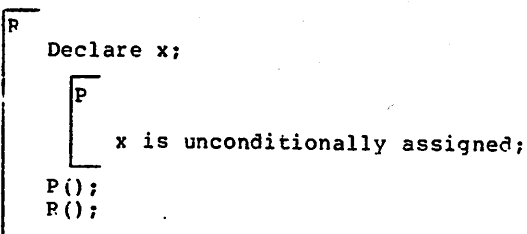
Def: Let <u>UPCALL</u> be a relation defined on PP x PP. A pair (P,Q) is in UPCALL iff the level of P is less than the level of Q.

NOTPRE := ((MUSTCALL and UPCALL)* DIRECTNOTPRE) and SCOPE (9)

Claim: NOTPRE/9 is correct.

Justification: Once again we build on the argument which followed formula 6. We know that there exists some procedure, Q, such that P (MUSTCALL and UPCALL)* Q and Q DIRECTNOTPRE x. The variable x must be declared at a lower level than P, since P SCOPE x. The call chain from P to Q can not produce new incarnations of variables at the level of x, since the minimum level of any procedure in that call chain is the level of P, which is greater than the level of x. Therefore, the incarnation of x which is assigned by Q is the same as the incarnation of x addressable at the point of call on P.

Formula 9 captures "must not be preserved" information in the important special case that a local variable of a recursive procedure is assigned by a nested procedure:



Here, R is recursive, so formula 8 does not compute P NOTPRE x but formula 9 finds that

P (MUSTCALL and UPCALL)* P,
P DIRECTNOTPRE x, and
P SCOPE x.

Since formulas 8 and 9 correctly compute subsets of possible pairs in NOTPRE, they can be combined:

NOTPRE := ((MUSTCALL* DIRECTNOTPRE) and NONRECURSIVE) or
(((MUSTCALL and UPCALL)* DIRECTNOTPRE) and SCOPE) (10)

This formula computes NOTPRE for globals, locals to nonrecursive procedures, and locals to recursive procedures which are assigned by inner procedures. This is the computational method recommended for use in languages with Algol naming conventions.

For completeness, the call chain characterization of NOTPRE will be derived.

Def: A <u>must call chain</u> is an ordered sequence of <u>procedures which</u> are pairwise in the MUSTCALL relation.

Def: Let MINCHAINLEVEL be a |PP| x |PP| matrix of integers where rows and columns are selected by supplying procedure names. MINCHAINLEVEL[P,Q] is the minimum call chain level for all must call chains from P to Q.

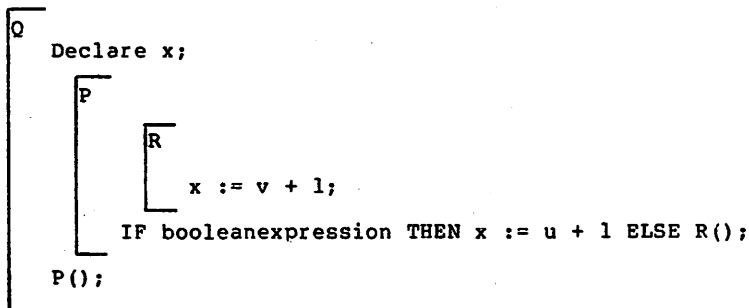
Calculating NOTPRE can be accomplished with this formula:

```
NOTPRE := { (P,x) | For some Q in PP,
                    P MUSTCALL* Q, Q DIRECTNOTPRE x, and
                    level x < MINCHAINLEVEL[P,Q]  }      (11)
```

Claim: NOTPRE/11 is correct.

Proof: Suppose that (P,x) ∈ NOTPRE as computed by formula 11. There is some procedure, Q, such that P MUSTCALL* Q and Q DIRECTNOTPRE x. The must call chain from P to Q could not have produced a new incarnation of x, since all procedures on that chain are at levels which exceed the level of x (by definition of MINCHAINLEVEL and level x < MINCHAINLEVEL[P,Q] by formula 11).

It is possible to argue that MOD and USE are computed precisely up to symbolic execution by formulas 5.1 and 5.2. No such claim is made for the calculation of NOTPRE by formula 11. An example of a program which prevents such a claim from being true follows:



It is obvious that P NOTPRE x, but

P ~MUSTCALL R and
P ~DIRECTNOTPRE x

so by any formula of this section, (P,x) would not have been

included in NOTPRE. This difficulty appears to be fundamental in the calculation of NOTPRE by relational techniques. It is possible to refine NOTPRE information by iterating over the procedure bodies in a manner similar to that described by Allen and Schwartz. [3]

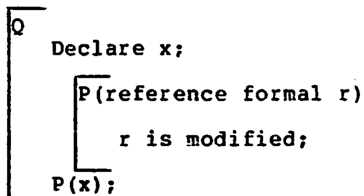
At first it seems plausible that some analogous problem might exist in the calculation of MOD and USE. Fortunately, it does not matter under what patterns of conditional execution procedures are called for these kinds of information. It will be shown in [5] that for data flow problems which have identical composition functions for straightline code as they have for conditionally executed code, the underlying control structure of procedures is irrelevant to the gathering of summary information.

Sharing, MOD and USE:

The task of collecting summary data flow information is made somewhat more difficult by the introduction of reference parameters into the program which is to be analyzed. A simple assignment statement like:

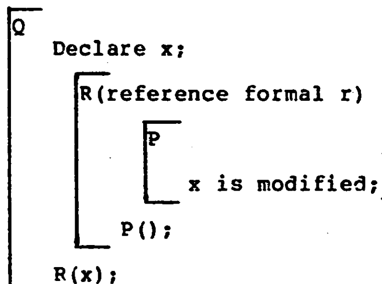
x := u + 1;

can affect variables other than x. These aliasing effects happen in two distinct ways, which we name for future reference:



Refmod Effect

Modifying a reference parameter results in the modification of the actual parameter bound to it. In this example, we must determine that P MOD x.



Varmod Effect

Modification of a variable may result in the modification of reference parameters. Here, P MOD r must be deduced.

In this section, we will study sharing effects on the MOD relation. It should be understood that all the reasoning applies equally well to USE information.

The relation that will enable us to compute summary data flow information in the presence of sharing is:

Def: Let <u>AFFECT</u> be a relation defined on VV x VV. A pair $(\overline{r,x})$ is in AFFECT iff formal reference parameter r is directly bound to actual parameter x at some point of call.

A formula for MOD can now be obtained from formula 1.1 which will be correct in the presence of reference parameters:

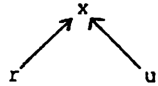
MOD := CALL* DIRECTMOD AFFECT* TRANS(AFFECT)* (12)

Intuitively, refmod effects are accounted for by AFFECT* and varmod effects are computed by TRANS(AFFECT)*. To aid in the correctness proof of formula 12, and for subsequent formulas in this section, we use a lemma which requires the introduction of a few new terms.

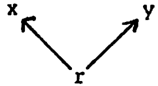
Def: Let {x | r AFFECT* x} be called the <u>set of actuals</u> which may be aliased to r.

Def: Let {r | x TRANS(AFFECT)* r} be called the <u>set of formals</u> which may be aliased to x.

The lemma will attempt to formalize a rather simple idea which is best understood by looking at a series of diagrams. Consider nodes of these graphs to represent variables and directed arcs to represent endpoints in the AFFECT relation. (A reverse arrow represents endpoints in the TRANS(AFFECT) relation.)



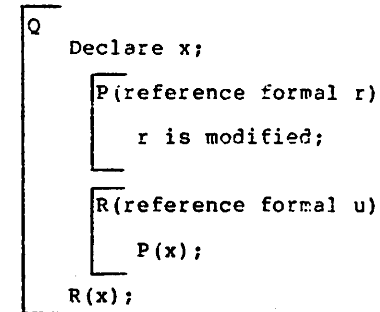
Here we see a graph which is induced by a program in which formal reference parameters r and u are bound to actual parameter x. If both r and u are bound to x simultaneously, modifying r modifies r, x, and u. (The proof of the lemma contains an example of a program in which this occurs.)



This graph represents a program in which formal parameter r takes either x or y as its actual parameter. There is no way to bind both x and y to the same incarnation of formal parameter r, hence modifying x can not result in the modification of y.

<u>Aliasing</u> <u>Lemma</u>: Altering a variable, r, may modify its set of actuals, and variables which are in the sets of formals of these actuals. No other variable may be modified as a sharing effect of the modification of r.

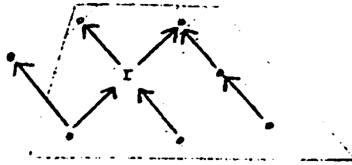
> Proof: The examples which illustrate refmod and varmod effects prove most of the first sentence of this lemma. The following example shows that formals of actuals can be modified through sharing effects:



```
Q
  Declare x;
    P(reference formal r)
      r is modified;
    R(reference formal u)
      P(x);
  R(x);
```

> P modifies r. This modifies x, an actual of r (refmod). Since at the calling site of P within R, u is a formal of x, u is modified by the call on P.
>
> What remains to prove is that no other kinds of effects can arise through sharing. In particular, it must be shown that no other actuals of formals can be modified. That is, if
> r AFFECT x and
> r AFFECT u
> that modifying x will not modify u. At the moment that the program modifies the value of x, it may be bound to some incarnation of the reference parameter r. This incarnation of r can not be simultaneously bound to x, because the incarnation of r is associated with a particular call of the procedure to which it is a formal parameter. Since x affected r (varmod), this call must have had actual parameter x, not u.

The aliasing lemma relates to formula 12 in that AFFECT* TRANS(AFFECT)* has the impact of widening the modification of a variable first to its set of actuals and then widening this to the sets of formals for these actuals. In a diagram, modifying r results in the computation of the possible modifications of all the nodes in the shaded region:



Claim: MOD/12 is correct.

Justification: Follows from aliasing lemma and the proof of formula 1.1.

Combining aliasing effects and scoping considerations is necessary for practical applications. It will turn out that the formulas derived are quite uniform in appearance, but the correctness arguments are quite different at each step of increasing complexity analogous to formulas 2.1, 3.1, and 4.1. For this reason, combining scoping and aliasing considerations will be done in stages which parallel the previous presentation of scoping.

The notion of SCOPE must be recast because in the presence of reference parameters, the modification of a local variable (formal parameter) may have global effects (see for instance the illustration of refmod effects).

Def. Let <u>GENSCOPE</u> be a relation defined on PP x VV which will generalize SCOPE. A pair (P,x) is in GENSCOPE iff the level of x is strictly less than the level of P or x is a formal reference parameter of some procedure.

This differs from SCOPE in that if x is a reference parameter, the pairs (P,x) are automatically in GENSCOPE for all procedures, P.

The first of the formulas which will combine scoping considerations and sharing effects follows:

MOD := CALL* (DIRECTMOD and GENSCOPE) AFFECT* TRANS(AFFECT)* (13)

Claim: MOD/13 is correct.

Justification: We will show for any Q ∈ PP and any y,z ∈ VV such that
P CALL* Q,
Q DIRECTMOD y,
y AFFECT* z, and
z TRANS(AFFECT)* x
that Q ~GENSCOPE y implies that P can not modify x as a summary effect. That is, a pair (P,x) not computed by formula 13, but calculated by formula 12, is superfluous.

Suppose Q ~GENSCOPE y and P MOD x, show contradiction.

Case 1: y is a reference parameter - This case is impossible by the definition of GENSCOPE.

Case 2: y is not a reference parameter - It must be true that y = z by the definition of AFFECT. If Q ~GENSCOPE z, then z must be a local variable of Q (level of z >= level of Q). If x = z, then P can not have a summary effect on x since the scoping lemma applies (modifying a local variable can not result in any summary effects). So, x must be modified by a varmod effect of modifying z. This is impossible since z is local to Q, and can not be bound to any reference parameters at the instant it is modified (Q is executing, so any formals which are potentially bound to the local copy of z are associated with bindings which are not currently in effect).

Formula 13 is analogous to formula 2.1. The next equation echoes formula 3.1:

MOD := ((CALL* DIRECTMOD) and GENSCOPE) AFFECT* TRANS(AFFECT)* (14)

Claim: MOD/14 is correct.

Justification: We will show for any Q ∈ PP and any y,z ∈ VV such that
P CALL* Q,
Q DIRECTMOD y,
y AFFECT* z, and
z TRANS(AFFECT)* x
that P ~GENSCOPE y implies that P can not modify x as a summary effect. That is, a pair (P,x) not computed by formula 14, but calculated by formula 12, is superfluous.

Suppose P ~GENSCOPE x and P MOD x, show contradiction.

Case 1: y is a reference parameter - This case is impossible by the definition of GENSCOPE.

Case 2: y is not a reference parameter - Again, y = z. We also know that level of z >= level of P since P ~GENSCOPE z. If x = z, then the scoping lemma shows that P can not have a summary effect on x. The only case left is that reference parameter x is modified through the varmod effect of modifying z. This implies that z must be bound to x at some call of the procedure to which z is a parameter. If this is a subcall of the call on P, then the incarnation of x in question can not be addressable at the calling site of P (it doesn't exist yet). If, on the other hand, x exists already bound to z, then z will cease to be addressable during the call on P and its subcalls (since level z >= level P) and it is impossible that Q could directly modify this incarnation of z.

Combining formulas 13 and 14 produces the recommended formula for use on programs in languages that allow Algol scoping and reference parameters.

MOD := ((CALL* (DIRECTMOD and GENSCOPE) and GENSCOPE) AFFECT* TRANS(AFFECT)* (15)

Claim: MOD/15 is correct.

Justification: Check that arguments from formulas 13 and 14 can be combined. Using the P, Q, y, z, and x from those arguments, we show that (Q ~GENSCOPE y) or (P ~GENSCOPE y) implies that P can not modify x as a summary effect.

Case 1: y is a reference parameter - contradiction by the definition of GENSCOPE.

Case 2: Q ~GENSCOPE y - argument of formula 13 applies.

Case 3: P ~GENSCOPE y - argument of formula 14 applies.

Sharing, NOTPRE:

An analysis which parallels the aliasing considerations for MOD could be developed for NOTPRE. A relation MUSTAFFECT could be defined to capture the notion that assigning a formal reference parameter, r, must also assign a particular actual parameter, x. There are two reasons for omitting this analysis.

It is highly doubtful that such an analysis would have much practical impact, because refmod effects would be limited to parameters which in the entire program were bound to one, and only one, actual parameter. Varmod effects are probably quite rare in real programs altogether, and there is no obvious way to calculate them for NOTPRE because of the "must" character of the relation.

The second reason to ignore sharing when computing this relation is that it is unnecessary for achieving correct formulas. In computing NOTPRE the formulas which neglect aliasing are correct, even if slightly less precise than would be possible using these techniques. In contrast, MOD and USE formulas which assume no reference parameters become incorrect until the refmod and varmod effects are incorporated into them.

Implementation:

One of the major advantages claimed for the interprocedural data flow analysis algorithm described in this paper is its strictly one pass nature. This section will describe how to implement the algorithm in the first pass of an optimizing compiler. The space and time requirements for the implementation are mentioned, with emphasis on an organization of data which supports bit vector operations.

The idea behind the single pass implementation is that CALL, MUSTCALL, DIRECTUSE, DIRECTMOD, DIRECTNOTPRE, and AFFECT are easily constructed from the program before any interprocedural information is available. In particular, since the order in which procedures are examined is unimportant, it is not necessary to construct a call graph in advance of performing the intraprocedural information gathering. It is also unnecessary to analyze the possible sharing relationships in advance, since they have no effect on the computation of any of the direct relations.

Here is a sketch of an implementation which computes MOD, USE, and PRE in a single pass over a program:

Step 1: FOR each procedure, P, in the program DO Perform an intraprocedural global flow analysis algorithm on P with the power to compute which procedures are in the MUSTCALL relation to P and which variables are in the DIRECTNOTPRE relation to P. This is relatively easy to do intraprocedurally, since it only involves intersecting information on paths of conditional execution for which P terminates normally. Flow analysis is not necessary for computing CALL, DIRECTMOD and DIRECTUSE. Computing the procedures in the CALL relation to P is trivial. Computing variables which are in the DIRECTMOD relation to P includes just those which are modified in statements of P. It is not necessary to worry about sharing effects, since these will be included at a later stage of the algorithm. The passing of an argument by reference to a procedure called in P should not be considered a modification of the variable. (That is, if a variable is passed by reference to a subroutine which does not modify its formal, the algorithm correctly computes the information that the actual parameter is not modified.) The variables in the DIRECTUSE relation to P should include variables passed by value to called procedures, but not those passed by reference. This need not be considered a use of the value of such a variable - the subprocedure does not necessarily use the value of its parameter. Finally, pairs in the AFFECT relation are augmented to include every (formal,actual) pair which is bound at calls with reference parameters within P.

Step 2: Construct GENSCOPE, GLOBAL, and UPCALL from level information associated with procedures and variables. Compute the transitive closure of CALL. Using this, it is now possible to compute NONRECURSIVE.

Step 3: Compute MOD, USE, and NOTPRE using formulas chosen from 1 through 15. To use formulas 5.1, 5.2, and 11 it will be necessary to construct MAXCHAINLEVEL and MINCHAINLEVEL from CALL, MUSTCALL, and level information. To obtain PRE, set PRE to ~NOTPRE.

This implementation assumes that a procedure is not called before it is known which of its arguments are reference parameters. By keeping a table of all arguments supplied to a procedure, this restriction can be eliminated. Since AFFECT is not used until the entire interprocedural information collection pass is complete, delaying its construction until the procedure parameter description has been obtained does not degrade the algorithm.

Having finished the summary data flow analysis pass over the entire program, an optimizing compiler will iterate over the procedures again. In this pass it will compute specific optimization transformations intraprocedurally and generate code. There is no way to combine these passes regardless of the interprocedural data flow analysis algorithm used, since in general, an entire pass over the program may be required to gather summary information for a recursive procedure.

In order to compute formulas on relations efficiently, a bit vector representation is recommended. Composing relations is achieved by conventional boolean matrix multiplication. The operations "and", "or", and "~" are implemented by their corresponding boolean algebra operations, performed bitwise. The "t" operation is matrix transposition.

On most hardware, packing bits into words will result in fast computation provided that all of the information in the word can be used for parallel operations. With this in mind, we describe bit vector representations for the relations which have the property that all computation on a bit vector is parallel. Orienting data in words should reflect the orientation of the bit vectors for greatest efficiency.

The major space considerations for an implementation involve limiting the number of matrixes required to hold intermediate computations (temporaries). This algorithm has certain additional characteristics which will enable compact representations for various data structures.

In order to present these considerations, the computation of

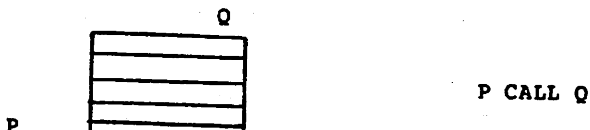
```
MOD := ((CALL* (DIRECTMOD and GENSCOPE)) and GENSCOPE)
            AFFECT* TRANS(AFFECT)*
```

will be illustrated for a large program with the following characteristics:

```
140 Procedures
770 Variables
40  Reference Parameters
10  Naming Levels
```

These are the approximate characteristics of the PASCAL2 6000/3.4 compiler. [4] In no way should this program be considered a typical large program, but it will be useful to use it as a basis for describing one implementation of the summary data flow analysis algorithm. (Note: because of the small number of reference parameters in this program, AFFECT can be represented more efficiently than by normal sparse matrix techniques.)

CALL is stored as rows:



P CALL Q

Warshall's algorithm can be used to compute CALL* in $O(|PP|^2)$ bit vector steps. [7] The resulting matrix will still be represented in rows.

DIRECTMOD should be constructed in columns:

x

P DIRECTMOD x

P

GENSCOPE is a peculiar relation in that it need not completely exist in actual memory. Consider the column representation of GENSCOPE. There are only 11 (the number of naming levels plus one) different columns in the matrix. Columns that correspond to reference parameters are uniformly 1, as are columns which correspond to globals. A variable which is declared at a particular level, induces a pattern characteristic of that level in its column.

Computing (DIRECTMOD and GENSCOPE), both stored as column vectors, requires O(|VV|) bit vector steps and the result is columnar.

Composing CALL* with (DIRECTMOD and GENSCOPE) is a matrix multiplication. It requires O(|PP| |VV|) bit vector steps regardless of the orientation of the resulting matrix. It is most efficient to orient the result in rows.

A row version of GENSCOPE can also be kept implicitly using only 10 rows, one for every procedure level. Just as in the column case, a row has a pattern induced by the level of the procedure it holds information about. Masking the row result of the last computation with the row version of GENSCOPE results in a row oriented relation for ((CALL* (DIRECTMOD and GENSCOPE)) and GENSCOPE) in O(|PP|) bit vector steps.

Storing AFFECT as a matrix of bits is wasteful in space, since it is normally quite sparse. There are a variety of methods which allow sparse matrices to be represented in less space, but one is particularly well suited for use with this algorithm if there are relatively few reference parameters. Let RR be the set of variables which are formal reference parameters. Consider AFFECT and AFFECT*:

	VV - RR	RR
VV - RR	0	0
RR	?	?

AFFECT

	VV - RR	RR
VV - RR	1 (diagonal), 0	0
RR	?	?

AFFECT*

It is obvious that AFFECT and AFFECT* can be stored as |RR| x |VV| bit matrices. It is possible to compute the transitive closure of AFFECT using O(|RR|^2) bit vector operations if the |RR| x |VV| matrix is stored by rows using the following modification of Warshall's algorithm:

```
FOR each column, x, in the |RR| x |RR| matrix in the
lower right corner of AFFECT DO
     FOR each row, r, in that column DO
          IF r AFFECT x THEN
               row r := (row r) or (row x);
```

This version of Warshall's algorithm is simply a restatement of the general algorithm in which steps that are known to merge zero rows into other rows are omitted. To compute the reflexive

transitive closure from the transitive closure, the "diagonal" pairs must be included.

The computed row version of AFFECT* is suitable for use (without modification) as a columnar version of TRANS(AFFECT)*. In order to obtain a useable version of AFFECT*, a columnar version should be computed. It is now possible to complete the computation of MOD using O(|PP| |VV|) bit vector steps with only a bit vector temporary, if the composition with AFFECT* and TRANS(AFFECT)* is done in the correct order.

Total space required for the large sample program which is being used as an example is summarized below for a machine with 60 bit words:

CALL, MUSTCALL, UPCALL	420 words each
DIRECTUSE, DIRECTMOD, DIRECTNOTPRE	1820 words each
GENSCOPE	170 words total
AFFECT, AFFECT*	770 + 520 words

CALL* and MUSTCALL* use the same storage as CALL and MUSTCALL. MOD, USE, and NOTPRE are overlaid on DIRECTMOD, DIRECTUSE, and DIRECTNOTPRE. PRE is overlaid onto NOTPRE. Only one |PP| x |VV| temporary is needed GLOBAL and NONRECURSIVE do not use any storage, since they are uniformly zero or uniformly one in each column. AFFECT becomes TRANS(AFFECT)* through the transitive closure process. AFFECT* is separate storage. The total space required for the large sample program is about 10,000 words, which is moderate by any standard for a problem of this magnitude.

(The existing implementation, written in PASCAL to collect data flow information for PASCAL programs was run on this com-

piler. [4,10] The program is about 6800 lines of code. Performing the data flow analysis took somewhat less time than it takes to compile the compiler using the standard translator. This indicates that not only are the space factors reasonable, but also that the method is sufficiently fast to be practical).

Complexity:

The computational complexity of the bit vector implementation which was described in the previous section is quadratic in $|PP| + |VV|$. Implementing the algorithm on a machine with operations on words of fixed size results in an algorithm of approximately cubic complexity. In this section, it will be shown that, under certain reasonable assumptions, this is asymptotically the best possible algorithm for gathering summary data flow information.

We will consider programs with no sharing and no recursion. In order to rule out gathering trivial summary information (no information), we assume that on a program with no loops, no conditionals, no local variables, and no gotos, an algorithm gathers information which is completely precise. The algorithm described has this property for both may and must information. Under these assumptions, it will be shown that computing summary information is asymptotically as complex as computing reflexive transitive closure.

It is well known that the asymptotic complexity of computing reflexive transitive closure is the same as the complexity of boolean matrix multiplication. [1] Using the implementation described in the previous section, the algorithm presented can be made to run asymptotically as quickly as boolean matrix multiplication and transitive closure, plus the time necessary to scan the program once. Since the program scan is inevitable for any conceivable algorithm, it will be argued that, at least in theory, the algorithm presented is the fastest possible.

The first observation necessary for the reduction to transitive closure is that the computational complexity of computing the reflexive transitive closure of a cycle free graph is the same as the complexity for arbitrary graphs. The reader may find a proof of this fact in the proof of Theorem 5.6 in Aho, Hopcroft, and Ullman, although they do not state this fact. [1]

Let M be an adjacency matrix for some acyclic graph. The computation of the reflexive transitive closure of M, M*, can be embedded in any of the summary data flow problems. We will produce a nonrecursive program with the property that MOD for the program is an interpretation M*. The program consists of procedures, P_i, which are of the following form:

```
P_i
    x_i is modified;
    Call every procedure P_j for which M[i,j] = 1;
```

It is quite obvious that P_i MOD x_j iff $M^*[i,j] = 1$. The program is nonrecursive because the graph represented by M is acyclic. Suppose that M is an n x n matrix. The program constructed has $|PP| = |VV| = n$ so the complexity of computing MOD expressed in $|PP| + |VV|$ is at least as great as the complexity of computing M* expressed in n (up to a constant factor).

It is particularly interesting to note that processing procedures in nonrecursive programs using the reverse invocation order (Allen [2]) does not lessen the computational complexity of the summary data flow problem.

Having shown that gathering summary data flow information in the no sharing and nonrecursive case is as complex as computing reflexive transitive closure, it follows that the sharing and recursive cases are at least as complex.

Conclusion:

A new interprocedural data flow analysis algorithm has been presented and analyzed. There are several fundamental observations which were necessary to arrive at the techniques described in this paper.

Viewing the problem as the calculation of a relation from other relations is a new formulation which permits the clean expression, and verification, of a family of related algorithms. The natural boolean matrix implementation turns out to be reasonable in practice.

In order to support this view of interprocedural data flow analysis, it must be observed that the specific control structure of each procedure is only of limited importance.

There is one observation which is absolutely critical if the data flow analysis is to be performed in a single pass. Under all previous schemes, sharing had to be preanalyzed. This followed from a belief that when processing a statement like:

```
r := u + 1;
```

for reference parameter, r, the modification impact on all variables potentially shared with r had to be recorded. That is, the statement generated the modification information for a set of variables. The new observation was that the fact "r may be modified" can be, as an afterthought, expanded to include the variables shared with r. The statement does not generate the information, but the effect on r does. With this in mind, AFFECT could be computed in parallel with the other direct relations, and composed into the summary information later.

Previous interprocedural data flow analysis techniques treat aliasing effects as operations on equivalence classes, rather than taking the care to notice that the only effects which are possible are characterized by composing AFFECT* with TRANS(AFFECT)*.

The explicit handling of incarnations of variables is new, as are the characterizations which use MAXCHAINLEVEL and MINCHAINLEVEL. These results followed from the initial goal of performing data flow analysis on recursive programs.

An implementation for this algorithm exists for PASCAL programs (written in PASCAL), and it appears to be quite inexpensive to use. In the most general of terms, the data flow analysis of a medium or small program (up to 50 procedures and 300 variables) will take about one third of the time it takes to compile the program using the standard translator. For large programs, on the order of the one described in the implementation section, the time for the data flow analysis approaches the compilation time.

The use of interprocedural data flow analysis in an optimizing compiler at these costs seems practical. In a system which is attempting to generate program diagnostics or automatic documentation, these costs are quite small compared to the surrounding system. The experience with this algorithm convincingly shows that interprocedural data flow analysis is well within what should be considered practical for use in programming language systems.

Bibliography:

[1] Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffrey D. The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Company, Reading Mass. (1974).

[2] Allen, F. E. Interprocedural data flow analysis. Proceedings IFIP Congress 1974, North Holland Publishing Co., Amsterdam (1974), 398-402.

[3] Allen, F. E. and Schwartz, J. T. Determining the data relationships in a collection of procedures. (unpublished detailed summary).

[4] Ammann, Urs. Compiler for PASCAL 6000 - 3.4. ETH, Institut Fuer Informatik, Zuerich (1974).

[5] Barth, Jeffrey M. A practical interprocedural data flow analysis algorithm and its applications. Ph.D. Dissertation, University of California, Berkeley (in preparation).

[6] Graham, Susan L. and Wegman, Mark. A fast and usually linear algorithm for global flow analysis. Journal of the ACM, Vol.23, No. 1 (Jan 1976), 172-202.

[7] Gries, David. Compiler Construction for Digital Computers. Wiley, New York (1971), 39.

[8] Hecht, Matthew S. and Shaffer, Jeffrey B. Ideas on the design of a "quad improver" for SIMPL-T, part I: overview and intersegment analysis. Computer Science Technical Report TR-405, University of Maryland, College Park (August 1975).

[9] Hecht, Matthew S. and Ullman, Jeffrey D. Analysis of a simple algorithm for global flow problems. Conference Record of the ACM SIGACT/SIGPLAN Symposium on the Principles of Programming Languages, Boston, Mass. (October 1973), 202-217.

[10] Jensen, Kathleen and Wirth, Niklaus. PASCAL User Manual and Report. Springer Verlag Lecture Notes in Computer Science No. 18, Berlin (1974).

[11] Lomet, David B. Data flow analysis in the presence of procedure calls. IBM Research Report RC5728, Thomas J. Watson Research Center, Yorktown Heights, New York (November 1975).

[12] Naur, Peter. Revised report on the algorithmic language Algol-60. Communications of the ACM (January 1963).

[13] Ritchie, Dennis M. C reference manual. Bell Telephone Laboratories, Murray Hill, New Jersey (1975).

[14] Rosen, Barry K. Data flow analysis for recursive PL/I programs. IBM Research Report RC5211, Thomas J. Watson Research Center, Yorktown Heights, New York (January 1975).

[15] Rosen, Barry K. High level data flow analysis, part 1 (classical structured programming). IBM Research Report RC5598 Thomas J. Watson Research Center, Yorktown Heights, New York (August 1975).

[16] Rosen, Barry K. High level data flow analysis, part 2, (escapes and jumps). IBM Research Report RC5744, Thomas J. Watson Research Center, Yorktown Heights, New York (December 1975).

[17] Rosen, Barry K. Data flow analysis for procedural languages. IBM Thomas J. Watson Research Center, Yorktown Heights, New York (1976).

[18] Spillman, T. C. Exposing side effects in a PL/I optimizing compiler. Proceedings IFIP Conference 1971, North Holland Publishing Company, Amsterdam (1971), 376-381.

[19] Tarjan, Robert E. Solving path problems on directed graphs. Stanford University Computer Science Department Technical Report STAN-CS-75-528, Palo Alto, Ca. (November 1975).

[20] Wulf, William A., et. al. Bliss: a basic language for implementation of system software for the PDP-10. Carnegie Mellon University Computer Science Department Report (1970).