An Interprocedural Data Flow Analysis Algorithm+

Jeffrey M. Barth

Computer Science Division, EECS
University of California, Berkeley
Berkeley, California 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 can compute all of this information in a 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. The algorithm has been implemented and it is practical for use even on quite large programs.

Introduction:

A great deal of recent research has been devoted to developing algorithms for intraprocedural global flow analysis. [6,9,15,16,19] These methods generally assume that the semantics of individual program statements are available. The purpose of global flow analysis is to determine what information is available at specific program statements by propagating the semantics of individual statements through the program in a manner which reflects the control structure. In practical applications of global flow analysis, such as program optimization, verification, and documentation, subroutine calls appear interspersed among program statements. The semantic effects of subroutine calls are not generally available. Traditionally, optimizers have treated calls as demonous black boxes which terminate the propagation of information. The aim of interprocedural data flow analysis is to summarize the semantic effects associated with subroutine calls, permitting global flow analysis to more effectively propagate information through programs.

We will motivate the study of interprocedural data flow analysis with an example code sequence:

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

+Research sponsored by National Science Foundation Grant MCS74-07644-A02

Whether y/z can be considered a common subexpression depends on the information that SOMEFUNCTION can not modify the values of y and z. This allows the information "y/z is available" to be propagated through the call on SOMEFUNCTION. There are a wide variety of applications for interprocedural data flow analysis, some of which are discussed in [5].

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.

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 nonrecursive programs, there is some ordering in which procedures can be examined which has the property that called subroutines are always analyzed in advance of the procedures which call them. This ordering has been called the "reverse invocation order" by Allen. [2] In the case of recursive programs, there is no ordering with this property. Thus, an interprocedural data flow analysis algorithm which is to be used on recursive programs must include some method of circumventing 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. It is fairly obvious that sharing can be analyzed by a prepass over the program. If a list of variables which are potentially shared with r is available, the processing of the above statement is no longer problematical. In order to perform the data flow analysis in a single pass over the program this strategy is

inadequate since there is no reason to believe that all the aliases of r will have been exposed before a particular statement which modifies r is processed.

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 possible which 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 first to 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 concepts of interprocedural data flow analysis have been developing for about 6 years. The principal algorithms for computing summary information are due to Spillman (1971), Allen (1974), and Rosen(1975). [2,14,18] Each of the algorithms uses a different computational strategy. In addition to this difference, each algorithm computes somewhat different information. Spillman's technique is only suitable for computing "modifies" information, but does so for the complete PL/I language.. Allen's method is usable for computing "modifies", "uses", and "preserves" information, but is poorly suited for use on recursive programs. Rosen's method is the most highly sophisticated, capable of computing arbitrary flow problems with a high degree of precision, even in programs with complex sharing patterns.

These techniques span a continuum of computational expense and expected quality of computed information. Which technique is most appropriate for a particular application will depend on the value placed on the quality of information computed. The present technique is computationally at least as simple as the easiest of the algorithms which preceded it. It does not produce information as precise as Rosen's, and so will not be suitable for the most demanding of applications (verification). It does, however, attempt to stretch as far as possible the precision of calculated information within the constraints of computational efficiency.

The oldest interprocedural data flow analysis technique of which the author is aware was developed by Thomas Spillman. [18] His paper, published in 1971, deals with the issue of exposing side effects of PL/I statements. Thus, the major emphasis of the work is directed toward determining the patterns under which variables share storage. Included in this analysis are language features such as call by reference parameters, pointer variables (possibly with offsets), and ON conditions. His techniques are suitable for determining the side effects of procedure call statements ("modifies" summary information) and for this reason are within the scope of interprocedural data flow analysis.

Spillman's method works essentially as follows: Each procedure is codified into a bit vector in which, loosely speaking, a bit represents the information whether the variable corresponding to its position is modified in the procedure. Bit vectors for procedures are merged into one another in the "reverse invocation order" (a procedure is not processed if possible until all procedures which it invokes are processed). Then, procedures are processed in the invocation order (reverse of above) to account for aliasing effects. If recursion is detected in the program, the above steps are iterated until the bits stabilize.

Cosmeticly, Spillman's method is the most similar to the techniques presented in this paper. We will use a similar representation of information, but will be able to achieve substantially better results in the computation of "modifies" information. In addition, we will use this kind of representation to compute arbitrary data flow problems, which would not have been possible within the conceptual framework available to Spillman. Although Spillman's method only requires a single pass over the program, it is not suitable for computing arbitrary interprocedural data flow information.

The interprocedural data flow analysis algorithm developed by Frances Allen is the most widely used. [2,8] Originally, the algorithm was limited to nonrecursive programs for which there is guaranteed to be a reverse invocation order. Each procedure is subjected to global flow analysis which determines its data flow properties given the

known data flow information associated with each statement.

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 for all but the most demanding of applications. The basic idea exploited by Rosen is that when analyzing (using global flow analysis) 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.

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.

Notation and Problem Definition:

The approach taken to computing summary data flow information by this author is that the process is essentially one of computing relations over the domain of procedures Xvariables. For example, "modifies" is a relation between procedures and the variables possibly modified as a summary effect of their invocation. The interprocedural data flow relations are computed by composition and transitive closure of other relations which are easy to construct from the source program. These easy to construct relations are referred to as <u>direct relations</u> since they are constructed directly from the program without considering subcalls. Relations which represent summary data flow information are referred to as <u>summary relations</u>.

The manner of presentation of the techniques will be to define various relations. The notions of correctness and precision will then be supplied for summary relations. Formulas will show how to compute summary relations from direct relations. The correctness of the summary relations will be verified based on the definitions of the direct relations and the computational formula specified. It will remain to show that the direct relations can be computed to satisfy their definitions easily. This can be accomplished by illustrating an implementation of the data flow techniques (which is only very briefly sketched in this paper).

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 to 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 CALL 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 MUSTCALL 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 DIRECTMOD 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 DIRECTUSE be a relation defined on PP x VV. A pair (P,x) is in DIRECTUSE if procedure P contains a use of the variable x. (may information)

Def: Let DIRECTNOTPRE 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 MOD, USE, and PRE 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 the variable x.

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

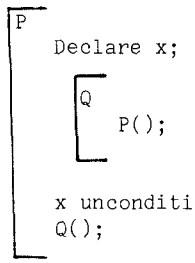
If (P,x) ∈ ~PRE then before P returns, the variable x will have been assigned.

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 NOTPRE 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.)

The above definition of USE is slightly different from what is conventionally calculated in summary data flow analysis. Under this definition, if a variable is read it is considered a "use" of the variable. Traditionally, it would only have been considered a "use" if the value read from the variable could have been the value at the point of call. Obviously, if a pair (P,x) ∈ ~USE then the value of x is also not used as a summary effect. Thus, USE can be substituted for traditional "uses" information in any application. Having made the definition this way, however, prevents certain pairs from being eliminated from USE. Specifically, consider the case in which a variable must be assigned before each program read action on it. By these definitions, there is no way to say that the variable is used, but not its value.

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:



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 correct at a particular calling site 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. $(P,x) \in \neg A$
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 instance 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 precise 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

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	
AFFECT	VV x VV	Direct	May

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 y := 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:

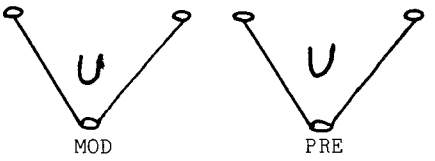
```
x := u + 1;
y := v + 1;
```

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:

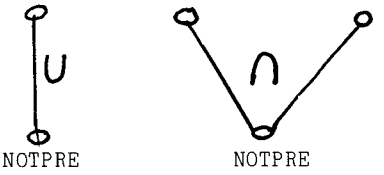
Now consider code which is conditionally executed:

```
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 level of a procedure 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 level of a variable 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 outer and higher levels as inner.

Def: Let SCOPE 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:

Scoping Lemma: 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: (omitted, see [5]).

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]

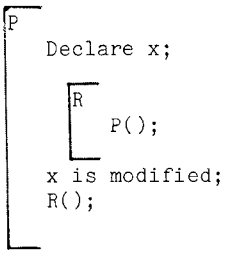
A series of examples which illustrate program skeletons will be useful in developing intuition as to the precision of various formulas used in calculating summary data flow relations. We begin with two examples for which formulas 2.1 and 2.2 are more precise than formulas 1.1 and 1.2. For all the examples, 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

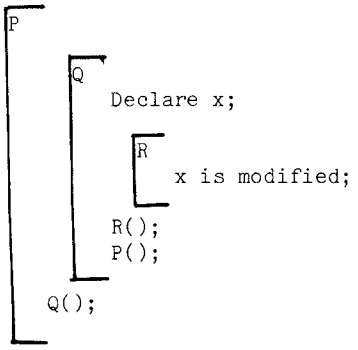
The call on P does not modify the addressable incarnation of x at the point of call. This is the important case of direct recursion.



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: (omitted, see [5]).

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

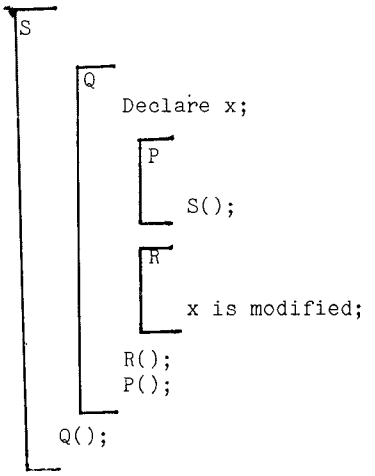
MOD := (CALL* (DIRECTMOD and SCOPE))
and SCOPE (4.1)
USE := (CALL* (DIRECTUSE and SCOPE))
and SCOPE (4.2)

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

Claim: MOD/4.1 is correct.

Justification: (omitted, see [5]).

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 call chain 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 call chain level 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 MAXCHAINLEVEL 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:

MOD := { (P,x) ¦ For some Q ∈ PP,
P CALL* Q, Q DIRECTMOD x, and
level x < MAXCHAINLEVEL[P,Q] } (5.1)

USE := { (P,x) ¦ For some Q ∈ PP,
P CALL* Q, Q DIRECTUSE x, and
level x < MAXCHAINLEVEL[P,Q] } (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 conditional 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)

---

(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.

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.

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:

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:

It is possible to obtain formulas for calculating NOTPRE from MUSTCALL and DIRECTNOTPRE. It turns out that the calculation of this summary data flow information involves very different considerations than those which applied to MOD and USE. An account of these methods appears in [5].

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:

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

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.

```
Q
   Declare x;
      R(reference formal r)
         P
            x is modified;
         P();
   R(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 AFFECT be a relation defined on VV x VV. A pair (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:

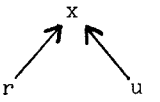
$$MOD := CALL^* \; DIRECTMOD \; AFFECT^* \; TRANS(AFFECT)^* \qquad (12)$$

Intuitively, refmod effects are accounted for by AFFECT* and varmod effects are computed by TRANS(AFFECT)*. (2) 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 set of actuals which may be aliased to r.

> Def: Let {r | x TRANS(AFFECT)* r} be called the set of formals 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.

Aliasing Lemma: 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 shar-

---

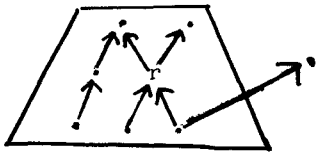
(2) Formulas are numbered to be consistent with [5], thus formulas 6 through 11 do not appear in this paper.

ing. 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). (3)

Def. Let GENSCOPE 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: (omitted, see [5]).

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: (omitted, see [5]).

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: (omitted, see [5]).

Implementation:

One of the major advantages claimed for the interprocedural data flow analysis algorithm described in this paper is its strictly one pass nature. 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 intraprocedural information gathering. It is also unnecessary to analyze the possible sharing relationships in advance, since they have no effect on any of the direct relations.

Complexity:

The computational complexity of the straight forward a bit vector implementation for this algorithm 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

---

(3) Recent work has resulted in an improved method for combining aliasing and scoping considerations. It is similar in spirit to the technique presented here, requires about the same amount of computation, and results in more precise information. It is, however, considerably more complex to present and obscures the intuitive notions emphasized here.

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 a standard matrix representation for relations, 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:

An implementation (written in PASCAL) for this algorithm exists for PASCAL programs , 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.

The implementation was run on the PASCAL 6000/3.4 compiler, which is a program with about 6800 lines of code, 140 procedures and 770 variables. [4,10] Performing the interprocedural data flow analysis (for MOD, USE, and PRE) took somewhat less time than it takes to compile the compiler. The space required was about 10,000 (60 bit) words.

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). This report is superceded by [17].

[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).