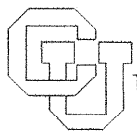
A Precise and Efficient Algorithm for Determining Existential Summary Data Flow Information *

E. W. Myers

CU-CS-175-80



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This work was supported by NSF Grant MCS 77-02194 and ARO #DAAG29-80-C-0094.

A PRECISE AND EFFICIENT ALGORITHM FOR DETERMINING EXISTENTIAL SUMMARY DATA FLOW INFORMATION*

by

E. W. Myers
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado

#CU-CS-175-80 March, 1980

*This work was supported by NSF Grant MCS 77-02194 and ARO #DAAG29-80-C-0094.

THE FINDINGS IN THIS REPORT ARE NOT TO BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, UNLESS SO DESIGNATED BY OTHER AUTHORIZED DOCUMENTS.

CONTENTS

Abstract.

Data flow analysis is well understood at the intra-procedural level and efficient algorithms are available. When inter-procedural mechanisms such as recursion, procedure nesting, and pass-by-reference parameters are introduced, the data flow problems become more difficult. Let ISIZE be the size of the problem and OSIZE be the size of the resulting data flow relation. A O(ISIZE + OSIZE) algorithm is demonstrated for the existential summary data flow problem.

INTRODUCTION:

Data flow analysis problems have been studied extensively for a number of purposes, among them global program optimization, software validation, and program documentation [FO]. Initially, interest centered on determining the data flow patterns visible at a given statement within a procedure. For example, the available expression and live variable problems are of this genre. The algorithms constructed for such analyses [AC, GW, UH, UK] are efficient but strictly intra-procedural in the sense that they do not take into account the effects of other procedures within a program.

More recent attempts have focused on determining these patterns in the presence of inter-procedural effects induced by procedure calls. Once the scope of the problem was so enlarged, an additional class of problems arose. To wit, one could ask what effect the execution of an entire procedure had on the variables of a program. Such information was coined summary data flow information. Current inter-procedural algorithms either compute an approximation to the precise answer up to symbolic execution [Bar], or are very slow [R1, R2, R3], or ignore one or more of the difficult effects such as recursion or aliasing [S,A].

Most data flow analysis problems are intractible in the presence of aliasing. However, one version of the summary problem is not. Just as live and avail are characterized by the use of existential and universal quantification respectively, there are two types of summary problems, one existential, the other universal. This paper presents an efficient and precise algorithm for solving the existential or MAY summary problem [Ban]*. The algorithm operates in two phases. The first

*Near the completion of writing this paper, Banning [Ban] published a paper containing the results given here.

phase determines the side effects of procedure calls. The second part establishes the aliasing pattern of the program. A subsequent paper by this author will deal with the intractible problems.

PROBLEM DEFINITION AND PROGRAM MODEL

The goal of this paper is to compute data flow information which summarizes the effect of every procedure and procedure invocation in a program. This information takes the form of relations between a procedure and the variables it can address. For an invocation the relations are between it and the variables the procedure containing the invocation can address. These relations will be referred to as summary relations and are of two types. Given some basic data flow event such as the usage or definition of a variable, the goal is to determine whether the event must or may occur as the result of executing an entire procedure.

Let P be the set of procedures in a program, I the set of invocations, and V the set of variables. In a block structured language such as Pascal, a name may refer to different variables depending on where the name occurs in a program. By V is meant the set of distinct variable declarations and not the set of names for these variables. This identification problem can easily be taken care of in a prepass of the program. The summary relations are subsets of $P \times V$ and $I \times V$. Let $P \in P$, $I \in I$, and $V \in V$ - -

P(I) MAYAFF V $\equiv$ V is addressable by P(I) and the data flow event in question may happen to V as a result of executing P (invoking I).

P(I) MUSTAFF V $\equiv$ V is addressable by P(I) and the data flow event in question must happen to V as a result of executing P (invoking I).

The degree of accuracy of the information computed is clearly dependent on the assumptions made about the flow of control in a program. Barth [Bar] defines the notions of correctness and precision in a formal way. The goal here is to produce summary information which is precise up to symbolic execution. That is, the aim is to compute the best possible information under the assumption that any path through a program is possible. This differs from Barth's treatment where the assumption for may information is less restrictive. The treatment of the VAR effect for the may case will require that any call chain is possible and hence necessitates the more liberal hypothesis.

It will be shown that the MUST problem is co-NP complete [GJ] in a subsequent paper and hence there is in all likelihood no polynomial procedure for solving it. The remainder of the paper deals exclusively with the MAY problem. The MAY problem has an efficient solution because it is flow insensitive, i.e., it does not depend on the intra-procedural structure of a program. Once one knows that an effect may occur within the body of a procedure, one can conclude that the effect may occur as the result of the execution of the entire procedure. This is not true for MUST problems as the effect must be known to occur on all paths through the procedure's flow graph before one can conclude that the effect must occur for the execution of the procedure as a whole.

The algorithm relies heavily on the assumption of bit vector arithmetic. Modelling sets as bit vectors allows for fast set manipulation, e.g., union, intersection, and complement are all O(1). A shift-until-one type operation is not assumed, however, and hence finding an element in a set will take O(log LEN) where LEN is the length of the bit vector.

The algorithm computes every summary relation as a set of bit vectors - - one for each procedure and invocation. The length of the bit vectors is equal to the size of V. The set of bits that are one represent the variables that satisfy the MAYAFF relation with the corresponding procedure or invocation. In algorithmic notation - -

```
type PROC   : range 1 . . |P|
type INVOKE : range 1 . . |I|
type VAR    : range 1 . . |V|
type VARVEC : bit vector of VAR
declare MAYAFF : array [PROC + INVOKE] of VARVEC
```

This paper assumes a Pascal or Algol-like language which features block structuring and a pass-by-reference parameter passing mechanism. Other parameter passing mechanisms will not affect the interprocedural analysis. The block structuring is modelled in the natural way by a nesting tree T. T is a rooted directed tree whose vertex set is P. $P \rightarrow R$ is an edge in T if and only if R is declared within the body of P. For algorithmic purposes T is modelled as follows - -

```
declare ROOT   : PROC
declare FATHER : array [PROC] of PROC
declare SONS   : array [PROC] of list of PROC
```

Each procedure has a set of pass-by-reference formal parameters and a set of variables which are declared within it. A given procedure P can address any variable declared within it and the variables declared within any ancestor of P in T. Formally, let LOCAL(P) be the set of variables declared in P. Then the set of variables addressable by P is

$$\text{ADDRESS}(P) = \bigcup_{R \in \text{ANCESTOR}(P)} \text{LOCAL}(R)$$

A slightly different set that will also be of interest is SCOPE(P) = ADDRESS(P) - LOCAL(P). For every variable x, LPROC[x] will be the unique procedure P for which $x \in$ LOCAL[P]. The address, scope, and formal parameter sets are modelled as bit vectors. For computational purposes the formal parameter set will also have a list representation. In algorithmic notation - -

```
declare ADDRESS, SCOPE, FORMVEC : array [PROC] of VARVEC
declare LOCAL, FORMLST : array [PROC] of list of VAR
declare LPROC : array [VAR] of PROC
```

The dynamic manner in which procedures invoke or call each other is modelled by a call graph *C*. *C* is a directed graph whose vertex set is *P* and whose edges are labelled with tuples of varying sizes. Each edge represents an invocation in *I*. $P \xrightarrow{<a_1, \ldots, a_n>} Q$ is an edge or invocation in *C* if procedure P contains a call on procedure Q and passes by reference to Q the actual variables a_1 through a_n. This assumes that each invocation has been statically checked, i.e., every formal parameter of Q corresponding to one of these actual parameters is a pass-by-reference parameter. Thus the edges pointing at a given procedure all have tuples of the same size.

C is implemented with a reverse adjacency structure, i.e., CALL[P] is a list of the invocations of procedure P. For each invocation, the calling procedure is given in the array CALLER. The parameter passing mechanism is not modelled as tuples associated with invocations. Instead, imagine that every invocation edge is split into as many edges as there are actual parameters labelling it and each such pseudo-edge is labelled with one of the actual parameters. The array ACTUAL associates each formal variable with a list of these pseudo-edges.

For a given formal parameter f, this list consists of those pseudo-edges which are calls on the procedure in which f is declared and which are further labelled with the actual parameters to which f would be bound on such a call. For each pseudo-edge the corresponding actual argument and invocation edge are in the arrays AVAR and AINV. If x is a variable which is not a formal parameter for any procedure then ACTUAL [x] is the empty list. Algorithmically - -

```
type PARM      : range 1..∑ |the tuple labelling I|
                           I ε INVOKE

declare CALL   : array [PROC] of list of INVOKE
declare CALLER : array [INVOKE] of PROC
declare ACTUAL : array [VAR] of list of PARM
declare AVAR   : array [PARM] of VAR
declare AINV   : array [PARM] of INVOKE
```

In a recursive language such as the one being hypothesized here, the call graph may contain cycles. Due to the static block structuring, however, certain edges are forbidden as indicated in the lemma below.

Invocation Lemma: If $P \rightarrow R \in I$ then FATHER(R) ε ANCESTOR(P)

Proof: Observe that P can address and hence call only those procedures declared in the ancestors of P. Thus since FATHER(R) is the procedure R is declared in, it must be that FATHER(R) ε ANCESTOR(P). ∎

Suppose that procedure P_1 invokes P_2 which in turn invokes P_3 and so on until finally procedure P_n is invoked. Such a sequence of invocations is called a call chain and clearly corresponds to a path from P_1 to P_n in C. The existence of this path is written as $P_1 \rightarrow^* P_n$. If the path from P_1 to P_n is simple (contains no cycles) then the chain is termed a simple chain. The simple invocation lemma has an interesting consequence for call chains - -

Chain Lemma: If $P \rightarrow^* R$ and $P \in ANCESTOR(R)$ then

1/ any chain from P to R contains every procedure in ANCESTOR(R) - ANCESTOR(P)

2/ there is a simple chain from P to R.

Proof: Proof is by induction on the length of the path from P to R.

The basis is trivial as $P \rightarrow R$ and $P \in ANCESTOR(R)$ and $FATHER(R) \in ANCESTOR(P)$ (invocation lemma) implies either $P = R$ or $P = FATHER(R)$.

Suppose the lemma is true for paths of length k. Let $P \rightarrow^* R \equiv P \rightarrow^* Q \rightarrow R$ be a path of length $k+1$. If $P = R$ the result is immediate. Otherwise $P \in ANCESTOR(R) - R$ and by the invocation lemma $FATHER(R) \in ANCESTOR(Q)$. This implies that $P \in ANCESTOR(Q)$ and thus by the inductive hypothesis there is a simple path from P to Q, $P \rightarrow^* Q$ and every element of ANCESTOR(Q) - ANCESTOR(P) is on the chain $P \rightarrow^* Q$.

Suppose R is not on the path $P \rightarrow^* Q$. Then clearly $P \rightarrow^* Q \rightarrow R$ is a simple chain from P to R. If R is on the path $P \rightarrow^* Q$ then truncate that part of the path that follows R's occurrence. The path from P to R that remains must be simple as it is a subpath of the simple chain $P \rightarrow^* Q$. Note that this part of the lemma does not depend on the invocation lemma.

$Q \rightarrow R$ implies by the invocation lemma that $FATHER(R) \in ANCESTOR(Q)$

$\Rightarrow ANCESTOR(R) - R \subseteq ANCESTOR(Q)$

$\Rightarrow ANCESTOR(Q) - ANCESTOR(P) \equiv [ANCESTOR(R) - ANCESTOR(P)] - R$

Thus the path from P to R must contain every element in ANCESTOR(R) - ANCESTOR(P). ∎

One last set of items is needed to complete this paper's hypothetical model. These sets reflect the particular data flow event of interest and are the only structures in which the specific event is relevant. For every variable in V one needs to know the set of procedures in which the event of interest is known a priori to happen. For example, if the definition of a variable is the data flow event in question then one needs to know every procedure which contains a statement assigning a value to the variable. The possible side effects of procedure invocations within basic blocks are ignored. The algorithm described here will use these direct events to determine whether the event may happen as the result of the execution or invocation of a procedure. This information takes the form of an array of bit vectors. For each procedure P, DIRAFF[P] will be the set of those variables directly used in P. Formally, - -

declare DIRAFF : array [PROC] of VARVEC

INTER-PROCEDURAL EFFECTS

The focus of this section is on the ways that the inter-procedural mechanisms of scope and parameter passing affected summary relations. There are three ways in which a procedure's summary relation can be affected as the result of invocations within it. The first effect is a result of the block structuring. The other two effects are the result of the pass-by-reference mechanism.

One of the difficulties in analyzing these effects is the fact that the recursive nature of the language being modelled allows for a variable to be instantiated more than once. For example, suppose procedure P calls itself and has a local variable x. Each time P is called a new instance or incarnation of x is created. In the analysis that follows, it must be guaranteed that if two procedures, P and R, refer to a variable x then they refer to the same incarnation of x. Suppose $P \rightarrow R$ in C and P and R can both address x. In this instance, it is clear that they address the same incarnation of x as long as a new instance of x is not created by the invocation of R, i.e., x is not a local variable of R. Using the invocation lemma one can arrive at the following subtle statement of this condition - -

Incarnation Basis: If $P \rightarrow R$ in C then P and R address the same incarnation of $x \Leftrightarrow x \in SCOPE(R)$

Proof: ($\Rightarrow$) R can address $x \Leftrightarrow x \in ADDRESS(R)$

R addresses the same incarnation as $P \Rightarrow x \notin LOCAL(R)$

$x \in ADDRESS(R)$ and $x \notin LOCAL(R) \Leftrightarrow x \in SCOPE(R)$

($\Leftarrow$) $x \in SCOPE(R) \Rightarrow$ R can address x and $x \notin LOCAL(R)$

$x \in SCOPE(R) \Rightarrow x \in ADDRESS(FATHER(R))$ (1)

$P \rightarrow R \Rightarrow FATHER(R) \in ANCESTOR(P)$ [invocation lemma] (2)

(1) and (2) $\Rightarrow x \in ADDRESS(P)$

Thus R and P can both address x and $x \notin LOCAL(R)$ implies they address the same incarnation of x. ∎

Extending this condition to a call chain $P \rightarrow^* R$ is simple. Let $\{P \rightarrow^* R\}$ denote the set of all procedures on the chain from P to R except for P. One must simply insure that x is not local to any procedure in $\{P \rightarrow^* R\}$.

Incarnation Lemma: If $P \rightarrow R$ is a call chain in C then P and R address the same incarnation of $x \Leftrightarrow x \in \bigcap_{Q \in \{P \rightarrow^* R\}} SCOPE(Q)$

Proof: Suppose the call chain is $P \equiv Q_0 \rightarrow Q_1 \rightarrow Q_1 \rightarrow Q_2 \rightarrow \ldots \rightarrow Q_{n-1} \rightarrow Q_n \equiv R$

P and R address the same incarnation of x

$\Leftrightarrow \forall i > 0$ (Q_{i-1} and Q_i address the same incarnation of x)

$\Leftrightarrow \forall i > 0$ ($x \in SCOPE(Q_i)$) [incarnation basis]

$\Leftrightarrow x \in \bigcap_{i>0} SCOPE(Q_i)$ ∎

The first interprocedural effect is called the SCOPE effect because of its connection with the block structuring of the language. Suppose $P \rightarrow R$ in C and $x \in MAYAFF[R]$. Then x satisfies the MAYAFF relation at the basic block containing the invocation of R if and only if P and R both address the same incarnation of x. Using the incarnation basis, the statement of the conditions for this effect is easy - -

SCOPE Effect: If $P \rightarrow R$ in C and $x \in SCOPE(R)$ then

$$x \in MAYAFF[R] \Leftrightarrow^{*} x \in MAYAFF[P \rightarrow R]$$

The first of the two aliasing effects is called the FORMAL effect because the addition of a formal parameter to the MAYAFF relation implies that its corresponding actual parameter is in its invocation's MAYAFF relation. Suppose $P \rightarrow R$ in C and the tuple labelling this invocation results in actual parameter, a, binding to formal parameter, x. Clearly if $x \in MAYAFF[R]$ then by the aforementioned alias, it must be that $a \in MAYAFF[P \rightarrow R]$.

FORMAL Effect: If $P \xrightarrow{<..a..>} R(..x..)$ then

$$x \in MAYAFF[R] \Leftrightarrow^{*} a \in MAYAFF[P \rightarrow R]$$

The final effect is quite different from the FORMAL and SCOPE effects. The latter can be viewed as a propagation of members of summary relations from an invoked procedure to the source of an invocation of that procedure. The ALIAS effect, however, is a broadening of the relation at each procedure and invocation due to the establishment of aliases along call chains. The effect is called the ALIAS effect because a variable is included in a summary relation due to its being aliased along a call chain to a variable already in the relation. Suppose an alias is established between a and x by the invocation $P \xrightarrow{<..a..>} R(..x..)$. This alias stays in effect on any chain continuing from R. These aliases concatenate along chains as follows - -

---

* The equivalence assumes that the effect in question is the only interprocedural mechanism in operation. If more than one effect is in operation then a variable may be included in an invocations MAYAFF relation by some other mechanism. The idea is that the formal condition describes exactly those conditions under which a specific effect takes place.

Alias Concatenation: Suppose there is a chain to P on which a and x are aliased, and that $P \xrightarrow{<..a..>} R(..y..)$. Then x is aliased to y along any extension of this chain continuing through $P \rightarrow R$.

Aliasing is transitive along specific chains - -

Alias Transitivity: Suppose there is a chain on which a and x are aliased and a and y are aliased. Then x and y are aliased along this chain.

A more accurate characterization of the structure of the chains that give rise to these aliases will be made in the section in which the ALIAS-effect is solved.

Suppose $x \in MAYAFF[P]$ and a is aliased to x at P on some chain to P. By the symbolic execution hypothesis this call chain can be executed and hence $a \in MAYAFF[P]$ as a could be aliased to x at P. The possibility that a is not addressable at P has not yet been ruled out. (Recall that $MAYAFF[P] \subseteq ADDRESS[P]$). One must show that when $a \notin ADDRESS[P]$ the inclusion of a in MAYAFF[P] does not result in the subsequent inclusion of valid summary relation members elsewhere. $a \notin ADDRESS[P] \Rightarrow a \notin SCOPE[P] \Rightarrow$ a cannot propagate due to the SCOPE-effect. $a \notin ADDRESS[P] \Rightarrow LPROC[a] \neq P \Rightarrow$ a cannot propogate due to the FORMAL-effect. Hence one can assume that $a \in ADDRESS[P]$.

ALIAS Effect: If x is aliased to f on some chain to Q and $f \in ADDRESS[Q]$ then

$$x \in MAYAFF[Q] \Leftrightarrow^* f \in MAYAFF[Q]$$

$$x \in MAYAFF[Q \rightarrow R] \Leftrightarrow^* f \in MAYAFF[Q \rightarrow R]$$

INTER-PROCEDURAL ALGORITHM: PROPAGATION PHASE

The inter-procedural algorithm consists of two distinct phases. The first phase is an iterative scheme which propagates SCOPE and FORMAL effects around the call graph. The ALIAS effect is the concern of the algorithm's second phase and will be dealt with later. The propagation algorithm is straightforward. Procedures in the call graph are processed in an arbitrary order but care is taken that a particular summary bit is propagated through a given procedure at most one time.

Observe that a variable may propagate from a procedure Q to an invocation of Q only by a SCOPE or FORMAL effect. The only bits which these effects can propagate are those in FORMVEC[Q] ∪ SCOPE[Q]. These variables are termed the critical variables of Q. The main correctness assertion for the algorithm is the fact that - -

CRIT[Q] ≡ those critical variables of Q which have been added to MAYAFF[Q] since the last time critical variables were propagated through Q.

A queue of those procedures P, for which CRIT[P] ≠ 0 is kept. Procedures are pulled off this queue at random and their new critical variables are propagated to their invocations, possibly enlarging the CRIT sets of the calling procedures. This process continues until the queue is empty at which point it is asserted that the MAYAFF sets are correct with regard to the FORMAL and SCOPE effects. Termination is guaranteed by the strictly increasing size of the MAYAFF relation.

The propagation of critical bits through a procedure P is quite simple - -

new-scope = CRIT[P] ∩ SCOPE[P] ≡ the critical bits that propagate because of the scope effects.

new-formal = CRIT[P] ∩ FORMVEC[P] ≡ the critical bits that are formal parameters.

new-actual[e] = {a|x ε new formal and x corresponds to a along edge e} ≡ the bits that are in MAYAFF[e] because of the FORMAL effect.

new-use[e] = (new-scope ∪ new-actual[e]) - MAYAFF[e] ≡ the new bits that are set in MAYAFF[e] because of critical bit propagation.

Let Q = CALLER [e]. Each of the bits in new-use [e] can be added to MAYAFF[Q] as the data flow effect is flow insensitive. If this results in an addition to CRIT[Q] then Q is put on the queue if it is not already there.

The queue is initialized as follows. Initially it is known that MAYAFF[P] is exactly DIRAFF[P] for each procedure P. Using this set, CRIT[P] is computed. If CRIT[P] is not empty then P is initially placed on the queue.

The primitive "turn VEC into list LIST" is assumed to turn the vector representation VEC, of a set into its corresponding list representation LIST. As stated earlier one can find each bit that is set in VEC in time O(S log LEN) where LEN is the length of the bit vector VEC and S is the number of bits set in VEC. The primitive operations for the queue QUEUE should be obvious.

PROPAGATION ALGORITHM:

```
declare  a:PARM, x:VAR, e:INVOKE
declare  p,q:PROC
declare  QUEUE:queue of PROC
declare  new-scope, new-form, new-use, new-crit:VARVEC
declare  new-f-list:list of VAR
declare  CRIT:array[PROC] of VARVEC
declare  new-act:array [INVOKE] of VARVEC
```

#Initialize the summary relation on each invocation, the CRIT-sets of each procedure, and the queue#

```
1.  for e ε INVOKE do
2.        MAYAFF [e] ← 0
3.  QUEUE ← 0
4.  for p ε PROC do
5.        MAYAFF[p] ← DIRAFF[p]
6.        CRIT[p] ← MAYAFF[p] ∩ (FORMVEC[p] ∪ SCOPE[p])
7.        if CRII[p] ≠ 0 then
8.              QUEUE ← QUEUE + p
```

#While the queue is not empty pick a procedure p off it#

```
9.  while QUEUE ≠ 0 do
10.       pick p from QUEUE arbitrarily
```

#Compute new-form and new scope. Reset CRIT[p]#

```
11.       new-form ← FORMVEC[p] ∩ CRIT[p]
12.       new-scope ← SCOPE[p] ∩ CRIT[p]
13.       CRIT[p] ← 0
```

```
#Compute new-act[e] for each e in CALL[p]#

14.        turn new-form into list new-f-list
15.        for e ε CALL[p] do
16.              new-act [e] ← 0
17.        for x ε new-f-list do
18.              for a ε ACTUAL[x] do
19.                    new-act [AINV [a]] ← new-act [AINV [a]] + AVAR [a]

#Compute new-use[e] for each e#

20.        for e ε CALL[p] do
21.              new-use ← (new-act [e] ∪ new-scope) - MAYAFF [e]
22.              if new-use ≠ 0 then

#Compute  MAFF[e].  Call MBLK with each new bit and for q = CALLER[e],
add q to the queue if its CRIT-set is nonempty.

23.                    MAYAFF[e] ← MAYAFF[e] ∪ new-use
24.                    q ← CALLER[e]
25.                    new-crit ← (new-use - MAYAFF[q]) ∩
                                  (FORVEC[q] ∪ SCOPE[q]
26.                    if CRIT[q] = ∅ and new-crit ≠ ∅ then
27.                           QUEUE ← QUEUE + q
28.                    CRIT[q] ← CRIT[q] ∪ new-crit
29.                    MAYAFF[q] ← MAYAFF[q] ∪ new-use
```

The worst case time estimates for this algorithm will be given not only in terms of the size of the input but also the size of the output. To this end let MAY_{PROC} be the size of the summary relation over all procedures, i.e., $MAY_{PROC} = \sum_{P \in PROC} |MAYAFF[P]|$. Let $MAY_{INVOKE} = \sum_{I \in INVOKE} |MAYAFF[I]|$

The cost analysis with regard to time demonstrates a $O(PROC+INVOKE+PARM+MAY_{INVOKE}+MAY_{PROC}* \log VAR)$ asymptotic behavior. The loop in lines 1 and 2 is executed exactly INVOKE times; line 3 is O(1); and the loop in lines 4 through 8 is executed PROC times at constant cost per iteration. Each time a procedure P is pulled off the queue, at least one new bit is in CRIT[P] since the last time P was pulled from the queue. Since CRIT[P] $\subseteq$ MAYAFF[P], the main loop (lines 9-29) is executed at most MAY_{PROC} times. Lines 10 through 13 of this loop take a constant amount of time. A maximum of MAY_{PROC} bits will be found in statement 14 as new-form $\subseteq$ CRIT[P]. Thus a maximum of $MAY_{PROC}* \log VAR$ time will be spent on this statement.

The bodies of the two invocation loops (statements 15-16 and 20-29) are executed at most MAY_{INVOKE} + PARM times by the following argument. If new-scope $\neq$ 0 then new-use $\neq$ 0, implying that a new bit is added to MAYAFF[e] for each e. Thus each iteration may be charged to one of these MAY_{INVOKE} additions. If new-scope = 0 then it must be that new-f-list $\neq$ 0 and thus at least one formal-actual correspondence is being made. These take place only once for each actual parameter and hence a total of PARM may be charged to this event. From the second part of the preceding argument, one can also conclude that line 19 is executed at most PARM times. The code block in lines 23 through 29 is executed at most MAY_{INVOKE} times as new-use $\neq$ 0 when this block is executed.

INTER-PROCEDURAL ALGORITHM: CLOSURE PHASE

The second and final phase of the inter-procedural analysis is basically a transitive-closure computation which solves for the ALIAS effect. In order to do this, one needs a better characterization of which aliases are relevant and the manner in which they come about. In order for the ALIAS effect to take place at a procedure Q, the two variables involved, say x and y, must be addressable at Q. So the concern is to find which variables in ADDRESS[Q] are aliased along some chain to Q. Suppose this is true for x and y. Then WLOG $x \in$ ADDRESS [LPROC[y]]. So it suffices to find those ordered pairs <x,y> for which $x \in$ ADDRESS[LPROC[y]] $\subseteq$ ADDRESS[Q] and x is aliased to y on some chain to Q. <y,x> can be inferred later from the symmetry of the relation. The next lemma simplifies matters even further by showing that it is only necessary to know when x is aliased to y on some chain to LPROC[y].

Local Lemma: <x,y> on some chain to Q $\Leftrightarrow$
$y \in$ ADDRESS[Q] and <x,y> on some chain to LPROC[y].

Proof: ($\Rightarrow$) $y \in$ ADDRESS[Q] $\Rightarrow$ Q is a descendant of LPROC[y] in the nesting tree. Suppose x is not aliased to y on all chains to LPROC[y]. Any chain to Q passes through LPROC[y] by the chain lemma. Thus x cannot be aliased to y on any chain to Q (Contradiction).

($\Leftarrow$) Since $y \in$ ADDRESS[Q], one concludes from the chain lemma that every chain to Q passes through LPROC[y] $\equiv$ P. For any chain to Q let $P \rightarrow^* Q$ be the simple-chain suffix. As every procedure on $P \rightarrow^* Q$ is a strict descendent of P in the nesting tree, application of the incarnation lemma demonstrates that P and Q

address the same incarnations of x and y. Thus the alias between these variables remain in effect until Q is reached. ■

The local lemma implies that the procedure at which x is aliased to f on some chain is not important; one need only concern himself with whether x is aliased to f at LPROC[f]. To this end, let MAYEQ[x] = {f| x ε ADDRESS[LPROC[f]] and x is aliased to f on some chain to LPROC[f]}. Once one has computed MAYEQ[x] its symmetric counterpart, call it $\overline{\text{MAYEQ}}$[x], is easily found with the equation - -

$$\overline{\text{MAYEQ}}[x] = \text{MAYEQ}[x] \cup \{y | x \in \text{MAYEQ}[y]\} \qquad [1]$$

Now, given the $\overline{\text{MAYEQ}}$ sets, the ALIAS-effect is solved by applying the equation - -

$$\text{MAYAFF}[Q] = \bigcup_{x \in \text{MAYAFF}_*[Q]} (\overline{\text{MAYEQ}}[x] \cap \text{ADDRESS}[Q]) \qquad [2a]$$

for procedures and the equation - -

$$\text{MAYAFF}[Q \to R] = \bigcup_{x \in \text{MAYAFF}_*[Q \to R]} (\overline{\text{MAYEQ}}[x] \cap \text{ADDRESS}[Q]) \qquad [2b]$$

for invocations. The sets MAYAFF_* are the summary relation sets computed in the propagation phase of the inter-procedural analysis. The validity of these formulas is a simple consequence of the local lemma. As $x \in \overline{\text{MAYEQ}}[x]$ it should be clear that $\text{MAYAFF}_* \subseteq \text{MAYAFF}$. In order to guarantee that the summary relations computed above are the final answer, it must be shown that the ALIAS effect does not interact with the FORMAL and SCOPE effects. The proof of this fact is delayed until the end of this section, when a better characterization of the ALIAS effect has been given.

The remaining problem is to characterize and compute the MAYEQ

sets. For the moment consider the consequence of alias concatenation in isolation; alias transitivity will be dealt with subsequently. It is clear that x is aliased to x on some chain to LPROC[x], i.e., $x \in MAYEQ[x]\ \forall x$. One can inductively characterize concatenations as follows - -

Alias Concatenation Lemma: $y \in MAYEQ[x]$ by alias concatenation if and only if $x \in SCOPE[LPROC[y]]$ and $\exists z(P \xrightarrow{<..z..>} R(..y..)$ and $(x \in MAYEQ[z]$ or $z \in MAYEQ[x]))$

Proof: ($\Rightarrow$) $y \in MAYEQ[x]$ by alias concatenation $\Leftrightarrow$ $x \in ADDRESS[LPROC[y]]$ and $\exists z\ (P \xrightarrow{<..z..>} R(..y..)$ and x is aliased to z along some chain to P). If $LPROC[x] = LPROC[y] \equiv R$ then the incarnation of x that is aliased to z is not the one referred to after the chain reaches R. So $x \in ADDRESS[LPROC[y]]$ and $LPROC[x] \neq LPROC[y]$ $\Rightarrow$ $x \in SCOPE[LPROC[y]]$. $x \in SCOPE[R] \Rightarrow LPROC[x] \in ANCESTOR[R]$ and $P \rightarrow R = FATHER[R] \in ANCESTOR[P]$ by the invocation lemma. Hence $x \in ADDRESS[P]$. $P \xrightarrow{<..z..>} R(..y..) \Rightarrow z \in ADDRESS[p]$. Thus either $x \in ADDRESS[LPROC[z]]$ or $z \in ADDRESS[LPROC[x]]$. But x is aliased to z along some chain $\Rightarrow$ $x \in MAYEQ[z]$ or $z \in MAYEQ[x]$.

($\Leftarrow$) $x \in SCOPE[LPROC[y]]$ and $P \xrightarrow{<..z..>} R(..y..)$ implies by the invocation lemma that $LPROC[x] \in ANCESTOR[P]$. Also $LPROC[z] \in ANCESTOR[P]$. Suppose $x \in MAYEQ[z] \Rightarrow$ there is a chain, $Q \rightarrow^* LPROC[x]$ to LPROC[x] along which x and z are aliased. By the chain lemma there is a simple chain from LPROC[x] to P. Clearly x is aliased to y on the chain $Q \rightarrow^* LPROC[x] \rightarrow^* P \xrightarrow{<..z..>} R(..y..)$. A similar chain can be constructed when $z \in MAYEQ[x]$. ∎

The alias transitivity effect is somewhat more involved. Its inductive characterization includes that in the concatenation characterization plus the starred condition in the lemma below - -

Alias Transitivity Lemma: $y \in MAYEQ[x]$ by alias transitivity if and only if $x \in SCOPE[LPROC[y]]$ and $\exists z(P \xrightarrow{<..z..>} R(..y..)$ and $(x \in MAYEQ[z]$ or $z \in MAYEQ[x]))$ or $LPROC[x] = LPROC[y]$ and $\exists z1,z2(P \xrightarrow{<..z1,z2..>} R(..y,x..)$ and $z1 \in MAYEQ[z2])$*

Proof: ($\Rightarrow$) $y \in MAYEQ[x]$ by alias transitivity $\Leftrightarrow$ $\exists a$ such that a is aliased to x and a is aliased to y along some chain and $x \in ADDRESS[LPROC[y]]$. Clearly one need only consider the case where $x \neq a \neq y$. Thus a is aliased to x = $\exists$ z1 (a is aliased to z1 on the chain and $P \xrightarrow{<..z1..>} R(..x..)$). Similarly $\exists$ z2 (a is aliased to z2 on the chain and $Q \xrightarrow{<..z2..>} S(..y..)$|.

First suppose $x \in SCOPE[S]$. Consider that part of the chain following the edge $P \rightarrow R$. The edge $Q \rightarrow S$ must be part of this suffix as otherwise $x \in SCOPE[S]$ implies the chain must loop through S again before x and y are both addressable again, but at this point a different incarnation y is being referred to. This suffix must also not pass through R again, as then a different incarnation of x is referred to. If a is aliased to z2 when R is reached then z2 is aliased to x by transitivity. z2 must be addressable at R as otherwise a new incarnation is created before reaching Q. Thus $x \in MAYEQ[z2]$. Otherwise

suppose that the alias between x and z2 is established at LPROC[z2] somewhere between R and Q. $x \in SCOPE[S]$ and the chain between R and Q does not loop through $R \Rightarrow x \in ADDRESS(LPROC[z2])$. Thus $z2 \in MAYEQ[x]$ as it is aliased to a and so is x at LPROC[z2].

Finally, suppose R = S. If $P \xrightarrow{<..z1..>} R(..x..) \neq Q \xrightarrow{<..z2..>} S(..y..)$ then on any chain different incarnations of x and y are being referred to. Clearly $z1, z2 \in ADDRESS[P \equiv Q]$. WLOG suppose $z2 \in ADDRESS[LPROC[z1]]$. Then by alias transitivity $z1 \in MAYEQ[z2]$.

($\Leftarrow$) The reverse direction is a simple extension of the alias concatenation lemma ■

Taking the two proceeding lemmas together one has a characterization of the MAYEQ sets.

Alias Lemma: $y \in MAYEQ[x]$ if and only if $x = y$

or $x \in SCOPE[LPROC[y]]$ and $\exists z (P \xrightarrow{<..z..>} R(..y..)$ and $(x \in MAYEQ[z]$ or $z \in MAYEQ[x]))$

or $LPROC[x] = LPROC[y]$ and $\exists z1, z2 (P \xrightarrow{<..z1,z2..>} R(..x,y..)$ and $z1 \in MAYEQ[z2])$

Every member of the MAYEQ relationship may be found iteratively by applying the alias lemma. MAYEQ is reflexive is equivalent to stating $x \in MAYEQ[x]$ for all x. All of these reflexive elements are put on a queue. New elements of the relation are found by examining pairs still on the queue. These new members are put on the queue and processed in turn. Given a relational element $y \in MAYEQ[x]$ on the queue one may infer from it by using the alias lemma that

a/ There is an invocation $P \xrightarrow{<..x..>} LPROC[a](..a..)$ and

$y \in SCOPE[LPROC[a]]$

$\Rightarrow a \in MAYEQ[y]$

and b/ There is an invocation $P \xrightarrow{<..y..>} LPROC[a](..a..)$ and

$x \in SCOPE[LPROC[a]]$

$\Rightarrow a \in MAYEQ[x]$

and c/ There is an invocation $P \xrightarrow{<..x,y..>} LPROC[a](..a,b..)$

$\Rightarrow a \in MAYEQ[b]$ and $b \in MAYEQ[a]$

The process stops when the queue becomes empty. It remains to show that every element of the MAYEQ relation is uncovered by this iterative process. The alias lemma states in the reverse direction that $y \in MAYEQ[x]$ implies there is another pair $a \in MAYEQ[b]$ such that

d/ $b = x$ and $x \in SCOPE[LPROC[y]]$ and there is an invocation $P \xrightarrow{<..a..>} R(..y..)$

or e/ $a = x$ and $x \in SCOPE[LPROC[y]]$ and there is an invocation $P \xrightarrow{<..b..>} R(..y..)$

or f/ $LPROC[x] = LPROC[y]$ and there is an invocation $P \xrightarrow{<..a,b..>} R(..x,y..)$

Note that if $a \in MAYEQ[b]$ then $y \in MAYEQ[x]$ will be inferred by one of the conditions a through c. The chain establishing the alias between a and b is shorter than the chain establishing the alias between x and y. Hence by repeatedly applying d through e, one eventually must reach a reflexive element $c \in MAYEQ[c]$. But this element is initially in the relation and the conditions a through c will thus eventually discover $y \in MAYEQ[x]$.

In order to quickly find invocations satisfying conditions a, b, or c a number of structures are used. For a procedure P, SVEC[P] is the set of invocations calling procedures which are strict descendants

of P in the nesting tree. For a variable x, EQUIV[x] is the set of invocations in which x occurs as an actual parameter. For a variable x and an invocation P→R , ATOF[x,P→R] is a list of the formals to which x is passed on P→R.

```
type INVOKEVEC:  bit vector of INVOKE
declare SVEC:  array [PROC] of INVOKEVEC
declare EQUIV:  array  [VAR] of INVOKEVEC
declare ATOF:  array [VAR,INVOKE] of list of VAR
declare AFTEMP:  array [VAR] of INVOKEVEC
declare QUEUE:  queue of struct  (xc:VAR, yc:VAR)
declare x, y, a, b : VAR, f: PARM, e: INVOKE

proc POSTR(p:PROC)          #This routine recursively computes SVEC#
   declare q : PROC
1.    for q ε SON[p] do
2.       POSTR(q)
3.       SVEC[p] ← SVEC[p] ∪ SVEC[q]
4.       for e ε CALL[q] do
5.          SVEC[p] ← SVEC[p] + e
6.    return
end POSTR

proc UPDATE(x,y : VAR)    #This is a utility routine which updates
                                        MAYEQ, and QUEUE#.
1.    if y ∉ MAYEQ[x] then
2.       MAYEQ[x] ← MAYEQ[x] + y
3.       QUEUE ← QUEUE + <xc : x, yc : y>
4.    return
end UPDATE
```

```
1.  QUEUE ← ∅                    #Initialize QUEUE, and MAYEQ.  Get ready
2.  for x ε VAR do                 to compute ATOF and EQUIV#
3.     AFTEMP[x] ← ∅
4.     QUEUE←QUEUE+ <xc : x, yc : x>
5.     EQUIV[x] ← ∅
6.     MAYEQ[x] ← {x}
7.  for q ε PROC do            #Compute EQUIV and ATOF.  Get ready to compute
8.     for x ε FORMLST[q] do        SVEC#
9.        for f ε ACTUAL[x] do
10.          e ← AINV[f]
11.          a ← AVAR[f]
12.          EQUIV[a] ← [a] + e
13.          if e ∉ AFTEMP[a] then
14.             AFTEMP[a] ← AFTEMP[a]+e
15.             ATOF[a,e] ← ∅
16.          ATOF[a,e] ← ATOF[a,e]+x
17.    SVEC[q] ← ∅
18. POSTR(ROOT)                    #Compute SVEC#
19. while QUEUE ≠ ∅ do              #Iteratively find members of MAYEQ#
20.    pick <xc : x, yc :y> from QUEUE arbitrarily
21.    for e ε EQUIV[x] ∩ SVEC[LPROC[y]] do  #Apply condition a#
22.       for a ε ATOF[x,e] do
23.          UPDATE(y,a)
24.    for e ε EQUIV[y] ∩ SVEC[LPROC[x]] do   #Apply condition b#
25.       for a ε ATOF[y,e] do
26.          UPDATE(x,a)
27.    for e ε EQUIV[x] ∩ EQUIV[y] do         #Apply condition c#
28.       for a ε ATOF[x,e] do
29.          for b ε ATOF[y,e] do
30.             UPDATE(a,b)
31.             UPDATE(b,a)
```

In lines 1 through 18 the structures SVEC, EQUIV, and ATOF are computed and the queue QUEUE is initialized. Lines 19 through 31 constitute an iterative application of the conditions a, b, and c discussed above. The worst case timing analysis is easy except for the iterative loop. Lines 1 through 6 clearly operate in time O(VAR). Lines 7 through 17 process each variable in every tuple labeling an invocation for a cost of O(PARM). The routine POSTR called in line 18 traverses each procedure and the invocations of it in a depth first search of the nesting tree for a time of O(PROC+INVOKE). Let $EQ_{MAY} = \sum_{x \in VAR} |MAYEQ[x]|$. The loop in lines 19 through 31 is executed EQ_{MAY} times as a member of the MAYEQ relation is put on QUEUE only once. The total time in this loop is proportional to the number of times UPDATE is called. Suppose there is a call UPDATE (x,y), that is, it is discovered that y ε MAYEQ[x] by a chain passing through some edge e. Suppose that $e \equiv p \xrightarrow{<..a..>} LPROC[y](..y..)$. By examining the predicates d through f, one realizes that there is only one element of the MAYEQ relation from which y ε MAYEQ[x] through e can be inferred unless it is condition f in which case there are at most two. Hence UPDATE is called with (x,y) for an edge e at most twice. Thus UPDATE is called at most

$\sum_{(x,y) \in EQ_{MAY}} |CALL[LPROC[y]]|$ times where $|CALL[LPROC[y]]|$ is the number of invocations of LPROC[y]. Let $MAX_{INVOKE} = \max_{P \in PROC} |CALL[P]|$. Then clearly the cost of the loop is $O(EQ_{MAY} * MAY_{INVOKE} * \log INVOKE)$ where the log term is the cost of finding the set bits in lines 21, 24, and 27. So the algorithms total running time is $O(PROC+PARM+EQ_{MAY} * MAY_{INVOKE} * \log INVOKE+VAR)$. For any reasonable program MAX_{INVOKE} will tend to remain constant, hence

the iterative loop will take an amount of time proportional to EQ_{MAY}.

Now that the MAYEQ sets are computed, the sets $\overline{MAYEQ}$ can be found in time $O(EQ_{MAY})$ by applying formula [1]. The final MAYAFF sets can be found by applying formulas [2a] and [2b]. To do so requires time $O(MAY_{PROC}+MAY_{INVOKE})$. It just remains to show that [2a] and [2b] give the final answer by demonstrating that the ALIAS effect does not interact with the FORMAL and SCOPE effects. For example, it might be that a bit included in a summary relation as a consequence of the ALIAS effect, subsequently requires propogation due to a FORMAL or SCOPE effect. It is shown that if such an instance arises, the ALIAS-effect will also broaden the relation to include such a propogated bit. Thus the FORMAL and SCOPE effects need never be reconsidered.

Independence Lemma: Suppose $x \in MAYAFF[R]$ as the result of an ALIAS-effect.

1. If $x \in MAYAFF[P\rightarrow R]$ because of a SCOPE effect then

   $x \in MAYAFF[P\rightarrow R]$ because of an ALIAS-effect.

2. If $y \in MAYAFF[P\rightarrow R]$ because of a FORMAL-effect with x then

   $y \in MAYAFF[P\rightarrow R]$ because of an ALIAS effect.

Proof: $x \in MAYAFF[R]$ as the result of a VAR-effect $\Leftrightarrow \exists$ f such that $f \in MAYAFF[R]$ and $x \in \overline{MAYEQ}[f] \cap ADDRESS[R]$.

Suppose $x \in MAYAFF[P\rightarrow R]$ by a SCOPE effect $\Rightarrow x \in SCOPE(R)$ $\Rightarrow x \in ADDRESS[P]$. If $f \in SCOPE[R]$ then $f \in MAYAFF_{*}[P\rightarrow R]$ by the SCOPE effect. Hence $x \in \overline{MAYEQ}[f] \cap ADDRESS[P] \Rightarrow$ $x \in MAYAFF[P\rightarrow R]$ by an ALIAS-effect. Otherwise LPROC[f] = R. It must be that $f \in MAYEQ[x] \Rightarrow$ f is a formal parameter.

Suppose $p \xrightarrow{<..a..>} R(..f..)$. It may be that x is not aliased to a, i.e., x is not aliased to f through this invocation. If so then x should not be included in MAYAFF[P→R]. But if $x \in \overline{MAYEQ}[a]$ then $x \in MAYAFF[P \to R]$ by the ALIAS effect as $a \in MAYAFF_*[P \to R]$ by the FORMAL-effect.

Suppose $y \in MAYAFF[P \to R]$ by a FORMAL effect, i.e., $p \xrightarrow{<..y..>} R(..x..)$. LPROC[x] = R and $f \in ADDRESS[R] = x \in MAYEQ[f]$. First suppose that $f \in SCOPE[R]$. It may be that f is not aliased to y in which case one must again conclude that y should not be in MAYAFF[P→R]. However, if $y \in \overline{MAYEQ}[f]$ then $y \in MAYAFF[P \to R]$ by the ALIAS-effect as $f \in MAYAFF_*[P \to R]$ by the SCOPE effect. Finally, suppose that LPROC[f] = R and assume $p \xrightarrow{<..y,z..>} R(..x,f..)$. Again, if y is not aliased to z then one should not conclude that $y \in MAYAFF[P \to R]$. Otherwise $y \in MAYAFF[P \to R]$ by the ALIAS effect as $z \in MAYAFF_*[P \to R]$ by the FORMAL-effect and $y \in \overline{MAYEQ}[z]$. ∎

CONCLUSION

Putting all the sub-algorithms together, one has a O(PROC+INVOKE+PARM+VAR+MAY_{INVOKE}+MAY_{PROC}* log VAR+EQ_{MAY}*MAX_{INVOKE}*logINVOKE) algorithm for computing MAY summary information for a program. Let I_{SIZE} = PROC+PARM+INVOKE+VAR and O_{SIZE} = MAY_{INVOKE}+MAY_{PROC}+EQ_{MAY}. The logarithmic terms arise from the necessity of listing the elements of a bit vector. In many bit vector systems it is assumed that this is a constant time operation. If one were to assume this and that MAX_{INVOKE} is bounded by some constant, then one could conclude that this algorithm runs in time $O(I_{SIZE}+O_{SIZE})$.

There remains the problem of computing MUST summary information and of extending live and avail to include inter-procedural effects. These problems are intractable in theory (unless P= NP). This intractability is due to the fact that at any procedure there can be an exponential number of aliasing patterns. In most realistic programs, however, the number of these patterns is small. In a subsequent paper, the problems will be proven (co-) NP-complete and an algorithm will be presented which operates quickly for "reasonable" programs.

References:

[AC] Allen, F.E. and Cocke, J. "A Program Data Flow Analysis Procedure." Comm. ACM 19, 3 (1976), 137-146.

[A] Allen, F.E. "Interprocedural Data Flow Analysis." Information Processing 74, North-Holland Pub. Co., Amsterdam (1974), 398-402.

[S] Spillman, T.C. "Exposing Side-Effects in a PL/1 Optimizing Compiler." Information Processing, North-Holland Pub. Co., Amsterdam (1971), 376-381.

[UK] Ullman, J.D. and Kam, J.B. "Global Data Flow Analysis and Iterative Algorithms." J. ACM 23, 1 (1976), 158-171.

[UH] Ullman, J.D. and Hecht, M.S. "A Simple Algorithm for Global Data Flow Analysis Problems." SIAM J. Computing 4, 4 (1975), 519-532.

[FO] Fosdick, L.D. and Osterweil, L.J. "Data Flow Analysis in Software Reliability." ACM Computing Surveys 8, 3 (1976), 305-330.

[Ban] Banning, J.P. "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables." Conf. Rec. Seventh ACM Symp. Principles of Programming Languages, (1980), 29-41.

[Bar] Barth, J.M. "A Practical Interprocedural Data Flow Analysis Algorithm." Comm. ACM 21, 9 (1978), 724-736.

[R1] Rosen, B.K. "High Level Data Flow Analysis, Pt.1 (Classical Structured Programming)." Res. Rep. RC5598, IBM T.J. Watson Res. Ct., Yorktown Heights, N.Y., Aug. 1975.

[R2] Rosen, B.K. "High Level Data Flow Analysis, Pt.2(Escapes and Jumps)." Res. Rep. RC5744, IBM T.J. Watson Res. Ct., Yorktown Heights, N.Y., Dec. 1975.

[R3] Rosen, B.K. "Data Flow Analysis for Procedural Languages." Res. Rep. RC5948, IBM T.J. Watson Res. Ct., Yorktown Heights, N.Y., April 1976.

[GW] Graham, S.L. and Wegman, M. "A Fast and Usually Linear Algorithm for Global Flow Analysis." J. ACM 23, 1 (1976), 172-202.

[AHU] Aho, A.V., Ullman, J.D. and Hopcroft, J.E. The Design and Analysis of Computer Algorithms. Addison Wesley (1971).

[GJ] Garey, M.R. and Johnson, D.S. Computers and Intractability -- A Guide to the Theory of NP-Completeness. W.H. Freeman and Co. (1978).