A PRECISE INTER-PROCEDURAL DATA FLOW ALGORITHM*

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

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 aliasing are introduced, the data flow problems become much more difficult. The avail, live, and must-summary data flow problems are shown to be NP-complete in the presence of aliasing. However, an algorithm is presented with O(SET*EDGE) time performance where EDGE is the size of the program's flow graph and SET is a possible exponential number which reflects the number of aliasing patterns of the program. It is argued that in practice SET is small and on the order of the number of variables of the program.

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. The available expression and live variable problems are of this genre. The algorithms constructed for such analyses [AC,GW,UH,UK] are very 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 efforts have focused on determining these patterns in the presence of the inter-procedural effects induced by procedure calls. Once the scope of the problem was so enlarged, an additional class of problems arose. Specifically, one could then ask what effect the execution of an entire procedure had on the variables of a program. Such problems were termed the must - and/or may - summary problems [Barth].

Current inter-procedural algorithms either compute approximations to the answers which are precise up to symbolic execution [Barth], or are very inefficient [R1,R2], or ignore one or more of the "difficult" effects such as recursion or aliasing [S,A]. The one notable exception to this is a fast and precise algorithm for the may-summary problem [Ban,M]. This paper focuses on the remaining problems — live, avail, and must-summary.

The inter-procedural live problem is shown to be NP-complete [GJ]. The avail and must-summary

*This work was supported in part by NSF grants MCS77-02194 and MCS-8000017

problems are co-NP-complete. These problems are intractable due to the presence of aliasing. However, the expected degree of exponentiality is felt to be small, especially when certain redundancies are eliminated. Iterative solutions to these problems are presented which are then related to the theory of monotone data flow analysis frameworks [Kildall]. An efficient algorithm based on these iterative solutions is then sketched which is superior to the one obtained by employing Kildall's general algorithm. The algorithm runs in time O(SET*EDGE) where EDGE is the size of the program's flow graph. SET is the possibly exponential parameter which reflects the number of aliasing patterns of the program. A pruning scheme is introduced which reduces the number of alias patterns to the extent that SET is expected to be asymptotically small (on the order of the number of variables in the program) for realistic programs. Thus the algorithm should be efficient in practical applications.

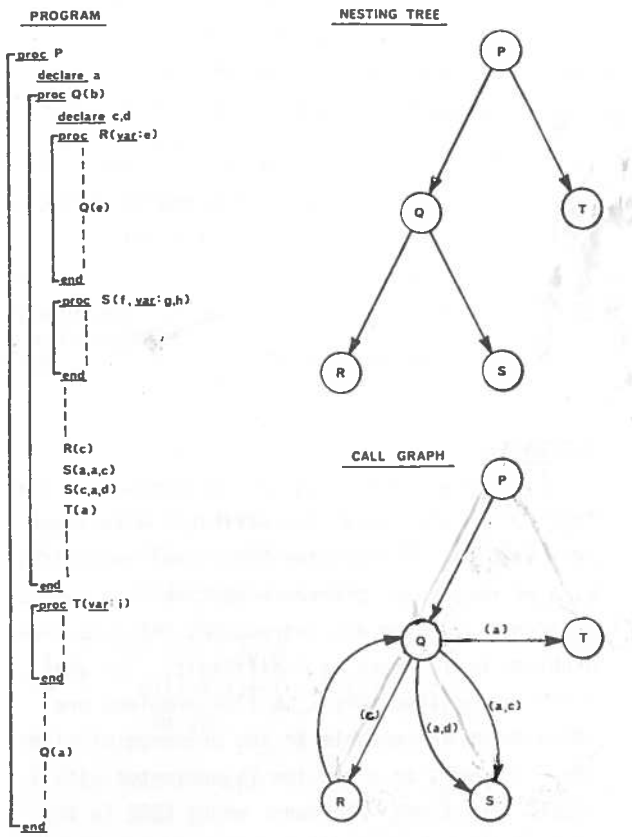

FIGURE 1: INTER-PROCEDURAL STRUCTURES

PROGRAM MODELS:

This paper investigates data flow problems in a typical block structured programming language such as PASCAL. The salient inter-procedural characteristics of such a language are the nesting of procedures, pass-by-reference parameters, and recursion. The analytic model employed in this paper is sketched below.

It is assumed that procedure definitions may be nested. This nesting is modelled by a directed rooted tree whose root, MAIN, is the main procedure. A procedure P is the father of a procedure Q if the definition of Q is directly declared within P. Figure 1 should make the construction of this nesting tree clear.

Within the definition of a procedure, P, a number of local variables, LOCAL[P], may be declared. This set includes the formal parameters of P. Those formal parameters which are passed-by-reference are termed reference parameters of P and are denoted by FORMAL[P]. A variable is global to P if it is the local variable of any proper ancestor of P in the nesting tree. The code part of P can address (manipulate) only its local and global variables. Formally --

GLOBAL[P] = $\cup$ (LOCAL[Q] || Q is a proper ancestor of P)

ADDRESS[P] = LOCAL[P] $\cup$ GLOBAL[P].

The code part of a procedure can also contain statements, called call-sites, which call (invoke) other procedures. One writes $P \xrightarrow{(a_1,\ldots,a_k)} R\ (f_1,\ldots,f_k)$ if there is a call-site in P calling R with actual parameters a_1 through a_k where each

actual a_i is passed-by-reference to the reference parameter f_i. Only reference parameteres are of interest as these are the only variables for which aliases are dynamically established. The call structure of a program is modelled by a directed multigraph, called a call graph, in which each edge corresponds to a call-site. Each edge is labelled with the tuple of actual parameters used at its call-site. The call graph may contain loops as recursion is permitted. Figure 1 illustrates the construction.

A call graph is assumed to satisfy the following requirements.

(1) Every procedure is reachable, i.e., there is a path from MAIN to every procedure.

(2) If P → Q (P calls Q) then Q is addressable by P, i.e., the father of Q is an ancestor of P in the nesting tree.

A simple consequence of (2) is that P → Q implies GLOBAL[Q] ⊆ ADDRESS[P]. A path in the call graph is called a call chain and represents a sequence of calls made in an execution of the program.

A program's execution involves two dynamic effects. Whenever a call is executed, a new activation of the called procedure R, is created. Each activation causes a new incarnation (instance) of each variable in LOCAL[R]-FORMAL[R] to be created. Recursion permits the simulaneous existence of many incarnations of the same variable -- a variable refers to its most recent incarnation. The following lemma determines when a variable addressed by two different procedures refers to the same incarnation.

Incarnation Lemma: Suppose $Q_0 \rightarrow Q_1 \rightarrow \ldots \rightarrow Q_n$ is a call chain. A variable x refers to the same incarnation in both Q_0 and Q_n iff $x \in \cap\,(\text{GLOBAL}[Q_i] \mid\mid i \geq 1)$.

This effect is called the SCOPE effect.

The other dynamic effect, called the ALIAS effect, is the aliasing of reference parameters to their actual parameters during the execution of an invocation. When $P \xrightarrow{(a_1,..,a_k)} R\ (f_1,\ldots,f_k)$ each f_i is aliased to a_i; that is, f_i is made to refer to the same incarnation that a_i referred to in P. In general, two variables x and y are aliased along some call chain p, written $x \overset{p}{<\!\!>} y$, if both x and y refer to the same incarnation after p is traversed. Aliasing is an equivalence relation along a specific call chain. However, different call chains establish different aliasing relations. The form of these relations will be discussed in the next section.

The data flow problems analyzed in this paper require that every procedure's internal control structure is represented. Each procedure is modelled as a flow graph [Karp] with unique exit and entry vertices. These flow graphs are linked interprocedurally by calls into a super graph as follows. Suppose E:P→R is an invocation of R from call-site vertex CSITE[E] in procedure P's flow graph. The vertex, CSITE[E], is split into two vertices, CPOINT[E] and RPOINT[E]. All in-coming edges to CSITE[E] are in-coming edges to CPOINT[E] and a call edge is added from CPOINT[E] to the entry vertex of R. All out-going edges from CSITE[E] are out-going edges from RPOINT[E] and a return edge is added from the exit vertex of procedure R to RPOINT[E]. All other edges in the resulting super graph are called simple edges. For a call or return edge e, let INVOKE[e] be the invocation E in the call graph responsible for the creation of e. Figure 2 depicts the construction.

The super graph model was chosen as it satisfies

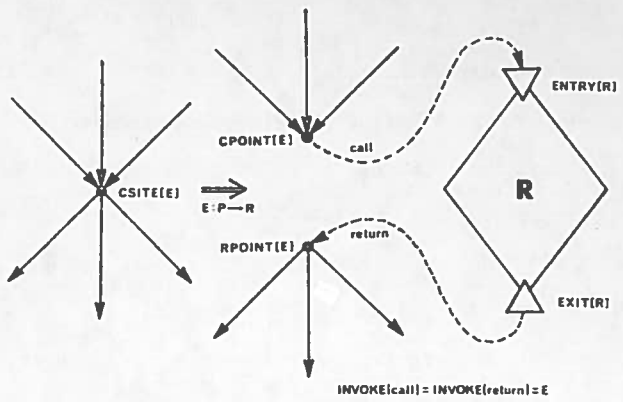

FIGURE 2: MODELLING CALLS IN THE SUPERGRAPH

the criterion that any execution sequence of the program can be modelled as a path in the super-graph. Note, however, that not every path is an execution sequence, despite the assumption of precision up to symbolic execution. The propagative algorithms in the following sections will have to insure that only execution paths are followed.

ALIASING SETS AND PARTITIONS

In this section the nature of the aliasing relation, $\overset{p}{<\!\!>}$, induced by a call chain p: MAIN → Q_1 → ... → Q_n is analyzed. As stated earlier $\overset{p}{<\!\!>}$ is an equivalence relation on the variables in ADDRESS[Q_n]. Let π_p be the partition of ADDRESS[Q_n] induced by $\overset{p}{<\!\!>}$. Each component or equivalence class of the alias partition π_p is termed an alias set of Q_n. By definition the variables in each alias set all refer to the same incarnation.

Let X be a set of variables and E denote the invocation $Q_n \xrightarrow{(a_1,..,a_k)} Q_{n+1}(f_1,..,f_k)$. Define the incarnation propagation function, f_I, as follows -- $f_I(X,E) = (X \cap GLOBAL[Q_{n+1}]) \cup \{ f_i \mid\mid a_i \in X \}$. Note that f_I is monotonic and distributive in the argument X. The utility of f_I is seen in the following theorem.

Alias Theorem: $X \in \pi_p \Rightarrow f_I(X,E) \in \pi_{p\cdot E}$ or $f_I(X,E) = \emptyset$.

This theorem gives the inductive basis needed to compute π_p for arbitrary p. Given π_p and an invocation E from procedure Q_n to Q_{n+1}, one can infer that

$$\pi_{p\cdot E} = \{f_I(X,E) \mid\mid X \in \pi_p \text{ and } f_I(X,E) \neq \emptyset\} \cup \{ \{x\} \mid\mid x \in LOCAL[Q_{n+1}] - FORMAL[Q_{n+1}]\}.$$

Figure 3 illustrates the construction of several alias partititons for a simple program.

It is sufficient for algorithmic purposes to just know all the possible alias sets, $\Sigma(Q)$, for each procedure Q. Formally, $\Sigma(Q) = \{X \mid\mid \exists$ call chain p from MAIN to Q such that $X \in \pi_p\}$. All the alias sets can be found by initially stipulating that every singleton set {x} such that $x \in LOCAL[Q] - FORMAL[Q]$ for some procedure Q is an alias set of Q and then repeatedly applying f_I to this basis and its offspring whenever possible until no new alias sets are generated. A work-list algorithm for this computation requires time O(SET*INVOKE*(SET+MAXF)) where SET is the number of alias sets, INVOKE is the number of call-sites, and MAXF = $\max_p(|FORMAL[P]|)$. The O(SET+MAXF) term comes from the cost of operating a hash table and computing f_I. This term has O(1) expected behavior. Thus the total expected time is O(SET*INVOKE). Note that SET is potentially as large as $O(2^{VAR})$ where VAR is the number of variables in the program.

INTER-PROCEDURAL DATA FLOW PROBLEMS

The framework for the inter-procedural live and avail data flow problems is obtained by annotating each vertex, v, in the super-graph with two sets, GEN[v] and KILL[v], of tokens. For simplicity it will henceforth be assumed that the tokens are the variables of the program, although one should realize that the tokens could be the set of expressions in the program or some other program entity set. A

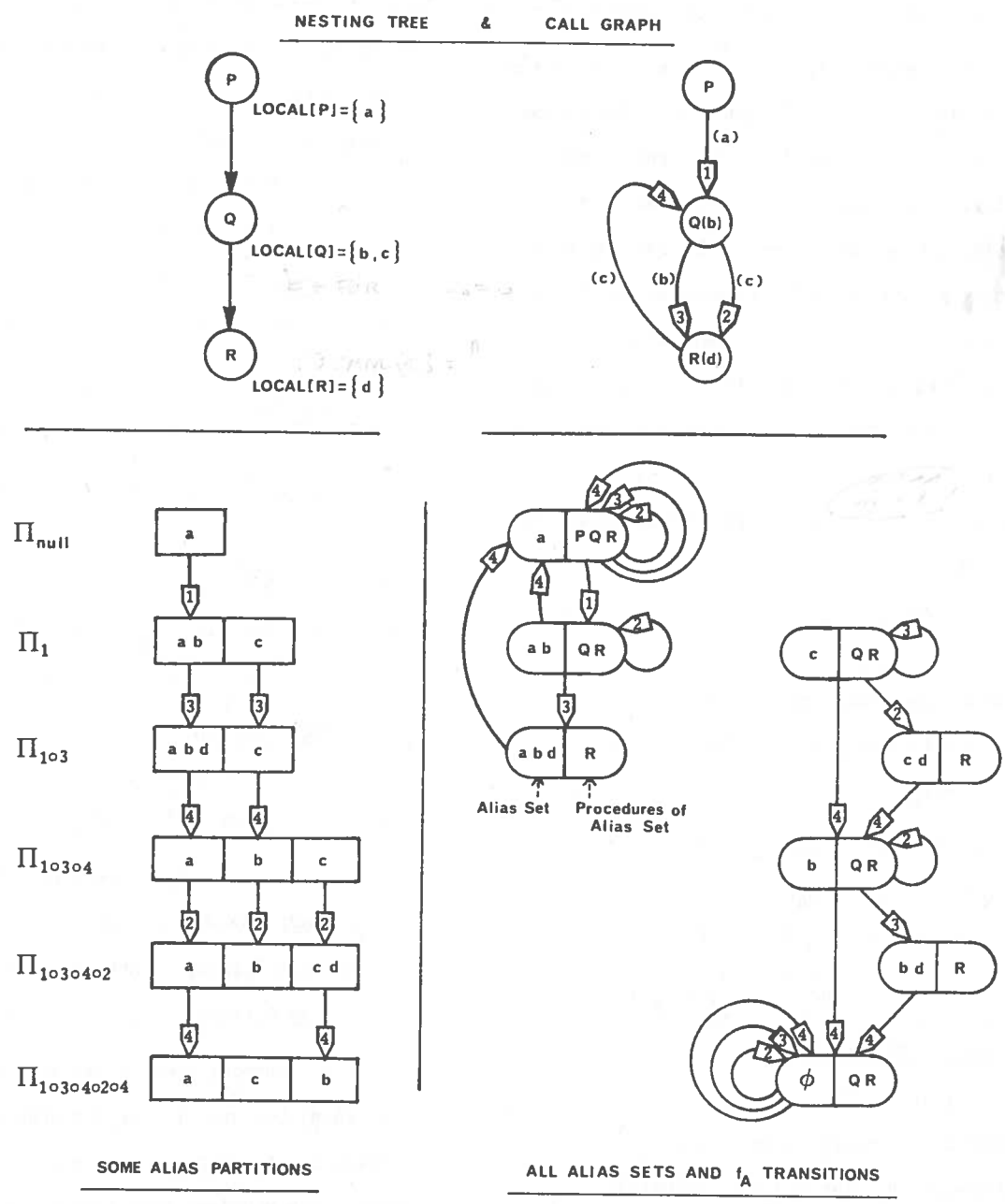

FIGURE 3: ALIAS SETS AND PARTITIONS

variable x is said to be generated at vertex v when x ε GEN[v], and x is killed at v when x ε KILL[v].

A rigorous formulation of these data flow problems involve some subtleties not found in their intra-procedural counterparts [Ken, Cocke]. First, a variable x can be live or avail at a vertex v in procedure P's flow graph, only when x ε ADDRESS[P]. As stated earlier not every path in the super graph represents an execution sequence. The second point then is that the path(s) in question must be an execution path(s).

The most complicating factor is the existence of aliases. When one asks if a variable x is live at a vertex v, one is really asking a question about the incarnation to which x refers at v. Thus the definition of live must involve the entire

execution path in question, including the part from the entry vertex of MAIN to the vertex v. The incarnation to which x refers is captured by the following notation. For an execution path p and variable x, let $S_A(x,p)$ be the alias set of the alias partition established by p, which refers to the same incarnation that x most recently referred to in the execution of p. Such an alias set may not exist and in this case let $S_A(x,p) = \emptyset$. The notation $S_A(x,p) \equiv S_A(x,q)$ asserts that the two alias sets refer to the same incarnation.

A formal definition of the live and avail data flow problems is now possible.

INTER-PROCEDURAL LIVE:

x is live at v iff

$x \in$ ADDRESS[procedure containing v]

and $\exists$ execution path p:ENTRY[MAIN] $\rightarrow \ldots \rightarrow v_0 (=v) \rightarrow v_1 \rightarrow \ldots \rightarrow v_n$

such that $S_A(x,p_n^*) \cap GEN[v_n] \neq \emptyset$

and $S_A(x,p_o) \equiv S_A(x,p_n)$

and $\forall\, 0 \leq j < n\ (S_A(x,p_j) \cap KILL[v_j] = \emptyset$

or $S_A(x,p_j) \not\equiv S_A(x,p_o)\ )$

INTER-PROCEDURAL AVAIL:

x is avail at v iff

$x \in$ ADDRESS[procedure containing v]

and $\forall$ execution paths $p: v_0 (=ENTRY[MAIN]) \rightarrow v_1 \rightarrow \ldots \rightarrow v_n (=v)$

$\exists\, i$ such that $S_A(x,p_i) \cap GEN[v_i] \neq \emptyset$

and $S_A(x,p_o) \equiv S_A(x,p_n)$

and $\forall i \leq j < n\ (S_A(x,p_j) \cap KILL[v_j] = \emptyset$

or $S_A(x,p_o) \not\equiv S_A(x,p_j)\ )$.

The must and may summary data flow problems are concerned with summarizing the effects of a procedure's execution. Each vertex, v, of the super-graph is annotated with a single set, AFFECT[v], of variables. A variable x is said to be affected by v whenever $x \in$ AFFECT[v]. The variable x may be affected by a procedure P if there is an execution path through P for which the appropriate incarnation of x is affected somewhere on this path. The variable x must be affected by P if x is affected on some vertex of every execution path through P. Formally - -

MAY SUMMARY PROBLEM:

x may be affected by P iff

$\exists$ execution path p : ENTRY[MAIN] $\rightarrow \ldots \rightarrow v_0 (=ENTRY[P] \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n (=EXIT[P])$

for which $S_A(x,p_o) \equiv S_A(x,p_n)$

and $\exists\, i$ such that $S_A(x,p_i) \cap AFFECT[v_i] \neq \emptyset$

and $S_A(x,p_o) \equiv S_A(x,p_i)$

MUST SUMMARY PROBLEM:

x must be affected by P iff

$\forall$ execution paths p : ENTRY[MAIN] $\rightarrow \ldots \rightarrow v_0 (=ENTRY[P]) \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n (=EXIT[P])$

for which $S_A(x,p_o) \equiv S_A(x,p_n)$

$\exists\, i$ such that $S_A(x,p_i) \cap AFFECT[v_i] \neq \emptyset$

and $S_A(x,p_o) \equiv S_A(x,p_i)$

The may-summary problem has an efficient solution [Ban] because it does not depend on the intra-procedural structure of a program. Banning has termed such problems flow insensitive. The remaining three problems -- live, avail, and must-summary -- are all flow sensitive and are the focus of the remainder of the paper.

COMPLEXITY OF THE PROBLEMS:

The theory of NP-complete problems is well established [GJ]. It is the consensus of the mathematical community that NP-complete problems do not have polynomial-time algorithms. The inter-

*Note: p_i is the prefix of p from ENTRY[MAIN] to v_i.

procedural live problem is shown to be NP-complete.

The complement of a problem is obtained by negating the problem statement. For example, the complement of avail, called co-avail, asks if a variable is not avail at a vertex. The co-avail and co-must problems are NP-complete. Hence, the avail and must-summary problems are said to be co-NP-complete [GJ].

To show that a problem is NP-complete it suffices to show that the problem is in NP and that a known NP-complete problem is polynomially transformable to it. The NP-complete problem used here is the classic 3-satisfiability problem [Cook]. One is given a set of variables X and a boolean expression E in 3-Conjunctive Normal Form on these variables. The problem is to determine whether E is satisfiable; that is, if there is a truth assignment to the variables in X for which E is true.

Theorem: The inter-procedural live data flow problem is NP-complete.

Proof: An arbitrary 3-satisfiability problem must be transformed into a live data flow problem. Suppose $X = \{x_1,\ldots,x_n\}$ and the expression $E = \prod_{i=1}^{k} (x_{i1}+x_{i2}+x_{i3})$ where $x_{ij} \in X \cup \overline{X}$. The data flow problem modelling E is depicted in Figure 4. It is asserted that E is satisfiable if and only if the variable T is live at the entry vertex of the main procedure P_0. □

Similar proofs show that the avail and must-summary problems are co-NP-complete. Note that in the proof above, recursion was not employed. The presence of just the ALIAS effect makes these problems NP-complete. Moreover, the problems are polynomial if the SCOPE effect is considered in isolation. The problems are intractable in the presence of aliasing as there may be an exponential number of alias sets. (Note that this is true for Figure 4).

THE ITERATIVE SOLUTION:

In this section an iterative solution to the live data flow problem is presented and related to the theory of monotone data flow analysis frameworks. The avail and must-summary problems have similar solutions. The iterative solution employs an alias set framework in which alias sets are propagated instead of variables and in which a propagated alias set retains enough information about its propagation path to guarantee that only execution paths are followed.

In the alias set framework, the vertices of the super graph are annotated with sets of alias sets, SGEN[v] and SKILL[v] where

$$SGEN[v] = \{X \,||\, X \in \Sigma(\text{procedure containing } v) \text{ and } X \cap GEN[v] \neq 0\}$$

$$SKILL[v] = \{X \,||\, X \in \Sigma(\text{procedure containing } v) \text{ and } X \cap KILL[v] \neq 0\}.$$

The entities that are propagated in this framework consist of ordered pairs, <X,E>, called alias pairs. X is an alias set and the component E, called the memory, is either an edge in the call graph or the special symbol '*'. Instead of finding the relation "live," the special relation "alias live" is desired and is defined as follows - -

ALIAS LIVE:

<X,E> ε SLIVE[v] (reads as "<X,E> is alias live at vertex v") iff

∃ execution path $p: ENTRY[MAIN] \rightarrow \ldots \rightarrow v_0 (=v) \rightarrow v_1 \rightarrow \ldots \rightarrow v_n$,

and variable x such that

$X = S_A(x,p_0)$

and x is live at v on path p

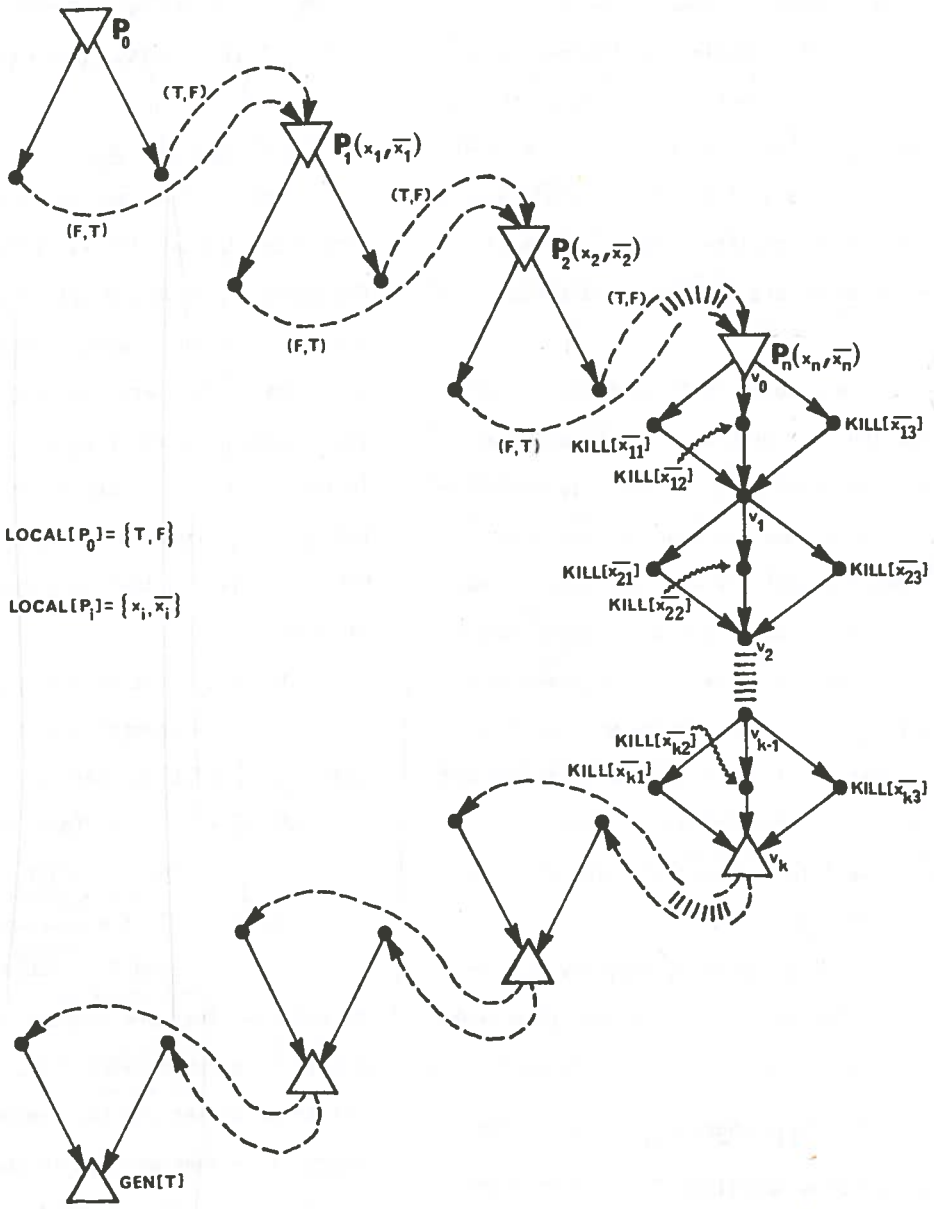

FIGURE 4: IS T LIVE AT ENTRY[P_0]?

and

$$E = \begin{cases} \text{INVOKE}[v_{k-1} \to v_k] \text{ for } k = \min_{1 \le j \le n} \{CE[p_{0j}] < RE[p_{0j}]\}^{\dagger} \\ \quad \underline{\text{if}}\ k \text{ exists} \\ * \quad \underline{\text{otherwise}} \end{cases}$$

†Note: $CE[p_{0j}]$ = the number of call edges on the subpath of p from v_0 to v_j

$RE[p_{0j}]$ = the number of return edges.

The introduction of the memory component is necessitated by the requirement that propagation is along execution paths only. From the definiton of E above, it follows that for any edge e from w into v, $e \circ p_{0j}$ is an execution path if and only if e is not a call edge or $E \in \{*, \text{INVOKE}[e]\}$. It should be clear that this observation provides the necessary leverage.

The relation between the alias live and live problems is expressed in the following simple theorem.

Theorem: $LIVE[v] = \cup (X \| \langle X,E\rangle \in SLIVE[v])$

The iterative solution employs a function I_L which maps a super graph vertex into a set of alias pairs and a function P_L which maps a set of alias pairs and a super graph edge into another set of alias pairs. The main theorem is --

Alias Live Theorem: $\langle X,E \in SLIVE[v]$ iff

$\exists$path $p: v_0 (=v) \rightarrow v_1 \rightarrow \ldots \rightarrow v_n$ such that $\langle X,E\rangle \in \overline{P}_L(I_L(v_n),p)$ †

The functions I_L and P_L are given in Figure 5 below.

$$I_L(v) = \{ \langle X,* \rangle \| X \in SGEN[v] \}$$

$P_L(Y,e)$ =

(1) For simple edge e:

$\{ \langle X,E\rangle \| X \notin SKILL[v]$ and $\langle X,E\rangle \in Y \}$

(2) For return edge e corresponding to call graph edge F:

$\{ \langle X,F\rangle \| X \notin SKILL[v]$ and $\exists \langle X',E\rangle \in Y (X = f_I(X',F) \neq \emptyset) \}$

(3) For call edge e corresponding to call graph edge F:

$\{ \langle X,E\rangle \| X \notin SKILL[v]$ and $((E = *$ and $\exists \langle X',*\rangle \in Y (X' = f_I(X,F)))$ or $(\langle X,E\rangle \in SLIVE[RPOINT[F]]$ and $(f_I(X,F)=\emptyset$ or $\exists \langle X',F\rangle \in Y (X' = f_I(X,F)))))\}$

FIGURE 5: PROPAGATION FUNCTIONS I_L AND P_L

The alias set framework for the live problem can be viewed as an instance of a distributive monotonic data flow analysis framework [Kildall]. The lattice consists of the set of all sets of alias pairs of the program with the meet operator being set union.

†$\overline{P}_L(X,p) = P_L(P_L(\ldots(P_L(X,v_{n-1}\rightarrow v_n)\ldots,v_1\rightarrow v_2),v_0\rightarrow v_1)$

The operation space consists of the functions $\overline{F}_p$ where for

$p = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_n$

$\overline{F}_p(X) = F_{v_0\rightarrow v_1}(F_{v_1\rightarrow v_2}(\ldots F_{v_{n-1}\rightarrow v_n}(X)\ldots))$

and $F_{v\rightarrow w}(X) = P_L(X,v\rightarrow w) \cup I_L(v)$.

The alias live theorem asserts that the solution to the alias live problem is the meet over all paths (MOP) solution. That is, $SLIVE[v] = \cup (\overline{F}_p(\emptyset) \| p$ is a path from v to EXIT[MAIN]). The distributive property of P_L implies that the MOP solution is also the minimum fixed point (MFP) solution to the set of equations --

$SLIVE[v] = \cup (F_{v\rightarrow w}(SLIVE[w]))$.

The MFP exists as the lattice is finite and P_L is a monotonic function.

AN EFFICIENT ALGORITHM:

The fact that the alias framework is a monotone data flow analysis framework allows one to employ Kildall's general algorithm in solving the live data flow problem. However, a better algorithm is presented.

The first improvement involves reducing the number of alias sets, SET. Let OCCUR[P] be the set of variables that occur in the code part of P. Let P_{REL} be a collection of sets of variables, REL[P], one for each procedure P in the program.

Definition: P_{REL} is a pruning system iff

(1) $\forall$ P $(REL[P] \supseteq OCCUR[P])$

and (2) $\forall$ invocations $P \rightarrow R$: $(f_I(REL[P],P\rightarrow R) \supseteq f_I(ADDRESS[P],P\rightarrow R) \cap REL[R])$.

The importance of pruning systems is that whenever X is an alias set in $\Sigma(P)$ one may use the pruned alias set $X \cap REL[P]$, in lieu of X as $X \cap REL[P] \supseteq X \cap OCCUR[P]$ and the relevant portion of X is retained under the mapping f_I (condition (2)).

In fact, if one lets $f_{REL}(X,P\to R) = f_I(X,P\to R) \cap REL[P]$, the monotonicity of f_I implies that $f_{REL}(X\cap REL[P],P\to R) = f_I(X,P\to R) \cap REL[R]$. Thus the mapping f_{REL} has the special property that a pruned alias set's image under f_{REL} is the pruned image of the alias set. Thus, as in the previous section, one may compute all the pruned alias set of a program directly using f_{REL}.

The optimal pruning system for a program consists of the sets

$$OPT[P] = OCCUR[P] \cup \{x \,|\, \exists \text{ call chain } p:Q_0\to\ldots\to Q_n \text{ such that } x \in OCCUR[Q_n] \text{ and } x \in \cap\,(GLOBAL[Q_i] \,|\, i \geq 1)\}.$$

The P_{OPT} pruning system is optimal in that it consists of the smallest possible sets. The OPT sets are the minimum fixed points of the system of equations --

$$OPT[P] = \cup\,(GLOBAL[R]\cap OPT[R] \,|\, P\to R) \cup OCCUR[P].$$

A round-robin [H] solution to this problem requires time $O(INVOKE^2)$ where INVOKE is the number of edges in the call graph.

The number of pruned alias sets for the P_{OPT} system is expected to be much smaller than the number of alias sets. In practice, the number SET is expected to be O(VAR) when the P_{OPT} system is employed. This expectation arises from the assumption that in practice OCCUR[P] rarely contains elements which are aliased.

The second improvement, stems from taking advantage of certain special properties of the propagation function P_L. Suppose an alias pair <X,E> is alias live at a vertex v. The alias pair is said to be <u>free</u> at v if E = * and <u>restricted</u> at v otherwise. The first observation is that all restricted alias pairs at vertices within a given procedure P were propagated from the exit vertex of P. This common origin for restricted alias pairs implies that the propagation of all restricted alias pairs with the same alias set can be accomplished by propagating just the alias set and an associated vector of memory components. This factorization reduces the problem to one of propagating the alias sets of each free and restricted alias pair. These representative alias sets are called the <u>factored alias pairs</u>.

Suppose there is a free and a restricted alias pair at a vertex v for which the alias sets are identical. From the properties of P_L it follows that wherever the restricted alias pair is subsequently found to be alias live, the free alias pair will also be alias live. Thus the second observation is that free alias pairs supercede restricted alias pairs whenever they meet at a vertex.

These observations lead to a significant performance increase as there are at most O(SET) factored alias pairs as opposed to the O(SET*INVOKE) collection of alias pairs. A work-list algorithm employing this propagative scheme runs in time

$$O(VERTEX+ANOTATE+INVOKE+\Sigma\ SET_P(EDGE_P+MAXS)) \text{ where}$$

VERTEX = the number of super-graph vertices.

ANOTATE = the sum of the cardinalities of every GEN and KILL set.

SET_p = the number of pruned alias sets of procedure P.

$EDGE_p$ = the number of super graph edges whose head vertices are in P's flow graph.

MAXS = the cardinality of the largest pruned alias set.

Note that the parameters SET_p, $EDGE_p$, and MAXS tend to remain constant (or grow quite slowly) as program size increases as procedure size tends to

remain constant.

A round-robin algorithm utilizing this approach has a time bound of O((R+2)*(SEDGE+SET*INVOKE)) where SEDGE is the number of simple edges in the super graph and R is the loop interconnectedness parameter of the super graph. However, this bound applies only when the super graph is reducible -- a requirement which may not hold very often. If the super graph is irreducible the R+2 term becomes EDGE. The round-robin approach is highly parallel in its operation and works well as long as there is not a small core of factored alias pairs which are reticent to stabilize. Empirical studies are needed to determine which of the two approaches -- round-robin or work-list -- is superior.

CONCLUSION:

The live, avail, and must summary data flow problems have been shown to be theoretically intractable in the inter-procedural context. Despite this, an iterative approach was developed to solve each problem which fit into the domain of monotone data flow analysis frameworks. A practical algorithm was then presented in which the degree of exponentiality was reduced to manageable levels. A work-list implementation of this algorithm was seen to have a worst case time bound of $O(VERTEX+ANOTATE+INVOKE+\Sigma_p SET_p*(EDGE_p+MAXS))$. Empirical studies are needed to determine the effectiveness of this approach. In particular, how large is SET_p in practice? How effective are the OPT pruning sets? How does performance vary with program size? Manual studies indicate a reasonable outcome.

The space requirement for the algorithm is O(PROC*VAR+SET*INVOKE+VERTEX+EDGE). The super graph is a large structure. It would be convenient to be able to arrange the solution so that it worked on a procedure at a time, thus removing the necessity of having the entire super graph in core at one time. Such an approach is possible, but it is not yet clear whether it is practical, due to the space-time tradeoff required. Another option for reducing the space requirement is to compute f_{OPT} and f_{OPT}^{-1} on the fly, thus removing the O(SET*INVOKE) tables. Again the space-time tradeoffs appear to be severe.

Another interesting consideration is the integration of this scheme into data flow problems including additional features such as parallelism and pointer data types.

ACKNOWLEDGMENT:

This author wishes to recognize the helpful collaboration of his colleagues -- Lee Osterweil, Lloyd Fosdick, and Dick Taylor -- at the University of Colorado.

REFERENCES:

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

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

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

[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.

[Cocke] Cocke, J. "Global Common Subexpression Elimination." SIGPLAN Notices 5(1970), 20-24.

[Cook] Cook, S.A. "The Complexity of Theorem Proving Procedures." Proc. 3rd Annual ACM Symposium on Theory of Computing (1971), 151-158.

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

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

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

[H] Hecht, M.S. Flow Analysis of Computer Programs. North-Holland Pub. Co., New York (1977).

[Karp] Karp, R.M. "A Note on the Application of Graph Theory to Digital Computer Programming." Information and Control, 3(1960), 179-190.

[Kildall] Kildall, G.A. "A Unified Approach to Global Program Optimization." Conf. Rec. First ACM Symp. Principles of Programming Languages (1973), 194-206.

[M] Myers, E.W. "A Precise and Efficient Algorithm for Determining Existential Summary Data Flow Information." Tech. Rep. CU-CS-175-80, Univ. of Colorado, Boulder, Colo., March 1980.

[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, New York, August 1975.

[R2] Rosen, B.K. "High Level Data Flow Analysis, Pt. 2(Escapes and Jumps)." Res. Rep. RC5744, IBM T.J. Watson Rec. Ct., Yorktown Heights, New York, April 1976.

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

[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.

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