Efficient Computation of Flow Insensitive Interprocedural Summary Information

Keith D. Cooper
Ken Kennedy

Department of Computer Science
Rice University
Houston, Texas

1. The Problem

To understand when it is safe to apply a given optimization, a compiler must have explicit knowledge about the impact of executing individual statements on the values of variables accessible to the statements. The impact of a statement is termed its *side effect*. This paper describes a method for annotating each statement s in a program with a set MOD(s) containing those variables whose values can be changed as a result of executing s. For statements which contain no procedure calls, the side effects can be determined by simple examination of the statement with some knowledge of the semantics of the source language.

The presence of procedure calls complicates the analysis of side effects in two distinct ways: inherited environmental aliasing patterns and cumulative side effects of procedure invocations. This paper focuses on the analysis of the cumulative side effects of procedure calls. The problem of analyzing aliasing patterns is complex but well understood. We show how to incorporate information about aliasing patterns into our analysis; algorithms for deriving such information may be found in [Bann] [Myer1] and [Coop].

If a statement s contains no procedure calls, the impact of procedure calls on its MOD set is limited to accounting for potential aliases present at the procedure invocation. If, however, a statement s contains one or more procedure calls, MOD(s) must account for the side effects of any statement in the called procedure which can be executed, including those which call yet other procedures. Because of this, MOD(s) summarizes the impact of all procedures directly or indirectly invoked from s. For this reason, such sets are called *summary information*. Other summary sets are discussed in the literature, notably USE and NOTPRE [Bart] [Bann] [Rose]. The modifications to our algorithms required to compute these other sets are straightforward.

Summary information can be divided into two categories, *flow insensitive* information and *flow sensitive* information. Flow insensitive information describes data flow events which occur on *at least one* path through a procedure. Because these sets contain facts about what may happen, they are often called *may* summary information. By contrast, flow sensitive information describes data flow events which occur on *every* path through a procedure. These sets contain facts which must be true on every execution of the statement, so they are also called *must* summary information. The algorithms given in this paper solve the flow insensitive version of the MOD problem.

Flow insensitive summary sets can be computed efficiently, because they are independent of the internal control flow structures of individual procedures. Once the analyzer discovers that an event occurs on a single control flow path through a procedure, it can conclude that the event is a side effect of the entire procedure. In contrast, a flow sensitive summary analysis can only include an event as a side effect by showing that the event occurs on every path through the procedure.

Myers has shown that the flow sensitive summary problems are co-NP complete in the presence of aliasing [Myer2]. Thus, these problems have no efficient solution, unless P=NP.

The computation of flow insensitive summary information has been discussed in the literature [Bart] [Bann] [Rose] [Myer2]. Previous treatments leave the impression that the computation is expensive relative to most of the analytical algorithms implemented in compilers. In particular, direct implementation of Barth's algorithm requires $\mathbf{O}(|edges|)$ time [Bart], while directly solving Banning's equations requires at least $\mathbf{O}(|nodes| \times |edges|)$ time [Coop]. This paper presents a set of algorithms for computing flow insensitive summary information in time nearly linear in the size of the program being analyzed. Where previous techniques have treated the computation as a single monolithic problem, our approach breaks the problem into a pair of subproblems, each of which has an efficient solution. The subproblem solutions can be combined to solve the original problem.

The techniques presented in this paper have been developed for implementation in a FORTRAN optimizing compiler imbedded in a programming environment [HoKe]. The programming environment provides the compiler with precisely the resources needed to compute and use interprocedural information. By retaining physical control of all program source, the environment provides the analyzer with easy access to any existing procedure. Because changes to the source must be made through the facilities of the environment, the compiler is assured of notification whenever a procedure is changed. This allows the compiler to determine which programs and procedures, if any, are impacted by the changes.

While the environment provides the analyzer with the support it needs, it also creates a need for efficient techniques for updating the MOD information in response to editing changes. We have developed techniques to update the information produced by our algorithms in response to editing changes.

Although this work is motivated by a programming environment for FORTRAN, we have tried to avoid language dependent assumptions. Throughout this paper, we discuss the problems in terms of the two level name scoping of FORTRAN; a later section of this paper shows that this is not a serious restriction. Programs are assumed to be recursive; this allows construction of concise programs which have pathological characteristics.

To further motivate the problem, consider the FORTRAN[1] program of figure 1. In compiling the main program, the compiler would like to know the impact of the call to S1 from MAIN on the value of the variables I and N. If the compiler knew that S1 did not modify either variable, it could replace the computation of RESULT with an assignment of the constant value 18, saving the multiplication.

Unfortunately, S1 modifies its first parameter, through a series of recursive calls. In fact, the MOD set for the statement containing the call to

```
      PROGRAM MAIN
      COMMON //,M,N
      INTEGER I,J,K,L,M,N,RESULT

      I = 6
      N = 3
      CALL S1(I, J, K, L)
      RESULT = N*I

      END

      SUBROUTINE S1(A, B, C, D)
      COMMON //,M,N
      INTEGER A,B,C,D,E

      M = 0
      D = 0
      E = 0

      CALL S1(E, A, B, C)

      END
```

Figure 1

[1]This example contains recursion, which is forbidden by all existing FORTRAN standards. This simply allows us to demonstrate points about the MOD computation in a concise manner.

S1 is {I,J,K,L,M}. Using this information, the compiler could replace the reference to N in the statement following the call with a constant 3, if this proved profitable.

2. Notation

We must introduce a moderate amount of notation to describe the components of programs. Because we are dealing with interprocedural phenomena, we need to discuss the program's call multi-graph, = (,). In , there is a node n ∈ representing each procedure and an edge e ∈ representing each call site. Whenever we refer to an edge e ∈ , it is assumed to be e = <u,v>, representing a call from a procedure u to a procedure v, unless otherwise stated.

The set of procedures which constitute a program is and the set of variables in a program is . The set of formal parameters of a procedure p is (p). For an edge e, the set of actual parameters of the call site is (e). We refer to the formal parameter in position i as (p), similarly for actual parameters (s). Finally, the set of variables which are global to a procedure p is (p).

A single statement can contain zero, one, or many procedure calls. To simplify equations dealing with the procedure calls in a single statement, we associate a set °(s) which contains the edges corresponding to the calls made from statement s. For a statement containing no procedure calls, °(s) is simply ∅.

3. Banning's framework

To provide a background understanding of the MOD computation, as well as establish terminology, it is useful to present an abbreviated treatment of Banning's framework for MOD. Banning posed the MOD computation as a data flow problem over the program's call multi-graph , allowing MOD information to be computed using any of a number of techniques developed for global data flow analysis.

Rather than computing MOD directly, Banning defines two sets which capture the effects underlying MOD. With every procedure p, he associates a set GMOD(p), the generalized modification side effect of p. Similarly with every statement s, he associates a set DMOD(s), the direct modification side effect of s. MOD(s) is simply DMOD(s) extended to account for inherited aliasing effects.

More formally, GMOD(p) contains any variables which are visible both inside and outside p which can be modified in some call to p, ignoring aliasing effects. Thus, GMOD(p) is simply the subset of $\{\cup_{s\in p}$ DMOD(s)$\}$ which intersects ((p)∪(p)). The formal parameters are included in GMOD(p) because they are visible externally as their corresponding actual parameters.

DMOD(s) contains those variables whose values can be modified by execution of s, ignoring potential aliases. Let IMOD(s) be the set of variables which can be modified by execution of s, exclusive of any procedure calls in s. Then, if s contains no procedure calls, DMOD(s) is just IMOD(s).[2] If, however, s contains a procedure call, DMOD(s) must also include those variables visible at s which are modified as a result of executing the body of the called procedure.

For later use, we will define (p) as $\cup_{s\in p}$ IMOD(s). (p) is the set of all variables modified by p, exclusive of any procedure calls in p. The name (p) arises from the observation that this set would be used as an initial approximation to GMOD(p) in an iterative framework for the problem. This notation differs from that used by Banning. His IMOD set is our ; our terminology makes it easier to discuss the side effects of a statement which are independent of any procedure calls it contains.

Banning posed a set of simultaneous equations over the procedures and statements of the program to determine the values of these sets. The following equations are equivalent to Banning's, although somewhat different in form:

$$\mathrm{GMOD}(u) = ((u)\cup(u)) \cap (\cup_{s\in u} \mathrm{DMOD}(s))$$

$$\mathrm{DMOD}(s) = \mathrm{IMOD}(s) \cup [\cup_{e\in °(s)} ((\mathrm{GMOD}(v)\cap(v)) \cup \mathit{backbind}_e\{\mathrm{GMOD}(v)\cap(v)\})]$$

where $\mathit{backbind}_e$ maps (v) into (e) to model reference parameter binding. Any of the standard global data flow analysis methods can be used to solve these equations.

---

[2] Unless we consider asynchronous traps and erroneous computations. See [Spil] and [Henn] for discussions of these problems.

Given a set of DMOD sets for the call sites of a program, the corresponding MOD sets can be produced by augmenting DMOD(s) with information about the aliasing relationships which hold on entry to procedures. An algorithm for this problem is given in section 4.3 of this paper.

It is useful to examine the results of applying these equations to the example of figure 1. This results in a set GMOD(S1) = {A,B,C,D,M}, a set DMOD(e_1) = {I,J,K,L,M} for the call in MAIN, a set DMOD(e_2) = {E,A,B,C,M} for the recursive call in S1, and a set GMOD(MAIN) = {M,N}. The GMOD set for MAIN contains only global variables because MAIN has no parameters. All other variables modified in MAIN are purely local in scope.

As an aid in understanding our approach to the MOD computation, it is helpful to formalize Banning's treatment. Because much of the underlying theory of data flow analysis has been developed using the machinery of lattice theory, we will use a semi-lattice data flow framework in the style of [KaUl]. Constructing such a framework yields insight into the properties of the problem.

A *semi-lattice framework* for a data flow analysis problem is a tuple (G,L,F,M), where G = (N,E) is a graph modelling the control flow structure of the text being analyzed, L=(S,) is a semi-lattice, F:S→S is a function space representing transformations on the elements of L, and M:E→F is a mapping from an edge of G into a function in F. For convenience, we refer to a framework by the name of the semi-lattice, relying on context to resolve any ambiguity. The framework for the GMOD computation is simply L = (,L,F,M), where C is the program's call graph.

L is defined as L = (2,), where 2^X denotes the power set of X, and is set union (∪). In this framework, is , the set of all variables, and is ∅, the empty set. Finally, the meet operation imposes a partial ordering on the elements of L, such that

$$a \geq b \quad \text{iff} \quad a \; b = b$$
$$a > b \quad \text{iff} \quad a \; b = b \text{ and } a \neq b.$$

In L, ≥ is equivalent to ⊆.

The framework's function space characterizes the propagation of information along edges in the data flow graph. For our framework L, F is the collection of all functions of the form:

$$f(X) = (A \cap X) \cup g\{X \cap B\}$$

where A and B are constant sets, A ≥ , B ≥ , and g is an arbitrary function mapping → 2.

Finally, M maps an edge in into a function in F. Given an edge e = <u,v> representing a call from u to v, the function $f_e \in F$ associated with e is defined by the assignment of edge-specific values to the constants A and B, and to the function g. For L and F, M assigns the value (v) to A, (v) to B, and *backbind*$_e$ to g. This yields edge-specific functions of the form:

$$f_e(X) = ((v) \cap X) \cup \textit{backbind}_e\{X \cap (v)\}$$

This propagation function reflects our intuitions about the problem. The first term, (v)∩X, selects those names in X which are global to v. The term *backbind*$_e${X ∩ (v)} contains the names of any actual parameters of the call whose corresponding formals are contained in X.

When applied to an approximation of the set GMOD(v), this function will return a set which contains the appropriate names for inclusion in the corresponding approximation to DMOD(s) as the contribution of procedure v. In an iterative framework, our initial approximation for GMOD(v) would be formed by setting DMOD(s) to IMOD(s), for all statements s in v, and unioning those values to obtain (v).

There are two primary motivations for presenting Banning's formulation of the GMOD problem in this paper: to provide a clear classical treatment of the problem, and to show the insights which lead to the new approach presented in the next section. In particular, we are interested in insights about the speed with which the problem, in this form, can be solved. Kam and Ullman present a class of data flow frameworks which can be solved efficiently using the iterative algorithm [KaUl]. To qualify for membership in this class, a framework must meet two sets of conditions, a set of admissibility criteria and a single rapidity condition. Our framework L is an *admissible* Kam-Ullman framework.

An admissible Kam-Ullman framework is *rapid* if:

$$f(X) \geq X \;\; f() \quad \forall f \in F, \forall X \in L.^{3}$$

There are functions in F which do not meet this condition, so our framework for GMOD is not *rapid*.

To demonstrate this, we plug our characterization of $f \in F$ into the Kam-Ullman condition:

$$f(X) \geq X \;\; f()$$
$$f(X) \geq X \cup ((A \cap) \cup g\{\cap B\})$$
$$f(X) \geq X \cup (A \cap \varnothing) \;\cup g\{\varnothing\}$$
$$f(X) \geq X$$

Expanding the left side yields

$$(A \cap X) \cup g\{X \cap B\} \geq X$$

Clearly, $A \cap X \geq X$. The question is, are there functions g for which $g\{X \cap B\} \not\geq X$. If such functions exist, our framework is not rapid. Since g is an arbitrary map, $g: \rightarrow 2$, g can carry $x \in X$ into the complement of X. Such a function is not rapid.

We should ask, will actual programs generate such functions? In practice, B is mapped to (v), so any procedure which passes a purely local variable as an actual parameter in a procedure call generates a function which is not rapid. Such a call will map the corresponding formal parameter, (v) into (e), where (e) is guaranteed to be outside $X \cap (v)$, since it is a local variable of u.

The problem with implementing a Banning style formulation of the GMOD/DMOD problem is computational speed. The presence of call-by-reference formal parameters complicates a data flow framework for the problem sufficiently to prevent it from being either Kam-Ullman rapid or Graham-Wegman fast. An algorithm for solving Banning's framework in $\mathbf{O}(\| \times \|)$ time is presented in [Coop]. The next section presents a significantly faster technique.

4. Dividing the Problem

The problem of computing MOD information divides naturally into two separate subproblems: analyzing modifications to global variables and analyzing modifications to call-by-reference formal parameters. If we call the subset of GMOD(u) dealing with global variables GMOD(u) and the subset dealing with reference formal parameters GMOD(u), and subdivide DMOD(s) in a similar manner, then the equations given in section 3 can be re-written as:

$$\text{GMOD(u)} = \text{GMOD(u)} \cup \text{GMOD(u)}$$
$$\text{DMOD(s)} = \text{DMOD(s)} \cup \text{DMOD(s)}$$

where

$$\text{GMOD(u)} = \cup_{s \in u} \text{DMOD(s)},$$
$$\text{DMOD(s)} = (\text{IMOD(s)} \cap (u)) \cup (\cup_{e \in \hat{e}(s)} \text{GMOD(v)} \cap (v))$$

$$\text{GMOD(u)} = \cup_{s \in u} \text{DMOD(s)},$$
$$\text{DMOD(s)} = (\text{IMOD(s)} \cap (u)) \cup (\cup_{e \in \hat{e}(s)} \mathit{backbind}_e\{\text{GMOD(v)} \cap (v)\})$$

This effectively splits the problem into independent systems of equations for GMOD/DMOD and GMOD/DMOD. Our technique is to efficiently solve each problem and combine this information to produce MOD(s) for each call site.

This division of the problem appears to assert that the modification of global variables is completely divorced from the modification of actual parameters. In fact, this is not the case. Fortunately, interaction between the two problems is limited. In the realm of FORTRAN programs, all such interactions arise as the result of aliasing relationships which are accounted for by the algorithm which extends a DMOD set to become a MOD set. In the more general name scoping of an Algol-like language, we must worry about a formal parameter being a global variable of an internal procedure. This case is discussed in section 7.

4.1. Computing GMOD

Computing GMOD is much simpler than computing GMOD for all variables, because the renaming effects which arise from call by reference formal parameters at call sites no longer need to be modelled. The data flow equations for GMOD are:

$$\text{GMOD(u)} = \{\cup_{s \in u} \text{DMOD(s)}\}$$

$$\text{DMOD(s)} = (\text{IMOD(s)} \cap (u)) \cup (\cup_{e \in \hat{e}(s)} \text{GMOD(v)} \cap (v))$$

It is instructive to examine the solution to these equations for the program given in figure 1. For the example, the values assigned are:

[3] [KaUl] See observation six, page 166.

GMOD(S1) = {M};
DMOD(e) = {M};
DMOD(e) = {M};
GMOD(MAIN) = {N,M}.

This section presents a data flow framework for computing GMOD(p) for all p ∈ .

To simplify the presentation, we show a data flow framework for GMOD, in the style of [KaUl]. The framework for GMOD is L = (,L,F,M), where
is the call multi-graph of the program, as defined in section 2.

L is defined as (2,), where is the set of all global variables, and is set union (∪). In L, is and is . Since is idempotent, commutative, and associative, L is a semi-lattice.

F consists of the set of all possible propagation functions for the GMOD computation. Each function f ∈ F is of the form:

$$f(X) = A \cap X$$

where A is a constant set, A ≥ . F is just the collection of all such functions.

The mapping M:→F consists of a rule for assigning a value to the constant A. For an edge e = <u,v>, M assigns the value (v) to A, producing a function

$$f_e(X) = (v) \cap X.$$

Formulating the GMOD problem in a semi-lattice framework allows us to show several useful properties which hold true for the framework. The function space F is an admissible Kam-Ullman function space, and the framework is *rapid* in the sense of [KaUl]. Because the framework is rapid, a depth first version of the iterative algorithm will halt on an instance of this problem in d()+2 passes.[4]

A version of the iterative algorithm is presented in [Coop] which efficiently uses the resources of the programming environment. That algorithm annotates the nodes and edges of the call graph with sufficient information to allow easy updating of the call graph in response to editing changes. The algorithm also allows deferred updates to simplify synchronization concerns in the environment's database and limits its attention to the affected area of the program's call graph. By visiting the procedures in an order analogous to the order specified in [KaUl], the algorithm can achieve the d()+2 pass time bound.

It should be noted that this framework is sufficiently simple to be solved quickly by any of the global data flow analysis techniques which achieve near linear time bounds. The propagation functions for GMOD are surely among the most simple functions of practical interest proposed in the literature.

4.2. Computing GMOD

Computing GMOD is conceptually more complex than computing GMOD. The complications which arise in a direct application of Banning's equations result from the renaming which occurs with call-by-reference formal parameters. We first present a conceptual framework for computing GMOD and then consider the problem of performing the computation efficiently.

4.2.1. A Conceptual Framework

To derive GMOD, we build a mapping which describes the patterns of parameter passing and renaming at each call site in the program, and then use this mapping to compute GMOD. The mapping provides, for each procedure p, a complete list of the formal parameters which can be bound to each (p), through any chain of call sites in the program. Because we are only interested in binding chains through the program, we can restrict our attention to actual parameters which are also formal parameters of the *calling* procedure.

To capture the effects of parameter bindings, we define the relation *map* on the set of formal parameters of the program. A pair (a,b) ∈ *map* if and only if a = (e), b = (v), and a ∈ (u), for some edge e=<u,v>. *Map* contains all such pairs. We will represent *map* as an m × m bit matrix, where m is $|\cup_{u\in}(u)|$. For each formal parameter of a procedure in the program, there will be an index 1≤i≤m such that the i^{th} row and column of *map* will contain information about that parameter.

The *map* information for our example program is shown in figure 2. Since MAIN has no

[4]For a directed graph G, d(G) is the loop connectedness of the graph, defined to be the maximum number of back edges in any cycle free path in the graph with respect to a given depth first spanning tree. For a reducible graph, d(G) is independent of the spanning tree chosen [KaUl].

Map for example program

	S1.A	S1.B	S1.C	S1.D
S1.A	0	1	0	0
S1.B	0	0	1	0
S1.C	0	0	0	1
S1.D	0	0	0	0

Figure 2

formal parameters, there are only four columns and rows, each labelled with a formal of S1. The call site in MAIN adds no entries to the *map*, since it passes no formals of MAIN. The recursive call in S1 adds entries indicating that A is bound to B, B is bound to C, and C is bound to D. It is important to realize that none of these entries imply an aliasing relationship between two formal parameters, since any call resulting in the binding of one formal to another formal of the same procedure makes the binding across different incarnations or generations of the procedure.

The reflexive transitive closure of *map*, denoted , can be used to compute GMOD. For every formal parameter of every procedure in the program, specifies those formal parameters to which it can be bound. Since *map* describes all binding chains of length one, will include all possible chains of bindings through the program. Thus, it completely specifies the passing of the formals of one procedure through other procedures.

Given , computing GMOD should be simple. *Map**[i,j] is true if the formal corresponding to i can be bound, along some call chain in the program, to the formal represented by the j^{th} position. Therefore, if any variable corresponding to a true bit in the i^{th} row of is in the set of the procedure which declares it, the formal corresponding to i should be in the GMOD set of its declaring procedure. We need not consider the case where a formal S1.x is bound to a formal S4.y which is in turn passed out of S4 as an actual and modified in some indirect call, since the structure of ensures us that there will also be an entry in showing S1.x bound to the formal which is eventually modified.

Given and the (p) set for each procedure p, a simple algorithm can be derived to compute GMOD(p). Let m be the total number of formal parameters in procedures of the program. Then is an $m \times m$ matrix, and G is an $m \times 1$ vector. The bit vector G will represent the sets ((p) $\cap$ (p)) for each procedure p. Since each formal parameter of a procedure in the program is represented in both and G, they are assumed to have the same layout. To initialize G, set the bit for a formal true if and only if the formal is in (p) and in (p). Thus a formal's position in G is set to true only if the procedure in which it is declared modifies it.

Now, GMOD = G + $\times$ G, where $\times$ denotes post-multiplication of a bit-matrix by a bit-vector. This operation can be performed in **O**(|G|) bit-vector operations of length |G|. The resulting vector, GMOD, has the same layout as G and . It can be dissected to produce individual sets GMODσR(p), for each p $\in$.

The simplicity of computing GMOD from makes this approach appealing. Unfortunately, computing directly is expensive. Using Warshall's algorithm, this would require **O**(m^3) time, where m is the dimension of [AhHU]. While there exist faster methods of directly computing the reflexive transitive closure of a relation, none of these methods are sufficiently fast. A version of the iterative algorithm can be constructed which directly solves Banning's equations in **O**($\| \times \|$) time, assuming that the maximum number of formal parameters in any procedure is bounded by a small constant, [Coop]. To improve significantly on that time bound will require a means of computing which takes advantage of the underlying structure of the problem.

4.2.2. A Computational Framework

To arrive at an efficient method for computing , we reformulate the problem to make it more readily recognizable as a data flow analysis problem. Observe that each column vector of represents a subset of $\cup_{n\in}$ (n). For a column i, the subset contains precisely those formal parameters which can eventually be bound to the formal mapped to i. We will call this set the formal's *bound* set. The *bound* sets summarize the chains of binding which occur on all paths between the root of the call graph and the procedure declaring the formal parameter.

With each node of the call graph, we associate a tuple

<b, b, b, ..., b.if 0>0 . tm Line 718 -- I cannot skip multiple pages
>

where the individual b's represent *bound* sets of formal parameters. To simplify the notation, each tuple has elements, indexed from 0 to Elements b through b represent the *bound* sets of each of the i formal parameters of the procedure corresponding to the node, and the remaining sets, b and b_{i+1} through b are all set to ∅.

The tuples associated with nodes in the call graph are related by the following equation:

$$tuple(v) = \cup_{e=<u,v>} f_e(tuple(u))$$

where $f_e(t) = s \cup C$. Here, s is a tuple constructed from the elements of t, and C is an edge specific constant. The tuple s is constructed as

$$s = <\emptyset, t_{back(1)}, t_{back(2)}, ..., t_{back()}>$$

where back is a mapping from the actual parameters of the call site to the formal parameters of the *calling* procedure. Back is defined as:

back(i)	= j	iff there is a j s.t. (e) = (u)
	= 0	otherwise.

If (e) is also a formal of u, back(i) is the ordinal position of that formal among the formals of u. Since back(i) is 0 for any (e) which is not a formal of u, the positions in s corresponding to such actuals contain t, which is defined as ∅.

In our propagation function, the constructed tuple s contains a subset of the *bound* sets of formals of the calling procedure. The constant C contains the names of the formals of the calling procedure which correspond to the elements of s which are not ∅. Thus, C is constructed as:

$$<\emptyset, {}_{back(1)}(u), {}_{back(2)}(u), ..., {}_{back()}(u)>$$

Here, we assume (u) is ∅. This term introduces the formal parameters themselves into the *bound* sets.

Tarjan defined the single source path expression problem as, given a graph G = (N,E) with distinguished source vertex s, annotating each n ∈ N with a regular expression P(s,n) which represents all paths from s to n in G [Tarj1] [Tarj2] [Tarj3]. He has formulated a particularly efficient method for solving this problem. The algorithm uses two steps to compute expressions for the paths of a graph. Because all of the call chains in a program begin with a single node in the call graph, the program's main procedure, our problem can be posed as a single source path expression problem.

The first step of Tarjan's algorithm computes a path sequence, an ordering in which to traverse the edges of the graph. For suitable frameworks, the algorithm computes this path sequence in $\mathbf{O}(\|\alpha(\|,\|))$ time, where α is the functional inverse of Ackerman's function. The path sequences it produces are of length $\mathbf{O}(\|\alpha(\|,\|))$.

The second step of Tarjan's algorithm is an elimination algorithm which uses the path sequence computed by the first phase to compute, for each node in the graph, a regular expression which describes the paths through the graph from the source to the node. By placing appropriate interpretations on the regular expression operators ∪, , and *, representing respectively the confluence of paths, the conjunction of paths, and a cycle in the graph, the elimination step can be used to compute our *bound* tuples. This step requires $\mathbf{O}(n + r)$ time, where n is the length of the path sequence and r is the cumulative time required for regular expression manipulation. For the complete algorithm to achieve a time bound of $\mathbf{O}(\|\alpha(\|,\|))$, the regular expression interpretations must all be computable in constant time.

Interpreting R1 R2

```
/* compute the joining edge */
temp_tuple ← f<w,x>(R1)

/* propagate changes across the path for R2 */
for i ← 1 to
   for j ← 1 to
      if (x) ∈ R2(y)
        then R2(y) ← R2(y) ∪ temp_tuple
```

Figure 3

Intuitively, the constraint on computing the closure of an expression in constant time implies that given the mapping of formal parameters generated by a single trip around a cycle in the call graph, we can compute, in constant time, the complete mapping which results from multiple trips around the cycle. In general, this is not possible. The following example illustrates this point:

```
SUBROUTINE S( x, x, ..., x )
        ...
        CALL S( y, x, x, ... x.nm 1
)
        ...
        END
```

If |(S)| is unbounded, then the computation of can take unbounded time.

Fortunately, it is unlikely that |(S)| is unbounded. We assume that there is a small constant which bounds |(p)| for each procedure p. If this is true, and is independent of the size of the program, then the time to interpret the closure of an expression R can be viewed as a constant. Hence, Tarjan's techniques can be used to compute in $\mathbf{O}(\|\alpha(\|,\|))$ time.

Tarjan expresses the requirements which a problem must meet to be suitable for solution with his technique as a *continuous data flow framework*. Such a framework can be formulated for the *bound*-tuple computation. To understand the time complexity of the resulting algorithm, we should examine the implementations of the regular expression operators on the bound sets.

The interpretation of $\cup$ is simply the tuple constructed by unioning the individual sets which comprise the tuple. Thus,

$$T \cup S = <t\cup s, t\cup s, ..., t\cup s>.$$

This operation can be done in $\mathbf{O}(1)$ time, assuming is a small constant.

Computing *R1 R2* is more complex. Here, *R1* is a tuple representing the set of paths from procedure v to procedure w, and tuple *R2* represents the set of paths from procedure x to procedure y. The junction of the two paths is an edge e = <w,x>. An algorithm for computing the correct tuple for the set of paths from v to y which go through e is given in figure 3. The algorithm simply computes the addition of the single edge e, and then uses the knowledge contained in *R2* to directly update itself. This algorithm takes $\mathbf{O}()$ time.

*Interpreting R1**

```
/* assuming a cycle from v → v      */
/* build β to represent cycle       */
for i ← 1 to
        for j ← 1 to
                if (v) ∈ R1(v)
                        then β(i,j) ← true
                        else β(i,j) ← false

/* reflexive, transitive closure  */

β ← β*

/* use β to build new bound tuple */

for i ← 1 to
   for j ← 1 to
      if β(i,j)
        then R1(v) ← R1(v) ∪ R1(v)
```

Figure 4

An algorithm to take the closure of the information represented by a tuple is required to interpret cycles in the graph. This is shown in figure 4. The algorithm capitalizes on the fact that all of the relevant behavior of the cycle is contained in the *bound* tuple for the node which begins and ends the cycle, v. The algorithm computes a side data structure, a $\times$ array β, takes its reflexive transitive closure, and uses that closure to compute the closure of the entire expression. The dominating cost is the reflexive transitive closure of β, which takes $\mathbf{O}()$ time.

Since we assume that is a small constant, all of the regular expression interpretations require constant time. Thus, we can annotate the call graph with the tuples containing the columns of in $\mathbf{O}(\|\alpha(\|,\|))$ time. This information can then be used to compute GMOD.

If the reader is uncomfortable with this use of Tarjan's algorithm, it should be noted that any of the structural data flow algorithms can be applied, using the same assumptions. Michael

Burke of IBM Research has described to us an interval analysis adaptation of this approach [Burk]. It is easy to envision equivalent strategies which would work with the Graham-Wegman method [GWeg] and the SSFG method [FaKZ].

4.3. Computing MOD

Given GMOD and GMOD, constructing MOD sets for each statement containing a call site is a three step process. First, GMOD(p) and GMOD(p) must be combined to form GMOD(p) for each procedure p. Next, DMOD(s) is produced for each call site by projecting the GMOD set of the called procedure back along the edge, according to the equation from Banning's formulation. Finally, DMOD(s) is extended to account for aliasing, yielding MOD(s).

Computing GMOD(p) is simple: we set GMOD(p) to GMOD(p) $\cup$ GMOD(p). This works because GMOD and GMOD are disjoint, except when a call site passes a global variable as an actual parameter. This case corresponds precisely to alias introduction by a global variable. The algorithm which derives MOD from DMOD accounts for aliasing effects and handles this interaction of GMOD(p) and GMOD(p) naturally.

Given the set GMOD(v), DMOD(s) for a statement containing a single call site e=<u,v> is produced by computing an edge specific propagation function for the GMOD information and applying it to GMOD(v). The function is based on the equations from Banning's framework. This is done by setting:

$$\begin{aligned} DMOD(s) &= IMOD(s) \\ &\cup (GMOD(v) \cap (v)) \\ &\cup backbind_e\{GMOD(v) \cap (v)\}. \end{aligned}$$

This also adds in IMOD(s) to account for any purely local side effects of s.

To obtain MOD(s), set MOD(s) to DMOD(s) and add every possible alias of a variable in DMOD(s). To model aliasing information, we assume that each procedure is annotated with a set of pairs <x,y>, where <x,y> is interpreted to mean that x and y can be aliased on some path leading to an invocation of the procedure.[5] The algorithm of figure 5 performs this computation. This computation requires time linear in the size of DMOD(s) and the number of aliases for variables in DMOD(s). While the number of potential aliases can be large, programs with such aliasing patterns are difficult to write and understand. We expect that actual programs do not exhibit such aliasing patterns.

Computing MOD from DMOD

```
for each statement s in u
  do
    MOD(s) ← DMOD(s)

    for each x ∈ MOD(s)
      do /* add aliases */
        for each y s.t. <x,y>
                    are potential aliases
                    on entry to u
          do
            MOD(s) ← MOD(s) ∪ y
            end
        end
```

Figure 5

5. Computational Complexity

Because the presentation of the algorithm has been interspersed with its derivation, a brief review of the algorithm and its complexity may be helpful. We have taken the GMOD computation proposed by Banning and split it into two distinct problems. The first of these involves annotating each node with a set GMOD and each edge with a set DMOD. These sets describe the cumulative side effects of procedure invocations on the values of global variables. This data flow problem can be solved using the iterative algorithm in $\mathbf{O}(\| \times d())$ time, where d() is expected to be a small constant.

[5] [Coop] shows an algorithm which computes aliasing information in a programming environment.

The second step involves annotating each node with a set GMOD and each edge with a set DMOD. These sets describe the cumulative MOD side effects which involve call by reference formal parameters. Using Tarjan's method, this data flow problem can be solved in **O**(||α(||,||)) time.

To obtain Banning's GMOD and DMOD sets, we simply union the GMOD and GMOD sets, and propagate along each graph edge the resulting information in the form of a DMOD set computed from the new GMOD sets. This requires visiting each node and each edge once, so it is a **O**(||) operation.

Finally, to derive MOD sets from the DMOD sets, we extend DMOD by adding in potential aliases. This operation requires a time linear in the size of the DMOD set and the number of potential aliases available on entry to the procedure containing s. Thus, the cost of computing the summary information is **O**((|| × d()) + (||α(||,||)) + A) where A is the total number of potential alias pairs in the program.

6. Updating the Information

For summary information to be useful in the context of a programming environment, there must be efficient techniques for updating the summary information of a program in response to editing changes in one of its constituent procedures. To update MOD(s), we update GMOD and GMOD separately. If this results in a change to GMOD, then we repeat the three step derivation for MOD(s) presented in section 4.3.

Deleting a Variable

```
G[i] ← false

GMOD ← GMOD  not(  .i)

for each true bit in  .i
   where j is its position
   do
     temp ← G  j.

     if temp is all false  G[j] is false
       then GMOD[j] ← false
       else GMOD[j] ← true
```

Figure 6

To update GMOD, we rely on techniques presented in [Coop]. Briefly, the call multi-graph is annotated with sufficient information to allow easy updating using a version of the iterative algorithm. The updates maintain the GMOD information in response to editing changes and limit their attention to that portion of the program's call graph where the data flow information has actually changed.

To update GMOD, we have developed a set of techniques for directly updating . Using the updated , GMOD is recomputed in the same manner as in section 4.2.1

Because the array concisely expresses the intricate relationships between the formal parameters of different procedures, it is a useful tool for updating itself. Textual changes to the program can impact the summary information in two distinct ways. A change to the source text can change the initial information from which we compute the summary information, by adding or deleting variable names from one of the (p) sets. Because such a modification does not change the call graph, we call this a non-structural change. A change to the source text can also modify the structure of the call graph and its binding patterns, by addition or deletion of a call site.

To examine non-structural changes, consider a single variable change in (p). If the change adds a variable to (p), then we must mark all variables which can be bound to that parameter as potentially modified. In a bit vector implementation, the addition of a modification to the i^{th} formal parameter in G can be accomplished by setting

$$G[i] \leftarrow \text{true}$$
$$GMOD \leftarrow GMOD \ \ _{.i}$$

where is the boolean *or* function and $_{.i}$ is the i_{th} column vector of . Recall that $_{.i}$ has a bit set to true for every variable which can be bound to the formal parameter corresponding to i, so this operation simply adds that set of variables to the complete set of GMOD information for the program in a single step.

If the change removes a variable from (p), the process is slightly more complex. Since each

variable in the GMOD vector can be included as the result of distinct entries in the sets of many other procedures, we can not simply set to false all GMOD bits arising from the changed variable. Instead, the update must recompute the potentially changed bits from scratch. A simple algorithm to perform this operation is shown in figure 6. Note that is used to represent the boolean *and* function rather than semi-lattice meet. It first removes all variables from GMOD which can be bound to the i^{th} formal. Next, for each of these formals, it examines all formals to which it can be bound. If one or more of these is in the set of the procedure declaring it, it is included in GMOD.

Just as the non-structural changes divide nicely into addition and deletion of individual variables from the sets, the structural changes divide into the addition and deletion of call sites. As with non-structural changes, the update for addition can only add variables to the GMOD sets, while deleting a call site can only remove variables from GMOD sets. Unfortunately, the updates for both of these cases are complex. In response to the addition of a call site, new bindings must be added and their impact propagated both forward and backward in the matrix. After accounting for the new bindings, an update must be applied which is identical to that for adding a variable to .

The actual update for adding an edge is shown in figure 7. The addition of an edge e = <u,v> requires detailed explanation. Since the call site can pass more than a single parameter, the update is specified for a single actual parameter, and then enclosed in a loop which iterates over all of the actuals at the new call site.

First, every formal parameter to which (v) can be bound, in any combination of calls out of v must have its column vector updated to reflect the fact that each of the formals which can reach (e) can now reach it, through the new call site. Recall that a column vector of , designated $_{.i}$, has a true bit for each formal parameter which can be bound to the formal indexed by i. Likewise, a row vector of , designated as $_{i.}$, has a true bit for every formal to which the formal indexed by i can be bound. Thus, we can perform this update by setting each column vector with a true bit in the row for the current formal to the logical or of itself and the column vector for the corresponding actual parameter in the new call site. This is done in the first inner loop.

Update for Adding a Call Site

```
for p ← 1 to |(e)|
   i ← index of p(e) in
   j ← index of p(v) in

   if not(ij) /** if already true, no work **/
     then
        for each true bit in j.
           where k is its index
                          .k ← .k .i

        for each true bit in .i
           where k is its index
           k. ← k. j.

        .j ← .j .i

        i. ← i. j.

        ij ← true
        if GMOD[j]
          then GMOD ← GMOD .j
```

Figure 7

Next, each variable which can reach (e), in any combination of calls leading to an invocation of u, must have its row vector updated to reflect the fact that it can now reach any of the formals to which (v) can be bound. In a manner analogous to the first inner loop, we need to set the appropriate row vectors to the logical or of themselves with the row vector for the current formal. The row vectors requiring updates are precisely those which have a true bit in the column vector corresponding to the current actual parameter. The second inner loop performs this part of the update.

Having accounted for all of the variables indirectly involved in the call site, the update must now account for those directly involved. It updates the column vector of (v) to reflect new variables which reach (v) through (e). Likewise, it modifies the row vector of (e) to account for new variables which (e) can reach through (v). Finally, it sets true the entry indicating that (e) is bound to (v), and updates the bit vector representing GMOD.

Removing Bindings for a Call Site

```
for i ← 1 to |(v)|

  if (e) ∈ (u)
    then
      j ← index of (e) in
      k ← index of (v) in
      jk ← false
```

Figure 8

Deleting an edge from a call site is more complex. Since the information retained by the algorithm is , it is not easy to back out the impact of an edge. We do not as yet have a bit-vector oriented solution to this problem. There is, however, a simple iterative approach which will work.

Assume that a call site corresponding to e=<u,v> has been removed from the program. The only information available about *map* is encoded in the relation. Deleting e removes a set of bindings from the program. It is not immediately apparent from , however, whether or not equivalent bindings appear along other call chains in the program. To update the relation, we can fall back on a fundamental fact about
and the *bound* tuples. Recall that we formulated the problem of annotating the graph with tuples as a data flow problem with the following equation:

$$\text{tuple}(v) = \underset{e=<u,v>}{} f_e(\text{tuple}(u)).$$

We can use this equation to produce a two step algorithm for updating
in response to a call site deletion.

In the first step, all of the bindings involved in the old call site are removed from . This is accomplished by the code fragment of figure 8.

As a second step, we can simply apply a version of the iterative algorithm like that given in [Coop] and used to update the GMOD information. This algorithm will update the tuples of the program until they become consistent, under the data flow equations described above.

7. Name Scoping

Throughout this exposition, we have assumed the two level name scoping rules of FORTRAN. It is important to understand the impact of more general name scoping rules on the computation.

The assumption of section 4 is that the GMOD and GMOD computations are independent, except for the interactions caused by the aliasing effects of reference formal parameters. In languages which allow lexically nested scoping, like Algol and PL/I, a second complication arises.

With nested definitions of procedures, there are cases where, for a variable v, $v \in (p)$ and $v \in (q)$. This can only occur if q is lexically nested inside of p, as in a PL/I internal procedure. In this case, however, it is easy to envision a program where $v \in \text{GMOD}(q)$, and q is called from p. As formulated in section 4, our GMOD computation will produce incorrect information, since GMOD(p) will not contain v.

Fortunately, the remedy for this situation is simple. By solving the GMOD computation before the GMOD computation, we can construct our initial vector G from the set

$$((p) \cup \text{GMOD}(p)) \cap (p)$$

rather than from $(p) \cap (p)$. If we observe this ordering constraint, the algorithm will produce correct information for the more general case of multiply-nested name scopes.

In fact, this observation suggests a slightly different computational approach to solving the entire GMOD problem. If we use Tarjan's algorithm to solve both the GMOD problem and the *bound*-tuple problem concurrently, the information needed for the computation will be available at the appropriate time and we will be able to use a single path sequence computation to solve both problems.

8. Conclusions

The computation of flow insensitive summary information has been broken into two subproblems, a computation for global variables and a computation for call-by-reference formal parameters. There are techniques to solve each of these subproblems in nearly linear time, using a rapid Kam-Ullman framework for the global problem and Tarjan's single source path expression

algorithm for the reference formal parameter problem. The information produced by these two techniques can be combined to solve the complete problem, yielding an efficient algorithm for computing flow insensitive summary sets. To apply these techniques in a programming environment, we have developed a series of techniques to update the summary information in response to editing changes to the program's source text.

References

[AhHU] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, Ma., 1974.

[Bann] J. Banning, "A Method for Determining the Side Effects of Procedure Calls", PhD Dissertation, Stanford University, Aug. 1978.

[Bart] J. Barth, "A Practical Interprocedural Data Flow Analysis Algorithm". *CACM* Vol. 21, No. 9, Sept. 1978, pp. 724-736.

[Burk] M. Burke, Private Communication, November 1983.

[Coop] K. Cooper, "Interprocedural Data Flow Analysis in a Programming Environment", PhD Dissertation, Rice University, May 1983.

[FaKZ] R. Farrow, K. Kennedy, and L. Zucconi, "Graph Grammars and Global Program Flow Analysis", *Proc. 17th Ann. IEEE Symp. on Foundations of Computer Science*, Houston, Tx, November 1975.

[GrWe] S. Graham and M. Wegman, "A Fast and Usually Linear Algorithm for Global Flow Analysis", *JACM* Vol. 23, No. 1, January 1976, pp. 172-202.

[Henn] J. Hennessy, "Program Optimization and Exception Handling", *Proceedings 8th POPL*, Jan. 1981.

[HoKe] R. Hood and K. Kennedy, "A Programming Environment for Fortran", Rice University, Department of Computer Science Technical Report, in preparation.

[KaUl] J. Kam and J. Ullman, "Global Data Flow Analysis and Iterative Algorithms", *JACM*, Vol. 23, No. 1, Jan. 1976, pp. 158-171.

[HoKe] R. Hood, and K. Kennedy, "A Programming Environment for Fortran", Rice University, Department of Mathematical Sciences TR83-22, 1983.

[Myer1] E. Myers, "A Precise and Efficient Algorithm for Determining Existential Summary Data Flow Information", University of Colorado at Boulder, Department of Computer Science Technical Report #CU-CS-175-80.

[Myer2] E. Myers, "A Precise Interprocedural Data Flow Algorithm", *Proceedings 8th POPL*, Jan. 1981.

[Rose] B. Rosen, "Data Flow Analysis for Procedural Languages", *JACM*, Vol. 26, No. 2, April 1979, pp. 322-344.

[Spil] T. Spillman, "Exposing Side-Effects in a PL/I Optimizing Compiler", IFIPS Proceedings, 1971, pp 376-381.

[Tarj1] R. Tarjan, "Applications of Path Compression on Balanced Trees" *JACM*, Vol. 26, No. 4, Oct. 1979, pp. 690-715.

[Tarj2] R. Tarjan, "A Unified Approach to Path Problems", *JACM*, Vol. 28, No. 3, July 1981, pp. 577-593.

[Tarj3] R. Tarjan, "Fast Algorithms for Solving Path Problems", *JACM*, Vol. 28, No. 3, July 1981, pp. 594-614.