Incremental Analysis of MOD Problem for C*

Jyh-shiarn Yur Barbara G. Ryder

Department of Computer Science
Rutgers University
Hill Center, Busch Campus
Piscataway NJ 08855, USA
{yur,ryder}@cs.rutgers.edu

Created August 23, 1995

Reformatted December 21, 1998

Abstract

Incremental data flow analysis seeks to efficiently and precisely update data flow information after source code changes, based on the knowledge of the former solution and the changes, without re-computation from scratch. Interprocedural modification side effect analysis (i.e., MOD) finds the set of variables modified by execution of a statement. Computing MOD information for a language with general purpose pointers, like C, is very complicated and costly. We study the applicability of the hybrid algorithm [MR90] to incrementalize $PMOD$ - a subproblem of the MOD problem for C. We also show how to update data flow information when non-structural or structural changes are made to the flow graph.

1 Introduction

Information about the uses and definitions of data in programs is crucial to software testing, debugging and maintenance, especially of large software systems [Ryd89]. This information is also essential in program optimization and parallelization [ASU86]. Since a software system evolves with time, this information should be updated to reflect the current state of the system. As the size of the system becomes very large, keeping the information up-to-date is costly. Instead of re-computing the data flow information from scratch, incremental analysis techniques seek to update the information based on the former solution and knowledge of the source changes. Two methods are commonly used in incremental analysis: iteration [BR90, PS89] and elimination [RP88]. Combining these two approaches, Marlowe and Ryder [MR90, MR91] proposed a hybrid algorithm for incremental analysis. Intuitively, the hybrid algorithm partitions the flow graph into components, then uses iterative techniques within components and applies the idea of elimination to information propagation on the reduced graph.

Interprocedural modification side effect analysis (i.e., MOD) finds the set of variables in a program whose values can be affected by the execution of an individual statement in the program. Cooper and Kennedy [CK87, CK89] decomposed the MOD problem for FORTRAN-like languages into side effects on global variables and side effects induced by parameter passing. Based on the decomposition, they proposed an efficient flow-insensitive[1] algorithm for the problem.

However, in the presence of general purpose pointers, as in C, the problem of finding interprocedural pointer aliasing becomes at least NP-hard [LR91, Lan92]. Landi and Ryder [LR92] proposed a safe approximate algorithm for the interprocedural pointer aliasing problem. A *safe* solution may be more conservative

*This work was supported, in part, by NSF Grant CCR-9501761.

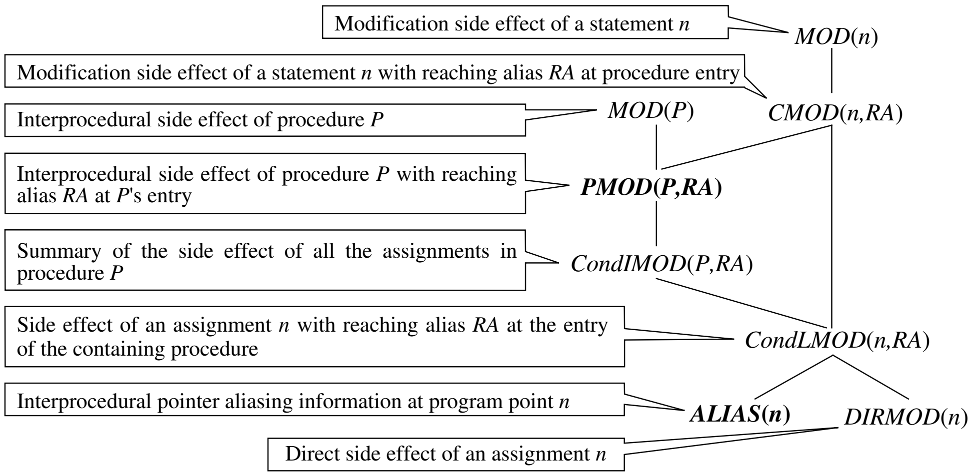


Figure 1: Decomposition of the MOD problem [LRZ93]

than a precise one, but contains the precise solution. This work led to a decomposition of the MOD problem for languages with general purpose pointers [LRZ93].

This paper describes how we have incrementalized $PMOD$—a subproblem of the MOD problem for C. We also consider two categories of changes, *non-structural* and *structural* changes, which can be made to the *call-RA graph*—the program representation used to solve $PMOD$, and show how to update the required information in computing $PMOD$ to handle these changes. Since one source change can result in several representation changes, we consider various types of source changes and analyze what representation changes correspond to a simple source change.

The rest of this paper is organized as follows. Section 2 describes the MOD problem for C and gives a brief review of the problem decomposition in [LRZ93]. In section 3, we give our formulation of $PMOD$, which is more suitable for factorization, and then demonstrate the applicability of the hybrid algorithm to $PMOD$. We propose information updating algorithms to handle various changes on the call-RA graph in section 4. In section 5, we discuss the correspondence between the changes of a source program and the changes of its call-RA graph. We summarize and preview future work in section 6.

2 MOD Problem for C

The MOD problem explores how executing a statement may affect the values of variables in the program. A statement can change the value of a variable by direct assignment to that variable or by indirect assignment through one of its aliases.

The MOD problem decomposition in [LRZ93] is pictured in Figure 1. The following briefly explains each subproblem:

DIRMOD

$DIRMOD(n)$ is defined as the direct side effect through assignments at program point n.

ALIAS

$ALIAS(n, RA)$ is the interprocedural pointer aliasing problems at program point (or statement) n, given reaching alias RA at the entry of the procedure containing n [LR92].

[1] With respect to the interprocedural criterion [IC] for flow insensitivity in [MRB95]: a data flow problem is *flow insensitive* if its solution does not depend on the intraprocedural structure of the program. On the other hand, a problem is *flow sensitive* if it requires propagation of information across calls and through paths in the procedures being called.

$$CondIMOD(P, RA) = \bigcup_{\substack{n \text{ is an assign-} \\ \text{ment in } P}} CondLMOD(n, RA) \tag{1}$$

$$PMOD(P, RA) = CondIMOD(P, RA) \cup \bigcup_{\substack{call_Q \text{ in } P \text{ and } RA' \in \\ Reach(call_Q, RA)}} b_{call_Q}(PMOD(Q, RA')) \tag{2}$$

$$CMOD(n, RA) = \begin{cases} CondLMOD(n, RA) & \text{if } n \text{ is an assignment;} \\ \bigcup_{RA' \in Reach(n,RA)} b_n(PMOD(Q, RA')) & \text{if } n \text{ is a call of } Q; \\ \emptyset & \text{otherwise.} \end{cases} \tag{3}$$

Figure 2: Data flow equations for MOD subproblems

CondLMOD

$CondIMOD(n, RA)$ denotes the set of fixed locations modified directly or through aliases by the assignment at program point n, given reaching alias RA. A *fixed-location* is an object name which is not a dereferenced pointer (e.g., object name `*p` is not a fixed location, but `p` is a fixed location [LRZ93].)

CondIMOD

For a procedure P and a reaching alias RA, $CondIMOD(P, RA)$ is the set of all fixed-locations modified by assignments in procedure P. Equation (1) of Figure 2 is the data flow equation for $CondIMOD$.

PMOD

$PMOD(P, RA)$ is the set of fixed-locations modified by a procedure P given reaching alias RA, including the effects of calls from within P. It is formed from local $CondIMOD$ information and $PMOD$ information propagated from procedures called by P. The data flow equation for $PMOD$ is given in Figure 2, Equation (2).

$Reach(call_Q, RA)$ represents the set of aliases reaching the entry of Q induced by the parameter bindings at the call or aliases that hold right before entering the called procedure. The function b_{call_Q} maps names from the called procedure Q to the calling procedure P and only returns fixed-locations.

CMOD

$CMOD$nRA, depending on the statement type, specifies the modification side effect of executing statement n, given a reaching alias RA at the entry of the procedure containing n. The data flow equation for $CMOD$ is given in Figure 2, Equation (3).

MOD

$MOD(n)$ summarizes the effects over all executions of n in procedure P for all reaching aliases of P. $MOD(P)$ summarizes the effects over all calls of P for all reaching aliases of P.

In the problem decomposition mentioned above, there are only two data flow problems: $ALIAS$ and $PMOD$, and all the others are just immediate unions of their subproblems. Thus, an incremental MOD algorithm for C programs needs to solve the $ALIAS$ and $PMOD$ problems incrementally. In the decomposition for FORTRAN by Banning [Ban79], aliases can be separated from MOD and their effect still computed correctly. That is because in FORTRAN, only procedure calls can create aliases and aliases created by a call hold throughout execution of the called procedure. But for C, because aliases vary intraprocedurally, $ALIAS$ is a problem with different solutions at each program point, each associated with a reaching alias. We feel that incrementalizing $ALIAS$ for arbitrary program changes will be very complicated; therefore, for this paper we assume that updated $ALIAS$ information is given after a program change. In the following section, we will show how the hybrid algorithm can be used to incrementalize the $PMOD$ problem.

Call graph $=< V, E >$
$V = \{P \mid \text{procedure } P \text{ in the program}\}$
$E = \{(P, Q) \mid P \in V \text{and } Q \in V \text{and there is a call site of procedure } Q \text{ in procedure } P\}$

Call-RA graph $=< V', E' >$
$V' = \{(P, RA) \mid RA \text{ reaches entry of procedure } P\ \}$
$E' = \{((P, RA), (Q, RA')) \mid (P, RA) \in V' \text{ and } (Q, RA') \in V' \text{ and}$
$\qquad P \text{ calls } Q \text{ and } RA' \in Reach(call_Q, RA)\}$

Figure 3: Call graph definitions

3 Reformulation of $PMOD$ for the Hybrid Algorithm

Before applying the hybrid algorithm to the $PMOD$ problem, we reformulate equation (2) for $PMOD$ to make it easier for factorization in the hybrid algorithm. Based on the new formulation, $PMOD$ is factored to make it suitable for solution by the hybrid algorithm. The new formulation uses the *call-RA graph* as the problem representation, which will be defined in the next subsection.

3.1 The Call Multigraph and Call-RA Graph

A *call multigraph* is a graph representation of the calling relation among procedures of a program, in which a node represents a procedure and an edge from node P to node Q represents a call from procedure P to procedure Q. Figure 3 shows definitions of a call-multigraph. A *call-RA graph* is basically a variation of a call graph, and can be obtained by augmenting the nodes in the original call graph with reaching aliases. In a call-RA graph, each node is a tuple of a procedure and an alias pair reaching its entry; each edge represents flow of information on a path from the entry of the calling procedure to the entry of the called procedure corresponding to a call site. Figure 3 also shows a definition of a call-RA graph.

For a simple program in Figure 4(a), its call graph and call-RA graph are given in Figures 4(b) and (c) respectively. Different nodes for a procedure indicate that the procedure is called with different reaching aliases. In Figure 4(c), node *main* denotes that procedure `main` is executed with null assumed alias $\emptyset$. Procedure `main` then calls procedure `P` by passing the address of `x` to a formal `a` of `P`. The parameter binding makes an alias pair `<*a,x>` to reach the entry of `P`. However, `x` is a local in `main`, and thus invisible to `P`. So we use a special symbol `nv`[2] [LR92] as an abstraction of the variables in the calling procedure which are invisible, but still accessible through their aliases in the called procedure. Then, we have `<*a,nv>` hold at the entry of `P`. We also keep the backbinding information for `nv` along the edge $e_1 = ((\texttt{main}, \emptyset), (\texttt{P}, < *\texttt{a}, \texttt{nv} >))$ by defining $nv\_backbind_{e_1}$ as a set of all object names in node (`main`,$\emptyset$) to which `nv` in node (`P`,`<*a,nv>`) can bind. Those invisible object names in $nv\_backbind_e$ could be either locals of calling procedure or an `nv` in source node. For example, given `<*a,nv>` at the entry of procedure `P`, i.e., node P, the call site of `Q` makes two alias pairs, `<*b,*nv>` and `<*b,nv>` at the entry of `Q`, reaching the entry of `Q`, and thus generates two call-RA graph nodes Q_1 and Q_2. Edges e_2 and e_3 denote this calling relationship from node P to node Q_1 and node Q_2 respectively. The `nv` in node Q_1 represents a local `a` in procedure `P`, and the `nv` in node Q_2 represents `nv` in node P. Figure 4(c) also shows the $nv\_backbind$ sets for the other edges.

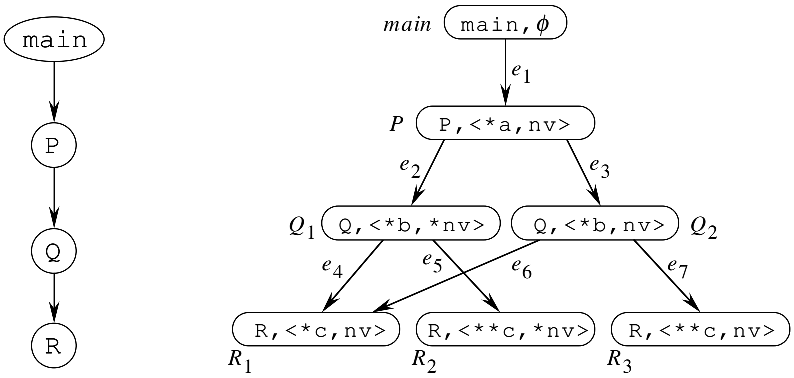
3.2 $PMOD$ on the Call-RA Graph

In equation (2), $PMOD$ of a procedure P with an assumed alias pair RA (i.e., node (P, RA) in the call-RA graph), is computed by taking the union of its $CondIMOD$ and $PMOD$ from its successors. Conceptually, $PMOD$ is computed over the call-RA graph. Examining equation (2), we can find that $CondIMOD(P, RA)$

[2] We abbreviate *non_visible* as `nv` in the given example.

```
main()
{ int x;
  P(&x);
}
P(int *a)
{
  Q(a);
}
Q(int *b)
{ int y;
  R(&b);
}
R(int **c)
{
  **c=...
}
```

(a)



(b)

Edge	e_1	e_2	e_3	e_4	e_5	e_6	e_7
nv_backbind	{x}	{a}	{nv}	{b}	{nv}	{b}	{nv}

Source,Destination	P,R_1	P,R_2	P,R_3
nv_backbind *	{ }	{a}	{nv}

(c)

(a) Source program (b) Call graph (c) Call-RA graph

Figure 4: An example of the call-RA graph

is associated with node (P, RA) in a call-RA graph, and the function $Reach(call, RA)$ is already "encoded" in the definition of edges of a call-RA graph. As for the function b_{call}, it maps object names from the scope of the called procedure to the scope of the calling procedure as follows:

- Maps global fixed-locations to themselves.
- Factors out all locals (including formals) of the called procedure.
- Maps *non_visible* in the called procedure to its corresponding fixed-locations and/or *non_visible* in the calling procedure.

Since global variables are visible in both the called and calling procedures, the name mapping of global variables is trivial. Parameter passing in C uses call-by-value only, so formal variables are just local variables

$$PMOD(P, RA) = CondIMOD(P, RA) \cup \bigcup_{e=((P,RA),(Q,RA'))\in E'} b(PMOD(Q, RA'), nv\_backbind_e) \quad (4)$$

$$nv\_backbind^*_{((P,RA),(R,RA''))} = \bigcup_{\substack{e = ((P, RA), (Q, RA')) \in E', \\ \text{and } \exists \text{ a } non\_visible \text{ binding} \\ \text{chain connecting } (Q, RA') \text{ and} \\ (R, RA'')}} nv\_backbind_e \quad (5)$$

$$PMOD(P, RA) = \bigcup_{\substack{(R, RA'') \text{ is} \\ \text{reachable}^3 \text{ from} \\ (P, RA)}} b(CondIMOD(R, RA''), nv\_backbind^*_{((P,RA),(R,RA''))}) \quad (6)$$

Figure 5: Reformulations of $PMOD$ equation using *nv_backbind*

$$LOC_1(P,RA) = CondIMOD(P,RA) \cup \bigcup_{\substack{e\,=\,((P,RA),(Q,RA''))\,\in\,E' \\ \text{and } Q\,\in\,Region(P)}} b(LOC_1(Q,RA')), nv\_backbind_e) \tag{7}$$

$$LOC_2(P,RA) = \bigcup_{\substack{((P,RA),(Q,RA'))\,\in\,E' \\ \text{and } (P,RA)\,\neq\,(Q,RA')}} \begin{cases} \{(P,RA)\} & \text{if } Q \neq Region(p) \\ LOC_2(Q,RA') & \text{otherwise} \end{cases} \tag{8}$$

$$LOC_3(P,RA) = \bigcup_{\substack{e\,=\,((P,RA),(Q,RA'))\,\in\,E' \\ \text{and } (P,RA)\,\neq\,(Q,RA')}} \begin{cases} \{(P,RA)\} & \text{if } Q \notin Region(P) \\ LOC_3(Q,RA') & \text{if } non\_visible \in nv\_backbind_e \\ \emptyset & \text{otherwise} \end{cases} \tag{9}$$

Figure 6: Factorization of $PMOD$ into local problems

with the values of actual parameters as their initial values. Local variables in the called procedure are inaccessible in the calling procedure; they are ignored in the mapping. Only the mapping of *non_visible* to its corresponding fixed-locations or *non_visible* in the calling procedure is necessary. In our call-RA graph, the *nv_backbind* set provides the required information.

Now $PMOD$ can be reformulated using *nv_backbind* as Equation (4) in Figure 5. The function b in the equation performs the same name mapping as the function b_{call} in equation (2). For the variables in $PMOD(Q,RA')$, $b(PMOD(Q,RA'),nv\_backbind_e)$ factors out all local variables of Q, maps global variables to themselves, maps *non_visible* variables to their corresponding variables in $nv\_backbind_e$, and only returns fixed-locations or *non_visible* variables.

We have seen $nv\_backbind_{((P,RA),(Q,RA'))}$ is used in the name-mapping for an immediate successor (Q,RA') of (P,RA). Using the same idea, we next want to derive a general function $nv\_backbind^*_{(P,RA),(R,RA'')}$ for any node (R,RA'') reachable from (P,RA), which represents the set of all object names in (P,RA) to which *non_visible* in (R,RA'') can bind. The set $nv\_backbind^*_{(P,RA),(R,RA'')}$ can be computed as equation (5) in Figure 5. In the equation, a *non_visible* binding chain is either a path $(e_1, e_2, \ldots, e_k)$ with $k \geq 1$ in the call-RA graph such that $non\_visible \in nv\_backbind_{e_i}$ for i=1, 2, ..., k or null. For a node (R,RA'') reachable from (P,RA), *non_visible* in (R,RA'') can map to a variable in (P,RA) only if the *non_visible* variable can map to another *non_visible* in an immediate successor (Q,RA') of (P,RA) (i.e., a *non_visible* binding chain from (Q,RA') to (R,RA'')) and the variable is in $nvb(P,RA)(Q,RA')$. For example, in Figure 4(c), $nv\_backbind^*_{(P,R_2)}$ is {`a`}, and $nv\_backbind^*_{(P,R_1)}$ is $\emptyset$. This is because there is a *non_visible* binding chain (e_6) connecting P and R_2, but there is no such a chain between a successor of P and R_1.

With *nv_backbind**, equation (4) for $PMOD$ can be further reformulated as equation (6) in Figure 5. The original data flow problem in equation (2) for $PMOD$ is now a reachability problem on the call-RA graph as in equation (6); that is, $PMOD(P,RA)$ collects $CondIMOD(R,RA'')$ of all reachable (R,RA'') and maps them into the scope of procedure P. In equation (5), *nv_backbind** is also defined by reachability in terms of *non_visible* binding.

3.3 Factorization of $PMOD$ for the Hybrid Algorithm

The hybrid algorithms partition a flow graph into single entry regions; the resulting reduced graph is acyclic. Intuitively, the hybrid algorithms use iteration techniques within regions, and elimination techniques in their information propagation between regions. They rely on factoring the data flow solution on each region into internal and external parts which are solved separately on the region [MR90]. For each region, we define

[3] Reachability is reflexive, i.e., (P,RA) is reachable from (P,RA).

$$nv\_backbind^*_{((P,RA),(X,RA''))} = \bigcup_{\substack{e = ((P,RA),(Q,RA')) \in \\ E' \text{ and } Q \in Region(P) \text{ and} \\ (X,RA'') \in LOC_3(Q,RA')}} nv\_backbind_e \qquad (10)$$

$$\begin{aligned} PMOD(P,RA) &= LOC_1(P,RA) \cup \qquad (11) \\ & \bigcup_{(X,RA') \in LOC_2(P,RA)} b\Big(\bigcup_{\substack{e = ((X,RA'),(H,RA'')) \in \\ E' \text{ and } H \notin Region(P)}} b(PMOD(H,RA''), nv\_backbind_e), \\ & \qquad nv\_backbind^*_{((P,RA),(X,RA'))}\Big) \end{aligned}$$

Figure 7: Integration of local problems to obtain the final $PMOD$ solution

some local problems which capture the internal data flow information within the region and the local effects by and on external information.

For the $PMOD$ problem, the hybrid algorithm first partitions the call multigraph into single entry regions, and obtains an acyclic reduced graph. For each region, we factor the $PMOD$ problem into three local problems: LOC_1, LOC_2, and LOC_3. The global $PMOD$ solution of a node in a region can be recovered from the solutions of these local problems. Intuitively, for a node in a region, LOC_1 summarizes the local effects by solving the $PMOD$ problem restricted to the region. LOC_2 recognizes which exit nodes of the region are reachable from that node and thus knows that the external information arriving at those exit nodes should be propagated to that node. However, such information propagation requires name-mapping for variables, especially for *non_visible* variables, from the exit nodes to the target node, and LOC_3 is used to derive the name-mapping for *non_visible* variables.

First, LOC_1 can be computed using equation (7) in Figure 6. It is easy to see that equation (7) is a restricted instance of equation (4). $LOC_1(P,RA)$ summarizes the effects of $CondIMOD$ of all reachable nodes from (P,RA) within $Region(P)$.

Unlike a classical data flow problem (e.g., Reaching Definitions problem) which generally has some variables to summarize the external information in the corresponding problem, $PMOD$ is an information collecting process and thus we need to define a local problem which describes how the external information can be incorporated into the global solution for a node in a region. For each node, exit nodes reachable from it are recognized. The information arriving at these exit nodes is then gathered and mapped back to the scope of the node. $LOC_2(P,RA)$, shown as equation (8), computes the set of exit nodes reachable from (P,RA).

In equation (6), we need to determine if a node (R,RA'') is reachable from another node (Q,RA') through a *non_visible* binding chain. Thus, we define $LOC_3(P,RA)$ as the set of exit nodes which are reachable from (P,RA) through a *non_visible* binding chain, and it can be computed as in equation (9). Using $LOC_3(P,RA)$, we can rewrite equation (5) for $nv\_backbind^*_{((P,RA),(X,RA''))}$ as equation (10) in Figure 7.

Combining the solutions to those local problems and the external information propagated from other regions, we are able to recover the global $PMOD$ solution for a node (P,RA) using equation (11). In the equation, $LOC_1(P,RA)$ captures the local effect of the region containing (P,RA), $PMOD(H,RA'')$ is the $PMOD$ information coming from other regions, $LOC_2(P,RA)$ specifies which exit nodes are reachable from (P,RA) and thus the external information arriving at those exit nodes should be collected, and $nv\_backbind^*$, computed based on $LOC_3(P,RA)$, is used to map the incoming external information back to (P,RA).

The hybrid algorithm first computes the solution to local problems LOC_1, LOC_2, and LOC_3 using iteration within each region. Then, in a reverse topological order of the reduced graph, $PMOD$ information is computed at the entry node of a region, and propagated to its predecessor regions. After inter-region information propagation, the external $PMOD$ information is mapped back to each node in a region and combined with the local effect to obtained its global $PMOD$ solution.

4 Incremental $PMOD$ Algorithm

When a change is made to the call-RA graph, we first determine its type and location. Then, we use the incremental version of the hybrid algorithm to update the solutions:

1. Update the decomposition and topological order, if necessary.
2. Update local information and local problems within changed (or new) regions.
3. Update solutions to the local problems within changed (or new) regions.
4. Propagate between regions if the information to be propagated changes.
5. Propagate the updated external information to the nodes of a region, and update their global solutions.

The incremental hybrid algorithm improves the efficiency of updating by exploiting the locality of a change. The effect of a change could be local, as within a region, and no work may have to be done on the other regions. Another possibility is that the effect of a change just covers a small subset of regions, and the former solutions in the other regions remain valid. After the local solutions are updated, if the information to be propagated to the other regions changes, it should be repropagated. The global solutions in those regions receiving updated external information are then recomputed. Next, we will consider two classes of changes: non-structural and structural changes, and analyze how to update the solutions to reflect these changes. The changes we discuss in this section are in terms of the call-RA graph. A change in the aliases reaching a call may cause structural changes (and/or non-structural changes) to the call-RA graph, whereas this would be considered a non-structural change on the call multigraph, because reaching aliases are not parts of call multigraph nodes. For example, in Figure 8, we add a statement `b=&y` into the source program, marked by shadow, and this results in:

- a change in the set of variables modified by procedure Q (i.e., changes to $CondIMOD(Q_1)$ and $CondIMOD(Q_2)$), and
- a change in the aliases reaching the call of R, which causes deletion of an old edge e_5, addition of a new edge e_8, and change to the nv_backbindassociated with edge e_7.

Figure 8(b) shows the resulting call-RA graph. However, this change does not cause any change to the edges in the call multigraph.

4.1 Non-structural Changes

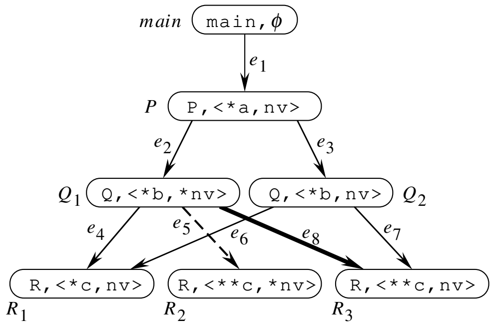
Non-structural changes only change the local information, like $CondIMOD$ and *nv_backbind*; the structure and region composition of the call-RA graph remain the same. We first consider changes of information associated with nodes. When the values of $CondIMOD$ at some nodes are changed, simply restarting iteration does not always yield a safe solution. Restarting iteration can get a correct solution only if the old solution is a safe initial estimate; otherwise, all or part of the solutions must be re-initialized to a safe initial value [RMP88]. We call those changes which can be accommodated by restarting iteration, additive.

Change CondIMOD
The algorithm $ModifyCondIMOD$ in Figure 9 handles changes of $CondIMOD$.

Only the LOC_1 information is affected by the change of $CondIMOD$. Step 2 checks which changes are additive and does the necessary re-initialization. A $CondIMOD$ change is additive if the set of global and *non_visible* variables of the new $CondIMOD$ is a superset of that of the old $CondIMOD$. Thus, if a change of $CondIMOD$ is additive, then the old LOC_1 solution is a safe initial estimate, and restarting iteration

```
main()
{ int x;
  P(&x);
}
P(int *a)
{
  Q(a);
}
Q(int *b)
{ int y;
  b=&y;
  R(&b);
}
R(int **c)
{
  **c=...
}
```

(a)



Edge	e_1	e_2	e_3	e_4	e_6	e_7	e_8
nv_backbind	{x}	{a}	{nv}	{b}	{b}	{y}	{y}

(b)

Node	*main*	*P*	Q_1	Q_2	R_1	R_2	R_3
Old *PMOD*	{x}	{nv}	{ }	{nv}	{ }	{ }	{nv}
New *PMOD*	{ }	{ }	{b,y}	{b,y}	{ }	{ }	{nv}

(c)

(a) Source program (b) Call-RA graph (c) $PMOD$ solutions

Figure 8: An example of source change

Algorithm1 ModifyCondIMOD

1. Worklist = $\{(P, RA) \mid CondIMOD(P, RA)$ has changed$\}$
2. For those nodes in Worklist whose $CondIMOD$ change is not additive, re-initialize LOC_1 of those nodes and all their ancestors in the same region to their respective $CondIMOD$s.
3. While Worklist is not empty
   - 3.1 Delete (P, RA) from Worklist.
   - 3.2 Re-compute $LOC_1(P, RA)$ using equation (7).
   - 3.3 If $LOC_1(P, RA)$ changes, then add predecessors of (P, RA) in $Region(P)$ to Worklist.
4. *PropagatePMOD*, starting at the changed region with the largest topological order.
   - 4.1 In a reverse topological order of the reduced graph, compute the $PMOD$ information at entry nodes using equation (11), and if the information changes, then propagate it to their predecessor regions. The inter-region propagation continues until there are no changes to the data flow solution at entry nodes.
   - 4.2 Propagate the updated $PMOD$ information to each node in a region whose exit node solution has changed, and combine it with the local effects to compute the new $PMOD$ solution at each node.

Figure 9: Algorithm for $ModifyCondIMOD$

can yield a correct solution. Otherwise, re-initialization is needed before restarting iteration, and a simple and safe initial estimate can be made by re-initializing LOC_1 at the influenced nodes (i.e., changed nodes and their region ancestors) to their own $CondIMOD$. Iteration using a worklist is then applied to compute the new local solutions in step 3. The worklist used here is prioritized by the reverse topological sort order on the call-RA graph. Step 4 updates the final $PMOD$ solutions in the influenced regions based on the updated local solutions. In step 4.1, starting at the changed region with the largest topological order and proceeding in a reverse topological order on the reduced graph, the $PMOD$ information at entry nodes of the changed regions is computed, and if the $PMOD$ information changes, it is propagated to the predecessor regions. The inter-region propagation continues until there are no changes at entry nodes. In step 4.2, the updated external $PMOD$ information is then propagated to each node of a influenced region, and combined with the local effect to obtain the global $PMOD$ solution at that node. One thing worth mentioning is that not every node in a region with new incoming information needs to re-compute their $PMOD$. Only those nodes whose LOC_3 contains an exit node with new incoming information need to re-compute $PMOD$. We will reuse step 4, $PropagatePMOD$, of algorithm $ModifyCondIMOD$ in other updating algorithms in this section.

Change nv_backbind
The change of $nv\_backbind_e$ associated with an edge e is another kind of non-structural change. Only the LOC_1 and LOC_3 solutions are affected by this kind of change. The algorithm $ModifyBackbind$ in Figure 10 deals with the change of *nv_backbind* case by case.

If the edge whose *nv_backbind* changes is inter-region, then we just call procedure $PropagatePMOD$ to redo inter-region propagation and intra-region propagation to update the $PMOD$ solutions in the influenced regions. If the changed edge, say $e = ((P, RA), (Q, RA'))$, is within a region, then we have to consider whether the change of *nv_backbind* is additive. A change of *nv_backbind* is additive if the change does not remove *non_visible* from *nv_backbind*. If the change is additive, then the old solutions to LOC_1 and LOC_3 are still safe initial estimates, and restarting iteration (in step 2.2) from these old solutions can yield correct solutions. But, if the change removes *nv_backbind* from $nv\_backbind_e$, these old solutions are not safe initial estimates. Thus, before restarting iteration, (in step 2.1.1) for (P, RA) and every ancestor of (P, RA), say (R, RA''), $LOC_1(R, RA'')$ is re-initialized by retaining only globals in current LOC_1 and adding $CondIMOD(R, RA'')$ to the new LOC_1. LOC_3 is re-initialized to an empty set.

Algorithm2 ModifyBackbind($e = (P, RA), (Q, RA')$)

1. If e is an inter-region edge, then

   1.1 Call $PropagatePMOD$, starting at $Region(P)$.

2. If e is an intra-region edge:

   2.1 If the change of $nv\_backbind_e$ is not additive:

   2.1.1 For (P, RA) and every ancestor of (P, RA) in the region, say (R, RA'')

   2.1.1.1 Re-initialize $LOC_1(R, RA'')$ by retaining only global variables in current LOC_1, and adding $CondIMOD(R, RA'')$ to the new LOC_1.

   2.1.1.2 Re-initialize LOC_3 to an empty set.

   2.2 Restart iteration on the region to get the solutions to LOC_1 and LOC_3 using equations (7) and (9) respectively

   2.3 Call $PropagatePMOD$, starting at $Region(P)$.

Figure 10: Algorithm for $ModifyBackbind$

Algorithm3 DeleteAnEdge(e)

1. If e is an inter-region edge $((X, RA), (H, RA'))$:

   1.1 Update the $PMOD$ information arriving at exit node (X, RA).

   1.2 Call $PropagatePMOD$, starting at $Region(X)$. Note: Removal of the only inter-region edge leaving from node (X, RA) would make it no longer an exit node; that is, (X, RA) should be removed from LOC_2 and LOC_3 at all nodes in $Region(X)$. Removal of the only incoming edge of $Region(H)$ would make it unreachable from the root.

2. If e is an intra-region edge $((P, RA), (Q, RA'))$:

   2.1 For (P, RA) and every ancestor of (P, RA) ins the region,

   2.1.1 Re-initialize LOC_1 to its $CondIMOD$.

   2.1.2 Re-initialize LOC_2 to an empty set.

   2.1.3 If *nv_backbind*$_e$ contains *nv_backbind* then re-initialize LOC_3 to an empty set, otherwise do nothing.

   2.2 Iterate LOC_1, LOC_2, and LOC_3 on the region to their solutions using equations (7), (8) and (9) respectively.

   2.3 Call $PropagatePMOD$, starting at $Region(P)$.

Figure 11: Algorithm for *DeleteAnEdge*

4.2 Structural Changes

Structural changes are those which can change the shape or hybrid algorithm decomposition of the call-RA graph, and thus are harder to handle than non-structural changes. There are a lot of ways to generate structural changes. An edge may be added or deleted if the aliases reaching the entry of a procedure change. Changing a parameter binding, adding or deleting a call, and adding or deleting aliases reaching a call all may change the aliases reaching the entry of the called procedure, and thus may cause structural changes. Figure 8 shows a structural change made by changing the aliases reaching a call (i.e., addition of edges e_6 and e_7). In the following discussion, we will consider the changes of edge deletion and addition. Changes to nodes can be converted to a sequence of edge changes. Deleting a node can be treated as deleting all edges to and from that node, and adding a node as adding an edge whose target is an isolated node.

Delete an edge

In general, deleting an inter-region edge causes less trouble than deleting an intra-region edge, since it just changes the incoming information to the target region, and the local solutions are still valid. All we need to do is to redo inter-region propagation between those regions whose incoming information changes, and then intra-region propagation within each region with new incoming information. Part 1 of the algorithm *DeleteAnEdge* in Figure 11 deals with deleting an inter-region edge; this can sometimes render an exit node no longer an exit, and then the node should be removed from LOC_2 and LOC_3 for all nodes in the region. Deleting an inter-region edge also can make a region unreachable from the root of the call-RA graph.

Part 2 in algorithm *DeleteAnEdge* summarizes what we should do while deleting an edge $e = ((P, RA), (Q, RA'))$ within a region. In this case, LOC_1, LOC_2, and LOC_3 at influenced nodes (i.e., (P, RA) and its ancestors) should be re-initialized as in step 2.1 before restarting iteration. Then, we follow the usual procedures to update the local solutions and global $PMOD$ solutions. Sometimes, edge removal within a region could make the region further decompose-able. In parallel data flow analysis, decomposition of large regions is beneficial to load balancing among processors.

Add an edge

Algorithm4 AddAnEdge($e = ((P, RA), (Q, RA'))$)

1. If e is an intra-region edge:

   1.1 Iterate LOC_1, LOC_2 and LOC_3 to their new solutions using equations (7), (8) and (9) respectively.

   1.2 Call $PropagatePMOD$, starting at $Region(P)$.

   Note: Adding (or connecting) a new node to the call-RA graph can be considered as adding an new edge whose target is an isolated node.

2. If e is an inter-region edge:

   2.1 If e is a forward edge and Q is the entry of a region.

   2.1.1 If (P, RA) was not an exit node before, then

   2.1.1.1 Iterate LOC_2 and LOC_3 to new solutions using equation (8) and (9) respectively to account for (P, RA) becoming an exit.

   2.1.2 Include the $PMOD$ information coming from e in exit node (P, RA).

   2.1.3 Call $PropagatePMOD$, starting at $Region(P)$.

   2.2 Otherwise: (A new decomposition is needed because e is a back edge or branches into the interior of a region)

   2.2.1 Determine the minimal set of regions, say $R_{i_1}, R_{i_2}, ..., R_{i_k}$, to be merged in order to make edge e become an edge within the resulting region.

   2.2.1.1 Find the immediate common dominating region of $Region(P)$ and $Region(Q)$.

   2.2.1.2 The regions between the dominating region and $Region(P)$ and $Region(Q)$ form the set of regions to be merged.

   2.2.2 Call $MergeRegions$ on $R_{i_1}, R_{i_2}, ..., R_{i_k}$, and return the resulting region R.

   2.2.3 Consider the problem as adding an edge within region R, and apply part 1 of the algorithm to solve the problem.

Figure 12: Algorithm for *AddAnEdge*

Adding an edge within a region is always an additive change with respect to the local problems, and can be accommodated by restarting iteration from previous solutions. The decomposition of the call-RA graph does not change at all. Part 1 of algorithm *AddAnEdge* in Figure 12 is for handling edge-addition within a region.

We now consider the case of adding an inter-region edge. Adding an edge, especially an inter-region edge, seems potentially to have more impact on the structure of the call-RA graph. In the hybrid algorithm, the decomposition of the flow graph should have single-entry property; that is, only one entry for each region, and the reduced graph should be acyclic. As a new edge connecting two regions is added to the flow graph, these properties could be violated. In step 2.1, we check if the newly added edge is a forward edge and its target is an entry node. If so, no new decomposition is needed. In this case, we further check if a new exit node is created at the source region of the edge. If so, LOC_2 and LOC_3 at the source region should be re-iterated (from old solutions) to their new solutions which cover the fact that a new exit is made. After the local solutions are updated, in step 2.1.2 the external $PMOD$ information coming along the new edge is included at the exit node. Then, in step 2.1.3, we redo inter-region propagation and intra-region propagation to update the $PMOD$ solutions.

If the newly added edge is a backward edge or its target is not an entry node, then a new decomposition of the call-RA graph is necessary to recover the acyclicity of the reduced graph and the single-entry property for regions. One way to implement the re-decomposition is to merge several regions into a big one so that the edge to be added becomes an intra-region edge within the resulting region. Of course, the resulting region

Algorithm5 MergeRegions($M = R_{i_1}, R_{i_2}, ..., R_{i_k}$)

1. Make a region $R = R_{i_1} R_{i_2}, ..., R_{i_k}$.
2. Determine the exit nodes for the resulting region, denoted as a set $EXIT(R)$.
3. Adjust LOC_1, LOC_2 and LOC_3 to make them consistent with the resulting region:
   - 3.1 In a reverse topological order among $R_{i_1}, R_{i_2}, ...$, and R_{i_k}, perform inter-region propagation (from entries to exits) on the LOC_1, LOC_2 and LOC_3 information using equations (12), (13), and (14) respectively.
   - 3.2 Perform intra-region propagation to make each node in the resulting region R have correct LOC_1, LOC_2 and LOC_3 solutions with respect to the new region using equations (12), (13), and (14) respectively.

Figure 13: Algorithm for $MergeRegions$

should have only one entry and the new reduced graph should be acyclic. In region merging, the fewer regions involved the better. Step 2.2.1 selects such a set of regions by first finding the immediate common dominator of the target and destination regions of the newly added edge, and then making those regions between them the set of regions to be merged. Given such a set of k regions, say $M = \{R_{i_1}, R_{i_2}, ..., R_{i_k}\}$, algorithm $MergeRegions$ in Figure 13 merges them into a large region and updates the LOC_1, LOC_2 and LOC_3 solutions at each node to make them consistent with the resulting region. In step 1, those regions are marked as a single region R. Next, the exit nodes for the composite region R are recognized and denoted as $EXIT(R)$. Then in step 3, the LOC_1, LOC_2 and LOC_3 solutions are updated.

To achieve better efficiency in updating the local solutions, we try to reuse the old solutions as much as possible. In updating LOC_1, the new LOC_1 solution at a node in one region of M has to include the local effects of the other reachable regions of M. This can be accomplished by applying the inter-region and intra-region propagation to the LOC_1 information among those regions in M. That is, in a reverse topological order of the regions in M, combining with the incoming LOC_1 information at exit nodes, the LOC_1 solution at the entry node of a region is computed by applying equation (7) on LOC_1, but restricted to the regions in M. Equation (12) in Figure 14 shows the resulting equation.

The updated LOC_1 information at the entry node of a region is then propagated to the exit nodes of the predecessor regions in M. The LOC_1 solution at the other interior nodes of the regions in M is updated by propagating the incoming LOC_1 information intra-regionally. This updating process is actually an application of the inter-region and intra-region propagation we use in the computation of $PMOD$. We can apply the same technique to update LOC_2 and LOC_3 with equation (13) and equation (14) respectively. The exit nodes in current LOC_2 and LOC_3 which are not exit nodes in the resulting region should not be included in new LOC_2 and LOC_3. The Region function in equation (12) - (14) is in terms of the old decomposition.

After the region-merging step, the problem of adding an inter-region edge now becomes adding an edge within the resulting region, and can be solved by simply applying part 1 of algorithm *AddAnEdge*.

5 Source Change Analysis

In this section, we will discuss how the changes of the source program affects the call-RA graph. In general, there are three types of statements in a procedure: assignment statements, function calls, and flow-controlling statements. All the three types of statement may change the aliasing formation reaching their subsequent statement. Such changes on the aliasing information reaching an assignment statement may change the side effect of the assignment, and thus change *CondIMOD* for the procedure. Beside, the changes reaching a

$$newLOC_1(P,RA) = LOC_1(P,RA) \cup \bigcup_{(X,RA') \in LOC_2(P,RA)} b\Big(\bigcup_{\substack{e = ((X,RA'),(H,RA'')) \\ \in E',\ H \notin Region(P), \text{ and} \\ Region(H) \in M}} b(newLOC_1(H,RA''), nv\_backbind_e), \; nv\_backbind^*_*((P,RA),(X,RA')) \Big) \tag{12}$$

$$newLOC_2(P,RA) = LOC_2(P,RA) \cap EXIT(R) \cup \bigcup_{(X,RA') \in LOC_2(P,RA)} \; \bigcup_{\substack{e = ((X,RA'),(H,RA'')) \\ \in E',\ H \notin Region(P), \text{ and} \\ Region(H) \in M}} newLOC_2(H,RA'') \tag{13}$$

$$newLOC_3(P,RA) = LOC_3(P,RA) \cap EXIT(R) \cup \bigcup_{(X,RA') \in LOC_3(P,RA)} \; \bigcup_{\substack{e = ((X,RA'),(H,RA'')) \\ \in E',\ H \notin Region(P), \\ \text{and } Region(H) \in M, \text{ and} \\ non\_visible \in \\ nv\_backbind_e}} newLOC_3(H,RA'') \tag{14}$$

Figure 14: new LOC equations

call site may change the aliasing information reaching the entry of the called procedure and the parameter binding information. Next we will discuss in details the call-RA graph changes corresponding to a source code change.

Assignment Statements An assignment statement with pointer involved is called a pointer assignment, otherwise, it is called a non-pointer assignment. A non-pointer assignment statement won't change the aliasing information reaching the statement, and will just pass it to the subsequent statements. Thus changes to a non-pointer assignment may cause changes to $CondIMOD$ of the containing procedure only. These changes to the call-RA graph are non-structural. If the statement we change is a pointer assignment, then it may cause non-structural and/or structural changes to the call-RA graph.

The impact on the $PMOD$ solution made by the change of an assignment does not only depend on what type of statement (pointer assignments or non-pointer assignments) it is, but also on what variables it modifies and how the containing procedure is used. For example, if we remove a non-pointer statement that modifies a local variable only, then such a source change will affect the $PMOD$ solutions for the containing procedure only. However, if the statement we removed modifies a global variable that is not modified in any other points, then doing so may cause changes to the $PMOD$ solutions for all its direct and indirect calling procedures. The more procedures this procedure is called by, the larger impact it causes.

Actual-Formal Bindings Changing actual-formal bindings at a call site may change the set of variables to which a *non_visible* variable in the called procedure can bind, and thus change the values of *nv_backbind* associated with the edges corresponding to the call. Of course, it may also cause changes in the reaching aliases for the called procedure. This could result in edge deletion and edge addition in the call-RA graph.

Procedure Calls Insertion of a call may cause new aliases to reach the entry of the called procedure. That is, new edges to new nodes in the call-RA graph may be created. On the other hand, deletion of a call may cause the opposite effect (i.e., edge deletion in the call-RA graph.) Basically, the impact on the call-RA graph by inserting/deleting a function call depends on the importance of the called function in the program. For example, if we remove the only call to a key function in the program, it sometimes disconnects many other functions invoked by that function, and thus causes lots of edges to be deleted. On the other hand, deleting a call to a function that is a leaf node in the call-RA graph causes much less impact on the graph. Any change at a call site may also kill and/or create aliases reaching those statements after the call site, so it may cause all kinds of call-RA graph changes.

In general, a change of the source program may cause a sequence of changes, structural and/or non-structural, to the call-RA graph. The algorithms mentioned in the previous section can then be invoked to handle these changes of the call-RA graph one by one. For the example in Figure 8, algorithm $ModifyCondIMOD$ is called to handle changes in $CondIMOD$ for Q_1 and Q_2, algorithm $ModifyBackbind$ to handle the *nv_backbind* change of edge e_7, algorithm *DeleteAnEdge* to handle deletion of edge e_5, and algorithm *AddAnEdge* to handle addition of edges e_8. Figure 8(c) shows both the old and the updated $PMOD$ solutions.

As we can see, an arbitrary, even small, change of the source program could result in a wide range of changes to the call-RA graph. Designing an effective incremental analysis tool, we should find which source code changes are reasonably common and can be handled efficiently.

By examining the algorithms, we can find that each of our proposed updating algorithms is composed of two major steps: local solution updating and inter-region information propagation. Although any change to the call-RA graph can be handled by applying the relevant algorithm, in the presence of multiple changes to the call-RA graph, we can achieve better efficiency by processing multiple changes as a whole. That is, instead of applying the relevant algorithms to handle the changes individually, for intra-region changes, we can combine the local solution updating steps of the required algorithms, and update the local solutions for each region in one single iteration. Then, one pass of inter-region information propagation is applied to handle inter-region changes and propagate updated information among regions. In handling multiple changes, this method avoids multiple iterations within each region and multiple passes of inter-region information propagation.

6 Conclusions and Future Work

In this paper, we apply the hybrid algorithm to incrementalize $PMOD$. We first reformulate the $PMOD$ problem on the call-RA graph. Based on this formulation, we factor $PMOD$ problem into three local problems which enables us to obtain the global $PMOD$ solution for each node in the call-RA graph. We then consider two classes of changes to the call-RA graph (non-structural and structural.) We discuss the necessary updates for these changes. We have experimented how a simple source change like deleting a statement may affect the call-RA graph and the $PMOD$ solution. In general, the impact on the call-RA graph and the $PMOD$ solution is small, and this result reveals a great opportunity for the incremental approach. We also empirically study how different kinds of statements we deleted affected the call-RA graph and the $PMOD$ solution.

What we want to do next is to prototype those ideas in this paper, and do some empirical work to see the overhead and performance improvement of the incremental algorithm. We also want to study which kinds of changes are often made in different software analyzing tools. We believe that the more we understand about which changes are common, the more efficiency improvements we can achieve in our incremental data flow analysis. We wish to apply these ideas to the incrementalization of $ALIAS$. That is, instead of handling an arbitrary change, we want to find a subset of changes with some nice property so that they can be handled easily.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, 1986.

[Ban79] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages,* pages 29-41, January 1979.

[BR90] M. G. Burke and B. G. Ryder. A Critical Analysis of Incremental data flow analysis algorithms. *IEEE Transactions on Software Engineering,* pages 723-728, 1990.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* pages 47-56, June 1988

[CK87] K. Cooper and K. Kennedy. Complexity of interprocedural side- effect analysis. Comp. Sci. Dept. TR87-61, Rice University, October 1987.

[CK89] K. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages,* pages 49-59, January 1989.

[Lan92] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems,* 1(4), December 1992.

[LR91] W. Landi and B. G. Ryder. Pointer-induced aliasing: a problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages,* pages 93-103, January 1991.

[LR92] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation,* pages 235-248, June 1992.

[LRZ93] W. Landi, B. G. Ryder and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation,* pages 56-67, June 1993.

[MR90] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages,* pages 184-196, January 1990.

[MR91] T. J. Marlowe and B. G. Ryder. Hybrid incremental alias algorithms. In *Proceedings of the Twenty-fourth Hawaii International Conference on System Sciences,* Volume II, Software, pages 428-437, January 1991.

[MRB95] T. J. Marlowe, B. G. Ryder, and M. G. Burke. Defining flow sensitivity in data flow problems. LCSR-TR-249, June 1995

[PS89] L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering,* vol. 15, no. 12, pages 1537-1549, December 1989.

[RMP88] B. G. Ryder, T. J. Marlowe, and M. C. Paull. Conditions for incremental iteration: examples and counterexamples. *Science of Computer Programming,* 11(1), pages 1-15, October 1988.

[RP86] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Survey,* vol. 18, no. 3, pages 277-316, September 1986.

[RP88] B. G. Ryder and M. C. Paull. Incremental data flow analysis algorithms. *ACM Transactions on Programming Languages and Systems,* vol. 10, no. 1, pages 1-50, January 1988.

[Ryd89] B. G. Ryder. Ismm: Incremental software maintenance manager. In *Proceedings of the IEEE Computer Society Conference on Software Maintenance,* pages 142-164, October 1989.