Efficient Computation of Flow-Insensitive Interprocedural Summary Information — A Correction

Keith D. Cooper
Ken Kennedy

Department of Computer Science[†]
Rice University
Houston, Texas 77251-1892

1. Introduction

In this note we present the correction for a significant error contained in our paper ''Efficient Computation of Flow Insensitive Interprocedural Summary Information'', which was published in the *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction* [CoKe 84]. That paper presented an algorithm for solving flow-insensitive interprocedural side effects problems, as exemplified by the problem of determining the variables that may be modified as a side effect of a procedure call in a given program. These problems have been widely discussed in the literature [Spil 71, Alle 74, Bart 78, Bann 79, Myer 80, CoKe 84, Burk 84, CaRy 86].

The fundamental insight in the SIGPLAN '84 paper was that the problem can be subdivided into two subproblems: the side effects to parameters passed by reference and the side effects to variables passed as global variables. Each of these subproblems can then be solved using algorithms adapted from single-procedure data-flow analysis.

The actual formulation of the subdivision was flawed in two ways. First, a step was omitted in the algorithm for computing the side effects to reference parameters. This oversight can be easily corrected. However, the second problem is more fundamental. The SIGPLAN '84 paper asserted that the two subproblems can be solved independently and the solutions merged later. This is not the case. To correctly compute the answer, one must solve the reference parameter problem *first*, then use the result in the solution of the global variable problem.

These problems have been noted in the literature [Burk 84, CaRy 86] and in private communications [Alle 87, Ryde 87]. A number of fixes have been proposed [Burk 84, Ryde 87]. The correction we present here, which has been incorporated into a revised version of the paper prepared for submission to a journal [CoKe 87], is essentially equivalent to one developed independently by Carroll and Ryder [Carr 87, Ryde 87].

Rather than attempt to patch the text of the original paper, we present below a new formulation of the problem decomposition that avoids the errors in the original, then we relate the solution method to the material in the original paper. The interested reader is encouraged to read the full revised treatment, which is available as a technical report [CoKe 87].

[†] This research has been supported by the National Science Foundation and by IBM Corporation.

2. The Problem

The problem is to determine, for each call site, which variables can have their values modified by its execution. To represent this information concisely, we annotate each call site s in the program with the set MOD(s) defined as follows. For a call site s and a variable v:

$$v \in \text{MOD}(s) \quad \leftrightarrow \quad \text{executing } s \text{ may change the value of } v$$

The problem of computing this set is called *flow-insensitive* because it must conclude that a procedure call has a side effect, like $v \in$ MOD(s), if that side effect can occur on *some* path through the called procedure or any procedure that it, in turn, invokes. By contrast, the *flow-sensitive* formulations of these problems would conclude that the call has the side effect if and only if the analyzer can determine that the side effect occurs on *every* path through the called procedure and all procedures that it, in turn, calls.

Rather than compute MOD sets directly, Banning [Bann 79][1] breaks the problem down into component parts. Aliasing is ignored until late in the computation; the method assumes that simple sets of alias pairs are available for each procedure. Define:

DMOD The computation of MOD(s) is complicated by aliasing effects. The treatment can be simplified by first computing DMOD(s), the set of variables that may be modified by execution of s, ignoring any aliasing effects in the procedure containing s, and factoring aliasing in later. In other words, MOD(s) can be computed by adding to DMOD(s) any variable that may be aliased to a member of DMOD(s). We call DMOD(s) the *directly modified set* for statement s.

GMOD The problem can be further simplified by observing that DMOD for any call site can be easily computed once we determine, for each procedure p in the program, a set GMOD(p) that contains all variables, including variables local to p, that might be modified as the result of an invocation of p. We call GMOD(p) the *generalized modification set* for procedure p.[2] Once it is computed, DMOD for any call site that invokes p can be computed by identifying the variables known at the call site that are bound by the call to variables in GMOD(p).

The virtue of these observations is that GMOD(p) can be formulated as the solution to a system of data-flow equations on the call graph. To introduce this formulation, we need some more definitions.

LOCAL For a procedure p, LOCAL(p) contains the names of all variables declared in p.

FORMAL For a procedure p, FORMAL(p) contains the names of p's formal parameters. Note that FORMAL(p) $\subseteq$ LOCAL(p).

LMOD For a statement s, LMOD(s) contains those variables that might be modified by an execution of s, exclusive of any procedure calls in s. We call LMOD(s) the *locally modified set* for statement s.

IMOD For a procedure p, IMOD(p) contains those variables that might be modified by an execution of p, exclusive of any procedure calls in p. We call LMOD(p) the *initially modified set* for procedure p. Note that

$$\text{IMOD}(p) = \bigcup_{s \in p} \text{LMOD}(s).$$

---

[1] This formulation is based on Banning's, but with different notation.

[2] In the SIGPLAN '84 paper we restricted GMOD to contain no local variables except for formal parameters. However, a careful reread- ing of Banning's original paper shows that he intended local variables to be included (except for the main program, whose GMOD set was defined to be empty) so we have revised our definition to be consistent with his. In fact, this definition leads to a much simpler formulation of the equations. The additional local variables in GMOD do not affect the MOD sets, since these variables are filtered out in the DMOD calculation. A clever implementation might avoid introducing them at all; however, we consider this an implementation de- tail.

We are now ready to introduce the system of equations for GMOD(p).

$$\text{GMOD}(p) = \text{IMOD}(p) \cup [\bigcup_{e=(p,q)} b_e(\text{GMOD}(q))] \tag{1}$$

Here, b_e is a function that maps names from q into names from p according to the name scoping and parameter binding that happens at the call site $e = (p, q)$. We call $b_e(x)$ the *projection* of x under the *binding* of e. It should be noted that b_e factors out all variables that are local to q and maps the formal parameters of q to the actual parameters at the call site.

Once GMOD(p) is known for each p, the DMOD set for a statement s can be computed by the following formula.

$$\text{DMOD}(s) = \text{LMOD}(s) \cup [\bigcup_{e=(p,q)\in s} b_e(\text{GMOD}(q))] \tag{2}$$

DMOD(s) contains those variables that are modified locally in s plus any variables that are modified as a result of executing any procedure calls contained in s. Thus, if s doesn't contain any procedure calls, DMOD(s) is identical to LMOD(s). If it does contain procedure calls, each such call contributes the projection of the GMOD set of the called procedure under the binding defined by the call site.

3. The Decomposition

The SIGPLAN '84 paper decomposed the problem into two subproblems: solving for effects due to reference parameter passing and solving for effects due to global variables.

Let us define IMOD$^+$(p) to be the set of all variables that are either directly modified in p or passed as a reference parameter to another procedure and modified as a side effect of the invocation of that procedure. In other words, IMOD$^+$(p) is IMOD(p) augmented to include all variables modified at call sites in p through side effects to reference parameters. If we can compute IMOD$^+$(p) for each procedure p in the program, then we can reduce the problem of computing GMOD(p) to the solution of a system of equations analogous to equation (1).

$$\text{GMOD}(p) = \text{IMOD}^+(p) \cup [\bigcup_{e=(p,q)} b_e(\text{GMOD}(q))]. \tag{3}$$

However, since we now have already solved for the effects of reference formal parameters, the function b_e takes on a particularly simple form. If procedure p calls procedure q, b_e only needs to model modifications to variables that still exist after q returns. Clearly this means everything that is not local to q, because all of the local variables of q are deallocated on return[3]. Hence, equation (3) reduces to

$$\text{GMOD}(p) = \text{IMOD}^+(p) \cup [\bigcup_{e=(p,q)} (\text{GMOD}(q) \cap \overline{\text{LOCAL}(q)})]. \tag{4}$$

The following theorem establishes correctness of this system of equations for languages without nested procedures — in other words languages that have only local and global variables. Correctness in the general case will be established later, after a discussion of lexical scoping.

> **Theorem 1.** *For languages without procedure declaration nesting, equation system (4) is a correct formulation for* GMOD.

Proof. Clearly, we must have

$$\text{GMOD}(p) \supseteq \text{IMOD}^+(p) \cup [\bigcup_{e=(p,q)} (\text{GMOD}(q) \cap \overline{\text{LOCAL}(q)})],$$

since any variable in IMOD$^+$(p) must be in GMOD(p) and so must any global variable in GMOD(q) for q

[3] In a block structured language like Pascal, all of the variables visible in q that are not local to q or some procedure defined in q are visible from within p. However, in Fortran a global variable modified by q may not be visible in p; nevertheless, it should be included in GMOD(p).

called from within p. We need only show that the equation accounts for all variables in GMOD(p). Suppose not. Then there exists some x that can be modified as a result of invoking p that is not in the right hand side of equation (4). The variable x can be modified in one of three ways: (1) it is directly modified in p, in which case it must be in IMOD(p) by definition; (2) it is passed as a reference parameter to a call site within p and modified as a side effect of the call, in which case it must be in IMOD$^+$(p) by definition or (3) it is a global variable that is modified along some call chain originating from p. In this call chain, there must be a first procedure q to which x is passed as a global — case (2) eliminated the possibility of its being passed as a reference parameter. Since x is not a member of the set defined by the right hand side of equation (4), it must not be a member of

$$\text{GMOD}(q) \cap \overline{\text{LOCAL}(q)}$$

and since x is global to q, we must conclude that x is not a member of GMOD(q). But this is impossible because we have already established the existence of a call chain through q to a modification of x. **Q.E.D.**

Thus, we have reduced the problem to the computation of the IMOD$^+$ sets. To this end, we further decompose the problem by introducing a new set RMOD(p) that contains all formal parameters to p that are modified as a side effect of invoking p. If we can compute this set for each procedure in the program, then IMOD$^+$(p) can be computed by the following equation.

$$\text{IMOD}^+(p) = \text{IMOD}(p) \cup [\bigcup_{e=(p,q)\in E} b_e(\text{RMOD}(q))] \tag{5}$$

where the function b_e is restricted to mappings arising from actual-formal parameter bindings.

Theorem 2. *Equation system (5) is a correct formulation for* IMOD$^+$.

Proof. This is a self-evident adaptation of equation (1), since RMOD(p) is the subset of GMOD(p) that contains only formal parameters of p. The only way variables can be bound to formal parameters of a called routine is by reference. **Q.E.D.**

The problem, then, becomes one of computing RMOD(p) correctly and efficiently. The significant correctness issue is whether or not the RMOD sets can be computed without any interaction with the global variable problem. This issue is easy to resolve in languages without nested procedure declarations. Postponing the discussion of lexical scoping, we present the following correctness result.

Theorem 3. *For languages without procedure declaration nesting, there exists an algorithm for computing* RMOD *sets that makes no use of information about side effects to global variables.*

Proof. We present such an algorithm. The key observation is that every variable in RMOD(p) is a formal parameter of p. Formal parameters are local variables. In a language with no lexical scoping (one that has only local and global variables), formal parameters cannot be passed to other procedures as globals. Hence, if $x \in$ RMOD(p), then there must exist a sequence of formal parameters $(f_0, f_1, \cdots, f_n)$ such that $f_0 = x$, $f_n \in$ IMOD(q), where q is the procedure to which f_n is a parameter, and for each i, $1 \le i < n$, there is a call site at which f_{i-1} is passed by reference to f_i. This suggests an algorithm based on transitive closure.

As in the SIGPLAN '84 paper, we establish a matrix called *map* that has a row and column position for each formal parameter in the program, where each of these has a unique index assigned it. The matrix is initialized such that $map[i, j] = 1$ if and only if i is the index of a formal parameter in the program that is passed at some call site directly to the formal parameter indexed by j. It is possible to compute the reflexive transitive closure map^* of this matrix and then determine RMOD as follows: RMOD(p) contains each formal parameter x of p for which there exists a formal parameter y of some other procedure q such that $map^*[index(x), index(y)] = 1$ and $y \in$ IMOD(q).

In this algorithm, all the initial quantities can be constructed by direct examination of the program and no information about global interprocedural effects is required. The algorithm clearly computes the desired quantity, because a 1-bit in map^* represents exactly the sequence of formal parameters specified

above. **Q.E.D.**

Section 4.2 of our SIGPLAN '84 paper presented an efficient algorithm for computing the map^* and the RMOD sets, which were called $GMOD_R$ sets in that work. However, that treatment failed to observe the ordering of subproblem solutions embodied in equations (3) and (4) above and omitted the step that translates RMOD sets to $IMOD^+$ sets, as in equation (5). The error in formulation also affects the handling of lexical scoping proposed in Section 7 of the SIGPLAN '84 paper. The following section discusses this issue.

4. Lexical Scoping

The method described thus far handles the case in which there are only two kinds of storage: local and global. C and FORTRAN are both in this class. However, languages like Pascal, which permit nested declaration of procedures, present a special problem because the algorithm we have presented determines effects to global variables *after* it determines effects to formal parameters. In a language with nested procedure declarations, a local variable for one routine is global to procedures declared within the body of that routine. This can affect the computation of RMOD in two ways:

1) The computation of IMOD sets is complicated because a variable local to procedure p may be modified inside a nested procedure, where it is a global variable. These effects must be modeled in the IMOD sets if the RMOD calculation is to work correctly.

2) A formal parameter of one procedure may be passed as a global to a call site within a nested procedure. This must be reflected in the construction of map^* described above and in section 4.2 of the SIGPLAN '84 paper.

Fortunately, these two problems can be solved easily. Assume that every procedure in the program is reachable by some call chain. If this is not the case, a linear-time algorithm that eliminates unreachable procedures can be invoked. Now any procedure q nested within a procedure p is reachable by a call chain starting at p because no procedure outside of p can invoke q directly, since it is not visible outside of p. Hence, if q is reachable, it is reachable from p. This means that if p is invoked, we must assume that q may be invoked.

Given these observations and the fact that we are solving flow-insensitive problems, the first problem above is solved by treating the bodies of procedures nested in p as extensions of the body of p. This is no different than assuming that each branch at a conditional statement is possible.

We extend the $\text{IMOD}(p)$ sets to include variables that are visible within p (global or local to p) and are directly modified within the body of p or passed as globals to some procedure whose declaration is nested within p and directly modified there. If we let $\text{IMOD}_0(p)$ be the set of variables directly modified in the body of p, and $Nest(p)$ be the set of procedures declared in p, we can formulate the following definition for $\text{IMOD}(p)$.

$$\text{IMOD}(p) = \text{IMOD}_0(p) \cup \bigcup_{q \in Nest(p)} [\ \text{IMOD}(q) \cap \overline{\text{LOCAL}(q)}\ ] \tag{6}$$

The IMOD sets can then be computed in a bottom up fashion — first for the most deeply-nested procedures and then for the procedures containing those. The IMOD computation is still linear in the size of the program.

The redefinition of IMOD leads to a corresponding redefinition of IMOD^+: a variable is in $\text{IMOD}^+(p)$ for one of four reasons:

1) it is directly modified in the body of p,

2) it is passed as an actual parameter at some call site and that parameter is modified as a side effect of the procedure invocation,

3) it is passed as a global to a procedure q whose declaration is nested within p and modified directly in q, or

4) it is passed as a global to a procedure q whose declaration is nested within p then passed as an actual parameter to some call site in q and that parameter is modified as a side effect of the procedure invocation.

Assuming we can compute IMOD$^+$ correctly, then the following theorem establishes the correctness of equation (4) in the general case.

Theorem 4. *Equation system (4) is a correct formulation for* GMOD.

Proof. To extend the proof of Theorem 1, we need to do two things. First, we need a more complex argument to establish that

$$\text{GMOD}(p) \supseteq \text{IMOD}^+(p) \cup [\bigcup_{e=(p,q)} (\text{GMOD}(q) \cap \overline{\text{LOCAL}(q)})\ ].$$

The only issue is whether variables in GMOD(q) that are not local to q are visible in p. By the rules of visibility in procedure declaration nesting, q may be in one of two categories: (1) declared within p, (2) declared in some procedure r that either contains the declaration of p or contains a nest of procedure declarations in which p is declared. In either case, all variables global to q are visible within p.[4] Note that it may also be the case that some variables local to q may be visible within p, as when the declaration of p is nested in q. However, since all variables local to q are deallocated on return to p, modifications to these variables in q cannot affect the GMOD set for p.

Second we must show that every variable in GMOD(p) is represented on the right hand side of equation (4). Let x be a variable in GMOD(p) that is not in the set specified by the right hand side of equation (4). Now x can be modified in one of five ways: (1) directly in the body of p, (2) as a side effect to a reference parameter at a call site in p, (3) directly in the body of some procedure declared within p, (4) as a side effect to a reference parameter at a call site in the body of a procedure declared in p, or (5) as a side effect of a procedure to which it is passed as a global. It should be easy to see that the new definitions of IMOD and IMOD$^+$ insure that cases (1-4) are taken care of, so these must be assumed impossible. The remainder of the proof follows the proof of Theorem 1. **Q.E.D.**

The problem of formal parameters passed as global variables is easily handled by a careful initialization of basic data structures in the *map** calculation.

Theorem 5. *There exists an algorithm for computing* RMOD *sets that makes no use of information about side effects to global variables.*

Proof. The proof is basically the same as the one for Theorem 3, except we must show that the case of formal parameters passed as globals can be handled. Note that a formal parameter can only be passed as a global to a more deeply nested procedure, so the linear time scan of procedures to develop IMOD can also notice when a call site passes a formal parameter of an enclosing procedure as an actual parameter. Since the *map* matrix has positions for every formal parameter in the program, it is easy to set the appropriate bits in *map* to reflect this information. In other words, whenever a call site is encountered that passes a formal parameter with index i to a formal parameter with index j, set *map*$[i, j] = 1$, regardless of which procedure in the upwards nesting structure has i as its formal parameter. The remainder of the proof follows the proof

[4] This is not strictly true in a language like FORTRAN, where global variables (COMMON blocks) are visible only if named. However, the correct observation is that they *can* be visible.

of Theorem 3. **Q.E.D.**

Thus, the problems caused by lexical scoping are surmountable and the same time bounds pertain.

5. Summary

We have shown how to correct a significant error in one of our previously-published conference papers. The corrections preserve the time bounds stated in the original paper, because the time bounds for the subproblems still hold, and the subproblems are solved in sequence.

6. Acknowledgements

Barbara Ryder and Frances Allen both corresponded with us about the problem with our original formulation. Barbara also suggested several corrections, one of which is essentially the same as the one we had developed independently and which we reported here. Our colleagues at Rice, particularly Linda Torczon and David Callahan, provided stimulating discussions on these issues.

References

[Alle 74] F.E. Allen, ''Interprocedural data flow analysis'', *Proceedings of the 1974 IFIPS Congress*, 1974.

[Alle 87] F.E. Allen, private communication, September 1987.

[Bann 79] J.P. Banning, ''An efficient way to find the side effects of procedure calls and the aliases of variables'', *Proceedings of the Sixth POPL*, Jan., 1979.

[Bart 78] J.M. Barth, ''A practical interprocedural data flow analysis algorithm'', *CACM* 21(9), Sept., 1978.

[Burk 84] M. Burke, ''An interval analysis approach toward interprocedural data flow'', Report RC 10640, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., July, 1984.

[Carr 87] M.D. Carroll, ''Dataflow update via attribute and dominator update,'' Ph.D. Dissertation, Rutgers University, expected 1987.

[CaRy 86] M.D. Carroll and B.G. Ryder, ''An incremental algorithm for software analysis'', *Proceedings of the* SIGSOFT/SIGPLAN *Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices* 22(1), Jan., 1987.

[CoKe 84] K.D. Cooper and K. Kennedy, ''Efficient computation of flow insensitive interprocedural summary information'', *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction, SIGPLAN Notices* 19(6), June, 1984.

[CoKe 87] K.D. Cooper and K. Kennedy, ''Complexity of interprocedural side-effect analysis'', TR87-61, Department of Computer Science, Rice University, October 1987.

[Myer 80] E. Myers, ''A precise and efficient algorithm for determining existential summary data flow information'', Technical Report CU-CS-175-80, Department of Computer Science, University of Colorado, March, 1980.

[Ryde 87] B. Ryder, private communication, July 31, 1987.

[Spil 71] T.C. Spillman, ''Exposing side-effects in a PL/I optimizing compiler'', *Proceedings of the IFIPS Congress*, 1971.

This paper also appears as Computer Science Technical Report 87-60, Rice University, October, 1987.