CHAPTER 10

Interprocedural Analysis and Optimization

10.1 Introduction

As machines and languages have become more complicated, compiler technology has necessarily become more sophisticated. With the advent of vector and parallel computers, single-procedure analysis will no longer be enough to produce high-quality parallelized code. In this chapter we introduce some of the problems that can be solved with the help of an interprocedural analysis and optimization system. We also present methods for the solution of these problems, summarizing research on interprocedural compilation over the past 15 years.

We begin with a definition of the terms interprocedural analysis and interprocedural optimization. *Interprocedural analysis* refers to gathering information about the entire program instead of a single procedure. This is usually comparable to the sort of information that a single procedure data-flow analysis system would collect. Examples of interprocedural analysis problems are determining the variables that are modified as side effect of a procedure call and finding whether a given pair of variable might be aliased to one another on entry to a given procedure.

Interprocedural optimization refers to program transformations that involve more than one procedure in a program. The most familiar example of an interprocedural optimization is *inlining*, by which the body of a procedure is substituted at the point of invocation. Although interprocedural optimizations typically modify more than one subroutine, it is reasonable to view any optimization that is based on interprocedural knowledge—gathered by an interprocedural analysis phase—as an interprocedural optimization. However, for the purposes of this chapter we will adhere to the narrower view.

10.2 Interprocedural Analysis

To illustrate the need for interprocedural analysis methods we will introduce several important problems through a series of examples.

10.2.1 Interprocedural Problems

10.2.1.1 Modification and Reference Side Effects

We begin with a simple vectorization problem. Suppose we have the following code fragment:

```
     COMMON X, Y
     ...
     DO 100 I = 1, N
S0      CALL S
50      X(I) = X(I) + Y(I)
100  CONTINUE
```

Without some sort of interprocedural optimization, it is impossible to vectorize the call, but might it be possible to vectorize the assignment at statement 50? Since both X and Y are in COMMON, we must be concerned with side effects to these variables due to the call at S_0. A statement can be vectorized if it is legal to distribute the loop around the two statements S_0 and 50. In turn, this is possible if there is no cycle of dependences involving both statements.We can be sure that there is no such dependence cycle if the call meets the following criteria:

1. it neither uses nor modifies X and

2. it does not modify Y.

The first condition ensures that there can be no dependence cycle involving both the call and statement 50 due to variable X. The second condition rules out a dependence cycle due to Y.

To address this problem, we introduce the interprocedural MOD and REF problems.

> **Definition 10.1.** At a given call site *s* the *modification side effect set* MOD(*s*) is the set of all variables that may be modified as a side effect of the call at *s*. The *reference side effect set* REF(*s*) is the set of all variables that may be referenced as a side effect of the call at *s*.

With these definitions, we can now formally restate the condition under which the assignment statement above can be vectorized, namely:

$$X \notin \mathrm{REF}(S0) \text{ and } X \notin \mathrm{MOD}(S0) \text{ and } Y \notin \mathrm{MOD}(S0).$$

10.2.1.2 Alias Analysis

Suppose we have the following subroutine:

```
      SUBROUTINE S(A,X,N)
         REAL A(*), X, Y
         COMMON Y
         DO 100 I = 1, N
S0          X = X + Y*A(I)
100      CONTINUE
      END
```

In compiling this loop for any machine, it would be efficient to keep both variables X and Y in registers throughout the duration of the loop, without storing X until the loop is completed. This looks straightforward, but what if subroutine S is invoked by a call like

```
CALL S(A,Y,N)?
```

In this case Y is *aliased* to X, so by not storing X on every iteration, we may be overlooking required updates to the variable Y within the loop[1].

To avoid problems like the compiler must determine when two variables might be aliased to one another on entry to a given subroutine.

Definition 10.2. For a procedure p and a formal parameter x passed to p, the *alias set* ALIAS(s,x), is the set of variables that may refer to the same location as x on entry to s.

In the example above, `X` and `Y` may be kept in registers without storing to memory if

$$Y \notin \text{ALIAS}(S,X).$$

10.2.1.3 Call Graph Construction

The call graph of a program is a graph that models the calling relationships between procedures in a program.

Definition 10.3. The *call graph* of a program is a graph $G = (N,E)$ where the vertices in N represent procedures in the program and the edges in E represent possible calls.

It is common for each distinct call site in a program to be represented by a distinct edge, in which case the call graph is actually a multigraph. We will adopt this convention in the remainder of the chapter.

The accuracy of the call graph directly affects the precision of the data-flow information produced. However, construction of a precise call graph is in itself an interprocedural analysis problem. In a language in which each call must be to a named constant procedure, the call graph is easy to construct—you need only examine the body of each procedure p, entering for each call site s in p an edge (p,q) to the procedure q called at s.

However, a precise call graph is more difficult to construct in a language that permits procedure variables. Even in Fortran, where there are no assignable procedure variables, problems arise due to *procedure parameters*—formal parameters which may be bound to procedure names at the point of call. Consider the following example:

```
     SUBROUTINE S(X,P)
S0      CALL P(X)
        RETURN
     END
```

1. Knowledgable readers will comment that the Fortran standard defines this usage to be illegal, saying that if two variables are aliases on entry to a loop then the program is not standard-conforming if the subroutine stores into either. Although this is an easy escape from the specific problem presented, it only works for Fortran, as C has no such prohibition.

The question is: What procedure or procedures may be called at statement `S0`? In other words, what values can `P` have at this call site? We could attempt to answer this question by examining the actual parameters at all the call sites for subroutine `S`, but any actual parameter could itself be a procedure parameter. Thus we must propagate procedure constants through the call graph before we can finish building it.

> **Definition 10.4.** For a given procedure *p* and call site *s* within *p,* the *call set* CALL(*s*) is the set of all procedures that may be invoked at *s.*

Although it is stated as a property of call sites, CALL(*s*) is not a side-effect problem. It is really asking what procedures can be passed to formal parameters on entry to the procedure containing *s.* Because it depends on the context in which the procedure containing *s* is invoked, it resembles alias analysis more closely than any side effect problem.

10.2.1.4 Live and Use Analysis

An important data-flow problem that has been studied extensively in the literature is the analysis of *live variables.* A variable *x* is said to be *live* at a given point *s* in a program if there is a control-flow path from *s* to a use of *x* that contains no definition of *x* prior to the use.

One important application of live analysis is in determining whether a private variable in a parallel loop needs to be assigned to a global variable at the end of the loop's execution. Consider the following code fragment:

```
DO I = 1, N
   T = X(I)*C
   A(I) = T + B(I)
   C(I) = T + D(I)
ENDDO
```

This loop can be parallelized by making `T` a local variable in the loop. However, if T is used later in the program before being redefined, the parallelized program must assign the last value of the local version of `T` to the global version of `T`. In other words, the code must be transformed as follows:

```
PARALLEL DO I = 1, N
   LOCAL t
   t = X(I)*C
   A(I) = t + B(I)
   C(I) = t + D(I)
   IF (I.EQ.N) T = t
ENDDO
```

Here we are using the typographic convention that variables introduced in small letters are compiler-generated.

This code could be simplified if we could determine that variable `T` is not live on exit from the original loop. In this case, the conditional at the end of the loop could be eliminated to produce:

```
PARALLEL DO I = 1, N
   LOCAL t
   t = X(I)*C
   A(I) = t + B(I)
   C(I) = t + D(I)
ENDDO
```

Although live analysis can itself be viewed as an interprocedural problem, it is conveniently dealt with in terms of another interprocedural problem. *Use analysis* is the problem of determining whether there is an *upwards-exposed use* of a variable on some path through a procedure invoked at a particular call site *s*. An upwards exposed use is one that is not preceded by a definition of the variable on some path to the use from the point of invocation.

> **Definition 10.5.** For a given call site *s*, which invokes procedure *p*, the *use side effect set* USE(*s*) is the set of variables that have an upwards-exposed use in *p*.

Given this definition, we can give a more formal specification of when a variable is live. A variable *x* is *live* at a call site *s* if $x \in$ USE(*s*) or if there is a path through the procedure called at *s* that does not assign a new value to *x* and *x* is live at some control-flow successor of *s*.

10.2.1.5 Kill Analysis

Problems like REF, MOD and USE ask questions about what might happen on some path through a called subprocedure. It is often useful to ask about what *must* happen on every path through a procedure. The following example illustrates this:

```
L    DO I = 1, N
S0      CALL INIT(T,I)
        T = T + B(I)
        A(I) = A(I) + T
     ENDDO
```

There are two problems that might keep this loop from being correctly parallelized. First, not knowing what the subroutine `INIT` does, we must assume that it assigns variables in a way that creates a cycle of dependences. One way to do this would be to use then assign a variable that is global to the program. For example, if `INIT` were defined as in the following code, parallelization would be precluded:

```
SUBROUTINE INIT(T,I)
   REAL T
   INTEGER I
   COMMON X(100)
   T = X(I)
   X(I+1) = T + X(1)
END
```

Here `INIT` creates a cycle of dependences involving the COMMON array `X`, with respect to the loop in the calling program.

However, even if we can prove that the loop does not modify any global variables or even any static local (i.e., "SAVE") variables, the call presents a more subtle problem. If this loop is to be parallelized, it must be possible to recognize that variable `T` can be made private to each iteration. This is possible, for example, if subroutine `INIT` is simply there for the purpose of initializing `T`, as in:

```
SUBROUTINE INIT(T,I)
   REAL T
   INTEGER I
   COMMON X(100)
   T = X(I)
END
```

Here `T` is initialized before being used on every iteration of the loop. Thus it is not involved in any kind of carried dependence within the loop and may be made private.

How can we discover this fact? Certainly, MOD can tell us that no global variables are modified within the subroutine and we assume that a similar analysis could be used to preclude the possibility that any static local variable in `INIT` is used before being modified. Therefore, the key to determining that the loop can be parallelized is to establish that variable `T` is assigned before being used *on every path* through `INIT`.

The problem of discovering whether a variable is assigned on every path through a called procedure is known as KILL, because an assignment is said to "kill" a previous value of the variable.

Definition 10.6. For a given call site s, the *kill side effect set* KILL(p) is the set of variables that are assigned on every path through the procedure p invoked at s and procedures invoked from within p.

Assuming that there are no global variables in MOD(p) and that p does not use any static local variables before they are assigned, then the loop `L` can be parallelized if variable `T` is killed on every path through the procedure called at `S0` and there is no path into that procedure on which a use appears before any kill. This can be expressed formally as

$$\texttt{T} \in (\text{KILL}(\texttt{S0}) \cap \neg\text{USE}(\texttt{S0})).$$

Assuming that call site s is in a block by itself, it is possible to define LIVE(s) within the procedure containing s as follows:

$$\text{LIVE}(s) = \text{USE}(s) \cup \left(\neg\text{KILL}(s) \cap \bigcup_{b \in succ(s)} \text{LIVE}(b)\right). \qquad \textbf{(EQ 10.1)}$$

This generalizes the live computation to the interprocedural case if we know, by some additional analysis, the set of variables that are live on exit from the procedure.

10.2.1.6 Constant Propagation

Constant propagation, one of the most important problems in single-procedure data-flow analysis, is also an important interprocedural problem. Consider the following program, which is abstracted from code in LINPACK:

```
     SUBROUTINE S(A,B,N,IS,I1)
        REAL A(*), B(*)
L       DO I = 0, N-1
S0         A(IS*I + I1) = A(IS*I + I1) + B(I+1)
        ENDDO
     END
```

If we wish to vectorize loop `L` in this subroutine, a problem arises because the variable `IS` might take on the value 0. If that is so, there is an output dependence. In that case statement `S0` is actually a reduction and cannot be vectorized using the usual techniques. Although we could test for this situation at run time, we can avoid it entirely if we can determine that, on every invocation of subroutine `S` in the program containing it, the value of `IS` is always `1` (the most common case).[1]

Definition 10.7. Given a program and a procedure p within that program, the *set of interprocedural constants* CONST(p) contains the variables that have known constant values on every invocation of p. For a variable $x \in$ CONST(p), *valin*(x,p) is a function returning the value of x on entry to p.

Although even the single-procedure constant propagation problem is intractable, approximate solutions in single procedures have been shown to be effective [72]. Similarly, approximate solutions of the interprocedural constant propagation problem have been shown to determine many facts that are useful for optimization and parallelization [36,42].

10.2.2 Interprocedural Problem Classification

We will now explore various classifications of interprocedural data-flow problems. These classifications are useful, because problems in the same class can generally be solved by the same algorithmic approach.

10.2.2.1 May and Must Problems

We have already seen that there is a distinction between problems that ask whether some event "may" happen or "must" happen. MOD, REF and USE are *may problems* because they compute sets of variables that may be modified, may be referenced or may be used before being defined, respectively. On the other hand, KILL is a *must problem* because it computes a set of variables that must be killed.

Although this distinction has been extensively discussed in the literature, it is not a deep one because the converse of every may problem is a must problem and vice versa. For example, ¬MOD(s) is the set of all variables that are not in MOD(s). Therefore, the ¬MOD problem seeks to find those variables that must not be modified as a side effect of a given call site. Thus ¬MOD is a must problem. Similarly, ¬REF is also a must problem. On the other hand, ¬KILL seeks to find for each call site s in the program the set of variables that may not be modified on some path through the procedure being invoked. Thus the converse of KILL, a must problem, is a may problem. Since the solution to any set problem may be converted to the solution of its converse by subtraction from the universal set (usually bit vector complementation), which takes time linear in the number of answer sets, there is no difference in complexity of a must problem and its corresponding may problem.

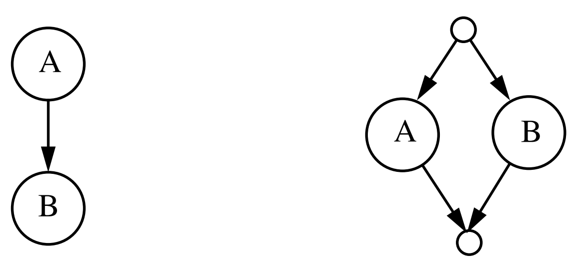
1. Barring that, it would be vectorizable if it can be established that `IS≠0` every time the loop is entered. Analysis of predicates like this can also be handled by a variation of constant propagation.

10.2.2.2 Flow-Sensitive and Flow-Insensitive Problems

Banning introduced a seemingly related notion of flow-sensitive and flow-insensitive problems. Intuitively, a *flow-sensitive* problem is one whose solution requires tracing individual control-flow paths through the body of the subroutine being called. A *flow-insensitive* problem, on the other hand can be solved by examination of the body of called subroutines without regard to control flow. Thus, MOD and REF are flow-insensitive problems because, assuming all code in a subroutine is reachable, any modification of a variable x in the body of a subroutine p means that $x \in \text{MOD}(s)$ for any call site s that invokes p. On the other hand, KILL is a flow-sensitive problem because, for a given variable x, its solution requires checking every path through the procedure body to ensure that it contains a definition of x.

Banning [5] presents a more formal definition of flow-sensitive and flow-insensitive problems based on how solutions on subregions of a call graph would be composed into solutions for a larger region. Consider the two call graph regions depicted in Figure 10.1.

FIGURE 10.1 Call graph subregions.



Region R_1 Region R_2

Suppose we have MOD for each subregion A and B of regions R_1 and R_2. Then we can compose these into solutions for the whole regions as follows:

$$\text{MOD}(R_1) = \text{MOD}(A) \cup \text{MOD}(B)$$

$$\text{MOD}(R_2) = \text{MOD}(A) \cup \text{MOD}(B)$$

This is because in either decomposition, a variable may be modified if it may be modified in either subregion. There is a similar pair of equations for REF:

$$\text{REF}(R_1) = \text{REF}(A) \cup \text{REF}(B)$$

$$\text{REF}(R_2) = \text{REF}(A) \cup \text{REF}(B)$$

Now let us carry out the same exercise for KILL. In the case of region R_1, a variable is killed if it is killed in the first region or if it is killed in the second region:

$$\text{KILL}(R_1) = \text{KILL}(A) \cup \text{KILL}(B)$$

In the case of region R_2, a variable must be killed in both regions, to be in $\text{KILL}(R_2)$:

$$\text{KILL}(R_2) = \text{KILL}(A) \cap \text{KILL}(B)$$

The equations for USE are equally easy to develop:

$$\text{USE}(R_1) = \text{USE}(A) \cup (\neg\text{KILL}(B) \cap \text{USE}(B))$$

$$\text{USE}(R_2) = \text{USE}(A) \cup \text{USE}(B)$$

Only this second equation is the same as the corresponding equation for MOD and REF.

In examining these equations, we notice that the equations for MOD and REF use only set union as a connector, while the equations for KILL and USE are more complicated, using other connectors and often other local sets. This leads us to the following definition.

> **Definition 10.8.** An interprocedural data-flow problem is *flow-insensitive* if and only if, on the parameter-free version of the problem, the value of the solution on both sequentially and alternatively composed regions (R_1 and R_2 of Figure 10.1) is determined by taking the union of the solutions for the subregions.

The literature on optimization sometimes refers to *flow-insensitive analysis* of a flow-sensitive problem. Our interpretation of this term is that it refers to approximating a flow-sensitive problem with a flow-insensitive one. As an example, suppose you wished to approximate a solution to the USE problem with the solution to one or more flow-insensitive problems. Note that most optimizations will be performed when it is known that a variable is not in USE(p) for a particular procedure p called from within the procedure being optimized. Therefore, we wish any approximation APUSE to be conservative in the sense that it contain all of USE:

$$\text{USE}(p) \subseteq \text{APUSE}(p) \text{ or } \neg\text{APUSE}(p) \subseteq \neg\text{USE}(p)$$

If this is the case we will never depend on an untrue fact. One possible approximation is given by:

$$\text{APUSE}(p) = \text{REF}(p)$$

which is clearly a superset of USE(p) and can be computed by solving a flow-insensitive problem. This might be useful because, as we shall see, flow-insensitive problems are easier to solve than flow-sensitive ones. However, we do not recommend the use of approximations like this one because more precise approximations can be computed in reasonable time using flow-sensitive analysis.

10.2.2.3 Side Effect Problems versus Propagation Problems

A final classification of interprocedural problems is by direction of data flow. Some interprocedural problems ask what may or must happen as a side effect of a procedure call that is about to be made. This class, which we refer to as *side-effect problems*, includes MOD, REF, KILL, and USE. Problems of this sort are analogous to the backward data-flow problems of single-procedure analysis.

A second class of problems asks what conditions may or must hold upon entry to the current procedure (which presumably we are interested in optimizing). We call these *propagation problems*. They include ALIAS, CALLS and CONST. The method we developed for classifying side-effect problems as flow-sensitive or flow-insensitive will not work for propagation problems, because these problems look back up the call chain. However, it is generally agreed that the ALIAS problem for Fortran is flow-insensitive since aliases can only be introduced through the use of reference formal parameters. Thus, flow through a subroutine to a call site is unimportant because aliases can only be introduced at procedure invocation.

On the other hand, the constant propagation problem for Fortran is flow-sensitive because, for a constant to propagate to a called procedure, the variable must receive the same constant value on every path through every procedure in the call chain. By the same argument, alias analysis for languages like C with pointer assignment is also flow-sensitive.

10.2.2.4 A Problem Classification Table

The breakdown of problems in the two dimensions is summarized in Table 8 below.

	propagation	side effect
flow-insensitive	ALIAS, CALLS	MOD, REF
flow-sensitive	CONST	KILL, USE

TABLE 8 Interprocedural problem classification

The classification into flow-sensitive and flow-insensitive is an important one because Banning established, by providing an algorithm, that flow-insensitive problems can be solved in time that is polynomial in the size of the call graph [5], while Myers showed that flow-sensitive problems are intractable in the presence of aliasing and arbitrary nesting [56].

10.2.3 Flow-Insensitive Side-Effect Analysis

We now turn to the solution of the flow-insensitive side-effect analysis problems. Throughout this section, we use the modification side-effect analysis (MOD) problem as our example. The reference side-effect analysis (REF) problem can be solved using exactly the same approach.

10.2.3.1 Assumptions

We begin by establishing a set of assumptions that are representative characteristics of Fortran and, in some cases, C. First, we assume that there is no procedure nesting—that is, variables are subdivided into local and global sets. Local variables are known only within their home procedure and global variables are known in every procedure. Although this might seem restrictive, the methods we will introduce here have easy extensions to the case of general nesting.

A second assumption, already hinted at, is that all parameters are passed by reference and that there are no pointer variables. The purpose of this restriction is to simplify the aliasing patterns that the algorithms must deal with. Under this restriction, aliases can only be introduced at call sites. Although both Fortran 90 and C have pointer variables, this assumption is valid for Fortran 77, which is the input language for most automatic parallelization systems.

Even though Fortran 77 does not support recursion, the algorithms presented here work correctly in the presence of recursive procedure calls. We do however make an assumption about the size of parameter lists to procedures, namely that the maximum number of formal parameters to a procedure does not grow with the size of the program. In other words, programmers do not typically deal with the increasing complexity of a large program by increasing the complexity of the procedure interfaces. Thus we will assume that there exists a constant μ, such that the number of formal parameters in any procedure p is less than or equal to μ.

10.2.3.2 MOD Problem Formulation

The goal is to compute, for each call site s in the program, the set MOD(s) that contains every variable that may be modified as a side effect of the procedure call at s. Our first step is to note that we can simplify the problem by disregarding potential aliases. Specifically, we will compute the set DMOD(s), the *direct modification side-effect set*, which contains all variables visible at s that are directly modified as a side effect of the call. DMOD(s) may be smaller than MOD(s) because it does not take into account the fact that a variable x that is directly modified as a side effect may have several possible aliases at the point of call. Each of these aliases must be in the final MOD(s) if the solution is to be precise. However, once DMOD(s) has been computed it can be updated to MOD(s) with the help of the ALIAS sets described in Section 10.2.1.2. Recall that for a given procedure p, ALIAS(p,x) contains the set of all variables that may be aliased to x on entry to p. Given these sets, we can update DMOD(s) to MOD(s) according to the following formula:

$$\mathrm{MOD}(s) = \mathrm{DMOD}(s) \cup \bigcup_{x \in \mathrm{DMOD}(s)} \mathrm{ALIAS}(p, x) \qquad \textbf{(EQ 10.2)}$$

where p is the procedure containing call site s. Construction of ALIAS(p,x) will be discussed in Section 10.2.4.

We will compute DMOD(s) for each call site by first computing, for each procedure p in the program, the set GMOD(p), called the *general modification side effect*. GMOD(p) contains the set of global variables and formal parameters of p that are modified, either directly or indirectly as a result of the invocation of p. Once these sets are available, we can compute DMOD(s) by the following formula:

$$\mathrm{DMOD}(s) = \{v \mid s \text{ invokes } p,\ z \xrightarrow{s} w \text{ and } w \in \mathrm{GMOD}(p)\},$$
$$\text{where} \qquad \textbf{(EQ 10.3)}$$
$$z \xrightarrow{s} w$$

means that at call site s actual parameter z is bound to formal parameter w of the called procedure. Equation 10.3 reduces the problem to computing, for every procedure p in the program, the set GMOD(p) of variables that may be changed as a side effect of invoking p. Generally, GMOD(p) will contain two types of variables:

1. those that are explicitly modified in the body of procedure p and
2. those that are modified as a side effect of some procedure invoked in the body of p.

Let IMOD(p), the *immediate modification side-effect set*, denote the set of variables explicitly modified in p. Then the following formula holds:

$$\mathrm{GMOD}(p) = \mathrm{IMOD}(p) \cup \bigcup_{s=(p,q)} \{z \mid z \xrightarrow{s} w \text{ and } w \in \mathrm{GMOD}(q)\} \quad \textbf{(EQ 10.4)}$$

Note that the union is over all call sites s within p.

10.2.3.3 Problem Decomposition

The system of data-flow equations in Equation 10.4 can be solved by the iterative method of data-flow analysis. However, the solution may take a long time because the system does not satisfy the conditions under which the iterative method can be guaranteed to converge rapidly [57]. Furthermore it is not formulated in a way that permits the use of a fast elimination method [35]. Fast solution methods rely on being able to limit the number of times that the analysis must traverse a loop in the problem graph. For flow-insensitive interprocedural analysis, the problem graph is the call graph, so convergence problems are limited to recursive regions.

A closer examination of the problem with recursion reveals that it is related to reference formal parameters. Consider the following example:

```
      SUBROUTINE P(F0,F1,F2,...,Fn)
         INTEGER X,F0,F1,F2,...,Fn
         ...
S0       F0 = some expression
         ...
S1       CALL P(F1,F2,...,Fn,X)
         ...
      END
```

In this example it is possible to see why, in the general case, an analysis routine must iterate around the recursive cycle an unbounded number of times. The question being asked is: How many of the parameters of subroutine P may be modified as a side effect of invoking P? Clearly, F_0 can be modified at statement S0, but we have to examine the recursive call at S_1 to discover that F_1 is passed to F_0, so it too can be modified. One more time around the recursive loop reveals that F_2 can also be modified. This procedure continues until it is discovered that F_n—the last parameter—may also be modified. If *n* is unbounded, the number of iterations over the recursive cycle is unbounded as well.

These observations make it clear why we wished to assume an upper bound on the number of parameters to any subroutine—it permits us to establish a constant upper bound on the number of iterations required for the process to converge. However, we can achieve a better time bound by decomposing the problem further, treating side effects to reference parameters separately from side effects to global variables. This is achieved by introducing an extended version of the immediate modification side effect set IMOD(p), called IMOD$^+$(p), which will contain all of IMOD(p) plus all those variables that may be modified as a result of side effects to reference formal parameters of procedures invoked from within p. In other words, a variable x is in IMOD$^+$(p) if

1. $x \in \mathrm{IMOD}(p)$ or
2. $x \xrightarrow{s} z$ and $z \in \mathrm{GMOD}(q)$, where $s=(p,q)$.

If we can compute IMOD$^+$(p) for every procedure p in the program, we can solve for GMOD(p) using the following simple system of equations:

$$\mathrm{GMOD}(p) = \mathrm{IMOD}^+(p) \cup \bigcup_{s=(p,q)} \mathrm{GMOD}(q) \cap \neg\mathrm{LOCAL} \qquad \textbf{(EQ 10.5)}$$

where LOCAL is the set of all local variables in the program, so its complement is the set of all global variables. Since all side effects to reference formal parameters of p are reflected in IMOD$^+$(p), we need only be sure that all side effects to global variables are added by the union of the GMOD sets for successors. If a global variable is modified as a side effect of invoking p it must be modified directly in the text of the subroutine, in which case it is in $\mathrm{IMOD}(p) \subseteq \mathrm{IMOD}^+(p)$, or it is passed as an actual parameter to another subroutine where it is modified, in which case it is in IMOD$^+$(p) by definition, or it is modified as a global by some subroutine called directly or indirectly from p, in which case it must be in GMOD(q) for some successor q of p, which establishes Equation 10.5.

We have thus decomposed the problem into two parts:

1. the computation of IMOD$^+$(p) for every procedure p in the program and
2. the propagation of global modification according to Equation 10.5.

We will cover these calculations in the next two subsections.

10.2.3.4 Solving for IMOD$^+$

We construct a special data structure, called the *binding graph*, as follows:

1. construct a vertex for each formal parameter f of every procedure in the program
2. construct an edge between a pair (f_1,f_2) of formal parameters if f_1 is a formal parameter of procedure p, f_2 is a formal parameter of procedure q and f_1 is bound to f_2 at a call site s=(p,q) in p.

A preliminary question we might ask is: How large is the binding graph? If N is the number of vertices in the call graph and E is the number of edges, then the number of vertices in the binding graph can be no more than μN, where μ is the upper bound on the number of parameters to any procedure in the program. This is true because there can be no more vertices than the number of formal parameters in the entire program, which is clearly bounded by μN. Similarly, each call graph edge can give rise to no more than μ edges, since no more than one formal reference parameter can appear in a single actual parameter position. Thus the total number of edges in the binding graph is no larger than μE and the binding graph is no more than a constant factor greater in size than the call graph; that is, its size is O(N+E).

Let RMOD(p) be the set of formal parameters in procedure p that may be modified in p, either directly or by assignment to a reference formal parameter of q as a side effect of a call to q from within p. Clearly, the following formula, which is a direct analog of Equation 10.4, holds:

$$\mathrm{IMOD}^+(p) = \mathrm{IMOD}(p) \cup \bigcup_{s=(p,q)} \{z \mid z \xrightarrow{s} w \text{ and } w \in \mathrm{RMOD}(q)\} \quad \textbf{(EQ 10.6)}$$

Thus we need only construct RMOD(p) for every procedure in the program, and then apply Equation 10.6, to construct IMOD+(p).

To construct RMOD(p), we will use a simple marking algorithm on the binding graph, in which each vertex is annotated with a logical mark implementable with a single bit. Initially, all the marks are set false, then the marks for formals of any procedure p that are in IMOD(p) are set true. True bits are propagated around the graph using the rule that any formal f_1 that is bound to formal f_2 with a true mark must also be marked true. When no more propagation is possible, RMOD(p) is the set of formals of p that are marked true. The algorithm is given in Figure 10.2.

FIGURE 10.2 Algorithm for constructing RMOD sets.

```
procedure computeRmod(P, N_B, E_B, IMOD, proc, RMOD)
      /* P is the collection of procedures in the program */
      /* N_B is the collection of formal parameters in the binding graph */
      /* E_B is the collection of edges in the binding graph */
      /* mark[f] maps formal parameters to their mark values */
      /* proc[f] maps a parameter to its procedure */
      /* IMOD[p] maps a procedure to its immediate mod side effect set */
      /* RMOD[p] is the collection of output sets */
      /* worklist is a working set of formal parameters */

L1    for each f∈N_B do mark[f] := false od;
      worklist := ∅;

L2    for each f∈N_B such that f∈IMOD[proc[f]] do
S1        mark[f] := true;
          worklist := worklist ∪ {f}
      od;

L3    while worklist ≠ ∅ do
          f := an arbitrary element in worklist;
          worklist := worklist – {f};
L4        for each v such that (v,f) ∈ E_B do
S2            if mark[v] = false then
S3                mark[v] := true;
                  worklist := worklist ∪ {v}
              fi
          od
      od;

L5    for each p ∈ P do RMOD[p] := ∅ od;
L6    for each f∈N_B do
          if mark[f] then RMOD[proc[f]] := RMOD[proc[f]] ∪ {f} fi
      od
end computeRmod
```

> **Theorem 10.1.** Algorithm computeRmod in Figure 10.2 correctly computes the RMOD sets—that is, on exit $f \in$ RMOD[*proc*[*f*]] if and only if *f* may be modified as a side effect of invoking the procedure $p = proc[f]$.

Proof. If. Assume that *f* may be modified. Then either it is modified in the procedure to which it is a parameter, in which case it is marked true in statement S_1, or there is a path in the binding graph to a formal parameter f_0 which is in IMOD[*proc*[f_0]]. Thus f_0 is marked true and added to the worklist in loop L_1. Let $f = f_n, f_{n-1}, ..., f_1, f_0$ be the sequence of parameters in the binding graph that constitute the path from *f* to f_0. Suppose $f = f_n$ is never marked true. Then there must be a minimum *k* such that f_k is never marked true by the algorithm but f_{k-1} is marked true. But since f_{k-1} is put on the worklist when it is marked true and it is taken off the worklist eventually and every incoming edge to f_{k-1} is examined at that time, f_k must be marked true, a contradiction. Thus every parameter that may be modified as a side effect of invoking *proc*[*f*] is put into RMOD[*proc*[*f*]].

Only if. Suppose *f* is a parameter that is marked true but cannot be modified. The algorithm only marks formal parameters true if they are modified within the body of their procedure or if there is a path in the binding graph to another parameter which is modified in its procedure. Since the binding graph has an edge only if the corresponding procedure call is possible, it must be the case that the original parameter *f* may be modified. □

> **Theorem 10.2.** Algorithm *computeRmod* operates in worst case $O(N+E)$ steps, where *N* and *E* are the number of vertices and edges in the call graph.

Proof. Let N_B and E_B be the number of vertices and edges, respectively in the binding graph. From the discussion in Section 10.2.3.4 on page 321 we know that

$$N_B \le \mu N \text{ and } E_B \le \mu E$$

Thus, if we can show that the algorithm runs in time proportional to the size of the binding graph, we will have established the result. Clearly loop L_1 requires N_B steps. Assuming that IMOD is implemented so that membership testing can be done in constant time (i.e. via a bit vector) loop L_2 also runs in N_B steps. Loop L_5 takes time proportional to the number of procedures in the program times the time to initialize the RMOD sets. If these sets are implemented as bit vectors of length μ then this takes $O(N_B)$ time. Loop L_6 takes $O(N_B)$ assuming that adding to RMOD takes constant time, as it would with a bit vector implementation.

All that remains is to analyze the running time of loops L_3 and L_4. If we assume that the worklist is implemented as a linked list so that taking an arbitrary element from the front takes constant time, the body of the loop will be executed no more than N_B times, since a vertex is put onto the worklist no more than once. Let us assume that, as in topological sort algorithms, the edges are arranged as predecessor lists, so that all predecessors of a particular vertex can be found in time proportional to the number of predecessors. Then the body of the loop is executed once for each predecessor of each vertex in the program, for a total of E_B times.

Thus, we have established that the algorithm takes $O(N_B+E_B)=O(N+E)$ steps in the worst case. □

To expand RMOD to IMOD^+ requires that we visit each call site in the program and each parameter at that call site to determine if it is bound to a variable in RMOD(p) for the called procedure p. If the RMOD sets are represented as bit vectors this determination takes constant time, so the conversion to IMOD^+ takes $O(N+E)$ time. This assumes that IMOD^+ can be initialized in $O(N+E)$ time. If $\text{IMOD}^+(p)$ is represented as a bit vector, it must have length proportional to the number V of global variables in the program plus the number of formal parameters to p. Since the number of such parameters is $\leq\mu$, the time to initialize the bit vector set for each procedure is $O(V+\mu) = O(V)$. Since this is done once for each procedure the initialization time is $O(NV)$. Thus the computation of IMOD^+ takes $O(NV+E)$ time, if we include the initializations.

10.2.3.5 Solving for GMOD

Once we have constructed $\text{IMOD}^+(p)$ for each procedure p in the program, we must use it to compute GMOD(p) according to Equation 10.5, which we repeat here:

$$\text{GMOD}(p) = \text{IMOD}^+(p) \cup \bigcup_{s=(p,q)} \text{GMOD}(q) \cap \neg\text{LOCAL}$$

This equation implies that a variable x is in GMOD(p) if it is in $\text{IMOD}^+(p)$ or x is global and there is a nonempty path in the control-flow graph from p to procedure q where $x \in \text{IMOD}^+(q)$. In other words we have reduced the problem to a variant of the reachability problem in the call graph. It is well known that reachability can be solved in time linear in the size of the problem graph using a depth-first search algorithm based on Tarjan's algorithm for finding strongly-connected components of a directed graph. Algorithm *findGmod* in Figure 10.3 accomplishes this.

Since this is a direct adaptation of depth-first search it runs in $O(N+E)$ steps, where each step may involve a bit-vector operation of length V, the number of variables in the program. Thus the algorithm takes $O((N+E)V)$ elementary steps in the worst case.

> **Theorem 10.3.** Algorithm *findGmod* in Figure 10.3 correctly computes the GMOD(p) for every procedure p in the program.

Proof. This is a direct adaptation of Tarjan's algorithm for finding strongly-connected components. However, as the algorithm backs up in reverse invocation order, instead of collecting a set of strongly connected regions, it updates the GMOD(p) computation at each node. When it reaches the head of a strongly-connected component, it updates GMOD(u), for every u in the component, to include the nonlocal part of the GMOD set for the head. Thus, the global parts of GMOD(u) for every procedure u in the SCR are identical, as they should be. The GMOD sets for nodes that are not in loops are correct by virtue of the order of visit and the GMOD sets for nodes that are in loops are correct because of the updates of all procedures in the cycle. □

Taken together the results from this section and the previous one establish that the entire computation can be done in $O((N+E)V)$ steps. Since DMOD can be computed from GMOD in $O(NV+E)$ time, the complete computation of DMOD also takes $O((N+E)V)$ steps. This is the best possible time bound as you must evaluate Equation 10.5 at least once at every node in the call graph, which would require $O((N+E)V)$ time.

FIGURE 10.3 Algorithm for propagation of global modification side effects.

```
procedure findGmod(N, E, n, IMOD+, LOCAL)

    integer dfn[n], lowlink[n], nexdfn, p, q, d,
        IMOD+[n], GMOD[n], LOCAL;
    integer stack Stack;

    procedure search(p);
        dfn[p] := nexdfn; nexdfn:= nexdfn + 1;
        GMOD[p] := IMOD+[p]; lowlink[p] := dfn[p];
        push p onto Stack;

        for each q adjacent to p do
            if dfn[q] = 0 then /* tree edge */
                search(q);
                lowlink[p] := min(lowlink[p], lowlink[q])
            fi;

            if dfn[q] < dfn[p] and q∈Stack then
                lowlink[p] := min(dfn[q], lowlink[q])
            else /* apply equation */
                GMOD[p] := GMOD[p] ∪ (GMOD[q] ∩ ¬LOCAL)
            fi
        od;

        /* test for root of strong component */
        if lowlink[p] = dfn[p] then

            /* adjust GMOD sets for each member of the SCR */
            repeat
                pop u from Stack;
                GMOD[u] := GMOD[u] ∪ (GMOD[p] ∩ ¬LOCAL)
            until u = p
        fi
    end search;

    /* subroutine body */
    nexdfn := 1; dfn[*] := 0; Stack := ∅;
    search(1); /* by convention root = 1 */
end findGmod
```

FIGURE 10.4 Naive alias update of DMOD to produce MOD.

```
L1  for each call site s do
```

```
S1          MOD[s] := DMOD[s];
L2          for each x ∈ DMOD[s] do
L3              for each v ∈ ALIAS[p,x] do
S2                  MOD[s] := MOD[s] ∪ {v}
                od
            od
        od
```

10.2.4 Flow-Insensitive Alias Analysis

10.2.4.1 Update of DMOD to MOD

Once the direct modification side effect sets have been computed, there still remains the problem of factoring in aliasing. Recall the formula from Equation 10.2, which we repeat here.

$$\mathrm{MOD}(s) = \mathrm{DMOD}(s) \cup \bigcup_{x \in \mathrm{DMOD}(s)} \mathrm{ALIAS}(p, x)$$

A naive algorithm for implementing this conversion is shown in Figure 10.4. Let us analyze the complexity of this loop nest. The loop L_1 is executed O(E) times, where E is the number of edges in the call graph. The assignment in statement S_1 takes time proportional to the length of the bit vector representation or O(V) steps, where V is the number of variables in the program. Since this is done O(E) times, the running time attributable to this statement is O(EV). For each iteration of L_1, L_2 is executed once for each variable in DMOD(s). Since this could be essentially all the variables O(V), this loop is entered an aggregate of O(EV) times. Finally, L_3 is entered O(EV) times and, each time, it executes one iteration for each element of ALIAS(p,x). If we assume that ALIAS(p,x) might contain every variable in the program, the loop body could be executed O(EV^2) times. Finally, statement S_2 takes constant time, if we use a bit vector implementation. Thus the total running time for the update procedure is O(EV^2).

If we cannot improve the running time of alias analysis, it may dominate the running time of the entire side effect analysis, making it impractical for use on programs with nontrivial aliasing patterns. However, we can achieve a significant improvement by carefully classifying aliasing patterns that may arise. First, we note that, in a two-level naming hierarchy, containing only global and local variables, two global variables can *never* be aliases of one another. They can only be aliased to reference formal parameters. Thus, for a global variable x, ALIAS(p,x) can only contain reference formal parameters of procedure p. This means that ALIAS(p,x) can be no larger than μ entries, where μ is the maximum number of formal parameters of any procedure in the program.

On the other hand, for a given formal parameter f of procedure p, ALIAS(p,f) may contain any global or any other formal parameter. Thus, it may be of size O(V).

These observations suggest that we break down the update of DMOD to MOD into two cases: one for formal parameters and the other for global variables. When considering the aliases of a particular variable $x \in$ DMOD(s),

1. if x is a global variable, we will add a very small set ($\leq\mu$ elements) to MOD(s), but we may need to do this O(V) times, and
2. if x is a formal parameter of the procedure p containing s, we will add up to O(V) elements to MOD(s), but we will only do this a small number of times ($\leq\mu$).

In either case, the amount of work is O(V), not O(V^2). The entire algorithm is given in Figure 10.5 below.

FIGURE 10.5 Fast combination of side-effect and alias information.

```
for each call site s in the program do
    t := ∅;                          /* t is a temporary bit vector of length μ*/
    for each global variable x ∈ DMOD(s) do /* O(V) iterations*/
        t := t ∪ {x};                /* constant time*/
        t := t ∪ ALIAS[p,x]          /* O(μ) time*/
    od;

    MOD[s] := MOD[s] ∪ t;            /* O(V) time*/

    for each formal parameter f ∈ DMOD[s] do /* O(μ) iterations*/
        MOD[s] := MOD[s] ∪ {f};      /* constant time*/
        MOD[s] := MOD[s] ∪ ALIAS(p,f)/* O(V) time*/
    od
od
```

It should be clear from the previous discussion that the overall running time of the combining strategy in Figure 10.5 is O(V) per call site for a total of O(EV) time. This is promising because, if we can compute the ALIAS in O((N+E)V), the time bound for the entire MOD computation will be O((N+E)V).

10.2.4.2 Computing Aliases

We now turn to the problem of computing the sets ALIAS(p,x) for each procedure p and each variable x that is either global or a parameter of some procedure. To do this as rapidly as possible, we will once again take advantage of the observation that globals can only be aliases of formal parameters. We first compute, for each formal parameter in the program, an intermediate quantity $A(f)$, which is defined to be the set of global variables that may be aliased to a formal parameter f through a sequence of parameter bindings in the binding graph:

$$g \rightarrow f_0 \rightarrow f_1 \rightarrow f_2 \rightarrow \ldots \rightarrow f_{n-1} \rightarrow f_n = f$$

To compute $A(f)$, we will use a forward propagation on a variation of the binding graph in which cycles have been reduced to single nodes. Note that $A(f)$ is an approximation to ALIAS(p,f), containing all the global variables in that set. The algorithm for computing $A(f)$ for each formal parameter in the program is given in Figure 10.6 below. In this algorithm, the set $A(f)$ will be represented by a bit vector of length O(V).

FIGURE 10.6 Algorithm for computing approximate alias sets for formal parameters.

procedure *computeA*(N, E, N_B, E_B, A)

    for each formal parameter $f \in N_B$ **do** $A[f] := \varnothing$ **od**;
    for each call site $s \in N$ **do**
        for each global g mapped to formal parameter f at s **do**
            $A[f] := A[f] \cup \{g\}$
        od
    od;

    replace every cycle in the binding graph with a single node,
        reducing the graph to a directed, acyclic form;
    for each f in the reduced graph in topological order **do**

$$A[f] := A[f] \cup \bigcup_{(f_0, f) \in E_B} A[f_0]$$

    od;
    for each cycle in the original binding graph **do**
        let C be the reduced binding graph node for the cycle;
        for each $f \in C$ **do** $A[f] := A[C]$ **od**
    od
end *computeA*

It is easy to see that the forward propagation phase of this algorithm requires O(NV) steps in the worst case and updating the sets in a cycle can also be done O(NV) steps.

Once the sets $A(f)$ are available for every formal parameter in the program, the alias sets for every global variable can be computed by a simple inversion, as shown in Figure 10.7. This computation can be subdivided into two components: initializing the alias sets and computing the inversion. Recall that we can represent an alias set for a global variable within a given procedure as a bit vector of length μ. Therefore the initializations in step S_0 of Figure 10.7 take no more than $O(NV\mu) = O(NV)$ time. However, since an update can be done in constant time, the total cost of the updates in statement S_1 is also O(NV).

FIGURE 10.7 Inversion to compute ALIAS(g) for each global variable g.

```
        for each procedure p do
            for each global variable g do
S0              ALIAS[p,g] := ∅
            od
        od

        for each formal parameter f in the program do
            let p be the procedure for parameter f;
            for each g ∈ A[f] do
S1              ALIAS[p,g] := ALIAS[p,g] ∪ {f}
            od
        od
```

All that remains is the computation of aliases for formal parameters. Note that $A(f)$ is an approximation for ALIAS(p,f), containing all the global aliases of f. To expand $A(f)$ to ALIAS(p,f), we need only add the formal parameters that may be aliased to f. In the simple two-level language we are considering, the only formals that may be aliased to f are other parameters of the same procedure. Thus there can be no more than $\mu(\mu+1)/2$ formal pairs in any given procedure.

To compute the formals that may be aliased to one another, the algorithm *computePairs* in Figure 10.8 keeps track of the set FPAIRS(p) of pairs of formal parameters that may be aliased to one another in a given procedure p. After initializing all the FPAIRS sets to the empty set, we examine each call site for alias introductions, occurring when the same variable is passed to two different parameters. Whenever one is found, we add the formal parameter pair in the alias to a worklist. Then we iterate over the worklist looking for possible alias propagations, which occur when two parameters that may be aliased are both passed to another procedure. If the resulting pair of formals in the called procedure is not already in the FPAIRS set, the pair is added to the worklist. This procedure continues until the worklist is empty.

Let us now analyze the running time of algorithm *computePairs*. Note that there can be no more than $\mu(\mu+1)/2$ parameter pairs in any subroutine. The initialization loop L_1 takes $O(N)$ time, since it is constant time for each procedure. Since it must look at each call site, the loop at L_2 takes $O(E)$ time, but the operations are all constant-time, assuming we use some sort of linked structure for both W and FPAIRS. Loop L_3 is entered once for each pair that is put on the worklist. Since there can be at most $\mu(\mu+1)/2$ pairs for each call site, the total number of times L_3 is entered is less than or equal to $\mu(\mu+1)N/2$, so L_3 is entered $O(N)$ times. Similarly, each call site in the program is examined no more than $\mu(\mu+1)/2$ times, so loop L_4 is entered $O(E)$ times. Since the body of L_4 takes constant time, the entire process takes $O(N+E)$ time.

FIGURE 10.8 Algorithm to compute formal parameter pairs that may be aliased.

```
procedure computePairs
      W := ∅;
L1    for each procedure p in the program do FPAIRS[p] := ∅ od;
L2    for each alias introduction site (e.g., "CALL P(X,X)") do
          insert the resulting formal pair in FPAIRS[p] and
              in the worklist W;
      od;
L3    while W ≠ ∅ do
          remove ⟨f1,f2⟩ from W;
L4        for each call site s in the procedure p passing both f1 and f2 do
              let q be the procedure invoked at s;
              if f1 and f2 are passed to f3 and f4 at s and
                  ⟨f3,f4⟩ ∉ FPAIRS[q] then
                  FPAIRS[q] := FPAIRS[q] ∪ {⟨f3,f4⟩};
                  W := W ∪ {⟨f3,f4⟩};
              fi
          od
      od
end computePairs
```

Thus, we have established that alias analysis can be done in $O((N+E)V)$ time, which means that the entire MOD solution can be completed in this time.

10.2.5 Constant Propagation

Constant propagation is an important data-flow analysis problem that improves the performance of a variety of optimizations [36,42]. Unfortunately the problem is difficult even in a single procedure—obtaining a precise solution has been shown unsolvable [58]. Even the usual single-procedure approximate problem is flow-sensitive in the interprocedural setting, hence intractable [56].

One important reason for the difficulty is that constants propagated into a program region can make it possible to evaluate program expressions in that region, yielding new constants on exit. An example illustrating the interprocedural case is shown below.

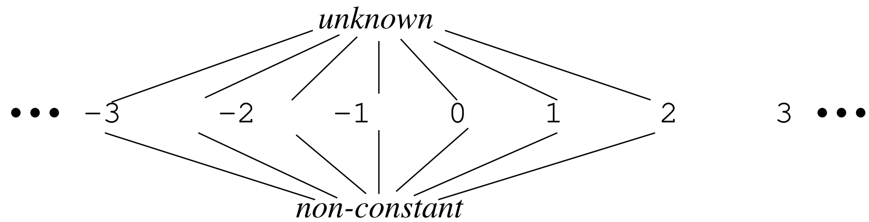
```
SUBROUTINE PHASE(N)
   INTEGER N,A,B
   CALL INIT(A,B,N)
   CALL PROCESS (A,B)
END

SUBROUTINE INIT(A,B,N)
   INTEGER A,B,N
   A = N+1
   B = (N*A)/2
END
```

The purpose of subroutine `INIT` is to initialize the variables `A` and `B`. Thus if `N` is a constant 10 on entry to procedure `PHASE`, `A` will be a constant 11 and `B` will be a constant 55 on exit from `INIT`. Hence these constant values will be available on entry to `PROCESS`. This section will develop an approach to determine these facts.

A single-procedure constant propagation algorithm was presented in Section 4.4.3 on page 132. This algorithm, which is shown in Figure 4.4 on page 133, uses an iterative process on the use-definition graph or the static single assignment (SSA) form [32]. The iterative method is guaranteed to converge because it is based on a constant propagation lattice shown in Figure 10.9. Furthermore, the algorithm is linear in the sixe of the base graph upon which it operates, because an instruction can have its output value lowered in the lattice at most twice and, hence, it is put on the worklist at most twice.

FIGURE 10.9 Constant propagation lattice.



Our strategy is to develop an interprocedural analog of the iterative single procedure constant propagation. In fact, we hope to use the same algorithm, but on a different use-definition graph that we call the *interprocedural value propagation graph*. In this graph, the vertices represent "jump functions" that compute values out of a given procedure from known values into a procedure. Jump functions are analogous to "transfer functions" from intraprocedural and interprocedural interval analysis [43].

Let *s* be a call site within procedure *p* and *x* a formal parameter of the procedure *q* called at *s*. The *jump function* for *x* at *s*, denoted J_s^x, determines the value of *x* in terms of the values of inputs to the procedure *p* containing *s*. The *support* of J_s^x is the set of inputs actually used in the evaluation of J_s^x. Jump functions can be computed in a preliminary phase that examines the individual procedures of the program.

We now return to the construction of the *interprocedural value graph*. Here are the steps:

1. construct a node for each forward jump function J_s^x
2. if $x \in$ support(J_t^y), where *t* is a call site in the procedure called at *s*, then construct an edge from J_s^x to J_t^y.

The iterative constant propagation algorithm can be applied to the resulting graph.

We now present a simple example of this process. Consider the program shown in Figure 10.10. From this example we can easily derive the jump functions:

$$J_\alpha^X = \{1\};\ J_\alpha^Y = \{2\}$$

$$J_\beta^U = \{\texttt{X+Y}\};\ J_\beta^V = \{\texttt{X-Y}\}$$

The call graph and the resulting interprocedural value propagation graph are shown in Figure 10.11.

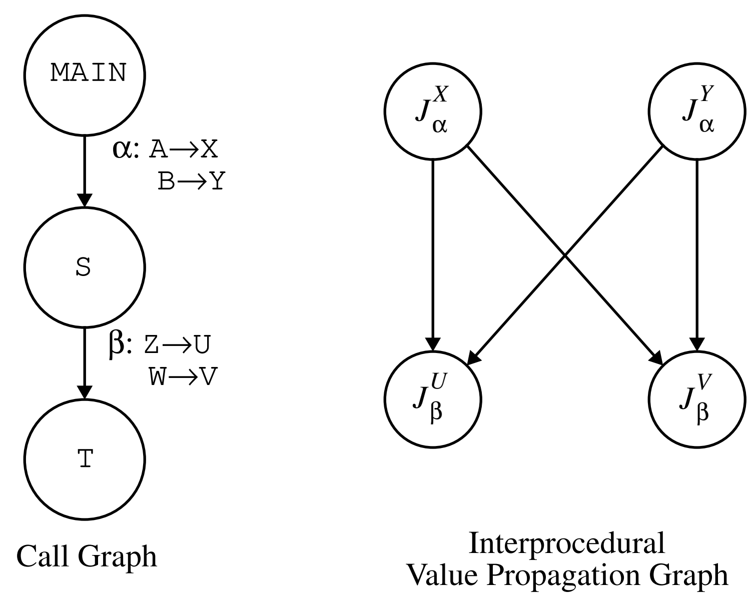
FIGURE 10.10 Interprocedural constant propagation example.

```
     PROGRAM MAIN
        INTEGER A,B
        A = 1
        B = 2
α       CALL S(A,B)
     END

     SUBROUTINE S(X,Y)
        INTEGER X,Y,Z,W
        Z = X + Y
        W = X - Y
β       CALL T(Z,W)
     END

     SUBROUTINE T(U,V)
        PRINT U,V
     END
```

FIGURE 10.11 Interprocedural value propagation graph example.



It can easily be seen that the constant propagation algorithm applied to the interprocedural value propagation graph in Figure 10.11 will quickly converge to the constant assignments:

```
X=1; Y=2; U=3; V=-1
```

To estimate the total cost of the algorithm, recall that the cost of the iterative constant propagation algorithm on which this is based is proportional to the number of vertices and the number of edges in the graph, assuming the jump function evaluations can be done in constant time. However, it is unrealistic to expect every jump function to be evaluated in constant time because, as we shall see, different strategies for constructing jump functions produce functions of varying execution costs.

Thus, we must argue about the number of times that a jump function is evaluated. A jump function J has *support*(J) inputs, each of which can be lowered at most twice. Thus a jump function J can be evaluated no more than $O(|support(J)|)$ times. Let *cost*(J) denote the cost of executing jump function J. For each jump function the total cost of execution will be $O(|support(J)| \cdot cost(J))$. Therefore the total cost of executing the interprocedural constant propagation algorithm is

$$O\left(\sum_{s}\sum_{x}\left|support(J_s^x)\right| \cdot cost(J_s^x)\right) \qquad \text{(EQ 10.7)}$$

where s ranges over the call sites in the program and x ranges over the input parameters to the subroutine.

10.2.5.1 Construction of Jump Functions

Jump functions can vary widely in precision and cost of the approximation. A jump function could involve full symbolic interpretation of the procedure it represents. On the other end of the spectrum, it could evaluate only the assignments that are on every path through the subroutine and not contained in any loop. In this case, variables assigned on optional control paths or in loops would receive the value $\perp$ (bottom) in the constant lattice.

One major problem is how to construct jump functions for procedures that have function calls within their bodies. Consider the example in Figure 10.12. To build a jump function for `T` at call site γ, we need to know what action will be taken by subroutine `INIT` invoked at call site β. We address this problem by defining “return jump functions,” which summarize the constants propagated out of a subprogram when it is called with a particular set of inputs.

If x is an output of the procedure p, the *return jump function* R_p^x determines the value of x on return from an invocation of p in terms of the values of input parameters to p. The *support* of R_p^x is the same as the support of a forward jump function. In the simple case of the subroutine `INIT` shown in Figure 10.12 above, we have the following return jump function:

$$R_{\mathrm{INIT}}^{\mathrm{X}} = \{\mathtt{2*Y}\}$$

and in the case of SOLVE, we have:

$$R_{\mathrm{SOLVE}}^{\mathrm{C}} = \{\mathtt{T*10}\}$$

FIGURE 10.12 A complex interprocedural constant folding example.

```
     PROGRAM MAIN
        INTEGER A
α       CALL PROCESS(15,A)
        PRINT A
     END

     SUBROUTINE PROCESS(N,B)
        INTEGER N,B,I
β       CALL INIT(I,N)
γ       CALL SOLVE(B,I)
        RETURN
     END

     SUBROUTINE INIT(X,Y)
        INTEGER X,Y
        X = 2*Y
        RETURN
     END

     SUBROUTINE SOLVE(C,T)
        INTEGER C,T
        C = T*10
        RETURN
     END
```

We can use return jump functions in the normal jump functions for constant propagation. For example, the jump function for call site γ is as follows

$$J_{\gamma}^{\mathrm{T}} = \{\mathbf{if}\ \mathtt{I} \in \mathrm{MOD}(\beta)\ \mathbf{then}\ R_{\mathrm{INIT}}^{\mathrm{X}}(\mathtt{N})\ \mathbf{else}\ \mathit{undefined\text{-}const}\}$$

where *undefined-const* is used to signify a special value given to uninitialized variables. To round out the example, we present the remainder of the jump functions:

$$J_{\alpha}^{\mathrm{N}} = \{\mathtt{15}\} \text{ and } J_{\beta}^{\mathrm{Y}} = \{\mathtt{N}\}$$

One important jump function remains to be determined, namely the return jump function for subroutine `PROCESS`, which is the key to determining whether a constant can be substituted for the variable `A` in the print statement in the main program. This return jump function must invoke the return jump function for both `INIT` and `SOLVE` but the call to the return jump function for `INIT` will be automatic if we use the forward jump function to determine the value of the input formal parameter `T` of `SOLVE`. The resulting function is given below:

$$R_{\mathrm{PROCESS}}^{\mathrm{B}} = \begin{array}{l}\{\mathbf{if}\ \mathtt{C} \in \mathrm{MOD}(\gamma) \\ \ \mathbf{then}\ R_{\mathrm{SOLVE}}^{\mathrm{C}}(J_{\gamma}^{\mathrm{T}}(\mathtt{N})) \\ \ \mathbf{else}\ \mathit{undefined\text{-}const}\}\end{array}$$

Using these jump functions we can see that the value of the variable `A` on exit from the procedure `PROCESS` is given by

$$R_{\mathrm{PROCESS}}^{\mathrm{B}} = (2*(15))*10 = 300$$

The study of scientific Fortran programs by Grove and Torczon [36] suggests that MOD should be computed prior to computing jump functions and prior to performing a global constant propagation to initialize the interprocedural propagation, because in any case where a variable is not in MOD for a given call site, the return jump function is simply the identity. In these cases, a return jump function is not needed, which simplifies the construction of other jump functions for the same variable whose scope includes the given call site. This is the single most useful type of return jump function. The same study also established that:

1. simple jump functions get almost all the constants in scientific Fortran, and
2. return jump functions only infrequently deliver new constants but, when they do, the payoff is high.

10.2.6 Kill Analysis

We can adapt some of the ideas used to solve the MOD and constant propagation problems to deal with kill analysis as well. Recall that KILL(p) is the set of all variables that must be modified on every path through procedure p. For convenience in the algorithm formulation, we will compute NKILL(p) = ¬KILL(p), the set of all variables that are not killed as a side effect of calling procedure p. A method similar to the one presented here can be used to compute USE(p) for every procedure p in the program.

Let us begin by considering how we would solve the single-procedure NKILL problem. Suppose we have, for each extended basic block b and each successor c of b, the set THRU(b,c) of all variables that are not killed on some path through b to c. Then the following set of data-flow equations can be used to solve for NKILL(b):

$$\mathrm{NKILL}(b) = \bigcup_{c \in succ(b)} \mathrm{THRU}(b, c) \cap \mathrm{NKILL}(c) \qquad \textbf{(EQ 10.8)}$$

If e is the exit node for the procedure then

$$\mathrm{NKILL}(e) = \Omega \qquad \textbf{(EQ 10.9)}$$

where Ω denotes the set of all variables. This set of equations can be solved using the simple iterative method of data-flow analysis.

An analogous process can compute NKILL(p) for every procedure p in the program. First we need a formula for computing NKILL(p) for a single procedure. This is more complicated than the case for single blocks, because it depends on procedure-specific control flow. To carry out the computation, we will first construct the *reduced control-flow graph* GTHRU for a procedure. GTHRU is a graph in which each vertex is a call site, the entry node or the exit node for the procedure. Every every edge (x,y) in GTHRU is annotated by the set THRU(x,y) of variables that are not killed on some path from x to y not containing a call site. GTHRU is constructed by the algorithm in Figure 10.13.

FIGURE 10.13 Algorithm for constructing the reduced control-flow graph.

```
remove all back edges from the control-flow graph;
let b0 denote the procedure entry node;
mark b0 processed;
worklist := ∅;
for each s∈successors(b0) do worklist := worklist ∪ {(b0,s)} od;

while worklist ≠ ∅ do
    take an arbitrary element (b,s) from the worklist,
        such that all predecessors of s have already been processed
        or merged into b;
    if s is a call site then
        for each t ∈ successors(s) do
            worklist := worklist ∪ {(s,t)} od;
        mark s as processed;
    else if s is the exit node then do nothing
    else /* s is normal node */
        merge s into b;
        for each t ∈ successors(s) do
            if THRU[b,t] is undefined then
                THRU[b,t] := THRU[b,s] ∩ THRU[s,t]
            else
                THRU[b,t] := THRU[b,t] ∪ (THRU[b,s] ∩ THRU[s,t])
            fi
        od
    fi
od
```

It is easy to see that the algorithm in Figure 10.13 can be implemented in time linear in the size of the control flow graph using a variant of topological sort.

Once we have the reduced control-flow graph, the algorithm shown in Figure 10.14 can be used to compute NKILL(p) for a procedure p given NKILL values for all procedures that can be called from within p .

FIGURE 10.14 Computing NKILL(p).

```
for each b in GTHRU(p) in reverse topological order do
    if b is the exit node then NKILL[b] := Ω
    else if b is a call site then
        NKILL[b] := ∅;
        for each successor s of b in GTHRU(p) do
            NKILL[b] := NKILL[b] ∪ NKILL[s]
        od;
        NKILL[b] := NKILL[b] ∩ NKILL[q],
            where q is the procedure called at b
    else /* b is the entry node for GTHRU(p) */
        NKILL[b] := ∅;
        for each successor s of b in GTHRU(p) do
            NKILL[b] := NKILL[b] ∪ NKILL[s]
        od;
        NKILL[p] := NKILL[b]
    fi
od
```

A simple iterative data-flow analysis algorithm, employing the procedure in Figure 10.14, can be used to compute NKILL, assuming that all variables in the program are global. Although this algorithm can take $O(N^2V)$ time in the worst case, if the call graph is reducible, the algorithm will require only $O((N+E)d)$ bit-vector steps, where d is the maximum number of back edges in any noncircular path in the call graph [47]. The total time required in this case is $O((N+E)dV)$. In practice, iterative methods are very fast.

The algorithm described so far can compute NKILL(p) rapidly only if there are no reference formal parameters in the program. If there are reference formal parameters, the algorithm can be adapted to produce the correct answers if the appropriate actual-to-formal mappings are observed. However, it may take significantly longer to converge because of the possibility of "shift-register effect" in which parameters are passed to other parameters in a recursive loop. This is the same problem we observed in dealing with the MOD problem in Section 10.2.3 on page 317.

FIGURE 10.15 Construct and mark the binding graph for NKILL.

```
initially, let the binding graph consist of a vertex
        for each formal parameter in the program;
worklist := ∅;

for each procedure p in the program do
        let NKILL0[p] be the result of applying the algorithm in
             Figure 10.14 on page 340 with NKILL[q] = Ω(q)
             for each successor q of p,
             where Ω(q) denotes the set of formal parameters of q;
        NKILL[p] := NKILL0(p);
        for each formal parameter f of p do
             if f ∈ NKILL0[p] then
                  killed(f) := false;
                  for each formal parameter g to which f is passed
                       at a call site within p do
                       add an edge (f,g) to the binding graph
                  od
             else
                  killed(f) := true;
                  worklist := worklist ∪ {f};
             fi
        od
od;

while worklist ≠ ∅ do
        select an arbitrary element f ∈ worklist;
        worklist := worklist – {f};
        for each g such that there is an edge (g,f) in the binding graph
             and killed(g) = false do
             let q be the procedure of which g is a formal;
             NKILL[q] := the result of applying the algorithm in
                  Figure 10.14 with the current NKILL sets
                  for its successors;
             if g ∉ NKILL[q] then
                  killed(g) := true;
                  worklist := worklist ∪ {g};
             fi
        od
od
```

Fortunately, we can use the same general technique—use of a formal parameter binding graph—to ameliorate this problem. We will construct a binding graph and mark it as shown in Figure 10.15.

This algorithm is simply a variation on the iterative algorithm, except that we update NKILL(p) for a procedure only if we discover that the status of one of its parameters might have changed to “killed” because a parameter that it is passed at some call site has changed its status. Since each parameter can be added to the worklist only once and since the algorithm visits each predecessor of a parameter taken from the worklist, the total number of NKILL updates is limited to $O(E_B+N_B) = O(E+N)$, where E_B and N_B are the number of edges and vertices in the binding graph, respectively and E and N are the number of edges and vertices in the call graph. The number of updates can be further reduced by carefully selecting the order of extraction of elements from the worklist.

We note that the kill sets computed by this process are not precise because they do not take aliasing into account. Suppose we know that on every visit to a procedure p the variables $x_1, x_2,...,x_n$ must all refer to the same location, then a variable is in KILL(s) if one of the aliases is killed on every path through the procedure called at s and any procedures invoked from within it. In other words, we may be able to increase the size of the KILL sets by taking aliasing into account. However, this will be computationally expensive, because it requires computing the set of all tuples of variables that refer to the same location on some invocation of the procedure, which could take time that is exponential in the number of variables used as parameters in the program.

10.2.7 Advanced Analysis

In addition to the basic analysis problems discussed so far, successful program parallelization requires that some more sophisticated interprocedural problems be solved. Two that are especially important are symbolic analysis and array section analysis [46,42].

10.2.7.1 Symbolic Analysis

Although constant propagation provides valuable information about variables on entry to a procedure, it is usually not possible to establish that a given variable is constant because, in most cases, it is not. However, it may not be necessary to prove a variable constant to improve global program analysis. It may be enough to establish bounds on its values or to prove that its value can be expressed as a function of the values of other variables at the same point in the program. Such symbolic relationships can be used to prove facts, such as the absence of a dependence. For example, consider the sample code below:

```
SUBROUTINE S(A,N,M)
   REAL A(N+M)
   INTEGER N, M
   DO I = 1, N
      A(I+M) = A(I) + B
   ENDDO
END
```

If we could prove that `N=M` on entry to `S`, we could show that the loop within the subroutine carries no dependence.

The goal of *interprocedural symbolic analysis* is to prove facts about the values of variables that may hold on entry to a given subroutine or on return from a given call site. There are three types of symbolic analyses that are often carried out:

1. *symbolic expression analysis,* which seeks to determine a symbolic expression for the value of a variable in terms of the values of other variables at the same point in the program;
2. *predicate analysis*, which seeks to establish a relationship between values that a pair of variables may have at a given point in the program and
3. *range analysis*, which seeks to establish a range of values with known constant lower and upper bounds (and possibly strides) that a variable may take on at a given point in the program.

Often, the results of these analyses can be substituted for one another. For example, the fact that `M=N` in the example above, could be established by either symbolic expression analysis or predicate analysis. The distinction between the two is that expression analysis can produce values that involve many other values, while predicate analysis typically relates only pairs of variables. On the other hand, the analysis that produces symbolic expressions can be more complicated than the analysis required for simple predicates.

Range analysis can be used effectively in program analysis to rule out certain possibilities. For example, consider the subroutine:

```
SUBROUTINE S(A,N,K)
   REAL A(0:N)
   INTEGER N, K
   DO I = 2, N
      A(I) = A(I) + A(K)
   ENDDO
END
```

If we can prove that K $\in$ [0:1] on entry to the subroutine, we can establish that the loop carries no dependence.

Within a single procedure, symbolic expression analysis is typically performed using some form of *value numbering* [4,60], which uniquely numbers each expression so that expressions with equal value numbers at a given point will have equal values at run time. Since global program value numbering is likely to be complex, it is typically used only within a procedure. Interprocedurally, a restricted set of relationships, such as those represented by predicates involving only two variables, are propagated across procedure boundaries [45].

It is easy to see that range analysis and symbolic expression analysis can be handled by variations on the constant propagation algorithm from Section 10.2.5. To use this algorithm, we will need to define three things:

1. the process by which new symbolic information is introduced in a program;
2. *jump functions* which produce information at a call site from information at the entry to a procedure containing that call site; and
3. *return jump functions* which determine the relationships on output from a procedure given the relationships that hold on entry.

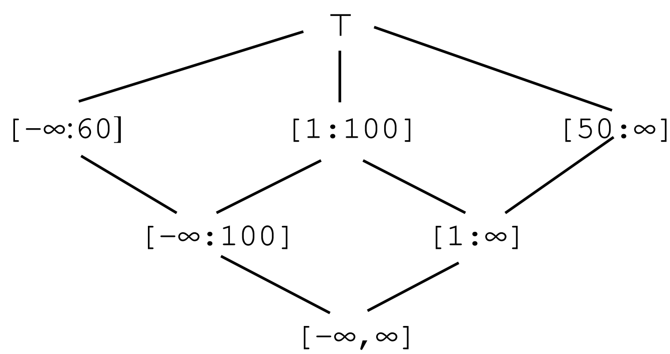
Let us consider how these three functions could be put together to compute range information. Range information is typically introduced at control-flow points in the program. For example, in a loop headed by

```
DO I = 1,N
```

it is safe to assume that I $\in$ [1:N]. Similarly, conditional statements introduce partial ranges that can be composed to produce full ranges.

In all the symbolic analysis methods, jump functions compute values in lattices that are significantly more complex than the simple constant propagation lattice. For example, in the case of range analysis, we might use a lattice in which the meet operation picks the larger upper and smaller lower bound of the pair of ranges on the two joining control-flow branches. A fragment from such a lattice is depicted in Figure 10.16. If we bound the number of times that an upper bound can be increased before being taken as ∞ and we similarly bound the number of times a lower bound can be decreased, this lattice can be used anywhere a finite descending chain lattice is required, such as in an iterative algorithm.

FIGURE 10.16 Simple value lattice for range analysis.



Symbolic predicate analysis is more complicated than range analysis because it examines the relationship between a pair of variables. For example, it is useful to know whether two variables x and y are related by equality or are offset from one another by a constant. This might be represented as

$$\mathrm{x} - y = c$$

where c is a compile-time constant. A more general linear relationship between variables can be characterized as:

$$c_1 x + c_2 y = c_0$$

where c_0, c_1 and c_2 are all constants. Note that this relationship is transitive—if variables y and z are related by

$$d_1 y + d_2 z = d_0$$

then we can find constants e_0, e_1 and e_2 such that

$$e_1 x + e_2 z = e_0$$

In particular,

$$c_1 d_1 x - c_2 d_2 z = c_0 d_1 - c_2 d_0$$

Thus, at any point in the program, we can find groups of variables that are linearly related to one another. The goal of symbolic predicate analysis is to propagate these sets throughout the program. This can also be done by employing a variant of interprocedural constant propagation.

10.2.7.2 Array Section Analysis

The analysis presented so far does not help much with one of the most important problems we need to solve if we are to automatically parallelize programs—namely, how to analyze dependences in loops that contain procedure calls. Consider the following code:

```
      SUBROUTINE S
         DIMENSION A(100,100)
         ...
         DO I = 1,N
S1          CALL SOURCE(A,I)  ! assigns to A
S2          CALL SINK(A,I)    ! uses A
         ENDDO
         RETURN
      END
```

If we wish to parallelize the loop in this subroutine, we must determine whether any dependence is carried by the loop. Interprocedural information of the sort described in previous sections is of little use, telling us only that array `A` is modified by `SOURCE` and used by `SINK`. With no better information than that, we must assume that there is an assignment in `SOURCE` to a location that is used by `SINK` on a later iteration of the loop—in other words, we must assume that the loop carries a dependence and cannot be parallelized.

The situation would be different if we were able to show that the accesses to `A` in both routines are confined to the Ith column, which is suggested by the use of `I` as a parameter to both routines. Then we know that different iterations of the loop deal with distinct portions of the array, so no carried dependence is possible. We would like to refine interprocedural analysis methods to be able to establish conditions like this, which means that our analysis needs to be able to recognize subarrays of the whole array.

Suppose that we are able to compute the set $M_A(I)$ of locations within the array that may be modified within `SOURCE` on the Ith iteration of the loop and the set $U_A(I)$ of locations that may be used in `SINK` on iteration I. These quantities might by computed by array versions of MOD and USE respectively. Then the loop carries true dependence if and only if there exist indices I_1 and I_2, $1 \le I_1 < I_2 \le N$, such that

$$M_A(I_1) \cap U_A(I_2) \neq \emptyset$$

In order to reason about subarrays, we need a method of representing them. The representation should be such that unions and intersections are reasonably easy to represent as well.

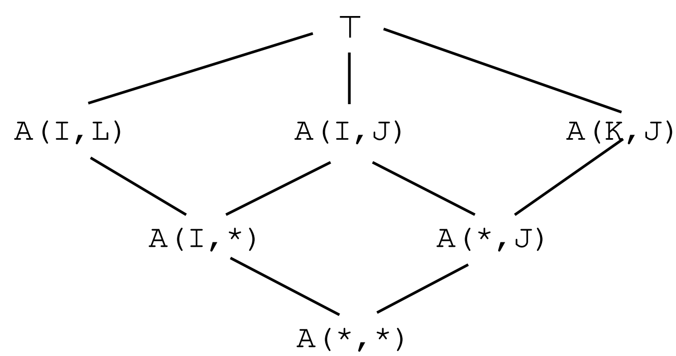
It is straightforward to extend the standard data-flow algorithms, which work on vectors of bits in which each bit can represent only two states (e.g., may be modified or must be preserved), to vectors of more general lattice elements. If we can find a lattice that represents subarrays accurately enough, we can use this lattice in our interprocedural data-flow analysis routines to determine side effects to subarrays.

Some important properties that a lattice representation should have are as follows:

1. the representation should be as accurate as possible;
2. the *meet* operation, which is invoked whenever two control-flow paths merge, must be efficient;
3. the dependence test, which usually involves intersection of region representations should also be efficient;
4. it should be possible to handle recursion in the analysis framework, which implies that the lattice should have the *finite descending chain property*—that is, every descending chain in the lattice must reach a lattice minimum after a finite number of steps; and
5. it should be possible to deal with the parameter transformations that occur at call sites.

Let us consider one possible lattice for subarrays, depicted in Figure 10.17 below, that satisfies a number of these requirements. The elements of this lattice are referred to as *simple regular sections* because they can represent a very limited number of regular subarrays, namely points, rows, columns and the entire matrix.

FIGURE 10.17 Simple regular section lattice.



Note that this lattice may extend to an infinite size because we can use arbitrary variables and constants in the subscripts. However, it has the finite descending chain property because no element in the lattice can be "lowered" more than three times.

Evaluating this lattice representation, we find that meet, which represents union in a case like the MOD calculation, has the following properties:

1. The depth of the lattice is k+1, where k is the number of subscript positions in the array represented.
2. The cost of a meet operation, which means "union" in the MOD calculation, is O(k), because each subscript position must be examined and compared for the two references to determine what the meet must be.
3. Intersection, which is essential to the dependence test, is a limited form of unification, which can also be done in linear time in the number of subscripts.

To see this last claim, note that each subscript position in the lattice is either a symbolic expression or a constant or "*", which represents an entire row or column. Thus, if the two subscripts are symbolic expressions then the result subscript should be the same expression if they are equal and "*" otherwise. If one is an expression and the other is "*", the result subscript should be"*".

One question remains: How accurate is this representation? It turns out that in practice this representation is too simple because it does not allow subarrays whose extents are bounded. Thus in an array `A(100,100)` the best approximation to subarray `A(1:10,1:10)` is `A(*,*)`. A much better representation is *bounded regular sections*, in which upper and lower bounds for each dimension are permitted. These can be thought of as any section represented by Fortran 90 triplet notation in which the stride is 1. Widely-used extensions include arbitrary stride triplet notation and triangular subarrays.

Interprocedural algorithms like the MOD solution can be directly adapted to deal with vectors of lattice elements, so long as the lattice has the finite descending chain property. The component of the MOD algorithm on the binding graph will converge because no formal parameter can be put on the worklist more than k times, where k is the maximum depth of the lattice. The reachability portion of the algorithm converges because, when a strongly-connected region is found, the element-vector for each element of the region is set to the minimum lattice element in the region.

Note that in the four side-effect problems described so far—MOD, REF, NKILL and USE—we always want to over-approximate the array section involved because we will optimize only when we *know* a variable cannot be in one of those sets for the call site in question. Consider the privatization example in Section 10.2.1.5 on page 310. We will be able to make a variable `P` private to a loop if we discover that, for the call site `S0` at the beginning of the loop:

$$\mathtt{P} \in (\mathrm{KILL}(\mathtt{S0}) \cap \neg\mathrm{USE}(\mathtt{S0})).$$

In other words, we optimize when

$$\mathtt{P} \notin (\mathrm{NKILL}(\mathtt{S0}) \cup \mathrm{USE}(\mathtt{S0}))$$

so we must overestimate rather than underestimate those sets to ensure that inaccuracy results only in missed opportunities, not in incorrect code.

10.2.8 Call Graph Construction

So far, we have assumed a precise call graph on which to solve interprocedural data-flow analysis problems. At first blush, it seems easy to construct such a graph—simply examine each procedure in the program and, for each call site, construct an edge from the calling procedure to the called procedure. This simple approach works well, so long as there are no procedure parameters. On the other hand, if there are procedure parameters, we can have code like this:

```
      SUBROUTINE SUB1(X,Y,P)
         INTEGER X, Y
S0       CALL P(X,Y)
         RETURN
      END
```

The problem here is determining what the called procedure might be at call site S_0. If there is only one invocation sequence for SUB1 it may be simple to follow the call chain back to determine the procedure passed to parameter P. However, we cannot assume that a single chain exists, because procedure parameters were added to the language to ensure that many different procedures could be passed to P, even in the same program.

To solve this problem, we must be able to determine, for each procedure parameter P, the names of procedures that may be passed to P, directly or indirectly. However, we must be careful to avoid loss of precision in cases where more than one procedure parameter is passed. Consider the following example:

```
      SUBROUTINE SUB2(X,P,Q)
         INTEGER X
S1       CALL P(X,Q)
         RETURN
      END
```

Suppose we have the following two calls:

```
      CALL SUB2(X,P1,Q1)
      CALL SUB2(X,P2,Q2)
```

where P1 and P2 simply invoke their procedure parameter on their integer parameter and return:

```
      SUBROUTINE P1(X,Q)
         INTEGER X
S2       CALL Q(X)
         RETURN
      END
```

Then at call site S_1 in subroutine SUB2, we can pass procedure Q1 to procedure P1 or we can pass procedure Q2 to procedure P2. We can *never* pass Q1 to P2 or Q2 to P1. In other words, P1 can only call Q1 and P2 can only call Q2 in this program. However a naive procedure-tracking scheme that simply maintains lists of procedures that could be passed to a given parameter might report edges from P1to Q2 and from P2 to Q1 because the list for possible procedures passed to parameter `Q` in `P1` includes both `Q1` and `Q2`.

To overcome this problem, a precise call graph construction algorithm must keep track of which pairs of procedure parameters may be *simultaneously* passed to the procedure formal parameters in `S2`. This suggests a general algorithm for call graph construction.

FIGURE 10.18 Algorithm for computing procedure parameter tuples.

for each procedure p in the program **do** PROCS(p) := ∅ **od**;
W := ∅;
for each call site s in the program **do**
    if the call site passes procedure names to all procedure
        parameters of the called procedure **do**
        let $t = \langle N_1, N_2, \ldots, N_k \rangle$ be the tuple of procedure names passed
            in order of the parameters to which they are passed;
        $W := W \cup \{\langle t,p \rangle\}$, where p is the procedure called
    fi
od;

while $W \neq \emptyset$ **do**
    let $\langle t = \langle N_1, N_2, \ldots, N_k \rangle, p \rangle$ be an arbitrary element of W;
    $W := W - \{\langle t,p \rangle\}$;
    PROCPARMS[p] := PROCPARMS[p] ∪ {t};
    let $\langle P_1, P_2, \ldots, P_k \rangle$ be the set of procedure parameters to which
        the elements of the tuple $t = \langle N_1, N_2, \ldots, N_k \rangle$ are mapped;
    for each call site s within p where P_i for some i, $1 \le i \le k$,
        is passed as a procedure parameter **do**
        let $u = \langle M_1, M_2, \ldots, M_k \rangle$ be the set of procedure names
            passed to the procedure q called at s, where each M_i is either
            the procedure name in the i*th* position or
            N_j if P_j is passed in the i*th* position;
        if $u \notin$ PROCPARMS[q] **then** $W := W \cup \{\langle u,q \rangle\}$ **fi**
    od
od

Suppose we collect, for every procedure p that accepts procedure parameters, a set PROCPARMS(p) of tuples of procedure names that may simultaneously be passed to p, where the order of the procedure names in the tuple corresponds to the order of the procedure parameters in the parameter list of p. Then the iterative algorithm in Figure 10.18 can be used to determine the correct set of procedure parameter tuples passed at each call site.

> **Theorem 10.4.** For every procedure p in the program, the algorithm in Figure 10.18 produces PROCPARMS(p) such that $\langle N_1,N_2,...,N_k\rangle \in$ PROCPARMS(p) if and only if there exists a call chain that passes parameter names $N_1,N_2,...,N_k$, in that order, to the procedure parameter positions for p.

Proof. Termination. If, for each procedure p, ν_p is the limit on the number of parameters and if N_P is the total number of procedure names passed as parameters in the program, then the total number of possible tuples in the program is

$$\sum_p N_P^{\nu_p} \tag{EQ 10.10}$$

which is clearly finite. Since no tuple is put on the worklist more than once and there can be only a finite number of tuples and one tuple is processed on each iteration of the while-loop, the algorithm must terminate.

If. Suppose there exists a call chain $p_0,p_1,...,p_l$ such that p_0 is the main program and $p_l = p$ that passes $N_1,N_2,...,N_k$ to the procedure parameters of p and suppose that $\langle N_1,N_2,...,N_k\rangle \notin$ PROCPARMS(p). At each p_i in the chain, there must be a tuple t_i of procedure names. Assume without loss of generality that $p = p_l$ is the first procedure in the chain for which the input tuple is not in PROCPARMS. That is, $t_i \in$ PROCPARMS(p_i) for all i such that $0 \le i < l$. If $l = 1$, then the parameters $N_1,N_2,...,N_k$ are names that are explicitly passed to p in the main program (which can have no procedure parameters). In this case, $\langle N_1,N_2,...,N_k\rangle$ is added to PROCPARMS(p) by the initialization loop. As this would be a contradiction of our assumption, it must be the case that $l > 1$. Therefore $t_{l-1} = \langle M_1,M_2,...,M_j\rangle$ must be taken from the worklist and put into PROCPARMS(p_{l-1}) at some point in the execution of the algorithm. At that point $\langle N_1,N_2,...,N_k\rangle$ must be put on the worklist for the call site invoking p because each N_i passed to p must either be an explicit procedure name or M_n, where n is the index of the procedure parameter of p_{l-1} that is passed in the ith position to p.

Only if. Suppose $\langle N_1,N_2,...,N_k\rangle \in$ PROCPARMS(p). Either $N_1,N_2,...,N_k$ are all explicit procedure names, in which case they are passed directly to p at some call site s (which would establish the result), or there is at least one procedure parameter name in the list. In the latter case, $\langle N_1,N_2,...,N_k\rangle$ must have been added to PROCPARMS(p) when $\langle\langle N_1,N_2,...,N_k\rangle,p\rangle$ was taken from the worklist W. This must have been put on to the worklist because there exists a procedure q that calls p with $N_1,N_2,...,N_k$ as parameters when q is invoked with $M_1,M_2,...,M_j$ and $\langle M_1,M_2,...,M_j\rangle \in$ PROCPARMS(q). Since there can only be a finite number $\leq N^{\mu}$ of different tuples of procedure names and since each tuple may be put on the worklist at most once, we must eventually work back to a call with explicit procedure names. Then the sequence of procedures visited during this process, taken in reverse order, form the call chain. This establishes the result. □

We saw above that the number of tuples is given by the summation in equation Equation 10.10. Let $\nu_{max} = max_p(\nu_p)$ and let N_C be the number of procedures with at least one procedure parameter and let N_P be the number of procedure names in that are passed to a procedure somewhere in the program. Then the running time can be approximated as

$$\sum_p N_P^{\nu_p} = O(N_C N_P^{\nu_{max}}) \leq O(NN^{\nu_{max}}) = O(N^{\nu_{max}+1}) \qquad \text{(EQ 10.11)}$$

where N is the number of procedures in the program. In the special case where there is no more than one procedure parameter to any procedure in the program, the running time is $O(N^2)$. In typical Fortran usage, the running time will not be a significant factor, because the use of procedure parameters is limited. However, for languages with more complex usage patterns, there exist approximate algorithms that run in linear or near-linear time in the size of the call graph [39,38].

10.3 Interprocedural Optimization

10.3.1 Inline Substitution

The most familiar interprocedural optimization is inline substitution, by which the text of a subroutine is substituted at the point of call, with formal parameters replaced by actual parameters in the substituted text.

An example of inline substitution is presented in the example below:

```
PROGRAM MAIN
   REAL A(100)
   CALL INPUT(A,N)
   DO I = 1,N
      CALL PROCESS(A,I)
   ENDDO
   CALL REPORT(A,N)
END
SUBROUTINE PROCESS(X,K)
   REAL X(*)
   X(K) = X(K) + K
   RETURN
END
```

If we inline subroutine `PROCESS`, substituting `A` and `I` for `X` and `K` respectively, we get the following code:

```
PROGRAM MAIN
   REAL A(100)
   CALL INPUT(A,N)
   DO I = 1,N
      A(I) = A(I) + I
   ENDDO
   CALL REPORT(A,N)
END
```

This code illustrates several well-known advantages of inline substitution:

1. procedure call overhead can be eliminated;
2. procedure body code can be tailored to the environment at the point of call, for example by using the index variable `I` from a register rather than memory; and
3. optimizations that would not be possible before substitution can be carried out—in this case, the loop can be vectorized.

The advantages of inlining are so compelling that many have suggested it as a generalized alternative to interprocedural analysis methods—if all the procedures in a program are inlined, ordinary single-processor analysis methods can be used to optimize the result.

However, overuse of inlining can cause a number of problems:

1. The massive substitution required can overwhelm the compilation system, since procedures after substitution might grow to unmanageable size, straining the capabilities of single-module compilers. In one

notable example, a program resulting from systematic inlining took 95 hours to compile [23].

2. The object code generated from an inlined program may run slower because optimizing compilers do not handle codes resulting from systematic inlining well [23].
3. Any code that is changed inside an inlined subroutine will force the recompilation of every procedure into which it has been substituted. In the limit, any change could require recompilation of the entire program.
4. Some subroutines are difficult to inline because of problems in substituting actual parameters for formals. This is illustrated by the example below:

```
PROGRAM MAIN
   REAL A(100, 100)
   ...
   CALL S(A(26,2),N)
   ...
END
SUBROUTINE S(X,M)
   REAL X(*)
   DO I = 1, M
      X(I) = X(I) + M
   ENDDO
   RETURN
END
```

The difficulty in this example is caused by the treatment of the two-dimensional actual parameter `A` as a single-dimensional array in subroutine `S`. We could introduce an equivalence between array A in the main program and a single-dimensional array, to produce the following code:

```
PROGRAM MAIN
   REAL A(100, 100), a$(10000)
   EQUIVALENCE (A(1,1),a$(1))
   ...
   DO I = 1, N
      a$(I+125) = a$(I+125) + N
   ENDDO
   ...
END
```

The main problem with this approach is that it loses the information about the independence of different rows and columns, which can be critical to dependence analysis. Furthermore, this fix would not be available to us if the call to `S` were in another procedure which was passed `A` as a parameter.

Instead of systematic inlining, we recommend a selective, goal-directed inlining that uses global program analysis to determine when inlining would be profitable [7].

10.3.2 Procedure Cloning

Often the main benefit of inlining is the ability to take advantage of some specific optimizations that are possible at some but not every call site for that procedure. Consider the following example:

```
PROCEDURE UPDATE(A,N,IS)
   REAL A(N)
   INTEGER N, IS
   DO I = 1,N*IS-IS,IS
      A(I) = A(I) + PI
   ENDDO
END
```

At first glance, this code looks vectorizable and it would be were it not for the possibility that the step size `IS=0`, in which case the computation would become a sum of `N*PI` into `A(1)`.

The obvious solution to this problem is to tailor the code to different specific versions based on the value of `A`. This could be done by a run-time test, but if we know the value of `IS` at compile time in every calling context, we can produce two versions of the program, one for the case `IS≠0` and another for the case `IS=0`. The compiler would be able to replace the call to `UPDATE` at the point of call with a call to one of the cloned procedures whenever it could determine the value at compile time.

Cloning is a particularly useful way to enhance the impact of constant propagation by treating parameters that are called with different constant values as constants in different clones of the original version of the procedure. This is the goal of cloning in the Convex Applications Compiler, the only commercial compiler we know that performs this optimization [55].

10.3.3 Hybrid Optimizations

There are cases where transformations involving more than one procedure can be used to gain some of the benefits of inlining without suffering the disadvantages. One such *hybrid optimization* is *loop embedding*, in which a loop is moved from one procedure to another [40].

Consider the original example for inlining in Section 10.3.1. If we interchanged the loop into subroutine PROCESS, we would get the following code inside that routine, which would vectorize well:

```
SUBROUTINE PROCESS(X,N)
   REAL X(*)
   DO K = 1,N
      X(K) = X(K) + K
   ENDDO
   RETURN
END
```

Note that the subroutine interface has changed to include the loop upper bound as a parameter instead of the loop index.

Interprocedural optimizations like this have been found useful in a number of cases, although there is as yet limited evidence as to their generality.

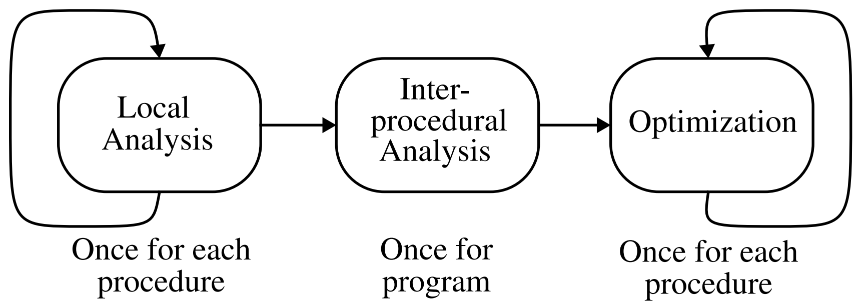
10.4 Managing Whole-Program Compilation

One of the problems presented by interprocedural compilation is the difficulty it causes for compilation management. In a conventional compilation system, the object code for any single procedure is a function only of the source code for that procedure. In an interprocedural compilation system, the object code for a procedure may depend on the source code for the entire program. This is a problem because it means that a change in source code could force recompilation of every procedure. Users will be unhappy if a large program needs to be completely recompiled after every small change.

We might expect that the interprocedural effects of changes made during the maintenance phase of a program, would be somewhat limited. Thus, global program analysis methods that can examine the effects of interprocedural information flow might be useful in reducing the amount of recompilation.

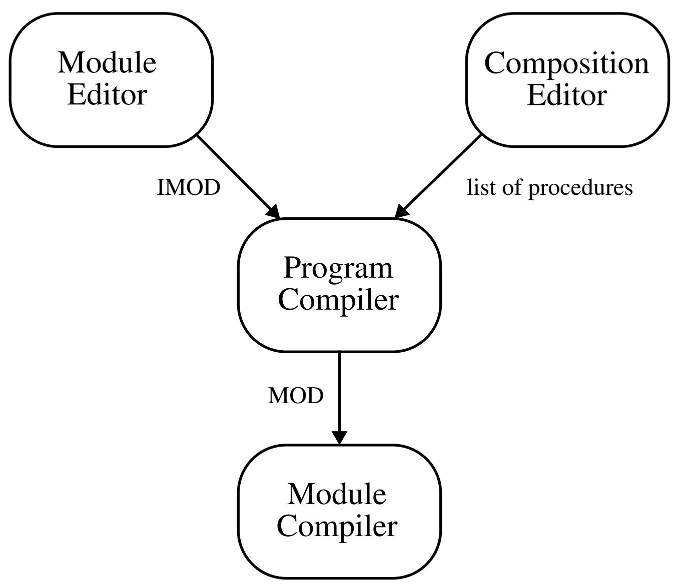
We begin by subdividing the procedural compilation into two distinct phases, one that depends on interprocedural information and one that does not. The first of these phases, which we shall call *local analysis*, includes many of the usual compilation tasks—lexical analysis, parsing, semantic analysis. At the same time, it examines the procedure for input to interprocedural analysis, determining the local sets used in each of the interprocedural analysis algorithms, such as the IMOD sets in MOD analysis. With this subdivision, the compilation process is structured as shown in Figure 10.19.

FIGURE 10.19 Interprocedural compilation process.



Although this organization permits interprocedural compilation, it still does not solve the recompilation problem. However, if intermediate representations are saved, the local analysis phase will not need to be reinvoked for any unchanged procedures.

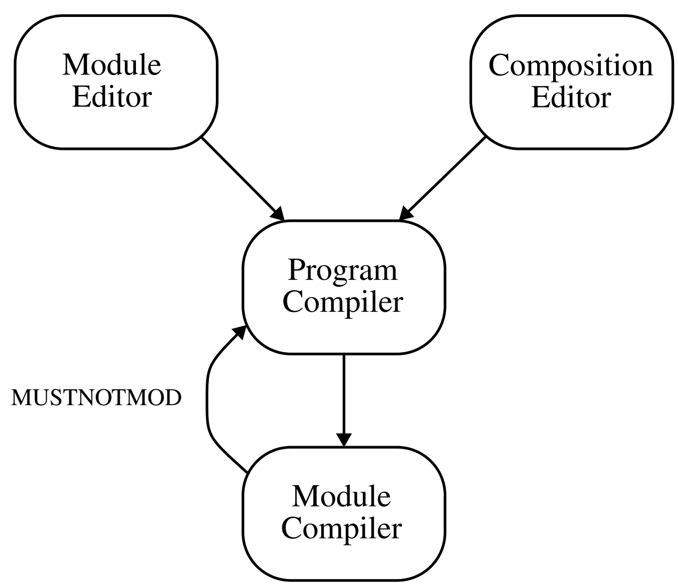
FIGURE 10.20 Interprocedural compilation system.



To address the recompilation problems more directly, we could organize our compilation system as shown in Figure 10.20. In this scheme, the job of lexical analysis is performed by a separate system component, which could reside in the module editor or in an importing tool. Information about each source procedure is stored in an intermediate representation that includes the parsed source. Programs are defined in this system by specification of a *program composition* via a *composition editor*, which can be viewed as a editor for lists of procedure names. The *program compiler* is the system component responsible for the compilation of the whole program. It reads all the local information for procedures in the program and carries out the various interprocedural analyses and optimizations required by the user. Once all interprocedural information is available, the *module compiler*, which is just a sophisticated optimization system, carries out the optimizing transformations on each procedure. Note that by separating the local analysis from the optimization, we have eliminated compilation-order dependencies between the various procedures in the program.

This system, by itself, does not solve the recompilation problem. It must be coupled with a feedback system from the individual module compilations that indicate which interprocedural analysis facts have been relied upon by the module compiler. To illustrate this process, consider the interprocedural recomputation of MOD sets, depicted in Figure 10.21.

FIGURE 10.21 Recompilation analysis.



We define the set MUSTNOTMOD(p,s) to contain all the variables that must not be modified as a side effect of the procedure call at site s within p if the code generated for procedure p is to remain correct. Thus if the source code of procedure q called at s is modified so that a variable in MUSTNOTMOD(p,s) is changed in q, then p must be recompiled. Think of MUSTNOTMOD(p,s) as a recording of the actions of the optimizing module compiler when it was last invoked on p. Whenever the optimizer used the fact that a variable was not modified at call site s to perform some optimization, it entered that variable into MUSTNOTMOD(p,s).

In focusing on MUSTNOTMOD, we are relying on the observation that an optimizer will make program transformations based upon information only when it is sure that an undesirable event that would invalidate the optimization is not possible. Thus optimizations are based upon the *absence* of variables from MOD(s) because that absence means that the variable cannot possibly be modified as a side effect of the call. Similarly, optimizations would be based on the absence of a variable from REF(s), so MUSTNOTREF(p,s) would be the corresponding recompilation set of interest. On the other hand, we will make optimizations like privatization based on the knowledge that a variable must be killed by a procedure call so MUSTKILL(p,s) could be used to hold recompilation information.

In the case of MOD, let us consider what must be done by the program compiler to ensure correctness after a change to some procedure in the program.

1. First, the program compiler must recompute the MOD sets for every call site in the program.
2. For each call site $s \in p$ in the program, if

   $$\text{MUSTNOTMOD}(p,s) \cap \text{MOD}(s) \neq \emptyset$$

   then recompile procedure p.

The computation of MUSTNOTMOD sets is dependent on the optimizations used in the module compiler, which may be burdensome to collect accurately. However, simple approximation methods often work well in practice. Here are two such approximations for MUSTNOTMOD:

1. MUSTNOTMOD(p,s) = $\neg$MOD(s) on the most recent compilation of p. Certainly, the module compiler cannot have depended on anything that was not true at the last compile.
2. MUSTNOTMOD(p,s) = $\neg$MOD(s) $\cap$ REF(p). This approximation goes one step farther than the previous one in that it does not force recompila-

tion as a result of changes in status to variables that are merely passed through *p*.

The second of these approximations was used by the designers of the $\mathbf{R}^n$ programming environment [17,11].

10.5 Some Implementations

The authors have been personally involved with several interprocedural compilation systems:

1. The vectorization and parallelization system PFC [3], developed at Rice University, used interprocedural constant propagation and array side effect analysis to improve its dependence analysis. The results of dependence analysis could be displayed in a special browser called PTOOL. PFC was also used as a dependence server by later systems, including ParaScope described below.
2. The $\mathbf{R}^n$ programming environment [17,28,30], also developed at Rice, was the first one designed to support interprocedural analysis in a practical compilation system. Recompilation analysis, discussed in Section 10.4 was first tried in the $\mathbf{R}^n$ system, which implemented constant propagation, ALIAS, MOD and REF.
3. ParaScope [19], a successor to $\mathbf{R}^n$, was designed to use interprocedural information in program parallelization. The FIAT interprocedural analysis framework [41] was originally developed for ParaScope, before being adapted into the SUIF compiler (see below). Although it computed solutions to several interprocedural problems, ParaScope initially used PFC as a server for interprocedural array section analysis.
4. The D System [1], which was initially based on ParaScope, is intended to support programming in High Performance Fortran. It includes a framework that extends FIAT to perform interactive recompilation of interprocedural information when the user is editing in the context of a given program. In addition to classical analyses and optimizations, the D System propagates array distribution information interprocedurally to support compilation of Fortran D and High Performance Fortran [19].
5. The Stanford SUIF compiler was constructed with the aid of the interprocedural framework FIAT, a tool that facilitates rapid prototyping of interprocedural systems [41]. It performs a complete set of interprocedural analyses including constant propagation, kill analysis for arrays and live analysis. In a recent experiment, programs from the NAS,

Spec-92 and Perfect benchmark suites were run through the compiler without any modification to the original source code. For this total of 27 programs, 16 of them yielded more parallel loops as a result of interprocedural optimization. For 4 of these, significant speedups have been obtained only as a result of interprocedural optimization [43].

6. The Convex Application Compiler [55], which was modeled after the $\mathbf{R}^n$ and ParaScope systems, is the first commercial system to perform interprocedural analysis on whole programs represented by multiple files. The Application Compiler does a complete job of interprocedural analysis and optimization for parallelization, computing the solutions to flow-sensitive interprocedural problems, cloning based on interprocedural constants, and employing array section analysis.

In addition to these, a number of commercial and research systems employ interprocedural information.

10.6 Historical Comments and References

The treatment in this chapter is based on the tutorial survey by Cooper, Hall, Kennedy and Torczon.

Interprocedural analysis was introduced in a number of works published in the 1970s. Allen showed how interprocedural data-flow analysis could be carried out on programs without recursion [2]. In an unpublished abstract, Allen and Schwartz later extended these techniques to programs with recursion. Spillman [67] described the interprocedural analysis that was available in the IBM PL/I compiler for procedures in a single file . A principle goal of this implementation was analysis of pointer targets.

Barth [6] was the first published paper to discuss may and must problems. Banning [5] introduced the notion of flow-sensitive and flow-insensitive problems and presented a polynomial-time algorithm for solving such problems. Myers [56] presented a general algorithm for flow-insensitive problems, which he proved to be Co-NP Complete in the presence of aliasing.

Algorithms for flow-sensitive analysis have been described by Myers [56], Sharir and Pnueli [66], Harrison [44], Landi and Ryder [53], Choi, Burke and Carini [31] and Hall, Murphy and Amarasinghe [42]. The algorithm for flow-insensitive interprocedural analysis presented in Section 10.2.3 is due to Cooper and Kennedy [24,25,27,26] as is the alias analysis algorithm in Section 10.2.4 [27]. The flow-sensitive constant propagation algorithm presented in Section 10.2.5, is based on the work of Callahan, Cooper, Kennedy and Torczon [15]. The flow-sensitive kill analysis algorithm is modeled after an algorithm proposed by Callahan [13], although this particular formulation is new.

Interprocedural symbolic analysis has been discussed by a number of researchers, including Haghighat and Polychronopoulos [37], Irigion, Jouvelot, and Triolet [50], and Hall, Murphy, and Amarasinghe [42]. The treatment in Section 10.2.7.1 follows Havlak [45].

Array section analysis, discussed in Section 10.2.7.2, has been the subject of work by a number of authors including Triolet, Irigoin and Feautrier [68], Burke and Cytron [9], Callahan and Kennedy [16,12], Li and Yew [54], Havlak and Kennedy [46], Irigoin, Jouvelot, and Triolet [50] and Hind, Burke, Carini, and Midkiff [48]. Several researchers have developed algorithms for flow-sensitive array analysis, including Irigion [51], IItsuka [49], Tu and Padua [69], and Hall, Murphy and Amarasinghe [42].

Call graph analysis has been studied by Walter [70], Weihl [73], Spillman [67], Burke [10] and Shivers[63,64,65].The call graph construction algorithm in Section 10.2.8 is based on an algorithm due to Ryder [62]. The proof that the same algorithm converges in the recursive case is due to Callahan, Carle, Hall and Kennedy [14]. An algorithm that is not precise but which generates its approximation in time linear in the size of the resulting call graph is given by Hall and Kennedy [39,38].

Inline substitution has been widely studied[34,59]. Cooper, Hall and Torczon noted its disadvantages [23]. Cloning was studied by Cooper, Hall and Kennedy [18,20,21]. Hybrid optimizations have been studied by Hall, Kennedy, and McKinley [40]. Techniques and algorithms closely related to cloning as described in this paper have been studied by Wegman [71] for intraprocedural analysis, Bulyonkov [8] and Ruf and Weise [61] for partial evaluation, Johnston [52] for dynamic compilation of APL, and Chambers and Ungar [33] for the language SELF.

The approach to program management described in Section 10.4 was pioneered in the $\mathbf{R}^n$ programming environment [17,26,27]. Recompilation analysis was developed by Burke, Cooper, Kennedy and Torczon [29,11].

The treatment in this chapter is based on the tutorial survey by Cooper, Hall, Kennedy and Torczon [22].

10.7 Summary

Although interprocedural analysis has limited applicability in optimizing code for uniprocessors, it is important for automatic parallelization systems. To be totally effective, a variety of analysis problems must be addressed. These include two types of problems:

1. forward propagation problems that determine the context in which a given procedure is called and
2. backward propagation problems which determine side effects of procedure calls.

In addition, interprocedural analysis problems can be classified as:

1. flow-insensitive problems, for which a precise solution does not require tracing control-flow through the procedures down the call chain, and
2. flow-sensitive problems, which require control-flow tracing for a precise solution.

It has been shown that flow-insensitive forward and backward problems can be solved in $O((N+E)V)$ time, where N is the number of procedures in the program, E is the number of call sites and V is the number of global variables and parameters in the program.

In the most general case, flow-sensitive problems have been shown to be intractable. However, for typical programming languages, good approximations can be achieved in time that is polynomial in the size of the call graph. In practice, these algorithms perform in near linear time.

Interprocedural problems for single variables can be extended in a natural way to handle analysis of value ranges and symbolic values. In addition they can also be extended to analyze side effects to regular sections of array variables. In both cases, the running time will be expanded by a factor proportional to the maximum depth of the lattice of approximations used.

To be usable, an interprocedural analysis system needs to minimize the number of procedures that must be recompiled as a result of local program changes. Users will not tolerate a recompilation system that requires recompiling an entire program after a simple change to one procedure. The functions required to do this may be best embedded in a more general program management system.

10.8 References

[1] V. Adve, A. Carle, E. Granston, S. Hiranandani, C. Koelbel, J. Mellor-Crummey, C. Tseng and S.K. Warren. The D System: support for data-parallel programming. Technical Report CRPC-TR94-378, Rice University, Center for Research on Parallel Computation, January 1994.

[2] F. E. Allen. Interprocedural data flow analysis. In *Proceedings of the IFIP Congress 1974*, pages 398-402, Amsterdam, 1974. North Holland.

[3] J. R. Allen and K. Kennedy. PFC: a program to convert Fortran to parallel form. In *Supercomputers: Design and Applications*, K. Hwang, editor, pages 186–203. IEEE Computer Society Press, August 1984.

[4] B. Alpern and M. N. Wegman and F. K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages.* ACM, 1988.

[5] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the Sixth Annual Symposium on Principles of Programming Languages*. ACM, January 1979.

[6] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724-736, September 1978.

[7] P. Briggs, K. D. Cooper, M. Hall and L. Torczon. Goal-directed interprocedural optimization, Technical Report TR90-148, Department of Computer Science, Rice University, November 1990.

[8] M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473-484, 1984.

[9] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*. ACM, June 1986.

[10] M. Burke. An interval-based approach to exhaustive and incremental interprocedural analysis. Research Report RC 12702, IBM Yorktown Heights, September 1987.

[11] M. Burke and L. Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems* 15(3):367–399, July 1993.

[12] D. Callahan, A Global Approach to the Detection of Parallelism, Ph.D. thesis, Rice University, Department of Computer Science, Houston, TX, March 1987.

[13] 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,* SIGPLAN Notices 23(7):47-56. ACM, July 1988.

[14] D. Callahan, A. Carle, M.W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, SE-16(4):483-487, April 1990.

[15] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, SIGPLAN Notices 21(7):152-161. ACM, July 1986.

[16] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.

[17] A. Carle, K.D. Cooper, R.T. Hood, K. Kennedy, L. Torczon and S. K. Warren. A practical environment for Fortran programming. *IEEE Computer* 20(11): 75–89, November 1987.

[18] K. D. Cooper. *Interprocedural Data Flow Analysis in a Programming Environment*, Ph.D. Thesis, Rice University, Computer Science Department, Houston, TX, April, 1983.

[19] K. D. Cooper, M. Hall, R. T. Hood, K. Kennedy, K. McKinley, J. Mellor-Crummey, L. Torczon and S.K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE* 81(2): 244–263, February 1993.

[20] K. D. Cooper, M. W. Hall and K. Kennedy. Procedure cloning. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 96–105. IEEE, April 1992.

[21] K. D. Cooper, M. W. Hall and K. Kennedy. A methodology for procedure cloning. *Computer Languages* 19(2), 1993.

[22] K. D. Cooper, M. W. Hall, K. Kennedy and L. Torczon. Interprocedural analysis and optimization. To appear in The Communications in Pure and Applied Mathematics.

[23] K. D. Cooper, M. W. Hall and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience* 21(6):581–601, June 1991.

[24] K. D. Cooper and K. Kennedy. Efficient computation of flow-insensitive interprocedural summary information. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, SIGPLAN Notices 19(6):247-258. ACM, June 1984.

[25] K. D. Cooper and K. Kennedy. Efficient computation of flow-insensitive interprocedural summary information—a correction. *SIGPLAN Notices* 23(4):35-42. ACM, April 1988.

[26] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time.In *Proceedings of the SIGPLAN ‘88 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 23(7):57-66. ACM, July 1988.

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

[28] K. Cooper, K. Kennedy and L. Torczon. The impact of interprocedural analysis and optimization on the design of a software development environment. In Proceedings of the SIGPLAN 85 Symposium on Compiler Construction. ACM, June 1985.

[29] K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: eliminating unnecessary recompilation. In *Proceedings of the SIGPLAN `86 Symposium on Compiler Construction*, SIGPLAN Notices 21(7):58-67, July 1986.

[30] K. Cooper, K. Kennedy and L. Torczon. The impact of interprocedural analysis and optimization in the R^n environment. *ACM Transactions on Programming Languages and Systems* 8(4):491–523, October 1986.

[31] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual Symposium on Principles of Programming Languages*. ACM, January 1993.

[32] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual Symposium on Principles of Programming Languages*, June 1989.

[33] C. Chambers and D. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 24(7): 146-160. ACM, July 1989.

[34] J. Davidson and A. Hollar. A study of a C function inliner. *Software—Practice and Experience* 18(8):775–790, August 1988.

[35] S. L. Graham and M. Wegman. A fast and usually linear algorithm for global data flow analysis. *Journal of the ACM* 23(1):172–202, 1976.

[36] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementations. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*. ACM, June 1993.

[37] M. Haghighat and C. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization, and scheduling of programs. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[38] M. W. Hall. *Managing Interprocedural Optimization*. Ph.D. thesis, Rice University, Department of Computer Science, Houston, TX, April 1991.

[39] M. W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems* 1(3), September 1992.

[40] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[41] M. W. Hall, J. Mellor-Crummey, A. Carle and R. Rodriguez. FIAT: a framework for interprocedural analysis and transformation," In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August, 1993.

[42] M. W. Hall, B. R. Murphy, and S. P. Amarasinghe. Interprocedural analysis for parallelization: A case study. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[43] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, M. S. Lam. Interprocedural analysis for parallelization: preliminary results. Stanford Computer Systems Laboratory Technical Report, CSL-TR-95-665, March, 1995.

[44] W. L. Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179-396, October 1989.

[45] P. Havlak. *Interprocedural symbolic analysis*. Ph.D. thesis, Rice University, Dept. of Computer Science, May 1994.

[46] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350 360, July 1991.

[47] M. Hecht and J. D. Ullman. A simple algorithm for global data flow analysis of programs. SIAM Journal of Computing 4: 519–532.

[48] M. Hind, M. Burke, P. Carini, and S. Midkiff. An empirical study of precise interprocedural array analysis. *Scientific Programming*, 3(3):255-271, 1994.

[49] T. IItsuka. Flow-sensitive interprocedural analysis method for parallelization. In *IFIP TC10/WG10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, January 1993.

[50] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[51] F. Irigoin. Interprocedural analyses for programming environments. In *NSF-CNRS Workshop on Environments and Tools for Parallel Scientific Programming*, September 1992.

[52] R. L. Johnston. The dynamic incremental compiler of APL\3000. In *Proceedings of the APL '79 Conference*, pages 82-87. ACM, June 1979.

[53] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 27(7):235-248, July 1992.

[54] Z. Li and P. Yew. Efficient interprocedural analysis for program restructuring for parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, New Haven, CT, July 1988.

[55] R. Metzger and S. Stroud. Interprocedural constant propagation: an empirical study. *ACM Letters on Programming Languages and Systems*, 1(3), December 1992.

[56] E. Myers. A precise inter-procedural data flow algorithm. In Conference Record of the *Eighth Annual Symposium on Principles of Programming Languages*. ACM, January 1981.

[57] J. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM* 23(1):159–171, January 1976.

[58] J. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica* 7(3): 305–318, 1977.

[59] S. Richardson and M. Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters* 321(3), August 1989

[60] B. K. Rosen and M. Wegman and F. K. Zadeck. Global value numbers and redundant computations.In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming,* ACM, 1988.

[61] E. Ruf and D. Weise. Using types to avoid redundant specialization. In *Proceedings of the PEPM '91 Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices 26(9), pages 321-333. ACM, September 1991.

[62] B. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, SE-5(3):216-225, 1979.

[63] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, SIGPLAN Notices 23(7):164-174. ACM, July 1988.

[64] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, May 1991.

[65] O. Shivers. The semantics of scheme control flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, SIG PLAN Notices 26(9), pages 190-198, September 1991.

[66] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981.

[67] T. C. Spillman. Exposing side-effects in a PL/I optimizing compiler. In *Proceedings of the IFIP Congress 1971*, pages 376-381. North Holland, 1971.

[68] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, SIGPLAN Notices 21(7), pages 176-185. ACM, July 1986.

[69] P. Tu and D. Padua. Automatic array privatization. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[70] K. Walter. Recursion analysis for compiler optimization. *Communications of the ACM*, 19(9):514-516, 1976.

[71] M. Wegman. *General and Efficient Methods for Global Code Improvement*. Ph.D. thesis, University of California, Berkeley, CA, December 1981.

[72] M. Wegman and K. Zadeck.Constant propagation with conditional branches.Technical Report CS-89-36,Dept. of Computer Science, Brown University, May 1989.

[73] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of the Seventh Symposium on Principles of Programming Languages*. ACM, January 1980.