Part II

Interprocedural Data Flow Analysis

7

Introduction to Interprocedural Data Flow Analysis

The intraprocedural optimizations that we have discussed so far have ignored the effect of a call under the assumption that a safe approximation of the effect of a call can be incorporated without inspecting the called procedures. This was illustrated in Section 1.1.2. A possible improvement of using interprocedural data flow information by analyzing the called procedures was also demonstrated in the same section. In this chapter we evolve the basic concepts of the latter.

7.1 A Motivating Example

We use the program in Figure 7.1 as a running example in this chapter. We perform constant propagation and dead code elimination over this program and introduce common variants of interprocedural analyses. Figure 7.1(a) shows our program. From the viewpoint of interprocedural analysis, its simplifying features are that it is non-recursive and contains global variables only.

The optimized program after performing interprocedural constant propagation is shown in part (b). Modified statements are shown in gray background. Constant propagation replaces uses of variables by their known values and potentially creates dead code. The statements shown in gray background in part (c) are the assignments that become dead code and can be deleted. Observe that when procedure p is called from procedure q, the value of variable d is 14. However, p is also called from main and the value of d in that call is not known. Hence we cannot conclude that d is constant in procedure p. Also observe that when procedure p is called the second time, since the values of b and d are known to be 2 and 14 respectively, the condition on line 17 is true and the assignment on line 18 is executed. Since a is assigned 1 in procedure q, the value of c becomes 3 and remains 3 in expression $a+c$ on line 12. Our analysis does not perform conditional constant propagation and fails to discover that the value of c is 3. However, it discovers the value of a in expression $a+c$ on line 12 to be 2 due to the assignment in line 25.

```
 0. int a,b,c,d;
 1.
 2. void main()
 3. {  a = 5;
 4.    b = 3;
 5.    c = 7;
 6.    read(d);
 7.    p();
 8.    a = a+2;
 9.    print(c+d);
10.    d = a*b;
11.    q();
12.    print(a+c);
13. }
14.
15. void p()
16. {  b = 2;
17.    if (b<d)
18.       c = a+b;
19.    print(c+d);
20. }
21.
22. void q()
23. {  a = 1;
24.    p();
25.    a = a*b;
26. }
```

(a) Original program

```
 0. int a,b,c,d;
 1.
 2. void main()
 3. {  a = 5;
 4.    b = 3;
 5.    c = 7;
 6.    read(d);
 7.    p();
 8.    a = 7;
 9.    print(7+d);
10.    d = 14;
11.    q();
12.    print(2+c);
13. }
14.
15. void p()
16. {  b = 2;
17.    if (2<d)
18.       c = a+2;
19.    print(c+d);
20. }
21.
22. void q()
23. {  a = 1;
24.    p();
25.    a = 2;
26. }
```

(b) Discovered constants

```
 0. int a,b,c,d;
 1.
 2. void main()
 3. {  a = 5;
 4.    b = 3;
 5.    c = 7;
 6.    read(d);
 7.    p();
 8.    a = 7;
 9.    print(7+d);
10.    d = 14;
11.    q();
12.    print(2+c);
13. }
14.
15. void p()
16. {  b = 2;
17.    if (2<d)
18.       c = a+2;
19.    print(c+d);
20. }
21.
22. void q()
23. {  a = 1;
24.    p();
25.    a = 2;
26. }
```

(c) Discovered dead code

FIGURE 7.1

An example program with interprocedural constant propagation and subsequent interprocedural dead code elimination. For simplicity, we assume built-in operations to read and print data.

7.2 Program Representations for Interprocedural Analysis

Figure 7.2 shows two intermediate representations of our example. A *call multigraph* is a directed graph which captures the caller-callee relationships in a program. Nodes in a call multigraph represent procedures whereas edges represent procedure calls and are labeled by the call sites. Since each call to a procedure is represented by a distinct edge, a call multigraph contains parallel edges when a procedure contains multiple calls to some procedure. Recursion in a program would cause cycles in the call multigraph. The call multigraph for our program does not contain parallel edges or cycles.

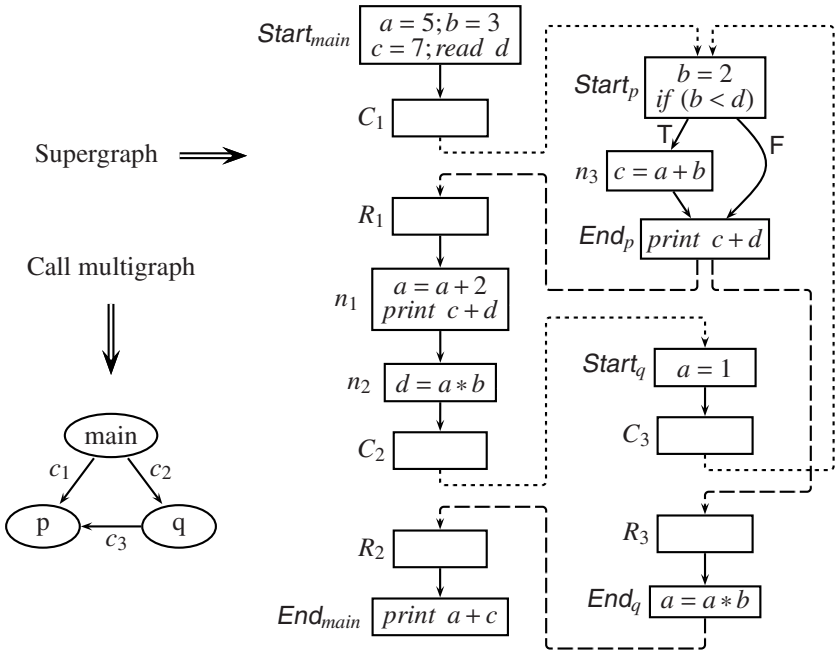


FIGURE 7.2
Common intermediate representations for interprocedural data flow analysis.

The second intermediate representation is also a directed graph called a *supergraph* which connects CFGs of callers and callees by edges indicating interprocedural control transfers. A simpler version of supergraph was introduced in Chapter 1. As illustrated in Figure 1.3 on page 5, it represented a call by a single basic block. Now we split a call site c_i into a *call node* C_i and the corresponding *return node* R_i. A call to procedure r at call site c_i is represented by an edge from C_i to $Start_r$. The corresponding return from procedure r is represented by an edge from End_r to R_i. These edges are interprocedural edges. The edges in the individual CFG are intraprocedural edges. The supergraph in Figure 7.2 shows the interprocedural edges by dashed lines and intraprocedural edges by solid lines. The program entry and exit is denoted by $Start_{main}$ and End_{main}.

A supergraph and the corresponding call multigraph are related to each other by a simple graph transformation. If every procedure in a supergraph of a program is represented by a single node by combining all blocks of a procedure and all return edges are removed, a supergraph reduces to the call multigraph of the program.

Observe that blocks $Start_{main}$, n_1, and $Start_p$ in our supergraph contain multiple statements in spite of the fact that for constant propagation a basic block consists of a single statement. However, it is possible to combine assignment statements into a single block when they do not have data dependence between them; we have done so for convenience.

7.3 Modeling Interprocedural Data Flow Analysis

In this section, we develop an abstract view of interprocedural data flow analysis with the goal of evolving basic concepts; details are postponed to subsequent chapters.

7.3.1 Summary Flow Functions

A simple view of interprocedural analysis is to model a procedure call as a basic block and represent the effect of the called procedure by a *summary* flow function. Since it needs to represent the effect of all calls to the procedure it represents, a summary flow function must be *context independent* and must be parametrized so that the data flow information from the calling context can be incorporated.

A summary flow function $f_r : L \mapsto L$ for procedure r can be modeled in the usual manner in terms of Gen and $Kill$ components as shown below:

$$\begin{aligned} f_r(\mathsf{x}) &= (\mathsf{x} - Kill_r(\mathsf{x})) \cup Gen_r(\mathsf{x}) \\ &= (\mathsf{x} - (ConstKill_r \cup DepKill_r(\mathsf{x}))) \cup (ConstGen_r \cup DepGen_r(\mathsf{x})) \end{aligned}$$

Note that this merely models the function f_r; whether f_r is actually constructed by identifying $ConstKill$, $DepKill$, $ConstGen$, and $DepGen$ is an independent matter and is discussed in Section 7.3.3. Chapter 8 discusses how it is constructed; we introduce some intuitions related to it in Section 7.6.

Although the notions of Gen_r and $Kill_r$ for a procedure r are similar to the notions of Gen_i and $Kill_i$ for a basic block i, there are some differences arising from the fact that the execution of a procedure may involve control transfers whereas a basic block involves a strictly sequential execution. Thus we need to distinguish between *may* and *must* properties. For example, when performing liveness analysis, $Kill_r$ must ensure that a variable is modified along all paths in r. This is represented by $MustKill_r$ which is different from $MayKill_r$; the latter says that a variable is modified along some path but not necessarily all. For available expressions analysis, $Kill_r$ should be $MayKill_r$ rather than $MustKill_r$.

We now describe the summary flow functions for constant propagation and liveness analysis of our example program. Consider the instance of constant propagation framework involving our example program. Let $\mathsf{x} \in L$ be the tuple $\langle \widehat{\mathsf{x}}_a, \widehat{\mathsf{x}}_b, \widehat{\mathsf{x}}_c, \widehat{\mathsf{x}}_d \rangle$ representing the constantness information of the four variables in our example program. Thus, $\widehat{\mathsf{x}}_a$, $\widehat{\mathsf{x}}_b$, $\widehat{\mathsf{x}}_c$, and $\widehat{\mathsf{x}}_d$ are values in the component lattice $\widehat{L}$ for constant propagation (Figure 4.5 on page 110).

From the supergraph in Figure 7.2, it is clear that the data flow values of a and d remain unaffected by procedure p since it does not modify them. Further, variable b is always 2 at the end of procedure p regardless of the flow of execution. The data flow value of variable c depends on result of the condition in block $Start_p$. If the execution follows edge $Start_p \rightarrow n_3$, the data flow value of c becomes $\widehat{\mathsf{x}}_a + 2$. The alternative execution path involving edge $Start_p \rightarrow End_p$ does not modify c. Static

summarization of the two possibilities results in $\widehat{x}_c \sqcap (\widehat{x}_a + 2)$. Thus, the flow function that summarizes the effect of procedures p is:

$$f_p(\langle \widehat{x}_a, \widehat{x}_b, \widehat{x}_c, \widehat{x}_d \rangle) = \langle \widehat{x}_a, 2, \widehat{x}_c \sqcap (\widehat{x}_a + 2), \widehat{x}_d \rangle$$

To see the flow function in terms of *Gen* and *Kill*, observe that the data flow information $\mathsf{x} = \langle \widehat{x}_a, \widehat{x}_b, \widehat{x}_c, \widehat{x}_d \rangle$ is merely a convenient notation for the set representation $\mathsf{x} = \{\langle a, \widehat{x}_a \rangle, \langle b, \widehat{x}_b \rangle, \langle c, \widehat{x}_c \rangle, \langle d, \widehat{x}_d \rangle\}$. Thus, the *Gen* and *Kill* components of f_p are:

$$\begin{aligned} \mathit{ConstGen}_p &= \{\langle b, 2 \rangle\} \\ \mathit{ConstKill}_p &= \emptyset \\ \mathit{DepGen}_p(\mathsf{x}) &= \{\langle c, \widehat{x}_c \sqcap (\widehat{x}_a + 2) \rangle\} \\ \mathit{DepKill}_p(\mathsf{x}) &= \{\langle b, \widehat{x}_b \rangle, \langle c, \widehat{x}_c \rangle\} \end{aligned}$$

Since procedure q calls procedure p, the definition of f_q depends on the definition of f_p. In particular, procedure q assigns 1 to a and then passes on the resulting data flow information $\langle 1, \widehat{x}_b, \widehat{x}_c, \widehat{x}_d \rangle$ to f_p. The resulting intermediate flow function defines the data flow at R_3 in terms of the assumed input value $\mathsf{x} = \langle \widehat{x}_a, \widehat{x}_b, \widehat{x}_c, \widehat{x}_d \rangle$ available at Start_q. When the flow function of block End_q is composed with it, we get

$$f_q(\langle \widehat{x}_a, \widehat{x}_b, \widehat{x}_c, \widehat{x}_d \rangle) = \langle 2, 2, \widehat{x}_c \sqcap 3, \widehat{x}_d \rangle$$

For live variables analysis, $\mathbb{V}\mathsf{ar} = \{a, b, c, d\}$ and L is $2^{\mathbb{V}\mathsf{ar}}$. We leave it for the reader to verify that the flow functions for procedures p and q are:

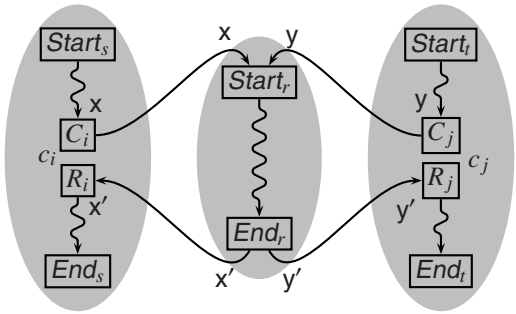
$$\begin{aligned} f_p(\mathsf{x}) &= (\mathsf{x} - \{b\}) \cup \{a, c, d\} \\ f_q(\mathsf{x}) &= (\mathsf{x} - \{a, b\}) \cup \{c, d\} \end{aligned}$$

where $\mathsf{x} \subseteq \{a, b, c, d\}$.

7.3.2 Inherited and Synthesized Data Flow Information

For a given call to procedure r in the body of procedure s, let x be the data flow information reaching the call point. Then, x represents the data flow information *inherited* by procedure r from the call site in s and $f_r(\mathsf{x})$ represents the data flow information *synthesized* by r at the call site in s. This is illustrated in Figure 7.3. The inherited data flow information is context sensitive. The synthesized data flow information has a context insensitive component represented by $\mathit{ConstGen}_p$ and a context sensitive component represented by $\mathit{DepGen}_p(\mathsf{x})$ and $\mathsf{x} - \big(\mathit{ConstKill}_p \cup \mathit{DepKill}_p(\mathsf{x})\big)$. The final data flow information at a program point u in procedure r is influenced by

- interprocedural data flow information inherited by r from all calls to r,
- interprocedural data flow information synthesized by calls appearing on the paths from Start_r to u for forward flows and from u to End_r for backward flows, and



Data Flow Information	
x	Inherited by r from call site c_i in s
y	Inherited by r from call site c_j in t
x′	Synthesized by r in s at call site c_i
y′	Synthesized by r in t at call site c_j

FIGURE 7.3
Inherited and synthesized data flow information.

- intraprocedural data flow information along the paths from $Start_r$ to u for forward flows and from u to End_r for backward flows.

In Part I of the book, the interprocedural data flow information was approximated as follows: The inherited data flow information was approximated by a conservative value of *BI* and the synthesized data flow information was approximated by using fixed conservative values for *Gen*(x) and *Kill*(x). These approximations were independent of calls and were same for all calls to all procedures in the program. Interprocedural data flow analysis tries to replace the above approximations by more precise values.

For constant propagation in our example, procedure p has a call from *main* and a call from q. The context insensitive synthesized data flow information of p is $\{\langle b, 2\rangle\}$. It inherits $\mathsf{x} = \langle 5, 3, 7, \perp\rangle$ from its call in *main*. Since we wish to separate the data flow information associated with different variables, we view x as $\{\langle a, 5\rangle, \langle b, 3\rangle, \langle c, 7\rangle, \langle d, \perp\rangle\}$. The context sensitive synthesized data flow information for this call is $\{\langle a, 5\rangle, \langle c, 7\rangle, \langle d, \perp\rangle\}$. This is the data flow information associated with block R_1 in the caller procedure *main*. The data flow information inherited by p from its call in q is $\langle 1, 2, 7, 14\rangle$. The corresponding context sensitive synthesized data flow information associated with block R_3 in the caller procedure q is $\{\langle a, 1\rangle, \langle c, \perp\rangle, \langle d, 14\rangle\}$.

7.3.3 Approaches to Interprocedural Data Flow Analysis

Various methods of interprocedural data flow analysis can be divided into two broad categories: *functional* approach or a *value-based* approach.

A functional approach to interprocedural analysis consists of two steps: In the first step, the summary flow functions that represent the effects of a call are computed. These functions are context independent and are parametrized. In the second step, inherited data flow information of a procedure is computed from its calling contexts. Then, the body of the procedure is analyzed and the summary flow functions corresponding to the callee are used to compute the synthesized data flow information.

Observe that using the summary functions does not require traversing the body of the caller procedures represented by the functions. In practice, computation of summary flow functions is possible only for a limited class of frameworks. In particular, it is easy for separable frameworks. In non-separable frameworks, it may not be possible to automatically construct summary flow functions unless the lattice is finite and flow functions are distributive. This is because constructing summary flow functions requires reducing expressions involving function compositions and intersections. Whether a systematic method of reductions can be devised or not depends on the nature of the flow functions and data flow values.

A *value-based* approach avoids computing summary flow functions. Instead, it directly computes data flow values by traversing a program during analysis. In particular, when it encounters a procedure call, the inherited data flow information is propagated to the callee and the method starts examining the callee's body. At the end of the analysis of the callee's body, synthesized data flow information is propagated back to the caller and the analysis of caller's body is resumed. This approach requires traversing a procedure repeatedly for different calling contexts. Conceptually, this approach is simpler than functional approach except that it may have to distinguish between a large number of contexts.

Both these approaches inherently handle recursion so long as the frameworks involve finite lattices. Although our example program in this chapter is non-recursive, subsequent chapters present these approaches for recursive programs.

7.4 Compromising Precision for Scalability

Recall that the scope of intraprocedural data flow analysis is restricted to individual procedures. By contrast, interprocedural data flow analysis needs to examine entire programs. Although this increases the precision of data flow information, practically interprocedural analysis could be very inefficient both in terms of space as well as time. Since real life applications often contain hundreds or thousands of procedures, a supergraph is many times larger than a single CFG. Hence efficiency and scalability issues assume much more significance in interprocedural data flow analysis than in intraprocedural data flow analysis. Most approaches that achieve efficiency and scalability, compromise on precision in one way or the other. Two common tradeoffs that enhance efficiency and scalability are:

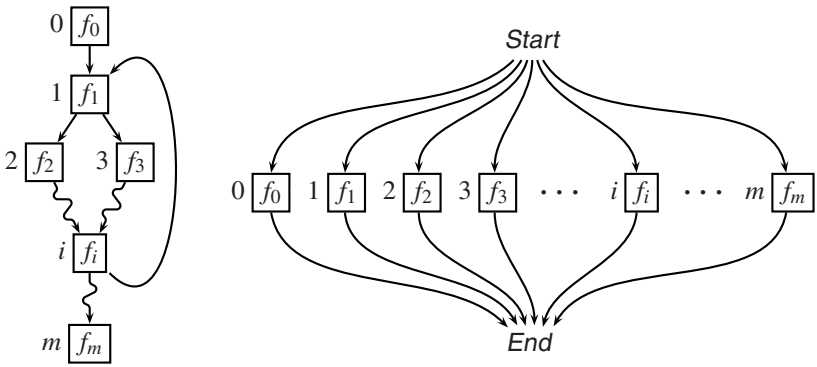
- Not distinguishing between actual and spurious control flow paths.

  This manifests itself in the form of flow or context insensitivity.

- Restricting the influences between caller and callees.

  This results in side effects analysis instead of whole program analysis.

In this section we explore these tradeoffs and explain how they affect the precision of interprocedural data flow analysis. Empirical investigations have revealed that



(a) CFG (b) Flow insensitive analysis

FIGURE 7.4
Modeling flow insensitive analysis.

these tradeoffs enhance the efficiency of analysis significantly. The resulting loss of precision has been found to be tolerable in many cases but not all.

7.4.1 Flow and Context Insensitivity

Recall that the *MOP* value associated with a program point u is the *glb* of data flow information computed along all paths reaching u (Definition 3.20). Let $P(u)$ denote the set of paths used for computing data flow information at u. If $P(u) \supseteq \mathit{paths}(u)$, then a data flow value computed along all paths in $P(u)$ is weaker than MOP_u and hence is safe. Precision of the data flow value computed by traversing paths in $P(u)$ depends on how close $P(u)$ is to $\mathit{paths}(u)$. The larger the number of spurious paths in $P(u)$, the more imprecise the computed data flow value is likely to be.

As observed in Section 3.4.3, computing the *MOP* assignment for arbitrary monotone frameworks is undecidable. Thus the algorithms that need to cover all potential paths can at best compute the *MFP* solution (Section 3.4.2). This involves merging data flow information at shared program points in $\mathit{paths}(u)$. If the flow functions are non-distributive, this has the effect of creating combinations of data flow values across paths (Example 4.6). This can be seen as traversing some paths that are not present in $\mathit{paths}(u)$. This source of imprecision shows the limit of static analysis and hence is accepted as inevitable.

We now describe two features called *flow* and *context insensitivity* that a method can employ as a matter of choice for achieving efficiency. They are orthogonal but are similar in the sense that both of them relate to spurious paths; they are different in the nature of paths they consider.

Flow insensitivity

As mentioned in Section 1.2, flow insensitive analysis disregards the flow of control by implicitly assuming that the block can be executed in all possible orders. This is achieved by accumulating the effect of each block in the same data flow value and the resulting value is a safe approximation of data flow information at each point.

For convenience, let the blocks in a procedure be numbered from 0 to m in any arbitrary order. Then, flow insensitive analysis computes $\mathsf{x} \in L$ as defined below:

$$\mathsf{x} = \prod_{i=0}^{m} f_i(BI) \tag{7.1}$$

where $BI \in L$ is the boundary information. This is illustrated in Figure 7.4.

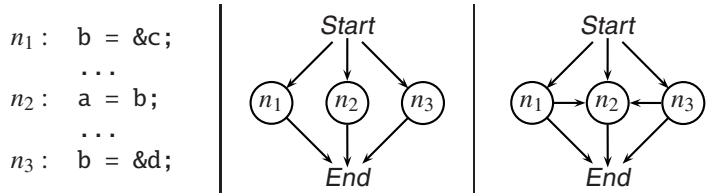
Intuitively, the operation of function composition employed in the usual flow sensitive data flow analysis is replaced by the operation of function confluence; the latter is commutative while the former is not. Thus just a single visit to each block in any arbitrary order approximates all possible orders between blocks. Section 8.1.2 shows that the value x computed by Equation (7.1) is a safe approximation of the corresponding flow sensitive data flow information at each program point.

In the case of flow functions with dependent parts, the above model of flow insensitive computation needs to be modified slightly. This is because the dependent component of f_i could depend on a value computed by some f_j and since the statements are assumed to be executed in an arbitrary order, this dependence must be taken care of. For example, consider flow insensitive *may* points-to analysis for the pointer assignments in Figure 7.5(a). Block n_3 generates a points-to pair $b \rightarrow d$ and since we assume that n_2 could be potentially executed after n_1 or n_3, our analysis should discover the points-to pairs $a \rightarrow c$ and $a \rightarrow d$. Following the strategy of Figure 7.4(b) we would get the flow graph in Figure 7.5(b) and it will not compute the desired points-to pairs. A simple way of modeling flow insensitive analysis in such a situation is to extend the graph by adding edges from n_1 and n_3 to n_2 as shown in Figure 7.5(c). Observe that these are data flow dependences captured by the primitive entity functions and the composite entity functions described in Section 4.5. They are different from the dependences of values of variables at run time which may or may not create dependences of data flow values.

In practice, instead of creating such flow graphs, the required dependences are remembered in a global data structure. Points-to analysis constructs a graph that contains points-to edges as well as constraints that result in points-to edges. Thus, edge $b \rightarrow c$ is added while processing n_1. When n_2 is processed, edge $a \rightarrow *b$ is also remembered apart from adding the edge $a \rightarrow c$. Whenever new points-to information for b becomes available, an appropriate points-to edge is added to the graph.

We now introduce the issues that arise when we wish to construct a flow insensitive summary flow function instead of computing a flow insensitive data flow value. These issues are handled in details in Chapter 8. We consider the following two cases in constructing flow insensitive summary functions:

- *When the flow functions do not have dependent parts.*



(a) Pointer assignment (b) Default modeling (c) Required modeling

FIGURE 7.5
Modeling flow insensitive analysis in presence of dependent parts in flow functions. Edges $n_1 \rightarrow n_2$ and $n_3 \rightarrow n_2$ represent the fact that $\mathit{DepGen}_{n_2}(\mathsf{x})$ depends on the data flow information computed at n_1 and n_3.

If flow functions have only constant parts (as in liveness analysis), then the summary flow function can be constructed by computing the values of the constant parts. In particular, for constructing side effect function of procedure r for liveness analysis, we need to compute only $\mathit{ConstKill}_r$ and $\mathit{ConstGen}_r$ sets.

The *ConstKill* set for live variables should include only those variables that are guaranteed to be modified within the called procedure regardless of the order of execution of basic blocks. This is represented by flow insensitive *MustKill* set which is computed using Equation (7.1) by intersecting the *Kill* sets of the individual basic blocks in the procedure. This should be contrasted with *ConstGen* computation which must record every variable that becomes live locally within the called procedure regardless of the control flow. This is represented by flow insensitive *MayUse* set which is computed using Equation (7.1) by taking a union of the *Gen* sets of individual basic blocks in the procedure. Then,

$$f_r(\mathsf{x}) = (\mathsf{x} - \mathit{MustKill}_r) \cup \mathit{MayUse}_r$$

where $\mathsf{x} \in L$. Both these approaches are demonstrated for our example program in Section 7.6 although their detailed formal definitions are provided later in Chapter 8.

- *When the flow functions have dependent parts.*

  In this case, merely combining the *Gen* and *Kill* sets does not work. Instead, we will have to replace $\mathit{BI} \in L$ by a symbolic value that represents the data flow value in the calling context and parametrizes the summary flow function. For simplicity, we describe this for liveness analysis. A symbolic value for liveness analysis could be the following set:

$$\mathit{BI} = \left\{ \langle a, \widehat{\mathsf{x}}_a \rangle \mid a \in \mathbb{G}\mathsf{var} \right\}$$

where $\widehat{\mathsf{x}}_a$ is a symbolic value that will be replaced by a concrete value *true* or *false* from the calling context. The flow function f_i to be used in Equation (7.1) will also have to be re-written as:

$$\begin{aligned} f_i(\mathsf{x}) &= (\mathsf{x} - \mathit{Remove}_i) \cup \mathit{Add}_i \\ \mathit{Remove}_i &= \{\langle a, \widehat{\mathsf{x}}_a\rangle \mid a \in \mathit{Kill}_i, a \notin \mathit{Gen}_i\} \\ \mathit{Add}_i &= \{\langle a, \mathit{true}\rangle \mid a \in \mathit{Gen}_i\} \cup \\ &\quad \{\langle a, \mathit{false}\rangle \mid a \in \mathit{Kill}_i, a \notin \mathit{Gen}_i\} \end{aligned}$$

$\langle a, \mathit{true}\rangle \in \mathsf{x}$ indicates that variable a is live and $\langle a, \mathit{false}\rangle \in \mathsf{x}$ indicates that variable a is not live; exactly one of them is in x by construction. The confluence operation over the sets of pairs is defined as follows for liveness analysis:

$$\mathsf{x} \cup \mathsf{y} \;=\; \{\langle a, \widehat{\mathsf{x}}_a + \widehat{\mathsf{y}}_a\rangle \mid \langle a, \widehat{\mathsf{x}}_a\rangle \in \mathsf{x}, \langle a, \widehat{\mathsf{y}}_a\rangle \in \mathsf{y}\}$$

where + denote the boolean OR operation.

In the presence of dependent parts in flow functions, it may not always be possible to construct summary flow functions. In Chapter 8 we characterize the class of frameworks for which summary flow functions can be directly constructed. For others, either it is not possible to construct summary flow functions or some adhoc mechanism may have to be employed. For example, it is not possible to construct flow sensitive summary flow functions for points-to analysis. However, flow insensitive summary flow functions can be constructed by building a points-to graph which has been explained before. An application of such a summary flow function requires traversing the graph.

In our example program, assuming that it is known that the value of b is 2 after every call to procedure p, a flow insensitive analysis of procedure q would conclude that a could be both 1 and 2 and hence is not constant in q. This is a safe conclusion when only gross information instead of fine grained point-specific information about q is desired.

In general, flow insensitive analysis is not common at the intraprocedural level.

Context insensitivity

A calling context is represented by the snapshot of the control stack at run time. During program analysis, it is determined by the sequence of unfinished calls in a path in the supergraph.

As explained in Chapter 1, context insensitive analysis does not distinguish between different calling contexts. Instead, the inherited data flow information from all contexts is merged and the resulting synthesized data flow information is propagated to all calling contexts indiscriminately. This implies traversing interprocedurally invalid paths—paths in which calls and returns do not match. In essence, there is no distinction between the interprocedural and intraprocedural edges in a supergraph.

In our program, procedure p inherits $\langle 5, 3, 7, \perp\rangle$ from its call in the *main* and $\langle 1, 2, 7, 14\rangle$ from its call in procedure q. The merged value is $\langle \perp, \perp, 7, \perp\rangle$ and the

resulting synthesized value $\langle \perp, 2, 7, \perp \rangle$ is propagated back to both the callers of p. As a consequence, such an analysis fails to discover the fact that a is constant with value 5 at the entry of block n_1. Effectively, this is a consequence of propagating the value $a = 1$ from Start_q to n_1. Although there is a path from Start_q to n_1 in the supergraph, it does not represent matching calls and returns: Data flow information computed along the path from Start_q to End_p should be propagated to R_3 and not to R_1 because the last call in this path represents a call to p from q and not from *main*. Context sensitive analysis excludes such paths and restricts $P(u)$ to interprocedurally valid paths.

The issue of context sensitivity does not arise at the intraprocedural level.

7.4.2 Side Effects Analysis

Interprocedural analysis requires incorporating the mutual influence of callers and callees on each other. This requires computing both inherited and synthesized part of data flow information. We call such an analysis, a *whole program* analysis. This should be contrasted with the situation when only callee's influence on callers is computed. This is achieved by computing the synthesized part of interprocedural data flow information; the inherited part is approximated by a fixed value for each procedure. Traditionally, such analyses have been called *side effects* analyses.

A side effects analysis can also have some variations depending upon whether only the context insensitive side effects are computed or the context sensitive side effects are also computed. For a given procedure p, the context insensitive side effects are represented by $\mathit{ConstGen}_p$ while the context sensitive side effects are represented by $\mathit{DepGen}_p(\mathsf{x})$ and $\mathsf{x} - (\mathit{ConstKill}_p \cup \mathit{DepKill}_p(\mathsf{x}))$. The former is much simpler but less useful compared to the latter.

For a given procedure call, side effect analysis restricts the scope of optimization to the caller whereas whole program analysis facilitates optimization in both caller and callee. For example, if interprocedural live variables analysis is performed using side effects, it is possible to decide whether a value in a register should be preserved across a procedure call. The transformation resulting from this decision is restricted to a caller's body. However, if whole program analysis is performed, it may be possible to assign the same register to a variable both within a caller and its callee.

7.5 Language Features Influencing Interprocedural Analysis

Interprocedural data flow analysis is influenced by language features that support high level abstractions related to procedure calls.

In this chapter, we have deliberately used a non-recursive program to introduce interprocedural data flow analysis. In the presence of recursion, functional approaches require fixed point computation to construct summary flow functions. Convergence

of this computation needs to be established by examining the flow functions and data flow values in the framework. Since the value-based approaches have to explicitly remember contexts, a mechanism of summarizing the contexts needs to be devised. For the frameworks with finite lattices, it is possible to bound the number of contexts by a finite number without compromising on precision. However, the number of contexts remains very large. Thus recursion affects both the feasibility and the efficiency of interprocedural data flow analysis significantly. Many practical value-based approaches perform context insensitive analysis in the recursive portions of programs. However, it is possible to perform context sensitive interprocedural analysis in the presence of recursions. We present such methods in Chapters 8 and 9.

The other simplifying feature of our program was that it did not involve parameters and local variables. In practice, parameterless procedures are rare and it is important to handle the parameter passing mechanism because computation of inherited data flow information requires transferring the data flow information of actual parameters to that of the corresponding formal parameters. For this purpose, the call by value parameter passing mechanism can be modeled by simple assignments whereas call by reference parameter passing mechanism should be modeled by pointer assignments. Further, distinction should be made between global variables and local variables for inherited and synthesized data flow information. Unless local variables are involved in the actual parameters of a procedure call, synthesized data flow information should not be computed for local entities nor should their data flow information be propagated as a part of the inherited data flow information of the callee. Recall that the motivating example of heap data analysis presented in Section 1.1 contains local pointer variables that are passed as actual parameters. Section 9.5 performs interprocedural liveness analysis for that example and describes how transfer of data flow information between actual and formal parameters can be modeled.

Further, in the presence of parameter passing by reference, depending upon the actual parameters a particular call may create aliasing between formal parameters or between formal parameters and global variables within the callee's body. This may affect the correctness or precision of the data flow information discovered. Section 8.2 shows how such aliasing can be discovered.

Some languages support local functions. This influences interprocedural analysis in the following ways: (a) The possible call structure in a program is governed by the scope rules of the language that restrict the visibility of local procedures. (b) The notion of global variables must now be replaced by the notion of non-local variables that depend on the scope of a procedure.

Function pointers and subtyping mechanism resulting in dynamic dispatch of function calls hide the identity of the called procedures at compile time making the static call structure imprecise. Exception handling mechanisms of a language have a similar effect. Interprocedural data flow analyses are restricted to single threads, similar to intraprocedural data flow analysis. Use of library functions imply that the entire source is not available to an interprocedural analyzer and a summary of their effects must be provided explicitly.

7.6 Common Variants of Interprocedural Data Flow Analysis

We introduce the following common variants using our running example.

- *Intraprocedural analysis with conservative approximation.* We use conservative approximation of inherited and synthesized data flow information for handling procedure calls.
- *Intraprocedural analysis with side effects.* We compute flow sensitive as well as flow insensitive side effects and represent them by context independent flow functions.
- *Whole program analysis.* We perform context sensitive as well as context insensitive analysis.

In each case, the data flow information in the caller procedures is computed in flow sensitive manner and the data flow value associated each program point is computed separately. In the case of flow insensitive side effects, only the effect of a call is flow insensitive—the data flow values computed in the caller's body are flow sensitive.

Although flow insensitive analysis of all procedures has also been used in practice, it computes a single summary data flow value per procedure which is usually very imprecise. For example, a flow insensitive constant propagation of our program computes the data flow value $\langle \widehat{\perp}, \widehat{\perp}, \widehat{\perp}, \widehat{\perp} \rangle$ for procedures *main* and q and, $\langle \widehat{\perp}, 2, \widehat{\perp}, \widehat{\perp} \rangle$ for procedure p. This value is same regardless of the variant. Flow sensitive version of these variants compute data flow values with varying degrees of precision.

7.6.1 Intraprocedural Analysis with Conservative Interprocedural Approximation

Intraprocedural analysis with conservative interprocedural approximation involves using safe values for inherited and synthesized data flow information. This approach was introduced in Section 1.1 analysis of heap data.

The inherited data flow information for constant propagation is represented by $BI_{main} = \langle 0,0,0,0 \rangle$ and $BI_p = BI_q = \langle \widehat{\perp}, \widehat{\perp}, \widehat{\perp}, \widehat{\perp} \rangle$. This distinction arises from the fact that all our variables are global variables which are initialized to 0; however, their values cannot be assumed to be known when other procedures are invoked. For local variables, the value in BI is $\widehat{\top}$ but our program does not have local variables. For live variables analysis, $BI_{main} = \emptyset$ because no variable is live at the end of the program. However, all global variables should be conservatively assumed to be live at the end of other procedures, hence $BI_p = BI_q = \{a,b,c,d\}$.

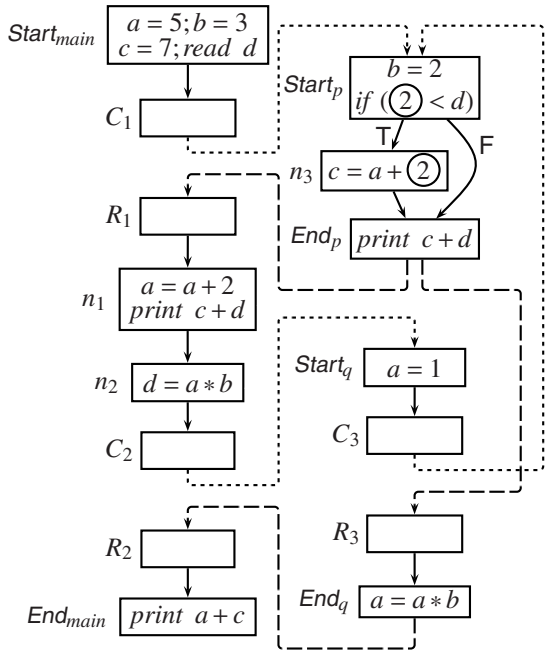
The synthesized data flow information for constant propagation is conservatively represented by $\langle \widehat{\perp}, \widehat{\perp}, \widehat{\perp}, \widehat{\perp} \rangle$ under the assumption that a function call could modify all variables. For live variables analysis, the synthesized data flow information is $\{a,b,c,d\}$ because it is conservatively assumed that all global variables are live at the

Block and associated data flow value		Intraprocedural analysis with conservative interprocedural approximation	Side Effects Analysis		Whole Program Analysis	
			Flow sensitivity of synthesized information		Context sensitivity of inherited information	
			Insensitive	Sensitive	Insensitive	Sensitive
$Start_m$	*In*	$\langle 0,0,0,0\rangle$	$\langle 0,0,0,0\rangle$	$\langle 0,0,0,0\rangle$	$\langle 0,0,0,0\rangle$	$\langle 0,0,0,0\rangle$
	Out	$\langle 5,3,7,\widehat{\perp}\rangle$	$\langle 5,3,7,\widehat{\perp}\rangle$	$\langle 5,3,7,\widehat{\perp}\rangle$	$\langle 5,3,7,\widehat{\perp}\rangle$	$\langle 5,3,7,\widehat{\perp}\rangle$
C_1	*In, Out*	$\langle 5,3,7,\widehat{\perp}\rangle$	$\langle 5,3,7,\widehat{\perp}\rangle$	$\langle 5,3,7,\widehat{\perp}\rangle$	$\langle 5,3,7,\widehat{\perp}\rangle$	$\langle 5,3,7,\widehat{\perp}\rangle$
R_1	*In, Out*	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 5,\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 5,2,7,\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle 5,2,7,\widehat{\perp}\rangle$
n_1	*In*	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 5,\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 5,2,7,\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle 5,2,7,\widehat{\perp}\rangle$
	Out	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 7,\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 7,2,7,\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle 7,2,7,\widehat{\perp}\rangle$
n_2	*In*	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 7,\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 7,2,7,\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle 7,2,7,\widehat{\perp}\rangle$
	Out	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 7,\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 7,2,7,14\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle 7,2,7,14\rangle$
C_2	*In, Out*	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 7,\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 7,2,7,14\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle 7,2,7,14\rangle$
R_2	*In, Out*	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 2,2,\widehat{\perp},14\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle 2,2,\widehat{\perp},14\rangle$
End_m	*In, Out*	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 2,2,\widehat{\perp},14\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle 2,2,\widehat{\perp},14\rangle$
$Start_p$	*In*	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},\widehat{\perp},7,\widehat{\perp}\rangle$
	Out	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,7,\widehat{\perp}\rangle$
n_3	*In*	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,7,\widehat{\perp}\rangle$
	Out	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$
End_p	*In, Out*	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$
$Start_q$	*In*	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle 7,2,7,14\rangle$
	Out	$\langle 1,\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 1,\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 1,\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 1,2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle 1,2,7,14\rangle$
C_3	*In, Out*	$\langle 1,\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 1,\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 1,\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 1,2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle 1,2,7,14\rangle$
R_3	*In, Out*	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 1,\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 1,2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle 1,2,\widehat{\perp},14\rangle$
End_q	*In*	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 1,\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 1,2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle 1,2,\widehat{\perp},14\rangle$
	Out	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},\widehat{\perp},\widehat{\perp},\widehat{\perp}\rangle$	$\langle 2,2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle \widehat{\perp},2,\widehat{\perp},\widehat{\perp}\rangle$	$\langle 2,2,\widehat{\perp},14\rangle$

FIGURE 7.6
Constant propagation in our example program using flow sensitive version of common variants of interprocedural data flow analysis.

entry of p and q. This does not contradict the assumption made for constant propagation because although a global variable may be modified in the callee procedure, it cannot be guaranteed to be modified along all paths before being used. These assumptions are safe because they cannot enable incorrect optimizations.

From Figure 7.6, it is easy to see that intraprocedural analysis limits the scope of constant propagation to a single procedure and disables it across function calls. As illustrated in Figure 7.7 on the following page, only variable b in blocks $Start_p$ and n_3 is replaced by its value which happens to be 2. The result of performing liveness analysis on the program obtained after constant propagation is shown in Figure 7.7 on the next page. Our analysis concludes that all left hand side variables in the assignments are live after the assignments. Thus this variant of analysis fails to enable dead code elimination in our example program.



Block n	In_n	Out_n
$Start_{main}$	$\emptyset$	$\{a,b,c,d\}$
C_1	$\{a,b,c,d\}$	$\{a,b,c,d\}$
R_1	$\{a,b,c,d\}$	$\{a,b,c,d\}$
n_1	$\{a,b,c,d\}$	$\{a,b,c\}$
n_2	$\{a,b,c\}$	$\{a,b,c,d\}$
C_2	$\{a,b,c,d\}$	$\{a,b,c,d\}$
R_2	$\{a,c\}$	$\{a,c\}$
End_{main}	$\{a,c\}$	$\emptyset$
$Start_p$	$\{a,c,d\}$	$\{a,b,c,d\}$
n_3	$\{a,b,d\}$	$\{a,b,c,d\}$
End_p	$\{a,b,c,d\}$	$\{a,b,c,d\}$
$Start_q$	$\{b,c,d\}$	$\{a,b,c,d\}$
C_3	$\{a,b,c,d\}$	$\{a,b,c,d\}$
R_3	$\{a,b,c,d\}$	$\{a,b,c,d\}$
End_q	$\{a,b,c,d\}$	$\{a,b,c,d\}$

FIGURE 7.7
Intraprocedural liveness analysis after intraprocedural constant propagation. Propagated constants are shown in circles.

7.6.2 Intraprocedural Analysis with Side Effects Computation

Side effect analysis discovers which variables are actually modified in a procedure calls. Hence it can compute more precise synthesized data flow information. Side effect computation could be flow insensitive or flow sensitive. We compute the data flow information in the body of a caller in a flow sensitive manner. In either case, since no data flow information is inherited, the value of *BI* is same as in intraprocedural analysis.

After computing the side effects, function calls are treated as basic blocks and conventional intraprocedural analysis is performed.

Flow insensitive side effects

We present two methods of computing flow insensitive side effects. The first method uses symbolic values to parametrize the context. The other method works for frameworks in which the flow functions do not contain dependent parts. This method computes the values of *ConstGen* and *ConstKill* components of the summary side effect flow function of a procedure explicitly. We illustrate the former method for constant propagation and the latter for live variables analysis.

- *Flow insensitive side effects for constant propagation.*

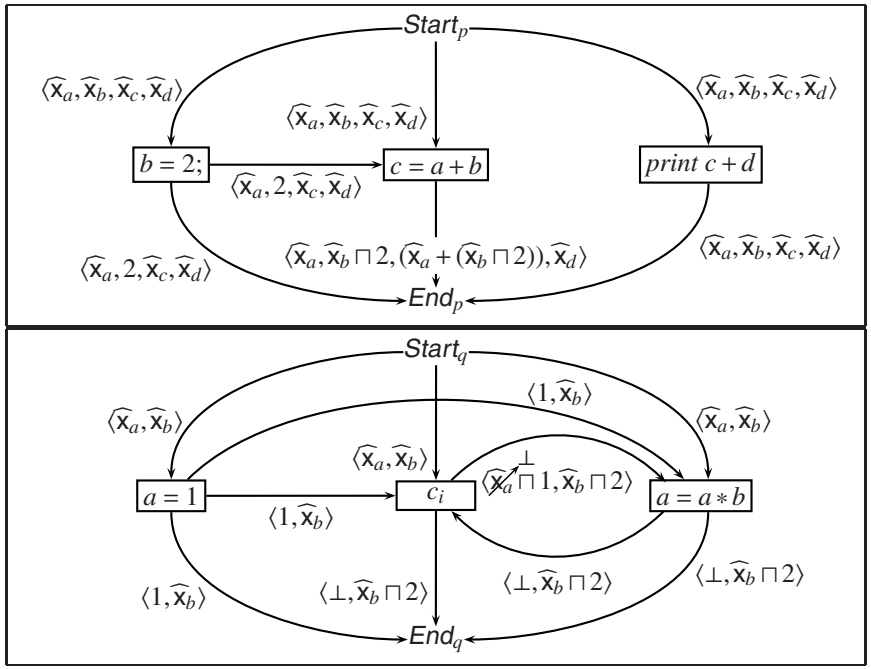


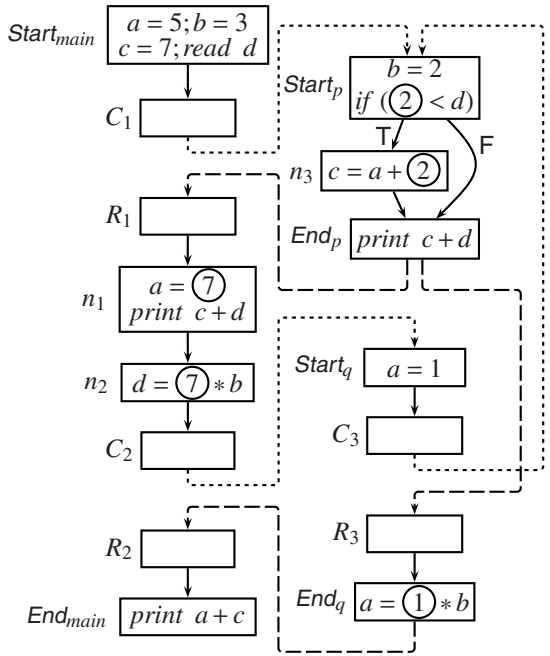
FIGURE 7.8
Computing flow insensitive side effect functions for procedures p and q. An edge $u \rightarrow v$ denotes the fact that $DepGen_v(\mathsf{x})$ depends on the data flow value at u; the required data flow values have been shown along with the out edges of u. For procedure q, we show the values of variables a and b only.

Recall that a systematic construction of summary flow functions is possible only for a limited set of data flow frameworks; in general, it is not possible for full constant propagation. Here we use Equation (7.1) intuitively to symbolically compute summary function for constant propagation and illustrate the difficulty in automatic construction of flow functions for constant propagation.

Computation of flow insensitive side effect summary flow functions for procedure p and q are illustrated in Figure 7.8. It is easy to see that:

$$f_p(\langle \widehat{\mathsf{x}}_a, \widehat{\mathsf{x}}_b, \widehat{\mathsf{x}}_c, \widehat{\mathsf{x}}_d \rangle) = \langle \widehat{\mathsf{x}}_a, \widehat{\mathsf{x}}_b \sqcap 2, \widehat{\mathsf{x}}_c \sqcap (\widehat{\mathsf{x}}_a + (\widehat{\mathsf{x}}_b \sqcap 2)), \widehat{\mathsf{x}}_d \rangle$$

For computing the summary side effect function for q, we need to incorporate the effect of procedure p too. For simplicity, only the computation for variables a and b for procedure p is illustrated in Figure 7.8. The expression that represents the data flow value of a after processing the assignment $a = a * b$ is $\Big(\widehat{\mathsf{x}}_a \sqcap 1 \sqcap \big((\widehat{\mathsf{x}}_a \sqcap 1) * (\widehat{\mathsf{x}}_b \sqcap 2)\big)\Big)$. Using monotonicity of the flow function repre-



Block n	In_n	Out_n
$Start_{main}$	$\emptyset$	$\{a,b,c,d\}$
C_1	$\{a,b,c,d\}$	$\{a,b,c,d\}$
R_1	$\{b,c,d\}$	$\{b,c,d\}$
n_1	$\{b,c,d\}$	$\{a,b,c\}$
n_2	$\{a,b,c\}$	$\{a,b,c,d\}$
C_2	$\{a,b,c,d\}$	$\{a,b,c,d\}$
R_2	$\{a,c\}$	$\{a,c\}$
End_{main}	$\{a,c\}$	$\emptyset$
$Start_p$	$\{a,c,d\}$	$\{a,b,c,d\}$
n_3	$\{a,b,d\}$	$\{a,b,c,d\}$
End_p	$\{a,b,c,d\}$	$\{a,b,c,d\}$
$Start_q$	$\{b,c,d\}$	$\{a,b,c,d\}$
C_3	$\{a,b,c,d\}$	$\{a,b,c,d\}$
R_3	$\{b,c,d\}$	$\{b,c,d\}$
End_q	$\{b,c,d\}$	$\{a,b,c,d\}$

FIGURE 7.9
Interprocedural liveness analysis after interprocedural constant propagation using flow insensitive side effects. The resulting values after constant propagation and constant folding are shown in circles.

senting multiplication in constant propagation, it can be reduced as follows:

$$\begin{aligned}\widehat{x}_a \sqcap 1 \sqcap \left((\widehat{x}_a \sqcap 1) * (\widehat{x}_b \sqcap 2)\right) &\sqsubseteq \widehat{x}_a \sqcap 1 \sqcap \left((\widehat{x}_a * \widehat{x}_b) \sqcap (\widehat{x}_a * 2) \sqcap (1 * 2) \sqcap (1 * \widehat{x}_b)\right) \\ &\sqsubseteq \widehat{x}_a \sqcap 1 \sqcap (\widehat{x}_a * \widehat{x}_b) \sqcap (\widehat{x}_a * 2) \sqcap 2 \sqcap \widehat{x}_b \\ &\sqsubseteq \widehat{\perp}\end{aligned}$$

Intuitively, a can be both 1 and 2, hence it must be $\widehat{\perp}$. Using this, the final summary side effect function is:

$$f_q(\langle \widehat{x}_a, \widehat{x}_b, \widehat{x}_c, \widehat{x}_d \rangle) = \langle \widehat{\perp}, \widehat{x}_b \sqcap 2, \widehat{\perp}, \widehat{x}_d \rangle$$

Observe that devising a systematic method that can perform the reductions such as above is not easy.

The details of constants discovered in each basic block are shown in Figure 7.6 on page 247. The resulting constant propagation is shown in Figure 7.9. Observe that the number 7 in blocks n_1 and n_2 is a result of constant folding. The use of flow insensitive side effects in intraprocedural analysis results in more precise data flow information compared to the data flow information computed using the conservative approximation of function calls.

- *Flow insensitive side effects for live variables analysis.*

  We need to compute *MustKill* and *MayUse* sets for procedures p and q. We compute them for the program in Figure 7.9 on the facing page.

$$
\begin{aligned}
\mathit{MustKill}_p &= \mathit{Kill}_{\mathit{Start}_p} \cap \mathit{Kill}_{n_3} \cap \mathit{Kill}_{\mathit{End}_p} &&= \emptyset \\
\mathit{MustKill}_q &= \mathit{Kill}_{\mathit{Start}_q} \cap \mathit{MustKill}_p \cap \mathit{Kill}_{\mathit{End}_q} &&= \emptyset \\
\mathit{MayUse}_p &= \mathit{Gen}_{\mathit{Start}_p} \cup \mathit{Gen}_{n_3} \cup \mathit{Gen}_{\mathit{End}_p} &&= \{a,c,d\} \\
\mathit{MayUse}_q &= \mathit{Gen}_{\mathit{Start}_q} \cup \mathit{MayUse}_p \cup \mathit{Gen}_{\mathit{End}_q} &&= \{a,b,c,d\} \\
f_p(\mathsf{x}) &= (\mathsf{x} - \mathit{MustKill}_p) \cup \mathit{MayUse}_p &&= \mathsf{x} \cup \{a,c,d\} \\
f_q(\mathsf{x}) &= (\mathsf{x} - \mathit{MustKill}_q) \cup \mathit{MayUse}_q &&= \mathsf{x} \cup \{a,b,c,d\}
\end{aligned}
$$

  Note that $f_p(\mathsf{x})$ is a little better than the conservative approximations used in Section 7.6.1 in that it does not contain b. However, due to flow insensitivity, it does not recognize that b is killed in procedure p. Hence, the use of b in block n_2 cause b to be considered live at the exit of Start_{main}. The resulting data flow information after performing constant propagation using flow insensitive side effects is shown in Figure 7.9 on the preceding page. Observe that no variable is dead immediately after its assignment hence dead code elimination is not possible using this variant also in spite of the fact that some more constant are discovered and liveness information has become more precise.

Flow sensitive side effects

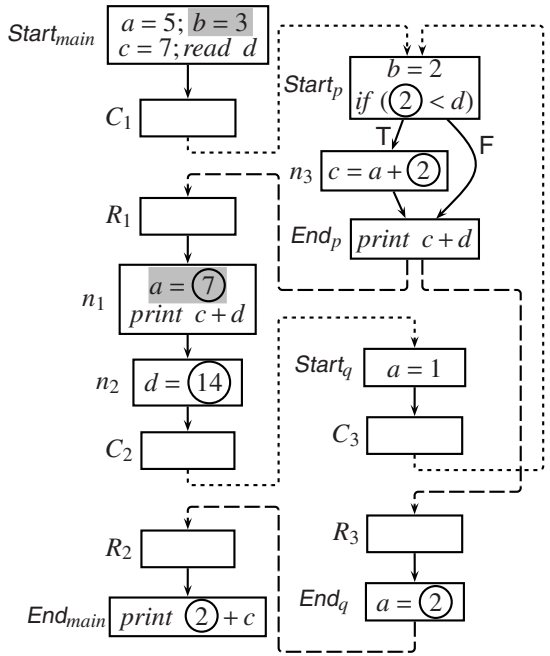
As in the previous section, we compute *Kill* and *Gen* implicitly for constant propagation and explicitly for live variables analysis.

Flow sensitive side effects for constant propagation can be computed by performing data flow analysis over a called procedure with $\mathit{BI} = \langle \widehat{\mathsf{x}}_a, \widehat{\mathsf{x}}_b, \widehat{\mathsf{x}}_c, \widehat{\mathsf{x}}_d \rangle$. The resulting flow functions are represented by the symbolic data flow values at the exit of the function. It is easy to see that:

$$
\begin{aligned}
f_p(\langle \widehat{\mathsf{x}}_a, \widehat{\mathsf{x}}_b, \widehat{\mathsf{x}}_c, \widehat{\mathsf{x}}_d \rangle) &= \langle \widehat{\mathsf{x}}_a, 2, \widehat{\mathsf{x}}_c \sqcap (\widehat{\mathsf{x}}_a + 2), \widehat{\mathsf{x}}_d \rangle \\
f_q(\langle \widehat{\mathsf{x}}_a, \widehat{\mathsf{x}}_b, \widehat{\mathsf{x}}_c, \widehat{\mathsf{x}}_d \rangle) &= \langle 2, 2, \widehat{\mathsf{x}}_c \sqcap 3, \widehat{\mathsf{x}}_d \rangle
\end{aligned}
$$

It is clear from Figure 7.6 on page 247 that flow sensitive side effects enable detecting more constants than flow insensitive side effects. The resulting constant propagation and constant folding is shown in Figure 7.10 on the next page.

For liveness analysis, we compute flow sensitive *MustKill* and *MayUse* by traversing the CFG in post order. *MustKill* is computed by discovering the sets of variables that are modified in basic blocks such that these modifications are upwards exposed. If a variable is used in a basic block, it is removed from the set. By contrast, *MayUse*



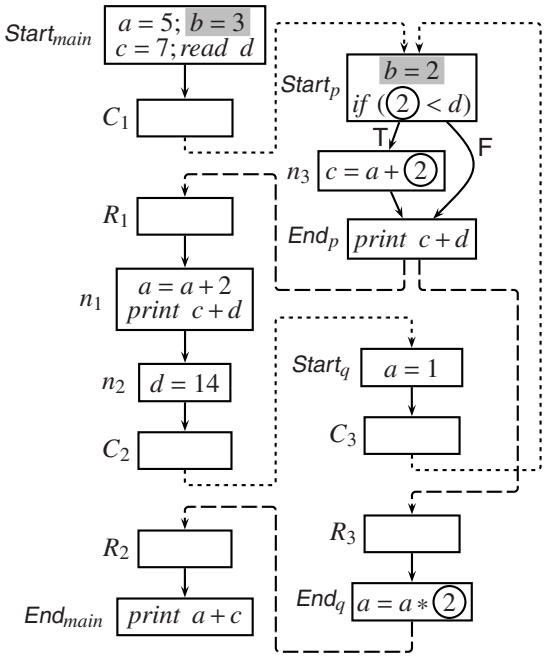
Block n	In_n	Out_n
$Start_{main}$	$\emptyset$	$\{a,c,d\}$
C_1	$\{a,c,d\}$	$\{a,c,d\}$
R_1	$\{c,d\}$	$\{c,d\}$
n_1	$\{c,d\}$	$\{c\}$
n_2	$\{c\}$	$\{c,d\}$
C_2	$\{c,d\}$	$\{c,d\}$
R_2	$\{c\}$	$\{c\}$
End_{main}	$\{c\}$	$\emptyset$
$Start_p$	$\{a,c,d\}$	$\{a,b,c,d\}$
n_3	$\{a,b,d\}$	$\{a,b,c,d\}$
End_p	$\{a,b,c,d\}$	$\{a,b,c,d\}$
$Start_q$	$\{c,d\}$	$\{a,c,d\}$
C_3	$\{a,c,d\}$	$\{a,c,d\}$
R_3	$\{b,c,d\}$	$\{b,c,d\}$
End_q	$\{b,c,d\}$	$\{a,b,c,d\}$

FIGURE 7.10
Interprocedural liveness analysis after interprocedural constant propagation using flow sensitive side effects. Highlighted statements are dead assignments and can be eliminated.

is computed using live variables analysis. *BI* is $\emptyset$ for both *MustKill* and *MayUse*. The resulting flow functions are:

$$
\begin{aligned}
MustKill_p &= \{b\} \\
MayUse_p &= \{a,c,d\} \\
f_p(\mathsf{x}) &= (\mathsf{x} - \{b\}) \cup \{a,c,d\} \\
MustKill_q &= \{a,b\} \\
MayUse_q &= \{c,d\} \\
f_q(\mathsf{x}) &= (\mathsf{x} - \{a,b\}) \cup \{c,d\}
\end{aligned}
$$

Figure 7.10 shows the liveness analysis on the program in which constant propagation has been performed. Both analyses use flow sensitive side effects to incorporate the effect of a procedure call. The results of both analyses are more precise compared to the results obtained by using flow insensitive side effects. Further, dead code elimination also becomes possible. Variable b is not live at the exit of $Start_{main}$ and $Start_p$. Hence the assignments to these variables in the respective blocks can



Block n	In_n	Out_n
$Start_{main}$	$\emptyset$	$\{a,c,d\}$
C_1	$\{a,c,d\}$	$\{a,c,d\}$
R_1	$\{a,c,d\}$	$\{a,c,d\}$
n_1	$\{a,c,d\}$	$\{a,c\}$
n_2	$\{a,c\}$	$\{c,d\}$
C_2	$\{c,d\}$	$\{c,d\}$
R_2	$\{a,c\}$	$\{a,c\}$
End_{main}	$\{a,c\}$	$\emptyset$
$Start_p$	$\{a,c,d\}$	$\{a,c,d\}$
n_3	$\{a,d\}$	$\{a,c,d\}$
End_p	$\{a,c,d\}$	$\{a,c,d\}$
$Start_q$	$\{c,d\}$	$\{a,c,d\}$
C_3	$\{a,c,d\}$	$\{a,c,d\}$
R_3	$\{a,c\}$	$\{a,c\}$
End_q	$\{a,c\}$	$\{a,c\}$

FIGURE 7.11
Interprocedural liveness analysis after interprocedural constant propagation using context insensitive whole program analysis.

be deleted. Observe that this still does not cover all dead assignments shown in Figure 7.1 on page 234.

7.6.3 Whole Program Analysis

We now present interprocedural analyses that compute both inherited and synthesized data flow information. As usual, the analyses are flow sensitive. Since inherited data flow information depends on the callers alone, we present two possible variants: (a) Context insensitive analysis, and (b) Context sensitive analysis.

Context insensitive whole program analysis

Conceptually, the simplest method of performing context insensitive whole program analysis is to treat a supergraph as single control flow graph and compute data flow properties with a block from all its neighbours without distinguishing between interprocedural and intraprocedural edges. Thus, the constants reaching $Start_p$ are a merge of the constants available at C_1 and C_3. This merged information is then used to compute the synthesized information which is then propagated to both R_1 and R_3. Thus the data flow information at C_3 influences the data flow information at R_1 in spite of the fact that there is no control flow from C_3 to R_1. Thus this method

propagates information flow along interprocedurally invalid paths too causing an imprecision in the context sensitive part of the synthesized data flow information; the context insensitive part remains unaffected.

The details of constants that get propagated to each program point in this method are presented in Figure 7.6 on page 247. The optimized program after constant propagation and the result of liveness analysis on this program are shown in Figure 7.11 on the previous page. Merging inherited data flow information results in loss of precision in the synthesized data flow information because interprocedurally invalid paths are also covered. This happens in spite of computing flow sensitive synthesized data flow information.

Interestingly, in our example program, this method discovers fewer constants than the method using flow insensitive side effects. Yet it performs some dead code elimination whereas the latter does not. This is because this method discovers more precise liveness information: With flow insensitive side effects, the liveness of variable b is not killed in procedure p. On the other hand, the synthesized information computed by flow sensitive side effects discovers that b is not live. However, this method does not compare favourably with the method that uses flow sensitive side effects—the latter computes precise synthesized information while merging inherited data flow information introduces some imprecision in the synthesized data flow information computed by this method.

Context sensitive whole program analysis

Our final method of interprocedural data flow analysis does not merge inherited data flow information while computing the synthesized data flow information. Thus the context sensitive part of synthesized data flow information is more precise than in the context insensitive whole program analysis.

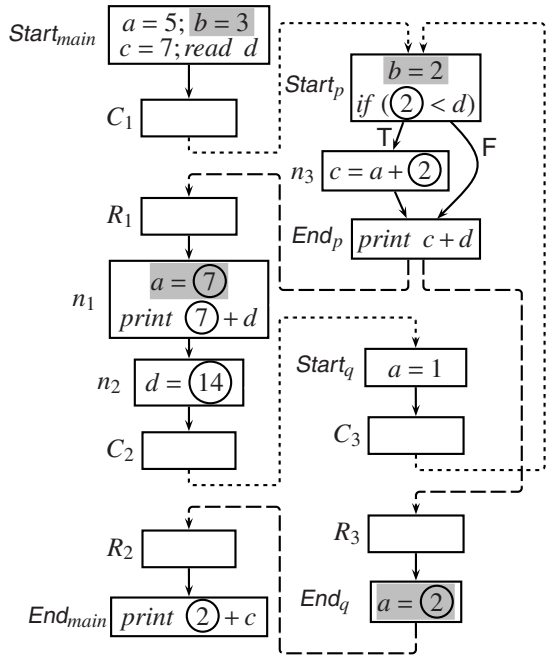
It uses the same flow functions as used by the flow sensitive side effects. The main difference is that in that method, the inherited data flow information was represented by a conservative approximation. This method computes the inherited information from calling contexts and propagates it within the callee's body.

The resulting optimization is shown in Figure 7.12 on the facing page. The dead code discovered by this method matches the dead code shown in Figure 7.1 on page 234.

Observe that this method fails to discover that the value of c is 3 in End_{main}. It can be discovered by context sensitive whole program conditional constant propagation.

7.7 An Aside on Interprocedural Optimizations

A lot of work that analyses programs at the interprocedural level is directed at interprocedural optimizations like procedure inlining and cloning. The analyses required for these optimizations are different from the analyses that are presented in



Block n	In_n	Out_n
$Start_{main}$	$\emptyset$	$\{a,c,d\}$
C_1	$\{a,c,d\}$	$\{a,c,d\}$
R_1	$\{c,d\}$	$\{c,d\}$
n_1	$\{c,d\}$	$\{c\}$
n_2	$\{a,c\}$	$\{c,d\}$
C_2	$\{c,d\}$	$\{c,d\}$
R_2	$\{c\}$	$\{c\}$
End_{main}	$\{c\}$	$\emptyset$
$Start_p$	$\{a,c,d\}$	$\{a,c,d\}$
n_3	$\{a,d\}$	$\{c,d\}$
End_p	$\{c,d\}$	$\{c\}$
$Start_q$	$\{c,d\}$	$\{a,c,d\}$
C_3	$\{a,c,d\}$	$\{a,c,d\}$
R_3	$\{c\}$	$\{c\}$
End_q	$\{c\}$	$\{c\}$

FIGURE 7.12
Interprocedural liveness analysis after interprocedural constant propagation using context sensitive whole program analysis.

this book. Often they involve a single traversal over program representation. For example, procedure inlining analyses parameters and checks that there is no recursive call. The final decision to inline is taken based on a collection of heuristics supported by empirical evidence. Then a transformation pass renames global variables and performs inlining by traversing the call graph bottom up. Procedure cloning is based on analyzing actual parameters from different call sites and their effects on the called procedures. Typically, the option of cloning is considered when constant values are passed as actual parameters. Again, the final decision depends on a collection of thumb rules. Most production compilers gainfully employ these optimizations. An additional advantage of these optimizations is that they enhance the possibility of intraprocedural optimizations.

The next set of interprocedural optimizations employed by production compilers are actually more aggressive intraprocedural optimizations using side effects of procedure calls. The common side effect that most compilers try to detect is potential modifications of global variables and reference parameters.

Finally, many interprocedural optimizations do involve systematic analyses. However, for reasons of efficiency and scalability, most of these analyses are rooted in specific optimizations e.g., constant propagation, side effect analysis, points-to analysis etc. There is a large body of work along these lines but it seems difficult to

build useful generalizations across these methods. Besides, efficiency and scalability concerns have often resulted in these methods being flow insensitive or context insensitive or both.

7.8 Summary and Concluding Remarks

In this part we focus on generalizations in keeping with the theme of the book and present generic methods that naturally allow interprocedural analysis of the formulations presented in Part I of the book.

This chapter has presented flow and context sensitivity as two features that influence the precision of interprocedural data flow information. Further, it has identified constructing summary flow functions versus computing values as two fundamentally different approaches of performing interprocedural analysis. Subsequent chapters use these concepts and primarily focus on methods that are flow and context sensitive. Chapter 8 presents general methods of constructing summary flow functions whereas Chapter 9 presents methods that compute data flow information at each point by maintaining distinct contexts.

7.9 Bibliographic Notes

The earliest studies of interprocedural data flow analysis were motivated by the need of discovering side effects. The work by Spillman [94] was directed at finding out side effects in terms of values modified by the called procedure. This analysis was performed by traversing a call graph from callees to callers. Allen [6] addressed a slightly more general problem of additionally finding out values used by callees also. However, unlike Spillman's method, Allen's method was not suited for recursive programs. Barth [13, 14] introduced a much more general formulation based on computing transitive closures of relationships. This method allowed asking a wider range of questions such as whether variables shared storage or not, whether variables were modified or used etc. More importantly, he introduced the notions of *must* and *may* in the data flow information discovered. Banning [12] was the first to make a distinction between flow sensitive and flow insensitive side effect computation.

The concept of context sensitivity was introduced by Sharir and Pnueli [93] which can be easily called the most influential work on interprocedural data flow analysis. We present their concepts in greater details in the next two chapters.

Effectiveness of interprocedural data flow analysis was studied by Richardson and Ganapathi [83], Grove and Torczon [39], and Martin [72]. Lhoták and Hendren [69] have empirically observed that in the presence of recursive calls, context insensitivity

leads to significant imprecision.

Duesterwald, Gupta and Soffa [32, 33] present an interesting alternative of computing interprocedural data flow information incrementally on demand.

An important issue in interprocedural data flow analysis is precise call graph construction. This becomes difficult in the presence of function pointers in a language like C and virtual functions and dynamic dispatch of methods in object oriented languages. Early works along these lines were done by Hall and Kennedy [43] and by Callahan, Carle, Hall and Kennedy. [20]. Grove and Chambers [38] present a more recent detailed treatment of call graph construction. We do not address this issue in this book.