8

Functional Approach to Interprocedural Data Flow Analysis

Functional approach to interprocedural data flow analysis constructs context independent summary flow functions which are then used in the calling contexts to compute the data flow information synthesized by called procedures in the body of the caller procedures. Data flow information inherited by a procedure is computed from the calling contexts of the procedure. The main advantage of constructing context independent summary flow functions is that a procedure needs to be analyzed only once regardless of the number of calls to it.

We begin by presenting the classical side effects analysis for bit vector frameworks as a special case of constructing summary flow functions. This is followed by context and flow sensitive whole program analysis. Finally we show how the explicit construction of summary flow functions can be avoided by enumerating the function in terms of pairs of input output values.

For simplicity, we focus on data flow analysis of global variables. We present orthogonal techniques of handling the effects of parameters. We restrict the analysis to languages that do not contain nested procedures.

8.1 Side Effects Analysis of Procedure Calls

Classical side effects analysis focuses on computing the effect of a callee procedure on the variables of the caller procedure in order to discover more optimization opportunities in the caller procedures. In particular, the following side effects are directly relevant: For a given variable v and a given callee s in a procedure r

- Is the execution of r guaranteed to modify the value of v?
- Can the execution of r modify the value of v?
- Is the execution of r guaranteed to use the value of v before modifying it?
- Can the execution of r use the value of v before modifying it?

The variables for which the above answers are in affirmative are contained in $\mathit{MustKill}_r$, $\mathit{MayKill}_r$, $\mathit{MustUse}_r$, and MayUse_r respectively. Clearly, the *must* properties are all paths properties whereas *may* properties are some path properties.

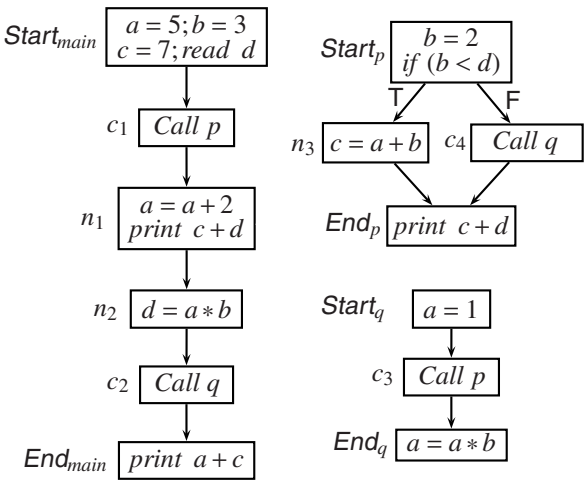


FIGURE 8.1
Modified version of the program in Figure 7.2.

These basic side effects of a procedure can be used to answer a variety of questions. For example, liveness analysis can handle call to procedure r by computing In_n of the call block as follows:

$$In_n = (Out_n - MustKill_r) \cup MayUse_r$$

Observe that liveness information of a variable is killed only when it is guaranteed to be modified in the callee along all execution paths. Available expression analysis, on the other hand, should use kill the availability of expressions whose operands are in $MayKill_r$. The variables that are guaranteed to preserve their values across a call to procedure r are contained in $\mathbb{G}\text{var} - MayKill_r$ where $\mathbb{G}\text{var}$ denotes the set of global variables. The variables that may preserve their values along some path through procedure r are contained in $\mathbb{G}\text{var} - MustKill_r$.

The $\sqcap$, $\top$ and $\bot$ values for computing the above side effect properties are:

Property	$\sqcap$	$\top$	$\bot$	Explanation of $\bot$
$MustKill_r$	$\cap$	$\mathbb{G}\text{var}$	$\emptyset$	No variable can be guaranteed to be necessarily killed by r
$MustUse_r$	$\cap$	$\mathbb{G}\text{var}$	$\emptyset$	No variable can be guaranteed to be necessarily used in r before being modified
$MayUse_r$	$\cup$	$\emptyset$	$\mathbb{G}\text{var}$	Any variable may be used in r along some path or the other
$MayKill_r$	$\cup$	$\emptyset$	$\mathbb{G}\text{var}$	Any variable may be killed in r along some path or the other

We use the program in Figure 8.1 as a running example in this section. This program is same as the program in the previous chapter except that we have now

included a call to procedure q in procedure p to make the program recursive.

8.1.1 Computing Flow Sensitive Side Effects

The side effect properties *MustKill*, *MayKill*, *MustUse*, *MayUse* are computed by assuming *BI* to be empty set.

For computing the *MustKill* and *MayKill*, a simple data flow analysis gathers the variables that are killed on the paths from $Start_r$ to End_r. The sets so computed do not include local variables of r because they are not visible in the caller procedures even if r is called recursively. The data flow value Out_r defines $MayKill_r$ or $MustKill_r$ as the case may be. The data flow equations for computing $MustKill_r$ are given below.

$$\begin{aligned} In_n &= \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcap_{p \in pred(n)} Out_p & \text{otherwise} \end{cases} \\ Out_n &= \begin{cases} In_n \cup MustKill_s & n \text{ is a call to } s \\ In_n \cup Gen_n & \text{otherwise} \end{cases} \\ MustKill_r &= Out_{End_r} \end{aligned} \tag{8.1}$$

The initial values of In_n, Out_n, and $MustKill_s$ are $\top = \mathbb{G}\text{var}$.

For computing $MayKill_r$, $\sqcap$ is $\cup$, and the initial values of In_n, Out_n, and $MayKill_s$ are $\top = \emptyset$.

Example 8.1

The computation of *MustKill* and *MayKill* properties of procedures p and q of our program in Figure 8.1 are shown in Figure 8.2 on the following page. Since procedures p and q are mutually recursive, their data flow values are mutually dependent and require a fixed point computation with $\top$ as the initial value. When procedure p is being analyzed $MustKill_q$ is assumed to be $\{a,b,c,d\}$. This results in $MustKill_p = \{b,c\}$ which is then used in computing $MustKill_q$. The resulting value $MustKill_q = \{a,b,c\}$ is used in the second iteration over p. Although it causes a change in Out_{c_4}, Out_{End_p} does not change. Thus neither $MustKill_p$, not $MustKill_q$ changes. For computing *MayKill*, $\top$ is $\emptyset$. Thus the initial value of $MayKill_q$ is $\emptyset$ resulting in $MayKill_p = \{a,b,c\}$. When this is used to compute $MayKill_q$, the result is $\{a,b,c\}$. However, the new value of $MayKill_q$ does not result in a change in the value of $MayKill_p$.

Observe that c is contained in $MustKill_p$ in spite of the fact that it is computed conditionally. This is because every path from $Start_p$ to End_p must pass through n_3: Even if the execution were to follow edge $Start_p \rightarrow c_4$, the only way to unwind the recursive call to q is to execute the path involving n_3. Since there is only one path through procedure q (with varying depths of recursion), $MustKill_q = MayKill_q$. Also observe that a is contained in $MayKill_p$ but not in $MustKill_p$. □

Procedure	Node	*MustKill*				*MayKill*			
		Iteration #1		Changes in Iteration #2		Iteration #1		Changes in Iteration #2	
		In	*Out*	*In*	*Out*	*In*	*Out*	*In*	*Out*
p	$Start_p$	$\emptyset$	$\{b\}$			$\emptyset$	$\{b\}$		
	n_3	$\{b\}$	$\{b,c\}$			$\{b\}$	$\{b,c\}$		
	c_4	$\{b\}$	$\{a,b,c,d\}$		$\{a,b\}$	$\{b\}$	$\{b\}$		$\{a,b,c\}$
	End_p	$\{b,c\}$	$\{b,c\}$			$\{b,c\}$	$\{b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$
		$MustKill_p = End_p = \{b,c\}$				$MayKill_p = End_p = \{a,b,c\}$			
q	$Start_q$	$\emptyset$	$\{a\}$			$\emptyset$	$\{a\}$		
	c_3	$\{a\}$	$\{a,b,c\}$			$\{a\}$	$\{a,b,c\}$		
	End_q	$\{a,b,c\}$	$\{a,b,c\}$			$\{a,b,c\}$	$\{a,b,c\}$		
		$MustKill_q = End_q = \{a,b,c\}$				$MayKill_q = End_q = \{a,b,c\}$			

FIGURE 8.2
Computing flow sensitive *MustKill* and *MayKill* for the program in Figure 8.1.

The data flow equations for computing *MayUse* have been provided below. Intuitively, $MayUse_r$ contains the variables that are live at the entry of r assuming that no variable is live at the exit of r. Thus except for a call statement, the data flow equations are identical to the data flow equations of liveness analysis. Gen_n contains the set of variables with upwards exposed uses in block n.

$$In_n = \begin{cases} (Out_n - MustKill_t) \cup MayUse_t & n \text{ is a call to } t \\ (Out_n - Kill_n) \cup Gen_n & \text{otherwise} \end{cases} \tag{8.2}$$

$$Out_n = \begin{cases} BI & n \text{ is } End \text{ block} \\ \bigcup_{s \in succ(n)} In_s & \text{otherwise} \end{cases}$$

$$MayUse_r = In_{Start_r}$$

For a call statement, the variables in *MustKill* set of the callee cease to be live whereas the variables in *MayUse* set of the callee become live. For *MustUse*, $\sqcap$ is $\cup$ and *Kill* for a call statement is *MayKill* instead of *MustKill*.

Example 8.2
The data flow analysis for computing *MayUse* and *MustUse* of our example program is provided in Figure 8.3 on the next page. Observe that for computing $MayUse_p$ we use $\top = \emptyset$ as the initial value of $MayUse_q$ whereas for computing $MustUse_p$ we use $\top = \{a,b,c,d\}$ as the initial value of $MustUse_q$. The data flow values for computing *MayUse* do not change in the second iteration whereas the data flow values for computing *MustUse* do. □

Procedure	Node	*MayUse*		*MustUse*			
		Iteration #1		Iteration #1		Changes in Iteration #2	
		Out	*In*	*Out*	*In*	*Out*	*In*
p	End_p	$\emptyset$	$\{c,d\}$	$\emptyset$	$\{c,d\}$		
	n_3	$\{c,d\}$	$\{a,b,d\}$	$\{c,d\}$	$\{a,b,d\}$		
	c_4	$\{c,d\}$	$\{d\}$	$\{c,d\}$	$\{a,b,c,d\}$		$\{d\}$
	$Start_p$	$\{a,b,d\}$	$\{a,d\}$	$\{a,b,d\}$	$\{a,d\}$	$\{d\}$	$\{d\}$
		$MayUse_p = End_p = \{a,d\}$		$MustUse_p = End_p = \{d\}$			
q	End_q	$\emptyset$	$\{a,b\}$	$\emptyset$	$\{a,b\}$		
	c_3	$\{a,b\}$	$\{a,d\}$	$\{a,b\}$	$\{a,d\}$		$\{d\}$
	$Start_q$	$\{a,d\}$	$\{d\}$	$\{a,d\}$	$\{d\}$	$\{d\}$	
		$MayUse_q = End_q = \{d\}$		$MustUse_q = End_q = \{d\}$			

FIGURE 8.3
Computing flow sensitive *MayUse* and *MustUse* for the program in Figure 8.1.

8.1.2 Computing Flow Insensitive Side Effects

Recall that flow insensitive computation accumulates the effect of each block using Equation (7.1). As explained in Figure 7.5 on page 242 and Figure 7.8 on page 249, dependence of data flow values on other data flow values has to be explicitly handled by adding dependence edges. In the general situation, a path in $paths(u)$ could consist of fragments where the dependent parts in the flow functions are $\emptyset$ as illustrated in Figure 8.4 on the following page.

The following lemma shows that if dependent parts are handled explicitly, flow insensitive analysis computes a safe approximation of the corresponding flow sensitive data flow information.

LEMMA 8.1
Consider a path fragment $\rho = (p_0, p_1, \ldots, p_k)$ *along which the dependent parts of flow functions* $f_{p_i \to p_{i+1}}$ *are* $\emptyset$. *Then,*

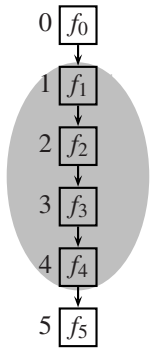
$$\forall \mathsf{x} \in L : \prod_{i=0}^{k} f_i(\mathsf{x}) \sqsubseteq f_\rho(\mathsf{x})$$

where $f_i = f_{p_i \to p_{i+1}}$ *and* $f_\rho = f_k \circ f_{k-1} \circ \ldots \circ f_1 \circ f_0$.

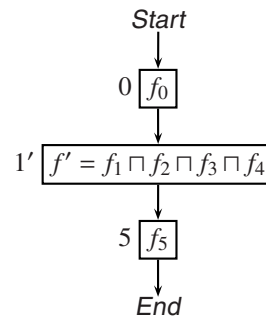
PROOF We prove this by induction on path length.

- *Basis:* Consider path of length of 1. We need to show that

$$\forall \mathsf{x} \in L : f_1(\mathsf{x}) \sqcap f_0(\mathsf{x}) \sqsubseteq f_1(f_0(\mathsf{x}))$$



(a) A path for flow sensitive analysis. Dependent parts in f_1, f_2, f_3, f_4 are $\emptyset$.



(b) Modeling the path for flow insensitive analysis. $In_{5'} \sqsubseteq In_5$.

FIGURE 8.4
Safety of flow insensitive analysis with dependent parts.

Since there are no dependent parts in flow functions, the flow function is separable. Thus we can prove the lemma for independent entities. Further, absence of dependent parts imply that entity functions are constant functions or the identity function; a non-constant non-identity flow function requires dependent part. Thus the proof obligation reduces to

$$\forall \alpha \in \Sigma, \forall\, \widehat{\mathsf{x}} \in \widehat{L} : \widehat{f_0}^{\alpha}(\widehat{\mathsf{x}}) \,\widehat{\sqcap}\, \widehat{f_1}^{\alpha}(\widehat{\mathsf{x}}) \;\widehat{\sqsubseteq}\; \widehat{f_0}^{\alpha}\left(\widehat{f_1}^{\alpha}(\widehat{\mathsf{x}})\right)$$

We consider the following two cases.

- $\widehat{f_0}^{\alpha}$ is the identity function. Then the proof obligation reduces to

$$\forall\, \widehat{\mathsf{x}} \in \widehat{L} : \widehat{\mathsf{x}} \,\widehat{\sqcap}\, \widehat{f_1}^{\alpha}(\widehat{\mathsf{x}}) \;\widehat{\sqsubseteq}\; \widehat{f_1}^{\alpha}(\widehat{\mathsf{x}})$$

  which trivially holds.

- $\widehat{f_0}^{\alpha}$ is some constant function resulting in a particular value $\widehat{\mathsf{y}} \in \widehat{L}$. Then the proof obligation reduces to

$$\forall\, \widehat{\mathsf{x}} \in \widehat{L}_{\alpha} : \widehat{\mathsf{y}} \,\widehat{\sqcap}\, \widehat{f_1}^{\alpha}(\widehat{\mathsf{x}}) \;\widehat{\sqsubseteq}\; \widehat{\mathsf{y}}$$

  which also trivially holds.

- *Inductive step:* Assume that the lemma holds for path of length i. Then, it follows that for $f_{\rho'} = f_i \circ f_{i-1} \circ \ldots \circ f_1 \circ f_0$,

$$\forall \mathsf{x} \in L : \prod_{i=0}^{i} f_i(\mathsf{x}) \;\sqsubseteq\; f_{\rho'}(\mathsf{x})$$

Property	Defining expression	Result	
		Iteration #1	Changes in iteration #2
$MustKill_p$	$Kill_{Start_p} \cap Kill_{n_3} \cap MustKill_q \cap Kill_{End_p}$	$\emptyset$	
$MayKill_p$	$Kill_{Start_p} \cup Kill_{n_3} \cup MayKill_q \cup Kill_{End_p}$	$\{b,c\}$	$\{a,b,c\}$
$MustUse_p$	$Gen_{Start_p} \cap Gen_{n_3} \cap MustUse_q \cap Gen_{End_p}$	$\emptyset$	
$MayUse_p$	$Gen_{Start_p} \cup Gen_{n_3} \cup MayUse_q \cup Gen_{End_p}$	$\{a,b,c,d\}$	
$MustKill_q$	$Kill_{Start_q} \cap MustKill_p \cap Kill_{End_q}$	$\emptyset$	
$MayKill_q$	$Kill_{Start_q} \cup MayKill_p \cup Kill_{End_q}$	$\{a,b,c\}$	
$MustUse_q$	$Gen_{Start_q} \cap MustUse_p \cap Gen_{End_q}$	$\emptyset$	
$MayUse_q$	$Gen_{Start_q} \cup MayUse_p \cup Gen_{End_q}$	$\{a,b,c,d\}$	

FIGURE 8.5
Flow insensitive computation of side effects for the program in Figure 8.1.

We need to show that

$$f_{i+1}(\mathsf{x}) \sqcap \left(\prod_{i=0}^{i} f_i(\mathsf{x})\right) \sqsubseteq f_{i+1}\left(f_{\rho'}(\widehat{\mathsf{x}})\right)$$

Since the flow functions are separable we can prove this independently for different entities by considering constant and identity entity functions $\widehat{f}_{i+1}^{\,\alpha}$ in a manner similar to that in the basis case.

∎

Recall that for flow insensitive analysis, we merge $f_i(BI)$ (Equation 7.1). Since BI is $\emptyset$ the flow functions defined in Equations (8.1) and (8.2) reduce to:

Property	Flow Function	Flow Function with $\mathsf{x} = BI = \emptyset$
$MustKill_r, MayKill_r$	$\mathsf{x} \cup Kill$	$Kill$
$MustUse_r, MayUse_r$	$(\mathsf{x} - Kill) \cup Gen$	Gen

Example 8.3
The flow insensitive computations of side effects for our example program of Figure 8.1 is shown in Figure 8.5. Observe the mutual dependence of the data flow values of procedures p and q due to mutual recursion. We first compute the values for procedure p by using $\top$ values for procedure q. The resulting values for procedure p are then used to compute the values of procedure q. The resulting values for procedure q are different from the initially assumed $\top$ values. However, they do not cause any change in the values of procedure p except for $MayKill_p$. When this changed value is used for recomputing $MayKill_q$, there is no change. □

```
 0. int a,b;
 1. main()
 2. {  int c,d;
 3.    read (a,b,c);
 4.    q(a,b,c,d);      /* Call site c1 */
 5.    p(c,c);          /* Call site c2 */
 6. }
 7. q(int w, int x, int y, int z)
 8. {  int e;
 9.    x = x + 1;
10.    if (x < y)
11.    {  q(x,y,z,e); /* Call site c3 */
12.       p(w,x);     /* Call site c4 */
13.    }
14. }
15. p(int m, int n)
16. {  n=m; }
```

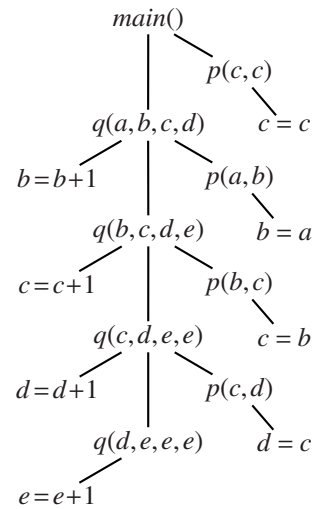


FIGURE 8.6
A C program assuming parameter passing by reference. A possible activation tree shows how variables may be modified in the program.

8.2 Handling the Effects of Parameters

Recall that we have excluded the effects of parameters from our descriptions of analyses by restricting them to global variables only. If the parameter passing mechanism is by value, the basic techniques do not change much except that the data flow information of actual parameters must be propagated as the data flow information of formal parameters. Thus formal parameters can be considered similar to local variables except that *BI* for formal parameters is computed from the calling contexts. Section 9.5 shows a way of modeling the effect of parameters to capture the transfer of data flow information between actual and formal parameters.

In this section we look at aliasing between formal parameters in the presence of parameter passing by reference. In this case the actual parameter and the formal parameter share the same address and hence are aliased. The main difference between this aliasing and the aliasing created by pointer assignments (Section 4.3.2) is that in pointer assignments, both variables involved in an alias are simultaneously visible; in the case of parameters this need not hold always. Thus discovering such aliases requires a different technique.

8.2.1 Defining Aliasing of Parameters

We discover *may* aliases created by call statements only. Further, we restrict the aliases to scalar variables only. Hence, all other statements including the control flow statements are ignored and our analysis is flow insensitive. Observe that there is no way of killing such aliases; they just become invisible when the variables involved go out of scope.

Example 8.4

We consider the program in Figure 8.6 for performing side effects analysis of procedure q. If we assume parameter passing by reference, it is easy to see that q will modify variable b. However, it is clear from the activation tree of the program that q can also modify the local variables of *main* (c and d) and q (e). This happens because the recursive call to q at the call site c_3 passes its formal parameters as actuals in a different order. As a consequence, the formal parameter x gets aliased to y and z in nested recursive calls. Observe that it does not get aliased to w and the global variable a cannot be modified anywhere in the program. The first call to p modifies c whereas the second call to p modifies b, c, and d. □

Our primary goal is to find out aliasing of formal parameters of a procedure. Consider two formal parameters x and y of procedure r. They may be aliased to each other because of any of the following reasons:

- *Direct generation*. There are two ways in which direct aliases are generated:
  - The actual parameters of both x and y are same in some call. They may well be global variables, local variables of the caller, or formal parameters of the caller.
  - In some call, the actual parameter for x (alternatively, y) is a global variable v and the actual parameter of y (alternatively, x) is a formal parameter of the caller and is aliased to v. Observe that a formal parameter of a procedure can never be aliased to a local variable of the same procedure.
- *Indirect generation*. x may be passed as y (or vice-versa) in a call in r in a recursive call sequence.
- *Propagation*. The actual parameters of x and y may be aliased in a caller's body.

We restrict ourselves to languages that do not support nested procedures. In the case of nested procedures, the formal parameters of an outer procedure are visible within the nested procedures and must be treated as global variables within them rather than as formal parameter of the outer procedure.

8.2.2 Formulating Alias Analysis of Parameters

We solve the problem in two steps. In the first step we find out the variables that may be aliased to formal parameters of a procedure along some call chain leading to the procedure. These variables may be global variables and formal parameters of callers. In the second step, we augment this information with the aliasing between formal parameters of the same procedure.

Let the local variables and formal parameters of procedure r be contained in Local_r and Formal_r respectively; we assume that formal parameter names are distinct for each procedure. We define a function ψ to map a formal parameter to the corresponding actual parameter at a call site. Let a call site c_i in procedure s call procedure r. Given $x \in \mathit{Formal}_r$, $\psi(c_i, x) = y$ where $y \in \mathbb{G}\mathsf{var} \cup \mathit{Local}_s \cup \mathit{Formal}_s$.

Our lattice consists of data flow values denoted by $x \overset{c_i}{\Mapsto} y$ where x is a formal parameter of a procedure being called at call site c_i and y is a variable which is represented by x in the called procedure; in the simplest case, y is the actual parameter corresponding to x. In order to compute data flow information inherited by the callee, the data flow information of y must be copied to the data flow information of x in BI of the callee. To compute the data flow information synthesized by the callee, the data flow information of x must be copied to the data flow information of y. In particular, if the callee modifies x, y should be considered to be modified in the caller's body. Similarly, if the callee uses x, y should be considered to be used in the caller's body. Thus the relation $x \overset{c_i}{\Mapsto} y$ is not symmetric; the exact direction of dependence is governed by the intended use of the data flow information.

The relation $x \overset{c_i}{\Mapsto} y$ between x and y becomes symmetric if both x and y are visible within the called procedure. This is possible only when y is a global variable or when y is a formal parameter that encloses the called procedures. In such a situation, a modification in y in the callee is should be considered a modification in x and vice-versa. This situation is more appropriately modeled by considering y as a global variable for the callee rather than as a formal parameter of the caller.

The flow function for a call site c_i in procedure s calling a procedure r is defined as follows:

$$f_{c_i}(\mathsf{x}) = \mathsf{x} \cup \mathit{ConstGen}_{c_i} \cup \mathit{DepGen}_{c_i}(\mathsf{x})$$

$$\mathit{ConstGen}_{c_i} = \{x \overset{c_i}{\Mapsto} y \mid x \in \mathit{Formal}_r,\ y = \psi(c_i, x),\ y \in \mathbb{G}\mathsf{var} \cup \mathit{Formal}_s\}$$

$$\mathit{DepGen}_{c_i}(\mathsf{x}) = \{x \overset{c_i}{\Mapsto} z \mid x \in \mathit{Formal}_r,\ y = \psi(c_i, x),\ y \in \mathit{Formal}_s,\ y \overset{c_i}{\Mapsto} z \in \mathsf{x}\}$$

Observe that $\mathit{ConstGen}_{c_i}$ excludes $y \in \mathit{Local}_r$ and z in DepGen_{c_i} could well be a formal parameter of a caller of s or some other ancestor of s along a call chain reaching c_i.

Let Calls denote all call sites in a program. In the first step, the aliasing information is computed as the MFP solution of the following equation with $\top = \emptyset$:

$$\mathit{PVA} = \bigcup_{c_i \in \mathit{Calls}} f_{c_i}(\mathit{PVA}) \tag{8.3}$$

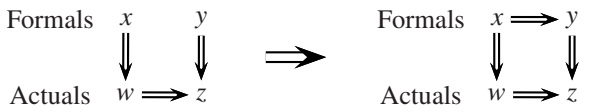


FIGURE 8.7
Aliasing relation between actual parameters should be propagated to the corresponding formal parameters.

where *PVA* is an abbreviation of "Parameters to Variables Aliasing". *PVA* contains the variables to which formal parameters of a function may be aliased. These variables could be global variables, formal parameters of caller procedures, or formal parameters of the same procedure in the case of recursion.

To see why *PVA* contains the indirectly generated aliases of formal parameters in the presence of recursion, consider $x, y \in \mathit{Formal}_r$ for procedure r that is part of a recursive call chain. There must be a sequence of corresponding formal parameters x', y', x'', y'', etc. of the procedures called in the call sequence. If one of these parameters (say y'') is passed as an actual parameter at a different position (say in the place of x''') in the subsequent call, it will result in a pair $x''' \overset{c_j}{\mapsto} y''$ in *PVA*. Due to transitive propagation defined in $\mathit{DepGen}_{c_j}(\mathsf{x})$, we will also have the pair $x''' \overset{c_j}{\mapsto} y$ in *PVA*. When this pair is propagated to r, we will get the pair $x \overset{c_k}{\mapsto} y$ in *PVA*. The pairs x, y'', y, x'' in *PVA* are not meaningful by themselves because they represent formal parameters of different procedures; their use is mainly in detecting and propagating indirectly generated aliases.

The semantics captured by the pair $x \overset{c_k}{\mapsto} y$ in *PVA* when both x and y are in Formal_r requires some explanation. Recall that this relation is not symmetric because it denotes the fact that y is represented by x in a nested call. Since both x and y are formal parameters of the same procedure, this is possible only when an incarnation of y in a call to r is represented by an incarnation of x in a nested recursive call to r. Thus, if we have $x \overset{c_k}{\mapsto} y$ in *PVA* and x is modified in r, we can conclude that y is modified in r. However, if y is modified in r, then we cannot conclude that x is modified in r unless we have $y \overset{c_k}{\mapsto} x$ in *PVA*. Observe that this is consistent with our semantics of $x \overset{c_k}{\mapsto} y$ when x and y are formal parameters of different procedures.

Let $\mathit{VPA}_r(x)$ (abbreviation for "Variables to Parameters Aliasing") denote the set of formal parameters of r that are aliased to variable x. It is defined as:

$$\mathit{VPA}_r(x) = \begin{cases} \{y \mid x \overset{c_i}{\mapsto} y \in \mathit{PVA},\ y \in \mathit{Visible}_r,\} & \text{if } x \in \mathit{Formal}_r \\ \{y \mid y \overset{c_i}{\mapsto} x \in \mathit{PVA},\ y \in \mathit{Formal}_r,\} & \text{if } x \in \mathbb{G}\mathsf{var} \end{cases} \tag{8.4}$$

The meaning of $y \in \mathit{VPA}_r(x)$ is that whenever x is modified in r, y should also be considered to be modified in r; similarly, whenever x is used in r, y should be considered to be used in r. Clearly, $\mathit{VPA}_r(x)$ as defined by Equation (8.4) is not symmetric.

- Computing *PVA*. An element m in the set in a row c_i and column l represents the data flow value $l \overset{c_i}{\mapsto} m$ computed in the corresponding iteration.

Iteration	*PVA* for procedure q					*PVA* for procedure p		
	Call site	w	x	y	z	Call site	m	n
#1	c_1	$\{a\}$	$\{b\}$	$\emptyset$	$\emptyset$	c_2	$\emptyset$	$\emptyset$
	c_3	$\{x\}$	$\{y\}$	$\{z\}$	$\emptyset$	c_4	$\{w\}$	$\{x\}$
#2	c_1	$\{a\}$	$\{b\}$	$\emptyset$	$\emptyset$	c_2	$\emptyset$	$\emptyset$
	c_3	$\{b,x,y\}$	$\{y,z\}$	$\{z\}$	$\emptyset$	c_4	$\{a,w,x\}$	$\{b,x,y\}$
#3	c_1	$\{a\}$	$\{b\}$	$\emptyset$	$\emptyset$	c_2	$\emptyset$	$\emptyset$
	c_3	$\{b,x,y,z\}$	$\{y,z\}$	$\{z\}$	$\emptyset$	c_4	$\{a,b,w,x,y\}$	$\{b,x,y,z\}$

- Computing VPA_r for calls with different set of actual parameters at call site c_2.

Call at c_2	VPA_q						VPA_p			
	w	x	y	z	a	b	m	n	a	b
`p(c,c)`	$\{x,y,z\}$	$\{y,z\}$	$\{z\}$	$\emptyset$	$\{w\}$	$\{w,x\}$	$\{n\}$	$\{m\}$	$\{m\}$	$\{m,n\}$
`p(c,d)`	$\{x,y,z\}$	$\{y,z\}$	$\{z\}$	$\emptyset$	$\{w\}$	$\{w,x\}$	$\emptyset$	$\emptyset$	$\{m\}$	$\{m,n\}$

FIGURE 8.8
Computing aliasing resulting from reference parameters for our example program.

The aliases contained in $VPA_r(x)$ are not complete. What remains is to detect and propagate the directly generated aliases of formal parameters. When x is a formal parameter, we augment $VPA_r(x)$ as shown below:

$$VPA_r(x) = VPA_r(x) \cup \left(\bigcup_{c_i \in CallsTo_r} propVPA_r(c_i, x) \right)$$

$propVPA_r(c_i, x)$ denotes the set of aliases that are propagated to $c_i \in CallsTo_r$: When two aliased formal parameters of a caller of r are passed as actual parameters in a call to r, the corresponding formal parameters of r get aliased; this has been illustrated in Figure 8.7. The identification of directly generated aliases and their propagation is achieved by:

$$propVPA_r(c_i, x) = directVPA_r(c_i, x) \cup$$
$$\{y \mid x \overset{c_j}{\mapsto} w \in PVA,\ y \overset{c_j}{\mapsto} z \in PVA,\ z \in VPA_s(w),$$
$$c_j \in CallsTo_r,\ x \in Formal_r,\ y \in Formal_r\ \}$$

where c_j is a call site in procedure s calling r, and

$$\begin{aligned} \mathit{directVPA}_r(c_i, x) = \{ & y \mid \psi(c_i, x) = \psi(c_i, y),\ x \in \mathit{Formal}_r,\ y \in \mathit{Formal}_r \} \cup \\ & \{ y \mid ((\psi(c_i, x) = v,\ y \overset{c_i}{\mapsto} v \in \mathit{PVA}) \text{ or } (\psi(c_i, y) = v,\ x \overset{c_i}{\mapsto} v \in \mathit{PVA})), \\ & \quad v \in \mathbb{G}\mathsf{var},\ x \in \mathit{Formal}_r,\ y \in \mathit{Formal}_r \} \end{aligned}$$

Observe that the propagation of aliases to callees results in context insensitive aliases because the aliases from all callers are combined. This is similar to the context insensitivity observed in *PVA*.

Example 8.5

The computation of aliases resulting from reference parameters in the program of Figure 8.6 has been shown in Figure 8.8 on the facing page. Beginning with $\top = \emptyset$, we compute successive approximations of *PVA* using Equation (8.3). Observe that the indirect aliases for procedure q capture the fact that w represents x, y, and z in recursive calls and x represents y and z. However, w is not represented by any other variable. What this means is that the assignment to x in procedure q cannot modify w although it can modify y and z and their actual parameters.

We augment this information with aliasing between formal parameters of the same procedure, under two different situations:

- When the call at call site c_2 is `p(c,c)`, and
- when it is `p(c,d)`.

When the call is `p(c,c)`, the formal parameters m and n of procedure p get aliased. Since the data flow information is context insensitive, our analysis assumes that this aliasing holds for all calls to p. If we change the call to `p(c,d)`, m and n are not aliased anymore. □

8.2.3 Augmenting Data Flow Analyses Using Parameter Aliases

Now $\mathit{VPA}_r(x)$ can be used to augment the data flow information computed by other analyses. We illustrate it for computing $\mathit{MayKill}_r$.

We define $\mathit{MayKill}_r$ to consist of two components:

$$\begin{aligned} \mathit{MayKill}_r = & \mathit{Kill}_r \cup \left(\bigcup_{c_j \in \mathit{CallsIn}_r} \mathit{MayKill}_t(c_j) \right) \cup \\ & \left\{ y \mid y \in \mathit{VPA}_r(x),\ x \in \mathit{MayKill}_r \right\} \end{aligned} \tag{8.5}$$

where $\mathit{MayKill}_r$ represents all variables visible in r that are killed by execution of r. They include local and global variables as well as formal parameters of r. This information, augmented with the killing of actual parameters by a call to r at call

Procedure	*Kill*	When call at c_2 is `p(c,c)`			When call at c_2 is `p(c,d)`		
		MayKill	Call specific *MayKill*		*MayKill*	Call specific *MayKill*	
			Call site c_i	*MayKill*(c_i)		Call site c_i	*MayKill*(c_i)
p	$\{n\}$	$\{m,n\}$	c_2	$\{c,c\}$	$\{n\}$	c_2	$\{d\}$
			c_4	$\{w,x\}$		c_4	$\{x\}$
q	$\{x\}$	$\{w,x,y,z\}$	c_1	$\{a,b,c,d\}$	$\{y,z\}$	c_1	$\{b,c,d\}$
			c_3	$\{w,x,y,z\}$		c_3	$\{x,y,z\}$

FIGURE 8.9
Side effects for the example program of Figure 8.6.

site c_i is contained in $MayKill_r(c_i)$ which is defined below in Equation (8.6). $Kill_r$ represents the variables that may be directly killed within r without incorporating the effect of calls made in r. The variables killed by a call to procedure s made at call site c_j in r are contained in $MayKill_s(c_j)$.

From the gross information $MayKill_r$, we extract $MayKill_r^G$ and $MayKill_r^F$ that denote the global variables and formal parameters of r killed by r. They are defined as shown below:

$$MayKill_r^F = MayKill_r \cap Formal_r$$
$$MayKill_r^G = MayKill_r \cap \mathbb{G}\text{var}$$

Now we need to find out the local variables of the caller that may be killed by r. This can happen only through the formal parameters of r. The complete side effect of a call to r made at call site c_i is represented by $MayKill_r(c_i)$ which is defined in terms of $MayKill_r^G$ and $MayKill_r^F$ as shown below:

$$MayKill_r(c_i) = MayKill_r^G \cup \left\{ y \mid \psi(c_i, x) = y,\ x \in MayKill_r^F \right\} \tag{8.6}$$

Observe that the definition of $MayKill_r$ (Equation 8.5) is flow insensitive. It can be made flow sensitive by computing In_i and Out_i along the control flow and using $MayKill_r(c_i)$ as the flow function for call site c_i. However, the aliases contained in $VPA_r(x)$ and PVA remain context and flow insensitive.

Example 8.6

The side effects computed for our example have been shown in Figure 8.9. When the call at c_2 is `p(c,c)`, our analysis concludes that the call at c_4 kills both w and x. Hence it concludes that q can kill the global variable a which has been passed as an actual parameter of w at call site c_1. If we change the call at c_2 to `p(c,d)`, m and n are not aliased. Hence our analysis concludes that the call at c_4 kills only x and not w. As a consequence, a is not contained in the side effect of call at c_1. □

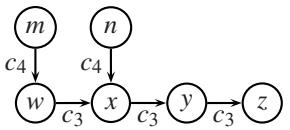


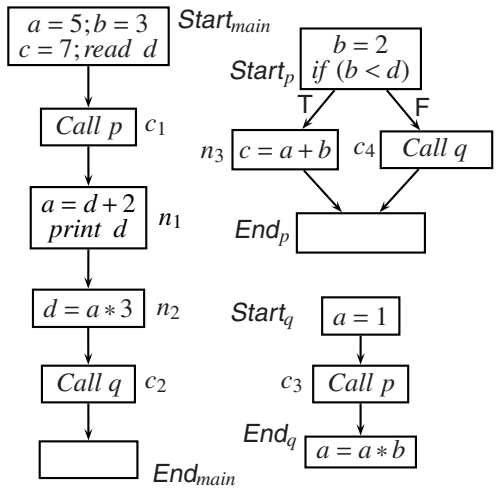
FIGURE 8.10
Parameter binding graph for the program in Figure 8.6.

8.2.4 Efficient Parameter Alias Analysis

The parameter analysis presented in Section 8.2.2 models the required computation instead of designing an efficient algorithm for performing the analysis. The resulting data flow analysis is non-separable and requires a lot of transitive computation that may be redundant. To see this, consider an alias $x \overset{c_i}{\mapsto} y$ computed by Equation (8.3). If the analysis discovers $y \overset{c_j}{\mapsto} z_1$ and $y \overset{c_k}{\mapsto} z_2$, it implies adding the pairs $x \overset{c_i}{\mapsto} z_1$ and $x \overset{c_i}{\mapsto} z_2$. Now if $w \overset{c_l}{\mapsto} x$ is discovered, the analysis computes $w \overset{c_l}{\mapsto} y$, $w \overset{c_l}{\mapsto} z_1$, and $w \overset{c_l}{\mapsto} z_2$. Thus every possible transitive effect of parameters is detected. Observe that it is not necessary to store all these relations. The core relations that need to be stored are $w \overset{c_l}{\mapsto} x$, $x \overset{c_i}{\mapsto} y$, $y \overset{c_j}{\mapsto} z_1$, and $y \overset{c_k}{\mapsto} z_2$; other aliases can be discovered from these relations when required.

A simple way of speeding up the analysis is to identify the dependence of formal parameters on each other and store them in a graph called *parameter binding graph*. For our example program, it has been illustrated in Figure 8.10. An edge $x \rightarrow y$ represents the relation $x \overset{c_i}{\mapsto} y$. This graph directly captures the dependence arising out of non-separability and hence avoid redundant traversals over a call graph. Constructing this graph is efficient because we only need to construct individual edges; computing *PVA* involves identifying all paths in the graph. After this graph is constructed, aliasing with global variables require only propagating them in the graph along the edges in the graph starting from the formal parameter for which the global variable is an actual parameter. Further, indirect aliases are represented by edges between the formal parameters of the same procedure. Thus *PVA* involves mapping formal variables to global variables. There is no need to record mapping between formal variables. This is particularly useful if there are very few recursive procedures; for non-recursive procedures, these mappings are irrelevant.

Observe that a use of parameter binding graph is similar to the use of points-to graph in that both these data structures capture the effect of the dependent part of flow functions and facilitate a flow insensitive computation in a single pass over the underlying control flow structure. The only difference between them is that for points-to analysis the control flow structure is either a CFG or a supergraph whereas for a parameter binding graph, it is a call graph.



Block	Flow function $f_i(\mathsf{x})$
$Start_{main}$	$\mathsf{x} \cup \{a,b,c,d\}$
c_1	$f_p(\mathsf{x})$
n_1	$(\mathsf{x} - \{d\}) \cup \{a\}$
n_2	$(\mathsf{x} - \{a \mid d \notin \mathsf{x}\}) \cup \{d\}$
c_2	$f_q(\mathsf{x})$
End_{main}	x
$Start_p$	$(\mathsf{x} - \{d\}) \cup \{b\}$
n_3	$(\mathsf{x} - \{a,b \mid c \notin \mathsf{x}\}) \cup \{c\}$
c_4	$f_q(\mathsf{x})$
End_p	x
$Start_q$	$\mathsf{x} \cup \{a\}$
c_3	$f_p(\mathsf{x})$
End_q	$\mathsf{x} - \{b \mid a \notin \mathsf{x}\}$

FIGURE 8.11
Example program for interprocedural faint variables analysis. This is a modified version of the program in Figure 8.1.

8.3 Whole Program Analysis

In the previous section, we constructed summary flow functions for specific analyses. In this section we present the general method of constructing summary flow functions. We consider flow sensitive methods; the flow insensitive versions can be devised along the lines described earlier.

We use liveness analysis and faint variables analysis to explain the method. For liveness analysis, we use the program of Section 8.1. Since it does not have much scope for faint variables analysis, we use the program in Figure 8.11.

8.3.1 Lattice of Flow Functions

Defining a data flow analysis requires setting up a lattice of the values that are to be computed by the analysis. This greatly simplifies reasoning about the analysis. The same approach can be used to define analyses to construct summary flow functions. The main difference is that the data flow values computed by other analyses we have seen so far represent certain semantics of the entities appearing in the program. The data flow values for the analysis that constructs summary flow functions are flow functions that compute the data flow values desired in the end.

When we view the set of flow function F as a lattice, we define the partial order relation over flow functions in terms of the partial order relation between the values

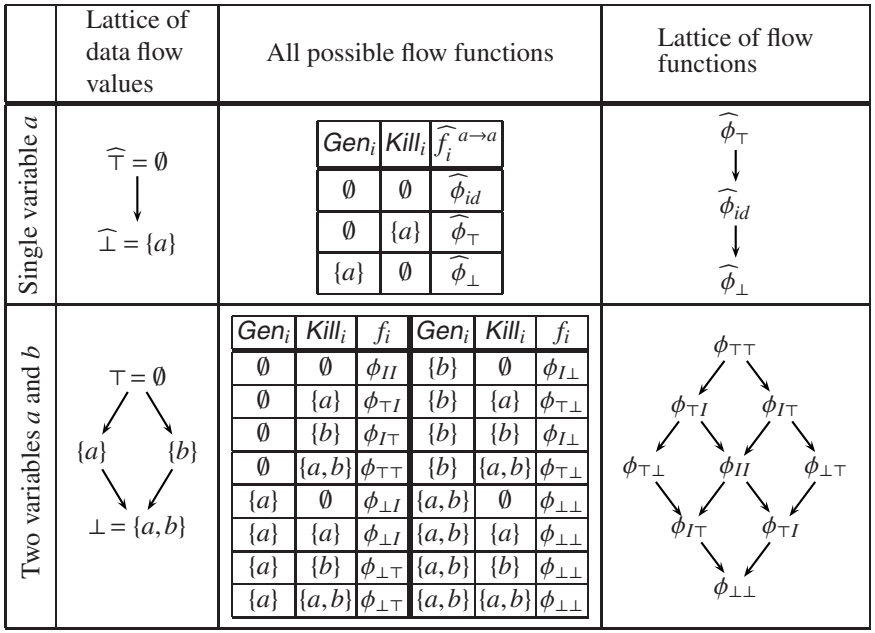


FIGURE 8.12
Example of lattices of functions for live variables analysis. ϕ_{xy} indicates that the component flow function for variable a is x and that for variable b is y. The possible values for x and y are: I for $\widehat{\phi}_{id}$, $\top$ for $\widehat{\phi}_{\top}$, and $\bot$ for $\widehat{\phi}_{\bot}$.

computed by the functions:

$$\forall f_i, f_j \in F:\ f_i \sqsubseteq_f f_j \ \Leftrightarrow\ \forall \mathsf{x} \in L:\ f_i(\mathsf{x}) \sqsubseteq f_j(\mathsf{x})$$

The function composition and confluence operations required for constructing flow functions are:

$$\forall f_i, f_j, f_k \in F:\ f_k = f_i \sqcap_f f_j \ \Leftrightarrow\ \forall \mathsf{x} \in L:\ f_k(\mathsf{x}) = f_i(\mathsf{x}) \sqcap f_j(\mathsf{x})$$
$$\forall f_i, f_j, f_k \in F:\ f_k = f_i \circ f_j \ \Leftrightarrow\ \forall \mathsf{x} \in L:\ f_k(\mathsf{x}) = f_i(f_j(\mathsf{x}))$$
$$\forall f \in F:\ f \sqcap_f \phi_{\top} = f$$
$$\forall f \in F:\ f \sqcap_f \phi_{\bot} = \phi_{\bot}$$

Recall that $\phi_{\top}$ and $\phi_{\bot}$ are constant functions.

8.3.2 Reducing Function Compositions and Confluences

Constructing flow functions requires reducing expressions involving compositions and confluences of flow functions to a canonical form. This is different from function applications to actual data flow values. As observed in Section 7.6.2, this is easy

to do when flow functions have only constant parts. However, when they have dependent parts also, systematic reductions can be devised only when the flow functions satisfy some additional requirements. In this section, we show how the function compositions and confluences can be reduced and characterize the class of flow functions for which this can be done.

Function compositions and confluences for bit vector frameworks

For bit vector frameworks the flow function $f(\mathsf{x}) = (\mathsf{x} - \mathit{Kill}) \cup \mathit{Gen}$ does not have dependent parts. To see how function composition can be reduced, let $f_2 \circ f_1 = f_3$. Then we wish to compute Kill_3 and Gen_3. It is easy to see that

$$\begin{aligned} f_3(\mathsf{x}) = f_2(f_1(\mathsf{x})) &= f_2((\mathsf{x} - \mathit{Kill}_1) \cup \mathit{Gen}_1) \\ &= \Big(((\mathsf{x} - \mathit{Kill}_1) \cup \mathit{Gen}_1) - \mathit{Kill}_2\Big) \cup \mathit{Gen}_2 \\ &= (\mathsf{x} - (\mathit{Kill}_1 \cup \mathit{Kill}_2)) \cup (\mathit{Gen}_1 - \mathit{Kill}_2) \cup \mathit{Gen}_2 \end{aligned}$$

Hence,

$$\mathit{Kill}_3 = \mathit{Kill}_1 \cup \mathit{Kill}_2 \tag{8.7}$$

$$\mathit{Gen}_3 = (\mathit{Gen}_1 - \mathit{Kill}_2) \cup \mathit{Gen}_2 \tag{8.8}$$

To see how function confluences can be reduced, let $f_2 \sqcap f_1 = f_3$. First we consider the case when $\sqcap$ is $\cup$.

$$\begin{aligned} f_3(\mathsf{x}) = f_2(\mathsf{x}) \cup f_1(\mathsf{x}) &= ((\mathsf{x} - \mathit{Kill}_2) \cup \mathit{Gen}_2) \cup ((\mathsf{x} - \mathit{Kill}_1) \cup \mathit{Gen}_1) \\ &= (\mathsf{x} - (\mathit{Kill}_1 \cap \mathit{Kill}_2)) \cup (\mathit{Gen}_1 \cup \mathit{Gen}_2) \end{aligned}$$

implying that

$$\mathit{Kill}_3 = \mathit{Kill}_1 \cap \mathit{Kill}_2 \tag{8.9}$$

$$\mathit{Gen}_3 = \mathit{Gen}_1 \cup \mathit{Gen}_2 \tag{8.10}$$

When $\sqcap$ is $\cap$,

$$\begin{aligned} f_3(\mathsf{x}) = f_2(\mathsf{x}) \cap f_1(\mathsf{x}) &= ((\mathsf{x} - \mathit{Kill}_2) \cup \mathit{Gen}_2) \cap ((\mathsf{x} - \mathit{Kill}_1) \cup \mathit{Gen}_1) \\ &= (\mathsf{x} - (\mathit{Kill}_1 \cup \mathit{Kill}_2)) \cup (\mathit{Gen}_1 \cap \mathit{Gen}_2) \end{aligned}$$

Kill and *Gen* are defined by

$$\mathit{Kill}_3 = \mathit{Kill}_1 \cup \mathit{Kill}_2 \tag{8.11}$$

$$\mathit{Gen}_3 = \mathit{Gen}_1 \cap \mathit{Gen}_2 \tag{8.12}$$

Thus the reduction of function composition and confluences for bit vector frameworks can be defined in terms of $\cup$ and $\cap$ alone.

Function compositions and confluences for general frameworks

When the flow functions have dependent parts also the dependence of data flow values must also be brought in. This dependence may be dependence of values of the same entity or across different entities. Using the notation from Definition 4.1,

$$f_i = \langle \widehat{f_i}^{\alpha}, \widehat{f_i}^{\beta}, \cdots, \widehat{f_i}^{\eta} \rangle$$

where $\widehat{f_i}^{\alpha} : L \mapsto \widehat{L}$ is a component function computing the data flow value of entity α. Given $f_2 \circ f_1 = f_3$, we need to construct $\widehat{f_3}^{\alpha}$ for all α. We know that:

$$\widehat{f_2}^{\alpha} = \bigsqcap_{\beta \in \Sigma} \left(\overline{f}_2^{\beta \to \alpha} \right)$$

where $\overline{f}_2^{\beta \to \alpha}$ is a primitive entity function computing the data flow value of α from β. In order to compute $\widehat{f_3}^{\alpha}$, we will need to compose $\overline{f}_2^{\beta \to \alpha}$ with every component function $\widehat{f_1}^{\beta}$ which is defined as:

$$\widehat{f_1}^{\beta} = \bigsqcap_{\gamma \in \Sigma} \left(\overline{f}_1^{\gamma \to \beta} \right)$$

In other words, we need to compose the *pef*s of various components function. This gives us the first restriction on the flow functions for systematic reduction of compositions: It is not possible to compose flow functions unless the component functions can be defined in terms of *pef*s. This rules out full constant propagation and points-to analysis.

For primary framework $\widehat{f_3}^{\alpha}$ can be constructed as follows:

$$\begin{aligned} \widehat{f_3}^{\alpha} &= \bigsqcap_{\beta \in \Sigma} \left(\overline{f}_2^{\beta \to \alpha} \circ \widehat{f_1}^{\beta} \right) \\ &= \bigsqcap_{\beta \in \Sigma} \left(\overline{f}_2^{\beta \to \alpha} \circ \left(\bigsqcap_{\gamma \in \Sigma} \overline{f}_1^{\gamma \to \beta} \right) \right) \end{aligned}$$

The need to reduce this suggests the second restriction: The *pef*s must be distributive. Although, it is not difficult to construct non-distributive *pef*s of the form $\widehat{L} \mapsto \widehat{L}$, most of the known such *pef*s in practical data flow frameworks are indeed distributive. Thus $\widehat{f_3}^{\alpha}$ reduces to

$$\widehat{f_3}^{\alpha} = \bigsqcap_{\beta,\gamma \in \Sigma} \left(\overline{f}_2^{\beta \to \alpha} \circ \overline{f}_1^{\gamma \to \beta} \right) \tag{8.13}$$

Function confluences are relatively easy to define. Given $f_3 = f_1 \sqcap f_2$,

$$\widehat{f_3}^{\alpha} = \widehat{f_1}^{\alpha} \sqcap \widehat{f_2}^{\alpha} \tag{8.14}$$

$$= \bigsqcap_{\beta \in \Sigma} \left(\overline{f}_1^{\beta \to \alpha} \sqcap \overline{f}_2^{\beta \to \alpha} \right) \tag{8.15}$$

due to distributivity.

When we restrict ourselves to primary frameworks, compositions can be reduced using the following identities. We use the superscript $\alpha \to \beta$ to show the dependency of entity β on the entity α only when required.

$$\forall \widehat{f}, \forall \widehat{\mathsf{z}} \in \widehat{L}: \quad \widehat{\phi}_{\mathsf{z}} \circ \widehat{f} = \widehat{\phi}_{\mathsf{z}}$$

$$\forall \alpha, \beta \in \Sigma: \widehat{\phi}_{id}^{\beta \to \gamma} \circ \widehat{\phi}_{id}^{\alpha \to \beta} = \widehat{\phi}_{id}^{\alpha \to \gamma}$$

$$\forall \alpha, \beta \in \Sigma: \quad \widehat{\phi}_{id}^{\beta \to \gamma} \circ \widehat{\phi}_{\mathsf{z}} = \widehat{\phi}_{\mathsf{z}}$$

$$\forall \alpha, \beta \in \Sigma, \forall a, b \in \mathbb{C}\mathsf{onst}: \widehat{\phi}_{id}^{\beta \to \gamma} \circ \widehat{\phi}_{ab}^{\alpha \to \beta} = \widehat{\phi}_{ab}^{\alpha \to \gamma}$$

$$\forall a, b \in \mathbb{C}\mathsf{onst}, \forall \widehat{\mathsf{z}} \in \widehat{L}: \quad \widehat{\phi}_{ab}^{\alpha \to \beta} \circ \widehat{\phi}_{\mathsf{z}} = \widehat{\phi}_{\mathsf{y}} \quad \text{where } \widehat{\mathsf{y}} = a \times \widehat{\mathsf{z}} + b$$

$$\forall a, b, c, d \in \mathbb{C}\mathsf{onst}: \widehat{\phi}_{ab}^{\beta \to \gamma} \circ \widehat{\phi}_{cd}^{\alpha \to \beta} = \widehat{\phi}_{mn}^{\alpha \to \gamma} \quad \text{where } m = a \times c,\ n = a \times d + b$$

Thus all compositions of *pef*s can be reduced to a single *pef*. In some cases, function confluences can also be reduced:

$$\forall \widehat{f}: \quad \widehat{f} \sqcap \widehat{\phi}_{\top} = \widehat{f}$$

$$\forall \widehat{f}: \quad \widehat{f} \sqcap \widehat{\phi}_{\bot} = \widehat{\phi}_{\bot}$$

$$\forall \widehat{\mathsf{x}}, \widehat{\mathsf{y}} \in \widehat{L}: \quad \widehat{\phi}_{\mathsf{x}} \sqcap \widehat{\phi}_{\mathsf{y}} = \widehat{\phi}_{\mathsf{z}} \quad \text{where } \widehat{\mathsf{z}} = \widehat{\mathsf{x}} \sqcap \widehat{\mathsf{y}}$$

$$\forall a, b, c, d \in \mathbb{C}\mathsf{onst}, a \neq c, b \neq d: \widehat{\phi}_{ab} \sqcap \widehat{\phi}_{cd} = \widehat{\phi}_{\bot}$$

$$\forall a, b \in \mathbb{C}\mathsf{onst}, \widehat{\mathsf{z}} \neq \widehat{\top}: \quad \widehat{\phi}_{ab} \sqcap \widehat{\phi}_{\mathsf{z}} = \widehat{\phi}_{\bot}$$

$$\forall a, b \in \mathbb{C}\mathsf{onst}, a \neq 1, b \neq 0: \widehat{\phi}_{ab} \sqcap \widehat{\phi}_{id} = \widehat{\phi}_{\bot}$$

Note that $\widehat{\phi}_{id}^{\alpha \to \beta} \sqcap \widehat{\phi}_{id}^{\beta \to \gamma}$ cannot be reduced any further.

Recall that in the case of bit vector frameworks,

$$\forall \alpha \neq \beta: \widehat{\phi}_{id}^{\alpha \to \beta} = \widehat{\phi}_{\top}$$

Hence every component function $\widehat{f}^{\,\alpha}$ in bit vector frameworks is guaranteed to be reduced to one of the following three functions: $\widehat{\phi}_{\top}$, $\widehat{\phi}_{\bot}$, and $\widehat{\phi}_{id}^{\alpha \to \alpha}$.

8.3.3 Constructing Summary Flow Functions

Having defined the reductions involving function compositions and confluences, it is now possible to construct summary flow functions by traversing the CFG. Let $\Phi_v^r : L \mapsto L$ denote the summary flow function associated with program point v in procedure r. It represents the effect of all paths from *Entry*(*Start*$_r$) to v and from v to *Exit*(*End*$_r$). If appropriate *BI* is computed for procedure r, applying Φ_v^r to it results in the desired data flow information associated with v.

Φ_v^r is computed from Φ_u^r of $u \in \mathsf{neighbours}(v)$ as defined below:

$$\Phi_v^r = \bigsqcap_{u \in \mathsf{neighbours}(v)} f_{u \to v} \circ \Phi_u^r$$

Block	Flow functions $f_i(\mathsf{x})$ in terms of *pef*s															
	*pef*s computing faintness of a from a,b,c,d				*pef*s computing faintness of b from a,b,c,d				*pef*s computing faintness of c from a,b,c,d				*pef*s computing faintness of d from a,b,c,d			
	a	b	c	d	a	b	c	d	a	b	c	d	a	b	c	d
$Start_m$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$
n_1	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\bot$	$\widehat{\phi}_\bot$	$\widehat{\phi}_\bot$	$\widehat{\phi}_\bot$
n_2	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$
End_m	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$
$Start_p$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\bot$	$\widehat{\phi}_\bot$	$\widehat{\phi}_\bot$	$\widehat{\phi}_\bot$
n_3	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$
End_p	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$
$Start_q$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$
End_q	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$

FIGURE 8.13
Flow functions for faint variables analysis of the program in Figure 8.11 expressed in terms of *pef*s.

where the flow function at the entry point is ϕ_{id} and does not change any further. More specifically, $\Phi_w^r = \phi_{id}$ and $\forall u : f_{u\to w} = \phi_\top$; such that w is $Entry(Start_r)$ for forward flows and $Exit(End_r)$ for backward flows. Φ_v^r is iteratively computed by taking the initial value as $\phi_\top$.

If edge $u \to v$ represents a basic block calls procedure s, $f_{u\to v}$ is replaced by the summary flow function for procedure s. It is defined by Φ_w^s where w is chosen based on the following criterion:

- If $f_{u\to v}$ is a forward flow function mapping the data flow information before the call to the data flow information after the call, w is $Exit(End_s)$.
- If $f_{u\to v}$ is a backward flow function mapping the data flow information after the call to the data flow information before the call, w is $Entry(Start_s)$.

The termination of construction of summary flow functions depends on the nature of component flow functions and the structure of lattice of data flow values. Since we require the *pef*s to be distributive and closed under composition, each component flow function $\widehat{f}^{\,\alpha}$ constituting Φ_v^p can be reduced to the following canonical form

$$\widehat{f}^{\,\alpha} = \bigsqcap_{i\geq 0, j\geq 0} (\phi_i^0 \circ \phi_i^1 \circ \ldots \circ \phi_i^j) \tag{8.16}$$

where ϕ_i^j could be any *pef* in the framework.

Procedure	Flow Function	Defining Expression	Iteration #1		Changes in iteration #2	
			Gen	*Kill*	*Gen*	*Kill*
p	$\Phi^p_{End_p}$	f_{End_p}	$\{c,d\}$	$\emptyset$		
	$\Phi^p_{n_3}$	$f_{n_3} \circ \Phi^p_{End_p}$	$\{a,b,d\}$	$\{c\}$		
	$\Phi^p_{c_4}$	$f_q \circ \Phi^p_{End_p}$	$\emptyset$	$\{a,b,c,d\}$	$\{d\}$	$\{a,b,c\}$
	$\Phi^p_{Start_p}$	$f_{Start_p} \circ (\Phi^p_{n_3} \sqcap \Phi^p_{c_4})$	$\{a,d\}$	$\{b,c\}$		
	f_p	$\Phi^p_{Start_p}$	$\{a,d\}$	$\{b,c\}$		
q	$\Phi^q_{End_q}$	f_{End_q}	$\{a,b\}$	$\{a\}$		
	$\Phi^q_{c_3}$	$f_p \circ \Phi^q_{End_q}$	$\{a,d\}$	$\{a,b,c\}$		
	$\Phi^q_{Start_q}$	$f_{Start_p} \circ \Phi^q_{c_3}$	$\{d\}$	$\{a,b,c\}$		
	f_q	$\Phi^q_{Start_q}$	$\{d\}$	$\{a,b,c\}$		

FIGURE 8.14
Summary flow functions of procedures p and q required for interprocedural liveness analysis of the program in Figure 8.1 on page 260. The flow functions compute the value at the entry of the blocks.

In order to guarantee the termination of construction of $\widehat{f}^{\alpha}$, it should be possible to bound both the number of terms as well as the size of each term in any expression of the form in Equation (8.16). Bounding the size of each term requires that it should be possible to reduce every unbounded sequence of compositions by a bounded sequence. Bounding the number of terms requires that an infinite meet must be equivalent to the meet of a finite number of terms.

For the primary frameworks, the sequence of compositions always reduces to a single *pef*. Note that this does not bound the number possible *pef*s. For example, consider the *pef* $\widehat{\phi}_{11}$ in linear constant propagation. It increments the value of its argument by 1. If we compose k such *pef*s, the resulting *pef* is $\widehat{\phi}_{1k}$ and the length of the sequence of compositions is bounded by 1 for all k. However, there is an unbounded number of $\widehat{\phi}_{1k}$; in particular, one for each k. Thus bounding the size of each term does not automatically bound the number of terms in the canonical form.

Thus it becomes important to ensure that the confluence of the terms in an expression of the form in Equation (8.16) can be reduced to a canonical form. This is possible because of the following reason. In the worst case, each term in the canonical form could compute a distinct value in L. Although, the number of such values may not be finite, we restrict our analysis to those frameworks in which every descending chain in L is finite. Thus every descending chain ending on a distinct i in the canonical form can be represented by a finite chain. All that remains is to identify

Flow function	*pef*s computing faintness of a from a,b,c,d				*pef*s computing faintness of b from a,b,c,d				*pef*s computing faintness of c from a,b,c,d				*pef*s computing faintness of d from a,b,c,d			
	a	b	c	d	a	b	c	d	a	b	c	d	a	b	c	d
$\Phi^p_{End_p}$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$
$\Phi^p_{n_3}$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$
$\Phi^p_{c_4}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$
$\Phi^p_{Start_p}$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\bot$	$\widehat{\phi}_\bot$	$\widehat{\phi}_\bot$	$\widehat{\phi}_\bot$
$\Phi^q_{End_q}$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$
$\Phi^q_{c_3}$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_{id}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\bot$	$\widehat{\phi}_\bot$	$\widehat{\phi}_\bot$	$\widehat{\phi}_\bot$
$\Phi^q_{Start_q}$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\top$	$\widehat{\phi}_\bot$

FIGURE 8.15
First iteration of constructing summary flow functions of procedure p and q for interprocedural faint variables analysis of the program in Figure 8.11 on page 274. The flow functions compute the value at the entry of the blocks. Highlighted entries show the *pef*s that differ from the corresponding *pef*s in the local flow function $f_i(\mathbf{x})$ provided in Figure 8.13 on page 279.

this when summary flow functions are being constructed.

If we examine the primary *pef*s, the only *pef* that may give rise an infinite number of terms in the canonical form is $\widehat{\phi}_{ab}$. Fortunately, its confluence with every flow function can be reduced due to the structure of $\widehat{L}$ in constant propagation. All other *pef*s are guaranteed to be finite in number. Hence the canonical form is always bounded and the construction of summary flow functions follows for primary frameworks.

Example 8.7

Figure 8.14 on the preceding page provides the summary flow functions for our example program. We first analyze procedure p. We compute summary flow functions associated with *Entry*(n) of each block n; the flow function associated with *Exit*(n) is left implicit. Each flow function is constructed by computing *Kill* and *Gen* sets for function composition using Equations (8.7) and (8.8). The confluence required in computing $\Phi^p_{Start_p}$ uses Equations (8.9) and (8.10) to compute *Kill* and *Gen* sets. The analysis initially uses $\phi_\top(\mathbf{x}) = (\mathbf{x} - \{a,b,c,d\})$ for f_q while computing $\Phi^p_{c_4}$. Hence the *Gen* and *Kill* sets of $\Phi^p_{c_4}$ are approximate in the first iteration. Using these sets, f_p is computed and is used in analyzing procedure q. The resulting f_q is different from $\phi_\top$. This causes change in $\Phi^p_{c_4}$ in the second iteration. However, this change does not affect Φ^p_{Start}, and hence

f_p remains same implying that f_q computed in the first iteration does not change.

It is not surprising that Kill_p and Kill_q computed above are identical to $\mathit{MustKill}_p$ and $\mathit{MustKill}_q$ computed in Figure 8.2 on page 262. Similarly, Gen_p and Gen_q computed here are identical to MayUse_p and MayUse_q computed in Figure 8.2 on page 262. □

Example 8.8

Figure 8.15 on the preceding page shows the first iteration in constructing summary flow functions for faint variables analysis. Since the description of the functions is very verbose, the relevant entries have been highlighted. While analyzing procedure p, the summary flow function for c_4 is assumed to be $\phi_\top$. When $\Phi^p_{\mathit{Start}_p}$ is computed, we discover that the faintness of a now depends on c also. This is a result of the composition $\overline{f}^{a\to a}_{\mathit{Start}_p} \circ \overline{f}^{c\to a}_{n_3}$. When this is merged with the earlier $\mathit{pef}\ \overline{f}^{c\to a}_{\mathit{Start}_p}$ which was $\widehat{\phi}_\top$, it becomes $\widehat{\phi}_{id}$. When procedure q is analyzed, the flow function for c_3 is f_p which is the same as $\Phi^p_{\mathit{Start}_p}$. The summary flow function as $\Phi^q_{\mathit{Start}_q}$ so computed represents the fact that b and c are faint at the entry of Start_q and d is not faint. Observe that this is due to the effect of the called procedure p. The resulting f_q is different from the earlier $\phi_\top$.

When the new value of f_q is used in the second iteration, it changes the $\mathit{pef}\ \overline{f}^{d\to d}_{c_3}$ from $\widehat{\phi}_\top$ to $\widehat{\phi}_\bot$ suggesting that d is not faint at the entry of c_4. However, this does not cause any change in $\Phi^p_{\mathit{Start}_p}$ and the process of constructing summary flow functions terminates. □

8.3.4 Computing Data Flow Information

Φ^r_u represents the effect of the paths from $\mathit{Entry}(\mathit{Start}_r)$ to u for forward problems and from u to $\mathit{Exit}(\mathit{End}_r)$ for backward problems. This effect includes the intraprocedural data flow information as well as synthesized data flow information resulting from the calls along these paths. Thus the data flow information associated with u can be computed by the function application $\Phi^r_u(\mathit{BI}_r)$ where BI represents the data flow information inherited by r from its callers.

Let $\mathit{CallsTo}_r$ denote the set of callers of r. These are simply the predecessors of r in the call graph. Let $\mathit{CallsTo}_r(s)$ denote the call sites calling r from procedure s. Given BI_{main}, the data flow information associated with program point u in procedure r, denoted x^r_u, is computed as follows:

$$\mathit{BI}_r = \bigsqcap_{\substack{s\in \mathit{CallsTo}_r \\ c_i\in \mathit{CallsTo}_r(s)}} \Phi^r_{c_i}(\mathit{BI}_s) \tag{8.17}$$

$$\mathsf{x}^r_u = \Phi^r_u(\mathit{BI}_r) \tag{8.18}$$

Procedure	BI	Data flow variable	Summary flow function		Data flow value
			Name	Definition	
main	$\emptyset$	In_{End_m}	$\Phi^m_{End_m}$	$BI_m \cup \{a,c\}$	$\{a,c\}$
		In_{c_2}	$\Phi^m_{c_2}$	$(BI_m - \{a,b,c\}) \cup \{d\}$	$\{d\}$
		In_{n_2}	$\Phi^m_{n_2}$	$(BI_m - \{a,b,c,d\}) \cup \{a,b\}$	$\{a,b\}$
		In_{n_1}	$\Phi^m_{n_1}$	$(BI_m - \{a,b,c,d\}) \cup \{a,b,c,d\}$	$\{a,b,c,d\}$
		In_{c_1}	$\Phi^m_{c_1}$	$(BI_m - \{a,b,c,d\}) \cup \{a,d\}$	$\{a,d\}$
		In_{Start_m}	$\Phi^m_{Start_m}$	$BI_m - \{a,b,c,d\}$	$\emptyset$
p	$\{a,b,c,d\}$	In_{End_p}	$\Phi^p_{End_p}$	$BI_p \cup \{c,d\}$	$\{a,b,c,d\}$
		In_{n_3}	$\Phi^p_{n_3}$	$(BI_p - \{c\}) \cup \{a,b,d\}$	$\{a,b,d\}$
		In_{c_4}	$\Phi^p_{c_4}$	$(BI_p - \{a,b,c\}) \cup \{d\}$	$\{d\}$
		In_{Start_p}	$\Phi^p_{Start_p}$	$(BI_p - \{b,c\}) \cup \{a,d\}$	$\{a,d\}$
q	$\{a,b,c,d\}$	In_{End_q}	$\Phi^q_{End_q}$	$(BI_q - \{a\}) \cup \{a,b\}$	$\{a,b,c,d\}$
		In_{c_3}	$\Phi^q_{c_3}$	$(BI_q - \{a,b,c\}) \cup \{a,d\}$	$\{a,d\}$
		In_{Start_q}	$\Phi^q_{Start_q}$	$(BI_q - \{a,b,c\}) \cup \{d\}$	$\{d\}$

FIGURE 8.16
Data flow information computed by interprocedural liveness analysis of the program in Figure 8.1 on page 260 using the summary flow functions defined in Figure 8.14 on page 280.

The initial value of BI_s is assumed to be $\top$; it is assumed that appropriate boundary point is chosen depending on the flows i.e., $Entry(Start_p)$ for forward flows and $Exit(End_p)$ for backward flows.

Example 8.9
Figure 8.16 shows the result of interprocedural liveness analysis of the program in Figure 8.1 on page 260 using the summary flow functions defined in Figure 8.14 on page 280. For simplicity, we show the information associated with block entries only. The liveness information of the *main* procedure (abbreviated by m) can be computed in a single iteration from $BI = \emptyset$. For live variables analysis

$$BI_p = In_{n_1} \cup In_{End_q} = In_{n_1} \cup \Phi^q_{End_q}(BI_q)$$
$$BI_q = In_{End_m} \cup In_{End_p} = In_{End_m} \cup \Phi^p_{End_p}(BI_p)$$

Procedure	BI	Data flow variable	Summary flow function		Data flow value
			Name	Definition	
main	$\{a,b,c,d\}$	In_{Start_m}	$\Phi^m_{Start_m}$	$BI_m \cup \{a,b,c,d\}$	$\{a,b,c,d\}$
		In_{c_1}	$\Phi^m_{c_1}$	$(BI_m - \{d\}) \cup \{b,c\}$	$\{a,b,c\}$
		In_{n_1}	$\Phi^m_{n_1}$	$(BI_m - \{d\}) \cup \{a,c\}$	$\{a,b,c\}$
		In_{n_2}	$\Phi^m_{n_2}$	$(BI_m - \{a\}) \cup \{c,d\}$	$\{b,c,d\}$
		In_{c_2}	$\Phi^m_{c_2}$	$(BI_m - \{d\}) \cup \{a,b,c\}$	$\{a,b,c\}$
		In_{End_m}	$\Phi^m_{End_m}$	BI_m	$\{a,b,c,d\}$
p	$\{a,b,c\}$	In_{Start_p}	$\Phi^p_{Start_p}$	$(BI_p - (\{d\} \cup \{a \mid c \notin BI_p\})) \cup \{b,c\}$	$\{a,b,c\}$
		In_{n_3}	$\Phi^p_{n_3}$	$(BI_p - \{a,b \mid c \notin BI_p\}) \cup \{c\}$	$\{a,b,c\}$
		In_{c_4}	$\Phi^p_{c_4}$	$(BI_p - \{d\}) \cup \{a,b,c\}$	$\{a,b,c\}$
		In_{End_p}	$\Phi^p_{End_p}$	BI_p	$\{a,b,c\}$
q	$\{a,b,c\}$	In_{Start_q}	$\Phi^q_{Start_q}$	$(BI_q - \{d\}) \cup \{a,b,c\}$	$\{a,b,c\}$
		In_{c_3}	$\Phi^q_{c_3}$	$(BI_q - (\{d\} \cup \{a \mid c \notin BI_q\})) \cup \{b,c\}$	$\{a,b,c\}$
		In_{End_q}	$\Phi^q_{End_q}$	$BI_q - \{b \mid a \notin BI_q\}$	$\{a,b,c\}$

FIGURE 8.17
Interprocedural faint variables analysis for the program in Figure 8.11 on page 274 using the summary flow functions constructed in Example 8.8.

Thus BI_p and BI_q are mutually dependent on each other. Since In_{n_1} is $\{a,b,c,d\}$ which is the $\perp$ element of the lattice, BI_p cannot change any further. From this, In_{End_p} is computed which turns out to be $\{a,b,c,d\}$. Thus BI_q is also $\{a,b,c,d\}$ and does not change any further.

Our analysis shows that Out_{Start_m} contains only a and d. Thus the assignments to b and c in $Start_m$ are redundant and can be eliminated. Observe that although c is used in End_p, it is found to be dead at the entry of c_4. This is because the recursion ending path must pass through block n_3 before the execution reaches End_p from c_4. Due to the assignment in n_3, c is not live at the entry of c_4. □

Example 8.10

Figure 8.17 shows the result of interprocedural faint variables analysis of the program in Figure 8.11 on page 274 using the summary flow functions computed in Example 8.8. In faint variables analysis, BI for *main* is $\{a,b,c,d\}$

because every variable is faint at the end of the program. The data flow information in *main* can be computed in a single iteration. BI for p and q is defined by:

$$BI_p = In_{n_1} \cap In_{End_q} = In_{n_1} \cap \Phi^q_{End_q}(BI_q)$$
$$BI_q = In_{End_m} \cap In_{End_p} = In_{End_m} \cap \Phi^p_{End_p}(BI_p)$$

For computing BI_p, In_{n_1} is $\{a,b,c\}$ and $\Phi^q_{End_q}(BI_q)$ is assumed to be $\top$ which is $\{a,b,c,d\}$ for faint variables analysis. Thus $BI_p = \{a,b,c\}$ and the data flow information for all blocks in p also turns out to be $\{a,b,c\}$. Thus BI_q is also $\{a,b,c\}$ and so is the data flow information for all blocks in q.

Thus we conclude that the only relevant assignment in procedures p and q is the assignment to b in $Start_p$ for local use in the condition. Local constant propagation can make even this assignment redundant. To see why the assignment in End_q is redundant, consider the paths starting at End_q. If all recursive calls to q are not over, the execution can only reach End_p and from there End_q. When all recursive calls to q finish, the execution reaches End_{main} directly or n_1 through End_p. Thus there is no use of the value assigned to a in End_q other than in the same assignment. Hence this assignment is redundant. This makes both a and b faint making the assignment to c in n_3 redundant. Discovering this through live variable analysis would require repeatedly performing dead code elimination and live variables analysis. □

8.3.5 Enumerating Summary Flow Functions

The construction method explained in previous sections requires the component flow functions to consist of primitive flow functions only. If a component flow function requires composite entity functions, the method is not applicable to the framework. The main difficulty is in being able to compose entity functions and reduce the compositions. Function applications on the other hand do not require any reduction to be performed. This leads to an interesting possibility: Instead of constructing closed form summary flow functions, the flow functions can be enumerated in terms of input output pairs by identifying the data flow values that could appear in the program as inputs to a flow function. This is possible because every program has a well defined BI and starting from BI the relevant data flow values can be constructed.

We write the enumerated form of function Φ^r_u as follows:

$$\mathcal{E}[\![\Phi^r_u]\!] = \{x \mapsto y \mid x \mapsto x \in \Phi^r_w, y = \Phi^r_u(x)\}$$

where w is the boundary point of r which is chosen depending upon the direction of flows. For backward flows w is $Exit(End_r)$ whereas for forward flows w is $Entry(Start_r)$.

Enumeration for the entire program begins at the boundary point w of the *main* procedure and is chosen to be the identity function restricted to BI.

$$\mathcal{E}[\![\Phi^{main}_w]\!] = \{BI \mapsto BI\}$$

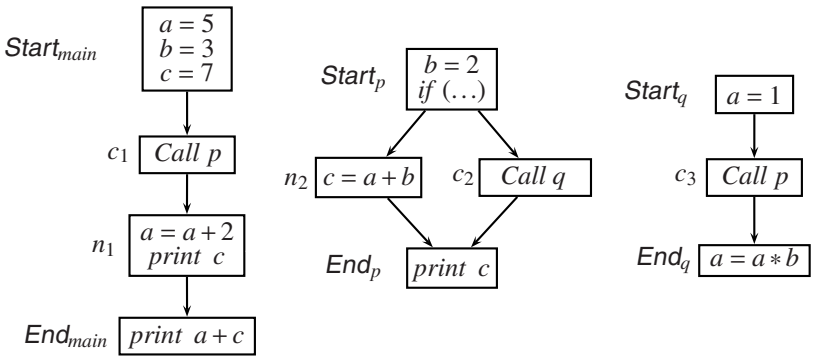


FIGURE 8.18
Program to illustrate enumeration of flow functions for full constant propagation.

Similarly, enumeration for a given procedure r also begins at its boundary point w; however, it gets defined by the enumerated summary flow functions corresponding to the call sites that call procedure r:

$$\mathcal{E}[\![\Phi_w^r]\!] = \left\{ \mathsf{x} \mapsto \mathsf{x} \mid \mathsf{y} \mapsto \mathsf{x} \in \mathcal{E}[\![\Phi_{c_i}^s]\!],\ c_i \text{ is a call to } r \text{ in procedure } s \right\} \tag{8.19}$$

For other program points v in p:

$$\mathcal{E}[\![\Phi_v^r]\!] = \left\{ \mathsf{x} \mapsto \mathsf{y} \mid \mathsf{y} = \bigsqcap_{u \in \text{neighbours}(v)} f_{u \to v}(\mathsf{z}),\ \mathsf{x} \mapsto \mathsf{z} \in \mathcal{E}[\![\Phi_u^r]\!] \right\} \tag{8.20}$$

If edge $u \to v$ represents a basic block calls procedure s, $f_{u \to v}$ is replaced by the summary flow function for procedure s. It is defined by $\mathcal{E}[\![\Phi_w^s]\!]$ where w is chosen based on the following criterion:

- If $f_{u \to v}$ is a forward flow function mapping the data flow information before the call to the data flow information after the call, w is $\mathit{Exit}(\mathit{End}_s)$.
- If $f_{u \to v}$ is a backward flow function mapping the data flow information after the call to the data flow information before the call, w is $\mathit{Entry}(\mathit{Start}_s)$.

Equations (8.19) and (8.20) are computed with $\emptyset$ as the initial value.

Once all summary flow functions are enumerated, the final data flow values are computed by simply merging the output of a summary flow function for each relevant input that has already been identified:

$$\mathsf{x}_u = \bigsqcap_{\mathsf{x} \mapsto \mathsf{y} \in \mathcal{E}[\![\Phi_u^r]\!]} \mathsf{y} \tag{8.21}$$

Observe that this is different from the method of using the closed form summary flow functions; Equations (8.17) and (8.18) require a fixed point computation whereas Equation (8.21) does not.

Procedure	Block	$\mathcal{E}[\![\Phi_u^r]\!]$		
		Iteration #1	Iteration #2	Iteration #3
p	Start_p	$\langle 5,3,7\rangle \mapsto \langle 5,3,7\rangle$	$\langle 5,3,7\rangle \mapsto \langle 5,3,7\rangle$ $\langle 1,2,7\rangle \mapsto \langle 1,2,7\rangle$	$\langle 5,3,7\rangle \mapsto \langle 5,3,7\rangle$ $\langle 1,2,7\rangle \mapsto \langle 1,2,7\rangle$
	n_2	$\langle 5,3,7\rangle \mapsto \langle 5,2,7\rangle$	$\langle 5,3,7\rangle \mapsto \langle 5,2,7\rangle$ $\langle 1,2,7\rangle \mapsto \langle 1,2,7\rangle$	$\langle 5,3,7\rangle \mapsto \langle 5,2,7\rangle$ $\langle 1,2,7\rangle \mapsto \langle 1,2,7\rangle$
	c_2	$\langle 5,3,7\rangle \mapsto \langle 5,2,7\rangle$	$\langle 5,3,7\rangle \mapsto \langle 5,2,7\rangle$ $\langle 1,2,7\rangle \mapsto \langle 1,2,7\rangle$	$\langle 5,3,7\rangle \mapsto \langle 5,2,7\rangle$ $\langle 1,2,7\rangle \mapsto \langle 1,2,7\rangle$
	End_p	$\langle 5,3,7\rangle \mapsto \langle 5,2,7\rangle$	$\langle 5,3,7\rangle \mapsto \langle 5,2,7\rangle$ $\langle 1,2,7\rangle \mapsto \langle 1,2,3\rangle$	$\langle 5,3,7\rangle \mapsto \langle 5\sqcap 2,2,7\sqcap 3\rangle$ $\langle 1,2,7\rangle \mapsto \langle 1\sqcap 2,2,3\rangle$
	f_p	$\langle 5,3,7\rangle \mapsto \langle 5,2,7\rangle$	$\langle 5,3,7\rangle \mapsto \langle 5,2,7\rangle$ $\langle 1,2,7\rangle \mapsto \langle 1,2,3\rangle$	$\langle 5,3,7\rangle \mapsto \langle \widehat{\perp},2,\widehat{\perp}\rangle$ $\langle 1,2,7\rangle \mapsto \langle \widehat{\perp},2,3\rangle$
q	Start_q	$\langle 5,2,7\rangle \mapsto \langle 5,2,7\rangle$	$\langle 5,2,7\rangle \mapsto \langle 5,2,7\rangle$ $\langle 1,2,7\rangle \mapsto \langle 1,2,7\rangle$	$\langle 5,2,7\rangle \mapsto \langle 5,2,7\rangle$ $\langle 1,2,7\rangle \mapsto \langle 1,2,7\rangle$
	c_3	$\langle 5,2,7\rangle \mapsto \langle 1,2,7\rangle$	$\langle 5,2,7\rangle \mapsto \langle 1,2,7\rangle$ $\langle 1,2,7\rangle \mapsto \langle 1,2,7\rangle$	$\langle 5,2,7\rangle \mapsto \langle 1,2,7\rangle$ $\langle 1,2,7\rangle \mapsto \langle 1,2,7\rangle$
	End_q	$\emptyset$	$\langle 5,2,7\rangle \mapsto \langle 1,2,3\rangle$ $\langle 1,2,7\rangle \mapsto \langle 1,2,3\rangle$	$\langle 5,2,7\rangle \mapsto \langle \widehat{\perp},2,3\rangle$ $\langle 1,2,7\rangle \mapsto \langle \widehat{\perp},2,3\rangle$
	f_q	$\emptyset$	$\langle 5,2,7\rangle \mapsto \langle 2,2,3\rangle$ $\langle 1,2,7\rangle \mapsto \langle 2,2,3\rangle$	$\langle 5,2,7\rangle \mapsto \langle \widehat{\perp},2,3\rangle$ $\langle 1,2,7\rangle \mapsto \langle \widehat{\perp},2,3\rangle$

FIGURE 8.19
Enumerated summary flow functions for constant propagation over procedure p and q of the program in Figure 8.18.

Example 8.11

Consider the program in Figure 8.18 on the preceding page for constant propagation analysis. When procedure p is called from *main*, the values of a, b, and c are 5, 3, and 7 respectively. If p does not call q at all, then the values of a, b, and c at End_p are 5, 2, and 7 respectively. However, if q is called, then the value of a is modified in Start_q and End_q. When the recursion unwinds, the value of c gets modified. Variable b is assignment 2 in every call to p. Thus when p returns in *main*, the value of b is 2 whereas a and c are not constants.

Iterative computation of $\mathcal{E}[\![\Phi_u^r]\!]$ for procedures p and q is shown in Figure 8.19. In the first iteration, $\mathcal{E}\left[\!\left[\Phi^q_{\mathit{End}_q}\right]\!\right]$ remains $\emptyset$: mapping $\langle 5,2,7\rangle \mapsto \langle 1,2,7\rangle$ at c_3 indicates that the value $\langle 5,2,7\rangle$ reaching Start_q is mapped to the value $\langle 1,2,7\rangle$ at c_3. However, the mapping for $\langle 1,2,7\rangle$ in procedure p has not been discovered so far; the only mapping for procedure p discovered in first iteration is $\langle 5,3,7\rangle \mapsto \langle 5,2,7\rangle$. Second iteration discovers $\langle 1,2,7\rangle \mapsto \langle 1,2,3\rangle$ for p. This leads to $\langle 5,2,7\rangle \mapsto \langle 2,2,3\rangle$ and $\langle 1,2,7\rangle \mapsto \langle 2,2,3\rangle$ for procedure q due the

assignment $a = a * b$ in End_q. This influences c_2 and End_p where it is discovered that a can be mapped to 5 if no call to q is made, 1 when p is called from q, and 2 when q returns in p. Similarly, c can be 7 and 3 depending upon whether q is called or not. Thus f_p records $\langle 5,3,7\rangle \mapsto \langle \widehat{\perp}, 2, \widehat{\perp}\rangle$ and $\langle 1,2,7\rangle \mapsto \langle \widehat{\perp}, 2, \widehat{\perp}\rangle$ whereas f_q contains $\langle 5,2,7\rangle \mapsto \langle \widehat{\perp}, 2, 3\rangle$ and $\langle 1,2,7\rangle \mapsto \langle \widehat{\perp}, 2, 3\rangle$. □

Although we have presented the enumeration of summary flow function as a fixed point computation performed using a round-robin iterative method, in practice, this may not be efficient since a program may consist of hundreds of procedures. The data flow values discovered as inputs to summary flow functions may reach limited portions of the program. In such situation, it is preferable to use a work list method and propagate the values to the relevant portions of the program.

We outline a work list based method in the following. The work list contains pairs (u, x) that represents the fact that $\mathcal{E}[\![\Phi_u^r]\!]$ has been computed for input value x and its effect needs to be propagated further. The work list is initialized with the pair (w, BI) where w is the boundary of the *main* procedure. As is typical in a work list based method, an entry from work list is removed, its effect is propagated to its neighbours, and new entries whose effect needs to be propagated are added to the work list. This process is repeated until the work list becomes empty.

We use the following notation to describe propagation.

- The meaning of $\mathcal{E}[\![\Phi_u^r]\!](\mathsf{x}) = y$ is that the mapping $\mathsf{x} \mapsto \mathsf{y}$ is included in $\mathcal{E}[\![\Phi_u^r]\!]$. Initially, $\mathcal{E}[\![\Phi_u^r]\!]$ is assumed to be empty.
- When $\mathcal{E}[\![\Phi_u^r]\!](\mathsf{x})$ appears on the right hand side of an assignment, it denotes y such that $\mathsf{x} \mapsto \mathsf{y} \in \mathcal{E}[\![\Phi_u^r]\!]$. If there is no mapping for x in $\mathcal{E}[\![\Phi_u^r]\!]$, $\mathcal{E}[\![\Phi_u^r]\!](\mathsf{x})$ denotes $\top$.

The meaning of propagating the effect of a pair (u, x) to its neighbour v is that the summary flow function $\mathcal{E}[\![\Phi_v^r]\!]$ should be constructed. Observe that v need not be in the same procedure. It could be a program point in a caller procedure or a called procedure. More precisely, propagation of (u, x) for forward flows is defined as follows.

- When u is a call node c_i calling procedure s. Let the successor of u be node v in the same procedure. Then $\mathcal{E}[\![\Phi_v^r]\!]$ should be updated with the value of the result of applying f_s to the value reaching c_i.

$$\mathcal{E}[\![\Phi_v^r]\!](\mathsf{x}) = \mathcal{E}[\![\Phi_v^r]\!](\mathsf{x}) \sqcap \mathcal{E}[\![\Phi_w^s]\!]\left(\mathcal{E}[\![\Phi_{c_i}^r]\!](\mathsf{x})\right)$$

  where w is $\mathit{Exit}(\mathit{End}_s)$. It is possible that the $\mathcal{E}[\![\Phi_w^s]\!]$ may not have been defined for x. Thus the effect should be propagated to Start_s as follows:

$$\mathcal{E}[\![\Phi_{w'}^s]\!]\left(\mathcal{E}[\![\Phi_{c_i}^r]\!](\mathsf{x})\right) = \mathcal{E}[\![\Phi_{c_i}^r]\!](\mathsf{x})$$

  where w' is $\mathit{Entry}(\mathit{Start}_s)$. If $\mathcal{E}[\![\Phi_v^r]\!]$ changes, add the pair (u, x) to the work list. If $\mathcal{E}[\![\Phi_{w'}^r]\!]$ changes, add the pair $\left(w', \mathcal{E}[\![\Phi_{c_i}^r]\!](\mathsf{x})\right)$ to the work list.

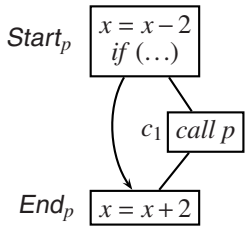


FIGURE 8.20
Example of linear constant propagation for which a closed form summary flow function can be created but summary flow functions cannot be enumerated.

- When u is $\mathit{Exit}(\mathit{End}_r)$. In this case, the summary flow function of callers need to be updated. Find out all callers t of r such that $\mathcal{E}\left[\!\left[\Phi^t_{c_j}\right]\!\right](\mathsf{y}) = \mathsf{x}$. Let the successor of c_j in t be v. Then,

$$\mathcal{E}[\![\Phi^t_v]\!](\mathsf{y}) = \mathcal{E}[\![\Phi^t_v]\!](\mathsf{y}) \sqcap \mathcal{E}[\![\Phi^r_u]\!](\mathsf{x})$$

  If $\mathcal{E}[\![\Phi^t_v]\!]$ changes, add the pair (v, y) to the work list.

- When u is some other program point. Update the summary flow function of every neighbour v of u:

$$\mathcal{E}[\![\Phi^r_v]\!](\mathsf{x}) = \mathcal{E}[\![\Phi^r_v]\!](\mathsf{x}) \sqcap f_{u\to v}(\mathcal{E}[\![\Phi^r_u]\!](\mathsf{x}))$$

  If $\mathcal{E}[\![\Phi^r_v]\!]$ changes, add the pair (u, x) to the work list.

The main difference between the two methods of enumerating the summary flow function is the fundamental difference between a round-robin method and a work list method: In a round-robin method, the relevant computation for a given program point u is performed by incorporating the effect of all its neighbours. In a work list method, the influence of a program point u is propagated to all its neighbours v and the value at u is updated.

The main advantage of enumerating summary flow functions is that there is no need to reduce function compositions because the method relies on computing actual values. However, the main limitation of computing values is that it may not terminate for a lattice with infinite values. If flow functions can be reduced, the closed form summary flow functions can be used for lattices with infinite values also.

Example 8.12

Figure 8.20 shows an example of linear constant propagation. If we construct closed form summary flow functions, we discover that $\widehat{f}^{\,x}$ in the summary flow function at $\mathit{Exit}(\mathit{End}_p)$ along the edge $\mathit{Start}_p \to \mathit{End}_p$ is a composition of $\widehat{\phi}_{1,-2}$ and $\widehat{\phi}_{1,2}$. Thus the flow function representing p along the call free path

is $\widehat{\phi}_{id}$. Along the other path, $\widehat{f}^{\,x}$ in $\Phi^p_{c_1}$ is $\widehat{\phi}_{1,-2}$. This is composed with the $\widehat{f_p}^{\,x} = \widehat{\phi}_{id}$ to construct $\Phi^p_{\mathit{End}_p}$ along this path resulting in $\Phi^p_{\mathit{End}_p} = \widehat{\phi}_{1,-2}$ when this is composed with f_{End_p}, $\widehat{f_p}^{\,x}$ is discovered to be $\widehat{\phi}_{id}$ along this path also. Thus f_p is found to be $\widehat{\phi}_{id}$ along all paths and the method concludes that the value of x before a call to x and after the call remains same.

To see how the enumeration method works for this program, we will need to know the value of x when the outermost call to p is made. Assume that x is 10 when p is called from outside. Let w denote $\mathit{Exit}(\mathit{End}_p)$. Then we have $\langle 10\rangle \mapsto \langle 10\rangle$ in $\mathcal{E}\left[\!\left[\Phi^p_{\mathit{Start}_p}\right]\!\right]$ and $\langle 10\rangle \mapsto \langle 8\rangle$ in $\mathcal{E}\left[\!\left[\Phi^p_{c_1}\right]\!\right]$. Along the other path, we get $\langle 10\rangle \mapsto \langle 10\rangle$ in $\mathcal{E}\left[\!\left[\Phi^p_w\right]\!\right]$. In order to propagate the effect of $\langle 10\rangle \mapsto \langle 8\rangle$ in $\mathcal{E}\left[\!\left[\Phi^p_{c_1}\right]\!\right]$, we find out if have $\langle 8\rangle \mapsto \langle \ldots\rangle$ in $\mathcal{E}\left[\!\left[\Phi^p_w\right]\!\right]$. Since we don't have it, we have to propagate this effect to Start_p thereby adding $\langle 8\rangle \mapsto \langle 8\rangle$ to both $\mathcal{E}\left[\!\left[\Phi^p_{\mathit{Start}_p}\right]\!\right]$ and $\mathcal{E}\left[\!\left[\Phi^p_w\right]\!\right]$. In the next iteration we check if have $\langle 6\rangle \mapsto \langle \ldots\rangle$ in $\mathcal{E}\left[\!\left[\Phi^p_w\right]\!\right]$. The process does not terminate because the recursive calls generate an infinite number of values for x. □

8.4 Summary and Concluding Remarks

In this chapter we have presented methods that construct context independent summary flow functions. Side effects analysis constructs summary flow functions for a fixed set of side effects. The key idea of this approach is to reduce expressions of sets representing flow functions. These reductions compute the sets that represent the required summary flow functions.

A natural extension of this idea results in a whole program analysis method that computes context independent summary flow functions for a given data flow framework. This extension attempts to reduce expressions of functions instead of expressions of sets. The feasibility of reducing expressions consisting of function compositions and confluences can be established in terms of the *pef*s that make up flow functions of a given data flow framework. A related concern of this method is that the canonical form of reduced expressions may not be compact and may require a lot of space. Both these concerns are addressed by the method that enumerates functions in terms of observed input output behaviour. This method does not need to reduce expressions of functions. However, its main limitation is that it may not terminate if the lattice of data flow values is not finite.

An orthogonal issue presented in this chapter is to construct functions that represent mappings of formal parameters across call sequences.

8.5 Bibliographic Notes

The side effect analysis presented in this chapter is a generalization of the work by Barth [13, 14] and Banning [12]. Callahan [19] has tried to solve the same problem using a different representation called *program summary graph*. The alias analysis of parameters is based on the work by Cooper [25] and by Cooper and Kennedy [26]. The whole program analysis is based on the classical functional approach defined by Sharir and Pnueli [93]. However, unlike Sharir and Pnueli, we use primitive entity functions to describe reductions of flow functions. The alternative approach of enumerating summary flow functions is an abstract model of the tabulation method proposed by Sharir and Pnueli. The concept of *partial transfer functions* by Wilson and Lam [107] can be viewed as similar to the tabulation method. However, it is context insensitive in recursive calls.

Another interesting method of interprocedural data flow analysis that belongs to the category of functional approaches is the method based on *graph reachability* proposed by Reps, Horwitz and Sagiv [82, 87]. This approach handles exactly the same class of frameworks that are handled by the method presented in this chapter.