5

Complexity of Iterative Data Flow Analysis

The round-robin iterative method of *MFP* computation presented in Chapter 3, was described in terms of forward data flow problems. It is a general method that can be used with suitable changes for separable and non-separable, forward and backward, unidirectional and bidirectional frameworks. We have already used the method in working out examples of various frameworks in Chapters 1, 2, and 4. However, its complexity was defined only for rapid frameworks (Chapter 3).

In this chapter we present a generic version of round-robin method and define a tight complexity bound for general monotone data flow frameworks. We also introduce work list based iterative algorithm which computes data flow information in a demand driven fashion. This algorithm forms the basis of formalizing the exact amount of work that a data flow analysis algorithm needs to perform.

5.1 Generic Flow Functions and Data Flow Equations

For simplicity of exposition, definitions of flow functions and data flow equations in Chapter 3 were restricted to forward unidirectional frameworks. They are applicable to backward unidirectional frameworks with a simple substitution of In_n by Out_n and $pred(n)$ by $succ(n)$. In either case, the following variations are possible and are equivalent in terms of the data flow information that is computed:

- Data flow equations can be defined in terms of In_n or Out_n. It is not necessary to define both In_n and Out_n. In other words, given a neighbour m of n (i.e., a successor for backward problems and a predecessor for forward problems), In_n can be computed from In_m. Similarly, Out_n can be computed from Out_m.
- The flow functions can be associated with nodes or edges. Thus the following two definitions of In_n are equivalent:

$$In_n = \prod_{p \in pred(n)} f_p(In_p)$$

$$In_n = \prod_{p \in pred(n)} f_{p \to n}(In_p)$$

The above variations are possible because the data flow information in unidirectional data flows depends on *either* the predecessors *or* successors but not on both. The

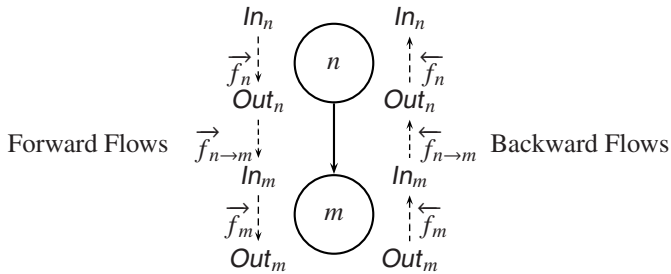


FIGURE 5.1
Associating flow functions with nodes and edges separately.

classical formulation of PRE (Section 2.4.4) does not meet these restrictions because data flow information associated with a node depends on both successors as well as predecessors. In particular, in classical PRE,

- In_n is computed from Out_n and Out_m where $m \in pred(n)$, and
- Out_n is computed from In_s where $s \in succ(n)$.

Such dependencies can be modeled by associating flow functions with nodes and edges separately as illustrated in Figure 5.1. $\overrightarrow{f}$ denotes a forward flow function whereas $\overleftarrow{f}$ denotes a backward flow function. The subscripts used in flow function notation distinguish node flow functions from edge flow functions. Defining separate node and edge flow functions requires explicating In_n and Out_n rather than leaving one of them implicit. This allows modeling the known flows as illustrated in Figure 5.2 by composing the node and edge flow functions appropriately. For forward unidirectional data flows, the forward flow functions associated with edges are identity functions ϕ_{id} and the backward node and edge flow functions are $\phi_{\top}$. Analogous remarks hold for backward unidirectional data flows. Figure 5.3 shows flow functions in forward, backward and bidirectional bit vector frameworks.

When separate flow functions are associated with nodes and edges, the generic data flow equations can be written as shown below.

$$In_n = \begin{cases} BI_{Start} \sqcap \overleftarrow{f}_n(Out_n) & n = Start \\ \left(\prod_{m \in pred(n)} \overrightarrow{f}_{m\to n}(Out_m)\right) \sqcap \overleftarrow{f}_n(Out_n) & \text{otherwise} \end{cases} \tag{5.1}$$

$$Out_n = \begin{cases} BI_{End} \sqcap \overrightarrow{f}_n(In_n) & n = End \\ \left(\prod_{m \in succ(n)} \overleftarrow{f}_{m\to n}(In_m)\right) \sqcap \overrightarrow{f}_n(In_n) & \text{otherwise} \end{cases} \tag{5.2}$$

These equations compute the *MFP* solution of an instance of a data flow framework. They can be written at an abstract level in terms of program points rather

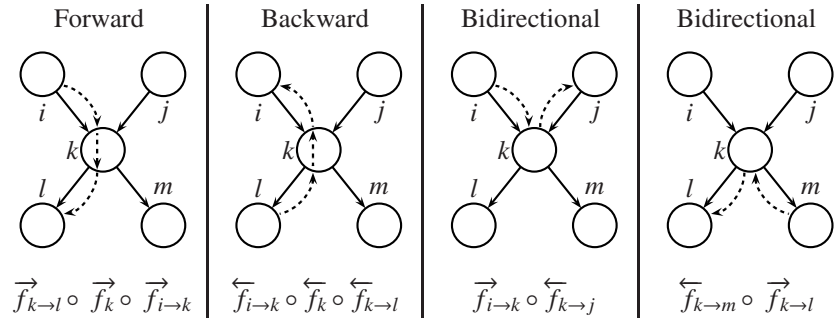


FIGURE 5.2
Representing different kinds of flows by composing node and edge flow functions.

than basic blocks as follows: Let $\mathbb{P}$oints denote the set of all program points in a given CFG and $\mathsf{x}_v \in L$ denote the data flow information associated with program point $v \in \mathbb{P}$oints. Let neighbours(v) denote the set of program points adjacent to v. Then,

$$\mathsf{x}_v = \mathit{Initial}_v \sqcap \left(\bigsqcap_{u \in \mathsf{neighbours}(v)} f_{u\to v}(\mathsf{x}_u) \right) \tag{5.3}$$

where $\mathit{Initial}_v$ is defined as

$$\mathit{Initial}_v = \begin{cases} \mathit{BI}_{\mathit{Start}} & \mathrm{v} = \mathit{Entry}(\mathit{Start}) \\ \mathit{BI}_{\mathit{End}} & \mathrm{v} = \mathit{Exit}(\mathit{End}) \\ \top & \text{otherwise} \end{cases}$$

$f_{u\to v}$ is a forward/backward node/edge flow function depending upon u and v as described below:

u	v	$f_{u\to v}$
$\mathit{Entry}(n)$	$\mathit{Exit}(n)$	$\overrightarrow{f}_n$
$\mathit{Exit}(n)$	$\mathit{Entry}(n)$	$\overleftarrow{f}_n$
$\mathit{Exit}(m)$	$\mathit{Entry}(n), m \in \mathit{pred}(n)$	$\overrightarrow{f}_{m\to n}$
$\mathit{Entry}(m)$	$\mathit{Exit}(n), m \in \mathit{succ}(n)$	$\overleftarrow{f}_{n\to m}$

This generalization can be viewed as replacing basic blocks by their entry and exit points with conceptual edges between them. The direction of these edges indicates the direction in which flow functions are applied. A given edge $u \to v$ represents a node flow function if u and v are the two end-points of the same basic block; otherwise it represents an edge flow function.

In Section 3.3.1 we have defined $\mathit{paths}(n)$ as the paths starting from *Start* reaching basic block n. We generalize this notion to define $\mathit{paths}(u)$ as the set of paths in the underlying undirected graph. These paths begin either at *Start* or *End* and

Data flow framework	$f_{u \to v}$			
	$u = \mathit{Entry}(n)$ $v = \mathit{Exit}(n)$	$u = \mathit{Exit}(n)$ $v = \mathit{Entry}(n)$	$u = \mathit{Entry}(n)$ $v = \mathit{Exit}(m)$ $m \in \mathit{pred}(n)$	$u = \mathit{Exit}(n)$ $v = \mathit{Entry}(m)$ $m \in \mathit{succ}(n)$
Reaching Definitions	$\overrightarrow{f_n}$	$\phi_\top$	ϕ_{id}	$\phi_\top$
Live Variables	$\phi_\top$	$\overleftarrow{f_n}$	$\phi_\top$	ϕ_{id}
PRE	$\phi_\top$	$\overleftarrow{f_n}$	ϕ_{id}	ϕ_{id}

FIGURE 5.3
Generic flow functions in forward, backward, and bidirectional bit vector frameworks.

reach program point u. We define the path function f_ρ for every path in $\mathit{paths}(u)$ as composition of generic flow functions along the conceptual edges in ρ.

In unidirectional forward frameworks, the *MOP* solution at node n is defined in terms of all paths in $\mathit{paths}(n)$. We define *MOP* solution at a program point u using the generalized definition of $\mathit{paths}(u)$ and generalized path function as follows:

$$\mathit{MOP}_u = \bigsqcap_{\rho \in \mathit{paths}(u)} f_\rho(\mathit{BI}_\rho) \tag{5.4}$$

where BI_ρ is $\mathit{BI}_{\mathit{Start}}$ if ρ begins at *Start*, $\mathit{BI}_{\mathit{End}}$ otherwise.

5.2 Generic Round-Robin Iterative Algorithm

A round-robin iterative algorithm for computing *MFP* assignment for forward data flow problems was described in Figure 3.9. Its version presented in Figure 3.15 uses reverse postorder traversal over the graph. This makes it efficient for forward data flow problems. Both the versions compute the data flow information at entry points of all blocks. We refer to the former version as *RR* (Round-Robin) and the latter version as *rpoRR* (Reverse PostOrder Round-Robin).

We now introduce further generalizations in terms of program points and the order of their traversal which can be chosen according to the data flow problem. We use the term *stoRR* (Specified Traversal Order Round-Robin) to refer to our algorithm. It is presented in Figure 5.4. For simplicity we assume the presence of the $\top$ element in the lattice, unlike *rpoRR*. If the lattice does not contain a $\top$ element, we replace the initialization on Line 5 by

$$\mathsf{x}_u = \mathit{Initial}_u \sqcap \left(\bigsqcap_{j \in \mathsf{neighbours}(i), j<i} f_{j \to i}(\mathsf{x}_j) \right) \tag{5.5}$$

Input: An instance $(\mathbb{G}, M_{\mathbb{G}})$ of a monotone data flow framework $(L_{\mathcal{G}}, \sqcap_{\mathcal{G}}, F_{\mathcal{G}})$. Adjacent program points i, j are mapped to $f_{i\to j}$ by $M_{\mathbb{G}}$. Program points are numbered from $0 \ldots N-1$ according to the chosen order of graph traversal.

Output: $\mathsf{x}_i, \forall\, i$ giving the output of the data flow analysis for each program point.

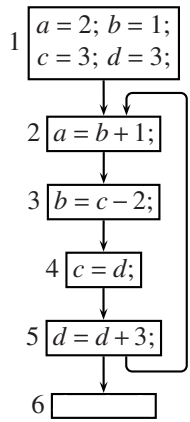
Algorithm:

```
 0  function stoRRMain()
 1  {  for all i = 0 to N − 1 do
 2     {  if i = Start then Initial_i = BI_Start
 3        else if i = End then Initial_i = BI_End
 4        else Initial_i = ⊤
 5        x_i = ⊤
 6     }
 7     change = true
 8     while change do
 9     {  change = false
10        for all i = 0 to N − 1 do
11        {  temp = Initial_i ⊓ ⊓_{j∈neighbours(i)} f_{j→i}(x_j)
12           if temp ≠ x_i then
13           {  x_i = temp
14              change = true
15           }
16        }
17     }
18  }
```

FIGURE 5.4
Round-robin algorithm for computing *MFP* assignment at each program point.

The preferred order of traversal depends on the flow functions in the data flow problem. For example, in forward problems, all node and edge flow functions are forward functions, hence reverse postorder is the most efficient order of traversal. In backward problems postorder traversal is preferable. The original bidirectional formulation of PRE contains three types of flow functions: Forward edge flow functions, backward edge flow functions and backward node flow functions. Thus a sequence of consecutive backward flow functions can be composed but a sequence of consecutive forward flow functions cannot be composed. Hence postorder traversal is the most efficient traversal.

Complexity of round-robin method is defined in context of the chosen order of graph traversal. In Chapter 3, depth of the CFG was used to define the complexity bound of round-robin method: The number of iterations required for *MFP* computation was shown to be $2 + d$ for forward bit vector frameworks and $3 + d$ for forward rapid frameworks, assuming a reverse postorder traversal. For other frameworks,



(a) A CFG with $d = 1$

	Iterations				
	1	2	3	4	5
Out_1	$\langle 2,1,3,3\rangle$	$\langle 2,1,3,3\rangle$	$\langle 2,1,3,3\rangle$	$\langle 2,1,3,3\rangle$	$\langle 2,1,3,3\rangle$
In_2	$\langle 2,1,3,3\rangle$	$\langle 2,1,3,\widehat{\bot}\rangle$	$\langle 2,1,\widehat{\bot},\widehat{\bot}\rangle$	$\langle 2,\widehat{\bot},\widehat{\bot},\widehat{\bot}\rangle$	$\langle \widehat{\bot},\widehat{\bot},\widehat{\bot},\widehat{\bot}\rangle$
Out_2	$\langle 2,1,3,3\rangle$	$\langle 2,1,3,\widehat{\bot}\rangle$	$\langle 2,1,\widehat{\bot},\widehat{\bot}\rangle$	$\langle \widehat{\bot},\widehat{\bot},\widehat{\bot},\widehat{\bot}\rangle$	$\langle \widehat{\bot},\widehat{\bot},\widehat{\bot},\widehat{\bot}\rangle$
Out_3	$\langle 2,1,3,3\rangle$	$\langle 2,1,3,\widehat{\bot}\rangle$	$\langle 2,\widehat{\bot},\widehat{\bot},\widehat{\bot}\rangle$	$\langle \widehat{\bot},\widehat{\bot},\widehat{\bot},\widehat{\bot}\rangle$	$\langle \widehat{\bot},\widehat{\bot},\widehat{\bot},\widehat{\bot}\rangle$
Out_4	$\langle 2,1,3,3\rangle$	$\langle 2,1,\widehat{\bot},\widehat{\bot}\rangle$	$\langle 2,\widehat{\bot},\widehat{\bot},\widehat{\bot}\rangle$	$\langle \widehat{\bot},\widehat{\bot},\widehat{\bot},\widehat{\bot}\rangle$	$\langle \widehat{\bot},\widehat{\bot},\widehat{\bot},\widehat{\bot}\rangle$
Out_5	$\langle 2,1,3,6\rangle$	$\langle 2,1,\widehat{\bot},\widehat{\bot}\rangle$	$\langle 2,\widehat{\bot},\widehat{\bot},\widehat{\bot}\rangle$	$\langle \widehat{\bot},\widehat{\bot},\widehat{\bot},\widehat{\bot}\rangle$	$\langle \widehat{\bot},\widehat{\bot},\widehat{\bot},\widehat{\bot}\rangle$

Data flow values of variables a, b, c, d are shown as $\langle \widehat{x}_a, \widehat{x}_b, \widehat{x}_c, \widehat{x}_d\rangle$. Initial values are $\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}\rangle$.

(b) Data flow values in round-robin method

FIGURE 5.5
Complexity of round-robin algorithm for constant propagation cannot be defined using depth of CFG.

depth of CFG is not sufficient to define complexity bounds for round-robin method.

Example 5.1

Consider the CFG in Figure 5.5(a). Statements in node 1 do not have any data dependence between them hence for simplicity we have combined them into a single block. Depth of this CFG is 1. Round-robin algorithm for constant propagation on this graph converges in 6 iterations. Part (b) of the figure shows the values at some program points each iteration. The last iteration is not shown since it is required only for detection of fixed point. It is not possible to explain the number of iterations in terms of d. □

5.3 Complexity of Round-Robin Iterative Algorithm

When *stoRR* algorithm is used for performing data flow analysis for a given instance of a framework, x_u values are initialized to $\top$ if $\top$ exists in the lattice of the framework. If the lattice does not contain $\top$, then x_u values are initialized to a suitably high value in the lattice using Equation (5.5). As the algorithm executes, the data

flow values gradually change towards $\perp$. The number of iterations required by the algorithm depends on the number of data flow value changes that can be accommodated in a single iteration. In this section we investigate the order of changes in data flow values and their impact on the number of iterations of *stoRR* algorithm.

Two main steps in our treatment of complexity analysis of *stoRR* algorithm are:

- Formalizing the notion of order of dependence of data flow values at different program points in a CFG.
- Devising a measure of how closely the order of traversal specified to the *stoRR* algorithm follows the order of dependences of data flow values.

For the first step, we present an algorithm that directly follows the dependence of data flow values. We show that this algorithm computes the same solution as the *stoRR*. This allows us to define the minimum work that any algorithm of data flow analysis must perform. Based on the observations made in the algorithm, we capture the order of dependence of data flow values at different program points by defining the concept of an *information flow path*. For a given order of traversal, it then becomes possible to quantify how close the order is to the order of the dependence of data flow values.

5.3.1 Identifying the Core Work Using Work List

In this section we describe an iterative algorithm called the *work list* algorithm which follows the order of data flow value changes, and hence is typically more efficient than round-robin method. However, it has an additional overhead of managing the work list. It follows the order of changes by restricting the computation of data flow values to paths along which changes in data flow values take place. This is different from round-robin method where a single change in the data flow information at a program point triggers another iteration which traverses all program points indiscriminately.

Figure 5.6 shows a work list based algorithm for performing data flow analysis using generic flow functions. The organization of the work list influences the efficiency of the algorithm significantly; it can be increased by incorporating heuristics such as insertion of program points in a preferred order of traversal.

Lines 1 to 7 in the algorithm initialize the data flow values at each program point to a value that is computed independently of the other program points. It is assumed that the lattice contains a $\top$ element; if it does not, then the assignment on line 5 must be modified to restrict computation of $f_{u\to v}$ to only those neighbours of v that have already been visited. Initialization of the work list involves adding the program points with non-$\top$ data flow values to the work list; a $\top$ does not influence any value. From these program points, data flow information is propagated to their neighbouring program point which in turn are added to the work list if their data flow values change.

In *stoRR* algorithm, the data flow value at a program point is *recomputed* in each iteration (line 11, Figure 5.4). This accumulates the effect of all neighbours of a

Input: An instance $(\mathbb{G}, M_{\mathbb{G}})$ of a monotone data flow framework $(L_{\mathcal{G}}, \sqcap_{\mathcal{G}}, F_{\mathcal{G}})$. Adjacent program points u, v are mapped to $f_{u\to v}$ by $M_{\mathbb{G}}$.
Output: $\mathsf{x}_u, \forall\, u$ giving the output of the data flow analysis for each program point.
Algorithm:

```
0   function worklist_dfaMain()
1   {   for all u ∈ ℙoints,
2       {   if u = Start then Initial_u = BI_Start
3           else if u = End then Initial_u = BI_End
4           else Initial_u = ⊤
5           x_u = Initial_u ⊓ ( ⊓_{v∈neighbours(u)} f_{v→u}(⊤) )
6           if x_u ⊏ ⊤ then add u to worklist
7       }
8       while worklist is not empty do
9       {   Remove the first program point u from worklist
10          for all v ∈ neighbours(u) do
11          {   temp = x_v ⊓ f_{u→v}(x_u)
12              if temp ⊏ x_v then
13              {   x_v =temp
14                  Add v to worklist
15              }
16          }
17      }
18  }
```

FIGURE 5.6
Work list algorithm for computing *MFP* assignment at each program point.

program point u on the data flow value x_u. By contrast, in a work list algorithm, a change in a value x_u is propagated to all its neighbours by *refining* their values. Refinement implied merging the old value at that point with the new value obtained from a single neighbour. Because of this difference between the two algorithms, we need to explicitly show that they compute the same assignment of data flow values. We do so by showing three important results:

- A work list algorithm terminates.
- When a work list algorithm terminates, the resulting data flow values constitute a fixed point assignment.
- Finally we show that the resulting fixed point assignment is actually the maximum fixed point assignment.

Since we know that *stoRR* algorithm also computes the *MFP* assignment, and that

the *MFP* assignment is unique, it follows that the two algorithms compute identical assignment.

For proving the properties of the work list algorithm, we define the notion of a step of the algorithm as follows. Step 1 refers to the execution of the **for** loop (lines 1 to 7). Each subsequent step corresponds to the refinement of some x_u on lines 11, 12, and 13. Each step i uses the values from step $i-1$; observe that the value used may have been computed in some earlier step. It follows that the values used in step 1 must be values from step 0; since the value used in step 1 is $\top$, we say that $\mathsf{x}_u^0 = \top$.

LEMMA 5.1

The work list algorithm terminates.

PROOF Consider some step i in the algorithm. If step i computes x_u, then due to refinement, $\mathsf{x}_u^i \sqsubseteq \mathsf{x}_u^{i-1}$. If x_u has not been modified in this step, then $\mathsf{x}_u^i = \mathsf{x}_u^{i-1}$ and u is not put on the work list. However, if x_u is modified and u is put on the work list, then $\mathsf{x}_u^i \sqsubset \mathsf{x}_u^{i-1}$. Thus the modifications in the value of x_u follow a strictly descending chain. Since all strictly descending chains are finite, each program point can be inserted in the worklist a finite number of times. Eventually the worklist becomes empty and the algorithm terminates. ∎

Now we prove that on termination, the work list algorithm computes a fixed point assignment.

LEMMA 5.2

Let the work list algorithm terminate in n steps. Then,

$$\forall u \in \mathbb{P}\mathsf{oints}: \ \mathsf{x}_u^n \sqsubseteq \mathit{Initial}_u \sqcap \left(\prod_{v \in \mathsf{neighbours}(u)} f_{v \to u}(\mathsf{x}_v^n) \right)$$

PROOF From Lemma 5.1

$$\begin{aligned} & \forall u \in \mathbb{P}\mathsf{oints}, \forall i \geq 1: \ \mathsf{x}_u^i \sqsubseteq \mathsf{x}_i^{i-1} \\ \Rightarrow \quad & \forall u \in \mathbb{P}\mathsf{oints}, \forall i \geq 1: \ \mathsf{x}_u^i \sqsubseteq \mathit{Initial}_u \qquad (\text{because } \forall u \in \mathbb{P}\mathsf{oints}: \ \mathsf{x}_u^1 \sqsubseteq \mathit{Initial}_u) \end{aligned}$$

Consider an arbitrary program point u and the last step m in which the value of x_u was computed. By the definition of refinement, we have

$$\begin{aligned} & \mathsf{x}_u^m = \mathsf{x}_u^{m-1} \sqcap f_{v \to u}(\mathsf{x}_v^{m-1}) \qquad (5.6) \\ \Rightarrow \quad & \mathsf{x}_u^m \sqsubseteq f_{v \to u}(\mathsf{x}_v^{m-1}) \end{aligned}$$

Since this is the last computation of x_u, the effect of changes in other neighbours v' of u has been incorporated by executing (5.6) for some $m' \leq m \leq n$. Hence,

$$\mathsf{x}_u^n \sqsubseteq \mathit{Initial}_u \sqcap \left(\bigsqcap_{v \in \mathsf{neighbours}(u)} f_{v \to u}(\mathsf{x}_v^{n-1}) \right) \tag{5.7}$$

The algorithm terminates when no program point is added to the work list. Thus,

$$\forall v \in \mathbb{P}\mathsf{oints} : \ \mathsf{x}_v^n = \mathsf{x}_v^{n-1}$$

Substituting the above in (5.7) results in,

$$\mathsf{x}_u^n \sqsubseteq \mathit{Initial}_u \sqcap \bigsqcap_{v \in \mathsf{neighbours}(u)} f_{v \to u}(\mathsf{x}_v^n)$$

∎

LEMMA 5.3

Let the work list algorithm terminate in n steps. Then,

$$\forall u \in \mathbb{P}\mathsf{oints} : \ \mathsf{x}_u^n \sqsupseteq \mathit{Initial}_u \sqcap \left(\bigsqcap_{v \in \mathsf{neighbours}(u)} f_{v \to u}(\mathsf{x}_v^n) \right)$$

PROOF We prove this by induction on the number of steps.

1. *Basis:* In step 1, we compute

$$\begin{aligned} & \forall u \in \mathbb{P}\mathsf{oints} : \ \mathsf{x}_u^1 = \mathit{Initial}_u \sqcap \left(\bigsqcap_{v \in \mathsf{neighbours}(u)} f_{v \to u}(\mathsf{x}_v^0 = \top) \right) \\ \Rightarrow \quad & \forall u \in \mathbb{P}\mathsf{oints} : \ \mathsf{x}_u^1 \sqsupseteq \mathit{Initial}_u \sqcap \left(\bigsqcap_{v \in \mathsf{neighbours}(u)} f_{v \to u}(\mathsf{x}_v^0 = \top) \right) \end{aligned}$$

2. *Inductive step:* Assume that for some step i

$$\forall u \in \mathbb{P}\mathsf{oints} : \ \mathsf{x}_u^i \sqsupseteq \mathit{Initial}_u \sqcap \left(\bigsqcap_{v \in \mathsf{neighbours}(u)} f_{v \to u}(\mathsf{x}_v^{i-1}) \right)$$

Consider an arbitrary program point u and step $i+1$. If x_u is not modified in step $i+1$, $\mathsf{x}_u^{i+1} = \mathsf{x}_u^i$ and by the inductive hypothesis, it trivially follows that,

$$\mathsf{x}_u^{i+1} \sqsupseteq \mathit{Initial}_u \sqcap \left(\bigsqcap_{v \in \mathsf{neighbours}(u)} f_{v \to u}(\mathsf{x}_v^i) \right)$$

Thus the interesting case that needs to be proved is when x_u is modified in step $i+1$. By the definition of refinement,

$$\mathsf{x}_u^{i+1} = \mathsf{x}_u^i \sqcap f_{v\to u}(\mathsf{x}_v^i)$$

Substituting for x_u^i from the inductive hypothesis

$$\mathsf{x}_u^{i+1} \sqsupseteq \mathit{Initial}_u \sqcap \left(\prod_{v\in \mathsf{neighbours}(u)} f_{v\to u}(\mathsf{x}_v^{i-1})\right) \sqcap f_{v\to u}(\mathsf{x}_v^i)$$

If the value of every neighbour v was modified in some step $j < i$, then $\mathsf{x}_v^i = \mathsf{x}_v^{i-1}$ and

$$\mathsf{x}_u^{i+1} \sqsupseteq \mathit{Initial}_u \sqcap \left(\prod_{v\in \mathsf{neighbours}(u)} f_{v\to u}(\mathsf{x}_v^i)\right)$$

and the lemma holds. For the other possibility, let there be a neighbour v' whose value was modified in step i. Then,

$$\mathsf{x}_u^{i+1} \sqsupseteq \mathit{Initial}_u \sqcap \left(\prod_{v\in \mathsf{neighbours}(u)} f_{v\to u}(\mathsf{x}_v^{i-1})\right) \sqcap f_{v'\to u}(\mathsf{x}_{v'}^i)$$

We rewrite the meet to separate the term for v'

$$\mathsf{x}_u^{i+1} \sqsupseteq \mathit{Initial}_u \sqcap \left(\prod_{\substack{v\in \mathsf{neighbours}(u),\\ v\neq v'}} f_{v\to u}(\mathsf{x}_v^{i-1})\right) \sqcap f_{v'\to u}(\mathsf{x}_{v'}^{i-1}) \sqcap f_{v'\to u}(\mathsf{x}_{v'}^i)$$

However, $\quad f_{v'\to u}(\mathsf{x}_{v'}^i) \sqsubseteq f_{v'\to u}(\mathsf{x}_{v'}^{i-1}) \qquad$ (because $\mathsf{x}_{v'}^i \sqsubseteq \mathsf{x}_{v'}^{i-1}$)

For all other v, the values in $i-1$ and i are same. Hence,

$$\mathsf{x}_u^{i+1} \sqsupseteq \mathit{Initial}_u \sqcap \left(\prod_{\substack{v\in \mathsf{neighbours}(u),\\ v\neq v'}} f_{v\to u}(\mathsf{x}_v^i)\right) \sqcap f_{v'\to u}(\mathsf{x}_{v'}^i)$$

$$\sqsupseteq \mathit{Initial}_u \sqcap \left(\prod_{v\in \mathsf{neighbours}(u)} f_{v\to u}(\mathsf{x}_v^i)\right)$$

Hence the lemma follows. ∎

LEMMA 5.4

The work list algorithm computes a solution of Equation (5.3).

PROOF Let the work list become empty after n steps. From Lemma (5.2), we know that

$$\forall u \in \mathbb{P}\mathsf{oints}: \ \mathsf{x}_u^n \sqsubseteq \mathit{Initial}_u \sqcap \left(\prod_{v\in \mathsf{neighbours}(u)} f_{v\to u}(\mathsf{x}_v^n)\right)$$

and from Lemma (5.3)

$$\forall u \in \mathbb{P}\text{oints}: \ \mathsf{x}_u^n \sqsupseteq \mathit{Initial}_u \sqcap \left(\bigsqcap_{v \in \text{neighbours}(u)} f_{v \to u}(\mathsf{x}_v^n) \right)$$

Hence it follows that,

$$\forall u \in \mathbb{P}\text{oints}: \ \mathsf{x}_u^n = \mathit{Initial}_u \sqcap \left(\bigsqcap_{v \in \text{neighbours}(u)} f_{v \to u}(\mathsf{x}_v^n) \right)$$

∎

LEMMA 5.5
The work list algorithm computes **MFP** *assignment of Equation (5.3).*

PROOF Consider an arbitrary solution FP of Equation (5.3). Clearly,

$$\forall u \in \mathbb{P}\text{oints}: \ FP_u = \mathit{Initial}_u \sqcap \left(\bigsqcap_{v \in \text{neighbours}(u)} f_{v \to u}(FP_v) \right)$$

Let the work list algorithm terminate after n steps. We need to prove that

$$\forall u \in \mathbb{P}\text{oints}: \ FP_u \sqsubseteq \mathsf{x}_u^n$$

We prove this by induction on step number in the work list algorithm.

1. *Basis:* From the definition of step 1,

   $$\forall u \in \mathbb{P}\text{oints}: \ \mathsf{x}_u^1 = \mathit{Initial}_u \sqcap \left(\bigsqcap_{v \in \text{neighbours}(u)} f_{v \to u}(\top) \right)$$

   Since $\forall v \in \mathbb{P}\text{oints}: \ FP_v \sqsubseteq \top$, it follows that

   $$\forall u, v \in \mathbb{P}\text{oints}: \ \left(\bigsqcap_{v \in \text{neighbours}(u)} f_{v \to u}(FP_v) \right) \sqsubseteq \left(\bigsqcap_{v \in \text{neighbours}(u)} f_{v \to u}(\top) \right)$$

   Since $\mathit{Initial}_v$ is constant,

   $$\forall u \in \mathbb{P}\text{oints}: \ FP_u \sqsubseteq \mathsf{x}_u^1$$

2. *Inductive Step:* Assume the inductive hypothesis

   $$\forall u \in \mathbb{P}\text{oints}: \ FP_u \sqsubseteq \mathsf{x}_u^i$$

   Consider an arbitrary program point u. If x_u is not modified in step $i+1$ then the inductive step trivially follows. Thus we have to show the inductive step when x_u is modified in step $i+1$. From the definition of a fixed point,

   $$\forall u \in \mathbb{P}\text{oints}, \ FP_u \sqsubseteq f_{v \to u}(FP_v) \qquad \forall v \in \text{neighbours}(u)$$

By the inductive hypothesis, $FP_v \sqsubseteq \mathsf{x}_v^i$ and hence

$$\forall u \in \mathbb{P}\text{oints}: \; FP_u \sqsubseteq f_{v \to u}(\mathsf{x}_v^i) \qquad \forall v \in \text{neighbours}(u)$$

However, from inductive hypothesis we also have

$$\forall u \in \mathbb{P}\text{oints}: \; FP_u \sqsubseteq \mathsf{x}_u^i$$

Combining the two,

$$\forall u \in \mathbb{P}\text{oints}: \; FP_u \sqsubseteq \mathsf{x}_u^i \sqcap f_{v \to u}(\mathsf{x}_v^i) \qquad \forall v \in \text{neighbours}(u)$$

From the definition of refinement,

$$\mathsf{x}_u^{i+1} = \mathsf{x}_u^i \sqcap f_{v \to u}(\mathsf{x}_v^i)$$

Hence it follows that

$$\forall u \in \mathbb{P}\text{oints}: \; FP_u \sqsubseteq \mathsf{x}_u^{i+1}$$

Since the assignment computed by the work list algorithm is a fixed point and it contains every possible fixed point FP, it must be the MFP. ∎

5.3.2 Information Flow Paths in Bit Vector Frameworks

For simplicity of exposition we begin our discussion with bit vector frameworks in which the data flow values of all entities are independent.

Recall that $\Sigma = \{\alpha, \beta, \ldots, \omega\}$ denotes the set of program entities whose data flow information is computed during data flow analysis. Since bit vector frameworks are separable, flow of information for each entity can be examined independently. Hence the discussion in this section refers to a single entity say α and its lattice $\widehat{L}$. The iterative algorithms defined in Figures 5.4 and 5.6 compute data flow information of all entities simultaneously.

Since $\widehat{L} = \{\widehat{\top}, \widehat{\bot}\}$ in bit vector frameworks, only the following three monotonic flow functions are possible: $\widehat{\phi}_\top$, $\widehat{\phi}_\bot$, and $\widehat{\phi}_{id}$ (Section 4.5). The data flow analysis of bit vector framework involves initializing data flow values to $\widehat{\top}$ and then propagating the $\widehat{\bot}$ value in the graph. The $\widehat{\bot}$ values are generated as a result of local analysis and are propagated to other program points during global analysis. We say that data flow information is *generated* at a program point if the information results from application of a constant function other than $\widehat{\phi}_\top$; in bit vector frameworks a data flow information is generated $\widehat{\phi}_\bot$. The point of generation, called *origin* of information flow is defined as follows.

DEFINITION 5.1 *A program point v is an origin of data flow information for entity α if any of the following conditions is satisfied:*

1. v *is* Entry(Start) *and* $\widehat{\mathsf{x}}_v^{\alpha} = \widehat{\perp}$ *in* BI_{Start}.
2. v *is* Exit(End) *and* $\widehat{\mathsf{x}}_v^{\alpha} = \widehat{\perp}$ *in* BI_{End}.
3. *If there exists a pair of adjacent program points* u, v *such that for some entity* α, $\widehat{f}_{u \to v} = \widehat{\phi}_{\perp}$.

DEFINITION 5.2 *An information flow path for an entity* α *in a bit vector framework is defined as a maximal acyclic sequence of adjacent program points* $p_0, p_1, \ldots p_m$ *such that* p_0 *is an origin of data flow information for* α, *and every flow function* $\widehat{f}_{p_i \to p_{i+1}}$ *is* $\widehat{\phi}_{id}$.

An information flow path represents a single thread of changes in the values of an entity in the program. In general, when there is a change in the data flow at a program point u, the flow of information terminates at u if the change at u does not cause a change in the data flow value of any neighbour v of u. In bit vector frameworks, data flow value of an entity at a program point can change only once. Since an *ifp* propagates a $\widehat{\perp}$ value, no more changes in data flow value are possible at any program point already present in the *ifp*. Hence, *ifp*s in bit vector frameworks are acyclic.

Information flow paths differ from paths in *paths*(u) in many ways: the paths in *paths*(u) always start from *Start* or *End*, *ifp*s may start from any program point. Further, a path in *paths*(u) ends on u, whereas an *ifp*s is not defined for a give program point. Paths in *paths*(u) may be cyclic, whereas *ifp*s in bit vector frameworks are always acyclic.

For brevity, we denote *Entry*(n) and *Exit*(n) by I_n and O_n respectively when depicting an information flow path. In Figure 5.2(c), the data flow indicated by the dashed line takes place along the subpath ($O_i \to I_k \to O_j$) of an *ifp*, while the data flow in Figure 5.2(d) takes place along the subpath ($I_l \to O_k \to I_m$) of an *ifp*. Figure 5.7 shows an information flow path in partial redundancy elimination for our example program. In this example, data flow information at *Exit*(n_6) is 0 as a result of assignment to c in n_4. The *ifp* responsible for propagating information from *Entry*(n_4) to *Exit*(n_6) is ($I_{n_4} \to O_{n_3} \to I_{n_5} \to O_{n_6}$) and is shown by a sequence of gray dashed arrows in the figure.

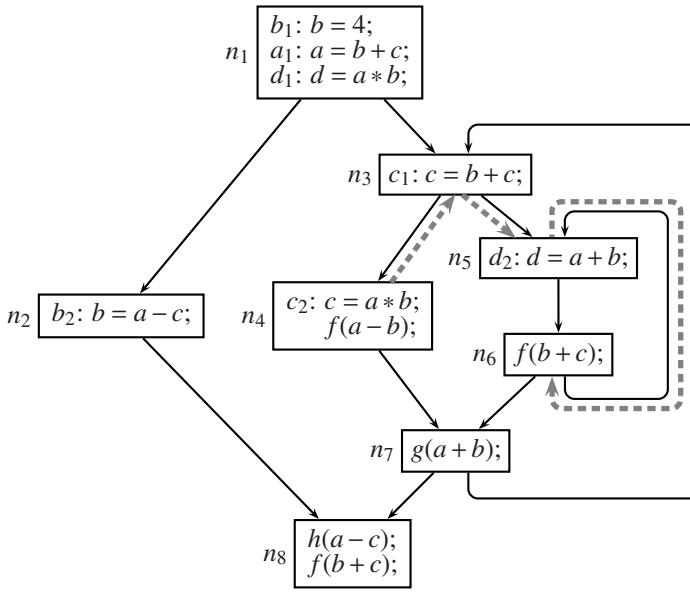
The information flow from p_0 to p_m is realized through the *path flow function* $\widehat{f}_{\rho}$ of ρ which is a composition of flow functions of all edges in ρ:

$$\widehat{f}_{\rho} = \widehat{f}_{p_{m-1} \to p_m} \circ \widehat{f}_{p_{m-2} \to p_{m-1}} \circ \cdots \circ \widehat{f}_{p_1 \to p_2} \circ \widehat{f}_{p_0 \to p_1} \tag{5.8}$$

Using the path flow function the data flow information reaching p_m from p_0 can be computed. In bit vector frameworks, the path flow function of an *ifp* is an identity function.

Information Flow Paths and the Work List Algorithm

Observe that the information flow paths in bit vector frameworks correspond to the paths traced by the generic work list based algorithm given in Figure 5.6. Program



The *ifp* ($I_{n_4} \rightarrow O_{n_3} \rightarrow I_{n_5} \rightarrow O_{n_6}$) (shown in dashed arrows) is responsible for suppressing hoisting of expression $b + c$ at $Exit(n_6)$.

FIGURE 5.7
An information flow path in PRE example.

points during initialization are essentially the origins of information flow for some entity. However, since work list algorithm operates on data flow values of all entities simultaneously, the paths traced by work list algorithms may correspond to multiple *ifp*s each referring to a different entity. Further, if a program point is added to the head of the work list, *ifp*s for an entity are traversed independently; if it is added to the rear, *ifp*s of an entity may be traversed in an interleaved fashion.

5.3.3 Defining Complexity Using Information Flow Paths

We now define the complexity of *stoRR* algorithm by relating each iteration of the algorithm to the fragment of an *ifp* that it can cover. Note that we consider the iterations of the **while** loop only; the initialization is not counted in the number of iterations unless the initialization is performed using Equation (5.5).

The discussion in this section is general and is not restricted to bit vector frameworks because it relies on the occurrence of program points in *ifp*s. Later when we define *ifp*s for fast frameworks and non-separable frameworks, the *ifp*s are extended to qualify the program points with additional information. It is done only to identify

the relevant sequences of program points and eventually the complexity is defined in terms of the sequence of program points only.

Consider an edge $p_i \rightarrow p_{i+1}$ in an *ifp*. In the *stoRR* algorithm, the order of visiting p_i and p_{i+1} depends upon the chosen order of graph traversal and is fixed throughout the analysis. This has the following consequences:

- Let p_i be visited before p_{i+1} in the order of traversal. In this case, the data flow value at p_i is computed first, and it is available during computation of the value at p_{i+1} in the same iteration. Hence, propagation of information from p_i to p_{i+1} takes place in the same iteration in which the value at p_i is computed.

- Let p_{i+1} be visited before p_i in the order of traversal. In this case, the data flow value at p_{i+1} is computed first. Hence, it must use the old value at p_i. The new value at p_i is computed in the same iteration, but can only be used for computing the value at p_{i+1} in a subsequent iteration. This implies that propagating information from p_i to p_{i+1} requires an additional iteration.

DEFINITION 5.3 *Traversal of adjacent program point p_i and p_{i+1} in an information flow path ρ is called* **conforming** *if p_i occurs before p_{i+1} in the chosen order of traversal. Otherwise, it is a* **non-conforming** *traversal.*

Conforming traversals do not contribute additional iterations in the *stoRR* algorithm whereas each non-conforming traversal requires one extra iteration.

DEFINITION 5.4 *Width of an information flow path ρ with respect to a given order of traversal is defined as the number of non-conforming traversals in ρ.*

We denote the width of an *ifp* ρ by $width(\rho)$. Width is a measure of the number of iterations required by *stoRR* algorithm to propagate information along ρ.

Example 5.2

In Figure 5.7, width of *ifp* $(I_{n_4} \rightarrow O_{n_3} \rightarrow I_{n_5} \rightarrow O_{n_6})$ is 2 since edge traversals $O_{n_3} \rightarrow I_{n_5}$ and $I_{n_5} \rightarrow O_{n_6}$ are non-conforming traversals as the CFG nodes are visited in postorder traversal. □

DEFINITION 5.5 *A span is a maximal sequence of conforming edge traversals in an ifp.*

Spans are separated by a non-conforming edge traversal and vice-versa. Thus two successive non-conforming edge traversals have a null span between them. An information flow path may begin and/or end with a null span.

The information along a span can be propagated in a single traversal over the graph. This traversal is same as the traversal of the preceding non-conforming edge.

DEFINITION 5.6 *Width of a CFG for an instance of data flow framework is defined with respect to a given order of traversal as the maximum width of any* ifp *for the given instance.*

THEOREM 5.1
If the width of a CFG for an instance of a bit vector framework is w then the round-robin iterative method stoRR *converges in* $w+1$ *iterations.*

PROOF The information flow can be initiated only after data flow values at all origins are computed. The *stoRR* algorithm achieves this in the first iteration after initialization. The same iteration propagates the information along a non-null span (if any) at the beginning of each *ifp*. Every non-conforming edge traversal and the span following it requires an additional iteration. Thus, $w+1$ iterations are sufficient along the *ifp*s that determine width of the CFG. ∎

Though the *stoRR* algorithm converges in $w+1$ iterations, in practice we do not know the width of a flow graph and the method terminates after discovering that there is no further change. Thus, practically, $w+2$ iterations are required.

The main advantage of using the notion of width is that it is uniformly applicable to general data flow frameworks including bidirectional and non-separable frameworks. Further, it is defined in terms of a specified order and hence explains the difference in the number of iterations when the order of traversal is changed.

Example 5.3
The depth of the program in Figure 5.7 for PRE is 1 whereas its width is 2. Hence the round-robin method requires at most 4 iterations to converge. □

For unidirectional data flow problems, if the direction of graph traversal is same as the direction of the data flows, the width of a graph reduces to its depth. However, depth is applicable to unidirectional data flow problems only.

5.3.4 Information Flow Paths in Fast Frameworks

Fast frameworks are separable 2-bounded frameworks. However, they are more general than bit vector frameworks in that they allow more than two elements in a component lattice, and also allow flow functions that compute incomparable values. The former requires generalizing the definition of origin while the latter requires gener-

alizing the value associated with a program point in an *ifp*.

In fast frameworks, the data flow value at a program point changes due to one of the following reasons: (a) Result of application of a flow function, or (b) Merging incomparable values from neighbours. In bit vector frameworks, the latter situation never arises because the component lattice does not contain incomparable values. In order to handle fast frameworks, the definition of information flow paths must be extended to incorporate merging of information. Also, in bit vector frameworks, an information flow path propagates the same data flow value ($\widehat{\perp}$) from an origin to all possible program points. In fast frameworks, a value at a program point may undergo more than one change due to non-identity non-constant functions and merging.

First we extend the definition of origin to allow the program point to be qualified with the generated data flow value.

DEFINITION 5.7 *A pair* $\langle v, \widehat{\mathsf{x}}_v^{\alpha} \rangle$ *is an origin of information flow for entity* α *if any of the following conditions is satisfied:*

1. v *is* **Entry**(*Start*) *and* $\widehat{\mathsf{x}}_v^{\alpha} \neq \widehat{\top}$ *in* BI_{Start}.
2. v *is* **Exit**(*End*) *and* $\widehat{\mathsf{x}}_v^{\alpha} \neq \widehat{\top}$ *in* BI_{End}.
3. *If there exists a pair of adjacent program points* u, v *such that for some entity* α, $\widehat{f}_{u \to v}^{\alpha}$ *is a constant* **pef** $\widehat{\phi}_{\mathsf{z}}$ *computing the value* $\widehat{\mathsf{z}} \neq \widehat{\top}$.

Apart from recording the data flow value, handling the merging of data flow values intermediate program points requires the following extensions:

- Merging may involve a data flow value generated by some other *ifp* traversed earlier. To remember the values computed by a different *ifp*, we define an *ifp* with respect to a given assignment $A : \mathbb{P}\mathsf{oints} \mapsto \widehat{L} \cup \{\mathit{undef}\}$. A_u^{α} denotes value of α at program point u in assignment A. Initial assignment is $\forall u \in \mathbb{P}\mathsf{oints}, A_u^{\alpha} = \widehat{\top}$ if the lattice contains a $\widehat{\top}$ element; it is $\forall u \in \mathbb{P}\mathsf{oints}, A_u^{\alpha} = \mathit{undef}$ otherwise.
- We need to define a function *latest*() to extract the latest data flow value of α at u when examining an *ifp* ρ.

An *ifp* for a fast framework is defined as follows.

DEFINITION 5.8 *Given an assignment* $A : \mathbb{P}\mathsf{oints} \mapsto \widehat{L} \cup \{\mathit{undef}\}$, *an information flow path* ρ *for an entity* α *in a fast framework is defined as a maximal acyclic sequence of tuples* $\langle p_0, \widehat{\mathsf{x}}_0 \rangle, \langle p_1, \widehat{\mathsf{x}}_1 \rangle, \ldots, \langle p_m, \widehat{\mathsf{x}}_m \rangle$ *such that* $\langle p_0, \widehat{\mathsf{x}}_0 \rangle$ *is an origin of information flow for* α, *and given* $\langle p_i, \widehat{\mathsf{x}}_i \rangle$, *its successor* $\langle p_{i+1}, \widehat{\mathsf{x}}_{i+1} \rangle$ *is defined as follows:*

1. p_i, p_{i+1} *are adjacent program points,*
2. *Let* ρ' *be the prefix of* ρ *containing* i *tuples. Then*

$$\widehat{\mathsf{x}}_{i+1} = \widehat{f}_{p_i \to p_{i+1}}(\widehat{\mathsf{x}}_i) \oplus \mathit{latest}(p_{i+1}, \rho')$$

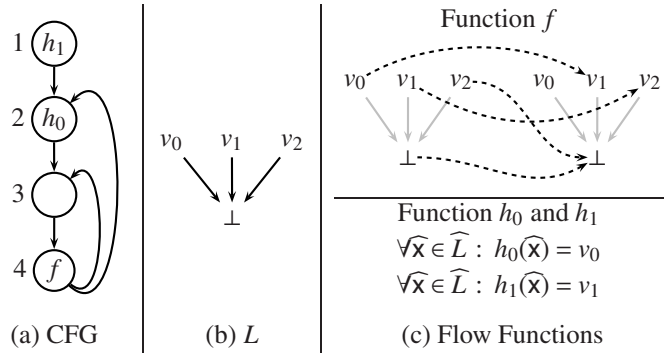


FIGURE 5.8
An instance of a distributive non-bit vector rapid framework reproduced from Figure 3.19. This instance requires $d(G,T)+3$ iterations of a round-robin algorithm with reverse post order traversal.

where

(a) $\widehat{f}^{\alpha}_{p_i \to p_{i+1}}$ *is a non-constant function.*

(b) *latest*(u,ρ') *returns value* $\widehat{x}^{\alpha}_j$ *if* p_j *is the last occurrence of* u *in* ρ'*; if* ρ' *does not contain* u*, then latest*(u,ρ') *returns* A^{α}_u.

(c) $\widehat{x} \oplus \widehat{x}' = \begin{cases} \widehat{x} & \text{if } \widehat{x}' = \textit{undef} \\ \widehat{x} \sqcap \widehat{x}' & \text{otherwise} \end{cases}$

The acyclicity condition prohibits the same pair $\langle p_i, \widehat{x}_i \rangle$ from occurring multiple times in the *ifp*; a program point may appear multiple times in the *ifp*.

Unlike bit vector frameworks, the path flow function $\widehat{f}_{\rho}$ for an *ifp* ρ need not be $\widehat{\phi}_{id}$. An assignment A used to define an *ifp* must be a valid assignment for a correct representation of information flows in a program. If it is arbitrarily chosen, the resulting complexity measures could be incorrect. For the first *ifp* traversed, $A : \forall\, u \in \mathbb{P}\text{oints}, \widehat{x}_u = \top$ is valid if $\top$ exists in the lattice of the framework; otherwise it must be $A : \forall\, u \in \mathbb{P}\text{oints}, \widehat{x}_u = \textit{undef}$. A must get updated by each subsequent *ifp*.

DEFINITION 5.9 *Given an assignment* $A : \mathbb{P}\text{oints} \mapsto \widehat{L} \cup \{\textit{undef}\}$*, and an* *ifp* ρ*, the resulting assignment* A' *is* $\forall u \in \mathbb{P}\text{oints}, A'_u = \textit{latest}(u,\rho)$ *where* $\textit{latest}(u,\rho)$ *is defined in Definition 5.8.*

Example 5.4
Consider the instance of a data flow framework shown in Figure 5.8 which has been reproduced from Example 3.11 on page 96. For simplicity, we have shown only the non-identity node flow functions and assume that there is

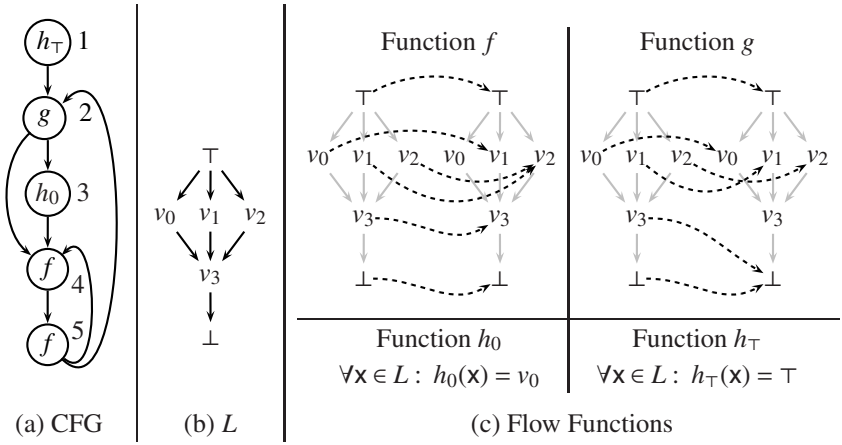


FIGURE 5.9
An instance of a distributive non-rapid fast framework that requires $d(G,T)+4$ iterations of a round-robin algorithm with reverse post order traversal.

a single unspecified entity. All edge flow functions are ϕ_{id}. Let the given assignment be $A : \forall\, u \in \mathbb{P}\text{oints}, x_u = \textit{undef}$. The constant function h_0 produces data flow value v_0 for entity *en*. Hence $\langle \textit{Exit}(2), v_0 \rangle$ is an origin of information flow. An *ifp* originating at $\langle \textit{Exit}(2), v_0 \rangle$ is

$$(\langle O_2, v_0\rangle \rightarrow \langle I_3, v_0\rangle \rightarrow \langle O_3, v_0\rangle \rightarrow \langle I_4, v_0\rangle \rightarrow \langle O_4, v_1\rangle \rightarrow$$
$$\langle I_3, \bot\rangle \rightarrow \langle O_3, \bot\rangle \rightarrow \langle I_4, \bot\rangle \rightarrow \langle O_4, \bot\rangle \rightarrow \langle I_2, \bot\rangle)$$

The round-robin algorithm requires 4 iterations to converge with a reverse post first order traversal. The data flow value at *Exit*(3) is v_0 in the first iteration. In the second iteration, it changes to $\bot$ as a result of merging the data value of *Exit*(4) and *Exit*(2). The third iteration is required to propagate this value to *Entry*(2) and the final iteration is required to detect convergence.

The depth of the CFG in example in Figure 5.8 is 1. The required number of iterations can be explained in term of width. Width of the above *ifp* is 2 due to the non-conforming edges $O_4 \rightarrow I_3$ and $O_4 \rightarrow I_2$. □

Example 5.5

Consider the instance of a data flow framework shown in Figure 5.9. We leave it for the reader to verify that is a distributive non-rapid fast framework. All edge flow functions are ϕ_{id}. Constant function h_0 produces data flow value v_0. With the initialization $\top$ at all program points, the round-robin algorithm converges in 5 iterations with a reverse post order traversal. The data flow value at *Entry*(4) changes from v_0 to v_3 to $\bot$ in the first three iterations. The

fourth iteration is required to propagate this change to *Entry*(2) and the fifth iteration is required to detect the fixed point.

The depth of the CFG is 1. The number of iterations can be explained by the following *ifp* whose origin is $\langle \mathit{Exit}(3), v_0 \rangle$.

$$(\langle O_3, v_0\rangle \rightarrow \langle I_4, v_0\rangle \rightarrow \langle O_4, v_0\rangle \rightarrow \langle I_5, v_0\rangle \rightarrow \langle O_5, v_1\rangle \rightarrow \langle I_2, v_1\rangle \rightarrow$$
$$\langle O_2, v_1\rangle \rightarrow \langle I_4, v_3\rangle \rightarrow \langle O_4, v_3\rangle \rightarrow \langle I_5, v_3\rangle \rightarrow \langle O_5, v_3\rangle \rightarrow \langle I_2, v_3\rangle \rightarrow$$
$$\langle O_2, \perp\rangle \rightarrow \langle I_4, \perp\rangle \rightarrow \langle O_4, \perp\rangle \rightarrow \langle I_5, \perp\rangle \rightarrow \langle O_5, \perp\rangle \rightarrow \langle I_2, \perp\rangle)$$

The width of this *ifp* is 3 due to three occurrences of non-conforming edge $O_5 \rightarrow I_2$; observe that the data flow values associated with the multiple occurrence of program points are different. □

5.3.5 Information Flow Paths in Non-separable Frameworks

Recall that in bit vector frameworks, only one change is possible in the data flow value of a given entity α at a given program point u. Further, the value of α at u is influenced only by the value of α at a neighbouring program point v; some other entity β cannot influence the value of α. In fast frameworks, the data flow value of α at u could change multiple times. Hence information flow paths for fast frameworks are defined in terms of a given assignment of values and a program point is qualified with the data flow value. Besides, they are also defined for a given entity due to the independence of entities.

In non-separable frameworks the possible changes in data flow values are still more general. A data flow value of an entity α at a program point u can be influenced by the data flow value of some other entity β at a neighbouring program point v. Similar to fast frameworks, data flow value of an entity could change multiple times. Thus multiple interdependent information flows are simultaneously possible at a given program point.

Example 5.6

Consider the CFG in Figure 5.10 on the next page. In constant propagation framework, the value of variable c in node 4 is influenced by the value of a computed in node 2 (via definition of b in node 5) as well as by the value of b computed in 3. We cover these influences in separate information flow paths. Also, the value of a generated in node 2 is propagated to the entry and exit points of nodes 4,5,6. This propagation is covered by a separate *ifp*. □

We continue to define an information flow path for a single thread of information flow. Since an *ifp* is defined in terms of a given assignment, using the data flow value of an entity at a the program point where multiple *ifp*s intersect, allows us to handle interdependence of information flows.

We use the concepts and notations from Section 4.5 that models the component flow functions in non-separable frameworks in terms of primitive and composite

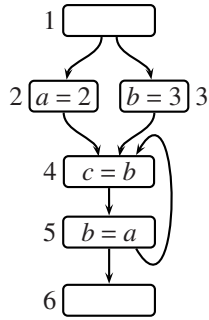


FIGURE 5.10
A CFG to illustrate information flow paths in copy constant propagation.

entity functions. We extend the notation by using program points u and v and edges between them as subscripts of a function. A component function $\widehat{f}^{\,\alpha}$ that computes the data flow value of an entity α at program point v from the values of other entities at a neighbouring program point u is denoted by $\widehat{f}^{\,\alpha}_{u\to v}$. If it can be defined in terms of primitive entity functions (*pef*s):

$$\widehat{f}^{\,\alpha}_{u\to v}(\mathsf{x}_u) = \prod_{\beta\in\Sigma} \overline{f}^{\,\beta\to\alpha}_{u\to v}\left(\widehat{\mathsf{x}}^{\beta}_{u}\right) \tag{5.9}$$

where Σ is the set of entities, and $\overline{f}^{\,\beta\to\alpha}_{u\to v}$ is the *pef* that computes the data flow value of α at program point v from the value of β at program point u.

Since we need to handle changes across different entities, we extend the notion of information flow to qualify a program point with the entity also.

DEFINITION 5.10 *A tuple* $\left\langle v, \widehat{\mathsf{z}}^{\alpha}_{v}, \alpha\right\rangle$ *is an origin of information flow for entity* α *if any of the following conditions is satisfied:*

1. v *is* Entry(Start) *and* $\widehat{\mathsf{x}}^{\alpha}_{v} \neq \widehat{\top}$ *in* BI_{Start}.
2. v *is* Exit(End) *and* $\widehat{\mathsf{x}}^{\alpha}_{v} \neq \widehat{\top}$ *in* BI_{End}.
3. *If there exists a pair of adjacent program points* u, v *such that for some entity* $\alpha \in \Sigma$, *pef* $\overline{f}^{\,\beta\to\alpha}_{u\to v}$ *is a constant pef* $\widehat{\phi}_{\mathsf{z}}$ *computing the value* $\widehat{\mathsf{z}} \neq \widehat{\top}$ *for every* $\beta \in \Sigma$.

Observe that any other *pef* cannot originate the flow of information. Similarly, a composite entity function (*cef*) also cannot originate the flow of information.

At each point in an *ifp*, we record the entity denoted *en* whose data flow value is modified at that point as a result of the application of a non-constant component

function, and use it to identify the candidate entity at the subsequent point. Changes in values due to merging of information are computed using the *latest*() function as discussed in the context of fast frameworks.

DEFINITION 5.11 *Given an assignment* $A : \mathbb{P}\text{oints} \mapsto L \cup \{\mathit{undef}\}$, *and an origin* $\langle p_0, \widehat{\mathsf{x}}_0^{\alpha}, \alpha\rangle$ *of information flow for some entity* α, *an information flow path* ρ *is defined as a maximal acyclic sequence of tuples*

$$(\langle p_0, \mathsf{x}_0, \alpha\rangle, \langle p_1, \mathsf{x}_1, \mathit{en}_1\rangle, \ldots, \langle p_m, \mathsf{x}_m, \mathit{en}_m\rangle)$$

where $\forall \beta \neq \alpha \in \Sigma, \widehat{\mathsf{x}}_0^{\beta} = A_0^{\beta}$ *and given* $\langle p_i, \mathsf{x}_i, \mathit{en}_i\rangle$, *its successor* $\langle p_{i+1}, \mathsf{x}_{i+1}, \mathit{en}_{i+1}\rangle$ *is defined as follows:*

1. p_i, p_{i+1} *are adjacent program points,*
2. *Let* ρ' *be the prefix of* ρ *containing* i *tuples. Select a* β *such that* en_i *influences* β *through a non-constant* **pef** *or a* **cef**. *Then,*

$$\mathit{en}_{i+1} = \beta$$

$$\widehat{\mathsf{x}}_{i+1}^{\gamma} = \begin{cases} \widehat{f}_{p_i \to p_{i+1}}^{\beta}(\mathsf{x}_i) \oplus \mathit{latest}(p_{i+1}, \rho', \beta) & \gamma = \beta \\ \mathit{latest}(p_{i+1}, \rho', \gamma) & \text{otherwise} \end{cases}$$

where

(a) $\mathit{latest}(u, \rho', \beta)$ *returns value* $\widehat{\mathsf{x}}_j^{\beta}$ *if* p_j *is the last occurrence of* u *in* ρ; *if* ρ *does not contain* u, *then* $\mathit{latest}(u, \rho', \beta)$ *returns* A_u^{β}.

(b) $\widehat{\mathsf{x}} \oplus \widehat{\mathsf{x}}' = \begin{cases} \widehat{\mathsf{x}} & \widehat{\mathsf{x}}' = \mathit{undef} \\ \widehat{\mathsf{x}} \sqcap \widehat{\mathsf{x}}' & \mathit{otherwise} \end{cases}$

When the changes in data flow values are not required explicitly, we denote an ***ifp*** by a sequence of program points $p_0, p_1, \ldots, p_n$. In the presence of cycles, a program point q contained in a cycle may appear multiple times in an ***ifp***. The condition of acyclicity in the definition of ***ifp*** implies that a tuple $\langle u, \mathsf{x}_u, \mathit{en}_u\rangle$ cannot appear twice in an ***ifp***, although a program point u may appear multiple times.

Example 5.7

Consider the instance of copy constant propagation for the CFG in Figure 5.10 on the facing page. Figure 5.11 on the next page shows some information flow paths for this instance. Changes in data flow values due to application of non-constant component functions are shown by adding edges from $\widehat{\mathsf{x}}_u^{\alpha} \to \widehat{\mathsf{x}}_v^{\beta}$ for each edge $\langle u, \mathsf{x}_u, \alpha\rangle \to \langle v, \mathsf{x}_v, \beta\rangle$ in the ***ifp***. Thick arrows indicate the traversal along the back edge $5 \to 4$. We leave identification of the ***ifp***s beginning at node 3 as an exercise. □

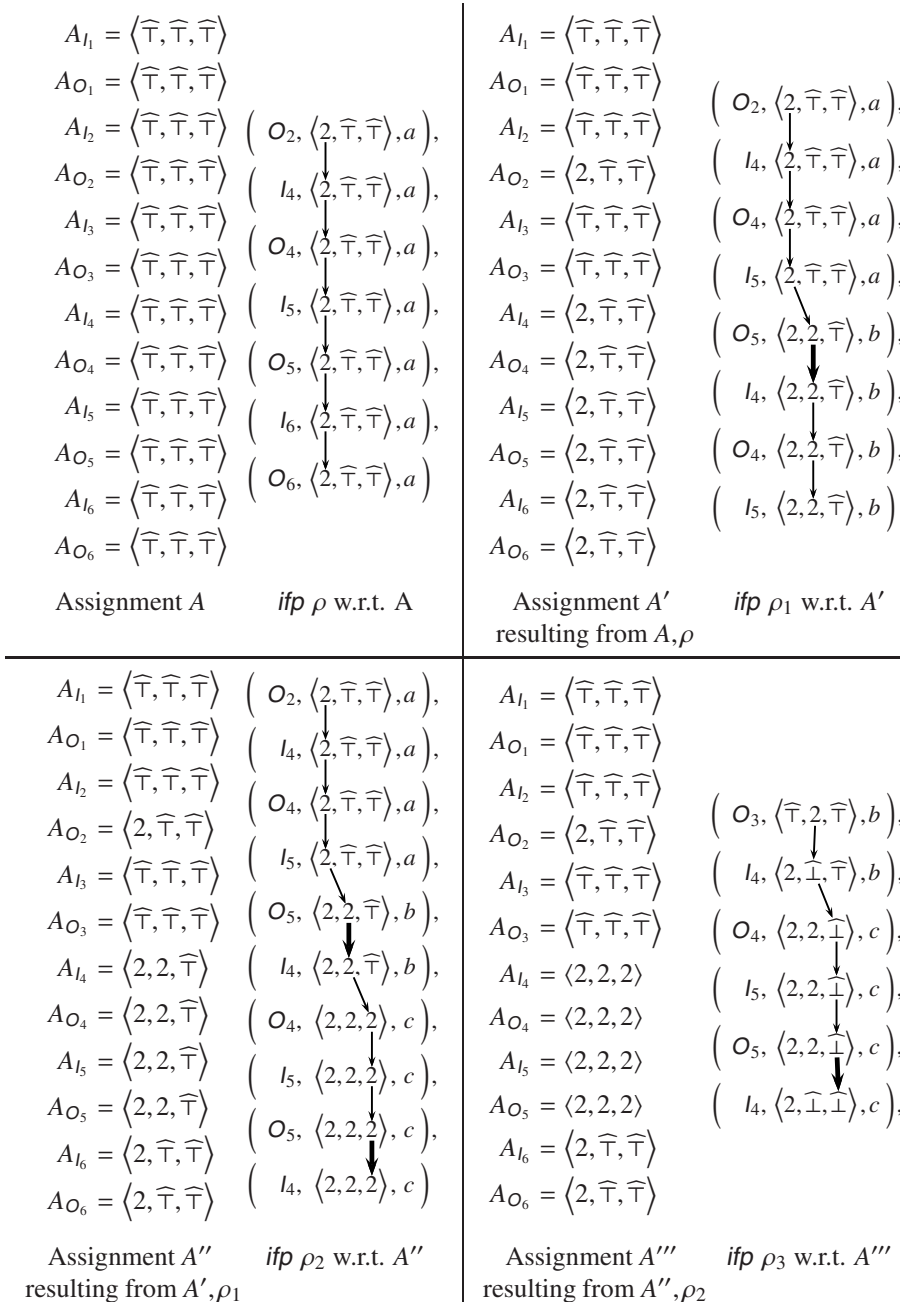
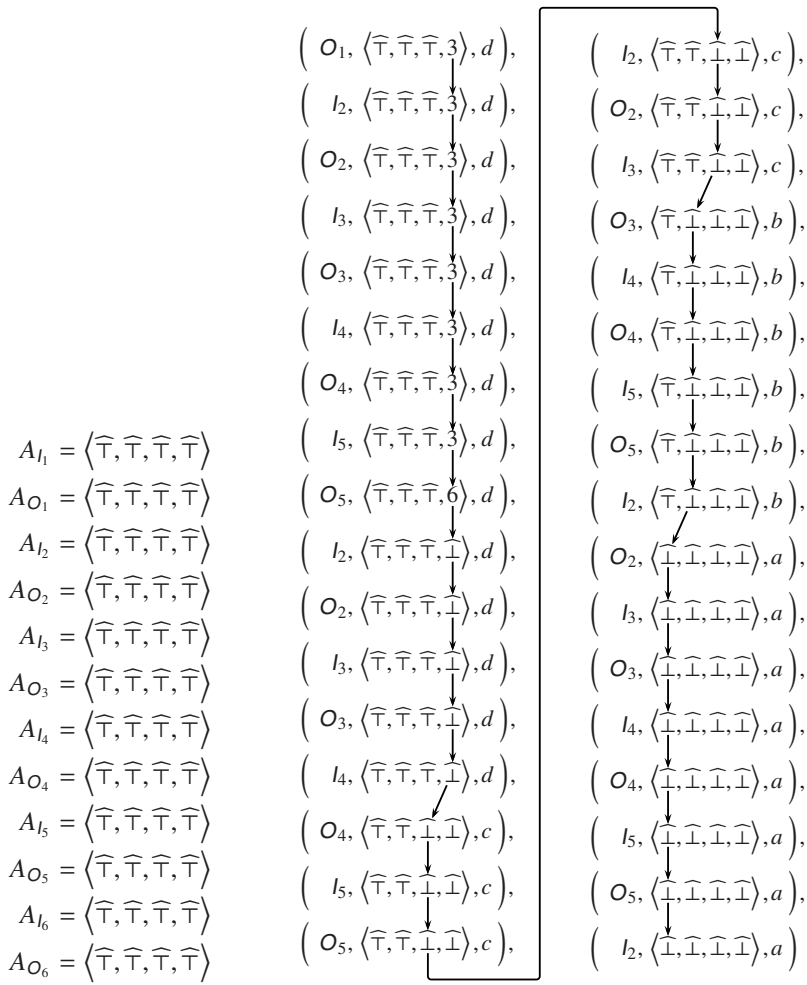


FIGURE 5.11

Some information flow paths in copy constant propagation for CFG in Figure 5.10



(a) Given assignment A

(b) An *ifp* ρ w.r.t. A. $width(\rho) = 4$

FIGURE 5.12
A width defining *ifp* in constant propagation problem in Figure 5.5 on page 164.

Example 5.8

Recall that round-robin method requires 6 iterations for Constant Propagation example for CFG with $d = 1$ in Figure 5.5 on page 164. This can be explained using the *ifp* shown in Figure 5.12. In this *ifp*, the non-conforming edge $Exit(5) \rightarrow Entry(2)$ appears 4 times, which makes width of this *ifp* 4. □

5.4 Summary and Concluding Remarks

This chapter is the culmination of generalizations across a large class of data flow frameworks. The first generalization was to define bit vector frameworks in terms of data flow equations using *Gen-Kill* components. A subsequent generalization extended the *Gen-Kill* components to general frameworks. The next step provided a uniform model of flow functions in terms of its constituent *pef*s.

This chapter has shown that such a modeling allows a clean extension of complexity measures for bit vector frameworks to the complexity measures for general frameworks. In particular, the underlying theme of information flow paths and the concept of width which governs the number of iterations of round-robin iterative analysis remains same. The only change is that the concept of the constituent points in an information flow path gets extended progressively with a transition from bit vector framework to fast frameworks and then to non-separable frameworks.

5.5 Bibliographic Notes

For a long time, the complexity measures in most of the classical literature were restricted to unidirectional data flow problems. This has also been reflected in Chapter 3 where the discussion is limited to unidirectional flows. Complexity of bidirectional problems like PRE [74] was first explained by Khedker and Dhamdhere [60] which also introduced the notion of information flow path in context of bit vector frameworks. This formed a generalized theory of bit vector data flow analyses [60, 59, 30] which provided a uniform treatment to unidirectional as well as bidirectional data flow frameworks. However it was limited to bit vector frameworks. This limitation was removed by the work by B. Karkare [53] which forms the basis of our discussion in this chapter.

We have restricted ourselves to iterative methods of data flow analysis. This is because both round-robin and work list variants of iterative data flow analysis are general methods and can be used for all data flow frameworks. For bit vector frameworks, a much larger class of methods exists. Among them, elimination methods use the structural properties of CFGs and have been widely studied. The pioneering works in elimination methods of data flow analysis are by Allen and Cocke [7], Graham and Wegman [37] and Tarjan [98]. Ryder and Paull [86] describe these methods in details. A much wider range of solution methods have been described by Hecht [44] and Kennedy [57].