# 3

# *Theoretical Abstractions in Data Flow Analysis*

The study of several examples of data flow problems suggests that they share similar features in terms of their specifications, their formulations as data flow equations, and their solution methods. In this chapter, we describe a general framework so that most of the data flow problems that we have seen earlier can be viewed as instances of this framework. Doing so yields two important benefits.

The first benefit is that which results from any generalization. When a data flow problem is shown to be an instance of the framework, it also suggests a solution method whose properties are apparent. We do not have to separately prove the correctness or estimate the complexity of the solution method.

The second benefit is that the generalization leads to the design of data flow analyzer generators, much in the way that lexer generators and parser generators have emerged from the study of formal languages. Instead of implementing each data flow analyzer separately, a general solution method that is parametrized with respect to the specific details of any analysis is implemented. When the specifics of a data flow analysis are supplied to this solution method, it yields a data flow analyzer for the particular analysis. This results in a rapid method of implementing data flow analyzers. Further, the reliability of the generated analyzers is related to the reliability of the generator. As the generator becomes more reliable through usage, the generated analyzers are likely to become more reliable than hand-coded analyzers.

This chapter deals with unidirectional data flow problems; generalizations for handling bidirectional data flow problems have been presented in Chapter 5. Further, although our descriptions are in terms of forward unidirectional problems, they are uniformly applicable to backward data flow problems. For such problems, the propagation of data flow information begins from the *End* node of the CFG instead of the *Start* node and computation of the data flow value at a node is in terms of its successors instead of its predecessors.

## 3.1 Graph Properties Relevant to Data Flow Analysis

Programs and their properties are often represented by directed or undirected graphs. A *path* in a directed graph is a sequence of nodes $(n_0, n_1, \ldots, n_k)$ such that there is an

***Input:*** A CFG $G$ with $N$ nodes.
***Output:*** A DFST $T$ for $G$ and an array $rpo[1..N]$ representing a reverse postorder listing of nodes in the graph.
***Algorithm:***

```
function dfstMain()
{   i = N
    make root(G) the root of T
    dfst(root(G))
}
function dfst(currnode)
{   mark currnode
    while there are unmarked successors of currnode do
    {   let child be an unmarked successor of currnode in
        {   add the edge (currnode → child) to T
            dfst(child)
        }
    }
    rpo[currnode] = i
    i = i − 1
}
```

**FIGURE 3.1**
An algorithm to compute a depth first spanning tree.

edge between any two consecutive nodes in the sequence. An edge between nodes $n$ and $m$ is denoted as $n \rightarrow m$. A non-null path whose starting and ending nodes are $n$ and $m$ is denoted as $n \xrightarrow{+} m$, and the corresponding unrestricted path as $n \xrightarrow{*} m$, i.e., we denote the path from $n$ to $n$ with no edges between them as $n \xrightarrow{*} n$. An edge connecting $n$ to $m$ in an undirected graph is denoted as $n — m$. The corresponding unrestricted path and non-null paths are denoted as $n \overset{*}{—} m$ and $n \overset{+}{—} m$. The *length* of the path $n_0, n_1, \ldots, n_k$ is $k$.

Recall that data flow analysis models programs in terms of CFGs which have been described in Section 2.1. As described in Chapter 2, data flow equations are defined by associating variables $In_n$ and $Out_n$ with every node $n$ in the CFG. The variables are related through data flow equations. In the examples presented in Chapter 2, the equations were solved using a round-robin iterative algorithm which traversed the CFG in a fixed order.

In this section, we present some properties of CFGs that are relevant to round-robin iterative data flow analysis. Since we restrict ourselves to CFGs, these properties are defined for connected directed graphs with a unique *Start* node.

A *spanning tree* of a directed graph $G$ is a connected subgraph of $G$ that includes all nodes of $G$ and is a tree. The root of a spanning tree is the same as the *Start* node of the graph. A *depth first spanning tree (DFST)* of $G$ is a spanning tree rooted at *Start* that is constructed by the algorithm in Figure 3.1.

***DEFINITION 3.1*** *Given a graph $G$ and its DFST $T$, the edges of $G$ can be categorized as follows:*

- *Tree edges are the edges that are in $T$.*
- *Backward edges are the edges from a node to one of its tree ancestors in $T$. A loop from a node to itself is also classified as a backward edge.*
- *Forward edges are the edges not in $T$ that connect a node to one of its tree descendants.*
- *Cross edges are the edges connecting nodes that are not related by the ancestor-descendant relation in the tree.*

The classification of edges allows us to define the following order of traversal over CFGs.

***DEFINITION 3.2*** *Given a graph $G$ and its DFST, consider the subgraph $G'$ obtained by eliminating the back edges of $G$. A reverse postorder is a topological sort of the nodes of $G'$.*

The algorithm shown in Figure 3.1 computes a reverse postorder listing of the input graph in the array *rpo*. The position of the node $n$ in this listing is *rpo*$[n]$. The nodes of the example graph of Figure 3.2 have been numbered in reverse postorder, i.e., *rpo*$[i]$ is $i$ for all nodes.

As we shall see later, the process of equation solving converges faster for forward data flow problems when the round-robin iterative algorithm traverses the CFGs graphs in reverse postorder. For backward data flow problems, the preferred order of traversal is a postorder traversal.

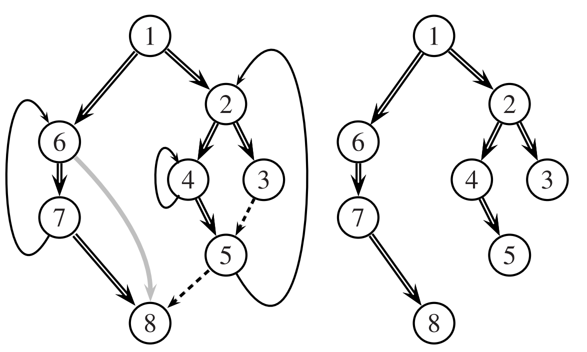
***OBSERVATION 3.1*** *Let G be a graph and T be a DFST of G. Then,*

1. *An edge $x \rightarrow y$ of G is a back edge iff rpo$[x] \geq$ rpo$[y]$.*
2. *Every cycle of G contains at least one back edge.*

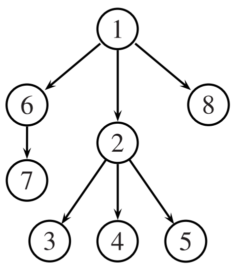
Back edges are important for unidirectional data flow problems since they propagate data flow information in a direction which is opposite to the chosen direction of graph traversal. Therefore, they may add to the number of iterations required for convergence of the analyses.

***DEFINITION 3.3*** *Let $G$ be a graph and $T$ be a DFST of $G$. The loop connectedness (more often called depth) of $G$ with respect to $T$, denoted as $d(G,T)$, is the largest number of back edges in any acyclic path in $G$.*

The depth of a graph could be different for different DFSTs. There is a special class of graphs called *reducible* graphs for which the choice of DFST does not matter because every DFST identifies exactly the same set of back edges.



(a) $G$ and one of its DFSTs. Tree edges in $G$ are shown by double lines, back edges by single black lines, the forward edge by a gray line, and cross edges by dashed lines.



(b) Dominator tree of $G$. Given an edge $x \rightarrow y$, $x = idom(y)$.

**FIGURE 3.2**
A graph, its depth first spanning tree and its dominator tree.

***DEFINITION 3.4*** *A graph G is* **reducible** *if and only if it does not contain the forbidden subgraph shown in Figure 3.3 on the next page.*

The forbidden subgraph is characterized by presence of a cycle that has two distinct entry points for paths from a node that does not appear in the cycle. The common control constructs in programs result in reducible control flow graphs. However, a compiler inserts *goto*s liberally in a program being compiled. The CFG of such a program could become irreducible after optimizations.

***DEFINITION 3.5*** *Let $n$ and $m$ be nodes in the CFG. The node $n$ is said to* **dominate** *$m$, denoted $n \geq m$, if every path from* Start *to $m$ passes through $n$.*

Dominance is, by definition, reflexive. It is also transitive. Figure 3.2(b) shows the dominator tree for our example graph.

We now prove an important result that relates dominance and reducibility.

***LEMMA 3.1***
*A graph G is reducible iff the head of every back edge in G dominates its tail.*

**PROOF** *If part*: We show that if $G$ is not reducible then there is a back edge in $G$ whose head does not dominate its tail. Indeed if $G$ is irreducible then it must contain the forbidden subgraph shown in Figure 3.3. Without any loss of generality, let us consider $b \rightarrow c$ to be a back edge. Then there is a path from *Start* to $b$ through $a$ which does not pass through $c$.

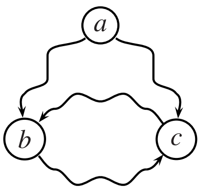


**FIGURE 3.3**
The forbidden subgraph for reducibility.

*Only if part*: We now show that if there is a back edge in $G$ whose head does not dominate its tail, $G$ is irreducible. Assume that $b \to c$ is such a back edge. Since $c$ does not dominate $b$, there is a path from *Start* to $b$ which bypasses $c$. Further, there is also a path from *Start* to $c$ which bypasses $b$, or else $b$ would have been visited before $c$ in any depth first traversal and $b \to c$ would not have been a back edge. Thus $G$ contains a forbidden subgraph with *Start*, $b$ and $c$ as the constituent nodes. ▮

The graph in Figure 3.2(a) is reducible. Some examples of edges whose addition could make it irreducible are: 1 → 7, 1 → 3, and 1 → 5.

## 3.2 Data Flow Framework

As we have said earlier, given a data flow problem, we associate data flow variables with entry and exit points of each basic block. The data flow variables are related through equations which are then solved to get data flow values at the program points. To obtain a solution of these equations, each data flow variable is initialized with a value, and the equations are iterated over till the value of each data flow variable converges.

Recall that in the case of available expressions analysis, the value of a data flow variable during an iteration is a subset of the value in the preceding iteration. In general there is an order between the values that a data flow variable takes in successive iterations during the solution process. In fact, one can impose an order on the entire space of data flow values. The order is related to the notion of approximation of data flow values that we discussed in Section 1.1.5 and is also important in reasoning about the termination of the solution procedure. Therefore the first step in the generalization of data flow problems and their solutions is to formalize this notion of order in the space of data flow values. A general way to express an order between objects is to embed them in a mathematical structure called a lattice.

The analyses studied in the previous chapter also illustrated the effect of a basic block on data flow values and the manner in which data flow values arriving along different paths are merged. Our generalization includes both these aspects of data flow analysis. The transformations effected by basic blocks on data flow values are called *flow functions*. The essential properties of flow functions and merge operations are identified as part of the generalization.

A data flow framework is an algebraic structure consisting of a set of data flow values, a set of flow functions and a merge operator.

### 3.2.1 Modeling Data Flow Values Using Lattices

Systematic computation of data flow values requires that the concept of approximations of data flow values and the operation of merging data flow values should satisfy certain properties. In this section we provide a lattice theoretic basis of these properties.

#### Partially Ordered Sets

The relation of partial order, defined below, captures the notion of approximations amongst data flow values.

***DEFINITION 3.6*** *A partial order $\sqsubseteq$ on a set $S$ is a relation over $S \times S$ that is*

1. *Reflexive. For all elements $x \in S$ : $x \sqsubseteq x$.*
2. *Transitive. For all elements $x, y, z \in S$ : $x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$.*
3. *Anti-symmetric. For all elements $x, y \in S$ : $x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$.*

*A partially ordered set (abbreviated as poset), denoted by $(S, \sqsubseteq)$, is a set $S$ with a partial order $\sqsubseteq$.*

We shall read $x \sqsubseteq y$ as "$x$ is weaker than $y$". If $x \sqsubseteq y$ and $x \neq y$, we shall say that "$x$ is strictly weaker than $y$", and denote this as $x \sqsubset y$. If $x \sqsubseteq y$ ($x \sqsubset y$), we shall also equivalently write $y \sqsupseteq x$ ($y \sqsupset x$) and read it as "$y$ is stronger than (strictly stronger than) $x$". The posets that we shall deal with will often have an element which is weaker than any other element in the poset. Such an element, if it exists, is called the *least element* and denoted as $\bot$. The *greatest element*, defined similarly, will be denoted as $\top$.

***Example 3.1***
The poset of data flow values in live variables analysis is shown in Figure 3.4 on the facing page. Here, the set of all data flow values, denoted by $L_{lv}$, is $2^{\mathbb{V}\mathsf{ar}}$, where $\mathbb{V}\mathsf{ar}$ denotes the set of variables in a program. The partial order is: For $x_i$ and $x_j$ in $L_{lv}$, $x_i \sqsubseteq_{lv} x_j$ iff $x_i \supseteq x_j$. The greatest element of this poset is $\emptyset$ and the least element is $\mathbb{V}\mathsf{ar}$. $\square$

$$\begin{tikzcd}
 & & & & & \emptyset \arrow[dlll] \arrow[dl] \arrow[dr] \arrow[drrr] & & & & & \\
 & & \{a\} \arrow[dll] \arrow[d] \arrow[drr] & & \{b\} \arrow[dllll] \arrow[drr] \arrow[drrrr] & & \{c\} \arrow[dllll] \arrow[d] \arrow[drrrr] & & \{d\} \arrow[dllll] \arrow[d] \arrow[drr] & & \\
\{a,b\} \arrow[drr] \arrow[drrrr] & & \{a,c\} \arrow[d] \arrow[drrrr] & & \{a,d\} \arrow[d] \arrow[drr] & & \{b,c\} \arrow[dllll] \arrow[drr] & & \{b,d\} \arrow[dllll] \arrow[d] & & \{c,d\} \arrow[dllll] \arrow[dll] \\
 & & \{a,b,c\} \arrow[drrr] & & \{a,b,d\} \arrow[dr] & & \{a,c,d\} \arrow[dl] & & \{b,c,d\} \arrow[dlll] & & \\
 & & & & & \{a,b,c,d\} & & & & &
\end{tikzcd}$$

**FIGURE 3.4**
$L_{lv}$ as a partially ordered set.

In the representation of the poset as a directed graph, $x_i \sqsubseteq x_j$, if there is a directed path from $x_j$ to $x_i$. Since paths of length 0 are also possible, for every element $x_i$, $x_i \sqsubseteq x_i$. Such representations of posets are called Hasse diagrams*.

***Example 3.2***
As a dual example, consider the poset of data flow values for available expressions analysis. If we denote the set of all expressions occurring in the program as $\mathbb{E}\mathsf{xpr}$, then the set of data flow values, $L_{av}$, is $2^{\mathbb{E}\mathsf{xpr}}$, the set of all subsets of $\mathbb{E}\mathsf{xpr}$. Consider the partial order $\sqsubseteq_{av}$ defined as: for all $x_i$ and $x_j$ in $L_{av}$, $x_i \sqsubseteq_{av} x_j$ iff $x_i \subseteq x_j$. The least element of this poset is the empty set $\emptyset$ and the top element is $\mathbb{E}\mathsf{xpr}$. □

In the context of data flow analysis, the relation $\sqsubseteq$ can be interpreted as "a conservative (safe) approximation of". If $x \sqsubseteq y$, then, in any context, the data flow value $x$ can be used in place of $y$ for optimization without affecting the correctness of the optimized program. As an example, consider the use of liveness analysis for either dead code elimination (Section 2.3.1) or freeing memory objects (Section 1.1.5). If $y$ is the set of variables that are actually live, then performing an optimization on the basis of a set $x$ that is larger than $y$ will not make the optimization unsafe. It is for this reason that the $\sqsubseteq$ relation in the case of live variables analysis is $\supseteq$. Similarly, for optimizations which are based on available expressions analysis like common subexpression or partial redundancy elimination, an optimization performed at a program

*Traditionally, Hasse diagrams are undirected graphs with the implicit assumption that $x_i \sqsubseteq x_j$ if $x_j$ is drawn at a higher level in the diagram than $x_i$. We have, instead, chosen to make the graph directed with the hope that this lends to clarity.

point on the basis of a set of expressions $y$ can safely be replaced by one based on a subset $x$ of $y$. Therefore the partial order in the case of available expressions analysis is $\subseteq$.

Conversely, $x \sqsupseteq y$ can be interpreted as follows: In any context, the data flow value $x$ provides more opportunities for optimization than $y$, or, using the terminology introduced in Chapter 1, $x$ is more exhaustive than $y$. This may be at the cost of safety, i.e., the optimization based on $x$ may result in a program that has a behavior different from the original program. We would like the data flow values resulting from our analyses to be safe and yet provide maximum opportunities for optimization.

***DEFINITION 3.7*** *Let $(L, \sqsubseteq)$ be a poset and let $S \subseteq L$. An element $x \in L$ is an upper bound of $S$ iff for all $y \in S$, $y \sqsubseteq x$. Similarly, an element $x \in L$ is a lower bound of $S$ iff for all $y \in S$, $x \sqsubseteq y$.*

In the graphical representation of a poset, $x$ is an upper bound of $S$ iff there are paths from $x$ to each element of $S$. Similarly, $x$ is an lower bound of $S$ iff there are paths from each element of $S$ to $x$. Also note that the definition above does not require the upper bound of a set to be in the set itself. As an example, let $S = \{\{a,b\},\{b,c\}\}$ in Figure 3.4. Then none of the upper or lower bounds of $S$ are in $S$.

***DEFINITION 3.8*** *The least upper bound (lub) of a set $S$ is an element $x$ such that (i) $x$ is an upper bound of $S$, and (ii) for all other upper bounds $y$ of $S$, $x \sqsubseteq y$. The greatest lower bound (glb) of a set is an element $x$ such that (i) $x$ is a lower bound of $S$, and (ii) for all other lower bounds $y$ of $S$, $y \sqsubseteq x$.*

Referring once again to Figure 3.4 on the previous page, $\{a,b\}$, $\{a\}$, $\{b\}$ are all upper bounds of the set $\{\{a,b,c\},\{a,b,d\}\}$. However the *lub* of this set is $\{a,b\}$.

The *lub* of a set $S$ is also called the *join* of $S$ and is denoted as $\bigsqcup S$. The *glb* of a set $S$ is also called the *meet* of $S$ and is denoted as $\bigsqcap S$. $\bigsqcup$ can also be used as an infix operator; $x \sqcup y$ denotes the *lub* of the two elements $x$ and $y$. The *lub* (*glb*) of a set, if it exists, is unique. It can be verified that the join (and meet) operator has the following properties:

1. *Idempotence*. $\forall x \in S : x \sqcap x = x$.
2. *Commutativity*. $\forall x, y \in S : x \sqcap y = y \sqcap x$.
3. *Associativity*. $\forall x, y, z \in S : (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$.

In the context of data flow analysis, the meet operator is used to merge data flow values along different paths and reaching a join node of the underlying CFG. The result of the meet operation is the most exhaustive safe approximation of data flow values along each of the paths.

***OBSERVATION 3.2*** *Let $L$ be a poset and $S$ be a subset of $L$ whose glb exists. Let $x \in L$. If $x \sqsubseteq y$ for each $y \in S$, then $x \sqsubseteq \bigsqcap S$. This is just a restatement of the fact that any lower bound of a poset is weaker than the glb of the poset.*

It is important to mention that the posets that represent data flow values may be infinite. However, since each data flow value is a finite quantity, the posets are countable. Since we want to present algorithms that search for solutions of equations in posets which may be countably infinite, we have to impose additional constraints on these posets to ensure termination of the algorithms.

***DEFINITION 3.9*** *A chain $S$ is a subset of a poset which is totally ordered, i.e., $\forall x,y \in S : x \sqsubseteq y$ or $y \sqsubseteq x$. A descending chain is a sequence of elements $\{x_1, x_2, \ldots\}$ from a poset such that $i \leq j$ implies $x_i \sqsupseteq x_j$.*

***DEFINITION 3.10*** *A descending chain $\{x_1, x_2, \ldots\}$ stabilizes eventually iff $\exists n, \forall m > n : x_m = x_n$.*

***DEFINITION 3.11*** *A poset satisfies the descending chain condition iff every descending chain in the poset stabilizes eventually.*

The importance of the descending chain condition is that it allows us to extend the guarantee of existence of meets to countably infinite sets. Let $S = \{x_1, x_2, x_3, \ldots\}$ be such a set. Then the values $\bigsqcap_{i=1}^{k} x_i$, $k = 1, 2, \ldots$, form a chain. Because of the descending chain condition, there is an $m$ such that for any $n > m$, $\bigsqcap_{i=1}^{m} x_i = \bigsqcap_{i=1}^{n} x_i$. Then $\bigsqcap_{i=1}^{m} x_i$ is the *glb* of $S$.

Analogous to the descending chain condition, we can also define the ascending chain condition. However, since data flow analysis uses the meet operator for confluence, the result of merging is a lower bound of the data flow values being merged. Hence we are interested in the descending chain condition rather than the ascending chain condition. In the rest of the chapter we restrict the discussions to posets that satisfy the descending chain condition.

## Lattices and Complete Lattices

During data flow analysis, we have to merge sets of data flow values. Therefore it is important to ensure that the meet of such sets exists.

***DEFINITION 3.12*** *A poset $(L, \sqsubseteq)$ is a lattice, iff, for each non-empty finite subset $S$ of $L$, both $\bigsqcup S$ and $\bigsqcap S$ are in $L$. $L$ is a complete lattice, iff, for each subset $S$ of $L$, both $\bigsqcup S$ and $\bigsqcap S$ are in $L$.*

The condition that every non-empty finite subset must have a *glb* and a *lub* in $L$ is equivalent to the condition that for any pair of elements $x$ and $y$, both $x \sqcup y$ and $x \sqcap y$ should be in $L$. For the lattice $L$ to be complete, even $\emptyset$ and infinite subsets of $L$ must have a *glb* and *lub* in $L$.

***Example 3.3***
The posets $(L_{lv}, \sqsubseteq_{lv})$ and $(L_{av}, \sqsubseteq_{av})$ are complete lattices. An example of a lattice which is not complete is the set of natural numbers $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$, ordered by $\leq$. In fact, any infinite set in this lattice does not have a *lub*. This set can be converted to a complete lattice by adding the element $\infty$ with the property that for any $x \in \mathbb{N}, x \leq \infty$. ▯

For a poset $L$, the conditions (i) $\bigsqcup S \in L$ for every subset $S$ of $L$ and (ii) $\bigsqcap S \in L$ for every subset $S$ of $L$ are equivalent. Thus for a poset to be a complete lattice, it is enough to require one of the two conditions to hold, the other is automatically satisfied. To see this, assume that the *glb* of every subset of $L$ exists in $L$. We have to show that for an arbitrary $S \subseteq L$, the *lub* of $S$ exists. Consider the set $B$ of all upper bounds of $S$. Since every element of $S$ is a lower bound of $B$, from Observation 3.2 $\bigsqcap B$ is an upper bound of $S$. In particular, it is the least upper bound of $S$.

If $L$ is a complete lattice, then we denote the top element of the lattice, $\bigsqcup L$, by $\top$. Similarly, the bottom element of the lattice, $\bigsqcap L$ is denoted by $\bot$. Since every subset of $L$ must have a *glb* and a *lub*, $\emptyset$ must also have a *glb* and a *lub*. It turns out that $\bigsqcap \emptyset$ is $\top$ and $\bigsqcup \emptyset$ is $\bot$. To see this, consider the definition of a lower bound of $S$: $x$ is a lower bound of $S$ iff $\forall y \in S : x \sqsubseteq y$. When $S$ is $\emptyset$, every element of $L$ is vacuously a lower bound of $S$. The greatest among them, $\top$, is the *glb* of $\emptyset$. For similar reasons $\bot$ is the same as $\bigsqcup \emptyset$. Observe that $\emptyset$ cannot be a complete lattice; the smallest poset which is complete must contain at least one element which can serve as both $\top$ and $\bot$.

Very often we shall consider tuples of values, each component of the tuple coming from a complete lattice. In such a case, the tuples themselves also form a complete lattice.

***DEFINITION 3.13*** *Let $L_i$, $1 \leq i \leq m$ be complete lattices with the partial order $\sqsubseteq_i$ and meet $\sqcap_i$. Then the cross-product $L = L_1 \times L_2 \times \ldots \times L_m$ is also a complete lattice with the partial order:*

$$\langle x_1, x_2, \ldots, x_m \rangle \sqsubseteq \langle y_1, y_2, \ldots, y_m \rangle \textit{ iff } x_i \sqsubseteq_i y_i \textit{ for all } i,\ 1 \leq i \leq m$$

*and the induced meet*

$$\langle x_1, x_2, \ldots, x_m \rangle \sqcap \langle y_1, y_2, \ldots, y_m \rangle = \langle x_1 \sqcap_1 y_1,\ x_2 \sqcap_2 y_2,\ \ldots,\ x_n \sqcap_n y_n \rangle$$

*The $L_i$s are called the components of the product lattice $L$.*

$$\begin{tikzcd}
\langle x \circeq y, \textsf{must}\rangle \arrow[dr] & & \langle x \circeq y, \textsf{no}\rangle \arrow[dl] \\
& \langle x \circeq y, \textsf{may}\rangle &
\end{tikzcd}$$

**FIGURE 3.5**
A meet semilattice for *may-must* alias analysis.

## Meet Semilattices

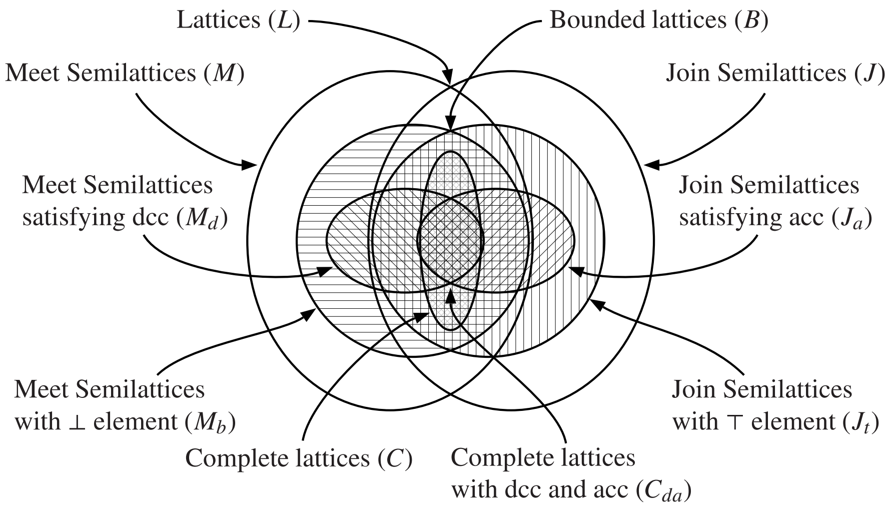
Although the data flow values of most of the analyses can be modeled as a complete lattice, there are analyses whose data flow values cannot be modeled even as a lattice. Hence we need a structure which is less restrictive than a lattice.

***Example 3.4***
As an example of a data flow analysis problem which cannot be modeled naturally as a lattice, consider the combined *may-must* alias analysis problem. This is a similar to the combined *may-must* analysis described in Section 2.5. Assume that a program has just two pointer variables $x$ and $y$. The data flow values for this problem can be modeled as a pair $(x \circeq y, d)$, where $d$ is one of the three values *must*, *may* and *no*. The data flow value $(x \circeq y, \textsf{must})$ at a program point $p$ indicates that $x$ and $y$ are aliased along all paths reaching $p$, $(x \circeq y, \textsf{may})$ indicates that $x$ and $y$ are aliased along some paths and not along all paths reaching $p$, and $(x \circeq y, \textsf{no})$ indicates that $x$ and $y$ are not aliased along any path reaching $p$. The poset of these data flow values is shown in Figure 3.5. □

In particular, we shall consider posets in which subsets have a *glb* but drop the requirement that they have a *lub* as well. Thus these lattices may not have a $\top$ element. The poset in Figure 3.5 is an example of a meet semilattice.

**DEFINITION 3.14** *A poset $(L, \sqsubseteq)$ is a meet semilattice, iff, for each non-empty finite subset $S$ of $L$, $\bigsqcap S$ is in $L$.*

We are interested in meet semilattices that satisfy the descending chain condition. Further, some of the algorithms that we discuss (algorithm in Figure 3.9, for example) assume the existence of the greatest element $\top$. In general, it is possible to modify these algorithms to avoid using $\top$ (algorithm in Figure 3.15). However, it is often convenient to use an element outside of the meet semilattice and give it the status of the $\top$ element. As an example, the algorithm used for *may-must* availability analysis in Section 2.5 required a $\top$ and hence a fictitious value *unknown* was added to the meet semilattice. Adding a new value requires us to define all flow functions for the value. We shall assume that for all functions $f$, $f(\top) = \top$. This extension preserves monotonicity of functions. Adding a $\top$ element to a meet semilattice results


Lattices ($L$)
Bounded lattices ($B$)
Meet Semilattices ($M$)
Join Semilattices ($J$)
Meet Semilattices satisfying dcc ($M_d$)
Join Semilattices satisfying acc ($J_a$)
Meet Semilattices with $\perp$ element ($M_b$)
Join Semilattices with $\top$ element ($J_t$)
Complete lattices ($C$)
Complete lattices with dcc and acc ($C_{da}$)

- dcc = descending chain condition
- acc = ascending chain condition

| | |
|---|---|
| $L = M \cap J$ | $M_d \subseteq M_b \subseteq M$ |
| $B = M_b \cap J_t$ | $J_a \subseteq J_t \subseteq J$ |
| $C_{da} = M_d \cap J_a$ | $C_{da} \subseteq C \subseteq B \subseteq L$ |

**FIGURE 3.6**
Relationships between different types of posets. The posets are assumed to be countable.

in a bounded lattice, i.e., a lattice with $\top$ and $\perp$ elements. Note that a bounded lattice need not be complete because arbitrary subsets may not have a *lub* or *glb*.

***Example 3.5***

Consider the poset $(A, \subseteq)$ of all finite subsets of the set of integers $\mathbb{I}$. Since every element of $A$ is a subset of $\mathbb{I}$, the poset $(A \cup \{\mathbb{I}\}, \subseteq)$ is a bounded lattice with $\mathbb{I}$ and $\emptyset$ as $\top$ and $\perp$. However, it is not a complete lattice because the join ($\cup$) of arbitrary subsets of $A \cup \{\mathbb{I}\}$ may not exist in $A \cup \{\mathbb{I}\}$. For example, the union of all sets that do not contain a given number (say 1) does not exist in $A \cup \{\mathbb{I}\}$. $\Box$

It is possible to define a join semilattice much in the same way as a meet semilattice. Figure 3.6 illustrates the relationships between different kinds of posets.

## 3.2.2 Modeling Flow Functions

Recall that the data flow equations for reaching definitions analysis are:

$$In_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcup\limits_{p\in pred(n)} Out_p & \text{otherwise} \end{cases} \tag{3.1}$$

$$Out_n = (In_n - Kill_n) \cup Gen_n \tag{3.2}$$

where $In_n$ and $Out_n$ are data flow variables, whose values are being defined by the data flow equations and $Gen_n$ and $Kill_n$ are constants whose values depend on the contents of node $n$. $BI$ is the information that is available at the *Start* block of the CFG.

For unidirectional problems, having two sets of variables $In_n$ and $Out_n$ is not essential—it just increases the readability of the equations. To avoid proliferation of variables in the ensuing discussion, we substitute for $Out$ in the equations for $In$, and get

$$In_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcup\limits_{p\in pred(n)} (In_p - Kill_p) \cup Gen_p & \text{otherwise} \end{cases} \tag{3.3}$$

Expressing $(In_p - Kill_p) \cup Gen_p$ as the application of a flow function $f_p$ on $In_p$, we have:

$$In_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcup\limits_{p\in pred(n)} f_p(In_p) & \text{otherwise} \end{cases} \tag{3.4}$$

We generalize Equations (3.4) so that the set of equations for any data flow analysis can be seen as an instance of the general set of equations shown below:

$$In_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \prod\limits_{p\in pred(n)} f_p(In_p) & \text{otherwise} \end{cases} \tag{3.5}$$

In Equation (3.5), $\sqcap$ is the meet operator used to merge data flow information along different paths. If the set of data flow values is $L$, then $f_n : L \mapsto L$ represents the transformation of the data flow values that reach the basic block $n$ by the statements in $n$. These functions are called *flow functions*. Two important and related properties of flow functions are monotonicity and distributivity.

**DEFINITION 3.15** *A function $f : L \mapsto L$ is called monotonic iff*

$$\forall x, y \in L : \; x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

Monotonicity implies that the flow functions are well-behaved in the sense that they preserve the order of approximations.

***DEFINITION 3.16*** *A function $f : L \mapsto L$ is called distributive iff*

$$\forall x, y \in L: \;\; f(x \sqcap y) = f(x) \sqcap f(y)$$

***OBSERVATION 3.3*** *If $f$ is monotonic then $f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y)$.*

***OBSERVATION 3.4*** *Every distributive function is also monotonic. If $x \sqsubseteq y$, then $x = x \sqcap y$ and distributivity gives $f(x) = f(x) \sqcap f(y)$. This implies $f(x) \sqsubseteq f(y)$.*

Distributivity is a stronger condition than monotonicity. A distributive function not only preserves the order of approximations but also guarantees that merging information before function application does not result in any loss of precision.

In our generalization we shall assume that the set of flow functions $F$ has the following properties:

1. The identity function $id \in F$. This is the flow function for the empty block of statements.
2. If $f \in F$ and $g \in F$, then $f \circ g \in F$. Composing the flow functions transformations of two basic blocks results in a flow function.
3. The functions in $F$ are monotonic.
4. For every $x \in L$, there is a finite set of flow functions $\{f_1, f_2, \dots f_m\}$ such that $x = \bigsqcap_{1 \le i \le m} f_i(\mathit{BI})$. This condition arises from the fact that solution procedures can only compute data flow values which are expressible as a finite meet of flow functions applied to $\mathit{BI}$. This condition can be seen either as a minimality condition on the set of data flow values or as a sufficiency condition on the set of flow functions.

The above four conditions characterize the set of admissible functions for data flow analysis.

### 3.2.3 Data Flow Frameworks

Having discussed lattice theoretic modeling of data flow values and the admissible flow functions, we now combine the two to present a generalization called data flow frameworks.

***DEFINITION 3.17*** *A data flow framework is a tuple $(L_{\mathcal{G}}, \sqcap_{\mathcal{G}}, F_{\mathcal{G}})$, where $\mathcal{G}$ is a symbol standing for a unspecified CFG, and :*

- $L_{\mathcal{G}}$ *is a description of a meet semilattice that represents the data flow values relevant to the problem.* $L_{\mathcal{G}}$ *must satisfy the descending chain condition.*

- $\sqcap_{\mathcal{G}}$ *is a description of the meet operator of the semilattice.* $\sqcap_{\mathcal{G}}$ *is, of course, derivable from* $L_{\mathcal{G}}$*.*

- $F_{\mathcal{G}}$ *is a description of the set of admissible flow functions from* $L_{\mathcal{G}}$ *to* $L_{\mathcal{G}}$*. Each flow function has an associated* direction *which could be along the control flow in the unspecified CFG* $\mathcal{G}$ *or against it.*

*Forward* flow functions indicate flow of information along the flow of control: The data flow information associated with a node is influenced by its predecessors. *Backward* flow functions indicate flow of information against the flow of control: The data flow information at a node is influenced by its successors. In *unidirectional* data flow frameworks, all functions have the same direction; *bidirectional* frameworks have a combination of flow functions in both directions.

Since we assume that the set of admissible functions are monotonic, we call the framework a *monotone data flow framework.* If the admissible functions are distributive, we call the framework a *distributive data flow framework.*

**Example 3.6**
As an example of a monotone data flow framework, consider available expressions analysis. In this framework $L_{\mathcal{G}}$ is $2^{\mathbb{E}\mathsf{xpr}}$, where $\mathbb{E}\mathsf{xpr}$ is the set of all expressions occurring in $\mathcal{G}$, $\sqcap_{\mathcal{G}}$ is $\cap$, and $F_{\mathcal{G}}$ consists of functions $f$ such that $f(X) = (X - \mathit{Kill}) \cup \mathit{Gen}$ for arbitrary subsets $\mathit{Kill}$ and $\mathit{Gen}$ of $\mathbb{E}\mathsf{xpr}$. When $\mathit{Kill} = \mathit{Gen} = \emptyset$, $f$ is the identity function. $\square$

**OBSERVATION 3.5** *Bit vector frameworks are distributive, i.e., if the flow functions $f : L \mapsto L$ of a framework can be expressed as $f(x) = (x - \mathit{Kill}) \cup \mathit{Gen}$ where $\mathit{Kill}, \mathit{Gen} \in L$, then*

$$\forall x, y \in L : \; f(x \sqcap y) = f(x) \sqcap f(y)$$

*It follows that bit vector frameworks are also monotonic.*

**DEFINITION 3.18** *An instance of a data flow framework is an instantiation of the framework to a particular CFG. It is a pair* $\langle \mathbb{G}, M_{\mathbb{G}} \rangle$ *where*

- $\mathbb{G} = \langle \mathbb{N}\mathsf{odes}, \mathbb{E}\mathsf{dges} \rangle$ *is an instance of* $\mathcal{G}$*. This yields concrete values* $L_{\mathbb{G}}$, $\sqcap_{\mathbb{G}}$ *and* $F_{\mathbb{G}}$ *for* $L_{\mathcal{G}}$, $\sqcap_{\mathcal{G}}$ *and* $F_{\mathcal{G}}$*.*

- $M_{\mathbb{G}}$ *is a mapping from blocks in* $\mathbb{G}$ *to* $F_{\mathbb{G}}$*.*

***Example 3.7***
An instance of the available expression analysis is a pair consisting of a concrete CFG $\mathbb{G}$ and a mapping function $M_{\mathbb{G}}$. A basic block consisting of a single statement $a = b * c$ would be mapped to the following function by $M_{\mathbb{G}}$.

$$f(X) = X - \mathbb{E}\mathsf{xpr}_a \cup \{b * c\}$$

where $\mathbb{E}\mathsf{xpr}_a$ is the set of all expressions in $\mathbb{G}$ that have $a$ as an operand. $\square$

***Example 3.8***
As a more interesting example, consider the *may* alias problem for a CFG $\mathbb{G}$. The goal here is to find at each program point the set of pointer variables whose values are the same, i.e., they point to the same location. The result of this analysis is used to sharpen the effect of optimizations in the presence of pointers. As an example, the fact that $a$ and $b$ are *not may* aliased ensures that the assignment $*b = 5$ does not kill the expression $*a + c$. Thus the expression $*a + c$ can be discovered as a common sub-expression.

- The meet semilattice $L_{\mathbb{G}}$ consists of sets of pairs $e_1 \doteq e_2$, where $e_1$ and $e_2$ are pointer expressions. The data flow value at a program point $p$ containing this pair indicates a possible aliasing of the expressions $e_1$ and $e_2$ at $p$.

- Since a larger set of *may* aliases represent safer approximation by disabling more optimization opportunities, the partial order is: $X \sqsubseteq_{\mathbb{G}} Y$ iff $X \supseteq Y$. Thus $\sqcap_{\mathbb{G}}$ is $\cup$.

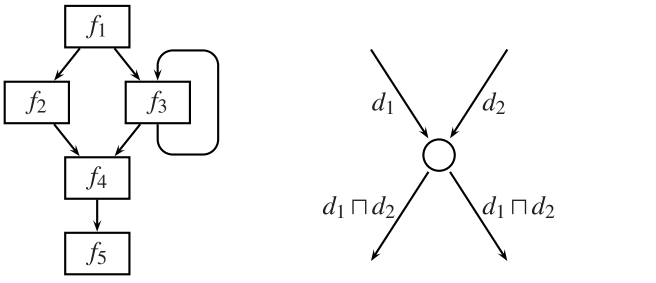
- Apart from the identity function, $F_{\mathbb{G}}$ consists of functions $f$ such that $f(X) = X - \mathit{Kill}(X) \cup \mathit{Gen}(X)$. Notice that unlike available expressions analysis the *Kill* and *Gen* sets are dependent on $X$.

Consider a basic block consisting of a single assignment statement $*x = y$. $\mathit{Kill}(X)$ consists of the set of pairs in $X$, one of whose components has $*x$ as a prefix.[†] $\mathit{Gen}(X)$ consists of all pairs $(*e_1 \doteq e_2)$ such that $e_1 \doteq x$ and $e_2 \doteq y$ in $X$. $\square$

## 3.3 Data Flow Assignments

Given an instance of a data flow framework, the desired data flow information is represented by the values of data flow variables $\mathit{In}_n$ for every node $n$. We define

[†]A more precise definition of $\mathit{Kill}(X)$ would include all those pairs in $X$, one of whose components has a prefix that is *must* aliased to $*x$.



(a) Example CFG (b) Merging information at a join node

**FIGURE 3.7**
Example to illustrate *MOP* assignment value and fixed point assignment.

a *data flow assignment* (or simply *assignment*) as a mapping from each data flow variable $\mathit{In}_n$ to a data flow value.

## 3.3.1 Meet Over Paths Assignment

Let $\mathit{paths}(p)$ denote the set of paths from *Start* to $p$. Given a path $\rho \in \mathit{paths}(p)$ consisting of basic blocks $(n_1, n_2 \ldots n_i)$, let $f_\rho$ denote the composition of functions corresponding to the blocks in $\rho$, i.e., $f_\rho = f_{n_{i-1}} \circ \ldots \circ f_2 \circ f_1$. If $\rho$ is a path $(n)$ consisting of a single block, $f_\rho$ is the identity function.

**DEFINITION 3.19** *An assignment represented by the values of data flow variables* $\mathit{In}_n$ *is safe iff*

$$\forall n \in \mathbb{N}\mathrm{odes}: \ \mathit{In}_n \sqsubseteq \bigsqcap_{\rho \in \mathit{paths}(n)} f_\rho(\mathit{BI}) \tag{3.6}$$

Observe that the informal definitions of analyses (2.1), (2.2) and (2.3) in Chapter 2 have been given in terms of paths from *Start* to $p$.

**DEFINITION 3.20** *A Meet Over Paths assignment, denoted* *MOP*, *is the maximum safe assignment.*

$$\forall n \in \mathbb{N}\mathrm{odes}: \ \mathit{MOP}_n = \bigsqcap_{\rho \in \mathit{paths}(n)} f_\rho(\mathit{BI}) \tag{3.7}$$

The existence of a *MOP* assignment follows from the closure and monotonicity properties of flow functions and the descending chain condition of the lattice of data flow values. A safe assignment is an approximation of the MOP assignment.

### 3.3.2 Fixed Point Assignment

Observe that the definition of the *MOP* assignment as the desired data flow information is a path-based definition whereas the data flow equations such as (3.5) form an edge-based specification: Data flow information of a node is computed from the data flow information at the predecessors.

***Example 3.9***
Consider Figure 3.7(a). The data flow information at the beginning of node 5 can be characterized by the following equations.

$$\begin{aligned} In_1 &= BI \\ In_2 &= f_1(In_1) \\ In_3 &= f_1(In_1) \sqcap f_3(In_3) \\ In_4 &= f_2(In_2) \sqcap f_3(In_3) \\ In_5 &= f_4(In_4) \end{aligned}$$

Unfolding the right hand side of $In_5$ partially, we get:

$$f_4(f_2(f_1(BI)) \sqcap (f_3(f_1(BI) \sqcap f_3(In_3)))) \tag{3.8}$$

The expression, represented as a tree in Figure 3.8(a), gives an idea of the nature of the solution of the equations. The solution computed by data flow equations at $p$ consider all paths to $p$ starting from the *Start* block and computes the data flow information along all these paths. However it merges the information at join nodes as shown in part (b) of Figure 3.7 on the previous page. The data flow information $d_1$ and $d_2$ is merged at the join node and the merged information $d_1 \sqcap d_2$ is propagated along all edges beyond the join node.

In contrast, the computation of *MOP* assignment does not involve merging values at intermediate points as shown in part (b) of Figure 3.8 on the facing page. ▯

As we shall see, merging is important for the existence of an algorithm for obtaining a solution. However it can also imply a potential loss of information.

To investigate whether the system of equations described by (3.5) have a solution, we first convert it into a single equation. The equations are of the form:

$$\begin{aligned} In_1 &= \mathbf{f}_1(In_1, \ldots, In_N) \\ In_2 &= \mathbf{f}_2(In_1, \ldots, In_N) \\ &\cdots \\ In_N &= \mathbf{f}_N(In_1, \ldots, In_N) \end{aligned}$$

where $In_i \in L_i$. Let the product lattice $L_1 \times L_2 \times \ldots L_N$ be denoted by $\overrightarrow{\mathbf{L}}$. Observe the difference between $f_i$ and $\mathbf{f}_i$. $f_i \in F : L_i \mapsto L_i$ is a flow function, whereas $\mathbf{f}_i : \overrightarrow{\mathbf{L}} \mapsto L_i$

$$\begin{tikzcd}
 & f_4 \arrow[d] & & \\
 & \cap \arrow[dl] \arrow[dr] & & \\
f_2 \arrow[d] & & f_3 \arrow[d] & \\
f_1 \arrow[d] & & \cap \arrow[dl] \arrow[dr, dash] & \\
\mathit{BI} & f_1 \arrow[d] & & f_3 \arrow[d] \\
 & \mathit{BI} & & \mathit{In}_3
\end{tikzcd}$$

(a) Expression tree for *MFP*

$$\begin{tikzcd}
 & & \cap \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
f_4 \arrow[d] & f_4 \arrow[d] & f_4 \arrow[d] & f_4 \arrow[d] & \cdots \\
f_2 \arrow[d] & f_3 \arrow[d] & f_3 \arrow[d] & f_3 \arrow[d] & \\
f_1 \arrow[d] & f_1 \arrow[d] & f_3 \arrow[d] & f_3 \arrow[d] & \\
\mathit{BI} & \mathit{BI} & f_1 \arrow[d] & f_3 \arrow[d] & \\
 & & \mathit{BI} & f_1 \arrow[d] & \\
 & & & \mathit{BI} &
\end{tikzcd}$$

(b) Expression tree for *MOP*

**FIGURE 3.8**
Unfoldings of $\mathit{In}_5$.

is formed by composing flow functions and the meet operator. The system of simultaneous equations can be rewritten as the single equation

$$\overrightarrow{\boldsymbol{In}} = \overrightarrow{\boldsymbol{f}}(\overrightarrow{\boldsymbol{In}}) \tag{3.9}$$

where $\overrightarrow{\boldsymbol{In}} \in \overrightarrow{\boldsymbol{L}}$ and $\overrightarrow{\boldsymbol{f}} : \overrightarrow{\boldsymbol{L}} \mapsto \overrightarrow{\boldsymbol{L}}$ is defined as

$$\overrightarrow{\boldsymbol{f}}(\overrightarrow{\boldsymbol{In}}) = \left\langle \boldsymbol{f}_1(\overrightarrow{\boldsymbol{In}}), \boldsymbol{f}_2(\overrightarrow{\boldsymbol{In}}), \ldots \boldsymbol{f}_N(\overrightarrow{\boldsymbol{In}}) \right\rangle$$

A solution of Equation (3.9) represents the data flow information computed by solving data flow equations.

***DEFINITION 3.21*** *A fixed point of a function $f : L \mapsto L$ is a value $v \in L$ that satisfies $f(v) = v$.*

A *fixed point* assignment is a solution of the data flow equations represented by (3.9). For a fixed point assignment $\mathit{FP}$, we denote the value of variable $\mathit{In}_n$ by $\mathit{FP}_n$. The *maximum fixed point* assignment is a fixed point assignment $\mathit{MFP}$ such that for any fixed point assignment $\mathit{FP}$,

$$\forall n \in \mathbb{N}\text{odes} : \ \mathit{FP}_n \sqsubseteq \mathit{MFP}_n$$

## 3.3.3 Existence of Fixed Point Assignment

The set of all fixed points of $f$ is denoted by $\mathit{fix}(f)$. We are interested in the existence and structure of $\mathit{fix}(\overrightarrow{\boldsymbol{f}})$ where $\overrightarrow{\boldsymbol{f}}$ is the function used for defining Equation (3.9). We require $\vec{f}$ to be monotonic; this in turn depends on the monotonicity of the flow functions in the data flow framework.

The desired properties of $\mathit{fix}(\vec{f})$ follow from the Knaster-Tarski fixed point theorem which we present below in a general setting.

***DEFINITION 3.22*** *Consider a monotonic function $f : L \mapsto L$. A value $v \in L$ is a* ***reductive point*** *of $f$ iff $f(v) \sqsubseteq v$. A value $v$ is an* ***extensive point*** *of $f$ iff $f(v) \sqsupseteq v$.*

The set of all reductive points of a function is denoted as $\mathit{red}(f)$ and the set of all extensive points of a function is denoted as $\mathit{ext}(f)$.

***THEOREM 3.1 (Knaster-Tarski fixed point theorem)***
*Let $f : L \mapsto L$ be a monotonic function on a complete lattice L. Then*

1. $\bigsqcap \mathit{red}(f) \in \mathit{fix}(f)$ *and* $\bigsqcap \mathit{fix}(f) = \bigsqcap \mathit{red}(f)$.
2. $\bigsqcup \mathit{ext}(f) \in \mathit{fix}(f)$ *and* $\bigsqcup \mathit{fix}(f) = \bigsqcup \mathit{ext}(f)$.
3. $\mathit{fix}(f)$ *is a complete lattice.*

**PROOF**

1. Let $\bigsqcap \mathit{red}(f)$ be $l$. We first prove that $l$ is a fixed point, i.e., $f(l) = l$. To show $f(l) \sqsubseteq l$, consider any element $x \in \mathit{red}(f)$. Since $l \sqsubseteq x$, $f(l) \sqsubseteq f(x)$ because of monotonicity of $f$. Further, since $x \in \mathit{red}(f)$, $f(x) \sqsubseteq x$. Therefore $f(l) \sqsubseteq x$. Since $x$ was an arbitrary element in $\mathit{red}(f)$, $f(l) \sqsubseteq l$ by Observation 3.2.

   We now show $l \sqsubseteq f(l)$. Interestingly, this can be derived from $f(l) \sqsubseteq l$. Because of monotonicity, $f(f(l)) \sqsubseteq f(l)$. Thus $f(l)$ is a reductive point of $\mathit{red}(f)$. Since $l$ is $\bigsqcap \mathit{red}(f)$, we have $l \sqsubseteq f(l)$.

   Since $\mathit{fix}(f) \subseteq \mathit{red}(f)$, $\bigsqcap \mathit{red}(f)$ is a lower bound of $\mathit{fix}(f)$. Further, since $\bigsqcap \mathit{red}(f) \in \mathit{fix}(f)$, $\bigsqcap \mathit{red}(f) = \bigsqcap \mathit{fix}(f)$.

2. Similar to 1.

3. Consider any arbitrary subset $Y$ of $\mathit{fix}(f)$. It is enough to show that $\bigsqcap Y$ exists in $\mathit{fix}(f)$. Let $X = \{x \mid x \sqsubseteq \bigsqcap Y, x \in L\}$. Since $L$ is a complete lattice, it is easy to see that $X$ is a complete lattice with $\bigsqcap Y$ as the top element and the bottom of $L$ as the bottom element of $X$. Now consider a restriction of $f$ to $X$ called $f'$. $f'$ is a monotonic function on the complete lattice $X$. Clearly $\mathit{fix}(f') \subseteq \mathit{fix}(f)$. Further, $\mathit{fix}(f') \subseteq X$. Thus every fixed point of $f'$ is weaker than $\bigsqcap Y$. Since $\mathit{fix}(f') \subseteq \mathit{fix}(f)$, $\bigsqcap Y$ is contained in $\mathit{fix}(f)$.

▮

***Input:*** An instance $(\mathbb{G}, M_{\mathbb{G}})$ of a monotone data flow framework $(L_{\mathcal{G}}, \sqcap_{\mathcal{G}}, F_{\mathcal{G}})$. The function to which $M_{\mathbb{G}}$ maps a node $n$ is denoted as $f_n$. The *Start* node is numbered 0. The rest of the nodes are arbitrarily ordered from 1 to $N-1$.
***Output:*** $In_k, 0 \le k \le N-1$ giving the output of the data flow analysis for each node.
***Algorithm:***

```
 0  function dfaMain()
 1  {  $In_0 = BI$
 2     for all $j$, $j \neq 0$ do $In_j = \top$
 3     $change = true$
 4     while $change$ do
 5     {  $change = false$
 6        for $j = 1$ to $N-1$ do
 7        {  $temp = \bigsqcap_{p \in pred(j)} f_p(In_p)$
 8           if $temp \neq In_j$ then
 9           {  $In_j = temp$
10              $change = true$
11           }
12        }
13     }
14  }
```

**FIGURE 3.9**
Round-robin iterative algorithm for computing *MFP* assignment for frameworks with a complete lattice.

## 3.4 Computing Data Flow Assignments

Given a complete lattice and a monotonic function defining data flow equations (which, in our case, is $\overrightarrow{f}$), the Knaster-Tarski fixed point theorem guarantees existence of fixed points. In this section we present an algorithm for computing the *MFP* assignment and show the computability of *MFP* assignment and undecidability of *MOP* assignment.

### 3.4.1 Computing *MFP* Assignment

Figure 3.9 provides an algorithm to solve the data flow equations. The iterations of lines 7-12 can be indexed using a pair $(i, j)$, where $i$, starting with 1, is the iteration number of the **while** loop, and $j$ is the iteration number (the index) of the **for** loop. Given an iteration $(i, j)$, we shall denote the next iteration in lexicographical ordering as $\mathcal{N}(i, j)$ and the value of $In_m$ before the $(i, j)$th iteration as $In_m^{(i,j)}$.

**LEMMA 3.2**

*The algorithm shown in Figure 3.9 terminates.*

**PROOF** We shall first show that the value of a data flow variable decreases across successive iterations. In other words, for all $m$

$$\mathit{In}_m^{(i,j)} \sqsupseteq \mathit{In}_m^{\mathcal{N}(i,j)} \tag{3.10}$$

This must be true for $m = 0$ for all $(i, j)$ as its value remains constant at $\mathit{BI}$. For other values of $m$, we show (3.10) by induction on the iteration count $(i, j)$.

*Basis:* True, because the value of $\mathit{In}_m^{\mathcal{N}(1,1)}$ is $\top$.

*Inductive Step:* Assume as the inductive hypothesis that $\mathit{In}_m^{(i,j)} \sqsupseteq \mathit{In}_m^{\mathcal{N}(i,j)}$ for all $m$. From monotonicity, it follows that

$$\forall m \in \mathbb{N}\mathrm{odes} : f_m\left(\mathit{In}_m^{(i,j)}\right) \sqsupseteq f_m\left(\mathit{In}_m^{\mathcal{N}(i,j)}\right) \tag{3.11}$$

We have to show that

$$\forall m \in \mathbb{N}\mathrm{odes} : \ \mathit{In}_m^{\mathcal{N}(i,j)} \sqsupseteq \mathit{In}_m^{\mathcal{N}(\mathcal{N}(i,j))} \tag{3.12}$$

The second component of $\mathcal{N}(i, j)$ gives the block number whose data flow variable is examined on line 8 in $\mathcal{N}(i, j)$th iteration. We shall denote this block number as $l$. If this is not the same as $m$, or, if the value of $\mathit{In}_m$ is the same since it was last examined, there is nothing to be proven. Otherwise, by lines 7 and 9 of the algorithm, the proof obligation (3.12) reduces to

$$\bigsqcap_{p \in \mathit{pred}(l)} f_p\left(\mathit{In}_p^{(i,j)}\right) \sqsupseteq \bigsqcap_{p \in \mathit{pred}(l)} f_p\left(\mathit{In}_p^{(\mathcal{N}(i,j))}\right) \tag{3.13}$$

The inductive step then follows from (3.11) and Observation 3.2.

The termination of the algorithm follows directly from (3.10) and the descending chain condition. ∎

We next show that algorithm (3.9) computes the $\mathit{MFP}$ assignment of the associated data flow equations.

**LEMMA 3.3**

*The algorithm in Figure 3.9 computes the* $\mathit{MFP}$ *assignment of the data flow equations represented by (3.5).*

**PROOF** The convergence of the algorithm implies that the values of $\mathit{In}$ found by the algorithm form a fixed point assignment of the equations represented by (3.5). We have to prove that it is the maximum fixed point by showing that for any other fixed point assignment $\mathit{FP}$, $\mathit{FP}_m \sqsubseteq \mathit{In}_m$ for every node $m$. We do this by showing that the value of $\mathit{In}_m$ computed at each step

$(i, j)$ of the algorithm is stronger than $\mathit{FP}_m$. This is true of $\mathit{FP}_0$ and $\mathit{In}_0$ since the value of $\mathit{FP}_0$ is $\mathit{BI}$ and so is the value of $\mathit{In}_0$ in each step of the algorithm. We therefore prove the lemma for values of $m$ other than 0. The proof is by induction on $(i, j)$.

*Basis:* Follows from the fact that $\mathit{In}_m(1,1) = \top$

*Inductive step:* We have to show that $\mathit{FP}_m \sqsubseteq \mathit{In}_m{}^{\mathcal{N}(i,j)}$. Since $\mathit{FP}$ is a fixed point assignment of Equation (3.5), $\mathit{FP}_m = \bigsqcap_{p \in \mathit{pred}(m)} f_p(\mathit{FP}_p)$. Further, from line 7 of the algorithm, $\mathit{In}_m{}^{\mathcal{N}(i,j)} = \bigsqcap_{p \in \mathit{pred}(m)} f_p\left(\mathit{In}_p{}^{(i,j)}\right)$. Therefore we have to show that

$$\bigsqcap_{p \in \mathit{pred}(m)} f_p(\mathit{FP}_p) \sqsubseteq \bigsqcap_{p \in \mathit{pred}(m)} f_p\left(\mathit{In}_p{}^{(i,j)}\right) \tag{3.14}$$

This once again follows from the induction hypothesis, monotonicity of the flow functions and Observation 3.2. ∎

## 3.4.2 Comparing *MFP* and *MOP* Assignments

In this section we show that the *MFP* assignment computed by the algorithm in Figure 3.9 is weaker than the *MOP* assignment. We also show examples of frameworks in which the *MFP* is strictly weaker than the *MOP* assignment.

**LEMMA 3.4**

*When the algorithm in Figure 3.9 terminates,* $\forall m \in \mathbb{N}\mathit{odes}$, $\mathit{In}_m \sqsubseteq \mathit{MOP}_m$.

**PROOF** Let $\mathit{paths}_l(m)$ denote the set of all paths of length $l$ from *Start* to $m$. We want to show by induction on $l$ that $\mathit{In}_m \sqsubseteq \bigsqcap_{\rho \in \mathit{paths}_l(m)} f_\rho(\mathit{BI})$.

*Basis:* $l = 1$. In this case the path being considered has the single node *Start*. The lemma holds because $\mathit{In}_0$, which represents the data flow value at the beginning of *Start* is held constant at *BI*.

*Inductive step:* We have to show that

$$\mathit{In}_m \sqsubseteq \bigsqcap_{\rho \in \mathit{paths}_l(m)} f_\rho(\mathit{BI}) \tag{3.15}$$

From Equation (3.5), we also have

$$\mathit{In}_m = \bigsqcap_{p \in \mathit{pred}(m)} f_p(\mathit{In}_p)$$

and as the induction hypothesis, we can assume that for all $p \in \mathit{pred}(m)$,

$$\mathit{In}_p \sqsubseteq \bigsqcap_{\rho \in \mathit{paths}_{l-1}(p)} f_\rho(\mathit{BI})$$

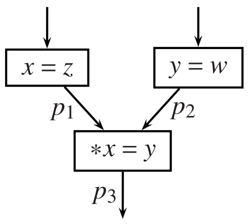


**FIGURE 3.10**
CFG illustrating the non-distributivity of *may* alias framework.

Monotonicity of flow functions gives

$$f_p(\mathit{In}_p) \sqsubseteq f_p\left(\bigsqcap_{\rho \in \mathit{paths}_{l-1}(p)} f_\rho(\mathit{BI})\right)$$

And from Observation 3.3,

$$f_p(\mathit{In}_p) \sqsubseteq \bigsqcap_{\rho \in \mathit{paths}_{l-1}(p)} f_p\left(f_\rho(\mathit{BI})\right)$$

Since $\rho$ is a path of length $l-1$ and $p$ is a predecessor of $m$, the composition $f_p \circ f_\rho$ corresponds to a path of length $l$ reaching $m$ and

$$f_p(\mathit{In}_p) \sqsubseteq \bigsqcap_{\rho \in \mathit{paths}_l(m)} f_\rho(\mathit{BI})$$

Therefore

$$\mathit{In}_m = \bigsqcap_{p \in \mathit{pred}(m)} f_p(\mathit{In}_p) \sqsubseteq \bigsqcap_{\rho \in \mathit{paths}_l(m)} f_\rho(\mathit{BI}) \sqsubseteq \mathit{MOP}_m$$

▮

We now show that for some data flow frameworks, the *MFP* assignment is strictly weaker than the *MOP* assignment.

***Example 3.10***
Consider a CFG fragment shown in Figure 3.10 as an instance of the *may* alias analysis framework. The data flow values at $p_1$ and $p_2$ are $\{x \doteq z\}$ and $\{y \doteq w\}$. While computing the *MFP* assignment, these data flow values will be merged to obtain the value $\{x \doteq z, y \doteq w\}$ at the input of the block containing the assignment $*x = y$. The flow function of this assignment will add to this value the cross product of all aliases of $*x$ and all aliases of $y$. The data flow value at $p_3$ is thus $\{x \doteq z, y \doteq w, *x \doteq y, *x \doteq w, *z \doteq y, *z \doteq w\}$.

The *MOP* assignment on the other hand finds the effect of the assignment $*x = y$ on the incoming data flow values $\{x \doteq z\}$ and $\{y \doteq w\}$ separately. This yields the sets $\{x \doteq z, *x \doteq y, *z \doteq y\}$ and $\{*x \doteq y, *x \doteq w, y \doteq w\}$. The value at $p_3$ is a union of the two sets, and this is clearly stronger than the corresponding *MFP* value. The *MFP* value includes an alias $*z \doteq w$ which is not possible along any execution path. □

In Chapter 4 we will see other examples of data flow frameworks for which the *MFP* assignment is strictly weaker than *MOP* assignment. We now show that the *MFP* and the *MOP* assignments coincide for distributive frameworks.

**LEMMA 3.5**
*For distributive frameworks,* $\forall m \in \mathbb{N}\text{odes},\ In_m = MOP_m$.

**PROOF** We replay the proof of Lemma 3.4 with $\sqsubseteq$ substituted by $=$ in (3.15). As the induction hypothesis, we can assume that for all $p \in pred(m)$,

$$In_p = \bigsqcap_{\rho \in paths_{l-1}(p)} f_\rho(BI)$$

Applying $f_p$ to both sides of the equation, we have:

$$f_p(In_p) = f_p\left(\bigsqcap_{\rho \in paths_{l-1}(p)} f_\rho(BI)\right)$$

Because $f_p$ is distributive, we have

$$f_p(In_p) = \bigsqcap_{\rho \in paths_{l-1}(p)} f_p\left(f_\rho(BI)\right)$$

which simplifies to

$$f_p(In_p) = \bigsqcap_{\rho \in paths_l(m)} f_\rho(BI)$$

Therefore

$$In_m = \bigsqcap_{p \in pred(m)} f_p(In_p) = \bigsqcap_{\rho \in paths_l(m)} f_\rho(BI)$$

▮

### 3.4.3 Undecidability of *MOP* Assignment Computation

We have seen that if a framework is not distributive, then the algorithm shown in Figure 3.9 on page 79 may produce a solution which is strictly weaker than the *MOP* value. Thus it is interesting to investigate whether there exists an algorithm which can compute the *MOP* of an arbitrary monotone data flow framework. We now show that the problem of finding the *MOP* value of a monotone data flow framework is undecidable. To do this we reduce an instance of an undecidable problem called the *Modified Post's Correspondence Problem (MPCP)* to an instance of a monotone data flow framework. MPCP is a decision problem defined as follows:

***DEFINITION 3.23*** *Given lists* $A = [a_1, a_2, \dots, a_k]$ *and* $B = [b_1, b_2, \dots, b_k]$*, where* $a_i$ *and* $b_i$ *are strings of 0's and 1's, is there an index list* $[1, i_1, i_2, \dots, i_r]$ *such that* $a_1 a_{i_1} a_{i_2} \dots a_{i_r} = b_1 b_{i_1} b_{i_2} \dots b_{i_r}$*?*

In the above definition, juxtaposition of strings denotes their concatenation. Given an instance $I$ of MPCP, we convert it into an instance of a monotone data flow framework as follows:

- The meet semilattice $L$ of data flow values consists of lists of integers between 1 and $k$. These play the role of index lists. The semilattice also includes $\perp$ and the special element \$ indicating that the instance of MPCP has no solution.
- The relation $\sqsubseteq_I$ is defined as $x \sqsubseteq_I y$ *iff* $x = y$ *or* $x = \perp$.
- The set of flow functions $F$ is formed by composing the following functions:
    1. The identity function *id*.
    2. A class of functions $f_i, 1 \leq i \leq k$, such that:

    $$f_i(\alpha) = \begin{cases} \perp, & \alpha \text{ is } \perp \\ \$, & \alpha \text{ is } \$ \\ \alpha\#i & \text{otherwise} \end{cases}$$

    $\alpha\#i$ extends the index list $\alpha$ by adding the integer $i$ at the end.
    3. A function $g$ such that

    $$g(\alpha) = \begin{cases} \perp, & \text{the index list } \alpha \text{ is a solution of the MPCP instance } I \\ \$ & \text{otherwise} \end{cases}$$

- *BI* is the singleton list containing 1.
- The CFG is shown in Figure 3.11.

***LEMMA 3.6***
*The data flow framework defined above is monotone.*

**PROOF** Obvious. ▮

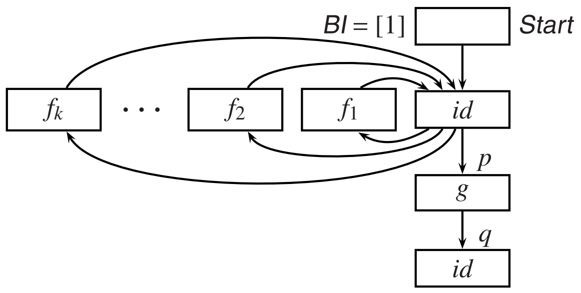


**FIGURE 3.11**
CFG for showing undecidability of *MOP* computation.

**THEOREM 3.1**

*The problem of finding the MOP assignment for any monotone data flow framework is undecidable.*

**PROOF** Given an MPCP instance $I$, we define an instance of a monotone data flow framework using the above construction. Each path to the program point $p$ generates a distinct index list as the data flow value. Conversely, for each possible index list there is a path to $p$ that generates the list. The function $g$ checks whether each of these lists is a possible solution of the MPCP instance. Therefore, the *MOP* value at the program point $q$ is \$ iff there is a solution to the MPCP instance, and $\perp$ otherwise. If an algorithm to compute the *MOP* assignment existed, we could use it to find a solution of the MPCP instance $I$. However, since MPCP is known to be undecidable, the problem of finding *MOP* for any monotone data flow framework is also undecidable. ∎

## 3.5 Complexity of Data Flow Analysis for Rapid Frameworks

Recall that the *MFP* algorithm presented in Figure 3.9 on page 79 does not assume an a-priori order in which nodes of the input CFG are visited during an iteration. In order to estimate the complexity of data flow analysis, we now consider a specialization of the *MFP* algorithm in which the nodes of the CFG are visited in reverse postorder. We also consider special properties of data flow frameworks that make the algorithm amenable to complexity analysis.

### 3.5.1 Properties of Data Flow Frameworks

Section 3.2.3 presented monotonicity and distributivity properties of data flow frameworks. They are related to the convergence of the *MFP* algorithm and characterize the data flow assignment computed by the *MFP* algorithm. In this section, we present properties of data flow frameworks based on algorithmic complexity.

**DEFINITION 3.24** *A monotone data flow framework is k-bounded iff*

$$\exists\, k \geq 1 \; s.t.\; \forall\, f \in F, \forall x \in L:\; f^0(x) \sqcap f^1(x) \sqcap f^2(x) \sqcap \cdots = \bigsqcap_{i=0}^{k-1} f^i(x) \tag{3.16}$$

*where $f^0$ is the identity function and $f^{j+1} = f \circ f^j$.*

The unbounded expression $f^0(x) \sqcap f^1(x) \sqcap f^2(x) \sqcap \cdots$ represents the *glb* of the data flow value computed in all possible traversals over a loop and is called the *loop closure* of $f$. Since we require $L$ to satisfy the descending chain condition, all loop closures are bounded. For a framework to be $k$-bounded, the loop closures must be bounded by a constant $k$.

A 2-bounded framework is called a *fast* framework. It can be shown that a framework is fast iff

$$\forall f \in F, \forall x \in L \;:\; f^2(x) \sqsupseteq x \sqcap f(x) \tag{3.17}$$

Intuitively, a single traversal over a loop is sufficient for computing loop closure.

**LEMMA 3.7**
*Bit vector frameworks are fast.*

**PROOF** Recall that the flow functions in bit vector frameworks can be expressed as $f(x) = (x - \mathit{Kill}) \cup \mathit{Gen}$ where $\mathit{Kill}, \mathit{Gen} \in L$. For such functions,

$$\begin{aligned} f^2(x) &= f\,((x - \mathit{Kill}) \cup \mathit{Gen}) \\ &= (((x - \mathit{Kill}) \cup \mathit{Gen}) - \mathit{Kill}) \;\cup\; \mathit{Gen} \\ &= (x - \mathit{Kill}) \;\cup\; (\mathit{Gen} - \mathit{Kill}) \;\cup\; \mathit{Gen} \\ &= (x - \mathit{Kill}) \;\cup\; \mathit{Gen} \\ &= f(x) \end{aligned}$$

This implies $f^2(x) \sqsupseteq x \sqcap f(x)$. ∎

There is an important subclass of fast frameworks called *rapid* frameworks in which traversing the loop independently of the value at the entry of the loop is sufficient for computing the final value at the entry of the loop.

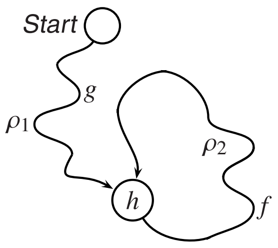


**FIGURE 3.12**
The significance of the rapidity condition.

***DEFINITION 3.25*** *A data flow framework is rapid, iff*

$$(\forall f, g \in F)\,(\forall x, \mathit{BI} \in L)\ :\ f(g(\mathit{BI})) \sqsupseteq g(\mathit{BI}) \sqcap f(x) \sqcap x \tag{3.18}$$

The condition is significant for paths which include loops. Figure 3.12 shows an example of such a path whose initial segment $\rho_1$ is from the *Start* node to the header $h$ of a loop. The second segment $\rho_2$ is from $h$ back to $h$ along the looping path. The flow functions corresponding to the two segments are $g$ and $f$. The result of $f(g(\mathit{BI}))$ represents the data flow value at $h$ along the path $\rho_1\rho_2$. The rapidity condition says that this is safely approximated by combining the data flow value generated along $\rho_1$ and the value obtained by traversing the loop with *any* data flow $x$ available at $h$. The important point is that the data value $g(\mathit{BI})$ may take several iterations to reach $h$ from *Start* because of the presence of back edges in $\rho_1$. However, the rapidity condition ensures that it is enough to traverse the loop with a data flow value that has reached $h$ earlier, say, through a back edge free path from *Start* to $h$.

We now state and prove a condition that is equivalent to Condition (3.18).

***LEMMA 3.8***
*The rapid condition (3.18) is equivalent to:*

$$(\forall f \in F)\,(\forall x, y \in L)\ :\ f(y) \sqsupseteq y \sqcap f(x) \sqcap x \tag{3.19}$$

**PROOF** It is easy to derive (3.18) from (3.19). If (3.19) holds for any $y$, in particular it holds for values expressible as $g(\mathit{BI})$ for any choice of $g$ and $\mathit{BI}$. To derive (3.19) from (3.18), we show that for arbitrary $f$, $y$ and $x$,

$$f(y) \sqsupseteq y \sqcap f(x) \sqcap x$$

Since any value $y$ can be expressed as $\bigsqcap_{i=0}^{k} g_i(\mathit{BI})$, our proof obligation becomes:

$$f\Big(\bigsqcap_{i=0}^{k} g_i(\mathit{BI})\Big) \sqsupseteq \bigsqcap_{i=0}^{k} g_i(\mathit{BI}) \sqcap f(x) \sqcap x$$

$$\begin{tikzcd}\top \arrow[d] \\ v_1 \arrow[d] \\ \bot\end{tikzcd}$$

(a) Lattice $L$

- Let $f(\top) = \top$, $f(v1) = \bot$, and $f(\bot) = \bot$.
- $f$ is fast because $\forall x \in L, \forall i \geq 2 : \; f^i(x) \sqsupseteq f(x) \sqcap x$
- $f$ is not rapid because, $f(v_1) \not\sqsupseteq v_1 \sqcap f(\top)$

(b) Flow function

**FIGURE 3.13**
A fast function need not be rapid.

Because of monotonicity, this is the same as

$$\prod_{i=0}^{k} f(g_i(BI)) \sqsupseteq \prod_{i=0}^{k} g_i(BI) \sqcap f(x) \sqcap x \tag{3.20}$$

Because of (3.18), $f(g_i(BI)) \sqsupseteq g_i(BI) \sqcap f(x) \sqcap x$ holds for each $i$. Therefore (3.20) holds because of Observation 3.2. ■

A consequence of this condition is that it is not necessary for an algorithm to traverse a loop twice. If $x$ is taken as the value at $h$ before iterating over the loop, then setting $y$ to $f(x)$ we have

$$\begin{aligned} f(f(x)) &\sqsupseteq f(x) \sqcap f(x) \sqcap x \\ &\sqsupseteq f(x) \sqcap x \end{aligned}$$

We have just shown that every rapid framework is fast. To show that fastness does not necessarily imply rapidity, it is sufficient to construct a framework with a flow function that is fast but is not rapid. Figure 3.13 defines such a function.

For complete lattices, the rapid condition can also be stated as

$$\forall z \in L, \forall f \in F : \; f(z) \sqsupseteq z \sqcap f(\top) \tag{3.21}$$

This is just an instance of (3.19). Observe this condition also has the same meaning: A loop can be analyzed independently of the incoming information.

We have already shown that bit vector frameworks are fast (Lemma 3.7). Now we show that they are also rapid.

***LEMMA 3.9***
*Bit vector frameworks are rapid.*

**PROOF** We first consider frameworks in which the $\sqsubseteq$ relation is $\subseteq$. For such frameworks, Condition (3.18) reduces to

$$(\forall f, g \in F)\ (\forall x, y \in L) : \; f(g(y)) \supseteq g(y) \cap f(x) \cap x$$

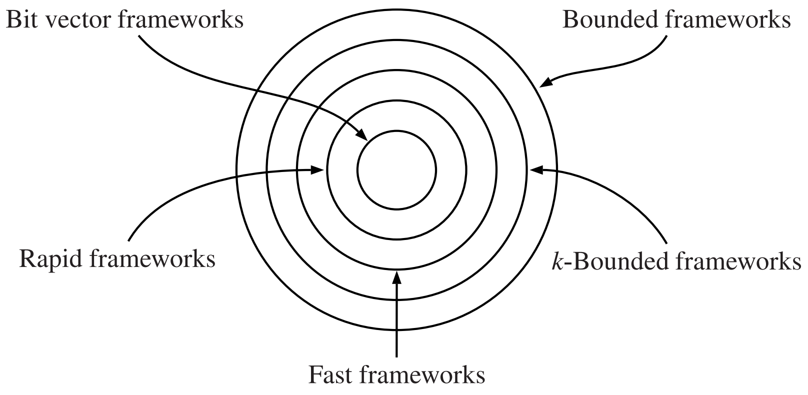


**FIGURE 3.14**
Monotone data flow frameworks with lattices containing $\top$ and satisfying the descending chain condition.

Let $g(y) = z$. Then our proof obligation becomes

$$(\forall f \in F)\,(\forall x, z \in L):\; f(z) \;\supseteq\; z \cap f(x) \cap x$$

The right hand side can be reduced to

$$\begin{aligned}
z \cap f(x) \cap x &= z \cap ((x - \mathit{Kill}) \cup \mathit{Gen}) \cap x \\
&= z \cap (x \cap (\neg \mathit{Kill}) \cup \mathit{Gen}) \cap x \\
&= (z \cap (\neg \mathit{Kill}) \cap x) \cup (\mathit{Gen} \cap z \cap x) \\
&\subseteq (z \cap (\neg \mathit{Kill})) \cup (\mathit{Gen}) \\
&\subseteq (z - \mathit{Kill}) \cup \mathit{Gen} \\
&\subseteq f(z)
\end{aligned}$$

Rapidity of frameworks in which $\sqsubseteq$ relation is $\supseteq$, can be shown similarly. ∎

Given arbitrary $x \in L$ and arbitrary $f \in F$, we have seen that for fast frameworks, $f^2(x) \sqsupseteq f(x) \sqcap x$ whereas for bit vector frameworks, $f^2(x) = f(x)$. We now show that the rapid frameworks satisfy the condition $f^2(x) \sqsupseteq f(x)$. This condition is stronger than the condition for fast frameworks but weaker that the condition for bit vector frameworks.

***LEMMA 3.10***
*If $f \in F$ is a flow function in a rapid framework and the underlying lattice is complete, then $\forall z \in L:\; f^2(z) \sqsupseteq f(z)$.*

**PROOF** Instantiating $g$ to $f$, $\mathit{BI}$ to $z$, and $x$ to $\top$ in the rapid condition (3.18), we have

***Input:*** An instance $(\mathbb{G}, M_{\mathbb{G}})$ of a distributive data flow framework $(L_{\mathcal{G}}, \sqcap_{\mathcal{G}}, F_{\mathcal{G}})$. The function to which $M_{\mathbb{G}}$ maps a node $n$ is denoted as $f_n$. The nodes are numbered from 1 to $N-1$ in reverse postorder.
***Output:*** $\mathsf{In}_k$, $0 \le k \le N-1$ giving the output of the data flow analysis for each node.
***Algorithm:***

```
 0  function dfaMain()
 1  {   $In_0 = BI$
 2      for $j = 1$ to $N-1$ do $In_j = \bigsqcap_{p \in pred(j) \wedge p<j} f_p(In_p)$
 3      $change = true$
 4      while $change$ do
 5      {   $change = false$
 6          for $j = 1$ to $N-1$ do
 7          {   $temp = \bigsqcap_{p \in pred(j)} f_p(In_p)$
 8              if $temp \neq In_j$ then
 9              {   $In_j = temp$
10                  $change = true$
11              }
12          }
13      }
14  }
```

**FIGURE 3.15**
An efficient and more general version of the *MFP* algorithm.

$$\begin{aligned} f^2(z) &\sqsupseteq f(z) \sqcap f(\top) \\ &\sqsupseteq f(z \sqcap \top) \sqsupseteq f(z) \end{aligned}$$

The second step follows from Observation 3.3 proving the lemma. ∎

Observe that conditions $f^2(x) \sqsupseteq f(x)$ and $f^2(x) = f(x)$ on rapid and bit vector frameworks respectively are only necessary conditions and are not sufficient. For the function $f$ defined in Figure 3.13 on page 88, $\forall x \in L: f^2(x) = f(x)$ and yet the framework is neither rapid nor bit vector. Figure 3.14 shows the relationship between various frameworks.

## 3.5.2 Complexity for General CFGs

The complexity analysis in this section is restricted to rapid frameworks that are distributive. The modified algorithm is shown in Figure 3.15. Note that the data flow variables are also initialized differently. Such an initialization obviates the need of the $\top$ element and allows handling frameworks with meet semilattices. This initialization has the effect of assigning to each node except *Start* with $\top$ and propagating the initial values by assuming $f(\top) = \top$.

We count the number of iterations of the algorithm as follows. The initialization of all data flow variables in the **for** loop in line 2 is counted as the first iteration. Following this, each iteration of the **while** loop is counted separately.

To prove the main complexity result, we need a couple of auxiliary results. The first result characterizes the data flow value at any program point after a given number of iterations in terms of paths containing a specified number of back edges. When the number of back edges in paths also needs to be denoted, we extend the notation $\mathit{paths}(j)$ to $\mathit{paths}^k(j)$ to denote the set of paths containing at most $k-1$ back edges.

**LEMMA 3.11**
*After k iterations of the algorithm, the data flow value at the entry of block j is given by* $\mathit{In}_j = \bigsqcap_{\rho \in \mathit{paths}^k(j)} f_\rho(\mathit{BI})$.

**PROOF** The proof is by induction on the number of iterations $k$.

- *Basis:* $k = 1$. This step corresponds to line 2 of the algorithm when we traverse only back edge free paths. To prove this case, we do an inner induction on the visiting order of nodes. The variable $j$ is used to denote the position of the nodes in this order.
  - *Basis:* $j = 0$. The node that is numbered 0 in the visiting order is *Start*. The relevant path in this case is (*Start*). Thus we have $\mathit{In}_0 = \mathit{BI} = f_{(\mathit{Start})}(\mathit{BI})$.
  - *Inductive step:* Recall that the nodes are visited in reverse postorder. Assume that the lemma holds for all nodes whose position in reverse postorder is less than $j$. Observe that back edge free paths from *Start* to $j$ consist of back edge free paths from *Start* to $p$, where $p < j$, followed by the forward edge $(p, j)$. Thus:

$$\begin{aligned}
\mathit{In}_j &= \bigsqcap_{p \in \mathit{pred}(j) \wedge p<j} f_p(\mathit{In}_p) \\
&= \bigsqcap_{p \in \mathit{pred}(j) \wedge p<j} f_p(\bigsqcap_{\rho \in \mathit{paths}^1(p)} f_\rho(\mathit{BI})) \\
&= \bigsqcap_{p \in \mathit{pred}(j) \wedge p<j} \; \bigsqcap_{\rho \in \mathit{paths}^1(p)} f_p(f_\rho(\mathit{BI})) \\
&= \bigsqcap_{\rho \in \mathit{paths}^1(j)} f_\rho(\mathit{BI})
\end{aligned}$$

- *Inductive step:* Assume that the lemma holds for $k-1$ iterations. We once again do an inner induction on the visiting order of nodes.
  - *Basis:* Trivial.

- *Inductive step:* Assume that the lemma holds for $k$ iterations for those nodes whose number in reverse postorder is less than $j$. Then:

$$\begin{aligned}
\mathit{In}_j &= \bigsqcap_{p \in \mathit{pred}(j)} f_p(\mathit{In}_p) \\
&= \left( \bigsqcap_{p \in \mathit{pred}(j) \wedge p<j} f_p(\mathit{In}_p) \right) \sqcap \left( \bigsqcap_{p \in \mathit{pred}(j) \wedge p \geq j} f_p(\mathit{In}_p) \right) \\
&= \left( \bigsqcap_{p \in \mathit{pred}(j) \wedge p<j} f_p\left( \bigsqcap_{\rho \in \mathit{paths}^k(p)} f_\rho(\mathit{BI}) \right) \right) \\
&\quad \sqcap \left( \bigsqcap_{p \in \mathit{pred}(j) \wedge p \geq j} f_p\left( \bigsqcap_{\rho \in \mathit{paths}^{k-1}(p)} f_\rho(\mathit{BI}) \right) \right) \\
&\quad \{\text{using induction hypothesis}\} \\
&= \left( \bigsqcap_{(p \in \mathit{pred}(j) \wedge p<j)} \; \bigsqcap_{(\rho \in \mathit{paths}^k(p))} f_p(f_\rho(\mathit{BI})) \right) \\
&\quad \sqcap \left( \bigsqcap_{(p \in \mathit{pred}(j) \wedge p \geq j)} \; \bigsqcap_{(\rho \in \mathit{paths}^{k-1}(p))} f_p(f_\rho(\mathit{BI})) \right) \\
&\quad \{\text{distributivity}\} \\
&= \bigsqcap_{\rho \in \mathit{paths}^k(j)} f_\rho(\mathit{BI})
\end{aligned}$$

The last step is justified because a path from *Start* to $j$ with $k$ back edges could either be made up of (i) a path with $k$ back edges from *Start* to $p$, where $p < j$, followed by a traversal along the forward edge $p \rightarrow j$, or (ii) a path from *Start* to $p$ with $k-1$ back edges, where $p \geq j$, followed by a traversal along the back edge $p \rightarrow j$.

Hence the lemma. ▮

Since $\mathit{paths}^{k-1}(j) \subset \mathit{paths}^k(j)$, the data flow value at any node decreases with increasing number of iterations. This is similar to the algorithm shown in Figure 3.9, and is crucial for the termination of the algorithm. Note that the algorithms have this property because of the choice of the initial values.

The second result relates the termination of the algorithm to the data flow values at program points.

**LEMMA 3.12**

*The algorithm terminates within $k$ iterations iff for each block $j$, each path $\rho$ in $\mathit{paths}(j)$ and any boundary value $\mathit{BI}$, there exists a finite set of paths $\rho_1, \ldots \rho_r$ from $\mathit{paths}^{k-1}(j)$ such that*

$$f_\rho(\mathit{BI}) \sqsupseteq \bigsqcap_{1 \leq i \leq r} f_{\rho_i}(\mathit{BI}) \tag{3.22}$$

**PROOF** *If part:* Let $\mathit{BI}$ be an arbitrary boundary value. Assume that

Condition (3.22) is satisfied after $k$ iterations. Since $\textsf{paths}^{k-1}(j) \subseteq \textsf{paths}^{k}(j)$,

$$\bigsqcap_{\rho\in\textsf{paths}^{k}(j)} f_\rho(\textsf{BI}) \sqsubseteq \bigsqcap_{\rho\in\textsf{paths}^{k-1}(j)} f_\rho(\textsf{BI}) \tag{3.23}$$

Further, following Condition (3.22), for each path $\rho \in \textsf{paths}^{k}(j)$ we have a finite set of paths $\rho_1,\ldots\rho_r$ from $\textsf{paths}^{k-1}(j)$ such that $f_\rho(\textsf{BI}) \sqsupseteq \bigsqcap_{1\leq i\leq r} f_{\rho_i}(\textsf{BI})$. Therefore $f_\rho(\textsf{BI}) \sqsupseteq \bigsqcap_{\rho'\in\textsf{paths}^{k-1}(j)} f_{\rho'}(\textsf{BI})$. Considering all paths in $\textsf{paths}^{k}(j)$ we have:

$$\bigsqcap_{\rho\in\textsf{paths}^{k}(j)} f_\rho(\textsf{BI}) \sqsupseteq \bigsqcap_{\rho\in\textsf{paths}^{k-1}(j)} f_\rho(\textsf{BI}) \tag{3.24}$$

Combining (3.23) and (3.24), we have:

$$\bigsqcap_{\rho\in\textsf{paths}^{k}(j)} f_\rho(\textsf{BI}) = \bigsqcap_{\rho\in\textsf{paths}^{k-1}(j)} f_\rho(\textsf{BI})$$

Therefore the data flow values at the end of iterations $k-1$ and $k$ coincide at every program point and the algorithm terminates.

*Only if part:* Suppose the algorithm halts after $m$ iterations, where $m \leq k$. From Lemma 3.11, the data flow information at any node $j$ after $m$ iterations is $\bigsqcap_{\rho\in\textsf{paths}^{m}(j)} f_\rho(\textsf{BI})$. Further, since the data flow framework is assumed to be distributive, the algorithm computes the $\textsf{MOP}$ solution. Thus

$$\textsf{In}_j = \bigsqcap_{\rho\in\textsf{paths}(j)} f_\rho(\textsf{BI}) = \bigsqcap_{\rho\in\textsf{paths}^{m}(j)} f_\rho(\textsf{BI})$$

Therefore, for an arbitrary path $\rho \in \textsf{paths}(j)$,

$$f_\rho(\textsf{BI}) \sqsupseteq \bigsqcap_{\rho\in\textsf{paths}^{m}(j)} f_\rho(\textsf{BI})$$

We now show that there is a finite set $S$ of paths in $\textsf{paths}^{m}(j)$ such that $\bigsqcap_{\rho\in\textsf{paths}^{m}(j)} f_\rho(\textsf{BI}) = \bigsqcap_{\rho\in S} f_\rho(\textsf{BI})$. Enumerate the paths in $\textsf{paths}^{m}(j)$ as $\rho_1,\rho_2\ldots$, and let $x_i = \bigsqcap_{1\leq n\leq i} f_{\rho_n}(\textsf{BI})$. It is clear that the $x_i$'s form a chain. Therefore, from the descending chain condition, there is a number $i'$ such that for all $i'' > i'$, $x_{i'} = x_{i''}$. Let $S$ be $\{\rho_1,\rho_2,\ldots\rho_{i'}\}$. We then have:

$$f_\rho(\textsf{BI}) \sqsupseteq \bigsqcap_{\rho\in\textsf{paths}^{m}(j)} f_\rho(\textsf{BI}) = \bigsqcap_{\rho\in S} f_\rho(\textsf{BI})$$

∎

Now we prove the main complexity result captured by Theorem 3.2. Note that the theorem asserts a property of data flow frameworks and not of particular instances of

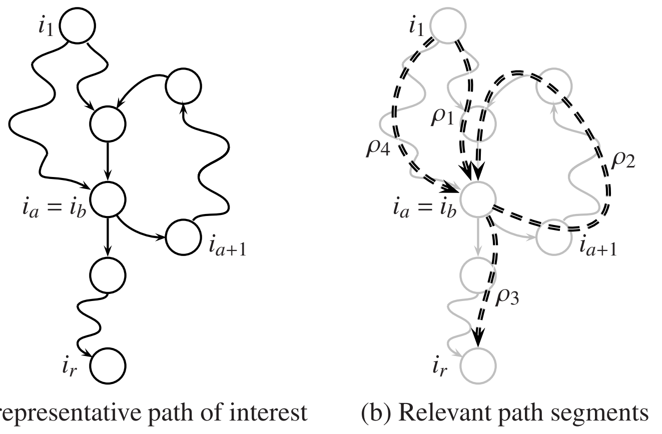


**FIGURE 3.16**
Paths of interest in the CFG.

the framework. It says that if the framework satisfies the rapid condition, the algorithm will terminate for every instance of the framework within an instance-related bound. Conversely, if the algorithm terminates for every instance of the framework within the specified bound, the framework is rapid. The theorem, however, does not say anything about the precision of the specified bound.

**THEOREM 3.2**
*Let $(\mathbb{G}, M_{\mathbb{G}})$ be an arbitrary instance of a distributive data flow framework $(L, \sqcap, F)$. Assume that the traversal of $\mathbb{G}$ is based on the DFST $T$. Then the rapid condition is both necessary and sufficient for the algorithm in Figure 3.15 to terminate within $d(\mathbb{G}, T) + 3$ iterations.*

**PROOF** *If part:* Following Lemma 3.12, it is enough to show that for an arbitrary program point $j$ and a path $\rho \in \textsf{paths}(j)$, there exists a set of paths $S = \{\rho_1, \rho_2, \ldots, \rho_m\} \subseteq \textsf{paths}^{d(\mathbb{G},T)+2}(j)$ and

$$f_\rho(\textsf{BI}) \sqsupseteq \bigsqcap_{\rho' \in S} f_{\rho'}(\textsf{BI})$$

We shall prove the above by induction on the number of back edges $l$ in the path $\rho$.

*Basis:* $l \le d(\mathbb{G}, T) + 1$. In this case $\rho$ itself is in $\textsf{paths}^{d(\mathbb{G},T)+2}(j)$ and $S = \{\rho\}$.

*Inductive Step:* $l > d(\mathbb{G}, T) + 1$. Since the number of back edges in $\rho$ exceeds the depth $d(\mathbb{G}, T)$, $\rho$ has a cycle. Let us enumerate the program points that constitute $\rho$ as $(i_0, i_1, \ldots, i_a, \ldots, i_b, \ldots, i_r)$, where $i_a$ is the last point in the path that is the same as a later point $i_b$ in the path. This situation is illustrated in Figure 3.16. We now identify the following subpaths of $\rho$ in the graph:

- The path $\rho_1 = (i_1,\dots i_a)$ contains at least one back edge. This is because the path $(i_{a+1},\dots,i_r)$ is cycle free and contains at most $d(\mathbb{G},T)$ edges, and even assuming the edge $i_a \rightarrow i_{a+1}$ to be a back edge, the number of back edges in $(i_a,\dots,i_r)$ is at most $d(\mathbb{G},T)+1$.
- The path $\rho_2 = (i_a,\dots i_b)$ constitutes a cycle and therefore must contain at least one back edge.
- The path $\rho_3 = (i_b,\dots i_r)$ is an acyclic path and therefore contains at most $d(\mathbb{G},T)$ back edges.
- Let $\rho_4$ be a back edge free path from $i_0$ to $i_a$. Such a path can always be found by following tree edges from $i_0$ to $i_a$.

Using the rapid condition, $f_\rho(\mathsf{BI})$ can be rewritten as:

$$\begin{aligned} f_\rho(\mathsf{BI}) &= f_{\rho_3}(f_{\rho_2}(f_{\rho_1}(\mathsf{BI}))) \\ &\sqsupseteq f_{\rho_3}\left(f_{\rho_1}(\mathsf{BI}) \sqcap f_{\rho_2}(x) \sqcap x\right) \end{aligned}$$

for any $x$. Instantiating $x$ to $f_{\rho_4}(\mathsf{BI})$, we have

$$f_\rho(\mathsf{BI}) \sqsupseteq f_{\rho_3}\left(f_{\rho_1}(\mathsf{BI}) \sqcap f_{\rho_2}(f_{\rho_4}(\mathsf{BI})) \sqcap f_{\rho_4}(\mathsf{BI})\right)$$

which, because of distributivity, gives

$$f_\rho(\mathsf{BI}) \sqsupseteq f_{\rho_3}(f_{\rho_1}(\mathsf{BI})) \sqcap f_{\rho_3}(f_{\rho_2}(f_{\rho_4}(\mathsf{BI}))) \sqcap f_{\rho_3}(f_{\rho_4}(\mathsf{BI}))$$

Recall that the original path $\rho_1\rho_2\rho_3$ had $l$ back edges. We observe that:

- The path $\rho_1\rho_3$ has at most $l-1$ back edges since the path $\rho_2 = (i_a,\dots i_b)$ has at least one back edge.
- The path $\rho_4\rho_2\rho_3$ has at most $l-1$ back edges since $\rho_1$, which had at least one back edge, has been replaced by $\rho_4$ which has none.
- The path $\rho_4\rho_3$ has less than $l-1$ back edges.

Thus the induction hypothesis applies and there exists sets $S_1$, $S_2$ and $S_3$, all of them subsets of $\mathsf{paths}^{d(\mathbb{G},T)+2}(j)$, such that

$$f_\rho(\mathsf{BI}) \sqsupseteq \bigsqcap_{\sigma\in S_1} f_\sigma(\mathsf{BI}) \sqcap \bigsqcap_{\sigma\in S_2} f_\sigma(\mathsf{BI}) \sqcap \bigsqcap_{\sigma\in S_3} f_\sigma(\mathsf{BI})$$

Thus the required set $S$ is $S_1 \cup S_2 \cup S_3$.

*Only if part:* Assume that condition (3.18) is violated for a data flow framework, i.e.,

$$(\exists f,g \in F)(\exists x,\ \mathsf{BI} \in L):\ f(g(\mathsf{BI})) \not\sqsupseteq g(\mathsf{BI}) \sqcap f(x) \sqcap (x)$$

Using the above $f$, $g$, $x$ and $\mathsf{BI}$, we have to create an instance of the framework for which the algorithm takes more than $d(\mathbb{G},T)+1$ iterations to terminate.

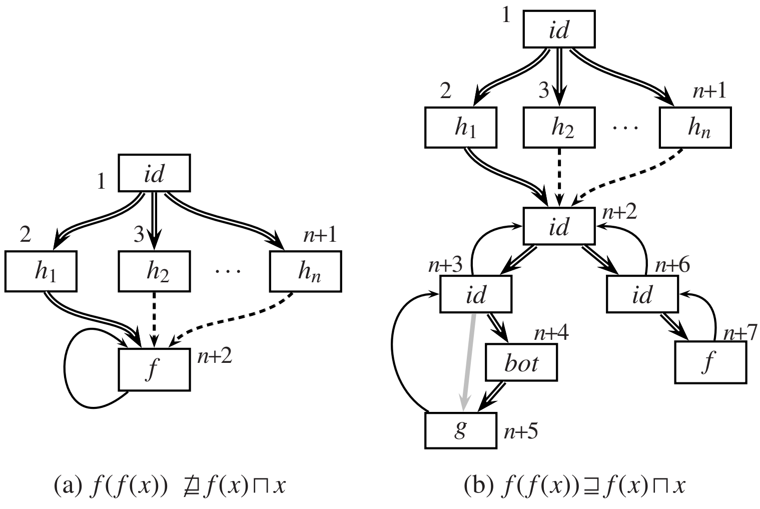


**FIGURE 3.17**
Instances that require more than $d(\mathbb{G},T)$ iterations to converge.

Because of the conditions on the values in the data flow lattice and the admissible flow functions (Section 3.2.2), we can assume that $x = \prod_{1 \leq i \leq n} h_i(\mathit{BI})$. There are two cases to consider. If $f(f(x)) \not\sqsupseteq f(x) \sqcap x$ then consider the CFG of part (a) of Figure 3.17. The tree edges are drawn with double lines, back edges are single lines, cross edges are dashed lines, and forward edges are gray lines. The depth of the graph for the indicated DFST is 0.[‡] The data flow values $\mathit{In}_{n+2}$ in the first three iterations are: $x$, $x \sqcap f(x)$, and $x \sqcap f(x) \sqcap f(f(x))$ respectively. Clearly, the algorithm will not terminate within 3 iterations.

If $f(f(x)) \sqsupseteq f(x) \sqcap x$, then we consider the instance in part (b) of Figure 3.17. The function *bot* in node $n$+4 is the constant function $\forall x \in L : \mathit{bot}(x) = \bot$. The depth of the graph in this case is 2. Figure 3.18 shows the data flow values at program points of interest in the first four iterations. In the fifth iteration, the data flow value at $n$+3 is $x \sqcap f(x) \sqcap g(\bot) \sqcap f(g(\bot))$. Under the assumed condition, this value is different from the value at $n+3$ in the fourth iteration. Therefore the algorithm takes at least six iterations to terminate. ▮

***Example 3.11***
Figure 3.19 provides an instance of a framework that requires $d(G,T)+3$ iterations. We leave it for the reader to verify that the framework is distributive and rapid. As usual, tree edges have been shown in double lines and back

[‡]Note that this is because we are not distinguishing between *In* and *Out* properties.

| Node $i$ | $In_i$ in various iterations | | | |
|---|---|---|---|---|
| | #1 | #2 | #3 | #4 |
| $n$+2 | $x$ | $x$ | $x \sqcap f(x) \sqcap g(\bot)$ | $x \sqcap f(x) \sqcap g(\bot)$ |
| $n$+3 | $x$ | $x \sqcap g(\bot)$ | $x \sqcap f(x) \sqcap g(\bot)$ | $x \sqcap f(x) \sqcap g(\bot)$ |
| $n$+4 | $x$ | $x \sqcap g(\bot)$ | $x \sqcap f(x) \sqcap g(\bot)$ | $x \sqcap f(x) \sqcap g(\bot)$ |
| $n$+5 | $\bot$ | $\bot$ | $\bot$ | $\bot$ |
| $n$+6 | $x$ | $x \sqcap f(x)$ | $x \sqcap f(x) \sqcap f(f(x)) \sqcap g(\bot)$ $= x \sqcap f(x) \sqcap g(\bot)$ | $x \sqcap f(x) \sqcap g(\bot) \sqcap f(g(\bot))$ |
| $n$+7 | $x$ | $x \sqcap f(x)$ | $x \sqcap f(x) \sqcap g(\bot)$ | $x \sqcap f(x) \sqcap g(\bot) \sqcap f(g(\bot))$ |

**FIGURE 3.18**
First four iterations for the instance in Figure 3.17 on the facing page.

edges are shown in single lines. The value of $d(G,T)$ is 1. The lattice does not have a $\top$ element but the graph is reducible.

As shown in the following table, the data flow values converge in the third iteration—one more iteration is required to detect convergence. This is independent of the *BI* value because of the presence of function $h_1$. We leave it for the reader to verify that if $L$ contains $\top$, three iterations are sufficient.

| Variables | Values in each iteration | | | |
|---|---|---|---|---|
| | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 |
| $In_1$ | $v_1$ | $v_1$ | $v_1$ | $v_1$ |
| $In_2$ | $v_1$ | $v_1$ | $\bot$ | $\bot$ |
| $In_3$ | $v_0$ | $\bot$ | $\bot$ | $\bot$ |
| $In_4$ | $v_0$ | $\bot$ | $\bot$ | $\bot$ |

$\square$

An non-rapid fast framework has been illustrated in Figure 5.9 on page 178.

### 3.5.3 Complexity in Special Cases

First we consider the situation when the CFG is reducible. The modified statement of Theorem (3.2) for reducible CFGs is as follows.

***THEOREM 3.3***
*Let $(\mathbb{G}, M_{\mathbb{G}})$ be an arbitrary instance of a distributive data flow framework $(L, \sqcap, F)$ such that $\mathbb{G}$ is reducible. Assume that the traversal of $\mathbb{G}$ is based on the DFST $T$. Then the rapid condition is both necessary and sufficient for the algorithm in Figure 3.15 to terminate within $d(\mathbb{G},T)+2$ iterations.*

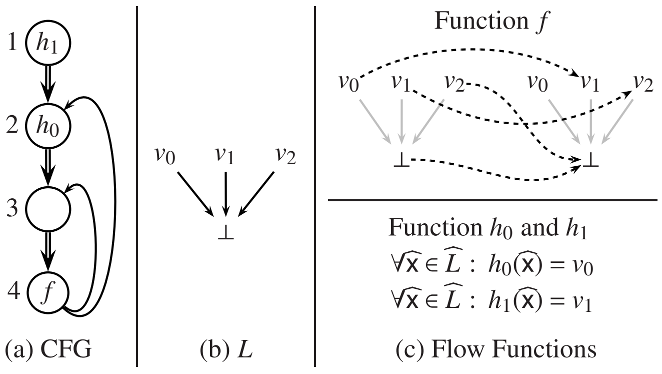

1 $h_1$
2 $h_0$
3
4 $f$
(a) CFG
$v_0$ $v_1$ $v_2$
$\perp$
(b) $L$
Function $f$
$v_0$ $v_1$ $v_2$ $v_0$ $v_1$ $v_2$
$\perp$ $\perp$
Function $h_0$ and $h_1$
$\forall \widehat{x} \in \widehat{L}: h_0(\widehat{x}) = v_0$
$\forall \widehat{x} \in \widehat{L}: h_1(\widehat{x}) = v_1$
(c) Flow Functions

**FIGURE 3.19**
A instance of a distributive rapid framework that requires $d(G,T)+3$ iterations.

**PROOF** We replay the earlier proof with the expression $d(\mathbb{G},T)+x$ uniformly replaced by $d(\mathbb{G},T)+(x-1)$. The only change is in the portion of the proof that asserts the sufficiency of the rapid condition (the *if* part).

The basis of the *if* part remains identical. We consider the changes required to prove the inductive case. Reducibility imposes some restrictions on the structure of the path $\rho$. We consider the following two cases:

- The edge $i_a \rightarrow i_{a+1}$ is not a back edge. Since the path from $i_{a+1}$ to $i_r$ is acyclic, it can have at most $d(\mathbb{G},T)$ back edges. Thus the only way in which the path $\rho_1\rho_2\rho_3$ can have $d(\mathbb{G},T)+1$ back edges is to have at least one back edge in $\rho_1$. As in the earlier proof, this path can be replaced by a back edge free path and the induction hypothesis applied.

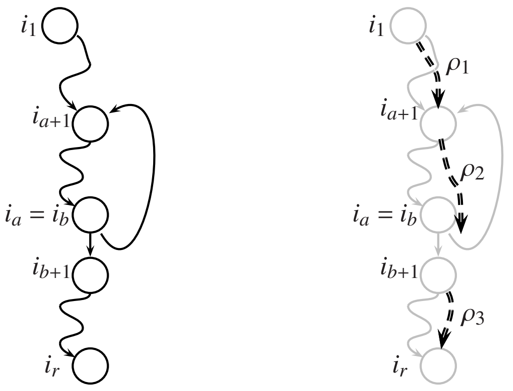
- The edge $i_a \rightarrow i_{a+1}$ is a back edge. Due to reducibility, $i_{a+1}$ must dominate $i_a$ and the path segment $\rho_1$ must also pass through $i_{a+1}$. We can then divide the path $\rho$ into path segments as illustrated in Figure 3.20 on the facing page. Due to the back edge $i_a \rightarrow i_{a+1}$, the path $\rho$ is $\rho_1\rho_2\rho_2\rho_3$. Since the path from $i_{a+1}$ to $i_r$ is acyclic, the path $\rho_2\rho_3$ can have at most $d$ back edges. The data flow value along path $\rho$ is:

$$\begin{aligned} f_\rho(BI) &= f_{\rho_3}\left(f_{\rho_2}\left(f_{\rho_2}\left(f_{\rho_1}(BI)\right)\right)\right) \\ &\sqsupseteq f_{\rho_3}\left(f_{\rho_2}\left(f_{\rho_1}(BI)\right)\right) \end{aligned}$$

since $f^2(x) \sqsupseteq f(x)$ from Lemma 3.10.

Observe that $\rho$ has been replaced by the path $\rho_1\rho_2\rho_3$ which excludes the back edge $i_a \rightarrow i_{a+1}$ and thus contains one back edge less. Hence, the induction hypothesis applies to the path $\rho_1\rho_2\rho_3$ and the result follows.

▌



(a) $i_a \rightarrow i_{a+1}$ is a back edge and graph is reducible.

(b) Relevant path segments

**FIGURE 3.20**
Paths of interest in the CFG.

Now we consider the special case when a $\top$ element exists in the meet semilattice. The bit vector data flow problems fall in this category. In this case, the algorithm in Figure 3.15 can be made more efficient by initializing *In* for each node except *Start* to the value $\top$. This is identical to the initialization on line 2 of Figure 3.9. Thus the two algorithms become similar except for the order of traversal. Only the iterations of the **while** loop are counted—the work done during the initialization step is ignored. This is reasonable because, unlike the algorithm in Figure 3.15, no attempt is being made to propagate the initial values in this step. We merely state the theorem and point to the source of the proof in bibliographic notes.

***THEOREM 3.4***
*Consider an instance $(\mathbb{G}, M_{\mathbb{G}})$ of a distributive data flow framework $(L, \sqcap, F)$, where $L$ has a $\top$ element. The rapid condition is both necessary and sufficient for the algorithm in Figure 3.15 with the modification mentioned above to terminate within $d(\mathbb{G},T)+2$ iterations. $T$ is the DFST used for deciding the order of traversal of $\mathbb{G}$.*

## 3.6 Summary and Concluding Remarks

In this chapter, we have presented generalizations of data flow frameworks based on mathematical abstractions and have presented lattice theoretic modelling of data flow frameworks. The generalizations include data flow values, operations to manipulate them, algorithms to compute the data flow information and the characteristics of the computed data flow information. All these generalizations are uniformly applicable to all unidirectional monotone data flow frameworks but are not directly applicable to bidirectional frameworks. Even among unidirectional frameworks, the generalizations related to complexity of data flow analysis are applicable to a limited class of frameworks, leaving out some important frameworks that have arisen in practical situations. In the subsequent chapters, we consider some of these data flow frameworks and then present a different view of data flow analysis to uniformly characterize the complexity of a larger class of data flow frameworks including bidirectional frameworks.

## 3.7 Bibliographic Notes

Of the graph theoretic concepts introduced early in the chapter, discussion on DFST can be found in the texts by Aho, Hopcroft and Ullman [2] and Cormen, Rivest, Leiserson and Stein [27]. Reducibility was introduced by Allen [4] and is further discussed by Hecht and Ullman in [45, 46]. Dominance was introduced by Lowry and Medlock [70]. Lengauer and Tarjan [68] present an algorithm that can be used to compute dominators efficiently. The text by Davey and Priestley [29] is a good introduction to lattice theory. The presentation of Tarski's fixed point theorem is from Tarski's original paper [99].

The initial attempt to model data flow values in terms of meet semilattices was by Kildall [63]. Kam and Ullman [49] introduced reverse postorder for visiting nodes in the CFG and also introduces the rapidity condition which guarantees convergence within $d(\mathbb{G}, T) + 3$ iterations. This work has given rise to the folklore that iterative data flow analysis is fast for many data flow frameworks. Much of Section 3.5.2 is based on this paper.

The papers so far dealt with distributive frameworks. Kam and Ullman [50] showed that for montonic frameworks, which are less restrictive then distributive frameworks, a round-robin iterative algorithm computes the MFP solution. However, for a monotonic framework that is not distributive, the MFP solution may be different from the MOP solution. They also showed the undecidability of the problem of finding MOP solution of an arbitrary monotonic data flow problem. Monotonicity of flow functions was also discussed by Graham and Wegman [37] where they introduced the concept of fast frameworks. A detailed treatment of these concepts can also be found in the book by Hecht [44]. Marlowe and Ryder [71] review properties of different data flow frameworks in lattice theoretic settings.