4

General Data Flow Frameworks

In bit vector frameworks the data flow information of different entities is independent of each other. However, there are many situations in which the data flow information of an entity could depend on the data flow information of some other entity. For example, the concept of transfer of liveness was described in Section 1.1.2 as follows:

> If access path $x{-}\!\triangleright\sigma$ is live after an assignment `x = y`, then σ is transferred to x and the access path $y{-}\!\triangleright\sigma$ becomes live before the assignment.

Here, the liveness of access path $x{-}\!\triangleright\sigma$ depends on the liveness of access path $y{-}\!\triangleright\sigma$. Capturing such interdependences requires a more general kind of flow functions and the frameworks involving such flow functions are called non-separable.

4.1 Non-Separable Flow Functions

This section defines the non-separability of flow functions, shows how it can be modeled in terms of *Gen* and *Kill* effects, and describes the limitations it imposes on the nature of basic blocks that can be constructed for performing data flow analysis.

Recall that a data flow framework $(L_{\mathcal{G}}, \sqcap_{\mathcal{G}}, F_{\mathcal{G}})$ is defined in terms of an unspecified CFG $\mathcal{G}$. For convenience, we drop the subscript $\mathcal{G}$ where not required. We assume that the entities occurring in $\mathcal{G}$ that are of interest to us for a given analysis are contained in a set $\Sigma = \{\alpha, \beta, \gamma, \ldots, \omega\}$. A given analysis discovers some properties of interest for a specific kind of entities e.g., expressions, variables, definitions, etc. Thus for any given analysis, all entities are of the same type. The lattice L is a product $\widehat{L}_\alpha \times \widehat{L}_\beta \times \cdots \times \widehat{L}_\omega$ where $\widehat{L}_\alpha$ is the component lattice containing the data flow values of entity α. In general, all $\widehat{L}$s are same. Data flow value $\mathsf{x} \in L$ is a tuple $\langle \widehat{\mathsf{x}}^\alpha, \widehat{\mathsf{x}}^\beta, \ldots, \widehat{\mathsf{x}}^\omega \rangle$.

The motivation behind modeling non-separability explicitly arises from the observation that an element in L is not atomic—it consists of a tuple of separate data flow values for each entity. Thus it is natural to ask if instead of viewing flow functions as atomic, they can also be modeled in terms of functions that compute data flow values of smaller granularities. This view allows us to explicate the dependence of the data flow value of an entity on the data flow values of the other entities. This leads to rich insights that are useful in defining tight complexity bounds as well as the feasibility conditions for systematic reduction of flow function compositions.

DEFINITION 4.1 *A flow function $f : L \mapsto L$ is separable iff it is a tuple $\langle \widehat{f}^{\,\alpha}, \widehat{f}^{\,\beta}, \cdots, \widehat{f}^{\,\omega} \rangle$ of component functions $\widehat{f} : \widehat{L} \mapsto \widehat{L}$. If $\widehat{f}$ is of the form $L \mapsto \widehat{L}$, then f is non-separable.*

A component function $\widehat{f}^{\,\alpha}$ computes the data flow value of entity α. Similar to the flow function, we use basic block as a subscript of the component function when required.

As the name suggests, separability is based on independence of data flow properties of entities for which data flow analysis is being performed. In order to model non-separable flow functions in terms of *Gen* and *Kill* components, instead of defining constant Gen_n and Kill_n, we define them as $\mathit{Gen}_n : L \mapsto L$ and $\mathit{Kill}_n : L \mapsto L$ by allowing dependent parts also:

$$\mathit{Gen}_n(\mathsf{x}) = \mathit{ConstGen}_n \cup \mathit{DepGen}_n(\mathsf{x}) \tag{4.1}$$

$$\mathit{Kill}_n(\mathsf{x}) = \mathit{ConstKill}_n \cup \mathit{DepKill}_n(\mathsf{x}) \tag{4.2}$$

The flow function f_n is defined as:

$$f_n(\mathsf{x}) = (\mathsf{x} - \mathit{Kill}_n(\mathsf{x})) \cup \mathit{Gen}_n(\mathsf{x}) \tag{4.3}$$

In bit vector frameworks, the dependent parts are absent resulting in constant *Gen* and *Kill* components. Rapid and fast frameworks require that the flow functions are separable, so that the rapidity condition (3.18) and fastness condition (3.17) are satisfied. In these and other separable frameworks, dependent parts may exist due to a possibility of dependence among data flow values of the same entity at different program points. In non-separable frameworks, the dependence can be of two types: The data flow value of a given entity may depend on the data flow value of the same entity or on data flow value of some other entity. Dependence captured by *DepGen* on the data flow value of the same entity must necessarily be a non-identity dependence because identity dependence is implicitly defined by ensuring that both *Gen* and *Kill* have no effect on the entity. The dependence on other entities may be identity or non-identity dependence. Unlike identity dependence on the same entity, identity dependence on other entities must be explicitly defined. We model these dependences in Section 4.5.

The presence of dependent parts in *Gen* and *Kill* makes it difficult to summarize the effect of multiple statements in a flow function. Hence, basic blocks for non-separable analyses consist of single statements. However, multiple consecutive statements which do not have any data dependence between them can still be combined into a basic block subject to the usual control flow restriction. If two consecutive statements can be executed in any order without affecting program semantics, then they can be grouped into the same basic block for data flow analysis of non-separable flows. Further, a conditional or unconditional jump need not always be a separate block. If it is included in a block, it must be the last statement of the block.

The statements relevant to data flow analysis are divided in the following categories: (a) assignment statements $x = e$ where $x \in \mathbb{V}\mathsf{ar}$, $e \in \mathbb{E}\mathsf{xpr}$, (b) input statements *read*(x) which assign a new value to x, (c) use statements *use*(x) which model uses of x for condition checking, printing and parameter passing etc., and (d) other statements. Since we restrict ourselves to intraprocedural analysis in this part, we assume that there are no function calls. Effectively, $\mathbb{V}\mathsf{ar}$ contains local variables only. Print statements and evaluation of branching condition etc. are modeled in terms of use statements.

4.2 Discovering Properties of Variables

In this section we present analyses to discover whether a given scalar variable is dead, or possibly undefined, or has a constant value.

4.2.1 Faint Variables Analysis

As discussed towards the end of Section 2.3.1, liveness analysis does not take into account interdependence of variables. This section describes a data flow analysis which takes into account such interdependence and discovers the transitive closure of deadness of a variable which is the complement of liveness.

DEFINITION 4.2 *A variable $x \in \mathbb{V}\mathsf{ar}$ is faint at a program point u if along every path from u to End, it is either not used before being defined or is used to define a faint variable.*

Clearly, this is a backward data flow problem. However, unlike liveness analysis this is an all-paths analysis. Hence the confluence operation is $\cap$. The lattice is $(2^{\mathbb{V}\mathsf{ar}}, \subseteq)$ and $\top$ is $\mathbb{V}\mathsf{ar}$. The initial value of In_n and Out_n for all n is $\mathbb{V}\mathsf{ar}$.

$$\mathit{In}_n = f_n(\mathit{Out}_n) \tag{4.4}$$

$$\mathit{Out}_n = \begin{cases} \mathit{BI} & n \text{ is } \mathit{End} \\ \bigcap_{s \in \mathit{succ}(n)} \mathit{In}_s & \text{otherwise} \end{cases} \tag{4.5}$$

All local variables are dead at the end of a procedure and $\mathit{BI} = \mathbb{V}\mathsf{ar}$.

The constant and dependent parts of $\mathit{Gen}_n(\mathsf{x})$ component are defined as follows. A variable x becomes faint before every assignment to it. There is no other way in which a variable that is live after a statement, could become faint before the statement.

$$\mathit{ConstGen}_n = \begin{cases} \{x\} & n \text{ is assignment } x = e,\ x \notin \mathit{Opd}(e) \\ \{x\} & n \text{ is } \mathit{read}(x) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{DepGen}_n(\mathsf{x}) = \emptyset$$

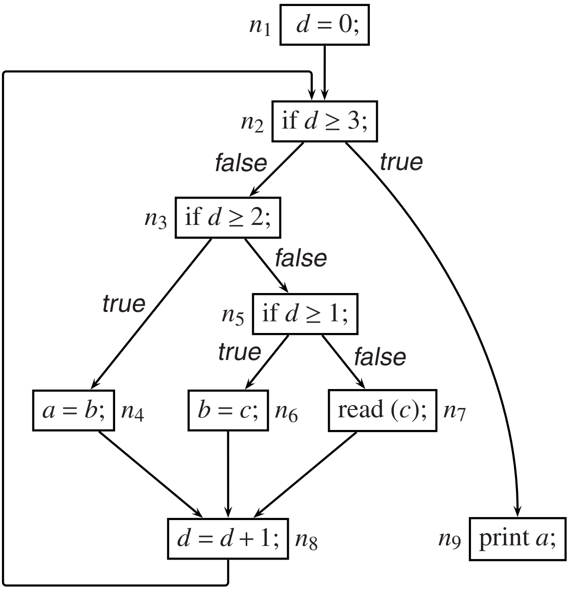


FIGURE 4.1
Program for illustrating faint variables analysis and possibly uninitialized variables analysis.

A variable x could cease to become faint before an assignment statement if it appears on the right hand side and the left hand side variable is faint. Alternatively, it could cease to become faint because of a use statement. The former represents the transitive effect of left hand side variable not being faint and is captured by the dependent part $\mathit{DepKill}_n(\mathsf{x})$ as follows:

$$\mathit{ConstKill}_n = \begin{cases} \{x\} & n \text{ is } use(x) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{DepKill}_n(\mathsf{x}) = \begin{cases} \mathit{Opd}(e) \cap \mathbb{V}\mathsf{ar} & n \text{ is assignment } x = e,\ x \notin \mathsf{x} \\ \emptyset & \text{otherwise} \end{cases}$$

where Opd(e) denotes the operands of expression e.

Example 4.1

The result of performing faint variables analysis for the program in Figure 4.1 has been shown in Figure 4.2. Since a is used in block n_9, it is not faint. As a consequence, variables b and c cease to be faint. Discovering these facts requires two additional iterations and propagating it against the back edge requires the fourth iteration.

If n_9 did not contain a use of a, the variables a, b, and c would have been

Node	Iteration #1		Changes in Iteration #2		Changes in Iteration #3		Changes in Iteration #4	
	Out_n	In_n	Out_n	In_n	Out_n	In_n	Out_n	In_n
n_9	$\{a,b,c,d\}$	$\{b,c,d\}$						
n_8	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{b,c\}$	$\{b,c\}$	$\{c\}$	$\{c\}$	$\emptyset$	$\emptyset$
n_7	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{b,c\}$	$\{b,c\}$	$\{c\}$	$\{c\}$	$\emptyset$	
n_6	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{b,c\}$	$\{b,c\}$	$\{c\}$	$\emptyset$	$\emptyset$	
n_5	$\{a,b,c,d\}$	$\{a,b,c\}$	$\{b,c\}$	$\{b,c\}$	$\emptyset$	$\emptyset$		
n_4	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{b,c\}$	$\{c\}$	$\{c\}$		$\emptyset$	$\emptyset$
n_3	$\{a,b,c\}$	$\{a,b,c\}$	$\{c\}$	$\{c\}$	$\emptyset$	$\emptyset$		
n_2	$\{b,c\}$	$\{b,c\}$	$\{c\}$	$\{c\}$	$\emptyset$	$\emptyset$	$\emptyset$	
n_1	$\{b,c\}$	$\{b,c,d\}$	$\{c\}$	$\{c,d\}$	$\emptyset$	$\{d\}$		

FIGURE 4.2
Performing faint variables analysis of program in Figure 4.1.

discovered to be faint. Liveness analysis would conclude that b and c are live regardless of the use of a in block n_9. □

It is interesting to explore the distributivity, rapidity, and fastness properties of faint variables analysis. Since $DepGen_n(\mathsf{x})$ is $\emptyset$, f_n can be rewritten as:

$$\begin{aligned} f_n(\mathsf{x}) &= (\mathsf{x} - Kill_n(\mathsf{x})) \cup Gen_n(\mathsf{x}) \\ &= (\mathsf{x} - (ConstKill_n \cup DepKill_n(\mathsf{x}))) \;\cup\; (ConstGen_n \cup DepGen_n(\mathsf{x})) \\ &= ((\mathsf{x} - ConstKill_n) \cup ConstGen_n) \;\cup\; (\mathsf{x} - DepKill_n(\mathsf{x})) \end{aligned}$$

Clearly the constant part of f_n is similar to flow functions in bit vector frameworks and hence is distributive, rapid and fast. Thus, in order to investigate whether these properties hold for f_n, it is sufficient to explore them for $(\mathsf{x} - DepKill_n(\mathsf{x}))$.

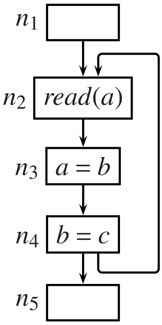
LEMMA 4.1
Faint variables analysis is distributive.

PROOF It is sufficient to prove that $\forall \mathsf{x}_1, \mathsf{x}_2 \in L, \forall f_n \in F$:

$$(\mathsf{x}_1 \cap \mathsf{x}_2) - DepKill_n(\mathsf{x}_1 \cap \mathsf{x}_2) = (\mathsf{x}_1 - DepKill_n(\mathsf{x}_1)) \;\cap\; (\mathsf{x}_2 - DepKill_n(\mathsf{x}_2))$$

From the definition of $DepKill_n(\mathsf{x})$, there are two cases to consider. First we consider the case when n is an assignment statement $x = e$ and $x \notin \mathsf{x}_1 \cap \mathsf{x}_2$. Assume that x is neither in x_1 nor in x_2.

$$\begin{aligned} (\mathsf{x}_1 \cap \mathsf{x}_2) - DepKill_n(\mathsf{x}_1 \cap \mathsf{x}_2) &= (\mathsf{x}_1 \cap \mathsf{x}_2) - (Opd(e) \cap \mathbb{V}\mathsf{ar}) \\ &= (\mathsf{x}_1 - (Opd(e) \cap \mathbb{V}\mathsf{ar})) \;\cap\; (\mathsf{x}_2 - (Opd(e) \cap \mathbb{V}\mathsf{ar})) \\ &= (\mathsf{x}_1 - DepKill_n(\mathsf{x}_1)) \;\cap\; (\mathsf{x}_2 - DepKill_n(\mathsf{x}_2)) \end{aligned}$$



Let $f = f_{n_2} \circ f_{n_3} \circ f_{n_4}$ and let x be $\mathbb{V}\mathsf{ar}$. $f^i(\mathsf{x})$ represents the set of faint variables at the entry of n_2 in iteration number i in postorder traversal over the graph.

$$\begin{aligned} \mathsf{x} &= \mathbb{V}\mathsf{ar} \\ f(\mathsf{x}) &= \mathbb{V}\mathsf{ar} - \{a\} \\ f^2(\mathsf{x}) &= \mathbb{V}\mathsf{ar} - \{a,b\} \\ \forall i \geq 3 : \ f^i(\mathsf{x}) &= \mathbb{V}\mathsf{ar} - \{a,b,c\} \\ \mathsf{x} \cap f(\mathsf{x}) \cap f^2(\mathsf{x}) \cap \ldots &\neq \mathsf{x} \cap f(\mathsf{x}) \end{aligned}$$

FIGURE 4.3
Faint variables analysis is not fast.

If $x \notin \mathsf{x}_1$ but $x \in \mathsf{x}_2$, $\mathit{DepKill}_n(\mathsf{x}_2)$ is $\emptyset$ and the proof obligation follows due to $\cap$ even if $\mathit{Opd} \cap \mathbb{V}\mathsf{ar}$ is not removed from x_2.

In other situations, $\mathit{DepKill}_n(\mathsf{x})$ is $\emptyset$ and the lemma trivially follows. ∎

Figure 4.3 contains an instance of faint variables analysis to show that it is neither rapid nor fast. It is easy to generalize the example to show that faint variables analysis is not k-bounded. It is bounded by height of the lattice which turns out to be $|\mathbb{V}\mathsf{ar}|$ and depends on the particular instance.

4.2.2 Possibly Uninitialized Variables Analysis

Section 2.3.3 described reaching definitions analysis which is primarily motivated by construction of def-use chains. If we use *BI* to include definitions $x = \mathit{undef}$ for all $x \in var$, reaching definitions analysis also discovers the program points where these definitions reach suggesting the possibility of a use before a variable is initialized. However, the transitive effect of such definitions is not handled by reaching definitions analysis. We present an analysis which handles these effects. Further, unlike reaching definitions analysis, this analysis is aimed at discovering only whether a given variable is possibly uninitialized—It does not collect the definitions of the variable. This simplifies the analysis and makes it very efficient.

DEFINITION 4.3 *A variable $x \in \mathbb{V}\mathsf{ar}$ is possibly uninitialized at a program point u if there exists a path from Start to u along which either no definition of the variable has been encountered or the definition uses a possibly uninitialized variable on the right hand side of the assignment.*

Clearly this is a forward data flow problem and uses $\cup$ as the confluence operation.

The lattice is $(2^{\mathbb{V}\mathsf{ar}}, \supseteq)$ and $\top$ is $\emptyset$. The initial value at each node is $\emptyset$.

$$In_n = \begin{cases} BI & n \text{ is } Start \\ \displaystyle\bigcup_{p \in pred(n)} Out_p & \text{otherwise} \end{cases} \tag{4.6}$$

$$Out_n = f_n(In_n) \tag{4.7}$$

Since every local variable is uninitialized at $Entry(Start)$, $BI = \mathbb{V}\mathsf{ar}$.

An interesting aspect of this analysis is that the possibility of a variable being uninitialized is generated only at $Entry(Start)$ and no other program point. Hence $ConstGen_n$ is $\emptyset$. The transitive effect of an uninitialized variable appearing on the right hand side of an assignment is captured by $DepGen_n(\mathsf{x})$.

$$ConstGen_n = \emptyset$$

$$DepGen_n(\mathsf{x}) = \begin{cases} \{x\} & n \text{ is assignment } x = e,\ Opd(e) \cap \mathsf{x} \neq \emptyset\} \\ \emptyset & \text{otherwise} \end{cases}$$

A variable ceases to be uninitialized if its value is read from input or a constant value is assigned to it. The transitive effect of such initializations is captured by $DepKill_n(\mathsf{x})$.

$$ConstKill_n = \begin{cases} \{x\} & n \text{ is assignment } x = e,\ Opd(e) \subseteq \mathbb{C}\mathsf{onst} \\ \{x\} & n \text{ is } read(x) \\ \emptyset & \text{otherwise} \end{cases}$$

$$DepKill_n(\mathsf{x}) = \begin{cases} \{x\} & n \text{ is assignment } x = e,\ Opd(e) \cap \mathsf{x} = \emptyset\} \\ \emptyset & \text{otherwise} \end{cases}$$

Example 4.2

For the program in Figure 4.1, the result of possibly uninitialized analysis is: $In_{n_1} = \{a,b,c,d\}$, $Out_{n_4} = \{b,c\}$, $Out_{n_6} = \{a,c\}$, $Out_{n_6} = \{a,b\}$. All other In_n and Out_n are $\{a,b,c\}$. □

LEMMA 4.2

Possibly uninitialized analysis is distributive.

PROOF It is sufficient to show that $\forall \mathsf{x}_1, \mathsf{x}_2 \in L, \forall f_n \in F$:

$$((\mathsf{x}_1 \cup \mathsf{x}_2) - DepKill_n(\mathsf{x}_1 \cup \mathsf{x}_2)) \cup Gen_n(\mathsf{x}_1 \cup \mathsf{x}_2) =$$
$$(\mathsf{x}_1 - DepKill_n(\mathsf{x}_1)) \cup (\mathsf{x}_2 - DepKill_n(\mathsf{x}_2)) \cup Gen_n(\mathsf{x}_1) \cup Gen_n(\mathsf{x}_2)$$

Further, it is sufficient to consider only the assignment statement $x = e$.

Consider the following three cases:

- $Opd(e) \cap \mathsf{x}_1 = \emptyset$ and $Opd(e) \cap \mathsf{x}_2 = \emptyset$. Thus $Opd(e) \cap (\mathsf{x}_1 \cup \mathsf{x}_2) = \emptyset$.

In this case.

$$\begin{array}{llll} \mathit{DepKill}_n(\mathsf{x}_1 \cup \mathsf{x}_1) & = \mathit{DepKill}_n(\mathsf{x}_1) & = \mathit{DepKill}_n(\mathsf{x}_2) & = \{x\} \\ \mathit{DepGen}_n(\mathsf{x}_1 \cup \mathsf{x}_1) & = \mathit{DepGen}_n(\mathsf{x}_1) & = \mathit{DepGen}_n(\mathsf{x}_2) & = \emptyset \end{array}$$

Hence the proof obligation is satisfied.

- $\mathit{Opd}(e) \cap \mathsf{x}_1 \neq \emptyset$ and $\mathit{Opd}(e) \cap \mathsf{x}_2 \neq \emptyset$. Thus $\mathit{Opd}(e) \cap (\mathsf{x}_1 \cup \mathsf{x}_2) \neq \emptyset$.

  In this case.

  $$\begin{array}{llll} \mathit{DepKill}_n(\mathsf{x}_1 \cup \mathsf{x}_1) & = \mathit{DepKill}_n(\mathsf{x}_1) & = \mathit{DepKill}_n(\mathsf{x}_2) & = \emptyset \\ \mathit{DepGen}_n(\mathsf{x}_1 \cup \mathsf{x}_1) & = \mathit{DepGen}_n(\mathsf{x}_1) & = \mathit{DepGen}_n(\mathsf{x}_2) & = \{x\} \end{array}$$

  Hence the proof obligation is satisfied.

- $\mathit{Opd}(e) \cap \mathsf{x}_1 \neq \emptyset$ and $\mathit{Opd}(e) \cap \mathsf{x}_2 = \emptyset$. Thus $\mathit{Opd}(e) \cap (\mathsf{x}_1 \cup \mathsf{x}_2) \neq \emptyset$.

  In this case.

  $$\begin{array}{llll} \mathit{DepKill}_n(\mathsf{x}_1 \cup \mathsf{x}_1) & = \mathit{DepKill}_n(\mathsf{x}_1) & = \emptyset & , \mathit{DepKill}_n(\mathsf{x}_2) = \{x\} \\ \mathit{DepGen}_n(\mathsf{x}_1 \cup \mathsf{x}_1) & = \mathit{DepGen}_n(\mathsf{x}_1) & = \{x\} & , \mathit{DepKill}_n(\mathsf{x}_2) = \emptyset \end{array}$$

  In this case also, the proof obligation is satisfied.

- $\mathit{Opd}(e) \cap \mathsf{x}_1 = \emptyset$ and $\mathit{Opd}(e) \cap \mathsf{x}_2 \neq \emptyset$. Thus $\mathit{Opd}(e) \cap (\mathsf{x}_1 \cup \mathsf{x}_2) \neq \emptyset$.

  This case is similar to the above case.

  ▮

This framework is not fast, and hence is not rapid. We leave it for the reader to construct suitable examples.

4.2.3 Constant Propagation

If it can be asserted at compile time that a given expression would compute a fixed known value at a given program point in every execution of the program, the expression computation can be replaced by the known constant value. This can then be propagated further as the value of the variable to which the result of the expression is assigned. This can help in identifying if other expressions that involve the variable compute a constant value.

For simplicity, we restrict our discussion to integer constants.

DEFINITION 4.4 *A variable $x \in \mathbb{V}\mathsf{ar}$ has a constant value $c \in \mathbb{C}\mathsf{onst}$ at a program point u if for every path reaching u along which a definition of x reaches u, the value of x is c.*

Note that this definition assumes that the program is correct in the sense that no execution path uses a variable before defining it and if the CFG contains a path

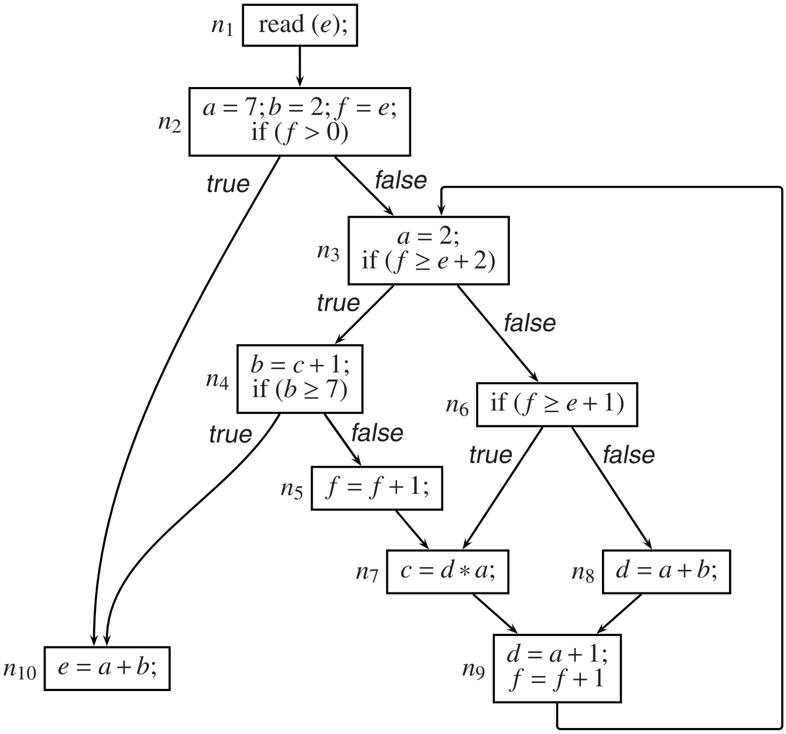


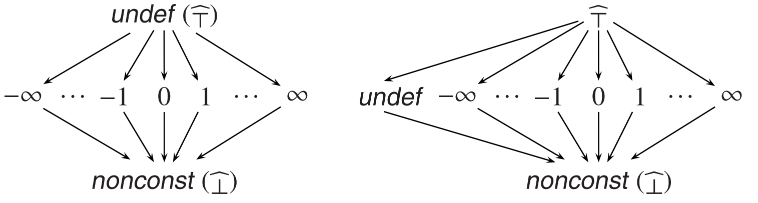
FIGURE 4.4
Program for illustrating constant propagation.

reaching u that does not have any definition of x, such a path can be ignored so long as at least one path containing a definition of x reaches u.

Example 4.3

We use the program in Figure 4.4 as a running example for constant propagation. We have included branching conditions and have labeled out edges of branch nodes with the branch outcomes to emphasize the above assumption about the correctness of program in terms of use and definitions of variables. Observe that, if we ignore the branching conditions, our basic blocks consist of single statements except n_2 and n_9 which contains multiple statements because they are independent of each other. Within the loop, the uses of following variables can be replaced by their statically known values: $a = 2$, $c = 6$, and $d = 3$. Further, $b = 7$ in block n_4. This results in the branching condition in block n_4 being *true* making block n_5 unreachable. ▯

Given a variable x and a program point u, apart from associating integer constants


undef ($\widehat{\top}$)
$-\infty$ ⋯ -1 0 1 ⋯ ∞
nonconst ($\widehat{\bot}$)
$\widehat{\top}$
undef $-\infty$ ⋯ -1 0 1 ⋯ ∞
nonconst ($\widehat{\bot}$)

(a) Assuming that every use is preceded by a definition

(b) Combining detection of possibly uninitialized variables

FIGURE 4.5
$\widehat{L}$ for constant propagation.

with x at u, this analysis associates two additional values: *undef* to indicate that no definition of x has been seen along any path reaching u, and *nonconst* to indicate that x can have different values at u along different paths reaching u. The component lattice $\widehat{L}$ for a variable is shown in Figure 4.5(a).

Observe that the structure of the lattice is governed by the choice of ignoring those control flow paths along which no definition of the variable has been seen. The assumption here is that the program is correct and such paths are not executed in any run of the program or an independent analysis to discover possibly uninitialized variables is being performed.

An alternative policy is to combine the possibly uninitialized variables analysis along with constant propagation. This would require declaring a variable to be *nonconst* at a join point if it has a constant value along a path but is undefined along some other path reaching the program point. This is fair under the assumption that all paths are potential execution paths so the value of the variable is known along some paths and is not known along some other paths. This results in a meet semilattice as illustrated in Figure 4.5(b). In this lattice $\top$ is an artificial element and is required for initialization. The flow functions will have to be suitably extended to include this value.

Such an analysis will discover fewer constants in the program and is more conservative compared to the analysis that excludes those paths that do not contain a definition of the variable under consideration. Hence practically, this policy is usually not adopted. In this book, we restrict ourselves to the common policy of assuming that the program is correct in the sense that every use of a variable is preceded by its definition.

Classical Constant Propagation Using Def-Use Chains

Constant propagation can be performed using def-use chains as described below:

1. Create a work list W_l consisting of definitions of the form $x_i : x = c_i$ occurring in the program, where $x \in \mathbb{V}\mathsf{ar}$ and $c_i \in \mathbb{C}\mathsf{onst}$. The *read*($x$) statement should be treated as a definition $x =$ *nonconst* and must be inserted in the work list.

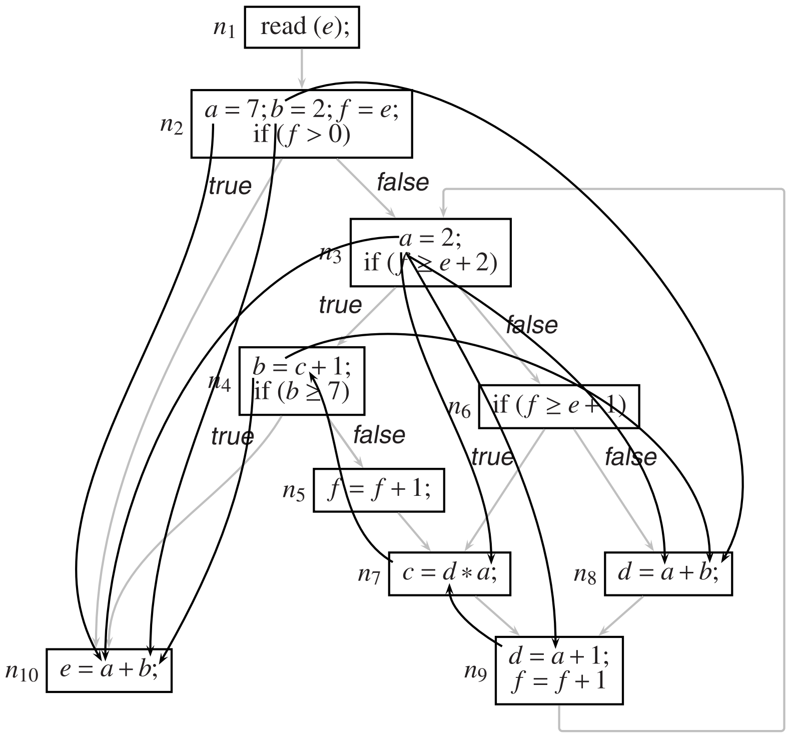

n_1 read (e);
n_2 $a = 7; b = 2; f = e;$ if ($f > 0$)
true
false
n_3 $a = 2;$ if ($f \geq e + 2$)
true
false
n_4 $b = c + 1;$ if ($b \geq 7$)
true
false
n_5 $f = f + 1;$
n_6 if ($f \geq e + 1$)
true
false
n_7 $c = d * a;$
n_8 $d = a + b;$
n_9 $d = a + 1;$ $f = f + 1$
n_{10} $e = a + b;$

FIGURE 4.6
Def-use chains of variables a, b, c, and d for constant propagation.

Repeat the following step until W_l becomes empty.

2. Remove a definition $x_i : x = c_i$ from W_l. Perform the following steps for each def-use chain of x_i.

   (a) Traverse the def-use chain to locate the use of x reachable by the chain.

   (b) Let the value of the use be denoted by x'. If the use of x has not been replaced by any value, then x' is $\widehat{\top}$.

   (c) Replace the use of x by $x' \widehat{\sqcap} c_i$. This then becomes the value of x.

   (d) Evaluate the expression in which the use of x occurs. If the result is a constant value and the expression appears in the right hand side of an assignment, replace the expression by the constant value and add the definition to W_l. If the result is *nonconst*, then add the definition to W_l without replacing the expression.

$\mathit{eval}(e, \mathsf{x})$ where $\mathit{Opd}(e) \cap \mathbb{V}\mathsf{ar} \neq \emptyset$

Notation: $d_1 = \mathit{val}(e_1, \mathsf{x}), d_2 = \mathit{val}(e_2, \mathsf{x})$ where $\{e_1, e_2\} \subseteq (\mathbb{V}\mathsf{ar} \cup \mathbb{C}\mathsf{onst})$

	$e \equiv (e_1\ \mathit{bop}\ e_2)$			$e \equiv (\mathit{uop}\ e_1)$	$e \equiv e_1$	Any other expression
	$d_2 = \widehat{\top}$	$d_2 = \widehat{\bot}$	$d_2 \in \mathbb{C}\mathsf{onst}$			
$d_1 = \widehat{\top}$	$\widehat{\top}$	$\widehat{\bot}$	$\widehat{\top}$	$\widehat{\top}$	$\widehat{\top}$	$\widehat{\bot}$
$d_1 = \widehat{\bot}$	$\widehat{\bot}$	$\widehat{\bot}$	$\widehat{\bot}$	$\widehat{\bot}$	$\widehat{\bot}$	
$d_1 \in \mathbb{C}\mathsf{onst}$	$\widehat{\top}$	$\widehat{\bot}$	$d_1\ \mathit{bop}\ d_2$	$\mathit{uop}\ d_1$	Not Applicable	

FIGURE 4.7
Evaluating constantness of expressions for constant propagation.

Example 4.4
The def-use chains for our running example are shown in Figure 4.6. Initially, the work list contains the assignments to a and b in blocks n_2 and n_7. When we traverse the def-use chains of the definition in block n_3, d is discovered to be 3 in block n_9. This is added to the work list and causes c to become 6 in block n_7. This cause b to become 7 in block n_4. Since this is a compile time evaluation, it is valid for every possible execution of n_4 and block n_5 is never executed. Interestingly, compile time analysis concludes that b can have different values in n_8 and n_{10} and hence is $\widehat{\bot}$. For n_8, this is conservative imprecision since the execution never reaches n_8 after b becomes 7 in n_4. ▯

Data Flow Analysis for Constant Propagation

Observe that the specification of constant propagation in terms of def-use chains has a highly operational flavor. Data flow equations provide a declarative mechanism of defining a program analysis and reduce the work to fixed point computation.

Data flow analysis for constant propagation uses an overall lattice L that is a product of $\widehat{L}$. For convenience of defining flow functions, we represent an element in L by sets of pairs $\langle x, d_x \rangle$ where $x \in \mathbb{V}\mathsf{ar}$ and $d_x \in \widehat{L}$.

This is a forward data flow analysis. The data flow equations are:

$$\mathit{In}_n = \begin{cases} \mathit{BI} & n \text{ is } \mathit{Start} \\ \displaystyle\bigsqcap_{p \in \mathit{pred}(n)} \mathit{Out}_p & \text{otherwise} \end{cases} \tag{4.8}$$

$$\mathit{Out}_n = f_n(\mathit{In}_n) \tag{4.9}$$

BI contains pairs $\langle x, \widehat{\top} \rangle$ for all variables $x \in \mathbb{V}\mathsf{ar}$. The confluence operation $\sqcap$ on elements in L is defined in terms of $\widehat{\sqcap}$ by applying it to pairs of the same variable:

$$\forall \mathsf{x}_1, \mathsf{x}_2 \in L,\ \mathsf{x}_1 \sqcap \mathsf{x}_2 = \{\langle z, d_x \widehat{\sqcap} d_y \rangle \mid \langle z, d_x \rangle \in \mathsf{x}_1, \langle z, d_y \rangle \in \mathsf{x}_2, x \in \mathbb{V}\mathsf{ar}\}$$

$$
\begin{aligned}
\mathit{ConstGen}_n &= \begin{cases} \{\langle x, \mathit{eval}(e, \top)\rangle\} & n \text{ is assignment } x = e, \mathit{Opd}(e) \subseteq \mathbb{C}\mathrm{onst} \\ \{\langle x, \widehat{\bot}\rangle\} & n \text{ is } \mathit{read}(x) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{DepGen}_n(\mathsf{x}) &= \begin{cases} \{\langle x, \mathit{eval}(e, \mathsf{x})\rangle\} & n \text{ is assignment } x = e, \mathit{Opd}(e) \cap \mathbb{V}\mathrm{ar} \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{ConstKill}_n &= \emptyset \\
\mathit{DepKill}_n(\mathsf{x}) &= \begin{cases} \{\langle x, d\rangle\} & n \text{ is assignment } x = e,\ \langle x, d\rangle \in \mathsf{x} \\ \{\langle x, d\rangle\} & n \text{ is } \mathit{read}(x),\ \langle x, d\rangle \in \mathsf{x} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

Function *eval* is defined in Figure 4.7. It uses $\mathit{val}(e, \mathsf{x})$ to denote the value of a simple expression (consisting of a variable or a constant) in the context of the given data flow information x:

$$
\mathit{val}(e, \mathsf{x}) = \begin{cases} c & \text{if } e \text{ is } c \in \mathbb{C}\mathrm{onst} \\ d & \text{if } e \text{ is } x \in \mathbb{V}\mathrm{ar},\ \langle x, d\rangle \in \mathsf{x} \end{cases}
$$

Example 4.5

The computation of data flow values for our running example of Figure 4.4 has been shown in Figure 4.8. For brevity, we represent the data flow information as a vector $\langle d_a, d_b, d_c, d_d, d_e\rangle$ where d_x represents the constantness value of variable x. *BI* is $\langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}\rangle$. The initial value of In_i and Out_i for all i is $\top = \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}\rangle$.

Observe that this analysis requires four traversals over the control flow graph in reverse postorder. In the first iteration, d is discovered to be 3 in block n_9. Thus, c is discovered to be 6 block n_7 in the third iteration. This makes b a constant with value 7 at $\mathit{Exit}(n_3)$ in the fourth iteration. At $\mathit{Entry}(n_2)$, b is 2 along the path from n_1 and 7 along the path from n_6. Observe the non-separability of constant propagation: The constantness of b depends on the constantness of a through c and d.

Also note that b is $\widehat{\bot}$ in n_8 due to the effect of n_4 in spite of the fact that control never reaches n_8 after execution n_4. □

Properties of Constant Propagation Data Flow Framework

In this section we show that Constant Propagation framework is monotonic but non-distributive.

THEOREM 4.1

Constant Propagation framework is monotonic.

	Iteration #1	Changes in iteration #2	Changes in iteration #3	Changes in iteration #4
In_{n_1}	$\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\top}$			
Out_{n_1}	$\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\top}$			
In_{n_2}	$\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$			
Out_{n_2}	$7,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$			
In_{n_3}	$7,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},2,6,3,\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$
Out_{n_3}	$2,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$
In_{n_4}	$2,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$
Out_{n_4}	$2,\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,\widehat{\top},\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,7,6,3,\widehat{\bot},\widehat{\bot}$	
In_{n_5}	$2,\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,\widehat{\top},\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,7,6,3,\widehat{\bot},\widehat{\bot}$	
Out_{n_5}	$2,\widehat{\top},\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,\widehat{\top},\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,7,6,3,\widehat{\bot},\widehat{\bot}$	
In_{n_6}	$2,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$
Out_{n_6}	$2,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$
In_{n_7}	$2,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$	
Out_{n_7}	$2,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$	
In_{n_8}	$2,2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$
Out_{n_8}	$2,2,\widehat{\top},4,\widehat{\bot},\widehat{\bot}$	$2,2,\widehat{\top},4,\widehat{\bot},\widehat{\bot}$	$2,2,6,4,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,\widehat{\bot},\widehat{\bot},\widehat{\bot}$
In_{n_9}	$2,2,\widehat{\top},4,\widehat{\bot},\widehat{\bot}$	$2,2,6,\widehat{\bot},\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,\widehat{\bot},\widehat{\bot},\widehat{\bot}$	
Out_{n_9}	$2,2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$2,2,6,3,\widehat{\bot},\widehat{\bot}$	$2,\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$	
$\mathit{In}_{n_{10}}$	$\widehat{\bot},2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$	
$\mathit{Out}_{n_{10}}$	$\widehat{\bot},2,\widehat{\top},\widehat{\top},\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},2,\widehat{\top},3,\widehat{\bot},\widehat{\bot}$	$\widehat{\bot},\widehat{\bot},6,3,\widehat{\bot},\widehat{\bot}$	

FIGURE 4.8
Constant propagation data flow analysis for the running example in Figure 4.4.

PROOF Showing monotonicity of $f_n(\mathsf{x})$ requires showing that $(\mathsf{x} - \mathit{DepKill}_n(\mathsf{x}))$ and $\mathit{DepGen}_n(\mathsf{x})$ are monotonic.

$\mathit{DepKill}_n(\mathsf{x})$ is $\{\langle x, d_x\rangle\}$ for assignment $x = e$ or $\mathit{read}(x)$. In all other cases it is $\emptyset$. Since it does not depend on x,

$$\forall \mathsf{x}_1 \sqsubseteq \mathsf{x}_2 \in L:\ (\mathsf{x}_1 - \mathit{DepKill}_n(\mathsf{x}_1)) \ \sqsubseteq\ (\mathsf{x}_2 - \mathit{DepKill}_n(\mathsf{x}_2))$$

Showing monotonicity of $\mathit{DepGen}_n(\mathsf{x})$ reduces to showing

$$\forall e \in \mathbb{E}\mathsf{xpr}, \forall \mathsf{x}_1, \mathsf{x}_2 \in L : \mathsf{x}_1 \sqsubseteq \mathsf{x}_2 \Rightarrow \mathit{eval}(e, \mathsf{x}_1) \mathrel{\widehat{\sqsubseteq}} \mathit{eval}(e, \mathsf{x}_2)$$

Function $\mathit{eval}(e, \mathsf{x})$ examines the data flow values of the operands of e. From its definition in Figure 4.7, it is easy to see that the data flow value computed by $\mathit{eval}(e, \mathsf{x})$ preserves the partial order. ∎

THEOREM 4.2
Constant Propagation framework is non-distributive.

PROOF Using the arguments similar to those in Theorem 4.1, it can be shown in terms of *eval*().

$$\exists e \in \mathbb{E}\mathsf{xpr},\ \exists \mathsf{x}_1, \mathsf{x}_2 \in L : \mathit{eval}(e, \mathsf{x}_1 \sqcap \mathsf{x}_2) \neq \mathit{eval}(e, \mathsf{x}_1)\ \widehat{\sqcap}\ \mathit{eval}(e, \mathsf{x}_2)$$

This is demonstrated by expression $(a+b)$ in block n_{10} in the program in Figure 4.4 for $\mathsf{x}_1 = \langle 7, 2, -, -, - \rangle$ and $\mathsf{x}_2 = \langle 2, 7, -, -, - \rangle$ where "–" indicates the values which do not matter. ▮

Presence of non-distributivity shows the limits of static analysis: Unless all paths are traversed independently, which may require exponential amount of work, a static analysis is likely to miss out on useful information even if the information is independent of program execution. This happens because of sharing of information across distinct paths as shown by the following example.

Example 4.6

Only two execution paths reach n_{10}: (n_1, n_2, n_7), and $(n_1, n_2, n_3, n_6, n_8, n_9, n_3, n_6, n_7, n_9, n_3, n_4, n_{10})$. The values of a, b, and e at *Exit*(n_{10}) along the first path are 7, 2, and 9 respectively whereas along the second path they are 2, 7, and 9. Static summary of constantness information should conclude that a and b are $\widehat{\bot}$ and e is 9. However, due to non-distributivity, our analysis concludes that all the three variables are $\widehat{\bot}$. Effectively, the flow function in n_{10} uses all possible combinations of a and b including those across different paths: $a = 7$ and $b = 2$ resulting in $e = 9$; $a = 2$ and $b = 7$ resulting in $e = 9$; $a = 2$ and $b = 2$ resulting in $e = 4$; $a = 7$ and $b = 7$ resulting in $e = 14$. Observe that the last two combinations are infeasible because there is no execution path reaching n_{10} along which a and b can both be 2 or both be 7. Fortunately, the imprecision caused by non-distributivity is safe because a $\widehat{\bot}$ variable does not enable any transformation. □

Constant propagation is not fast, and hence is not rapid. We leave it for the reader to construct suitable examples.

4.2.4 Variants of Constant Propagation

Constant propagation is a very useful analysis in practice. It improves the efficiency of programs by advancing some computations to compile time. It facilitates many other optimizations such as elimination of dead code (i.e., assignments which define variables which are not used later) as well as unreachable code. The latter simplifies control flow and may reduce branch delays on pipelined architectures. It can help in strength reduction and may enable many loop optimizations that require loop iterations bounds to be known at compile time.

It is not surprising that many variants of constant propagation have been proposed. The formulation which we have presented in the preceding sections is called *full* constant propagation to distinguish it from other variants of constant propagation which restrict the analysis in some ways.

Conditional Constant Propagation

As observed in Examples 4.3, 4.4, and 4.5, the value $b = 7$ in n_4 causes the control flow to leave the loop. Block n_5 is never executed and the value 7 does not reach n_8 resulting in both b and d being constant in n_8. Conditional constant propagation can discover this by evaluating the branching conditions appearing on execution paths.

In order to achieve the above, we create a lattice $\{\mathit{reachable}, \mathit{notReachable}\}$ with the partial order $\mathit{notReachable} \sqsubseteq \mathit{reachable}$. Let L be the product lattice of $\widehat{L}$. We create a new product lattice $L_c = \{\mathit{reachable}, \mathit{notReachable}\} \times L$. Values in L_c are pairs $\langle \mathit{status}, \mathsf{x}\rangle$ where status is either $\mathit{reachable}$ or $\mathit{notReachable}$ and x is the constantness information as discovered in the unconditional constant propagation. The confluence operation $\sqcap_c$ of values in L_c ignores the values which are not reachable and is as defined below.

$\langle \mathit{status}_1, \mathsf{x}_1\rangle \sqcap_c \langle \mathit{status}_2, \mathsf{x}_2\rangle$		
	$\mathit{status}_2 = \mathit{reachable}$	$\mathit{status}_2 = \mathit{notReachable}$
$\mathit{status}_1 = \mathit{reachable}$	$\langle \mathit{reachable}, \mathsf{x}_1 \sqcap \mathsf{x}_2\rangle$	$\langle \mathit{reachable}, \mathsf{x}_1\rangle$
$\mathit{status}_1 = \mathit{notReachable}$	$\langle \mathit{reachable}, \mathsf{x}_2\rangle$	$\langle \mathit{notReachable}, \top\rangle$

Reachability status is determined by evaluating branching conditions using function $\mathit{evalCond}(m, \mathsf{x})$ which computes true, false, or $\mathit{undefined}$ depending upon the following: If basic block m contains a condition at the end and data flow information x contains constant values for all variables required to evaluate the condition, then $\mathit{evalCond}(m, \mathsf{x})$ is the result of the condition. Otherwise, $\mathit{evalCond}(m, \mathsf{x})$ is $\mathit{undefined}$. Propagation of data flow information along the out edge associated with the outcome is ensured by using an edge flow function.

An alternative to such a data flow analysis is to simply delete the edge that will not be executed instead of qualifying data flow information with $\mathit{reachable}$ and $\mathit{notReachable}$ values. However, this may not be possible if branch outcome is likely to be influenced by calling contexts. In particular, when context sensitive interprocedural data flow analysis is performed a branch outcome may be different in different contexts and deletion of an edge may not be possible. Further, the abstraction of conditional propagation along edges is a powerful mechanism that can compute more precise data flow information for analyses such as null pointer analysis: For this analysis, a condition that checks for the nullity of a pointer can propagate different outcomes along the two out edges of a condition.

The propagation function for an edge $m \rightarrow n$, is defined as follows:

$$g_{m\rightarrow n}(\mathit{status}, \mathsf{x}) = \begin{cases} \langle \mathit{notReachable}, \top\rangle & \mathit{evalCond}(m, \mathsf{x}) \neq \mathit{undefined} \text{ and} \\ & \mathit{evalCond}(m, \mathsf{x}) \neq \mathit{label}(m \rightarrow n) \\ \langle \mathit{status}, \mathsf{x}\rangle & \text{otherwise} \end{cases}$$

	Iteration #1	Changes in iteration #2	Changes in iteration #3
In_{n_1}	$R, \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$		
Out_{n_1}	$R, \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\top} \rangle$		
In_{n_2}	$R, \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$		
Out_{n_2}	$R, \langle 7, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$		
In_{n_3}	$R, \langle 7, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
Out_{n_3}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
In_{n_4}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
Out_{n_4}	$R, \langle 2, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, \widehat{\top}, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 7, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
In_{n_5}	$R, \langle 2, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, \widehat{\top}, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 7, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
Out_{n_5}	$R, \langle 2, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, \widehat{\top}, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 7, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
In_{n_6}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$N, \top = \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$
Out_{n_6}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$N, \top = \langle \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top}, \widehat{\top} \rangle$
In_{n_7}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
Out_{n_7}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
In_{n_8}	$R, \langle 2, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
Out_{n_8}	$R, \langle 2, 2, \widehat{\top}, 4, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, \widehat{\top}, 4, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 4, \widehat{\bot}, \widehat{\bot} \rangle$
In_{n_9}	$R, \langle 2, 2, \widehat{\top}, 4, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, \widehat{\bot}, 6, \widehat{\bot}, \widehat{\bot}, \widehat{\bot} \rangle$
Out_{n_9}	$R, \langle 2, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, 2, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle 2, \widehat{\bot}, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
$In_{n_{10}}$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, \widehat{\bot}, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$
$Out_{n_{10}}$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, \widehat{\top}, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, 2, \widehat{\top}, 3, \widehat{\bot}, \widehat{\bot} \rangle$	$R, \langle \widehat{\bot}, \widehat{\bot}, 6, 3, \widehat{\bot}, \widehat{\bot} \rangle$

FIGURE 4.9
Conditional constant propagation for the running example in Figure 4.4.

The data flow equations remain much the same except that now they must honor the reachability status as shown below.

$$In_n = \begin{cases} \langle \mathit{reachable}, \mathit{BI} \rangle & n \text{ is } \mathit{Start} \\ \displaystyle\prod_{p \in \mathit{pred}(n)}{}_{C} \; g_{p \to n}(\mathit{Out}_p) & \text{otherwise} \end{cases}$$

$$Out_n = \begin{cases} \langle \mathit{reachable}, f_n(\mathsf{x}) \rangle & In_n = \langle \mathit{reachable}, \mathsf{x} \rangle \\ \langle \mathit{notReachable}, \top \rangle & \text{otherwise} \end{cases}$$

In the beginning, only the *Start* block is assumed to be reachable and the initial value associated with all other program points is $\langle \mathit{notReachable}, \top \rangle$. This is required for computing the *MFP* solution. If we use the initial value $\langle \mathit{reachable}, \top \rangle$, the analysis will converge on a fixed point that may not be maximum. The result would be imprecise but safe.

Example 4.7

Figure 4.9 provides the data flow values for conditional constant propagation in our running example. Since the data flow information is not propagated from n_4 to n_5, b remains constant in the loop and analysis converges in three iterations rather than four. □

It is easy to see that conditional constant propagation can discover more precise information than unconditional constant propagation. It is guaranteed to be at least as good, if not better.

Copy Constant Propagation

Copy constant propagation limits the expressions appearing on the right hand side of an assignment to simple variables or constants. Such statements have been called copies in Section 2.3.4 to describe copy propagation using reaching definitions. There are two fundamental differences between the analysis presented here and the copy propagation described in Section 2.3.4: (a) the analysis presented here allows replacement of variables by constants only whereas the earlier analysis allowed replacement of variables by other variables also, and (b) the analysis presented here takes care of transitive effects of replacements whereas the earlier analysis does not do so.

Copy constant propagation does not generate new constants based on the values of variables. Hence it is guaranteed to compute only a finite number of constants. Thus the component lattice $\widehat{L}$ is finite. The flow function does not evaluate any expression involving a variable. Thus the definitions of $\mathit{ConstKill}_n$ and $\mathit{DepKill}_n(\mathsf{x})$ do not change. $\mathit{ConstGen}_n$ and $\mathit{DepGen}_n(\mathsf{x})$ change in the following manner. $\mathit{DepGen}_n(\mathsf{x})$ is restricted to copy assignments and $\mathit{DepGen}_n(\mathsf{x})$ computes $\widehat{\bot}$ value for non-copy assignments. The new definitions are:

$$\mathit{ConstGen}_n = \begin{cases} \{\langle v, \mathit{eval}(e, \top)\rangle\} & n \text{ is assignment } v = e, \mathit{Opd}(e) \subseteq \mathbb{C}\mathsf{onst} \\ \{\langle v, \widehat{\bot}\rangle\} & n \text{ is } \mathit{read}(v) \text{ or a non-copy assignment to } v \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{DepGen}_n(\mathsf{x}) = \begin{cases} \{\langle v, d\rangle\} & n \text{ is assignment } v = w,\ \langle w, d\rangle \in \mathsf{x} \\ \emptyset & \text{otherwise} \end{cases}$$

Observe that the expression evaluation in the above definition is restricted to constant operands only.

Full constant propagation is non-distributive due to the use of function $\mathit{eval}(e, \mathsf{x})$. All other terms involved in defining f_n are distributive. Copy constant propagation is distributive because it does not involve $\mathit{eval}(e, \mathsf{x})$. However, due to non-separability, the framework remains non-fast.

Since expressions are not evaluated, this analysis finds fewer constants and is limited in scope compared to the full constant propagation.

Linear Constant Propagation

A slightly more general formulation than copy constant propagation allows expressions to appear on the right hand side but these expressions may contain at most a single variable. This requires a restricted version of *eval*. Since this analysis computes new constants, the lattice $\widehat{L}$ is infinite, unlike copy constant propagation. However, similar to copy constant propagation, linear constant propagation is distributive because the number of variables in the right hand side is restricted to one. However, due to non-separability, the framework remains non-fast.

4.3 Discovering Properties of Pointers

Pointers allow indirect modification of data thereby making it difficult to discover useful information from programs. They reduce the effectiveness of program analysis tools. This is because, in the absence of precise analysis of pointer manipulations, program analysis must conservatively assume that any data object could be modified by any pointer. Practically, this can be mitigated somewhat by using type information and by confining the conservative assumptions within variables of the same type. However, if information about the possible manipulations performed through pointers is available, it can enhance the precision of other analyses.

This section presents pointer analyses for stack and static data. These analyses capture relationships between pointers and other variables or pointers. This is different from other analyses which we have seen because the domain of data flow values of an entity did not involve other entities.

Our model of pointer manipulations is based on C except that we do not take into account pointer arithmetic. Since the size of stack and static data is fixed, pointers can point to only a fixed set of locations that are known at compile time. We assume that field references of a structure are flattened out into a new pointer name: A reference like $x.f$ occurring in a statement can be modeled as a new pointer x_f to which the pointer x points to. Further, a *null* assignment to a pointer x is treated as assigning address 0 to x. Thus assignment x = *null* is treated as x = &*zero* where *zero* is a special symbol whose address is 0.

4.3.1 Points-To Analysis of Stack and Static Data

This analysis establishes *points-to* relation between pointer variables and memory locations under the assumption that the program is type correct in terms of pointer manipulations.

DEFINITION 4.5 *A pointer variable x **points to** variable y at a program point u, denoted $x \rightarrow y$, if it holds the address of variable y at u.*

Points-to relation is neither reflexive (because $x \rightarrow x$ may not hold), nor symmetric (because $x \rightarrow y \not\Rightarrow y \rightarrow x$), nor transitive (because $x \rightarrow y, y \rightarrow z \not\Rightarrow x \rightarrow z$).

We assume that the left hand side of a pointer assignment is either a pointer variable x or a pointer indirection $*x$. The right hand side could be either an address expression $\&x$, a pointer variable x, or a pointer indirection $*x$.

The pointers which are likely to be modified by a pointer assignment are called *left locations* of the assignment. The addresses which may be assigned to the left locations are called the *right locations* of the assignment. Let x be the set representing the points-to relations that hold just before assignment statement n. The left and the right locations of n which depend on x are denoted by $\mathit{DepLeftL}_n(\mathsf{x})$ and $\mathit{DepRightL}_n(\mathsf{x})$. The left and right locations which depend solely on the local effect of n are denoted by $\mathit{ConstLeftL}_n$ and $\mathit{ConstRightL}_n$.

Consider an assignment statement $\mathit{lhs}_n = \mathit{rhs}_n$. The left and the right locations of n are defined as follows:

Left Locations			Right Locations		
lhs_n	$\mathit{ConstLeftL}_n$	$\mathit{DepLeftL}_n(\mathsf{x})$	rhs_n	$\mathit{ConstRightL}_n$	$\mathit{DepRightL}_n(\mathsf{x})$
x	$\{x\}$	$\emptyset$	x	$\emptyset$	$\{y \mid (x \rightarrow y) \in \mathsf{x}\}$
$*x$	$\emptyset$	$\{y \mid (x \rightarrow y) \in \mathsf{x}\}$	$*x$	$\emptyset$	$\{z \mid \{x \rightarrow y, y \rightarrow z\} \subseteq \mathsf{x}\}$
			$\&x$	$\{x\}$	$\emptyset$

Points-to relation between the left and the right locations is established in terms of new points-to pairs which are generated and the points-to pairs which cease to hold due to the effect of a basic block.

$$\begin{aligned}
\mathit{ConstGen}_n &= \{x \rightarrow y \mid x \in \mathit{ConstLeftL}_n, y \in \mathit{ConstRightL}_n\} \\
\mathit{DepGen}_n(\mathsf{x}) &= \{x \rightarrow y \mid (x \in \mathit{ConstLeftL}_n, y \in \mathit{DepRightL}_n(\mathsf{x})), \text{ or} \\
&\qquad\qquad (x \in \mathit{DepLeftL}_n(\mathsf{x}), y \in \mathit{ConstRightL}_n), \text{ or} \\
&\qquad\qquad (x \in \mathit{DepLeftL}_n(\mathsf{x}), y \in \mathit{DepRightL}_n(\mathsf{x}))\} \\
\mathit{ConstKill}_n &= \{x \rightarrow y \mid x \in \mathit{ConstLeftL}_n\} \\
\mathit{DepKill}_n(\mathsf{x}) &= \{x \rightarrow y \mid x \in \mathit{DepLeftL}_n(\mathsf{x})\}
\end{aligned}$$

$\mathit{DepKill}_n(\mathsf{x})$ depends on $\mathit{DepLeftL}_n(\mathsf{x})$ which involves pointer indirection on the left hand side of a pointer assignment. Thus it captures the indirect effect of an assignment due to pointer indirection and hence the choice of x is critical for ensuring conservative approximation on the safer side. We explain this below.

DEFINITION 4.6 *If a pointer z is modified by a pointer assignment regardless of the execution path taken to reach the assignment, then such a modification is called a strong update of z. If z may be modified by the assignment when the execution reaches along some path and may not be modified when it reaches along some other path, such a modification of z is called a weak update of z.*

An assignment $z = w$ causes a strong update of z. Contrast this with the assignment $*x = w$ such that $x \rightarrow z$ holds along some path reaching the assignment. If the execution follows this path, then the assignment modifies z, otherwise it does not modify z. If $x \rightarrow z$ holds along every path, then z is modified by the assignment in every execution. In order to capture the indirect effect of such an assignment, there is a need to make a distinction between the points-to relations which cause weak updates from those which cause strong updates. The former is called *may* points-to relation while the latter is called *must* points-to relation.

DEFINITION 4.7 *If pointer x holds the address of variable y at program point u along some path from* Start *to u, then $x \rightarrow y$ at u under* may *points-to relation. If x holds the address of y along every path from* Start *to u, then $x \rightarrow y$ at u under* must *points-to relation.*

It is easy to see that a *may* points-to relation is weaker than a *must* points-to relation: If x must point to y at u then x may point to y at u but not vice-versa.

Since *may* points-to relations must not miss any points-to pair which may hold at a program point, only the pairs affected by a strong update must be removed. Thus, for computing MayOut_n, $\mathit{DepKill}_n$ must depend on MustIn_n. Since *must* points-to relations should include a points-to pair only if it is guaranteed to hold, all pairs affected by weak update must be removed. Thus, for computing $\mathit{MustOut}_n$, $\mathit{DepKill}_n$ must depend on MayIn_n. We explain this in Example 4.9.

The data flow equations for points-to analysis are:

$$\mathit{MayIn}_n = \begin{cases} \mathit{BI} & n \text{ is } \mathit{Start} \\ \bigcup\limits_{p \in \mathit{pred}(n)} \mathit{MayOut}_n & \text{otherwise} \end{cases} \tag{4.10}$$

$$\mathit{MayOut}_n = f_n(\mathit{MayIn}_n, \mathit{MustIn}_n) \tag{4.11}$$

$$\mathit{MustIn}_n = \begin{cases} \mathit{BI} & n \text{ is } \mathit{Start} \\ \bigcap\limits_{p \in \mathit{pred}(n)} \mathit{MustOut}_n & \text{otherwise} \end{cases} \tag{4.12}$$

$$\mathit{MustOut}_n = f_n(\mathit{MustIn}_n, \mathit{MayIn}_n) \tag{4.13}$$

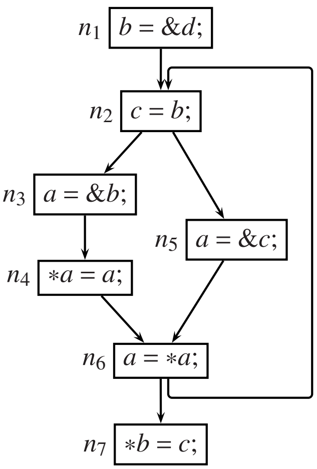
where flow function f_n is defined as follows:

$$f_n(\mathsf{x}_1, \mathsf{x}_2) = (\mathsf{x}_1 - \mathit{Kill}_n(\mathsf{x}_2)) \cup \mathit{Gen}_n(\mathsf{x}_1) \tag{4.14}$$

Observe the use of different sets as arguments to $\mathit{Gen}_n(\mathsf{x})$ and $\mathit{Kill}_n(\mathsf{x})$. The $\mathit{Gen}_n(\mathsf{x})$ and $\mathit{Kill}_n(\mathsf{x})$ are defined in the usual manner:

$$\begin{aligned} \mathit{Gen}_n(\mathsf{x}) &= \mathit{ConstGen}_n \cup \mathit{DepGen}_n(\mathsf{x}) \\ \mathit{Kill}_n(\mathsf{x}) &= \mathit{ConstKill}_n \cup \mathit{DepKill}_n(\mathsf{x}) \end{aligned}$$

In the intraprocedural context, *BI* is $\emptyset$ for both *may* and *must* point-to because no pointer points to any variable at *Start*.



- $\mathbb{V}\mathsf{ar} = \{a, b, c, d\}$
  $\mathbb{U} = \{\ a{\to}a,\ a{\to}b,\ a{\to}c,\ a{\to}d,$
  $b{\to}a,\ b{\to}b,\ b{\to}d,\ b{\to}d,$
  $c{\to}a,\ c{\to}b,\ c{\to}c,\ c{\to}d,$
  $d{\to}a,\ d{\to}b,\ d{\to}c,\ d{\to}d\ \}$
- $L_{\mathsf{may}} = \langle 2^{\mathbb{U}}, \supseteq \rangle$, $\top_{\mathsf{may}} = \emptyset, \bot_{\mathsf{may}} = \mathbb{U}$
- $L_{\mathsf{must}} = \widehat{L}_a \times \widehat{L}_b \times \widehat{L}_c \times \widehat{L}_d$
  We show the component lattice $\widehat{L}_a$:

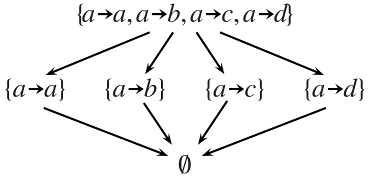


FIGURE 4.10
Example program for points-to analysis.

Figure 4.10 illustrates the lattices for *may* and *must* points-to analysis. Observe that the $\widehat{\top}$ value for *must* points-to in Figure 4.10 is $\{a{\to}a, a{\to}b, a{\to}c, a{\to}d\}$. It is easy to see to that this is a value that cannot naturally occur in any instance of *must* points-to analysis because a pointer can pointer to at most one location in *must* points-to analysis. This is an example of an artificial value added to a meet semilattice for convenience. Since the descending chain condition is satisfied, the resulting lattice is a complete lattice. By contrast, the lattice for *may* points-to analysis is a naturally complete lattice and its $\widehat{\top}$ element can actually occur during *may* points-to analysis.

Technically, the lattice for *must* points-to analysis is a tuple of values from component lattice. For example, given the lattice in Figure 4.10, if a points to c, b does not point to any location, c points to d, and d points to b, then the *must* points-to information should be represented as $\langle \{a{\to}b\}, \emptyset, \{c{\to}d\}, \{d{\to}b\} \rangle$. However, for compatibility with *may* points-to analysis, we treat it as a flattened set rather than as a vector of sets for each pointer variable. Thus, we represent the same data flow information by $\{a{\to}b, c{\to}d, d{\to}b\}$.

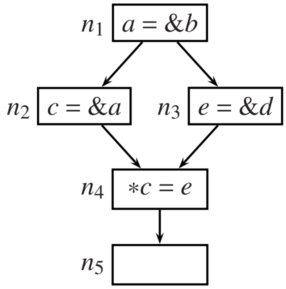
Example 4.8
Consider the example program in Figure 4.10. The computation of *may* and *must* points-to relations has been shown below. The $\top$ for *may* is $\emptyset$ whereas that for *must* is the universal set $\mathbb{U}$ of points-to pairs. The computation of *may* and *must* proceeds in an interleaved fashion.

	Iteration #1	Changes in Iteration #2	Changes in Iteration #3
MayIn_{n_1}	$\emptyset$		
MustIn_{n_1}	$\emptyset$		
MayOut_{n_1}	$\{b\rightarrow d\}$		
$\mathit{MustOut}_{n_1}$	$\{b\rightarrow d\}$		
MayIn_{n_2}	$\{b\rightarrow d\}$	$\{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow d\}$	$\{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d\}$
MustIn_{n_2}	$\{b\rightarrow d\}$	$\emptyset$	
MayOut_{n_2}	$\{b\rightarrow d, c\rightarrow d\}$	$\{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d\}$	
$\mathit{MustOut}_{n_2}$	$\{b\rightarrow d, c\rightarrow d\}$	$\emptyset$	
MayIn_{n_3}	$\{b\rightarrow d, c\rightarrow d\}$	$\{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d\}$	
MustIn_{n_3}	$\{b\rightarrow d, c\rightarrow d\}$	$\emptyset$	
MayOut_{n_3}	$\{a\rightarrow b, b\rightarrow d, c\rightarrow d\}$	$\{a\rightarrow b, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d\}$	
$\mathit{MustOut}_{n_3}$	$\{a\rightarrow b, b\rightarrow d, c\rightarrow d\}$	$\{a\rightarrow b\}$	
MayIn_{n_4}	$\{a\rightarrow b, b\rightarrow d, c\rightarrow d\}$	$\{a\rightarrow b, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d\}$	
MustIn_{n_4}	$\{a\rightarrow b, b\rightarrow d, c\rightarrow d\}$	$\{a\rightarrow b\}$	
MayOut_{n_4}	$\{a\rightarrow b, b\rightarrow b, c\rightarrow d\}$	$\{a\rightarrow b, b\rightarrow b, c\rightarrow b, c\rightarrow d\}$	
$\mathit{MustOut}_{n_4}$	$\{a\rightarrow b, b\rightarrow b, c\rightarrow d\}$	$\{a\rightarrow b, b\rightarrow b\}$	
MayIn_{n_5}	$\{b\rightarrow d, c\rightarrow d\}$	$\{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d\}$	
MustIn_{n_5}	$\{b\rightarrow d, c\rightarrow d\}$	$\emptyset$	
MayOut_{n_5}	$\{a\rightarrow c, b\rightarrow d, c\rightarrow d\}$	$\{a\rightarrow c, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d\}$	
$\mathit{MustOut}_{n_5}$	$\{a\rightarrow c, b\rightarrow d, c\rightarrow d\}$	$\{a\rightarrow c\}$	
MayIn_{n_6}	$\{a\rightarrow b, a\rightarrow c, b\rightarrow b, b\rightarrow d, c\rightarrow d\}$	$\{a\rightarrow b, a\rightarrow c, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d\}$	
MustIn_{n_6}	$\{c\rightarrow d\}$	$\emptyset$	
MayOut_{n_6}	$\{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow d\}$	$\{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d\}$	
$\mathit{MustOut}_{n_6}$	$\{c\rightarrow d\}$	$\emptyset$	
MayIn_{n_7}	$\{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow d\}$	$\{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d\}$	
MustIn_{n_7}	$\{c\rightarrow d\}$	$\emptyset$	
MayOut_{n_7}	$\{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow d, d\rightarrow d\}$	$\{a\rightarrow b, a\rightarrow d, b\rightarrow b, b\rightarrow d, c\rightarrow b, c\rightarrow d, d\rightarrow b, d\rightarrow d\}$	
$\mathit{MustOut}_{n_7}$	$\{c\rightarrow d\}$	$\emptyset$	

Since $(a\rightarrow b) \in \mathit{MayIn}_{n_4}$, assignment $*a = a$ generates $(b\rightarrow b) \in \mathit{MayOut}_{n_4}$. Further, since $(a\rightarrow b) \in \mathit{MustIn}_{n_4}$, this assignment causes a strong update of b causing the removal of $b\rightarrow d$ from MayIn_{n_4}. The third iteration is required $c\rightarrow b$ from MayOut_{n_2} to MayIn_{n_2}. □

Example 4.9

Consider the program flow graph in Figure 4.11 on the next page which illus-



- $a \rightarrow b$ in block 5 along path $1,3,4,5$ but not along path $1,2,4,5$.
- Required: $a \rightarrow b \in \mathit{MayIn}_{n_5}$ and $a \rightarrow b \notin \mathit{MustIn}_{n_5}$
- If $\mathit{DepKill}_{n_4}$ for MayOut_{n_4} is defined in terms of MayIn_{n_4} then $a \rightarrow b \notin \mathit{MayOut}_{n_4}$ because a is in $\mathit{DepLeftL}_{n_4}(\mathit{MayIn}_{n_4})$
- If $\mathit{DepKill}_{n_4}$ for $\mathit{MustOut}_{n_4}$ is defined in terms of MustIn_{n_4} then $a \rightarrow b \notin \mathit{MustOut}_{n_4}$ because a is in $\mathit{DepLeftL}_{n_4}(\mathit{MustIn}_{n_4})$

FIGURE 4.11
Inverse dependence of *may* and *must* points-to relations for *Kill*.

trates that the dependence between *may* and *must* points-to relations for *Kill* is not only mutual, but is also inverse. In block n_5, the relation $a \rightarrow b$ holds along the path (n_1,n_3,n_4,n_5) but not along the path (n_1,n_2,n_4,n_5). This is because along the latter path, $c \rightarrow a$ and the assignment in n_4 modifies a. Since $c \rightarrow a \in \mathit{MayIn}_{n_4}$ and $c \rightarrow a \notin \mathit{MustIn}_{n_4}$, a is a left location in *may* points-to relation but not in *must* points-to relation. Thus if MayIn_{n_4} is used for defining $\mathit{DepKill}_{n_4}$ for computing MayOut_{n_4}, $a \rightarrow b$ will not exist in MayOut_{n_4} which is incorrect. Similarly, if MustIn_{n_4} is used for defining $\mathit{DepKill}_{n_4}$ for computing $\mathit{MustOut}_{n_4}$, $a \rightarrow b$ will exist in $\mathit{MustOut}_{n_4}$ which is incorrect. □

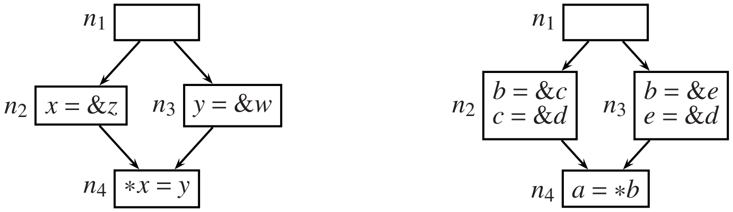
If *may* and *must* analyses are performed independently, then

$$\begin{aligned}\mathit{MayOut}_n &= f_n(\mathit{MayIn}_n, \emptyset)\\ \mathit{MustOut}_n &= f_n(\mathit{MustIn}_n, \mathbb{U})\end{aligned}$$

In other words, in the absence of *must* points-to information, no points-to pair can be killed by indirect effect of an assignment since no strong update is known. In the absence of *may* points-to information, every points-to pair must be assumed to be killed by indirect effect of an assignment since no weak update is known.

Observe that unlike any other flow function, the flow function for points-to analysis given by Equation (4.14) is a binary function rather than a unary function. It has been defined so to capture the inverse dependence of *may* and *must* information through the $\mathit{DepKill}_n$ part. The overall lattice for the data flow Equations (4.11) through (4.13) is a product lattice of the lattices for *may* and *must* points-to relations and the flow function is a unary function for the values in this overall lattice. The $\top$ element of the overall lattice is the pair $\langle \emptyset, \mathbb{U} \rangle$ whereas $\bot$ is $\langle \mathbb{U}, \emptyset \rangle$.

Given a constant *must* points-to information, it is easy to see that the flow functions in *may* points-to analysis are monotonic. This is because the $\mathit{DepKill}_n(\mathsf{x})$ component becomes constant and given a larger x, $\mathit{DepGen}_n(\mathsf{x})$ computes a larger set of points-to pairs. Since *must* points-to analysis has also been defined using the same components, given a constant *may* points-to information, the flow functions of *must* points-to analysis are also monotonic.



(a) Example for *may* points-to analysis (b) Example for *must* points-to analysis

FIGURE 4.12
Non-distributivity of points-to analysis.

Example 4.10
Figure 4.12 shows the non-distributivity of points-to analysis using the flow function associated with node n_4.

Consider the example for *may* points-to analysis. Assuming that the *must* points-to information is constant, non-distributivity of *may* points-to analysis depends on $\mathit{DepGen}_n(\mathsf{x})$. Let x_1 be the *may* points-to information along the edge $n_2 \rightarrow n_4$ and let x_2 be the *may* points-to information along the edge $n_3 \rightarrow n_4$. Then $\mathsf{x}_1 = \{x{\rightarrow}y\}$, $\mathsf{x}_2 = \{y{\rightarrow}w\}$ and:

$$\begin{aligned}
\mathit{DepGen}_n(\mathsf{x}_1 \cup \mathsf{x}_2) &= \{x{\rightarrow}y, y{\rightarrow}w, z{\rightarrow}w\} \\
\mathit{DepGen}_n(\mathsf{x}_1) &= \{x{\rightarrow}y\} \\
\mathit{DepGen}_n(\mathsf{x}_2) &= \{y{\rightarrow}w\} \\
\mathit{DepGen}_n(\mathsf{x}_1 \cup \mathsf{x}_2) &\supset \mathit{DepGen}_n(\mathsf{x}_1) \cup \mathit{DepGen}_n(\mathsf{x}_2)
\end{aligned}$$

Consider the example for *must* points-to analysis under similar situations. In this case $\mathsf{x}_1 = \{b{\rightarrow}c, c{\rightarrow}d\}$, $\mathsf{x}_2 = \{b{\rightarrow}e, e{\rightarrow}d\}$ and:

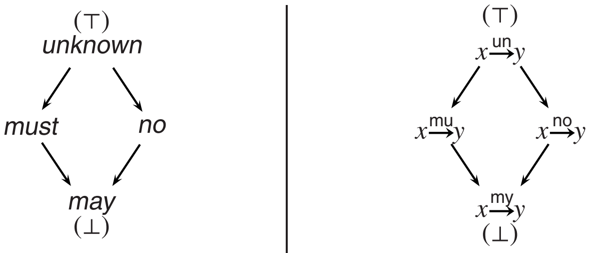
$$\begin{aligned}
\mathit{DepGen}_n(\mathsf{x}_1 \cap \mathsf{x}_2) &= \emptyset \\
\mathit{DepGen}_n(\mathsf{x}_1) &= \{a{\rightarrow}d\} \\
\mathit{DepGen}_n(\mathsf{x}_2) &= \{a{\rightarrow}d\} \\
\mathit{DepGen}_n(\mathsf{x}_1 \cap \mathsf{x}_2) &\subset \mathit{DepGen}_n(\mathsf{x}_1) \cap \mathit{DepGen}_n(\mathsf{x}_2)
\end{aligned}$$

□

We leave it to the reader to construct examples to show that the data flow framework of points-to analysis is not fast.

Points-To Analysis with Degree of Certainty

Instead of computing separate *may* and *must* points-to sets, a points-to pair $x{\rightarrow}y$ can be qualified with degrees of certainties *may* and *must* and can be denoted $x \xrightarrow{\mathsf{my}} y$ and $x \xrightarrow{\mathsf{mu}} y$. This reduces computation of *may* and *must* points-to sets to a single


(⊤)
unknown
must
no
may
(⊥)
(⊤)
$x\xrightarrow{\text{un}}y$
$x\xrightarrow{\text{mu}}y$
$x\xrightarrow{\text{no}}y$
$x\xrightarrow{\text{my}}y$
(⊥)

(a) Degree of certainty

(b) Points-to relation between x and y

FIGURE 4.13
Lattices for points-to analysis with degree of certainty.

analysis unlike *MayIn*/*MayOut* and *MustIn*/*MustOut*. In order to define data flow analysis, we add two more degrees of certainty: $x\xrightarrow{\text{no}}y$ indicates that x does not point to y and $x\xrightarrow{\text{un}}y$ indicates that nothing is known about the points-to relation between x and y.[*] This results in the component lattices shown in Figure 4.13. The confluence operations used in defining the data flow analysis are induced by these lattices and are left implicit in the description of the analysis.

The left and right locations are now qualified with degrees of certainty. However, values *unknown* and *no* are irrelevant in the context of a pointer assignment. The left locations are defined as follows:

lhs_n	$\mathit{ConstLeftL}_n$	$\mathit{DepLeftL}_n(\mathsf{x})$
x	$\{\langle x, \mathit{must}\rangle\}$	$\emptyset$
$*x$	$\emptyset$	$\{\langle y, d\rangle \mid (x\xrightarrow{d}y) \in \mathsf{x}, d \in \{\mathit{may}, \mathit{must}\}\}$

The right locations are defined as follows:

rhs_n	$\mathit{ConstRightL}_n$	$\mathit{DepRightL}_n(\mathsf{x})$
$\&x$	$\{\langle x, \mathit{must}\rangle\}$	$\emptyset$
x	$\emptyset$	$\{\langle y, d\rangle \mid (x\xrightarrow{d}y) \in \mathsf{x}, d \in \{\mathit{may}, \mathit{must}\}\}$
$*x$	$\emptyset$	$\{\langle z, d_1 \sqcap d_2\rangle \mid \{x\xrightarrow{d_1}y, y\xrightarrow{d_2}z\} \subseteq \mathsf{x}, \{d_1, d_2\} \subseteq \{\mathit{may}, \mathit{must}\}\}$

When the left hand side is variable x, all points-to pairs with x as the source are removed. If the right hand side is an address expression, new *must* and *no* points-

[*] *no* and *unknown* need not be represented explicitly. $x\xrightarrow{\text{no}}y$ can be represented by ensuring that $x\xrightarrow{\text{mu}}y$ or $x\xrightarrow{\text{my}}y$ is not present in the set enumerating the points-to relation. For $x\xrightarrow{\text{un}}y$, it is sufficient to record whether the data flow values associated with a node have been computed or not. While combining the data flow information from predecessors, if the values have not been computed for a predecessor m, it can be ignored in the merge operation; this has the effect of assuming that the data flow information associated with m is ⊤. This has been achieved on line 2 of the algorithm presented in Figure 3.15 on page 90 by excluding the predecessors along a back edge during the initialization.

to pairs are generated purely due to local effect regardless of the existing points-to relations. $\mathit{ConstGen}_n$ is defined only in the context of assignments such as $x = \&y$ whereas $\mathit{ConstKill}_n$ is defined only when the left hand side is a variable such as x.

$$\begin{aligned}\mathit{ConstGen}_n &= \{x \xrightarrow{\mathsf{mu}} y \mid \langle x, \mathit{must}\rangle \in \mathit{ConstLeftL}_n, \langle y, \mathit{must}\rangle \in \mathit{ConstRightL}_n\} \cup \\ &\qquad \{x \xrightarrow{\mathsf{no}} z \mid \langle x, \mathit{must}\rangle \in \mathit{ConstLeftL}_n, \mathit{ConstRightL}_n \neq \emptyset, \\ &\qquad\quad \langle z, d\rangle \notin \mathit{ConstRightL}_n\} \\ \mathit{ConstKill}_n &= \{x \xrightarrow{d} y \mid \langle x, \mathit{must}\rangle \in \mathit{ConstLeftL}_n\}\end{aligned}$$

In other situations $\mathit{ConstLeftL}_n \cap \mathit{ConstRightL}_n = \emptyset$. For these situations let

$$\begin{aligned}\mathit{Left}_n(\mathsf{x}) &= \mathit{ConstLeftL}_n \cup \mathit{DepLeftL}_n(\mathsf{x}) \\ \mathit{Right}_n(\mathsf{x}) &= \mathit{ConstRightL}_n \cup \mathit{DepRightL}_n(\mathsf{x})\end{aligned}$$

The dependent information that is generated and killed by a pointer assignment is defined as follows:

$$\begin{aligned}\mathit{DepGen}_n(\mathsf{x}) &= \{x \xrightarrow{d} y \mid \langle x, d_l\rangle \in \mathit{Left}_n(\mathsf{x}), \langle y, d_r\rangle \in \mathit{Right}_n(\mathsf{x}), d = d_l \sqcap d_r\} \cup \\ &\qquad \{x \xrightarrow{\mathsf{my}} y \mid \langle x, \mathit{may}\rangle \in \mathit{Left}_n(\mathsf{x}), x \xrightarrow{\mathsf{mu}} y \in \mathsf{x}\} \cup \\ &\qquad \{x \xrightarrow{\mathsf{no}} y \mid \langle x, \mathit{must}\rangle \in \mathit{Left}_n(\mathsf{x}), \langle y, d\rangle \notin \mathit{Right}_n(\mathsf{x})\}\end{aligned}$$

The first term in the definition of $\mathit{DepGen}_n(\mathsf{x})$ is the result of a combination of the left and right hand sides. The second term lowers the degree of certainty of $x \xrightarrow{\mathsf{mu}} y$ in x to $x \xrightarrow{\mathsf{my}} y$ due to a possible modification of x by the assignment. The third term is a replacement of points-to pairs killed by the assignment.

$$\begin{aligned}\mathit{DepKill}_n(\mathsf{x}) &= \{x \xrightarrow{d} y \mid \langle x, \mathit{must}\rangle \in \mathit{DepLeftL}_n(\mathsf{x})\} \cup \\ &\qquad \{x \xrightarrow{\mathsf{mu}} y \mid \langle x, \mathit{may}\rangle \in \mathit{DepLeftL}_n(\mathsf{x})\}\end{aligned}$$

The first term in $\mathit{DepKill}_n(\mathsf{x})$ represents the guaranteed modification of x by the pointer assignment n. The second term removes $x \xrightarrow{\mathsf{mu}} y$ so that it can be replaced by the generated pair $x \xrightarrow{\mathsf{my}} y$.

The final data flow equations are:

$$\begin{aligned}\mathit{In}_n &= \begin{cases} \mathit{BI} & n \text{ is } \mathit{Start} \\ \displaystyle\bigsqcap_{p \in \mathit{pred}(n)} \mathit{Out}_p & \text{otherwise}\end{cases} \\ \mathit{Out}_n &= f_n(\mathit{In}_n) \;=\; (\mathit{In}_n - \mathit{Kill}_n(\mathit{In}_n)) \cup \mathit{Gen}_n(\mathit{In}_n)\end{aligned}$$

where $\mathit{BI} = \{x \xrightarrow{\mathsf{no}} y \mid x \text{ is a pointer variable and } y \text{ is any variable }\}$.

Example 4.11

We show the computation of points-to pairs qualified with the degree of certainty for the example program in Figure 4.10 on page 122. For simplicity, we omit the pairs representing *no* except for In_n where we list the *BI*. Since we perform round-robin analysis and traverse the graph in reverse postorder, the pairs representing *unknown* are required only in the first iteration and only for Out_{n_6}. We leave them also implicit. Observe that now *may* and *must* are mutually exclusive and the resulting information is more precise.

	Iteration #1	Changes in Iteration #2	Changes in Iteration #3
In_{n_1}	$\{x \xrightarrow{\textsf{no}} y \mid x, y \in \{a, b, c, d\}\}$		
Out_{n_1}	$\{b \xrightarrow{\textsf{mu}} d\}$		
In_{n_2}	$\{b \xrightarrow{\textsf{mu}} d\}$	$\{a \xrightarrow{\textsf{my}} b, a \xrightarrow{\textsf{my}} d, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{my}} d\}$	$\{a \xrightarrow{\textsf{my}} b, a \xrightarrow{\textsf{my}} d, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{my}} b, c \xrightarrow{\textsf{my}} d\}$
Out_{n_2}	$\{b \xrightarrow{\textsf{mu}} d, c \xrightarrow{\textsf{mu}} d\}$	$\{a \xrightarrow{\textsf{my}} b, a \xrightarrow{\textsf{my}} d, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{my}} b, c \xrightarrow{\textsf{my}} d\}$	
In_{n_3}	$\{b \xrightarrow{\textsf{mu}} d, c \xrightarrow{\textsf{mu}} d\}$	$\{a \xrightarrow{\textsf{my}} b, a \xrightarrow{\textsf{my}} d, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{my}} b, c \xrightarrow{\textsf{my}} d\}$	
Out_{n_3}	$\{a \xrightarrow{\textsf{mu}} b, b \xrightarrow{\textsf{mu}} d, c \xrightarrow{\textsf{mu}} d\}$	$\{a \xrightarrow{\textsf{mu}} b, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{my}} b, c \xrightarrow{\textsf{my}} d\}$	
In_{n_4}	$\{a \xrightarrow{\textsf{mu}} b, b \xrightarrow{\textsf{mu}} d, c \xrightarrow{\textsf{mu}} d\}$	$\{a \xrightarrow{\textsf{mu}} b, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{my}} b, c \xrightarrow{\textsf{my}} d\}$	
Out_{n_4}	$\{a \xrightarrow{\textsf{mu}} b, b \xrightarrow{\textsf{mu}} b, c \xrightarrow{\textsf{mu}} d\}$	$\{a \xrightarrow{\textsf{mu}} b, b \xrightarrow{\textsf{mu}} b, c \xrightarrow{\textsf{my}} b, c \xrightarrow{\textsf{my}} d\}$	
In_{n_5}	$\{b \xrightarrow{\textsf{mu}} d, c \xrightarrow{\textsf{mu}} d\}$	$\{a \xrightarrow{\textsf{my}} b, a \xrightarrow{\textsf{my}} d, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{my}} b, c \xrightarrow{\textsf{my}} d\}$	
Out_{n_5}	$\{a \xrightarrow{\textsf{mu}} c, b \xrightarrow{\textsf{mu}} d, c \xrightarrow{\textsf{mu}} d\}$	$\{a \xrightarrow{\textsf{mu}} c, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{my}} b, c \xrightarrow{\textsf{my}} d\}$	
In_{n_6}	$\{a \xrightarrow{\textsf{my}} b, a \xrightarrow{\textsf{my}} c, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{mu}} d\}$	$\{a \xrightarrow{\textsf{my}} b, a \xrightarrow{\textsf{my}} c, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{my}} b, c \xrightarrow{\textsf{my}} d\}$	
Out_{n_6}	$\{a \xrightarrow{\textsf{my}} b, a \xrightarrow{\textsf{my}} d, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{mu}} d\}$	$\{a \xrightarrow{\textsf{my}} b, a \xrightarrow{\textsf{my}} d, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{my}} b, c \xrightarrow{\textsf{my}} d\}$	
In_{n_7}	$\{a \xrightarrow{\textsf{my}} b, a \xrightarrow{\textsf{my}} d, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{mu}} d\}$	$\{a \xrightarrow{\textsf{my}} b, a \xrightarrow{\textsf{my}} d, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{my}} b, c \xrightarrow{\textsf{my}} d\}$	
Out_{n_7}	$\{a \xrightarrow{\textsf{my}} b, a \xrightarrow{\textsf{my}} d, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{mu}} d, d \xrightarrow{\textsf{my}} d\}$	$\{a \xrightarrow{\textsf{my}} b, a \xrightarrow{\textsf{my}} d, b \xrightarrow{\textsf{my}} b, b \xrightarrow{\textsf{my}} d, c \xrightarrow{\textsf{my}} b, c \xrightarrow{\textsf{my}} d, d \xrightarrow{\textsf{my}} b, d \xrightarrow{\textsf{my}} d\}$	

The analysis still requires three iterations. ▯

Example 4.12

We illustrate non-distributivity of points-to analysis with the degree of certainty by enumerating *MOP* and *MFP* assignments for the example in part (a) of Figure 4.12 on page 125.

	MOP Assignment	*MFP* Assignment
Out_{n_2}	$\{x \xrightarrow{\mathsf{mu}} z\}$	$\{x \xrightarrow{\mathsf{mu}} z\}$
Out_{n_3}	$\{y \xrightarrow{\mathsf{mu}} w\}$	$\{y \xrightarrow{\mathsf{mu}} w\}$
Out_{n_4}	$\{x \xrightarrow{\mathsf{my}} z, y \xrightarrow{\mathsf{my}} w\}$	$\{x \xrightarrow{\mathsf{my}} z, y \xrightarrow{\mathsf{my}} w, z \xrightarrow{\mathsf{my}} w\}$

□

4.3.2 Alias Analysis of Stack and Static Data

An alternative way of representing information about pointers is to use the relation of aliasing. Aliasing is defined between pointer expressions which may use dereferencing operations, such as x, $*x$, $**x$ etc.

DEFINITION 4.8 *A pointer expression e_1 is aliased to pointer expression e_2 at program point u, denoted $e_1 \overset{\circ}{=} e_2$, if the expressions e_1 and e_2 evaluate to the same address at u.*

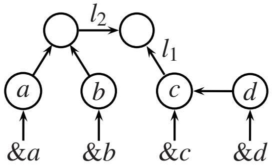
A Comparison of Points-to and Alias Relations

Similar to points-to relation, an alias pair $e_1 \overset{\circ}{=} e_2$ that holds along all paths reaching u is a *must* alias; if it holds along some paths then it is a *may* alias. The lattices of *may* and *must* aliases are similar to the lattices for *may* and *must* points-to relations illustrated in Figure 4.10.

Aliasing is different from points-to relation in the following sense. Although it is possible to create points-to pairs between pointer expressions such as $(*x) \rightarrow (**y)$, the points-to analysis represents the same information by a pair $z \rightarrow w$, where by construction, z and w are both variable names such that z is the target of x and w is the target of $**y$. This is possible since points-to analysis is defined in terms of locations that are compile time constants whereas aliasing is a relation defined in terms of address expressions that can be evaluated only at run time. This information cannot be represented as an alias by using w because an alias does not relate a pointer expression to the address it holds but relates pointer expressions that hold the same address and the target of the two pointer expressions is left implicit. Hence, alias pair $*x \overset{\circ}{=} **y$ needs to be stored.

An alternative way of comparing points-to relations and alias relations is to view them in terms of a memory graph in which edges represent points-to pairs. Alias pairs represent paths that reach the same node in the graph. As a consequence, unlike points-to relation, alias relation is both symmetric and reflexive. *must* aliases are transitive and *may* aliases are not transitive.

$$\begin{aligned} a &= b \\ d &= \&c \\ *a &= c \end{aligned}$$



(a) Sequence of assignments (b) Resulting memory graph

FIGURE 4.14
The need of link aliases in computing node aliases.

The difference that aliasing involves pointer expressions whereas points-to relations involves names of variables implies that in points-to relations, an edge in the memory graph is always represented by a single points-to pair. In the presence of cycles in a data structure, there are an infinite number of paths reaching a node. Thus alias analysis may derive infinite aliases. To see this, consider the assignments sequence $a = \&b$ followed by $b = \&b$ creating a self loop around b. Thus we have an alias $a \doteq *b$. However, since $*b$, $**b$, $***b$ etc. all point to b, we have all possible aliases $a \doteq **b$, $a \doteq ***b$, $b \doteq *b$, $*b \doteq **b$, $**b \doteq *b$, etc. In points-to analysis, the same information is represented by two pairs $a \rightarrow b$ and $b \rightarrow b$.

Due to the presence aliases resulting from pointer indirections, it becomes important to distinguish between *node* and *link* aliases which are defined below.

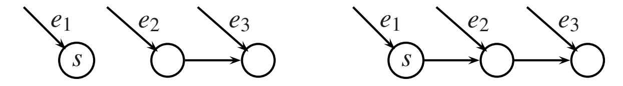
DEFINITION 4.9 *Pointer expressions e_1 and e_2 are* node aliases *if their r-values are same but l-values are different; they are* link aliases *if their l-values are also same. An assignment $a = b$ creates a node alias $a \doteq b$ and link aliases $*a \doteq *b$, $**a \doteq **b$, etc.*

In terms of paths in memory graph, link aliases relate paths that have a non-empty common suffix whereas node aliases relate paths with disjoint non-empty suffixes.

In order to define node aliases for an assignment $a = \&b$, we also introduce a fictitious pointer expression $\&b$ which is assumed to have a unique l-value; its r-value is b. By definition, $*\&b = b$. Assignment $a = \&b$, results in a node alias $a \doteq \&b$. If this is not done, we will have to capture the effect of the assignment by alias pair $*a \doteq b$ which is not a node alias but a link alias.

Link aliases can be computed from node aliases and we restrict our analysis to node aliases only. However, it is necessary to identify link aliases at intermediate stages as explained in the following example. In the rest of this section, we reserve the notation $e_1 \doteq e_2$ to node aliases only; where link aliases are required, they are explicitly defined in terms of node aliases.

Example 4.13
Consider the assignment sequence in Figure 4.14. The assignment $*a = c$ creates the link l_2 in the memory graph thereby creating the node aliases $*a \doteq *d$,



(a) Memory graph before the assignment $s = e_2$

(b) Memory graph after the assignment $s = e_2$

FIGURE 4.15
Direct and indirect node aliases generated as a result of a pointer assignment.

$*b \doteq *d$ and $*b \doteq c$. In order to discover these node aliases, we need to use the fact that $*b$ is a link alias of $*a$ (sharing the link l_2) and $*d$ is a link alias of c (sharing the link l_1). □

In the general situation, given an assignment $\mathit{lhs}_n = \mathit{rhs}_n$ we say that all link aliases of lhs_n get node-aliased to all node and link aliases of rhs_n that are not modified by the assignment. Unlike points-to analysis, alias analysis is significantly influenced by the choice of representation of the alias information. When alias relation is represented in the form of pairs, the node aliases computed by relating appropriate aliases of lhs_n and rhs_n, the resulting aliases are *direct* aliases. However, due to possible indirections of aliases of lhs_n and rhs_n, *indirect* node aliases are also created as explained in the following example.

Example 4.14

Consider the memory graphs in Figure 4.15. As a result of the assignment $s = e_2$, direct aliases $s \doteq e_2$ and $*e_1 \doteq e_2$ are created. However, node aliases $*s \doteq e_3$ and $**e_1 \doteq e_3$ must also be identified. These are examples of indirect node aliases. □

Computing indirect aliases can be avoided by representing alias relations using graphs rather than pairs but the graph representation results in imprecision due to transitivity: When graph representation of two alias pairs $x \doteq y$ and $y \doteq z$ are merged at a join point, their targets are represented by the same node in the graph resulting in a spurious alias $x \doteq z$. This makes the *may* alias information transitive even though the *may* alias relation is not transitive.

Points-to analysis does not have any of the above problems because it is restricted to stack locations and there is a one-to-one mapping between the points-to pairs and the edges in the memory graph. A comparison of points-to relations and alias relations for all possible assignments in our language has been provided in Figure 4.16. It is easy to see that points-to information is much more compact than alias information. On the flip side, using points-to information would require traversing paths in the memory graph; alias information explicates these paths in the pointer expressions used in the alias information.

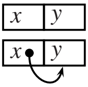
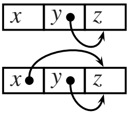
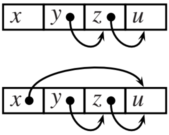
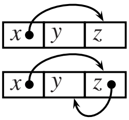
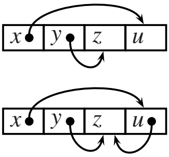
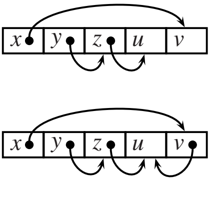
Statement	Memory	Points-to		Aliases	
$x = \&y$	Before After 	Existing		Existing	
		New	$x \to y$	New Direct	$x \doteq \&y$
$x = y$	Before After 	Existing	$y \to z$	Existing	$*y \doteq z$
		New	$x \to z$	New Direct	$x \doteq y$
				New Indirect	$*x \doteq z$
$x = *y$	Before After 	Existing	$y \to z$ $z \to u$	Existing	$*y \doteq z$ $*z \doteq u$ $**y \doteq u$
		New	$x \to u$	New Direct	$x \doteq *y$ $x \doteq z$
				New Indirect	$*x \doteq u$
$*x = \&y$	Before After 	Existing	$x \to z$	Existing	$*x \doteq z$
		New	$z \to y$	New Direct	$*x \doteq \&y$ $*z \doteq y$
$*x = y$	Before After 	Existing	$x \to u$ $y \to z$	Existing	$*x \doteq u$ $*y \doteq z$
		New	$u \to z$	New Direct	$*x \doteq y$ $y \doteq u$
				New Indirect	$*u \doteq z$ $**x \doteq z$
$*x = *y$	Before After 	Existing	$x \to v$ $y \to z$ $z \to u$	Existing	$*x \doteq v$ $*y \doteq z$ $*z \doteq u$ $**y \doteq u$
		New	$v \to u$	New Direct	$*x \doteq *y$ $*x \doteq z$ $v \doteq z$ $v \doteq *y$
				New Indirect	$**x \doteq u$ $*v \doteq u$

FIGURE 4.16
A comparison of points-to and alias relations.

4.3.3 Formulating Data Flow Equations for Alias Analysis

In order to facilitate creation and detection of link aliases, we define a prefix relation on pointer expressions as follows:

$$e_1 \lessdot^k e_2 \Leftrightarrow e_2 \equiv (*)^k e_1$$

where $(*)^k$ denotes k occurrences of the pointer indirection operator $*$. With this notation $x \lessdot^1 *x$, $x \lessdot^2 **x$, and $*x \lessdot^1 **x$. Observe that $\&b \lessdot^1 b$. We also use the & operator with the following semantics:

$$\&e = \begin{cases} \&x & e \text{ is a pointer variable } x \\ e_1 & \text{otherwise, where } e_1 \lessdot^1 e \end{cases}$$

Given a set of node aliases x, we identify all aliases of a pointer expression e as the maximum fixed point of the equation:

$$\mathit{Aliases}(e,\mathsf{x}) = \begin{cases} \{e_1 \mid e_1 \circeq e \in \mathsf{x}\} & e = \&x, x \in \mathbb{V}\mathsf{ar} \\ \{e_1 \mid e_1 \circeq e \in \mathsf{x}\} \cup \{*e_1 \mid e_1 \in \mathit{Aliases}(\&e,\mathsf{x})\} & \text{otherwise} \end{cases}$$

In the presence of cycles in data structures, $\mathit{Aliases}(e,\mathsf{x})$ could be infinite; this would require employing suitable summarization mechanism. We shall see one such mechanism in the context of heap data analysis.

Now we identify the right and left pointer expressions of a pointer assignment for computing alias relations. Consider a pointer assignment $\mathit{lhs}_n = \mathit{rhs}_n$. The definitions of $\mathit{ConstLeftL}_n$ and $\mathit{ConstRightL}_n$ given below are easy to follow. $\mathit{DepLeftL}_n(\mathsf{x})$ represents the set of all link aliases of lhs_n. They are computed from all link and node aliases of $\&\mathit{lhs}_n$. $\mathit{DepRightL}_n(\mathsf{x})$ represents all node and link aliases of rhs_n.

$$\begin{aligned}
\mathit{ConstLeftL}_n &= \{\mathit{lhs}_n\} \\
\mathit{ConstRightL}_n &= \begin{cases} \emptyset & \mathit{lhs}_n \lessdot \mathit{rhs}_n \\ \{\mathit{rhs}_n\} & \text{otherwise} \end{cases} \\
\mathit{DepLeftL}_n(\mathsf{x}) &= \begin{cases} \emptyset & \mathit{lhs}_n \text{ is } \&x, x \in \mathbb{V}\mathsf{ar} \\ \{*e \mid e \in \mathit{Aliases}(\&\mathit{lhs}_n,\mathsf{x})\} & \text{otherwise} \end{cases} \\
\mathit{DepRightL}_n(\mathsf{x}) &= \begin{cases} \emptyset & \mathit{lhs}_n \text{ is } \&x, x \in \mathbb{V}\mathsf{ar} \\ \mathit{Aliases}(\mathit{rhs}_n,\mathsf{x}) & \text{otherwise} \end{cases}
\end{aligned}$$

Observe that we can use only those right pointer expressions that are not modified by the assignment. The pointer expressions that are modified by the assignment are the pointer expressions that have a prefix that is *must* link aliased to lhs_n.

$$\mathit{Mod}_n(\mathsf{x}_1,\mathsf{x}_2) = \{e \mid e_1 \lessdot^i e,\ i \geq 0,\ e_1 \circeq e_2 \in \mathsf{x}_2,\ e_2 \in (\{\mathit{lhs}_n\} \cup \mathit{DepLeftL}_n(\mathsf{x}_1))\}$$

Similar to the inverse dependence of *may* and *must* points-to relations for *Kill*, if x_1 is the set of *may* aliases, then x_2 is the set of *must* aliases and vice-versa.

In the case of points-to analysis, Mod_n is not required because target of the resulting points-to pair is referred to by a variable name rather than through rhs_n. However, in the case of alias analysis, the pointer expression rhs_n is used in the generated aliases and if the resulting pointer expression is link aliased to lhs_n before the assignment, its target changes due to the assignment. Thus it should not participate in the generation of new alias pairs.

Now we define the flow functions for alias analysis. The generated alias pairs are defined by:

$$\mathit{Gen}_n(\mathsf{x}_1,\mathsf{x}_2) = \mathit{ConstGen}_n \cup \mathit{DepGen}_n^D(\mathsf{x}_1,\mathsf{x}_2) \cup \mathit{DepGen}_n^I(\mathsf{x}_1,\mathsf{x}_2)$$

where $\mathit{DepGen}_n^D(\mathsf{x}_1,\mathsf{x}_2)$ represents the direct aliases and $\mathit{DepGen}_n^I(\mathsf{x}_1,\mathsf{x}_2)$ represents indirect aliases and are defined as follows:

$$\begin{aligned}
\mathit{ConstGen}_n &= \{e_1 \circeq e_2 \mid e_1 \in \mathit{ConstLeftL}_n, e_2 \in \mathit{ConstRightL}_n\} \\
\mathit{DepGen}^D_n(\mathsf{x}_1,\mathsf{x}_2) &= \{e_1 \circeq e_2 \mid e_2 \notin \mathit{Mod}_n(\mathsf{x}_1,\mathsf{x}_2), \text{ and} \\
&\qquad\qquad (e_1 \in \mathit{ConstLeftL}_n, e_2 \in \mathit{DepRightL}_n(\mathsf{x}_1)), \text{ or} \\
&\qquad\qquad (e_1 \in \mathit{DepLeftL}_n(\mathsf{x}_1), e_2 \in \mathit{ConstRightL}_n), \text{ or} \\
&\qquad\qquad (e_1 \in \mathit{DepLeftL}_n(\mathsf{x}_1), e_2 \in \mathit{DepRightL}_n(\mathsf{x}_1))\} \\
\mathit{DepGen}^I_n(\mathsf{x}_1,\mathsf{x}_2) &= \{(*)^k e_1 \circeq e_2 \mid e_2 \notin \mathit{Mod}_n(\mathsf{x}_1,\mathsf{x}_2), (*)^k \mathit{rhs}_n \circeq e_2 \in \mathsf{x}_1,\ k > 0, \\
&\qquad\qquad e_1 \in (\mathit{ConstLeftL}_n \cup \mathit{DepLeftL}_n(\mathsf{x}_1))\}
\end{aligned}$$

The aliases killed by the assignment are defined by

$$\mathit{Kill}_n(\mathsf{x}_1,\mathsf{x}_2) = \mathit{ConstKill}_n \cup \mathit{DepKill}_n(\mathsf{x}_1,\mathsf{x}_2)$$

where

$$\begin{aligned}
\mathit{ConstKill}_n &= \{e_1 \circeq e_2 \mid \mathit{lhs}_n \lessdot^k e_1,\ k \geq 0\} \\
\mathit{DepKill}_n(\mathsf{x}_1,\mathsf{x}_2) &= \{e_1 \circeq e_2 \mid e_1 \circeq e_2 \in \mathsf{x}_2,\ e_3 \lessdot^k e_1,\ k \geq 0,\ e_3 \in \mathit{DepLeftL}_n(\mathsf{x}_1)\}
\end{aligned}$$

The top level data flow equations for alias analysis are identical to that of points-to analysis; the flow function f_n is slightly different.

$$\mathit{MayIn}_n = \begin{cases} \mathit{BI} & n \text{ is } \mathit{Start} \\ \displaystyle\bigcup_{p \in \mathit{pred}(n)} \mathit{MayOut}_p & \text{otherwise} \end{cases} \tag{4.15}$$

$$\mathit{MayOut}_n = f_n(\mathit{MayIn}_n, \mathit{MustIn}_n) \tag{4.16}$$

$$\mathit{MustIn}_n = \begin{cases} \mathit{BI} & n \text{ is } \mathit{Start} \\ \displaystyle\bigcap_{p \in \mathit{pred}(n)} \mathit{MustOut}_p & \text{otherwise} \end{cases} \tag{4.17}$$

$$\mathit{MustOut}_n = f_n(\mathit{MustIn}_n, \mathit{MayIn}_n) \tag{4.18}$$

where flow function f_n is defined as follows:

$$f_n(\mathsf{x}_1,\mathsf{x}_2) = (\mathsf{x}_1 - \mathit{Kill}_n(\mathsf{x}_1,\mathsf{x}_2)) \cup \mathit{Gen}_n(\mathsf{x}_1,\mathsf{x}_2) \tag{4.19}$$

In the intraprocedural context, *BI* is $\emptyset$ because no aliases exist at *Start*.

Example 4.15

Recall that the program in Figure 4.10 on page 122 results in a cycle in the data structure because the assignment $*a = a$ in node 4 creates the points-to pair $b \rightarrow b$ in both *may* and *must* points-to analysis. This results in an infinite number of aliases when $\mathit{Aliases}(b, \mathsf{x})$ is computed. Hence we perform *may* alias analysis for a simplified version provided in Figure 4.17 on the facing page. The initialization and *BI* for *may* alias analysis is $\emptyset$. For simplicity, we assume

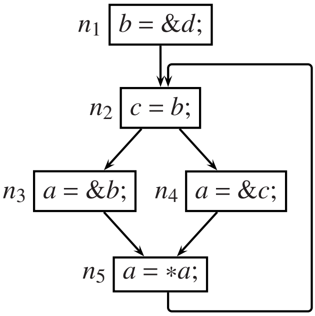


FIGURE 4.17
Example program for alias analysis.

that the *must* alias information is $\emptyset$ at each program point; this causes fewer aliases to be killed and hence is a safe approximation for *may* alias analysis.

Node	Iteration #1		Changes in Iteration #2	
	In_n	Out_n	In_n	Out_n
n_2	$\{b \doteq \&d\}$	$\{b \doteq \&d, c \doteq b, c \doteq \&d\}$	$\{b \doteq \&d, c \doteq \&d, c \doteq b, a \doteq c, a \doteq d\}$	
n_3	$\{b \doteq \&d, c \doteq b, c \doteq \&d\}$	$\{b \doteq \&d, c \doteq b, c \doteq \&d, a \doteq \&b, *a \doteq c, *a \doteq \&d\}$		
n_4	$\{b \doteq \&d, c \doteq b, c \doteq \&d\}$	$\{b \doteq \&d, c \doteq b, c \doteq \&d, a \doteq \&c, *a \doteq c, *a \doteq \&d\}$		
n_5	$\{b \doteq \&d, c \doteq b, c \doteq \&d, a \doteq \&c, a \doteq \&b, *a \doteq c, *a \doteq \&d\}$	$\{b \doteq \&d, c \doteq b, c \doteq \&d, a \doteq c, a \doteq d\}$		

Observe that the pairs $*a \doteq c$ and $*a \doteq \&d$ in Out_{n_3} and Out_{n_4} are indirect aliases. All other aliases are direct aliases. □

Similar to points-to analysis, alias analysis is neither fast nor distributive. Example 3.10 in Chapter 3 (Figure 3.10) showed the non-distributivity of *may* alias analysis. We leave it for the reader to construct examples to demonstrate the non-distributivity of *must* alias analysis and non-fastness of *may* and *must* alias analysis.

4.4 Liveness Analysis of Heap Data

The data flow analyses described earlier referred to data objects resident in the stack or the static area. In this section, we describe an analysis for data objects residing

```
   x->succ = y->rptr->lptr
   z->lptr = y->rptr
    ...
p: ...
    ...
   if (u == z->lptr->lptr)
   ...
```

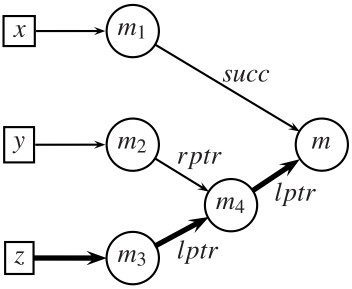


FIGURE 4.18
An example to motivate liveness analysis. The path consisting of thick edges is explicitly live at p.

on the heap. An optimization that requires this analysis was described in Chapter 1. The key idea was to identify heap objects that would not be used in the future, even if they were reachable. Such objects can be freed and the memory space occupied by them can be reused. This optimization brings down the overall memory requirement of the program. If the run time support of the language includes a garbage collector, then the garbage collector can be expected to collect more garbage per collection. Further, if the collector is a copying collector, then the collection itself will be faster since copying collectors process live data only.

To identify the nature of the analysis required for this purpose, consider the example shown in Figure 4.18. The declared variables x, y and z are local or global pointers and accordingly reside in the stack or the static area. We call these *root variables*. The objects pointed to by these variables are on the heap. In this analysis we ignore non-pointer variables. Though our language resembles C, we assume that the programs being analyzed do not make use of the & (address of) operator. Thus root variables cannot point to other root variables. We view the heap at a program point as a directed graph called *memory graph*. The root variables form the entry nodes of the memory graph. Other nodes in the graph correspond to objects on the heap and edges correspond to pointers. The out-edges of entry nodes are labeled by root variable names while out-edges of other nodes are labeled by field names. The edges in the memory graph are called *links*.

Example 4.16
Figure 4.18 shows the memory graph at the program point p. If we can discover that the links $m_4 \rightarrow m$ and $m_1 \rightarrow m$ are never used in any execution path starting from p, then we can free the object m at p by inserting the statements `z->lptr->lptr=NULL` and `x->succ = NULL`. Here, by usage of a link we mean either dereferencing it to access an object or testing it for comparison. In the example shown, the statement `z->lptr->lptr=NULL` cannot be inserted because the link $m_4 \rightarrow m$ is subsequently used by the condition

`if (u == z->lptr->rptr)` for comparison. Thus the object m cannot be freed at p. □

In this section, we consider the analysis that discovers whether a link is live i.e., whether it will be used in the sense described above.

4.4.1 Access Expressions and Access Paths

A program accesses data through expressions which have l-values. Such expressions are called *access expressions*. They can be scalar variables such as x, or may involve an array access such as $a[2*i]$, or can be reference expressions such as $*x$ or $y \rightarrow rptr \rightarrow lptr$. Since we are concerned with analysis of heap-resident data, from now on we shall limit our attention to reference expressions. These are the expressions that are primarily used to access the heap. In Figure 4.18, the access expression $y \rightarrow rptr \rightarrow lptr$ refers to the heap data denoted as m.

In order to discover liveness and other properties of heap, we need a way of naming links in the memory graph. We do this using access paths. An *access path* ρ_x is a root variable name followed by a sequence of zero or more field names and is denoted by $x \rightarrowtriangle f_1 \rightarrowtriangle f_2 \rightarrowtriangle \cdots \rightarrowtriangle f_k$. Since an access path represents a path in a memory graph, it can be used for naming links and nodes. An access path consisting of just a root variable name is called a *simple* access path; it represents a path from a root variable to the object pointed to by it. In the context of C, one could think of this as the path followed to access an object using an access expression such as $*x$. $\mathcal{E}$ denotes an empty access path.

The last field name in an access path ρ is called its *frontier* and is denoted by $\mathsf{frontier}(\rho)$. The frontier of a simple access path is the root variable name. The access path corresponding to the sequence of names in ρ excluding only its frontier is called its *base* and is denoted by $\mathsf{base}(\rho)$. The base of a simple access path is the empty access path $\mathcal{E}$. The object reached by traversing an access path ρ is called the *target* of the access path and is denoted by $\mathsf{target}(\rho)$. When we use an access path ρ to refer to a link in a memory graph, it denotes the last link in ρ i.e., the link corresponding to $\mathsf{frontier}(\rho)$.

Example 4.17

Consider the access path $\rho_z = z \rightarrowtriangle lptr \rightarrowtriangle lptr$ at program point p. $\mathsf{target}(\rho_z)$ denotes the node m and $\mathsf{frontier}(\rho_z)$ denotes the link $m_4 \rightarrow m$. As we have said earlier, access paths are also used to denote links in memory graph. The link denoted by ρ_z is also $m_4 \rightarrow m$. $\mathsf{base}(\rho_z)$ is the access path $z \rightarrowtriangle m_3 \rightarrowtriangle m_4$. □

In the rest of the section, α will denote an access expression, ρ will denote an access path and σ will denote a (possibly empty) sequence of field names separated by $\rightarrowtriangle$. Let the access expression α_x be $x \rightarrow f_1 \rightarrow f_2 \ldots \rightarrow f_n$. Then, the corresponding access path ρ_x is $x \rightarrowtriangle f_1 \rightarrowtriangle f_2 \ldots \rightarrowtriangle f_n$. When the root variable name is not required, we drop the subscripts from α_x and ρ_x.

We assume that our method does a context insensitive interprocedural analysis. To simplify the description of analysis we assume that the conditions that alter flow of control are made up only of simple variables. If not, the offending reference expression is assigned to a fresh simple variable before the condition and is replaced by the fresh variable in the condition.

The statements that we handle fall in one of the following categories:

- *Function Calls*. These are statements $x = f(\alpha_y, \alpha_z, \ldots)$ where the functions involve access expressions in arguments. The variable x can be a reference or a non-reference variable.

- *Assignment Statements*. These are assignments to references and are denoted by $\alpha_x = \alpha_y$. Only these statements can modify the structure of the heap.

- *Use Statements*. These statements use heap references to access heap data but do not modify heap references. For the purpose of analysis, these statements are abstracted as lists of expressions $\alpha_y.d$ where α_y is an access expression and d is a non-reference.

- *Return Statement* of the type *return* α_x involving reference variable x.

- *Other Statements*. These statements include all statements which do not refer to the heap. We ignore these statements since they do not influence heap reference analysis.

As is customary in static analysis, when we talk about execution paths, we shall refer to a trace of the program that ignores the evaluation of condition checks. For simplicity of exposition, we present the analyses assuming that the program to be analyzed does not create cycles in the heap during execution.

4.4.2 Liveness of Access Paths

A link l is *live* at a program point p if it is used in some control flow path starting from p. As noted earlier, l may be used in two different ways; it may be dereferenced to access an object or tested for comparison. Figure 4.18(b) shows links that are live before program point p by thick arrows. For a link l to be live, there must be at least one access path from some root variable to l such that every link in this path is live. This is the path that is actually traversed while using l.

Since the freeing of nodes is through access paths, we need to express the notion of liveness of links in terms of access paths. An access path is defined to be *live* at p if the link corresponding to its frontier is live along some path starting at p.

We limit ourselves to a subset of live access paths, whose liveness can be determined without taking into account the aliases created before p. These access paths are live solely because of the execution of the program beyond p. We call access paths which are live in this sense as *explicitly live* access paths. An interesting property of explicitly live access paths is that they form the minimal set covering every live link. In this section, we further restrict ourselves to the computation of explicit liveness.

Example 4.18

The access paths z, $z{\to}lptr$, $z{\to}lptr{\to}lptr$ and $y{\to}rptr{\to}lptr$ are all live at p. All these paths except $y{\to}rptr{\to}lptr$ are also explicitly live. The access path $y{\to}rptr{\to}lptr$ is live because of the alias created before p. Also note that if an access path is explicitly live, so are all its prefixes. □

Example 4.19

We illustrate the issues in determining explicit liveness of access paths by considering the assignment $x.r.n = y.n.n$.

- *Killed Access Paths*. Since the assignment modifies $\mathsf{frontier}(x{\to}r{\to}n)$, any access path which is live after the assignment and has $x{\to}r{\to}n$ as prefix will cease to be live before the assignment. Access paths that are live after the assignment and not killed by it are live before the assignment also.
- *Directly Generated Access Paths*. All prefixes of $x{\to}r$ and $y{\to}n$ are explicitly live before the assignment due to the local effect of the assignment.
- *Transferred Access Paths*. If $x{\to}r{\to}n{\to}\sigma$ is live after the assignment, then $y{\to}n{\to}n{\to}\sigma$ will be live before the assignment. For example, if $x{\to}r{\to}n{\to}n$ is live after the assignment, then $y{\to}n{\to}n{\to}n$ will be live before the assignment. The sequence of field names σ is viewed as being *transferred* from $x{\to}r{\to}n$ to $y{\to}n{\to}n$.

□

We now define liveness by generalizing the above observations. We use the notation $\rho_x{\to}*$ to enumerate all access paths which have ρ_x as a prefix. The summary liveness information for a set S of access paths is defined as follows:

$$\mathsf{summary}(S) = \bigcup_{\rho \in S} \{\rho{\to}*\}$$

Further, the set of all global variables is denoted by *Globals* and the set of formal parameters of the function being analyzed is denoted by *Params*.

DEFINITION 4.10 *The set of explicitly live access paths at a program point p, denoted by $\mathsf{liveness}_p$ is defined as follows.*

$$\mathsf{liveness}_p = \bigcup_{\psi \in paths(p)} (\mathsf{pathLiveness}_p^{\psi})$$

```
0.    w = x
1.    while (x->data < max)
2.    {
3.       x = x->rptr
4.    }
5.    y = x->lptr
6.    z = malloc(...)
7.    y = y->lptr
8.    z->sum = x->lptr->data
            + y->data
```

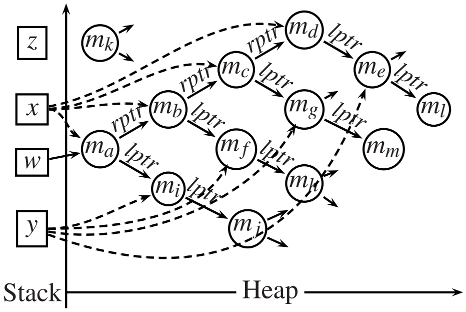


FIGURE 4.19

An example program and possible memory graphs before line 6. Depending on whether the *while* loop is iterated 0, 1, 2, or 3 times, x will point to m_a, m_b, m_c, or m_d. Accordingly y will point to m_i, m_f, m_g, or m_e.

where, $\psi \in paths(p)$ is a control flow path p to Start *and* $\mathsf{pathLiveness}_p^\psi$ *denotes the liveness at p along ψ and is defined as follows. If p is not program exit then let the statement which follows it be denoted by s and the program point immediately following s be denoted by p'. Then,*

$$\mathsf{pathLiveness}_p^\psi = \begin{cases} \emptyset & p = \mathsf{Exit}(main) \\ \mathsf{summary}(\mathsf{Globals}) & p = \mathsf{Exit}(f),\ f \neq main \\ \mathsf{statementLiveness}_s(\mathsf{pathLiveness}_{p'}^\psi) & otherwise \end{cases}$$

where the flow function for s is defined as follows:

$$\mathsf{statementLiveness}_s(X) = (X - \mathsf{LKill}_s) \cup \mathsf{LDirect}_s \cup \mathsf{LTransfer}_s(X)$$

LKill_s *denotes the sets of access paths which cease to be live before statement s,* $\mathsf{LDirect}_s$ *denotes the set of access paths which become live due to local effect of s and* $\mathsf{LTransfer}_s(X)$ *denotes the set of access paths which become live before s due to transfer of liveness from live access paths after s. They are defined in Figure 4.20.*

Observe that the definitions of LKill_s, $\mathsf{LDirect}_s$, and $\mathsf{LTransfer}_s$ ensure that the $\mathsf{liveness}_p$ is prefix-closed.

When we view the above definition in terms of the constant and dependent parts of flow functions as defined in Section 4.1, it is clear that LKill_s represents $\mathsf{DepKill}_s$ and $\mathsf{ConstKill}_s$ is $\emptyset$. Liveness information is generated by $\mathsf{LDirect}_s$ which represents $\mathsf{ConstGen}_s$ and $\mathsf{LTransfer}_s$ which represents DepGen_s.

Example 4.20

In Figure 4.19, it cannot be statically determined which link is represented by

Statement s	LKill_s	$\mathit{LDirect}_s$	$\mathit{LTransfer}_s(X)$
$\alpha_x = \alpha_y$	$\{\rho_x \rightarrowtriangle *\}$	$\mathit{prefixes}(\mathit{base}(\rho_x)) \cup \mathit{prefixes}(\mathit{base}(\rho_y))$	$\{\rho_y \rightarrowtriangle \sigma \mid \rho_x \rightarrowtriangle \sigma \in X\}$
$\alpha_x = f(\alpha_y)$	$\{\rho_x \rightarrowtriangle *\}$	$\mathit{prefixes}(\mathit{base}(\rho_x)) \cup \mathit{prefixes}(\mathit{base}(\rho_y)) \cup \mathit{summary}(\{\rho_y\} \cup \mathit{Globals})$	$\emptyset$
$\alpha_x =$ *new*	$\{\rho_x \rightarrowtriangle *\}$	$\mathit{prefixes}(\mathit{base}(\rho_x))$	$\emptyset$
$\alpha_x =$ *null*	$\{\rho_x \rightarrowtriangle *\}$	$\mathit{prefixes}(\mathit{base}(\rho_x))$	$\emptyset$
Use $\alpha_y.d$	$\emptyset$	$\mathit{prefixes}(\rho_y)$	$\emptyset$
return α_y	$\emptyset$	$\mathit{prefixes}(\mathit{base}(\rho_y)) \cup \mathit{summary}(\{\rho_y\})$	$\emptyset$
other	$\emptyset$	$\emptyset$	$\emptyset$

FIGURE 4.20
Defining flow functions for liveness. *Globals* denotes the set of global references and *Params* denotes the set of formal parameters. For simplicity, we have shown a single access expression on the RHS.

access expression *x.lptr* at line 5. Depending upon the number of iterations of the while loop, it may be any of the links represented by thick arrows. Thus at line 0, we have to assume that all access paths $\{x \rightarrowtriangle lptr \rightarrowtriangle lptr$, $x \rightarrowtriangle rptr \rightarrowtriangle lptr \rightarrowtriangle lptr$, $x \rightarrowtriangle rptr \rightarrowtriangle rptr \rightarrowtriangle lptr \rightarrowtriangle lptr, \ldots\}$ are explicitly live. □

4.4.3 Representing Sets of Access Paths by Access Graphs

In the presence of loops, the set of access paths may be infinite and the lengths of access paths may be unbounded. If the algorithm for analysis tries to compute sets of access paths explicitly, termination cannot be guaranteed. We solve this problem by representing a set of access paths by a graph of bounded size. The structure that we use for the representation is called an access graph.

An *access graph*, denoted by G_v, is a directed graph $\langle n_0, N, E\rangle$ representing a set of access paths starting from a root variable v.[†] N is the set of nodes, $n_0 \in N_F$ is the entry node with no in-edges and E is the set of edges. Every path in the graph represents an access path. The *empty graph* $\mathcal{E}_G$ has no nodes or edges and does not accept any access path.

The entry node of an access graph is labeled with the name of the root variable while the non-entry nodes are labeled with a unique label created as follows: If a field name f is referenced in basic block b, we create an access graph node with a label $\langle f, b, i\rangle$ where i is the instance number used for distinguishing multiple occurrences of the field name f in block b. Note that this implies that the nodes with the same

[†] Where the root variable name is not required, we drop the subscript v from G_v.

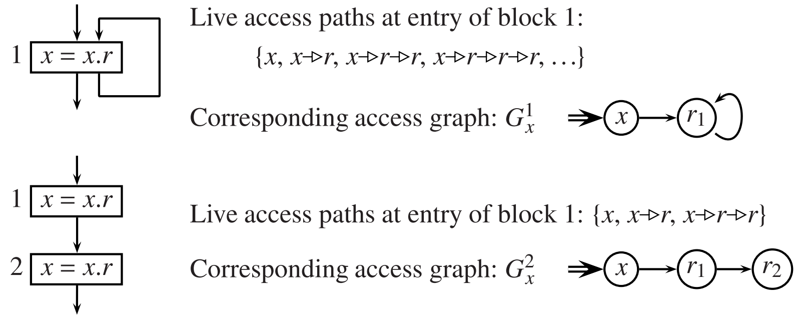

1 $x = x.r$
Live access paths at entry of block 1:
$\{x, x \rightarrowtriangle r, x \rightarrowtriangle r \rightarrowtriangle r, x \rightarrowtriangle r \rightarrowtriangle r \rightarrowtriangle r, \ldots\}$
Corresponding access graph: G_x^1
x
r_1
1 $x = x.r$
2 $x = x.r$
Live access paths at entry of block 1: $\{x, x \rightarrowtriangle r, x \rightarrowtriangle r \rightarrowtriangle r\}$
Corresponding access graph: G_x^2
x
r_1
r_2

FIGURE 4.21
Approximations in access graphs.

label are treated as being identical. Often, i is 0 and in such a case we denote the label $\langle f, b, 0\rangle$ by f_b for brevity.

A node in the access graph represents one or more links in the memory graph. Additionally, during analysis, it represents a state of access graph construction (explained in Section 4.4.3). An edge $f_n \rightarrow g_m$ in an access graph at program point p indicates that a link corresponding to field f dereferenced in block n may be used to dereference a link corresponding to field g in block m on some path starting at p. This has been used in Section 4.4.4 to argue that the size of access graphs in practical programs is small.

Pictorially, the entry node of an access graph is indicated by an incoming double arrow.

Summarization

Recall that a link is live at a program point p if it is used along some control flow path from p to *Start*. Since different access paths may be live along different control flow paths and there may be infinitely many control flow paths in the case of a loop following p, there may be infinitely many access paths which are live at p. Hence, the lengths of access paths will be unbounded. In such a case summarization is required.

Summarization is achieved by merging appropriate nodes in access graphs, retaining all in and out edges of merged nodes. We explain merging with the help of Figure 4.21:

- Node r_1 in access graph G_x^1 indicates references of n at *different execution instances of the same* program point. Every time this program point is visited during analysis, the same state is reached in that the pattern of references after r_1 is repeated. Thus all occurrences of r_1 are merged into a single state. This creates a cycle which captures the repeating pattern of references.

- In G_x^2, nodes r_1 and r_2 indicate referencing n at *different* program points. Since the references made after these program points may be different, r_1 and r_2 are not merged.

Summarization captures the pattern of heap traversal in the most straightforward way. Traversing a path in the heap requires the presence of reference assignments $\alpha_x = \alpha_y$ such that ρ_x is a proper prefix of ρ_y. Assignments in Figure 4.21 are examples of such assignments. The structure of the flow of control between such assignments in a program determines the pattern of heap traversal. Summarization captures this pattern without the need of control flow analysis and the resulting structure is reflected in the access graphs as can be seen in Figure 4.21. More examples of the resemblance of program structure and access graph structure can be seen in the access graphs in Figure 4.24.

Operations on Access Graphs

Section 4.4.2 defined liveness by applying certain operations on access paths. In this subsection we define the corresponding operations on access graphs. Unless specified otherwise, the binary operations are applied only to access graphs having same root variable. The auxiliary operations and associated notations are:

- $\mathsf{root}(\rho)$ denotes the root variable of access path ρ, while $\mathsf{root}(G)$ denotes the root variable of access graph G.

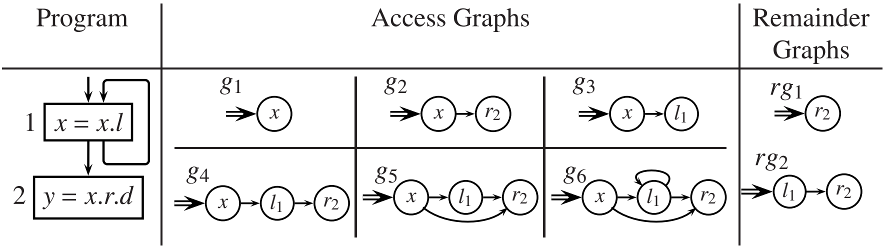
- $\mathsf{field}(n)$ for a node n denotes the field name component of the label of n.

- $\mathsf{makeGraph}(\rho)$ constructs access graphs corresponding to ρ. It uses the current basic block number and the field names to create appropriate labels for nodes. The instance number depends on the number of occurrences of a field name in the block. $\mathsf{makeGraph}(\rho \rightarrowtriangle *)$ creates an access graph for ρ and connects the final node of the access graph to a special node $n_\star$ called summary node. In addition, there is a self loop over $n_\star$. Both the new edges are assumed to have all field names as labels.

- $\mathsf{lastNode}(G)$ returns the last node of a *linear graph* G constructed from a given access path ρ.

- $\mathsf{cleanUp}(G)$ deletes the nodes which are not reachable from the entry node.

- $\mathsf{CN}(G, G', S)$ computes the set of nodes of G which correspond to the nodes of G' specified in the set S. To compute $\mathsf{CN}(G, G', S)$, we define $\mathsf{ACN}(G, G')$, the set of pairs of *all corresponding nodes*. Let $G \equiv \langle n_0, N, E\rangle$ and $G' \equiv \langle n'_0, N', E'\rangle$. A node n in access graph G corresponds to a node n' in access graph G' if there exists an access path ρ which is represented by a path from n_0 to n in G and a path from n'_0 to n' in G'.



Union	Path Removal	Factorization	Extension
$g_3 \uplus g_4 = g_4$	$g_6 \ominus x \rightarrowtriangle l = g_2$	$g_2/(g_1,\{x\}) = \{rg_1\}$	$(g_3,\{l_1\})\#\{rg_1\} = g_4$
$g_2 \uplus g_4 = g_5$	$g_5 \ominus x = \mathcal{E}_G$	$g_5/(g_1,\{x\}) = \{rg_1, rg_2\}$	$(g_3,\{x,l_1\})\#\{rg_1,rg_2\} = g_6$
$g_5 \uplus g_4 = g_5$	$g_4 \ominus x \rightarrowtriangle r = g_4$	$g_5/(g_2,\{r_2\}) = \{\epsilon_{RG}\}$	$(g_2,\{r_2\})\#\{\epsilon_{RG}\} = g_2$
$g_5 \uplus g_6 = g_6$	$g_4 \ominus x \rightarrowtriangle l = g_1$	$g_4/(g_2,\{r_2\}) = \emptyset$	$(g_2,\{r_2\})\#\emptyset = \mathcal{E}_G$

FIGURE 4.22
Examples of operations on access graphs.

Formally, $\mathit{ACN}(G,G')$ is the least solution of the following equation:

$$\mathit{ACN}(G,G') = \begin{cases} \emptyset & \mathit{root}(G) \neq \mathit{root}(G') \\ \{\langle n_0, n'_0\rangle\} \cup \{\langle n_j, n'_j\rangle \mid & \text{otherwise} \\ \quad \mathit{field}(n_j) = \mathit{field}(n'_j), & \\ \quad n_i \rightarrow n_j \in E, n'_i \rightarrow n'_j \in E', & \\ \quad \langle n_i, n'_i\rangle \in \mathit{ACN}(G,G')\} & \end{cases}$$

$$\mathit{CN}(G,G',S) = \{n \mid \langle n,n'\rangle \in \mathit{ACN}(G,G'),\, n' \in S\}$$

Note that $\mathit{field}(n_j) = \mathit{field}(n'_j)$ would hold even when n_j or n'_j is the summary node $n_\star$.

Let $G \equiv \langle n_0, N, E\rangle$ and $G' \equiv \langle n_0, N', E'\rangle$ be access graphs (having the same entry node). G and G' are equal if $N = N'$ and $E = E'$.

The main operations of interest are defined below and are illustrated in Figure 4.22.

1. *Union* ($\uplus$). $G \uplus G'$ combines access graphs G and G' such that any access path contained in G or G' is contained in the resulting graph.

   $$G \uplus G' = \langle n_0, N \cup N', E \cup E'\rangle$$

   The operation $N \cup N'$ treats the nodes with the same label as identical. Because of associativity, $\uplus$ can be generalized to arbitrary number of arguments in an obvious manner.

2. *Path Removal* ($\ominus$). The operation $G \ominus \rho$ removes those access paths in G which have ρ as a prefix.

$$G \ominus \rho = \begin{cases} G & \rho = \mathcal{E} \text{ or } \mathsf{root}(\rho) \neq \mathsf{root}(G) \\ \mathcal{E}_G & \rho \text{ is a simple access path} \\ \mathsf{cleanUp}(\langle n_0, N, E - E_{del} \rangle) & \mathit{otherwise} \end{cases}$$

where

$$\begin{aligned} E_{del} = \{ n_i \rightarrow n_j \mid\ & n_i \rightarrow n_j \in E, n_i \in \mathsf{CN}(G, G_B, \{\mathsf{lastNode}(G_B)\}), \\ & \mathsf{field}(n_j) = \mathsf{frontier}(\rho), G_B = \mathsf{makeGraph}(\mathsf{base}(\rho)), \\ & \mathsf{uniqueAccessPath?}(G, n_i) \} \end{aligned}$$

$\mathsf{uniqueAccessPath?}(G, n)$ returns true if in G, all paths from the entry node to node n represent the same access path. Note that path removal is conservative in that some paths having ρ as prefix may not be removed. Since an access graph edge may be contained in more than one access path, we have to ensure that access paths which do not have ρ as prefix are not erroneously deleted.

3. *Factorization* (/). Recall that the *LTransfer* term in Definition 4.10 requires extracting suffixes of access paths and attaching them to some other access paths. The corresponding operations on access graphs are performed using factorization and extension. Given a node $m \in (N - \{n_0\})$ of an access graph G, the *Remainder Graph* of G at m is the subgraph of G rooted at m and is denoted by $\mathsf{RG}(G, m)$. If m does not have any outgoing edges, then the result is the empty remainder graph ϵ_{RG}. Let M be a subset of the nodes of G' and M' be the set of corresponding nodes in G. Then, $G/(G', M)$ computes the set of remainder graphs of the successors of nodes in M'.

$$G/(G', M) = \{ \mathsf{RG}(G, n_j) \mid n_i \rightarrow n_j \in E, n_i \in \mathsf{CN}(G, G', M) \} \qquad (4.20)$$

A remainder graph is similar to an access graph except that (a) its entry node does not correspond to a root variable but to a field name and (b) the entry node can have incoming edges.

4. *Extension*. Extending an empty access graph $\mathcal{E}_G$ results in the empty access graph $\mathcal{E}_G$. For non-empty graphs, this operation is defined as follows.

   (a) *Extension with a remainder graph* ($\cdot$). Let M be a subset of the nodes of G and $R \equiv \langle n', N^R, E^R \rangle$ be a remainder graph. Then, $(G, M) \cdot R$ appends the suffixes in R to the access paths ending on nodes in M.

$$\begin{aligned} (G, M) \cdot \epsilon_{\mathsf{RG}} &= G \\ (G, M) \cdot R &= \left\langle n_0, N \cup N^R, E \cup E^R \cup \{ n_i \rightarrow n' \mid n_i \in M \} \right\rangle \end{aligned} \qquad (4.21)$$

Operation	Access Graphs	Access Paths
Union	$G_3 = G_1 \uplus G_2$	$\mathsf{P}(G_3, M_3) \supseteq \mathsf{P}(G_1, M_1) \cup \mathsf{P}(G_2, M_2)$
Path Removal	$G_2 = G_1 \ominus \rho$	$\mathsf{P}(G_2, M_2) \supseteq \mathsf{P}(G_1, M_1) - \{\rho \rightarrowtriangle \sigma \mid \rho \rightarrowtriangle \sigma \in \mathsf{P}(G_1, M_1)\}$
Factorization	$S = G_1/(G_2, M)$	$\mathsf{P}(S, M_s) = \{\sigma \mid \rho' \rightarrowtriangle \sigma \in \mathsf{P}(G_1, M_1),\ \rho' \in \mathsf{P}(G_2, M)\}$
Extension	$G_2 = (G_1, M) \# S$	$\mathsf{P}(G_2, M_2) \supseteq \mathsf{P}(G_1, M_1) \cup \{\rho \rightarrowtriangle \sigma \mid \rho \in \mathsf{P}(G_1, M),\ \sigma \in \mathsf{P}(S, M_s)\}$

FIGURE 4.23
Safety of access graph operations. $\mathsf{P}(G, M)$ is the set of paths in graph G terminating on nodes in M. For graph G_i, M_i is the set of all nodes in G_i. S is the set of remainder graphs and $\mathsf{P}(S, M_s)$ is the set of all paths in all remainder graphs in S.

(b) *Extension with a set of remainder graphs* (#). Let S be a set of remainder graphs. Then, $G\#S$ extends access graph G with every graph in S.

$$\begin{aligned} (G, M)\#\emptyset &= \mathcal{E}_G \\ (G, M)\#S &= \biguplus_{R \in S} (G, M) \cdot R \end{aligned} \tag{4.22}$$

Safety of Access Graph Operations

Since access graphs are not exact representations of sets of access paths, the safety of approximations needs to be defined explicitly. The constraints defined in Figure 4.23 capture safety in the context of liveness in the following sense: Every access path which can possibly be live should be retained by each operation. Since the complement of liveness is used to free heap data by nullifying links, this ensures that no live access path is considered for nullification.

4.4.4 Data Flow Analysis for Explicit Liveness

For a given root variable v, $\mathsf{ELIn}_v(i)$ and $\mathsf{ELOut}_v(i)$ denote the access graphs representing explicitly live access paths at the entry and exit of basic block i. We use $\mathcal{E}_G$ as the initial value for $\mathsf{ELIn}_v(i)/\mathsf{ELOut}_v(i)$.

$$\mathsf{ELIn}_v(i) = (\mathsf{ELOut}_v(i) \ominus \mathsf{ELKillPath}_v(i)) \uplus \mathsf{ELGen}_v(i) \tag{4.23}$$

$$\mathsf{ELOut}_v(i) = \begin{cases} \mathsf{makeGraph}(v \rightarrowtriangle *) & i = \mathsf{Start},\ v \in \mathsf{Globals} \\ \mathcal{E}_G & i = \mathsf{Start},\ v \notin \mathsf{Globals} \\ \biguplus_{s \in succ(i)} \mathsf{ELIn}_v(s) & \text{otherwise} \end{cases} \tag{4.24}$$

where

$$\mathit{ELGen}_v(i) = \mathit{LDirect}_v(i) \uplus \mathit{LTransfer}_v(i)$$

The term $\mathit{LDirect}_v(i)$ represents the $\mathit{ConstGen}_i$ component for variable v whereas $\mathit{LTransfer}_v(i)$ represents the DepGen_i component for v. Liveness information is killed using path removal which is implemented by deleting an edge in an access graph. In our case, this edge is $\mathit{frontier}(\rho_x)$ where ρ_x denotes the access path representing the access expression appearing on the left hand side of an assignment. Hence $\mathit{ELKillPath}_v(i)$ represents $\mathit{ConstGen}_i$. This is unlike LKill_s (Definition 4.10 on page 139) which represents $\mathit{DepKill}_s$ rather than $\mathit{ConstKill}_s$. This is because LKill_s is not a fixed set but depends on the liveness information that holds after statement s.

The definitions of $\mathit{ELKillPath}_v(i)$, $\mathit{LDirect}_v(i)$, and $\mathit{LTransfer}_v(i)$ depend on statement i as follows:

1. *Assignment statement* $\alpha_x = \alpha_y$. Apart from defining the desired terms for x and y, we also need to define them for any other variable z. In the following equations, G_x and G_y denote $\mathit{makeGraph}(\rho_x)$ and $\mathit{makeGraph}(\rho_y)$ respectively, whereas M_x denotes $\mathit{lastNode}(\mathit{makeGraph}(\rho_x))$ and M_y denotes $\mathit{lastNode}(\mathit{makeGraph}(\rho_y))$.

$$\begin{aligned}
\mathit{LDirect}_x(i) &= \mathit{makeGraph}(\mathit{base}(\rho_x))\\
\mathit{LDirect}_y(i) &= \begin{cases} \mathcal{E}_G & \alpha_y \text{ is } \mathit{New} \ldots \text{ or } \mathit{null}\\ \mathit{makeGraph}(\mathit{base}(\rho_y)) & \text{otherwise}\end{cases}\\
\mathit{LDirect}_z(i) &= \mathcal{E}_G, \text{for any variable } z \text{ other than } x \text{ and } y\\
\mathit{LTransfer}_y(i) &= \begin{cases} \mathcal{E}_G & \alpha_y \text{ is } \mathit{New} \text{ or } \mathit{null}\\ (G_y, M_y)\# & \text{otherwise}\\ \quad (\mathit{ELOut}_x(i)/(G_x, M_x)) & \end{cases} \qquad (4.25)\\
\mathit{LTransfer}_z(i) &= \mathcal{E}_G, \text{ for any variable } z \text{ other than } y\\
\mathit{ELKillPath}_x(i) &= \rho_x\\
\mathit{ELKillPath}_z(i) &= \mathcal{E}, \text{ for any variable } z \text{ other than } x
\end{aligned}$$

   As stated earlier, the path removal operation deletes an edge only if it is contained in a unique path. Thus fewer paths may be killed than desired. This is a safe approximation. Another approximation which is also safe is that only the paths rooted at x are killed. Since assignment to α_x changes the link represented by $\mathit{frontier}(\rho_x)$, for precision, any path which is guaranteed to contain the link represented by $\mathit{frontier}(\rho_x)$ should also be killed. Such paths can be discovered through *must*-alias analysis.

2. *Function call* $\alpha_x = f(\alpha_y)$. We conservatively assume that a function call may make any access path rooted at y or any global reference variable live. Thus

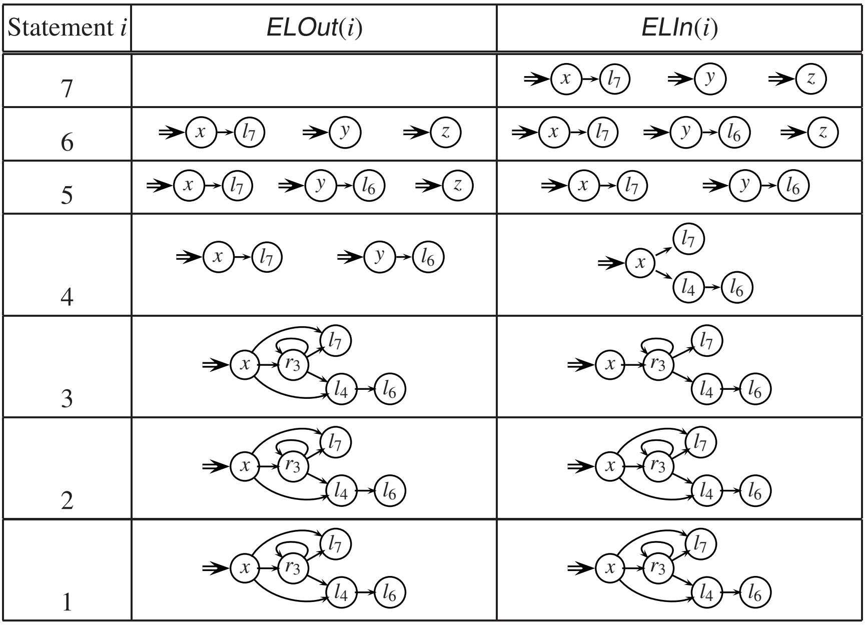


FIGURE 4.24
Explicit liveness for the program in Figure 4.19 on page 140 under the assumption that all variables are local variables.

this version of our analysis is context insensitive.

$$\mathsf{LDirect}_x(i) = \mathsf{makeGraph}(\mathsf{base}(\rho_x))$$
$$\mathsf{LDirect}_y(i) = \mathsf{makeGraph}(\mathsf{base}(\rho_y)) \uplus \mathsf{makeGraph}(\rho_y {-\!\triangleright} *)$$
$$\mathsf{LDirect}_z(i) = \begin{cases} \mathsf{makeGraph}(z {-\!\triangleright} *) & \text{if } z \text{ is a global variable} \\ \mathcal{E}_G & \text{otherwise} \end{cases}$$
$$\mathsf{LTransfer}_z(i) = \mathcal{E}_G, \text{ for all variables } z$$
$$\mathsf{ELKillPath}_x(i) = \rho_x$$
$$\mathsf{ELKillPath}_z(i) = \mathcal{E}, \text{ for any variable } z \text{ other than } x$$

3. *Return Statement* $\mathsf{return}\ \alpha_x$.

$$\mathsf{LDirect}_x(i) = \mathsf{prefixes}(\mathsf{base}(\rho_x)) \cup \mathsf{makeGraph}(\rho_x {-\!\triangleright} *)$$
$$\mathsf{LDirect}_z(i) = \begin{cases} \mathsf{makeGraph}(z {-\!\triangleright} *) & \text{if } z \text{ is a global variable} \\ \mathcal{E}_G & \text{otherwise} \end{cases}$$
$$\mathsf{LTransfer}_z(i) = \mathcal{E}_G, \text{ for any variable } z$$
$$\mathsf{ELKillPath}_z(i) = \mathcal{E}, \text{ for any variable } z$$

4. *Use Statements*

$$\begin{aligned}
\mathit{LDirect}_x(i) &= \biguplus \mathit{makeGraph}(\rho_x) \text{ for every } \alpha_x.d \text{ used in } i\\
\mathit{LDirect}_z(i) &= \mathcal{E}_G \text{ for any variable } z \text{ other than } x\\
\mathit{LTransfer}_z(i) &= \mathcal{E}_G, \text{ for every variable } z\\
\mathit{ELKillPath}_z(i) &= \mathcal{E}, \text{ for every variable } z
\end{aligned}$$

Example 4.21

Figure 4.24 lists explicit liveness information at different points of the program in Figure 4.19 on page 140 under the assumption that all variables are local variables. ▯

Observe that computing liveness using Equations (4.23) and (4.24) results in an *MFP* solution of data flow analysis whereas Definition 4.10 specifies an *MOP* solution of data flow analysis. Since the flow functions are non-distributive, the two solutions may be different.

Convergence of Explicit Liveness Analysis

We now show the termination of explicit liveness analysis using the properties of access graph operations. In particular, we show that the flow functions are monotonic and the data flow values form a finite complete lattice.

For a program there are a finite number of basic blocks, a finite number of fields for any root variable, and a finite number of field names in any access expression. Hence the number of access graphs for a program is finite. Further, the number of nodes and hence the size of each access graph, is bounded by the number of labels which can be created for a program.

Access graphs for a variable x form a complete lattice with a partial order $\sqsubseteq_G$ induced by $\uplus$. Note that $\uplus$ is commutative, idempotent, and associative. Let $G = \langle x, N_F, N_I, E\rangle$ and $G' = \langle x, N'_F, N'_I, E'\rangle$ where subscripts F and I distinguish between the final and intermediate nodes. The partial order $\sqsubseteq_G$ is defined as

$$G \sqsubseteq_G G' \Leftrightarrow \left(N'_F \subseteq N_F\right) \wedge \left(N'_I \subseteq (N_F \cup N_I)\right) \wedge (E' \subseteq E)$$

Clearly, $G \sqsubseteq_G G'$ implies that G contains all access paths of G'. We extend $\sqsubseteq_G$ to a set of access graphs as follows:

$$S_1 \sqsubseteq_S S_2 \Leftrightarrow \forall G_2 \in S_2, \exists G_1 \in S_1 \text{ s.t. } G_1 \sqsubseteq_G G_2$$

It is easy to verify that $\sqsubseteq_G$ is reflexive, transitive, and antisymmetric. For a given variable x, the access graph $\mathcal{E}_G$ forms the $\top$ element of the lattice while the $\bot$ element is a greatest lower bound of all access graphs.

The partial order over access graphs and their sets can be carried over unaltered to remainder graphs ($\sqsubseteq_{RG}$) and their sets ($\sqsubseteq_{RS}$), with the added condition that ϵ_{RG} is incomparable to any other non empty remainder graph.

Operation	Monotonicity
Union	$G_1 \sqsubseteq_G G'_1 \wedge G_2 \sqsubseteq_G G'_2$ $\Rightarrow G_1 \uplus G_2 \sqsubseteq_G G'_1 \uplus G'_2$
Path Removal	$G_1 \sqsubseteq_G G_2$ $\Rightarrow G_1 \ominus \rho \sqsubseteq_G G_2 \ominus \rho$
Factorization	$G_1 \sqsubseteq_G G_2$ $\Rightarrow G_1/(G,M) \sqsubseteq_{\mathsf{RS}} G_2/(G,M)$
Extension	$\mathsf{RS}_1 \sqsubseteq_{\mathsf{RS}} \mathsf{RS}_2 \wedge G_1 \sqsubseteq_G G_2 \wedge M_1 \subseteq M_2$ $\Rightarrow (G_1,M_1)\#\mathsf{RS}_1 \sqsubseteq_G (G_2,M_2)\#\mathsf{RS}_2$
Link-Alias Closure	$G_1 \sqsubseteq_G G'_1 \wedge G_2 \sqsubseteq_G G'_2$ $\Rightarrow \mathsf{LnG}(G_1,G_2,\langle g_x,g_y\rangle) \sqsubseteq_S \mathsf{LnG}(G'_1,G'_2,\langle g_x,g_y\rangle)$

FIGURE 4.25
Monotonicity of access graph operations.

Access graph operations are monotonic as described in Figure 4.25. Path removal is monotonic in the first argument but not in the second argument. Similarly factorization is monotonic in the first argument but not in the second and the third argument. However, we show that in each context where they are used, the resulting functions are monotonic:

1. Path removal is used only for an assignment $\alpha_x = \alpha_y$. It is used in liveness analysis and its second argument is ρ_x which is constant for any assignment statement $\alpha_x = \alpha_y$. Thus the resulting flow functions are monotonic.

2. Factorization is used during liveness analysis. It is used for the flow function corresponding to an assignment $\alpha_x = \alpha_y$. In this context, its second and third arguments are $\mathsf{makeGraph}(\rho_x)$ and $\mathsf{lastNode}(\mathsf{makeGraph}(\rho_x))$. Both these are constants for a given assignment statement $\alpha_x = \alpha_y$. Thus, the resulting flow functions are monotonic.

Thus we conclude that all flow functions are monotonic. Since lattices are finite, termination of explicit liveness analysis follows.

Efficiency of Explicit Liveness Analysis

This section discusses the issues which influence the efficiency of performing explicit liveness analysis.

The data flow frameworks defined in this paper are not *separable* [59] because the data flow information of a variable depends on the data flow information of other variables. Thus the number of iterations over control flow graph is not bounded by the depth of the graph [3, 44, 59] but would also depend on the number of root variables that depend on each other.

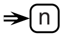
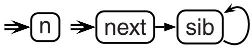
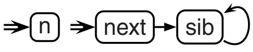
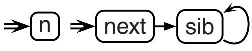
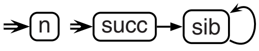
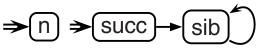
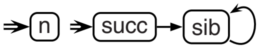
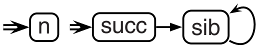
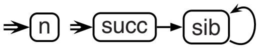
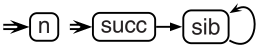
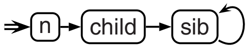
The amount of work done in each iteration is not fixed but depends on the size of access graphs. Of all operations performed in an iteration, only $\mathsf{CFN}(G,G')$ is costly. In practice, the access graphs are quite small because of the following reason: Recall that edges in access graphs capture dependence of a reference made at one program point on some other reference made at another point (Section 4.4.3). In real programs, traversals involving long dependences are performed using iterative constructs in the program. In such situations, the length of the chain of dependences is limited by the process of summarization because summarization treats nodes with the same label as being identical. Thus, in real programs chains of such dependences, and hence the access graphs, are quite small in size. Hence the complexities of access graph operations is not a matter of concern.

4.4.5 The Motivating Example Revisited

For our motivating example in Section 1.1, we had performed liveness analysis of heap data intuitively. The liveness information was represented using access paths which were summarized by combining all field names beyond the second field by a summary field "⋆". We now present the result of liveness analysis of the program in Figure 1.1 on page 2 in terms of access graphs.

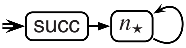
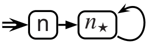
Intraprocedural Analysis by Ignoring the Interprocedural Effects

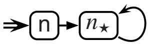
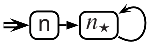
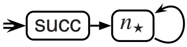
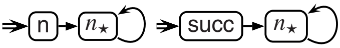
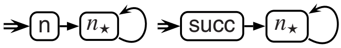
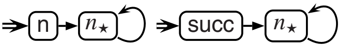
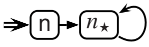
In this case we treat a function call as a statement that reads its actual parameters and assume that *BI* is $\mathcal{E}_G$.

Node	*Out*$_n$	*In*$_n$
n_6	$\mathcal{E}_G$	
n_5		
n_4		
n_3		
n_2		
n_1		

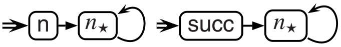
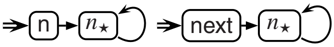
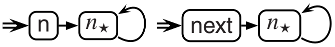
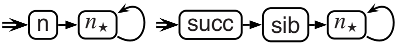
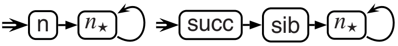
When we compare these results with the corresponding liveness information computed in Section 1.1.2, we observe that the above access graphs do not include access paths such as *succ*→*child*→*sib* or *succ*→*sib*→*child* whereas they are included in the liveness information computed in Section 1.1.2. This difference arises because of the difference between the summarization of access paths using access graphs and the summarization by restricting the lengths of access paths.

Intraprocedural Analysis with Conservative Interprocedural Approximation

As described earlier, the effect of the function call in our example can be incorporated conservatively by assuming that every access path rooted at n is live at the exit of `dfTraverse` and that every access path rooted at `succ` is live at the entry of n_3 due to the call. We use the special summary node $n_\star$ defined for access graph to denote any field name. Thus we assume that the function call creates the access graph  and *BI* is . With these assumptions, the data flow information after first iteration is:

Node	Iteration #1	
	Out$_n$	*In*$_n$
n_6		
n_5	$\mathcal{E}_G$	$\mathcal{E}_G$
n_4	$\mathcal{E}_G$	$\mathcal{E}_G$
n_3	$\mathcal{E}_G$	
n_2		
n_1		

If there is an edge $n \rightarrow n_\star$, then n cannot have any other out edge because all its successors are consumed by $n_\star$. The data flow values after second iteration are:

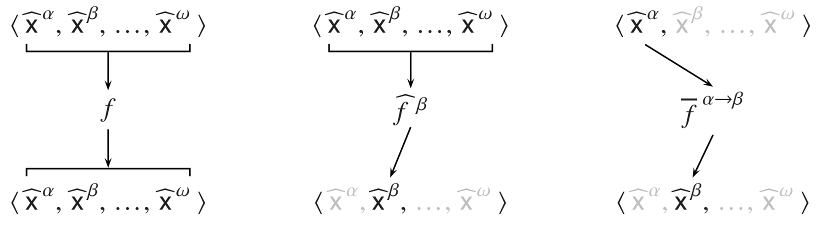
Node	Changes in Iteration #2	
	Out$_n$	*In*$_n$
n_6		
n_5		
n_4		
n_3		
n_2		
n_1		

There are no further changes. Observe that the values of *In*$_{n_4}$ and *Out*$_{n_3}$ are more precise than those in Section 1.1.2. This is because unlike the earlier summarization, access graphs do not restrict the length of access paths to two.

Interprocedural analysis of this example is presented in Section 9.5.

4.5 Modeling Entity Dependence

Recall that a component function $\widehat{f}^{\,\alpha} : L \mapsto \widehat{L}$ computes the data flow value of entity α. The domain of $\widehat{f}^{\,\alpha}$ is not atomic reflecting the fact that the data flow value of α


$\langle \widehat{\mathsf{x}}^{\alpha}, \widehat{\mathsf{x}}^{\beta}, \ldots, \widehat{\mathsf{x}}^{\omega} \rangle$
f
$\langle \widehat{\mathsf{x}}^{\alpha}, \widehat{\mathsf{x}}^{\beta}, \ldots, \widehat{\mathsf{x}}^{\omega} \rangle$
$\langle \widehat{\mathsf{x}}^{\alpha}, \widehat{\mathsf{x}}^{\beta}, \ldots, \widehat{\mathsf{x}}^{\omega} \rangle$
$\widehat{f}^{\beta}$
$\langle \widehat{\mathsf{x}}^{\alpha}, \widehat{\mathsf{x}}^{\beta}, \ldots, \widehat{\mathsf{x}}^{\omega} \rangle$
$\langle \widehat{\mathsf{x}}^{\alpha}, \widehat{\mathsf{x}}^{\beta}, \ldots, \widehat{\mathsf{x}}^{\omega} \rangle$
$\overline{f}^{\alpha\to\beta}$
$\langle \widehat{\mathsf{x}}^{\alpha}, \widehat{\mathsf{x}}^{\beta}, \ldots, \widehat{\mathsf{x}}^{\omega} \rangle$

(a) Overall function (b) Component function (c) *pef*

FIGURE 4.26
Defining overall flow function in terms of component and primitive entity functions.

could depend on the data flow values of other entities also. Thus even $\widehat{f}^{\alpha}$ need not be atomic. For some frameworks, it can be defined in terms of simpler functions that use the value of an entity to compute the value of another entity.

4.5.1 Primitive Entity Functions

We define *primitive entity functions* (abbreviated as *pef*) as the functions that compute the data flow value of an entity α from the data flow value of some entity β. We denote such a *pef* by $\overline{f}^{\beta\to\alpha}: \widehat{L}_{\beta} \mapsto \widehat{L}_{\alpha}$. The component function $\widehat{f}^{\alpha}_{u\to v}$ is defined as:

$$\widehat{f}^{\alpha}(\widehat{\mathsf{x}}^{\alpha}) = \bigsqcap_{\beta\in\Sigma} \overline{f}^{\beta\to\alpha}(\widehat{\mathsf{x}}^{\beta}) \tag{4.26}$$

Figure 4.26 illustrates how an overall flow function f can be a defined in terms of component functions $\widehat{f}^{\beta}$, and a *pef*s $\overline{f}^{\alpha\to\beta}$. $\mathsf{x} = \langle \widehat{\mathsf{x}}^{\alpha}, \widehat{\mathsf{x}}^{\beta}, \ldots, \widehat{\mathsf{x}}^{\omega} \rangle$ is the input data flow value and $\mathsf{x} = \langle \widehat{\mathsf{x}}^{\alpha}, \widehat{\mathsf{x}}^{\beta}, \ldots, \widehat{\mathsf{x}}^{\omega} \rangle$ is the output data flow value.

Modeling component functions in terms of *pef*s is interesting because it allows the component functions to be defined in terms of a very small set of *pef*s. We explain this by distinguishing between general unspecified functions and specific known functions. Our notation f denotes a general unspecified function. When we wish to denote specific known functions computing specific values, we use the notation ϕ. Unlike the subscript of f which denotes a program point or an edge, the subscript of ϕ distinguishes it from other specific functions. A couple of common special functions are:

$$\forall \mathsf{x} \in L:\ \phi_{id}(\mathsf{x}) = \mathsf{x}$$
$$\forall \mathsf{x} \in L:\ \phi_{\mathsf{z}}(\mathsf{x}) = \mathsf{z}$$

There are two special values of ϕ_{z} that are used very frequently: They are $\phi_{\top}$ and $\phi_{\bot}$. The specific functions that can be used for component functions and *pef*s are denoted by $\widehat{\phi}$ that are defined as follows:

$$\forall \widehat{\mathsf{x}} \in \widehat{L} : \widehat{\phi}_{\mathsf{z}}(\widehat{\mathsf{x}}) = \widehat{\mathsf{z}}$$
$$\forall \widehat{\mathsf{x}} \in \widehat{L} : \widehat{\phi}_{id}(\widehat{\mathsf{x}}) = \widehat{\mathsf{x}}$$
$$\forall \widehat{\mathsf{x}} \in \widehat{L}, \forall m, n \in \mathbb{C}\mathsf{onst} : \widehat{\phi}_{m,n}(\widehat{\mathsf{x}}) = m \times \widehat{\mathsf{x}} + n$$

$\widehat{\phi}_{\mathsf{z}}$ are constant functions. They include $\widehat{\phi}_{\top}$ and $\widehat{\phi}_{\perp}$ also. Some other examples of constant functions are: *pef*s corresponding to constant value assignments such as $a = 2$ in constant propagation, *pef*s corresponding to constant address assignments such as $a = \&b$ in point-to analysis etc. The latter is possible because the address of each named variable is a compile time constant.[‡]

$\widehat{\phi}_{id}$ is an identity *pef*. Note that the domain of $\widehat{\phi}_{id}$ could be $\widehat{L}_{\alpha}$ and the range could be $\widehat{L}_{\beta}$. Yet, it is an identity function because the component lattices $\widehat{L}_{\alpha}$ and $\widehat{L}_{\beta}$ are identical in terms of values and structure. In separable frameworks, for every identity *pef*, $\alpha = \beta$. Examples of $\widehat{\phi}_{id}$ with $\alpha \neq \beta$ are functions corresponding to copy statements such as $a = b$ in non-separable frameworks like possibly uninitialized variable analysis, constant propagation or points-to analysis.

Together, $\widehat{\phi}_{\mathsf{z}}$ and $\widehat{\phi}_{id}$ cover all bit vector frameworks, all fast frameworks, all non-separable frameworks in which the data flow values can be represented by bit vectors (e.g., faint variables analysis, possibly uninitialized variables analysis), and copy constant propagation. They also cover a restricted points-to analysis if the right hand side does not involve indirection. The last *pef* $\widehat{\phi}_{m,n}$ is included to cover linear constant propagation. It is easy to see that all these *pef*s are distributive and are closed under composition. The frameworks whose component functions can be defined using the above *pef*s are called *primary* frameworks.

If an entity β does not influence α, then $\overline{f}^{\beta \to \alpha} = \phi_{\top}$. A separable framework is a special case of non-separable frameworks such that

$$\alpha \neq \beta \Rightarrow \forall \widehat{\mathsf{x}}^{\beta} \in \widehat{L}_{\beta}, \overline{f}^{\beta \to \alpha}\left(\widehat{\mathsf{x}}^{\beta}\right) = \widehat{\top}$$

This reduces $\widehat{f}^{\alpha}$ from $L \mapsto \widehat{L}_{\alpha}$ to $\widehat{L}_{\alpha} \mapsto \widehat{L}_{\alpha}$.

Example 4.22

Given an assignment $a = b * c$, some examples of component functions for some data flow frameworks are as follows:

- Available expressions analysis: $\widehat{f}^{b*c} = \widehat{\phi}_{\top}$; for any expression e that involves a, $\widehat{f}^{e} = \widehat{\phi}_{\perp}$; and for an expression e that does not involve a, $\widehat{f}^{e} = \widehat{\phi}_{id}$.

[‡] As is customary, addresses defined in terms of fixed offsets from frame pointers in activations records are considered compile time constants even if the actual address depends on run time.

- Live variables analysis: $\widehat{f}^{\,a} = \widehat{\phi}_{\perp}$; $\widehat{f}^{\,b} = \widehat{f}^{\,c} = \widehat{\phi}_{\top}$; and for any variable x other than a, b, and c, $\widehat{f}^{\,x} = \widehat{\phi}_{id}$.

- Faint variables analysis $\widehat{f}^{\,a} = \widehat{\phi}_{\perp}$; $\widehat{f}^{\,b} = \widehat{\phi}_{id}^{b} \sqcap \widehat{\phi}_{id}^{a}$; $\widehat{f}^{\,c} = \widehat{\phi}_{id}^{c} \sqcap \widehat{\phi}_{id}^{a}$; and for any variable x other than a, b, and c, $\widehat{f}^{\,x} = \widehat{\phi}_{id}$.

For an assignment $a = 2$ in constant propagation, $\widehat{f}^{\,a} = \widehat{\phi}_{2}$; for an assignment $a = b$, $\widehat{f}^{\,a} = \widehat{\phi}_{id}^{b}$; and for an assignment $a = b + 2$, $\widehat{f}^{\,a} = \widehat{\phi}_{1,2}^{b}$. For any variable x other than a, $\widehat{f}^{\,x} = \widehat{\phi}_{id}$. □

Example 4.23
Consider the flow functions in explicit liveness analysis of heap data. An access graph consists of edges between nodes. Since the set of nodes that may occur in any access graph is fixed for an instance of explicit liveness analysis, the set of possible edges is also fixed. Thus we define the following *pef*s for an edge: $\widehat{\phi}_{\perp}$ adds an edge to the given graph, $\widehat{\phi}_{\top}$ removes an edge from the given graph whereas $\widehat{\phi}_{id}$ copies an edge. Thus the flow functions of liveness analysis defined in Section 4.4.4 can be formulated in terms of these three *pef*s. □

4.5.2 Composite Entity Functions

The component functions of full constant propagation and points-to analysis cannot be defined in terms of *pef*s. Such frameworks are not primary frameworks.

Flow functions in full constant propagation evaluate an arithmetic expression and if we wish to define component functions in terms of simpler functions, we will have to use the functions of the form $\widehat{L} \times \widehat{L} \mapsto \widehat{L}$. Such functions are neither distributive nor closed under composition. In points-to analysis a right hand side could involve an indirection like $*x$. In such a situation computing right locations requires collecting points-to information of all z that x could point to. Contrast this with the right hand side x; in this case, the right locations consist of only the points-to information of x. Thus the required function has the form $L \mapsto \widehat{L}$.

The component functions that cannot be defined in terms of primitive entity functions are defined in terms of *composite entity functions* (abbreviated as *cef*) where *cef*s themselves are defined as combinations of *pef*s. For example, addition of two variables in full constant propagation is represented by the composite entity function $\widehat{\phi}_{+}^{\alpha,\beta} : \widehat{L} \times \widehat{L} \mapsto \widehat{L}$ defined below:

$$\widehat{\phi}_{+}^{\alpha,\beta} = \widehat{\phi}_{id}^{\alpha} + \widehat{\phi}_{id}^{\beta}$$

Indirection in points-to analysis is defined in terms of composite entity function $\widehat{\phi}_{*}^{\alpha} : L \mapsto \widehat{L}$ defined below:

$$\widehat{\phi}_{*}^{\alpha} = \bigsqcap_{\alpha \rightarrow \beta} \widehat{\phi}_{id}^{\beta}$$

Example 4.24
For an assignment $x = y + z$ in constant propagation, $\widehat{f}^{\,x} = \widehat{\phi}_{+}^{y,z}$ and for every variable w other than x, $\widehat{f}^{\,w} = \widehat{\phi}_{id}$. For an assignment $x = *y$ in points-to analysis, $\widehat{f}^{\,x} = \widehat{\phi}_{*}^{y}$. Observe that modeling assignment $*x = y$ does not require a special function because we define $\widehat{f}^{\,z} = \widehat{\phi}_{id}$ for every z such that $x \rightarrow z$. ▯

4.6 Summary and Concluding Remarks

In this chapter we have extended the *Gen-Kill* model of bit vector frameworks to general frameworks. The largest class of practical problems that can be described using this extended model are non-separable frameworks. In principle, separable frameworks can also have dependent parts and this model captures such frameworks also. However, the focus of this chapter has been on non-separable frameworks because we are not aware of a practical separable framework that is not a bit vector framework.

The extended *Gen-Kill* model can be seen as a uniform specification model for semantics captured by an analysis. This is useful because it allows flow functions to be decomposed in similar parts so that flow functions of different frameworks can be compared and contrasted. This facilitates modeling flow functions at a finer granularity in terms of primitive and composite entity functions. Surprisingly, a very small set of *pef*s is sufficient to model flow functions in most frameworks despite the diversity of the data flow information. As shown in Section 4.5, four *pef*s are enough to model almost all frameworks except full constant propagation and points-to analysis in which addresses of pointers are taken. This should be contrasted with the conventional modeling where flow functions remain at a much higher level of abstraction $f : L \mapsto L$ and no attempt is made to examine their constituents. Two significant benefits of modeling flow functions in terms of *pef*s are that

- it becomes possible to devise tight complexity bounds for round-robin iterative analysis of a large class of data flow frameworks. We do so in Chapter 5.
- it becomes possible to devise feasibility conditions for systematic reduction of flow function compositions.

4.7 Bibliographic Notes

The term separability was coined by Khedker and Dhamdhere [60]. Separable frameworks were called “decomposable” by Sharir and Pnueli [93] whereas Rosen [84] had called them “factorizable”.

Constant propagation was described by Kildall [63] and it has been widely studied in literature. Some important works include conditional constant propagation by Wegman and Zadeck [102] and complexity study of many variants of constant propagation by Müller-Olm and Rüthing [78]. Strongly live variables analysis, which is a dual of faint variables analysis can be found in the text by F. Nielson, H. R. Nielson and Hankin [80].

There is a plethora of literature on pointer analysis. Unlike our presentation which tries to present a clean model of pointer analysis independently of other concerns, most of the works on pointer analysis have almost always given a much higher preference to practical concerns such as efficiency and effectiveness in interprocedural settings. Thus many combinations of flow sensitivity and context sensitivity have been explored. Even among flow insensitive approaches, two separate categories of *equality-based* and *subset-based* methods have been studied. Equality-based method assumes that if a can point to b and c, then b can point to everything that c can point to and vice-versa. Subset-based does not unify the points-to sets of b and c. Equality-based approach was pioneered by Steensgaard [97] whereas subset-based approach was pioneered by Andersen [9]. Fahndrich, Foster, Su and Aiken [35] presented an Andersen style of context insensitive pointer analysis which was followed up by Steensgaard style of context sensitive pointer analysis [36]. Andersen style context sensitive pointer analysis was reported by Whaley and Lam [104]. Among other influential works on pointer analysis, Landi and Ryder [66, 67] have presented flow sensitive pointer analysis which is also context sensitive in non-recursive parts of programs. The work done by Choi, Burke and Carini [21, 48] belongs to the same category. The only pointer analysis that is flow sensitive and also context sensitive for recursive programs is by Emami, Ghiya and Hendren [34]. Our version of points-to analysis is based on its reformulation by Kanade, Khedker and Sanyal [51]. An excellent discussion of the state of art of pointer analysis has been presented by Hind [47].

Liveness analysis of heap data using access graphs is a recent work by Khedker, Sanyal and A. Karkare [62]. We have only presented explicit liveness analysis. Actual nullification requires some other analyses such as alias analysis, availability analysis, anticipability analysis, and nullability analysis [62].

The earlier attempt at discovering the liveness of heap was by Agesen, Detlefs and Moss [1] and was restricted to the liveness of root variables. Our approach of heap data analysis can be seen as some kind of *shape analysis* [88, 106] which is a general method of creating suitable abstractions (called Shape Graphs) of heap memory with respect to the relevant properties. Program execution is then modeled as operations on shape graphs. However, it is not clear how shape analysis can be directly used for discovering future properties like liveness that require analysis against control flow. Shaham, Yahav, Kolodner and Sagiv [92] have devised a restricted version of liveness of heap data using shape analysis.

The concept of modeling flow functions in terms of primitive entity functions has been proposed by B. Karkare [53].