Part I

Intraprocedural Data Flow Analysis

2

Classical Bit Vector Data Flow Analysis

Data flow analysis originated with what was later termed as "bit vector" data flow frameworks. The term "bit vector" arises from the fact that not only can the data flow information be represented using bit vectors, it can also be computed using bit vector operations alone. There are data flow former for which although the data flow information can be represented using bit vectors, computing it requires additional operations. We make this notion more precise in the chapter summary with the help of the examples presented in the chapter.

2.1 Basic Concepts and Notations

Data flow analysis views computation of data through expressions and transition of data through assignments to variables. Properties of programs are defined in terms of properties of program entities such as expressions, variables, and definitions appearing in a program. In this chapter, we restrict expressions to primitive expressions involving a single operator. Variables are restricted to scalar variables and definitions are restricted to assignments made to scalar variables. Data flow analyses of other program entities such as composite expressions, array variables, pointer variables, statement numbers etc. have also been devised; we present some of them in later chapters.

For a given program entity such as an expression, data flow analysis of a program involves the following two steps (a) discovering the effect of individual statements on the expression, and (b) relating these effects across statements in the program. For reasons of efficiency, both these steps are often carried over a *basic block* instead of a single statement. A basic block is a maximal group of consecutive statements that are always executed together with a strictly sequential control flow between them. Step (a) is called *local* data flow analysis and is performed for a basic block only once. Step (b) constitutes *global* data flow analysis* and may require repeated traversals over basic blocks in a CFG. Since global analysis correlates local properties, combining local analysis of several statements together and performing global

*Observe that the term global data flow analysis is restricted to data flow analysis of a single procedure.

analysis over the resulting basic blocks rather than individual statements implies lesser work for global analysis.

Relating the effects across basic blocks involves propagating data flow information from a basic block to another along the direction of control flow or against it. Propagation along the direction of control flow constitutes a *forward flow* whereas propagation against the direction of control flow constitutes a *backward flow*. As observed in Sections 1.1.2 and 1.1.3, liveness analysis involves backward flows and alias analysis involves forward flows.

Global data flow information is associated with the entry and exit points of a basic block. For block n these points are denoted by $\mathit{Entry}(n)$ and $\mathit{Exit}(n)$; they represent the possible states of the program just before the execution of the first statement and the just after the execution of the last statement in block n. Data flow information associated with them is usually denoted by In_n and Out_n. For bit vector frameworks, the local data flow information is usually expressed in terms of Gen_n and Kill_n. Gen_n denotes the data flow information which is generated within block n whereas Kill_n denotes the data flow information which becomes invalid in block n.

The relationship between local and global data flow information for a block (i.e., Gen_n, Kill_n, In_n, and Out_n) and between global data flow information across different blocks is captured by a system of linear simultaneous equations called *data flow equations*. In general, these equations have multiple solutions. This makes it important to choose the initial values of In_n and Out_n carefully.

Edges in CFGs denote the *predecessor* and *successor* relationships: If there is an edge $n_1 \rightarrow n_2$, then n_1 is a predecessor of n_2 and n_2 is a successor of n_1. Observe that this is different from the notions of *ancestors* and *descendants* which are the transitive closures of predecessors and successors respectively. Predecessors and successors of a block n are denoted by $\mathit{pred}(n)$ and $\mathit{succ}(n)$ respectively.

We assume that the CFG has two distinguished unique nodes: Start which has no predecessor and End which has no successor. If such nodes do not exist, dummy nodes can be added without affecting the program semantics. It is further assumed that every basic block n is reachable from the Start block and that the End block is reachable from n. We use the terms nodes and blocks interchangeably.

2.2 Discovering Local Data Flow Information

The manner in which the effect of a statement is modeled varies from one analysis to another. In any case, there is a common pattern of generation of data flow information or invalidation of data flow information. In this chapter we are interested in the following entities and operations related to data flow analysis:

Entity	Operations	
Variable $x \in \mathbb{V}\mathsf{ar}$	Reading the value of x	Modifying the value of x
Expression $e \in \mathbb{E}\mathsf{xpr}$	Computing e	Modifying an operand of e
Definition $d_i : x = e$, $d_i \in \mathbb{D}\mathsf{efs}$, $x \in \mathbb{V}\mathsf{ar}$, $e \in \mathbb{E}\mathsf{xpr}$	Occurrence of d_i	Any definition of x

Reading the value of a variable is also termed as the *use* of the variable. A variable may be used or an expression may be computed (a) in the right hand side of an assignment statement, (b) in a condition for altering flow of control, (c) as an actual parameter in a function call, or (d) as a return value from a function. All other operations in the above table involve an assignment statement to a relevant variable. Note that reading a value of a variable from input can be safely considered as an assignment statement assigning an unknown value to the variable.

The set Gen_n and Kill_n are computed from the operations described above. It is easy to see that the operation in one column nullifies the effect of the operation in the other column. From that viewpoint, the operation in one column is an inverse of the operation in the other column. Computing Gen_n and Kill_n requires identifying operations that are exposed in the direction of analysis i.e., are not followed by the inverse operation in the direction of analysis. For forward problems, we are interested in the operations that are *downwards exposed* and for the backward problems we are interested in the operations that are *upwards exposed*. This is illustrated by the following example.

Example 2.1

Consider an assignment statement $x = x + 1$. In this statement, the use of variable x and the computation of expression $x + 1$ are upwards exposed because they are not preceded by a modification of the value of x. They are not downwards exposed because they are followed by a modification of the value of x. As a contrasting example, the use of x and computation of $x + 1$ are both upwards and downwards exposed in an assignment $y = x + 1$ if x and y do not have the same address (i.e., they are not aliased). □

Traditionally, the definitions of Gen_n and Kill_n have not been symmetric with respect to the chosen operation. In particular, the operations which contribute to Gen_n are required to be downwards exposed for forward flows and upwards exposed for backward flows. The operations which contribute to Kill_n may be preceded or followed by their inverses. We explain this asymmetry later in the specific contexts of the data flow problems presented in this chapter.

Local property computation isolates global analysis from the intermediate representation (IR) in that it is the former which needs to examine the IR statements. In practice, IRs in real compilers are very complicated since they need to store a lot of information about each statement across different phases of a compiler. Hence local property computations are tedious and error-prone. Global data flow analyzers are relatively much simpler and cleaner.

2.3 Discovering Global Properties of Variables

In this section, we describe two analyses involving variables: *Live Variables Analysis* and *Reaching Definitions Analysis*. Although we have listed a definition as a separate entity, here we club its analysis with those of variables.

2.3.1 Live Variables Analysis

Section 1.1.2 has introduced liveness analysis for heap data. Liveness analysis for scalar variables essentially involves determining whether a variable is used in future and is relatively much simpler because it does not have to consider pointer dereferencing.

DEFINITION 2.1 *A variable $x \in \mathbb{V}\mathsf{ar}$ is live at a program point u if some path from u to End contains a use of x which is not preceded by its definition.*

The data flow equations which define live variables analysis are:

$$In_n = (Out_n - Kill_n) \cup Gen_n \tag{2.1}$$

$$Out_n = \begin{cases} BI & n \text{ is } End \text{ block} \\ \bigcup_{s \in succ(n)} In_s & \text{otherwise} \end{cases} \tag{2.2}$$

where In_n, Out_n, Gen_n, $Kill_n$, and BI are sets of variables.

Liveness at $Exit(End)$ is represented by BI. This is required because different categories of variables have to be treated differently. Local variables are not live at $Exit(End)$ whereas liveness of the return value, global variables, and parameters passed by reference depends on the calling contexts. If there is no interprocedural analysis, all variables other than local variables are assumed to be live. We assume that all our analyses in Part I are restricted to local entities only. Thus we will define BI for local entities only. Under the assumption of parameter passing by value as in C, this also allows us to ignore function calls completely.

Observe the use of $\cup$ in Equation (2.2). It essentially means that the liveness information at $Exit(n)$ is a superset of the liveness information at $Entry(s)$ where s is a successor of n. This is consistent with the "any path" nature of the definition of liveness: Subsequent use along a single path is sufficient to make a variable live. Further, since data flow information at a node depends on the successor nodes, this is a backward data flow problem.

Gen_n contains the variables whose liveness is generated within n. Clearly, these variables have upwards exposed uses in n. $Kill_n$ contains the variables whose liveness is killed in n. These are the variables which appear on the left hand side of an assignment anywhere in n. Observe that Gen_n and $Kill_n$ need not be mutually exclusive.

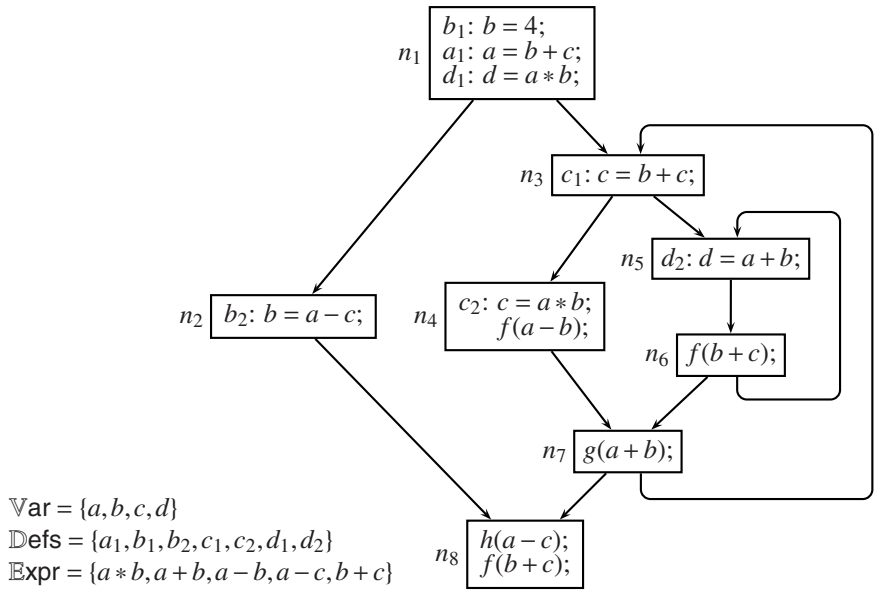


FIGURE 2.1
Program for illustrating bit vector data flow frameworks.

Example 2.2
In Figure 2.1, variable c is contained in both Gen_{n_3} and $Kill_{n_3}$. □

In general, assuming that variable x is live at $Exit(n)$, there are four possibilities with four distinct semantics:

Case	Local Information		Effect on Liveness
1	$x \notin Gen_n$	$x \notin Kill_n$	Liveness of x is unaffected in block n
2	$x \in Gen_n$	$x \notin Kill_n$	Liveness of x is generated in block n
3	$x \notin Gen_n$	$x \in Kill_n$	Liveness of x is killed in block n
4	$x \in Gen_n$	$x \in Kill_n$	Liveness of x is unaffected in block n in spite of x being modified in n.

Variable x is live at $Entry(n)$ in cases 1, 2, and 4 but the reason for its liveness is different in each case. In particular, case 4 captures the fact that the liveness of x is killed in n but is re-generated within n. The reason why this needs to be distinguished from case 1 and case 2 is that in some instances, it is important to know whether the value of a variable is modified in a block or not.

Example 2.3
We provide a trace of liveness analysis for the program flow graph in Figure 2.1. Since this analysis involves backward flows, we prefer to traverse the

CFG in the reverse postorder. We use $\emptyset$ as the initialization and assume that all variables are local implying that BI is $\emptyset$.

Block	Local Information		Global Information			
			Iteration # 1		Iteration # 2	
	Gen_n	$Kill_n$	Out_n	In_n	Out_n	In_n
n_8	$\{a,b,c\}$	$\emptyset$	$\emptyset$	$\{a,b,c\}$	$\emptyset$	$\{a,b,c\}$
n_7	$\{a,b\}$	$\emptyset$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$
n_6	$\{b,c\}$	$\emptyset$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$
n_5	$\{a,b\}$	$\{d\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$
n_4	$\{a,b\}$	$\{c\}$	$\{a,b,c\}$	$\{a,b\}$	$\{a,b,c\}$	$\{a,b\}$
n_3	$\{b,c\}$	$\{c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$	$\{a,b,c\}$
n_2	$\{a,c\}$	$\{b\}$	$\{a,b,c\}$	$\{a,c\}$	$\{a,b,c\}$	$\{a,c\}$
n_1	$\{c\}$	$\{a,b,d\}$	$\{a,b,c\}$	$\{c\}$	$\{a,b,c\}$	$\{c\}$

Observe that the data flow values computed in the second iteration are identical to the values computed in the first iteration indicating convergence. We leave it to the reader to verify that the final result would be same even if the graph is traversed in postorder; the only difference is that it will take many more iterations.

Observe that the result would be different if we had used the universal set (in this case $\{a,b,c,d\}$) as the initialization. Then, d would have been live at $Exit(n_7)$ whereas d is not used anywhere in the program. □

For brevity, we will show only new values computed in an iteration in subsequent examples—if a value is same as in the previous iteration, we will not show it explicitly. Hence we will not show the data flow values in the last iteration.

Two major applications of liveness analysis are in *register allocation* and *dead code elimination*. If a variable x is live at a program point, the current value of x is likely to be used along some execution path and hence x is a potential candidate for being allocated a register. On the other hand, if x is not live, the register allocated to x can be allocated to some other variable without the need of storing the value of x in memory. If x is not live at a exit of an assignment of x, then this assignment can be safely deleted.[†] For example, in Figure 2.1, variable d is not live anywhere. Thus all assignments of d can be safely eliminated.

In some cases deleting such assignments can have a transitive effect because the variables used in the right hand side of such an assignment may cease to be live. Instead of repeating the sequence of liveness analysis and dead code elimination, it is possible to discover such transitive effects through a single data flow analysis before dead code elimination is performed. This analysis is called *faint variables analysis* and will be presented in Chapter 4. Note that such an analysis cannot be restricted to a single variable at a time because the liveness of variables occurring on

[†]Deletion of code which is unreachable is also called dead code elimination but we will restrict dead code elimination to deletion of assignments to values which have no further use.

the right hand side of an assignment now also depends on the liveness of the variable on the left hand side. Such analyses are not bit vector analyses in spite of the fact that some of them use bit vector representation for data flow information. This is because the effect of basic blocks in these analyses are not expressible in terms of constant functions defined using *Gen* and *Kill* due to inter-dependence of various entities. Such frameworks are called non-separable. We describe them in Chapter 4.

For a given variable x, liveness analysis discovers a set of *liveness paths*. Each liveness path is a sequence of blocks $(b_1, b_2, \ldots, b_k)$ which is a prefix of some potential execution path starting at b_1 such that:

- b_k contains an upwards exposed use of x, and
- b_1 is either *Start* or contains an assignment to x, and
- no other block on the path contains an assignment to x.

Example 2.4

Some liveness paths for variable c in our example program are: (n_4, n_7, n_8), $(n_3, n_5, n_6, n_7, n_8)$, $(n_3, n_5, n_6, n_5, n_6, n_7, n_8)$, and (n_1, n_2, n_8). □

2.3.2 Dead Variables Analysis

A variable is dead (i.e., not live) if it is dead along all paths. If we wish to perform dead variables analysis instead of live variables analysis, the interpretation of In_n and Out_n changes: If a variable is contained in In_n or Out_n, it is dead instead of being live. This requires the following changes:

- The definitions of Gen_n and $Kill_n$ will change. Gen_n will now contain all variables whose values are modified in the block such that the modifications are upwards exposed (i.e., are not preceded by a use of the variable). $Kill_n$ will contain variables which are used anywhere regardless of what precedes or follows the uses. Observe that this is different from merely swapping Gen_n and $Kill_n$ of liveness analysis.
- We will have to use $\cap$ rather than $\cup$ for merging information.
- We will have to use the universal set as initialization rather than empty set. Similarly, *BI* will now have a different set of variables.

2.3.3 Reaching Definitions Analysis

A definition of a variable x is a statement which assigns a value to x. For the purpose of analysis, a unique label is associated with each assignment and these labels are used to represent the definitions. As a consequence, different occurrences of the same assignment become different definitions. This is different from uses of variables or

computation of expressions—labels are not associated with them and hence lexically same computations are not treated as different entities for analysis.

DEFINITION 2.2 *A definition $d_i \in \mathbb{D}\mathsf{efs}$ of a variable $x \in \mathbb{V}\mathsf{ar}$ reaches a program point u if d_i occurs on some path from Start to u and is not followed by any other definition of x on this path.*

The data flow equations which define the required analysis are:

$$In_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcup_{p \in pred(n)} Out_p & \text{otherwise} \end{cases} \tag{2.3}$$

$$Out_n = (In_n - Kill_n) \cup Gen_n \tag{2.4}$$

where In_n, Out_n, Gen_n, $Kill_n$, and BI are sets of definitions. Observe the use of $\cup$ to capture the "any path" nature of data flow. This is similar to liveness analysis except that now the data flow is forward rather than backward.

For every local variables x, it is assumed that a fictitious definition $x = undef$ reaches $Entry(Start)$. This is required for the optimization of copy propagation (described in Section 2.3.4). If definition $x = undef$ reaches a use of x, it suggests a potential use before definition. Whether this happens at run time depends on the actual results of conditions along the path taken to reach the program point.

Gen_n contains downwards exposed definitions in n whereas $Kill_n$ contains all definitions of all variables modified in n. Thus $Gen_n \subseteq Kill_n$ for reaching definitions analysis.

Example 2.5

The labels of assignments in the program in Figure 2.1 consist of variable names and an instance number. We use them to represent the definitions in the programs. Definitions a_0, b_0, c_0, and d_0 represent the special definitions $a = undef$, $b = undef$, $c = undef$, and $d = undef$ respectively. Since the confluence operation is $\cup$, the initial value at each program point is $\emptyset$.

The result of performing reaching definitions analysis has been shown in Figure 2.2. The definitions which reach $Exit(n_6)$ and $Exit(n_7)$ in first iteration have to be propagated to $Entry(n_5)$ and $Entry(n_3)$ respectively requiring an additional iteration. □

Reaching definitions analysis is used for constructing *use-def* and *def-use* chains which connect definitions to their uses as illustrated in the following example. These chains facilitate several optimizing transformations.

Example 2.6

Figure 2.3 shows the use-def and def-use chains of variables a and c in our example program. For simplicity, we have not shown the chains for other

Block	Local Information		Global Information			
			Iteration # 1		Changed values in iteration # 2	
	Gen_n	$Kill_n$	In_n	Out_n	In_n	Out_n
n_1	$\{a_1, b_1, d_1\}$	$\{a_0, a_1, b_0, b_1, b_2, d_0, d_1, d_2\}$	$\{a_0, b_0, c_0, d_0\}$	$\{a_1, b_1, c_0, d_1\}$		
n_2	$\{b_2\}$	$\{b_0, b_1, b_2\}$	$\{a_1, b_1, c_0, d_1\}$	$\{a_1, b_2, c_0, d_1\}$		
n_3	$\{c_1\}$	$\{c_0, c_1, c_2\}$	$\{a_1, b_1, c_0, d_1\}$	$\{a_1, b_1, c_1, d_1\}$	$\{a_1, b_1, c_0, c_1, c_2, d_1, d_2\}$	$\{a_1, b_1, c_1, d_1, d_2\}$
n_4	$\{c_2\}$	$\{c_0, c_1, c_2\}$	$\{a_1, b_1, c_1, d_1\}$	$\{a_1, b_1, c_2, d_2\}$	$\{a_1, b_1, c_1, d_1, d_2\}$	$\{a_1, b_1, c_2, d_1, d_2\}$
n_5	$\{d_2\}$	$\{d_0, d_1, d_2\}$	$\{a_1, b_1, c_1, d_1\}$	$\{a_1, b_1, c_1, d_2\}$	$\{a_1, b_1, c_1, d_1, d_2\}$	
n_6	$\emptyset$	$\emptyset$	$\{a_1, b_1, c_1, d_2\}$	$\{a_1, b_1, c_1, d_2\}$		
n_7	$\emptyset$	$\emptyset$	$\{a_1, b_1, c_1, c_2, d_1, d_2\}$	$\{a_1, b_1, c_1, c_2, d_1, d_2\}$		
n_8	$\emptyset$	$\emptyset$	$\{a_1, b_1, b_2, c_0, c_1, c_2, d_1, d_2\}$	$\{a_1, b_1, b_2, c_0, c_1, c_2, d_1, d_2\}$		

FIGURE 2.2
Reaching definitions analysis for Example 2.5.

variables. Observe that the definition c_0 reaches some uses of c. This suggests a potential use before any assigning meaningful value. This, in turn, makes variable b potentially undefined. □

Transitive effects of undefined variables are captured by *possibly uninitialized variables analysis*. Similar to faint variables analysis which captures transitive effect of dead variables, possibly uninitialized variables analysis is also non-separable—whether a variable is possibly undefined may depend on whether other variables are possibly undefined.

For definition x_i of variable x, reaching definitions analysis discovers a set of *definition reaching paths*. This path is a sequence of blocks $(b_1, b_2, \ldots, b_k)$ which is a prefix of some potential execution path starting at b_1 such that:

- b_1 contains the definition x_i
- b_k is either *End* or contains a definition of x
- no other block in the path contains a definition of x.

Example 2.7

Some definition reaching paths for variable c in our example program are: (n_4, n_7, n_3), $(n_3, n_5, n_6, n_5, n_6, n_7, n_8)$, and $(n_3, n_5, n_6, n_7, n_3)$. □

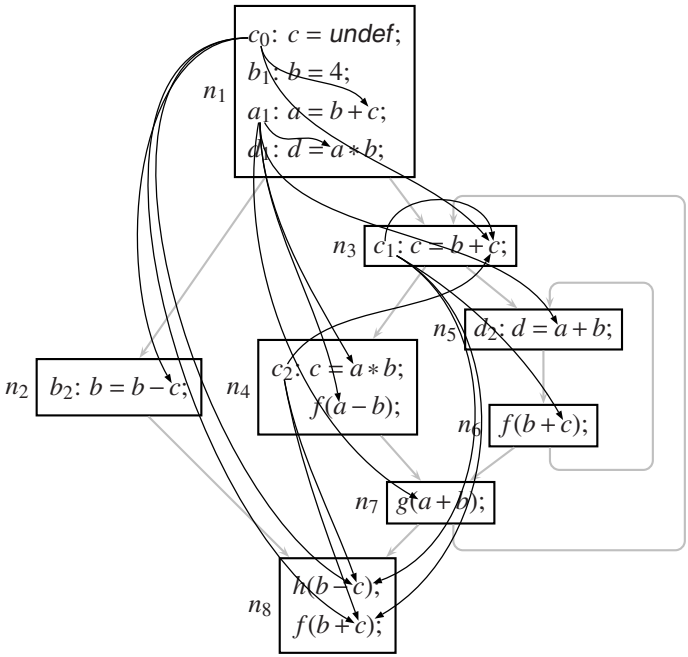


FIGURE 2.3
Def-use chains of variables a and c in our example program.

2.3.4 Reaching Definitions for Copy Propagation

Another application of reaching definitions analysis is in performing *copy propagation*. A definition of the form $x = y$ is called a *copy* because it merely copies the value of y to x. When such a definition reaches a use of x, and no other definition of x reaches that use then the use of x can be replaced by y.

Example 2.8
Copy $b = 4$ in block n_1 in our example program is the only definition which reached the uses of b in blocks n_3, n_4, n_5, n_6 and n_7. Thus all these uses can be replaced by constant 4. □

In the above example, the right hand side value is constant. When they are variables, as in $x = y$, replacing the uses of x by y requires an additional check that the value of y has not been modified along the path from the copy to the use. We can define a variant of reaching definitions analysis to accomplish this. The main difference between this variant and the analysis presented in Section 2.3.3 is that we restrict the definitions to copies and a definition $x = y$ is contained in

- Gen_n if it is downwards exposed in n in the sense of not being followed by a

definition of x or y, and in

- $Kill_b$ if n contains a definition of x or y.

With these changes, we can now perform reaching definitions analysis. If one definition reaches a use, we can perform copy propagation.

Note that this optimization does not improve the program on its own but it has the potential of creating dead code: When copy propagation is performed using $x = y$, it is possible that all uses of x are replaced by y thus making x dead after the assignment. Thus this assignment can be safely deleted.

We leave it for the reader to define a variant of copy propagation analysis using intersection rather than union.

2.4 Discovering Global Properties of Expressions

In this section we present analyses for eliminating redundant computations of expressions. Our first analysis involves replacing an expression by its precomputed value. The remaining analyses facilitate *code movement* which involve advancing computation an expression to earlier points in control flow paths.

2.4.1 Available Expressions Analysis

Given a program point u, this analysis discovers the expressions whose results at u are same as the their previously computed values regardless of the execution path taken to reach u.

DEFINITION 2.3 *An expression $e \in \mathbb{E}\mathsf{xpr}$ is available at a program point u if all paths from* Start *to u contain a computation of e which is not followed by an assignment to any of its operands.*

The data flow equations which define available expressions analysis are:

$$In_n = \begin{cases} BI & n \text{ is } Start \text{ block} \\ \bigcap_{p \in pred(n)} Out_p & \text{otherwise} \end{cases} \tag{2.5}$$

$$Out_n = (In_n - Kill_n) \cup Gen_n \tag{2.6}$$

where In_n, Out_n, Gen_n, $Kill_n$, and BI are sets of expressions. Observe the use of $\cap$ to capture the "all paths" nature of data flow. This is different from liveness and reaching definitions analyses. However, similar to reaching definitions analysis, the direction of data flow is forward.

BI assumes that expressions involving local variables are not available at entry of *Start* since the local variables come into existence with function invocations.[‡] Gen_n contains downwards exposed expressions in n whereas $Kill_n$ contains all expressions whose operands are modified in n.

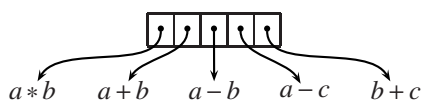
The availability information is useful in an optimization called *common subexpression elimination* in which computation of an expression is marked as redundant if the expression is available at that point. Let the set of expressions whose upwards exposed computations exist in block n be denoted by $AntGen_n$[§]. Let $Redundant_n$ denote expressions which can be eliminated in block n. Then,

$$Redundant_n = AntGen_n \cap In_n \tag{2.7}$$

Values of the previous computations are stored in a temporary variable and the redundant computations are replaced by that temporary variable. Most production compilers such as gcc perform common subexpression elimination.

Example 2.9

The program in Figure 2.1 contains expressions $(a*b)$, $(a+b)$, $(a-b)$, $(a-c)$, and $(b+c)$. We represent the set of expression by a bit vector; the position a bit indicates the expression which it represents as shown below.



Bit string 11111 represents the set $\{a*b, a+b, a-b, a-c, b+c\}$ whereas bit string 00000 represents $\emptyset$. The result of available expressions analysis has been shown below. Since this is an all paths analysis, the initial value at each

[‡]There could be exceptions to this in languages which allocate activation records in static area instead of stack e.g., FORTRAN IV.

[§]*AntGen* is the *Gen* set for Anticipability analysis described in Section 2.4.3. Here we use a different name to avoid confusion with *Gen* of the current analysis.

program point is the universal set (i.e., 11111).

Block	Local Information			Global Information				
				Iteration # 1		Changed values in iteration # 2		
	Gen_n	$Kill_n$	$AntGen_n$	In_n	Out_n	In_n	Out_n	$Redundant_n$
n_1	10001	11111	00000	00000	10001			00000
n_2	00010	11101	00010	10001	00010			00000
n_3	00000	00011	00001	10001	10000	10000		00000
n_4	10100	00011	10100	10000	10100			10000
n_5	01000	00000	01000	10000	11000			00000
n_6	00001	00000	00001	11000	11001			00000
n_7	01000	00000	01000	10000	11000			00000
n_8	00011	00000	00011	00000	00011			00000

Expression $(a*b)$ in n_4 is redundant. Its value can be stored in a temporary variable say t_0. Then the assignment $d = a*b$ in n_1 can be replaced by $d = t_0$ and the assignment $c = a*b$ in n_4 can be replaced by $c = t_0$.

If we had used 00000 as the initial value, expression $(a*b)$ would not have been available anywhere in the loops except at $Exit(n_4)$. Thus we would have missed the opportunity of eliminating the computation of $(a*b)$ in n_4. □

For a given expression e, available expressions analysis discovers a set of *availability paths*. Each availability path is a sequence of blocks $(b_1, b_2, \ldots, b_k)$ which is a prefix of some potential execution path starting at b_1 such that:

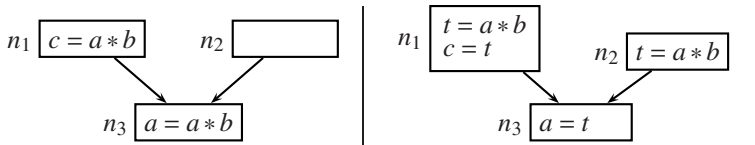
- b_1 contains a downwards exposed computation of e,
- b_k is either *End* or contains a computation of e, or an assignment to some operand of e,
- no block in the path contains a computation of e, or an assignment to any operand of e, and
- every path ending on b_k is an availability path for e.

Note that because of the last condition, we cannot talk about an availability path in isolation from other paths ending on a node—we must talk about a group of availability paths.

In terms of availability paths, common subexpression elimination in block n involves storing the value of redundant expression in a temporary at the start of every availability path terminating at n and replacing the computation of the expression in n by the temporary.

Example 2.10

Some availability paths for expression $(a*b)$ in our example program are: (n_1, n_3, n_4), $(n_1, n_3, n_5, n_6, n_7, n_3, n_4)$, and (n_4, n_7, n_3, n_4). □



(a) Partial Redundancy (b) Eliminating Partial Redundancy

FIGURE 2.4
Partial availability and partial redundancy.

2.4.2 Partially Available Expressions Analysis

An important variant of available expressions analysis relaxes the condition that an expression should be available along all paths—it is sufficient if the expression is available along some path.

If a block contains an upwards exposed computation of an expression and the expression is available at the entry of the block, then the upwards exposed computation is totally redundant. If the expression is partially available at the entry of the block, then the upwards exposed computation is partially redundant as illustrated in Figure 2.4. This information is used in partial redundancy elimination described in Section 2.4.4.

We need to make a simple change in available expressions analysis to discover partially available expressions: Data flow information should be merged using $\cup$ instead of $\cap$. This also means that the initial value is $\emptyset$ instead of the universal set.

Partially redundant computations in block n are defined by

$$ParRedund_n = AntGen_n \cap In_n \tag{2.8}$$

where $AntGen_n$ denotes the set of expressions whose upwards exposed computations exist in block n.

Example 2.11

The result of partially available expressions analysis on our example program has been shown below. Since the confluence operation is $\cup$, the initial value of In_i and Out_i for all i is 00000.

Block	Local Information			Global Information				
				Iteration # 1		Changed values in iteration # 2		
	Gen_n	$Kill_n$	$AntGen_n$	In_n	Out_n	In_n	Out_n	$ParRedund_n$
n_1	10001	11111	00000	00000	10001			00000
n_2	00010	11101	00010	10001	00010			00000
n_3	00000	00011	00001	10001	10000	11101	11100	00001
n_4	10100	00011	10100	10000	10100	11100	11100	10100
n_5	01000	00000	01000	10000	11000	11101	11101	01000
n_6	00001	00000	00001	11000	11001	11101	11101	00001
n_7	01000	00000	01000	11101	11101			01000
n_8	00011	00000	00011	11111	11111			00011

Observe that for every n, $ParRedund_n \supseteq Redundant_n$ suggesting partial redundancies subsume total redundancies. Also note that in our program, there are many partial redundancies which are not total. □

The paths discovered by partial available expressions analysis are a special case of the availability paths discovered by available expressions analysis: The last condition in the definition of availability paths does not apply to partial availability paths. Thus unlike availability paths, we can talk about individual partial availability paths.

2.4.3 Anticipable Expressions Analysis

Common subexpression elimination explained in Section 2.4.1 involves "in-place" transformation. As observed in the beginning of Section 2.4, some transformations involve inserting expressions at program points where they were not computed in the original program. Preserving the semantics of programs requires ensuring that a computation should not be inserted in a path along which the computation was not performed in the original program.

Example 2.12

Consider our running example of Figure 2.1. It is easy to see that expression $(a+b)$ is invariant in both the loops and it is desirable to move it out of the loops and place it at $Exit(n_1)$. However, the control flow path $n_1 \rightarrow n_2 \rightarrow n_8$ does not have any computation of the expression. Hence inserting the expression at $Exit(n_1)$ is not safe. □

The decision such as above can be arrived at by performing *anticipable expressions analysis* (also called *very busy expressions* analysis).

DEFINITION 2.4 *An expression $e \in \mathbb{E}\mathsf{xpr}$ is anticipable at a program*

Block	Local Information		Global Information			
			Iteration # 1		Changed values in iteration # 2	
	Gen_n	$Kill_n$	Out_n	In_n	Out_n	In_n
n_8	00011	00000	00000	00011		
n_7	01000	00000	00011	01011	00001	01001
n_6	00001	00000	01011	01011	01001	01001
n_5	01000	00000	01011	01011	01001	01001
n_4	10100	00011	01011	11100	01001	11100
n_3	00001	00011	01000	01001	01000	01001
n_2	00010	11101	00011	00010		
n_1	00000	11111	00000	00000		

FIGURE 2.5
Anticipable expressions analysis for Example 2.13.

point u if every path from u to End contains a computation of e which is not preceded by an assignment to any operand of e.

The data flow equations which define anticipable expressions analysis are:

$$In_n = (Out_n - Kill_n) \cup Gen_n \tag{2.9}$$

$$Out_n = \begin{cases} BI & n \text{ is } End \text{ block} \\ \bigcap_{s \in succ(n)} In_s & \text{otherwise} \end{cases} \tag{2.10}$$

where In_n, Out_n, Gen_n, $Kill_n$, and BI are sets of expressions. Similar to available expressions analysis, these equations use $\cap$ to capture the "all paths" nature of data flow. However, the data flow is backward similar to live variables analysis.

BI assumes that the expressions involving local variables are not anticipated at $Exit(End)$. Gen_n contains upwards exposed expressions in n whereas $Kill_n$ contains all expressions whose operands are modified in n.

Example 2.13

The result of anticipable expressions analysis on our example program has been shown in Figure 2.5. Since the confluence operation is $\cap$, the initial value of In_i and Out_i for all i is 11111. □

For a given expression e, anticipable expressions analysis discovers a set of *anticipability paths*. Each anticipability path is a sequence of blocks $(b_1, b_2, \ldots, b_k)$ which is a prefix of some potential execution path starting at b_1 such that:

- b_k contains an upwards exposed computation of e,

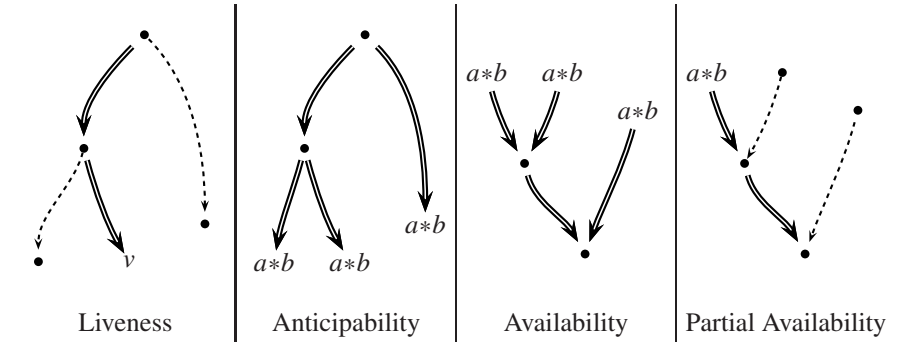


FIGURE 2.6
Data flow paths discovered by data flow analysis (shown by double lines).

- b_1 is either *Start* or contains a computation of e, or an assignment to some operand of e,
- no block in the path contains a computation of e, or an assignment to any operand of e, and
- every path starting at b_1 is an anticipability path.

Similar to availability paths, we talk about a group of anticipability paths rather than a single anticipability path.

Example 2.14
Some anticipability paths for expression $(a+b)$ in our example program are: (n_5,n_6,n_5), (n_5,n_6,n_7), (n_3,n_4,n_7), and (n_3,n_5). Note that $(a+b)$ is not anticipable at *Exit*(n_1). □

2.4.4 Classical Partial Redundancy Elimination

This section presents the classical approach to *partial redundancy elimination* (PRE) which involves a *bidirectional* formulation of data flows. This section also describes its limitations and shows how they are overcome by some of its variants.

The basic principle of PRE has been illustrated in Figure 2.4. It can be viewed as an instance of *code hoisting* along a *hosting path*. This hoisting subsumes loop invariant movement and common subexpression elimination.

Example 2.15
In Figure 2.4 the hoisting path is (n_2,n_3); path (n_1,n_3) is an availability path. In Figure 2.7, expression $b*c$ is loop invariant and is partially available due the availability path along the back edge. This is a special case of partial redundancy and can be eliminated along the hoisting path (n_1,n_2). □

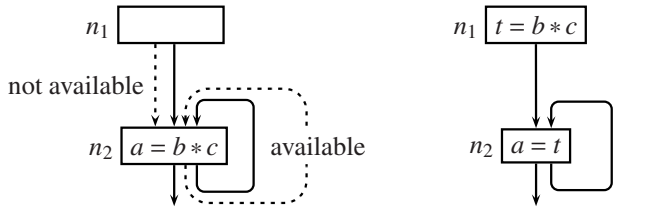


FIGURE 2.7
Loop invariant is a special case of PRE.

Hoisting Path of an Expression

Informally, the safety and desirability of hoisting an expression are defined as follows: An expression can be safely hoisted to a program point u if it is anticipable at u. It should be hoisted to ancestors of u if it is partially available at u.

For an expression e, a hoisting path is a maximal sequence of blocks $(b_1, b_2, \ldots, b_k)$ which is a prefix of a potential execution path starting at b_1 such that:

- b_k contains an upwards exposed computation of e,
- e is anticipable and partially available at $\mathit{Entry}(b_i)$ and $\mathit{Exit}(b_i)$ of each block b_i (other than b_1 and b_k), and at $\mathit{Entry}(b_k)$,
- e is not available at $\mathit{Exit}(b_1)$, or can be hoisted to $\mathit{Entry}(b_1)$, and
- no block in the path contains a computation of e, or an assignment to any operand of e.

A key design idea in defining a hoisting path is that an expression is hoisted to $\mathit{Entry}(n)$ only if it can be hoisted out of n into its predecessors. This means that if an expression has to be inserted at the start of a hoisting path, it is inserted at the exit of the first block rather than at its entry. The conditions for hoisting an expression into and out of a block are defined as follows:

- *Safety of hoisting to* $\mathit{Exit}(n)$.

  An expression e should be hoisted to $\mathit{Exit}(n)$ only if

  (S.1) it can be hoisted to $\mathit{Entry}(s)$ for every successor s of n.

  This is captured by the equation:

$$\mathit{Out}_n = \begin{cases} \mathit{BI} & n \text{ is } \mathit{End} \text{ block} \\ \bigcap_{s \in \mathit{succ}(n)} \mathit{In}_s & \text{otherwise} \end{cases} \tag{2.11}$$

- *Safety of hoisting to* $\mathit{Entry}(n)$.

  An expression e should be hoisted to $\mathit{Entry}(n)$ only if

  (S.2) n contains an upwards exposed computation of e, or

  (S.3) e can be hoisted to $\mathit{Exit}(n)$ and n does not contain an assignment to any operand of e.

  Condition S.2 is satisfied by Gen_n of Anticipability analysis which is denoted by AntGen_n to distinguish it from Gen_n of other analyses. Condition S.3 is satisfied by the term $(\mathit{Out}_n - \mathit{Kill}_n)$.[¶] Thus the safety of placement at $\mathit{Entry}(n)$ is captured by the term

$$\mathit{In}_n \subseteq (\mathit{AntGen}_n \cup (\mathit{Out}_n - \mathit{Kill}_n)) \tag{2.12}$$

- *Desirability of hoisting*.

  By design, an expression e should be hoisted to $\mathit{Entry}(n)$ only if it can be hoisted out of it into a predecessor of n. If it can be hoisted into some predecessor but not all predecessors then safety requires that one evaluation of the expression should be made in n and then it is not profitable to hoist it into any predecessor.

  Further, if it is not partially available, hoisting it does not eliminate any partial redundancy. Hence an expression e should be hoisted to $\mathit{Entry}(n)$ only if

  (D.1) e is partially available at $\mathit{Entry}(n)$, and

  (D.2) for each predecessor p of n,

  (D.2.a) e can be hoisted to $\mathit{Exit}(p)$, or

  (D.2.b) e is available at $\mathit{Exit}(p)$ (and hence need not be inserted at $\mathit{Exit}(n)$).

  Condition D.1 is captured by the term

$$\mathit{In}_n \subseteq \mathit{PavIn}_n \tag{2.13}$$

  Condition D.2 is captured by the term

$$\mathit{In}_n \subseteq \bigcap_{p \in \mathit{pred}(n)} \left(\mathit{Out}_p \cup \mathit{AvOut}_p\right) \tag{2.14}$$

Combining Conditions (2.12), (2.13), and (2.14) results in Equation (2.15) below which defines In_n. Out_n is defined by Equation (2.11).

[¶]Note that Kill_n is same for all analyses involving expressions: Available expressions analysis, partially available expressions analysis, anticipable expressions analysis, and PRE.

Block	Local information		Global Information							
			Constant information		Iteration # 1		Changes in iteration # 2		Changes in iteration # 3	
	Gen_n	$Kill_n$	$PavIn_n$	$AvOut_n$	Out_n	In_n	Out_n	In_n	Out_n	In_n
n_8	00011	00000	11111	00011	00000	00011				00001
n_7	01000	00000	11101	11000	00011	01001	00001			
n_6	00001	00000	11101	11001	01001	01001			01000	
n_5	01000	00000	11101	11000	01001	01001		01000		
n_4	10100	00011	11100	10100	01001	11100		11000		
n_3	00001	00011	11101	10000	01000	01001		00001		
n_2	00010	11101	10001	00010	00011	00000			00001	
n_1	00000	11111	00000	10001	00000	00000				

FIGURE 2.8
Partial redundancy elimination.

$$In_n = PavIn_n \cap (AntGen_n \cup (Out_n - Kill_n)) \cap \bigcap_{p \in pred(n)} \left(Out_p \cup AvOut_p\right) \tag{2.15}$$

Observe that if we drop the desirability terms from Equations (2.11) and (2.15), they reduce to the anticipability equations (Equations 2.9 and 2.10).

Example 2.16
We illustrate the conditions defining hosting criteria with the help of expression $(a+b)$ in our running example of Figure 2.1. Since this expression is not computed along path (n_1, n_2, n_8), it is not anticipable at the exit of n_1. Hence inserting it at the exit of n_1 violates safety. However, it is anticipable at the exit of n_3 and inserting it there is safe. Feasibility condition S.2 for $(a+b)$ is satisfied by block n_7 and n_5 whereas condition S.3 is satisfied by block n_4. Condition D.1 is satisfied by blocks n_4, n_5, n_6, and n_7. Condition D.2.a is satisfied by n_3 whereas Condition D.2.b is satisfied by n_7. □

Example 2.17
The computation of PRE data flow properties of our running example is shown in Figure 2.8. Since the confluence operation is ∩, the initial value of In_i and Out_i for all i is 11111.

Figure 2.9 shows the hoisting paths in our example. Observe that there is no hoisting path for expression $(a*b)$ since it is totally redundant and

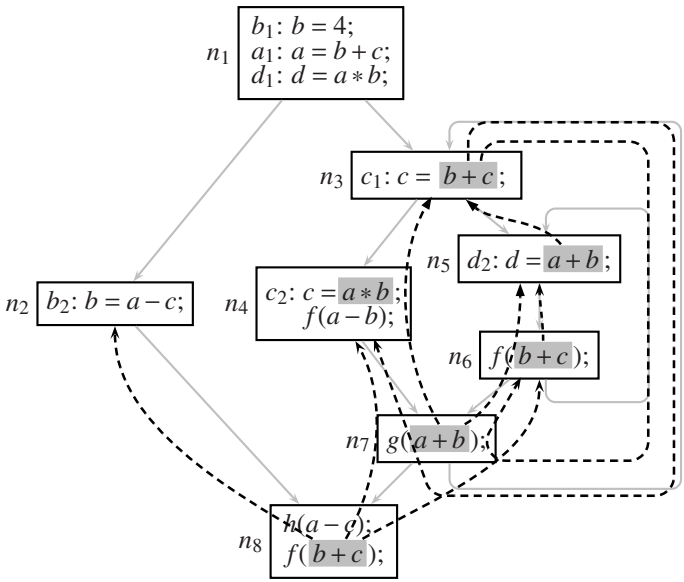


FIGURE 2.9
Hoisting paths in PRE of the running example.

need not be inserted anywhere. For expression $(a+b)$ there are three hoisting paths: (n_3,n_4,n_7), (n_3,n_5) and (n_5,n_6,n_7). Since the last path also happens to be an availability path, there is no need to insert the expression in n_5. Expression $(b+c)$ has the following hoisting paths: (n_2,n_8), (n_6,n_7,n_8), (n_6,n_7,n_3), (n_4,n_7,n_8), (n_4,n_7,n_3), and (n_5,n_6). Observe that there is no hoisting path for expressions $(a-b)$ and $(a-c)$.

Also observe the need of the third iteration for suppressing the hoisting of expressions $(a-c)$, and $(b+c)$. The initial values of the bits corresponding to these expressions is 1 in In/Out values. Expression $(a-c)$ cannot be hoisted out of the outer loop because it is neither partially available anywhere in the loop nor is it invariant in the loop due to assignment to c. Thus the bit corresponding to this expression becomes 0 in In_{n_3} in the first iteration. The fact that it cannot be placed at $Exit(n_7)$ because of this reason, can be discovered only in the second iteration when its bit in Out_{n_7} becomes 0. Its hoisting out of n_8 is suppressed in the third iteration when its bit in In_{n_8} becomes 0 in the third iteration.

Expression $(b+c)$ is not anticipated at $Exit(n_3)$ and hence its bit in Out_{n_3} becomes 0 in the first iteration. Setting the corresponding bit in In_{n_5} to 0 requires the second iteration. Its placement at $Exit(n_6)$ is suppressed in the third iteration. □

Transformation Using Hoisting Path

Having identified hoisting paths, and complementary availability paths for an expression e, the following transformations need to be performed by creating a new temporary variable t:

- *At the Start of a Hoisting or an Availability Path.*

  Insert an assignment $t = e$, just before the computation of e. Replace the original computation of e by t at the start of an availability path.

  Note that there is no need to detect an availability path explicitly. All downwards exposed computations of e can be safely assumed to start an availability path. Thus the main task is to identify the start of that hoistability path where the expression has to be inserted. The necessary conditions for block n to start a hoistability path are:

  – It should be possible to hoist the expression to $Exit(n)$, and
  – It should not be possible to hoist the expression at $Entry(n)$, or some operand of the expression should be modified in n.

  These conditions are captured by the following:

$$Insert_n = Out_n \cap (\neg In_n \cup Kill_n) \tag{2.16}$$

- *At the End of a Hoisting Path.*

  Replace the original computation of e by t.

  Identifying this is easy: It should be possible to hoist e to $Entry(n)$ and there should be an upwards exposed computation of e in n. These conditions are captured by the following:

$$Replace_n = In_n \cap AntGen_n \tag{2.17}$$

Example 2.18

In our running example, the data flow information which enables the transformation is:

Block	Local Information		Global Information			
			Iteration # 3			
	$AntGen_n$	$Kill_n$	In_n	Out_n	$Replace_n$	$Insert_n$
n_1	00000	11111	00000	00000	00000	00000
n_2	00010	11101	00000	00001	00000	00001
n_3	00001	00011	00001	01000	00001	01000
n_4	10100	00011	11000	01001	10000	00001
n_5	01000	00000	01000	01001	01000	00001
n_6	00001	00000	01001	01000	00001	00000
n_7	01000	00000	01001	00001	01000	00000
n_8	00011	00000	00001	00000	00001	00000

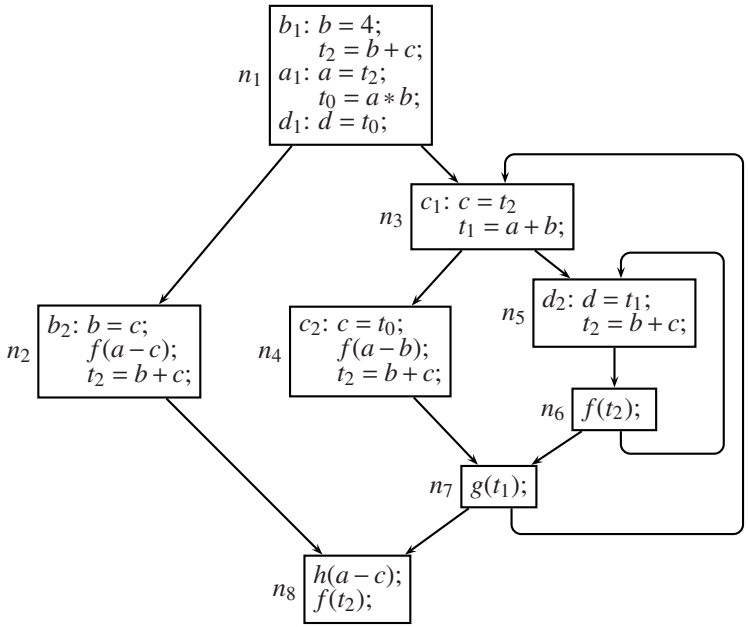


FIGURE 2.10
Optimized program after PRE.

Figure 2.10 shows the optimized program after performing PRE. □

An important property of this transformation is that on any path in the program, the number of computations in the optimized program is guaranteed to not exceed the number of computations in the original program.

Limitations of Partial Redundancy Elimination

PRE combines many flows: Partial availability is a forward flow with union as the confluence, total availability is a forward flow with intersection as the confluence, and anticipability is backward flow with intersection as the confluence. Combining these flows results in conservative approximations. Thus in some cases, partial redundancies cannot be eliminated; in some cases, elimination causes some undesirable side effects; and in most cases, efficiency of performing analysis is a matter of concern.

Example 2.19
We illustrate the above limitations with our running example.

- *Inability to eliminate all partial redundancies*.

It is clear from the optimized program in Figure 2.10 that expression $(a+b)$ has been moved out of the inner loop but cannot be moved out of the outer loop. Similarly, expression $(a-c)$ in n_8 and expression $(a-b)$ in n_4 are not eliminated in spite of being partially redundant.

- *Increase in lifetimes of values of expressions, and hence increase in register pressure.*

  Expression $(b+c)$ is merely hoisted from block n_6 to n_5 without reducing the number of computations of $(b+c)$ in that path. Such redundant hoisting increases register pressure since the result of $(b+c)$ must be kept in a register for a longer duration.

- *Concern about efficiency of performing PRE.*

  In_n/Out_n computation for PRE requires three iterations. For liveness analysis this computation converged in one iteration whereas for all other analyses discussed in this chapter, it converged in two iterations.

□

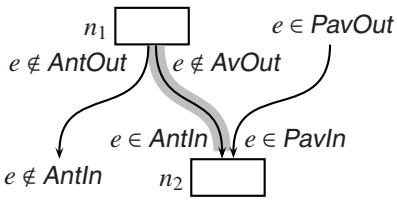
PRE is blocked by a combination of data flows in the presence of the following two structures in CFGs: *Critical edges*, and *critical nodes*. A critical edge is an edge that runs from a *fork node* (i.e., a node with more than one successor) to a *join node* (i.e., a node with more than one predecessor). A critical node is a fork node which has multiple paths reaching it.

Figure 2.11 illustrates the effect of critical edges and nodes on hoisting. Edge $n_1 \rightarrow n_2$ in Figure 2.11(a) is a critical edge whereas node n_2 in Figure 2.11(b) is a critical node. In each case, expression e is a possible candidate for hoisting from $Entry(n_2)$ to $Exit(n_1)$ but is not anticipated at $Exit(n_1)$. In the case of a critical edge, e is partially available at $Entry(n_2)$ due to another predecessor of n_2 whereas in the case of a critical node, e is partially available at $Entry(n_2)$ due to n_1.

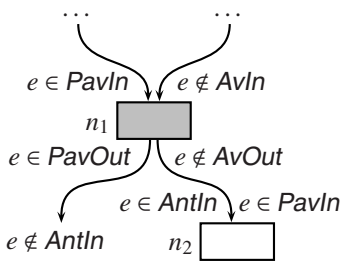
Observe that if e were available at $Exit(n_1)$, the critical edge or critical node would not have any adverse effect because there would be no need of hoisting e out of n_2; it would be totally redundant in n_2. Alternatively, if e were anticipated at $Exit(n_1)$, then e would be hoisted out of n_2—in the case of critical edge, it would be placed in n_1 and in the case of critical node, it would be hoisted further out of n_1.

Example 2.20

Edges $n_1 \rightarrow n_3$, $n_3 \rightarrow n_5$, $n_6 \rightarrow n_5$, $n_6 \rightarrow n_7$, and $n_7 \rightarrow n_8$ in our running example are critical edges. Nodes n_3, n_6, and n_7 are critical nodes. Edge $n_1 \rightarrow n_3$ blocks hoisting expression $(a+b)$ from n_3 to n_1, $n_3 \rightarrow n_5$, blocks hoisting expression $(b+c)$ from n_5 to n_3, and $n_7 \rightarrow n_8$, blocks hoisting expression $(a-c)$ from n_8 to n_7. Critical node n_3 blocks hoisting expression $(a-b)$ from n_4 to n_3. □



(a) Edge $n_1 \rightarrow n_2$ is a critical edge



(b) Block n_1 is a critical node

FIGURE 2.11
Critical edges and critical nodes block PRE. Expression e cannot be hoisted out of n_2 into the exit of n_1.

Handling Critical Edges

A careful examination of the effect of critical edges reveals that this limitation arises due to the fact that the data flow value represented by In_n plays a dual role: It captures the property of safety of placement (Constraint 2.12) as well as the desirability of placement (Constraints 2.13 and 2.14).

In Figure 2.11(a), the bit corresponding to e becomes 0 in Out_{n_1} due to safety constraint ($e \notin AntIn$ of the successor on the left). This makes the corresponding bit 0 in In_{n_2} due to desirability constraint which in turn make the corresponding bit 0 in Out of the right predecessor of n_2. If a new node n_{12} is inserted along edge $n_1 \rightarrow n_2$ in Figure 2.11, the repercussions of the desirability constraint are restricted to $In_{n_{12}}$ since it does not have any predecessor other than n_1. Further, since $e \in AntOut_{n_{12}}$ even if $e \notin AntOut_{n_1}$, it becomes possible to hoist e out of n_2 into the newly created node n_{12}. Note that this hoisting is not redundant because e is not partially available in n_{12}.

Example 2.21

Figure 2.12 shows PRE after splitting critical edges in our running example. This allows hoisting $(b+c)$ out of the inner loop and $(a+b)$ out of the outer loop. Besides, $(a-c)$ is hoisted out of n_8. Note that this has no effect on the placement of loop invariant expression $(a-b)$. □

Edge splitting has a pleasant side effect of increasing the efficiency of analysis. Intuitively, an all-path analysis can be seen as optimistically assuming bits to be 1 in the CFG and then resetting them to 0 due to the influence of corresponding bits at neighbouring program point. Thus analysis involves propagating 0 in the graph along arbitrarily long paths. The corresponding view for any-path analyses assumes the bits to be 0 initially and then propagates 1 in the graph. Edge splitting prunes this propagation for PRE because it prohibits the repercussions of the desirability

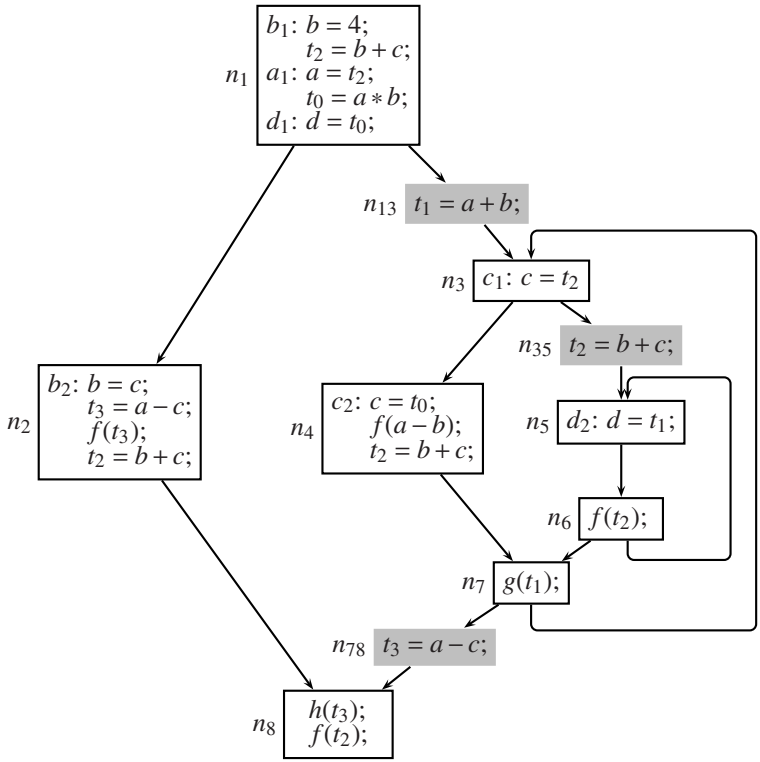


FIGURE 2.12
PRE after splitting critical edges. Among the new blocks, we have retained only non-empty blocks.

constraints: Propagation of 0 from Out_{n_1} to In_{n_2} is truncated at $In_{n_{12}}$: $Out_{n_{12}}$ cannot become 0 even if $In_{n_{12}}$ becomes 0 and hence In_{n_2} remains 1.

A variant of edge-splitting is *edge-placement* which essentially achieves the same effect except that instead of splitting critical edges a-priori, the approach is to change data flow analysis to discover the edges along which expressions should be placed. Then the required edges are split and expressions placed in the new node. Thus this can be seen as edge-splitting on demand.

Handling Critical Nodes

Edge splitting does not help in the case of critical nodes even if we decide to split the out edges of critical nodes regardless of whether these edges are critical or not. If we split edge $n_1 \rightarrow n_2$ in Figure 2.11(b), it would be possible to hoist e from n_2 into the new node but it will continue to be partially redundant. What is required is a transformation which will enable hoisting e out of n_1 to those ancestors m of n_1 such

that $e \notin PavOut_m$.

A transformation which achieves this involves duplicating the critical node, and along with it some other nodes such that in one copy of these nodes, the expression is available whereas in the other copy, the expression is not available. The region in which the expression is available does not need hoisting since the expression becomes totally redundant. The region in which the expression is not available can be optimized by edge-splitting.

Example 2.22

Figure 2.13 shows code duplication involving a critical node which blocks hoisting. The basic idea is to identify *code motion preventing* (*CMP*) region which is a set of nodes characterized by the following:

$$n \in CMP(e) \Leftrightarrow e \in (PantOut_n \cap PavOut_n \cap PantIn_n \cap PavIn_n)$$

For our running example,

$$CMP(a-b) = \{n_3, n_{35}, n_5, n_6, n_7\}$$

A critical node is that node in *CMP* where the expression is not anticipated along one set of out edges and is anticipated along the other set of edges. It is this node which blocks the hoisting of expressions into the region. In our case n_3 is the critical node.

The transformation involves duplicating each *CMP* region such that for one copy the expression is available and for the other copy it is not available. This involves retaining the availability edge in one copy and not in the other. In our example, the expression is available in the nodes with dashed labels through edge $n_4 \rightarrow n_7'$. Note that the other copy does not have the corresponding edge.

There are two copies of the critical node and since their out edges are retained, the out edges along which the expression is anticipated become critical edges because these edges go to a unique node out of *CMP* region. Splitting these edges facilitates hoisting into a new successor of the critical node. □

2.4.5 Lazy Code Motion

This section presents an alternative approach to PRE which minimizes lifetimes by separating safety and desirability constraints. It allows placement of expressions at the entry of a block and incorporates the desirability through separate analyses. These analyses employ a stronger notion of desirability to minimize the lifetimes of temporary variables. Unlike the classical formulation of PRE, all analyses involved in this approach are unidirectional.

This approach is called *lazy* code motion because it performs as little code motion as possible suppressing it where it does not result in profitable placement. The main steps of this approach are:

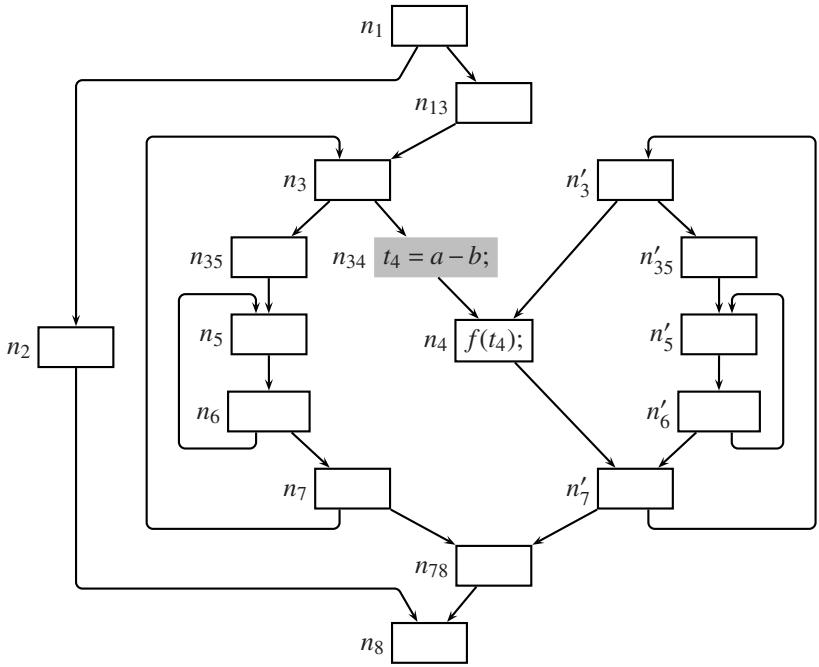


FIGURE 2.13
PRE after duplicating a code motion preventing region rooted at a critical node (n_3). Duplicate copies have dashed labels. Additional edge-splitting is required for the technique to work.

1. Splitting critical edges. Observe that in classical PRE, edge-splitting only enhances the effectiveness of redundancy elimination but is not required for its correctness. However, it is crucial for the correctness of this approach.

2. Discovering a region of safe placement of expressions.

   This involves anticipability analysis (Section 2.4.3) for discovering hoisting paths where the expressions could be placed anywhere to make the original computations redundant. A safe region of placement for an expression e is the set of program points where the expression is anticipable. Equations (2.10) and (2.9) are used to discover the region of safe placement.

3. Discovering entry points of region of safe placements.

   Entry points of a region of safe placement are the points in the region where the expression can be inserted in order to make the original computations in the region totally redundant.

   Entry points are the earliest points and form the smallest such set where expressions can be placed. These points are discovered by combining the results

of availability analysis (Section 2.4.1) with the result of anticipability analysis. Placing expressions at earliest points amounts to hoisting them from their original points of computation.

We use the prefix *Ant* and *Av* to denote data flow values in anticipability and availability. Let $EarliestIn_n$ and $EarliestOut_n$ denote the entry points. Then,

$$EarliestIn_n = AntIn_n \cap \left(\bigcap_{p \in pred(n)} \neg \left(AvOut_p \cup AntOut_p \right) \right) \tag{2.18}$$

$$EarliestOut_n = (AntOut_n \cup AvGen_n) \cap Kill_n \tag{2.19}$$

Availability is computed using Equations (2.5) and (2.6).

Edge splitting ensures that *AntOut* of all predecessors of a node is identical. Thus the earliest points are

- *Entry*(n) of block n where it is safe to insert the expression, cannot be hoisted into any predecessor, and is not available along any predecessor.
- *Exit*(n) of block n that contains a downwards exposed computation of the expression such that it cannot be hoisted to *Entry*(n) due the presence of an assignment to some operand of the expression.

Note that it is possible that both *Exit*(n) and *Entry*(n) are earliest points for some expression e. This happens when e is anticipable at *Exit*(n) and n contains both downwards and upwards exposed computations of e and an assignment to an operand of e.

4. Discovering the latest points of region of safe placements.

   In order to minimize the lifetimes of temporary variables, the expressions placed at earliest points can be *sunk* to later points along the control flow in the region of safe placements. This analysis is an all-paths forward analysis:

$$SinkIn_n = EarliestIn_n \cup \begin{cases} \emptyset & n \text{ is } Start \text{ block} \\ \bigcap_{p \in pred(n)} (SinkOut_p - AvGen_p) & \text{otherwise} \end{cases} \tag{2.20}$$

$$SinkOut_n = EarliestOut_n \cup (SinkIn_n - AntGen_n) \tag{2.21}$$

Sinking begins at the earliest points of placements and discovered path along which the expressions can be sunk. The latest placement points of expressions are the end points of these paths and are defined by:

$$LatestIn_n = SinkIn_n \cap AntGen_n \tag{2.22}$$

$$LatestOut_n = SinkOut_n \cap \left(AvGen_n \cup \left(\bigcup_{s \in succ(n)} \neg SinkIn_s \right) \right) \tag{2.23}$$

The above equations use $AntGen_n$ and $AvGen_n$ to ensure that sinking is applicable only to new placements—an original computation in a block cannot be sunk.

5. Discovering those expressions whose values need not be preserved in temporary variables.

   When the expressions are sunk to their latest points, some computations might have only a local use within a block. Such computations need not be preserved in a temporary variable. Discovering the variables whose values need not be preserved is a simple variation of deadness analysis.

$$NoUseIn_n = EarliestIn_n \cup NoUseOut_n \tag{2.24}$$

$$NoUseOut_n = \bigcap_{s \in succ(n)} (EarliestIn_s \cup (NoUseIn_s - AntGen_s)) \tag{2.25}$$

6. Inserting assignments to temporary variables at insertion points and replacing original expressions by temporary variables.

   The values of expressions should be stored in temporary variables at the latest computation points provided the values have some use in future. This is identified by

$$InsertIn_n = LatestIn_n - NoUseIn_n \tag{2.26}$$

$$InsertOut_n = LatestOut_n - NoUseOut_n \tag{2.27}$$

   The original computations which should be replaced by temporary variables are defined by the following:

$$ReplaceIn_n = AntGen_n - (LatestIn_n \cap NoUseIn_n) \tag{2.28}$$

$$ReplaceOut_n = AvGen_n - (LatestOut_n \cap NoUseOut_n) \tag{2.29}$$

Example 2.23

Consider our running example after edge splitting: Edge $n_1 \rightarrow n_3$ in Figure 2.1 is split to create node n_{13}, edge $n_3 \rightarrow n_5$ is split to create node n_{35}, and edge

Block		Availability			Anticipability			Earliest Placement	
	Kill	*AvGen*	*AvIn*	*AvOut*	*AntGen*	*AntIn*	*AntOut*	*EarliestIn*	*EarliestOut*
n_1	11111	10001	00000	10001	00000	00000	00000	00000	10001
n_2	11101	00010	10001	00010	00010	00010	00011	00010	00001
n_{13}	00000	00000	10001	10001	00000	01001	01001	01000	00000
n_3	00011	00000	10000	10000	00001	01001	01000	00000	00000
n_{35}	00000	00000	10000	10000	00000	01001	01001	00001	00000
n_4	00011	10100	10000	10100	10100	11100	01001	00100	00001
n_5	00000	01000	10000	11000	01000	01001	01001	00000	00000
n_6	00000	00001	11000	11001	00001	01001	01001	00000	00000
n_7	00000	01000	10000	11000	01000	01001	00001	00000	00000
n_{78}	00000	00000	11000	11000	00000	00011	00011	00010	00000
n_8	00000	00011	00000	00011	00011	00011	00000	00000	00000

FIGURE 2.14
Early placement points for lazy code motion.

$n_7 \rightarrow n_8$ is split to create node n_{78}. The early placement points are shown in Figure 2.14. As shown in Figure 2.15, the earliest placement points also happen to be the latest points for this particular example. This is because of the early placement opportunities created by edge splitting. The optimized program after lazy code motion is identical to that shown in Figure 2.12. □

Example 2.24

If we do not split critical edges in our running example, lazy code motion replaces all occurrence of expressions $(a-c)$ and $(a+b)$ by temporaries. However, the value of $(a-c)$ is stored in its temporary only in n_2 and hence it is not available along the paths reaching n_8 from n_7. The value of $(a+b)$ is not stored in its temporary anywhere. □

2.5 Combined *May-Must* Analyses

Classical PRE requires both total availability and partial availability analysis. Such a need is not uncommon and often both any-path and all-paths variants of information are required. The all-path variant of data flow information is also called *must* information. Analogously, the any-path variant of data flow information is called *may*

Block	*SinkIn*	*SinkOut*	*LatestIn*	*LatestOut*	*NoUseIn*	*NoUseOut*
n_1	00000	10001	00000	10001	11101	01100
n_2	00010	00001	00010	00001	11101	11100
n_{13}	01000	01000	00000	01000	00100	00100
n_3	00000	00000	00000	00000	00100	00100
n_{35}	00001	00001	00000	00001	00100	00100
n_4	00100	00001	00100	00001	00101	00100
n_5	00000	00000	00000	00000	00100	00100
n_6	00000	00000	00000	00000	00100	00100
n_7	00000	00000	00000	00000	00100	00100
n_{78}	00010	00010	00000	00010	11100	11100
n_8	00000	00000	00000	00000	11111	11111

FIGURE 2.15
Latest placement points for lazy code motion.

information. It is possible to define a single analysis which discovers both *may* and *must* information. We explain this with the help of availability analysis.

Defining *may-must* availability analysis requires us to define four possible values which can be associated an expression at any program point. For an expression e, the value *unknown* at a program point u indicates that sufficient information is not available at u; the value *must* indicates that e is available along all paths reaching u; the value *may* indicates that e is available along some but not along all paths reaching u; and the value *no* indicates that e is not available along any path reaching u. We view them as *degrees of certainty*. We define a new confluence operation which combines the degree of certainties of a given expression e as shown in Figure 2.16.

These values can be represented using 2 bits. If we represent *unknown* by 11, *must* by 10, *no* by 01, and *may* by 00, then $\sqcap$ can be implemented using simple bitwise AND. An alternative representation is to swap the bit strings for *unknown* and *may* and use bitwise OR for $\sqcap$.

The data flow information is defined in terms of sets of pairs $\langle e, d_e \rangle$ where d_e is the degree of certainty of expression e. The local data flow information is defined as follows:

$$\begin{aligned} \mathit{Kill}_n &= \{\langle e, d\rangle \mid e \in (\mathit{AvGen}_n \cup \mathit{AvKill}_n), d \in \{\mathit{may}, \mathit{must}, \mathit{no}, \mathit{unknown}\}\} \\ \mathit{Gen}_n &= \{\langle e, \mathit{must}\rangle \mid e \in \mathit{AvGen}_n\} \cup \{\langle e, \mathit{no}\rangle \mid e \in \mathit{AvKill}_n\} \end{aligned}$$

where AvGen_n and AvKill_n represent Gen_n and Kill_n for availability (or partial availability) analysis.

Observe that when an expression e is in AvGen_n or AvKill_n, it belongs to both Gen_n as well as Kill_n. This is because the local effect of block n may change the degree of certainty of e. Effectively, the pairs are neither removed nor added to in In_n and Out_n—only the degrees of certainties change. In other words, these sets have the

$\sqcap$	$\langle e, unknown\rangle$	$\langle e, must\rangle$	$\langle e, no\rangle$	$\langle e, may\rangle$
$\langle e, unknown\rangle$	$\langle e, unknown\rangle$	$\langle e, must\rangle$	$\langle e, no\rangle$	$\langle e, may\rangle$
$\langle e, must\rangle$	$\langle e, must\rangle$	$\langle e, must\rangle$	$\langle e, may\rangle$	$\langle e, may\rangle$
$\langle e, no\rangle$	$\langle e, no\rangle$	$\langle e, may\rangle$	$\langle e, no\rangle$	$\langle e, may\rangle$
$\langle e, may\rangle$	$\langle e, may\rangle$	$\langle e, may\rangle$	$\langle e, may\rangle$	$\langle e, may\rangle$

FIGURE 2.16
Confluence operation for combined *may* and *must* analysis.

same size at each program point. This is different from other bit vector frameworks which we have seen in this chapter,

Since the un-availability of an expression e is reflected by recording its degree of certainty as *no* instead of removing it from the set, $Kill_n$ does not imply that e ceases to be available; it captures the fact that the data flow information of e is killed.

The data flow equations are defined in the usual manner. The confluence $\sqcap$ defined over pairs $\langle e, d_e\rangle$ is lifted to the sets by applying it to pairs of the same expression.

$$In_n = \begin{cases} \{\langle e, no\rangle \mid e \in \mathbb{E}\text{xpr}\} & n \text{ is } Start \\ \prod\limits_{p \in pred(n)} Out_p & \text{otherwise} \end{cases} \tag{2.30}$$

$$Out_n = (In_n - Kill_n) \cup Gen_n \tag{2.31}$$

Example 2.25

For brevity, we represent the sets of pairs $\langle e, d_e\rangle$ in terms of vectors of d_e such that there is a positional correspondence between e and d_e. We retain the order of expressions as described in Example 2.9 except that now there are two bits for every expression instead of a single bit. The boundary information BI is $\langle no, no, no, no, no\rangle$ and the initial value of In_n and Out_n for all n is the tuple $\langle unknown, unknown, unknown, unknown, unknown\rangle$. With our first choice of representation, these values are represented by $\langle 01, 01, 01, 01, 01\rangle$ and $\langle 11, 11, 11, 11, 11\rangle$ respectively.

The data flow values are presented in Figure 2.17. Note that this information is same as availability and partial availability information computed in Example 2.9 and 2.11 except that partial availability includes total availability whereas *may* and *must* availabilities are mutually exclusive. ∎

An efficient implementation of the computation of Out_n is as follows:

$$Out_n = \{\langle e, \widehat{f_n}(e, X)\rangle \mid e \in \mathbb{E}\text{xpr}\} \tag{2.32}$$

where $\widehat{f_n}(e, X)$ represents the local effect of a block on the availability of expression e. The actual implementation of $\widehat{f_n}$ in terms of bit vector operations depends on

Block	Iteration #1		Iteration #2	
	In_n	Out_n	In_n	Out_n
n_1	⟨01,01,01,01,01⟩	⟨10,01,01,01,10⟩		
n_2	⟨10,01,01,01,10⟩	⟨01,01,01,10,01⟩		
n_3	⟨10,01,01,01,10⟩	⟨10,01,01,01,01⟩	⟨10,00,00,01,00⟩	⟨10,00,00,01,01⟩
n_4	⟨10,01,01,01,01⟩	⟨10,01,10,01,01⟩	⟨10,00,01,01,01⟩	⟨10,00,10,01,01⟩
n_5	⟨10,01,01,01,01⟩	⟨10,10,01,01,01⟩	⟨10,00,00,01,01⟩	⟨10,10,00,01,01⟩
n_6	⟨10,10,01,01,01⟩	⟨10,10,01,01,10⟩	⟨10,10,00,01,01⟩	⟨10,10,00,01,10⟩
n_7	⟨10,00,00,01,00⟩	⟨10,10,00,01,00⟩		
n_8	⟨00,10,00,00,00⟩	⟨00,10,00,10,10⟩		

FIGURE 2.17
Combined *may* and *must* availability analysis.

the choice of representation for the degrees of certainty. Assuming that we use the representation *unknown* ≡ 11, *must* ≡ 10, *no* ≡ 01, and *may* ≡ 00, and use bitwise AND as ⊓, $\widehat{f_n}$ can be implemented as follows:

$$\widehat{f_n}(e,X) = A_e + B_e \cdot d_e \tag{2.33}$$

where $\langle e, d_e \rangle \in X$, "+" denotes bitwise OR and "·" denotes bitwise AND. The values of A_e and B_e are governed by local information:

Local Information of e		A_e	B_e
$e \in AvGen_n$	$e \in Kill_n$	10	00
$e \in AvGen_n$	$e \notin Kill_n$	10	00
$e \notin AvGen_n$	$e \in Kill_n$	01	00
$e \notin AvGen_n$	$e \notin Kill_n$	00	11

2.6 Summary and Concluding Remarks

It is clear from the data flow frameworks presented in this chapter that data flow equations have a common form which can be customized for each analysis. The customization of this common form involves specifying the direction of flow, the confluence operation, and the flow functions which are defined in terms of Gen_n and $Kill_n$ components.

All flow functions in this chapter can be implemented using the bitwise operations AND and OR (or set operations ∩ and ∪). There are two important points associated with this observation:

- $Kill_n$ used in the operation $X - Kill_n$ is a constant value. Thus set complement (or bitwise NOT) is applied only to constant value. This computation can be performed once and the desired operation can be applied during the data flow analysis by $X \cap \neg Kill_n$.

- Gen_n and $Kill_n$ do not depend on In_n and Out_n and are purely local effects. Since Gen_n and $Kill_n$ are constant values, In_n and Out_n can be computed unconditionally without examining the operands.

In summary, in bit vector frameworks, the data flow information can be represented and computed using aggregate operations on bits; there is no need to examine the bits individually. Although the data flow value of an entity in common bit vector frameworks is a boolean value and hence can be represented by a single bit, this is neither necessary nor sufficient for a framework to qualify as a bit vector framework. For example, the combined *may-must* availability analysis described in Section 2.5 requires two bits but is a bit vector framework. Chapter 4 presents faint variables analysis in which data flow value is boolean and hence can be represented using a single bit. However, it is not a bit vector framework.

Subsequent chapters relax both these constraints and describe frameworks which capture more powerful semantics.

2.7 Bibliographic Notes

Bit vector frameworks are some of the oldest data flow problems. Among the initial works that introduced most common bit vector problems, Cocke [24] and Ullman [100] described available expressions analysis and its use in common subexpression elimination, Allen [4, 5] presented reaching definitions analysis, and live variables analysis was described by Kennedy [55, 56]. Partial redundancy elimination was introduced by Morel and Renvoise [74]. Bodik, Gupta and Soffa [17] discuss a combination of *must* and *may* availability and its use in complete removal of redundancies. Knoop, Rüthing and Steffen [65] introduced lazy code motion. Almost every book on compiler construction discusses bit vector data flow frameworks. A detailed treatment can be found in the advanced texts on compilers such as Aho, Lam, Sethi, and Ullman [3], Appel [10], or Muchnick [76] or in the books devoted to static analysis such as by Hecht [44], Muchnick and Jones [77], and F. Nielson, H. R. Nielson and Hankin [80]. The first formal definition of bit vector frameworks was provided by Khedker and Dhamdhere [60].