Chapter 9

Machine-Independent Optimizations

High-level language constructs can introduce substantial run-time overhead if we naively translate each construct independently into machine code. This chapter discusses how to eliminate many of these inefficiencies. Elimination of unnecessary instructions in object code, or the replacement of one sequence of instructions by a faster sequence of instructions that does the same thing is usually called "code improvement" or "code optimization."

Local code optimization (code improvement within a basic block) was introduced in Section 8.5. This chapter deals with *global* code optimization, where improvements take into account what happens across basic blocks. We begin in Section 9.1 with a discussion of the principal opportunities for code improvement.

Most global optimizations are based on *data-flow analyses*, which are algorithms to gather information about a program. The results of data-flow analyses all have the same form: for each instruction in the program, they specify some property that must hold every time that instruction is executed. The analyses differ in the properties they compute. For example, a constant-propagation analysis computes, for each point in the program, and for each variable used by the program, whether that variable has a unique constant value at that point. This information may be used to replace variable references by constant values, for instance. As another example, a liveness analysis determines, for each point in the program, whether the value held by a particular variable at that point is sure to be overwritten before it is read. If so, we do not need to preserve that value, either in a register or in a memory location.

We introduce data-flow analysis in Section 9.2, including several important examples of the kind of information we gather globally and then use to improve the code. Section 9.3 introduces the general idea of a data-flow framework, of which the data-flow analyses in Section 9.2 are special cases. We can use essentially the same algorithms for all these instances of data-flow analysis, and

we can measure the performance of these algorithms and show their correctness on all instances, as well. Section 9.4 is an example of the general framework that does more powerful analysis than the earlier examples. Then, in Section 9.5 we consider a powerful technique, called "partial redundancy elimination," for optimizing the placement of each expression evaluation in the program. The solution to this problem requires the solution of a variety of different data-flow problems.

In Section 9.6 we take up the discovery and analysis of loops in programs. The identification of loops leads to another family of algorithms for solving data-flow problems that is based on the hierarchical structure of the loops of a well-formed ("reducible") program. This approach to data-flow analysis is covered in Section 9.7. Finally, Section 9.8 uses hierarchical analysis to eliminate induction variables (essentially, variables that count the number of iterations around a loop). This code improvement is one of the most important we can make for programs written in commonly used programming languages.

9.1 The Principal Sources of Optimization

A compiler optimization must preserve the semantics of the original program. Except in very special circumstances, once a programmer chooses and implements a particular algorithm, the compiler cannot understand enough about the program to replace it with a substantially different and more efficient algorithm. A compiler knows only how to apply relatively low-level semantic transformations, using general facts such as algebraic identities like $i + 0 = i$ or program semantics such as the fact that performing the same operation on the same values yields the same result.

9.1.1 Causes of Redundancy

There are many redundant operations in a typical program. Sometimes the redundancy is available at the source level. For instance, a programmer may find it more direct and convenient to recalculate some result, leaving it to the compiler to recognize that only one such calculation is necessary. But more often, the redundancy is a side effect of having written the program in a high-level language. In most languages (other than C or C++, where pointer arithmetic is allowed), programmers have no choice but to refer to elements of an array or fields in a structure through accesses like $A[i][j]$ or $X \rightarrow f1$.

As a program is compiled, each of these high-level data-structure accesses expands into a number of low-level arithmetic operations, such as the computation of the location of the (i, j)th element of a matrix A. Accesses to the same data structure often share many common low-level operations. Programmers are not aware of these low-level operations and cannot eliminate the redundancies themselves. It is, in fact, preferable from a software-engineering perspective that programmers only access data elements by their high-level names; the

programs are easier to write and, more importantly, easier to understand and evolve. By having a compiler eliminate the redundancies, we get the best of both worlds: the programs are both efficient and easy to maintain.

9.1.2 A Running Example: Quicksort

In the following, we shall use a fragment of a sorting program called *quicksort* to illustrate several important code-improving transformations. The C program in Fig. 9.1 is derived from Sedgewick,[1] who discussed the hand-optimization of such a program. We shall not discuss all the subtle algorithmic aspects of this program here, for example, the fact that $a[0]$ must contain the smallest of the sorted elements, and $a[max]$ the largest.

```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Figure 9.1: C code for quicksort

Before we can optimize away the redundancies in address calculations, the address operations in a program first must be broken down into low-level arithmetic operations to expose the redundancies. In the rest of this chapter, we assume that the intermediate representation consists of three-address statements, where temporary variables are used to hold all the results of intermediate expressions. Intermediate code for the marked fragment of the program in Fig. 9.1 is shown in Fig. 9.2.

In this example we assume that integers occupy four bytes. The assignment `x = a[i]` is translated as in Section 6.4.4 into the two three-address statements

[1] R. Sedgewick, "Implementing Quicksort Programs," *Comm. ACM*, **21**, 1978, pp. 847–857.

```
 (1)    i = m-1                 (16)    t7 = 4*i
 (2)    j = n                   (17)    t8 = 4*j
 (3)    t1 = 4*n                (18)    t9 = a[t8]
 (4)    v = a[t1]               (19)    a[t7] = t9
 (5)    i = i+1                 (20)    t10 = 4*j
 (6)    t2 = 4*i                (21)    a[t10] = x
 (7)    t3 = a[t2]              (22)    goto (5)
 (8)    if t3<v goto (5)        (23)    t11 = 4*i
 (9)    j = j-1                 (24)    x  = a[t11]
(10)    t4 = 4*j                (25)    t12 = 4*i
(11)    t5 = a[t4]              (26)    t13 = 4*n
(12)    if t5>v goto (9)        (27)    t14 = a[t13]
(13)    if i>=j goto (23)       (28)    a[t12] = t14
(14)    t6 = 4*i                (29)    t15 = 4*n
(15)    x = a[t6]               (30)    a[t15] = x
```

Figure 9.2: Three-address code for fragment in Fig. 9.1

```
t6 = 4*i
x = a[t6]
```

as shown in steps (14) and (15) of Fig. 9.2. Similarly, `a[j] = x` becomes

```
t10 = 4*j
a[t10] = x
```

in steps (20) and (21). Notice that every array access in the original program translates into a pair of steps, consisting of a multiplication and an array-subscripting operation. As a result, this short program fragment translates into a rather long sequence of three-address operations.

Figure 9.3 is the flow graph for the program in Fig. 9.2. Block B_1 is the entry node. All conditional and unconditional jumps to statements in Fig. 9.2 have been replaced in Fig. 9.3 by jumps to the block of which the statements are leaders, as in Section 8.4. In Fig. 9.3, there are three loops. Blocks B_2 and B_3 are loops by themselves. Blocks B_2, B_3, B_4, and B_5 together form a loop, with B_2 the only entry point.

9.1.3 Semantics-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes. Common-subexpression elimination, copy propagation, dead-code elimination, and constant folding are common examples of such function-preserving (or *semantics-preserving*) transformations; we shall consider each in turn.

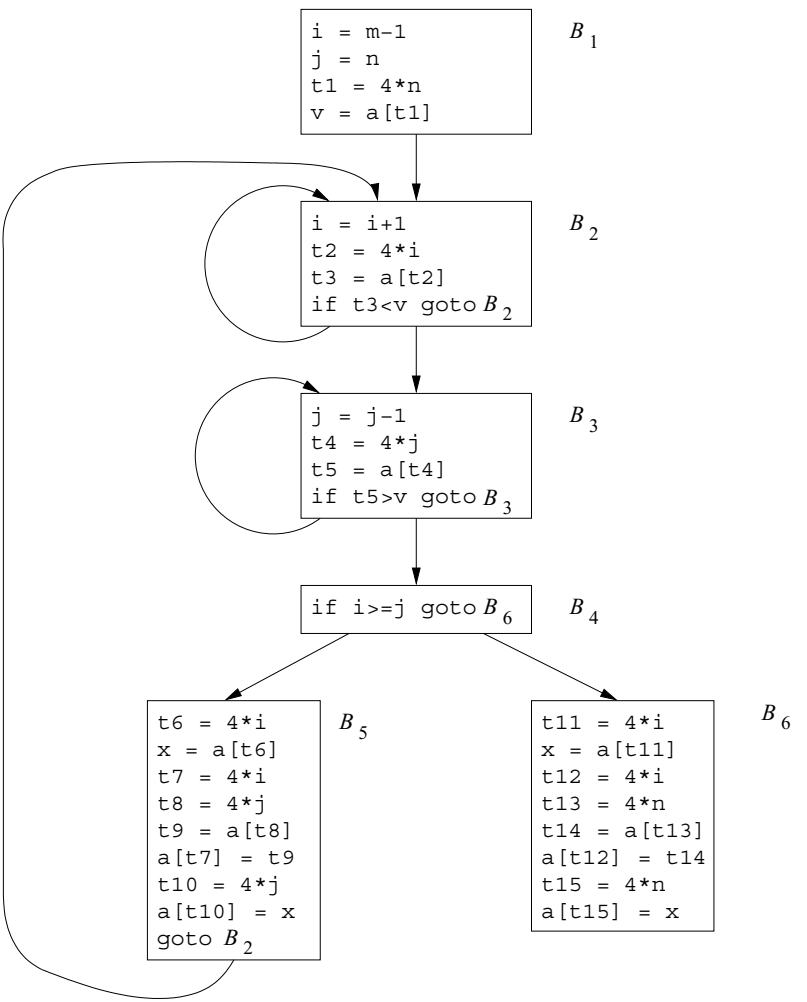
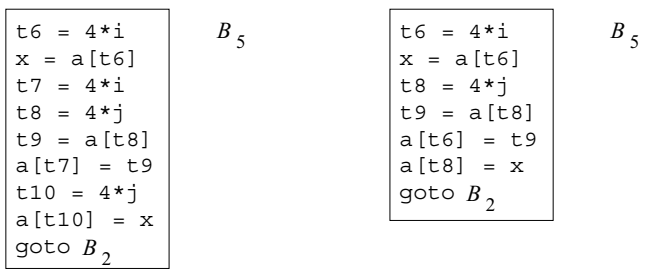


Figure 9.3: Flow graph for the quicksort fragment

Frequently, a program will include several calculations of the same value, such as an offset in an array. As mentioned in Section 9.1.2, some of these duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language. For example, block B_5 shown in Fig. 9.4(a) recalculates $4*i$ and $4*j$, although none of these calculations were requested explicitly by the programmer.



(a) Before. (b) After.

Figure 9.4: Local common-subexpression elimination

9.1.4 Global Common Subexpressions

An occurrence of an expression E is called a *common subexpression* if E was previously computed and the values of the variables in E have not changed since the previous computation. We avoid recomputing E if we can use its previously computed value; that is, the variable x to which the previous computation of E was assigned has not changed in the interim.[2]

Example 9.1: The assignments to `t7` and `t10` in Fig. 9.4(a) compute the common subexpressions $4*i$ and $4*j$, respectively. These steps have been eliminated in Fig. 9.4(b), which uses `t6` instead of `t7` and `t8` instead of `t10`. □

Example 9.2: Figure 9.5 shows the result of eliminating both global and local common subexpressions from blocks B_5 and B_6 in the flow graph of Fig. 9.3. We first discuss the transformation of B_5 and then mention some subtleties involving arrays.

After local common subexpressions are eliminated, B_5 still evaluates $4*i$ and $4*j$, as shown in Fig. 9.4(b). Both are common subexpressions; in particular, the three statements

[2] If x has changed, it may still be possible to reuse the computation of E if we assign its value to a new variable y, as well as to x, and use the value of y in place of a recomputation of E.

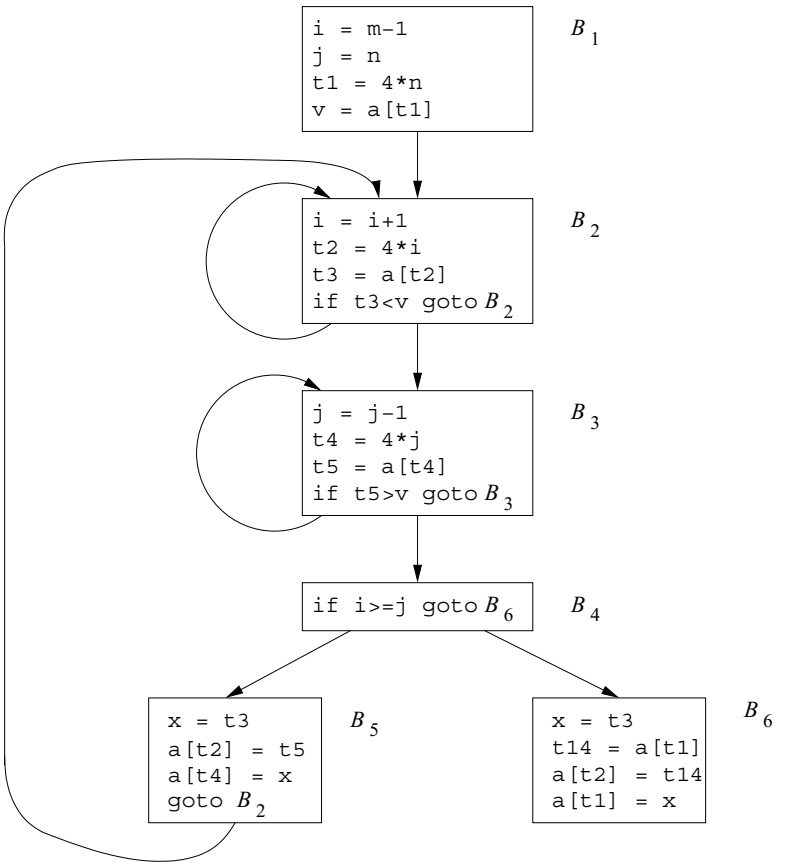


Figure 9.5: B_5 and B_6 after common-subexpression elimination

```
t8 = 4*j
t9 = a[t8]
a[t8] = x
```

in B_5 can be replaced by

```
t9 = a[t4]
a[t4] = x
```

using $t4$ computed in block B_3. In Fig. 9.5, observe that as control passes from the evaluation of $4 * j$ in B_3 to B_5, there is no change to j and no change to $t4$, so $t4$ can be used if $4 * j$ is needed.

Another common subexpression comes to light in B_5 after $t4$ replaces $t8$. The new expression $a[t4]$ corresponds to the value of $a[j]$ at the source level. Not only does j retain its value as control leaves B_3 and then enters B_5, but

$a[j]$, a value computed into a temporary $t5$, does too, because there are no assignments to elements of the array a in the interim. The statements

```
t9 = a[t4]
a[t6] = t9
```

in B_5 therefore can be replaced by

```
a[t6] = t5
```

Analogously, the value assigned to x in block B_5 of Fig. 9.4(b) is seen to be the same as the value assigned to $t3$ in block B_2. Block B_5 in Fig. 9.5 is the result of eliminating common subexpressions corresponding to the values of the source level expressions $a[i]$ and $a[j]$ from B_5 in Fig. 9.4(b). A similar series of transformations has been done to B_6 in Fig. 9.5.

The expression $a[t1]$ in blocks B_1 and B_6 of Fig. 9.5 is *not* considered a common subexpression, although $t1$ can be used in both places. After control leaves B_1 and before it reaches B_6, it can go through B_5, where there are assignments to a. Hence, $a[t1]$ may not have the same value on reaching B_6 as it did on leaving B_1, and it is not safe to treat $a[t1]$ as a common subexpression. □

9.1.5 Copy Propagation

Block B_5 in Fig. 9.5 can be further improved by eliminating x, using two new transformations. One concerns assignments of the form `u = v` called *copy statements*, or *copies* for short. Had we gone into more detail in Example 9.2, copies would have arisen much sooner, because the normal algorithm for eliminating common subexpressions introduces them, as do several other algorithms.

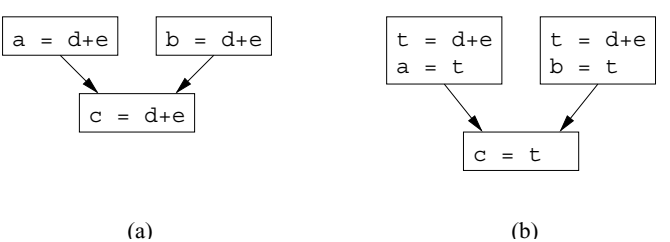


Figure 9.6: Copies introduced during common subexpression elimination

Example 9.3 : In order to eliminate the common subexpression from the statement `c = d+e` in Fig. 9.6(a), we must use a new variable t to hold the value of $d+e$. The value of variable t, instead of that of the expression $d+e$, is assigned to c in Fig. 9.6(b). Since control may reach `c = d+e` either after the assignment to a or after the assignment to b, it would be incorrect to replace `c = d+e` by either `c = a` or by `c = b`. □

The idea behind the copy-propagation transformation is to use v for u, wherever possible after the copy statement `u = v`. For example, the assignment `x = t3` in block B_5 of Fig. 9.5 is a copy. Copy propagation applied to B_5 yields the code in Fig. 9.7. This change may not appear to be an improvement, but, as we shall see in Section 9.1.6, it gives us the opportunity to eliminate the assignment to x.

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

Figure 9.7: Basic block B_5 after copy propagation

9.1.6 Dead-Code Elimination

A variable is *live* at a point in a program if its value can be used subsequently; otherwise, it is *dead* at that point. A related idea is *dead* (or *useless*) *code* — statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

Example 9.4: Suppose `debug` is set to `TRUE` or `FALSE` at various points in the program, and used in statements like

```
if (debug) print ...
```

It may be possible for the compiler to deduce that each time the program reaches this statement, the value of `debug` is `FALSE`. Usually, it is because there is one particular statement

```
debug = FALSE
```

that must be the last assignment to `debug` prior to any tests of the value of `debug`, no matter what sequence of branches the program actually takes. If copy propagation replaces `debug` by `FALSE`, then the print statement is dead because it cannot be reached. We can eliminate both the test and the print operation from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as *constant folding*. □

One advantage of copy propagation is that it often turns the copy statement into dead code. For example, copy propagation followed by dead-code elimination removes the assignment to x and transforms the code in Fig 9.7 into

```
a[t2] = t5
a[t4] = t3
goto B2
```

This code is a further improvement of block B_5 in Fig. 9.5.

9.1.7 Code Motion

Loops are a very important place for optimizations, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.

An important modification that decreases the amount of code in a loop is *code motion*. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a *loop-invariant computation*) and evaluates the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop, that is, one basic block to which all jumps from outside the loop go (see Section 8.4.5).

Example 9.5: Evaluation of $limit - 2$ is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2) /* statement does not change limit */
```

Code motion will result in the equivalent code

```
t = limit-2
while (i <= t)  /* statement does not change limit or t */
```

Now, the computation of $limit - 2$ is performed once, before we enter the loop. Previously, there would be $n+1$ calculations of $limit - 2$ if we iterated the body of the loop n times. □

9.1.8 Induction Variables and Reduction in Strength

Another important optimization is to find induction variables in loops and optimize their computation. A variable x is said to be an "induction variable" if there is a positive or negative constant c such that each time x is assigned, its value increases by c. For instance, i and $t2$ are induction variables in the loop containing B_2 of Fig. 9.5. Induction variables can be computed with a single increment (addition or subtraction) per loop iteration. The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as *strength reduction*. But induction variables not only allow us sometimes to perform a strength reduction; often it is possible to eliminate all but one of a group of induction variables whose values remain in lock step as we go around the loop.

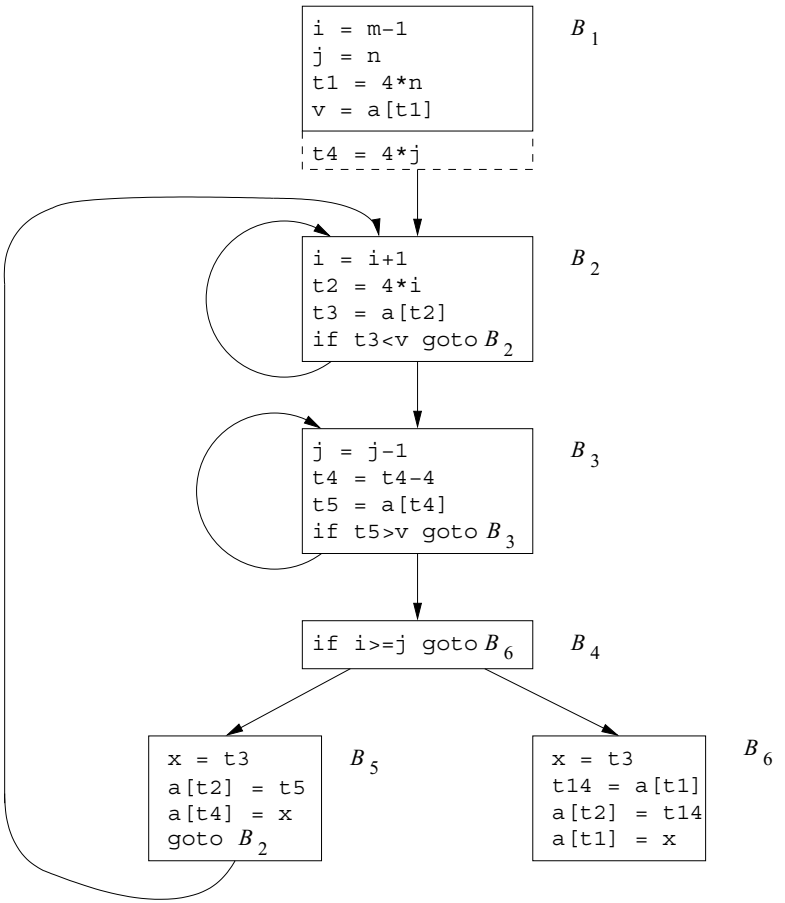


Figure 9.8: Strength reduction applied to $4 * j$ in block B_3

When processing loops, it is useful to work "inside-out"; that is, we shall start with the inner loops and proceed to progressively larger, surrounding loops. Thus, we shall see how this optimization applies to our quicksort example by beginning with one of the innermost loops: B_3 by itself. Note that the values of j and $t4$ remain in lock step; every time the value of j decreases by 1, the value of $t4$ decreases by 4, because $4 * j$ is assigned to $t4$. These variables, j and $t4$, thus form a good example of a pair of induction variables.

When there are two or more induction variables in a loop, it may be possible to get rid of all but one. For the inner loop of B_3 in Fig. 9.5, we cannot get rid of either j or $t4$ completely; $t4$ is used in B_3 and j is used in B_4. However, we can illustrate reduction in strength and a part of the process of induction-variable elimination. Eventually, j will be eliminated when the outer loop consisting of blocks B_2, B_3, B_4 and B_5 is considered.

Example 9.6 : As the relationship $t4 = 4 * j$ surely holds after assignment to $t4$ in Fig. 9.5, and $t4$ is not changed elsewhere in the inner loop around B_3, it follows that just after the statement `j = j-1` the relationship $t4 = 4 * j + 4$ must hold. We may therefore replace the assignment `t4 = 4*j` by `t4 = t4-4`. The only problem is that $t4$ does not have a value when we enter block B_3 for the first time.

Since we must maintain the relationship $t4 = 4 * j$ on entry to the block B_3, we place an initialization of $t4$ at the end of the block where j itself is initialized, shown by the dashed addition to block B_1 in Fig. 9.8. Although we have added one more instruction, which is executed once in block B_1, the replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines. □

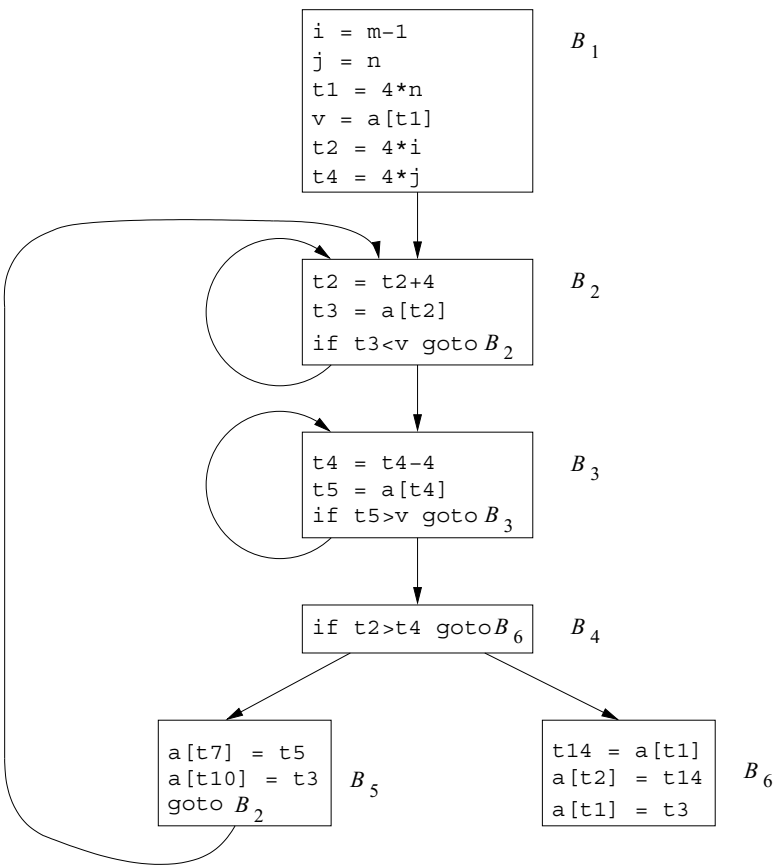


Figure 9.9: Flow graph after induction-variable elimination

We conclude this section with one more instance of induction-variable elim-

ination. This example treats i and j in the context of the outer loop containing B_2, B_3, B_4, and B_5.

Example 9.7: After reduction in strength is applied to the inner loops around B_2 and B_3, the only use of i and j is to determine the outcome of the test in block B_4. We know that the values of i and $t2$ satisfy the relationship $t2 = 4 * i$, while those of j and $t4$ satisfy the relationship $t4 = 4 * j$. Thus, the test $t2 \geq t4$ can substitute for $i \geq j$. Once this replacement is made, i in block B_2 and j in block B_3 become dead variables, and the assignments to them in these blocks become dead code that can be eliminated. The resulting flow graph is shown in Fig. 9.9. □

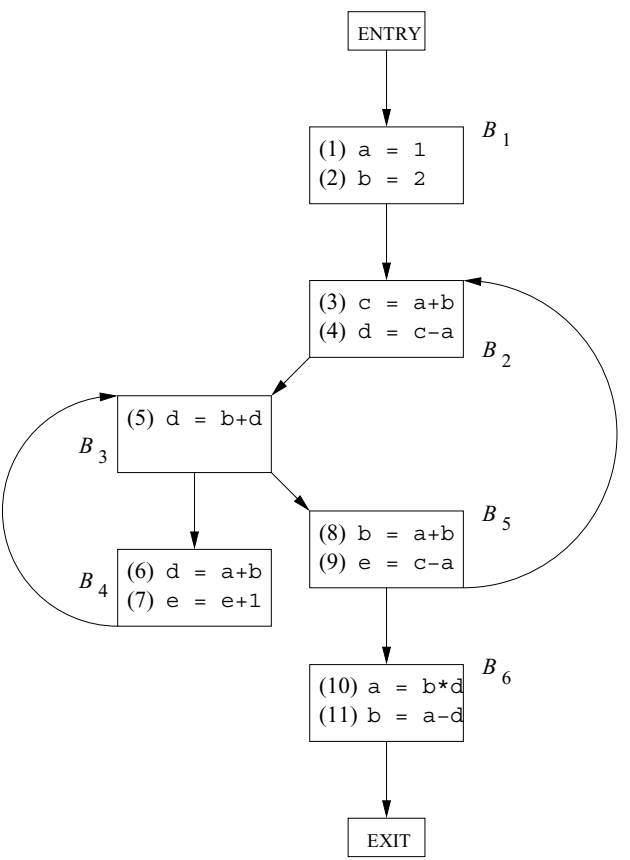


Figure 9.10: Flow graph for Exercise 9.1.1

The code-improving transformations we have discussed have been effective. In Fig. 9.9, the numbers of instructions in blocks B_2 and B_3 have been reduced from 4 to 3, compared with the original flow graph in Fig. 9.3. In B_5, the number

has been reduced from 9 to 3, and in B_6 from 8 to 3. True, B_1 has grown from four instructions to six, but B_1 is executed only once in the fragment, so the total running time is barely affected by the size of B_1.

9.1.9 Exercises for Section 9.1

Exercise 9.1.1: For the flow graph in Fig. 9.10:

a) Identify the loops of the flow graph.

b) Statements (1) and (2) in B_1 are both copy statements, in which a and b are given constant values. For which uses of a and b can we perform copy propagation and replace these uses of variables by uses of a constant? Do so, wherever possible.

c) Identify any global common subexpressions for each loop.

d) Identify any induction variables for each loop. Be sure to take into account any constants introduced in (b).

e) Identify any loop-invariant computations for each loop.

Exercise 9.1.2: Apply the transformations of this section to the flow graph of Fig. 8.9.

Exercise 9.1.3: Apply the transformations of this section to your flow graphs from (a) Exercise 8.4.1; (b) Exercise 8.4.2.

Exercise 9.1.4: In Fig. 9.11 is intermediate code to compute the dot product of two vectors A and B. Optimize this code by eliminating common subexpressions, performing reduction in strength on induction variables, and eliminating all the induction variables you can.

```
    dp = 0.
    i  = 0
L:  t1 = i*8
    t2 = A[t1]
    t3 = i*8
    t4 = B[t3]
    t5 = t2*t4
    dp = dp+t5
    i  = i+1
    if i<n goto L
```

Figure 9.11: Intermediate code to compute the dot product

9.2 Introduction to Data-Flow Analysis

All the optimizations introduced in Section 9.1 depend on *data-flow analysis*. "Data-flow analysis" refers to a body of techniques that derive information about the flow of data along program execution paths. For example, one way to implement global common subexpression elimination requires us to determine whether two textually identical expressions evaluate to the same value along any possible execution path of the program. As another example, if the result of an assignment is not used along any subsequent execution path, then we can eliminate the assignment as dead code. These and many other important questions can be answered by data-flow analysis.

9.2.1 The Data-Flow Abstraction

Following Section 1.6.2, the execution of a program can be viewed as a series of transformations of the program state, which consists of the values of all the variables in the program, including those associated with stack frames below the top of the run-time stack. Each execution of an intermediate-code statement transforms an input state to a new output state. The input state is associated with the *program point before* the statement and the output state is associated with the *program point after* the statement.

When we analyze the behavior of a program, we must consider all the possible sequences of program points ("paths") through a flow graph that the program execution can take. We then extract, from the possible program states at each point, the information we need for the particular data-flow analysis problem we want to solve. In more complex analyses, we must consider paths that jump among the flow graphs for various procedures, as calls and returns are executed. However, to begin our study, we shall concentrate on the paths through a single flow graph for a single procedure.

Let us see what the flow graph tells us about the possible execution paths.

- Within one basic block, the program point after a statement is the same as the program point before the next statement.
- If there is an edge from block B_1 to block B_2, then the program point after the last statement of B_1 may be followed immediately by the program point before the first statement of B_2.

Thus, we may define an *execution path* (or just *path*) from point p_1 to point p_n to be a sequence of points $p_1, p_2, \ldots, p_n$ such that for each $i = 1, 2, \ldots, n-1$, either

1. p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that same statement, or
2. p_i is the end of some block and p_{i+1} is the beginning of a successor block.

In general, there is an infinite number of possible execution paths through a program, and there is no finite upper bound on the length of an execution path. Program analyses summarize all the possible program states that can occur at a point in the program with a finite set of facts. Different analyses may choose to abstract out different information, and in general, no analysis is necessarily a perfect representation of the state.

Example 9.8 : Even the simple program in Fig. 9.12 describes an unbounded number of execution paths. Not entering the loop at all, the shortest complete execution path consists of the program points $(1, 2, 3, 4, 9)$. The next shortest path executes one iteration of the loop and consists of the points $(1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9)$. We know that, for example, the first time program point (5) is executed, the value of a is 1 due to definition d_1. We say that d_1 *reaches* point (5) in the first iteration. In subsequent iterations, d_3 reaches point (5) and the value of a is 243.

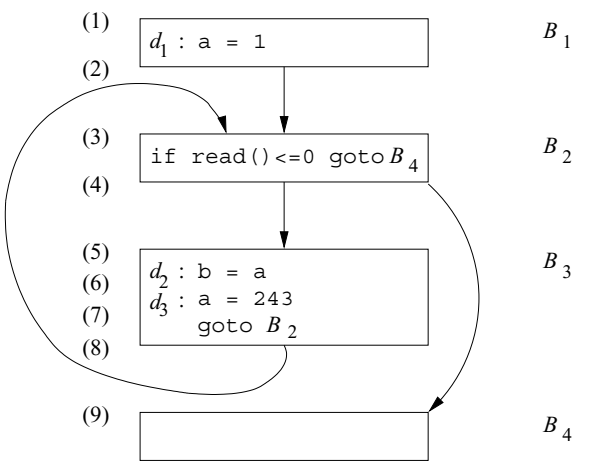


Figure 9.12: Example program illustrating the data-flow abstraction

In general, it is not possible to keep track of all the program states for all possible paths. In data-flow analysis, we do not distinguish among the paths taken to reach a program point. Moreover, we do not keep track of entire states; rather, we abstract out certain details, keeping only the data we need for the purpose of the analysis. Two examples will illustrate how the same program states may lead to different information abstracted at a point.

1. To help users debug their programs, we may wish to find out what are all the values a variable may have at a program point, and where these values may be defined. For instance, we may summarize all the program states at point (5) by saying that the value of a is one of $\{1, 243\}$, and that it may be defined by one of $\{d_1, d_3\}$. The definitions that *may* reach a program point along some path are known as *reaching definitions*.

2. Suppose, instead, we are interested in implementing constant folding. If a use of the variable x is reached by only one definition, and that definition assigns a constant to x, then we can simply replace x by the constant. If, on the other hand, several definitions of x may reach a single program point, then we cannot perform constant folding on x. Thus, for constant folding we wish to find those definitions that are the unique definition of their variable to reach a given program point, no matter which execution path is taken. For point (5) of Fig. 9.12, there is no definition that *must* be the definition of a at that point, so this set is empty for a at point (5). Even if a variable has a unique definition at a point, that definition must assign a constant to the variable. Thus, we may simply describe certain variables as "not a constant," instead of collecting all their possible values or all their possible definitions.

Thus, we see that the same information may be summarized differently, depending on the purpose of the analysis. □

9.2.2 The Data-Flow Analysis Schema

In each application of data-flow analysis, we associate with every program point a *data-flow value* that represents an abstraction of the set of all possible program states that can be observed for that point. The set of possible data-flow values is the *domain* for this application. For example, the domain of data-flow values for reaching definitions is the set of all subsets of definitions in the program. A particular data-flow value is a set of definitions, and we want to associate with each point in the program the exact set of definitions that can reach that point. As discussed above, the choice of abstraction depends on the goal of the analysis; to be efficient, we only keep track of information that is relevant.

We denote the data-flow values before and after each statement s by IN[s] and OUT[s], respectively. The *data-flow problem* is to find a solution to a set of constraints on the IN[s]'s and OUT[s]'s, for all statements s. There are two sets of constraints: those based on the semantics of the statements ("transfer functions") and those based on the flow of control.

Transfer Functions

The data-flow values before and after a statement are constrained by the semantics of the statement. For example, suppose our data-flow analysis involves determining the constant value of variables at points. If variable a has value v before executing statement `b = a`, then both a and b will have the value v after the statement. This relationship between the data-flow values before and after the assignment statement is known as a *transfer function*.

Transfer functions come in two flavors: information may propagate forward along execution paths, or it may flow backwards up the execution paths. In a forward-flow problem, the transfer function of a statement s, which we shall

usually denote f_s, takes the data-flow value before the statement and produces a new data-flow value after the statement. That is,

$$\text{OUT}[s] = f_s(\text{IN}[s]).$$

Conversely, in a backward-flow problem, the transfer function f_s for statement s converts a data-flow value after the statement to a new data-flow value before the statement. That is,

$$\text{IN}[s] = f_s(\text{OUT}[s]).$$

Control-Flow Constraints

The second set of constraints on data-flow values is derived from the flow of control. Within a basic block, control flow is simple. If a block B consists of statements $s_1, s_2, \ldots, s_n$ in that order, then the control-flow value out of s_i is the same as the control-flow value into s_{i+1}. That is,

$$\text{IN}[s_{i+1}] = \text{OUT}[s_i], \text{ for all } i = 1, 2, \ldots, n-1.$$

However, control-flow edges between basic blocks create more complex constraints between the last statement of one basic block and the first statement of the following block. For example, if we are interested in collecting all the definitions that may reach a program point, then the set of definitions reaching the leader statement of a basic block is the union of the definitions after the last statements of each of the predecessor blocks. The next section gives the details of how data flows among the blocks.

9.2.3 Data-Flow Schemas on Basic Blocks

While a data-flow schema technically involves data-flow values at each point in the program, we can save time and space by recognizing that what goes on inside a block is usually quite simple. Control flows from the beginning to the end of the block, without interruption or branching. Thus, we can restate the schema in terms of data-flow values entering and leaving the blocks. We denote the data-flow values immediately before and immediately after each basic block B by $\text{IN}[B]$ and $\text{OUT}[B]$, respectively. The constraints involving $\text{IN}[B]$ and $\text{OUT}[B]$ can be derived from those involving $\text{IN}[s]$ and $\text{OUT}[s]$ for the various statements s in B as follows.

Suppose block B consists of statements $s_1, \ldots, s_n$, in that order. If s_1 is the first statement of basic block B, then $\text{IN}[B] = \text{IN}[s_1]$, Similarly, if s_n is the last statement of basic block B, then $\text{OUT}[B] = \text{OUT}[s_n]$. The transfer function of a basic block B, which we denote f_B, can be derived by composing the transfer functions of the statements in the block. That is, let f_{s_i} be the transfer function of statement s_i. Then $f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$. The relationship between the beginning and end of the block is

$$\text{OUT}[B] = f_B(\text{IN}[B]).$$

The constraints due to control flow between basic blocks can easily be rewritten by substituting IN[B] and OUT[B] for IN[s_1] and OUT[s_n], respectively. For instance, if data-flow values are information about the sets of constants that *may* be assigned to a variable, then we have a forward-flow problem in which

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P].$$

When the data-flow is backwards as we shall soon see in live-variable analysis, the equations are similar, but with the roles of the IN's and OUT's reversed. That is,

$$\begin{aligned} \text{IN}[B] &= f_B(\text{OUT}[B]) \\ \text{OUT}[B] &= \bigcup_{S \text{ a successor of } B} \text{IN}[S]. \end{aligned}$$

Unlike linear arithmetic equations, the data-flow equations usually do not have a unique solution. Our goal is to find the most "precise" solution that satisfies the two sets of constraints: control-flow and transfer constraints. That is, we need a solution that encourages valid code improvements, but does not justify unsafe transformations — those that change what the program computes. This issue is discussed briefly in the box on "Conservatism" and more extensively in Section 9.3.4. In the following subsections, we discuss some of the most important examples of problems that can be solved by data-flow analysis.

9.2.4 Reaching Definitions

"Reaching definitions" is one of the most common and useful data-flow schemas. By knowing where in a program each variable x may have been defined when control reaches each point p, we can determine many things about x. For just two examples, a compiler then knows whether x is a constant at point p, and a debugger can tell whether it is possible for x to be an undefined variable, should x be used at p.

We say a definition d *reaches* a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path. We *kill* a definition of a variable x if there is any other definition of x anywhere along the path.[3] Intuitively, if a definition d of some variable x reaches point p, then d might be the place at which the value of x used at p was last defined.

A definition of a variable x is a statement that assigns, or may assign, a value to x. Procedure parameters, array accesses, and indirect references all may have aliases, and it is not easy to tell if a statement is referring to a particular variable x. Program analysis must be conservative; if we do not

[3] Note that the path may have loops, so we could come to another occurrence of d along the path, which does not "kill" d.

Detecting Possible Uses Before Definition

Here is how we use a solution to the reaching-definitions problem to detect uses before definition. The trick is to introduce a dummy definition for each variable x in the entry to the flow graph. If the dummy definition of x reaches a point p where x might be used, then there might be an opportunity to use x before definition. Note that we can never be absolutely certain that the program has a bug, since there may be some reason, possibly involving a complex logical argument, why the path along which p is reached without a real definition of x can never be taken.

know whether a statement s is assigning a value to x, we must assume that it *may* assign to it; that is, variable x after statement s may have either its original value before s or the new value created by s. For the sake of simplicity, the rest of the chapter assumes that we are dealing only with variables that have no aliases. This class of variables includes all local scalar variables in most languages; in the case of C and C++, local variables whose addresses have been computed at some point are excluded.

Example 9.9 : Shown in Fig. 9.13 is a flow graph with seven definitions. Let us focus on the definitions reaching block B_2. All the definitions in block B_1 reach the beginning of block B_2. The definition d_5: `j = j-1` in block B_2 also reaches the beginning of block B_2, because no other definitions of j can be found in the loop leading back to B_2. This definition, however, kills the definition d_2: `j = n`, preventing it from reaching B_3 or B_4. The statement d_4: `i = i+1` in B_2 does not reach the beginning of B_2 though, because the variable i is always redefined by d_7: `i = u3`. Finally, the definition d_6: `a = u2` also reaches the beginning of block B_2. □

By defining reaching definitions as we have, we sometimes allow inaccuracies. However, they are all in the "safe," or "conservative," direction. For example, notice our assumption that all edges of a flow graph can be traversed. This assumption may not be true in practice. For example, for no values of a and b can the flow of control actually reach *statement* 2 in the following program fragment:

`if (a == b)` *statement* 1; `else if (a == b)` *statement* 2;

To decide in general whether each path in a flow graph can be taken is an undecidable problem. Thus, we simply assume that every path in the flow graph can be followed in some execution of the program. In most applications of reaching definitions, it is conservative to assume that a definition can reach a point even if it might not. Thus, we may allow paths that are never be traversed in any execution of the program, and we may allow definitions to pass through ambiguous definitions of the same variable safely.

Conservatism in Data-Flow Analysis

Since all data-flow schemas compute approximations to the ground truth (as defined by all possible execution paths of the program), we are obliged to assure that any errors are in the "safe" direction. A policy decision is *safe* (or *conservative*) if it never allows us to change what the program computes. Safe policies may, unfortunately, cause us to miss some code improvements that would retain the meaning of the program, but in essentially all code optimizations there is no safe policy that misses nothing. It would generally be unacceptable to use an unsafe policy — one that speeds up the code at the expense of changing what the program computes.

Thus, when designing a data-flow schema, we must be conscious of how the information will be used, and make sure that any approximations we make are in the "conservative" or "safe" direction. Each schema and application must be considered independently. For instance, if we use reaching definitions for constant folding, it is safe to think a definition reaches when it doesn't (we might think x is not a constant, when in fact it is and could have been folded), but not safe to think a definition doesn't reach when it does (we might replace x by a constant, when the program would at times have a value for x other than that constant).

Transfer Equations for Reaching Definitions

We shall now set up the constraints for the reaching definitions problem. We start by examining the details of a single statement. Consider a definition

$$d\text{: } \texttt{u = v+w}$$

Here, and frequently in what follows, + is used as a generic binary operator.

This statement "generates" a definition d of variable u and "kills" all the other definitions in the program that define variable u, while leaving the remaining incoming definitions unaffected. The transfer function of definition d thus can be expressed as

$$f_d(x) = gen_d \cup (x - kill_d) \tag{9.1}$$

where $gen_d = \{d\}$, the set of definitions generated by the statement, and $kill_d$ is the set of all other definitions of u in the program.

As discussed in Section 9.2.2, the transfer function of a basic block can be found by composing the transfer functions of the statements contained therein. The composition of functions of the form (9.1), which we shall refer to as "*gen-kill* form," is also of that form, as we can see as follows. Suppose there are two functions $f_1(x) = gen_1 \cup (x - kill_1)$ and $f_2(x) = gen_2 \cup (x - kill_2)$. Then

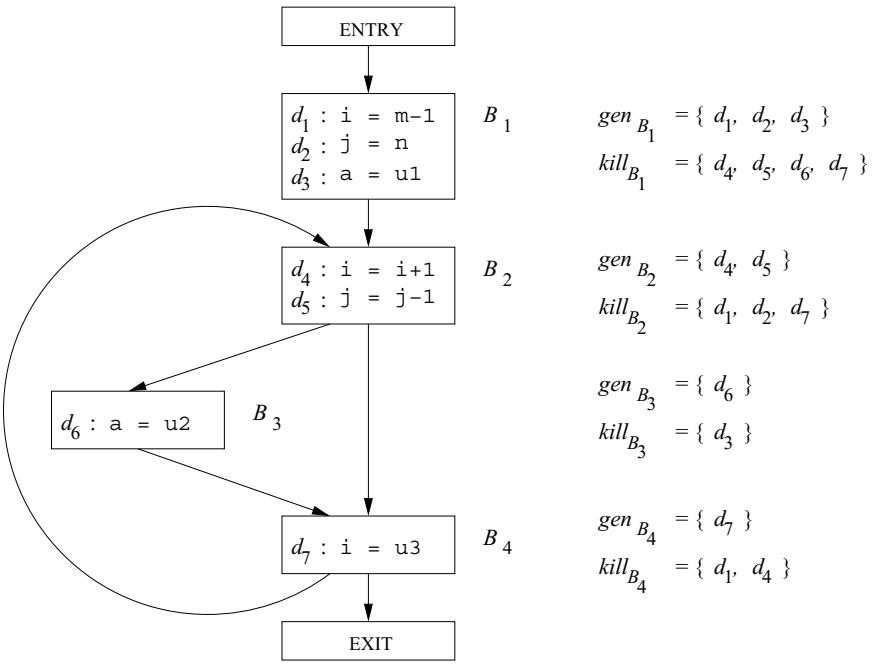


Figure 9.13: Flow graph for illustrating reaching definitions

$$\begin{aligned} f_2\big(f_1(x)\big) &= gen_2 \cup \big(gen_1 \cup (x - kill_1) - kill_2\big) \\ &= \big(gen_2 \cup (gen_1 - kill_2)\big) \cup \big(x - (kill_1 \cup kill_2)\big) \end{aligned}$$

This rule extends to a block consisting of any number of statements. Suppose block B has n statements, with transfer functions $f_i(x) = gen_i \cup (x - kill_i)$ for $i = 1, 2, \ldots, n$. Then the transfer function for block B may be written as:

$$f_B(x) = gen_B \cup (x - kill_B),$$

where

$$kill_B = kill_1 \cup kill_2 \cup \cdots \cup kill_n$$

and

$$\begin{aligned} gen_B &= gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \\ &\quad \cdots \cup (gen_1 - kill_2 - kill_3 - \cdots - kill_n) \end{aligned}$$

Thus, like a statement, a basic block also generates a set of definitions and kills a set of definitions. The *gen* set contains all the definitions inside the block that are "visible" immediately after the block — we refer to them as *downwards exposed*. A definition is downwards exposed in a basic block only if it is not "killed" by a subsequent definition to the same variable inside the same basic block. A basic block's *kill* set is simply the union of all the definitions killed by the individual statements. Notice that a definition may appear in both the *gen* and *kill* set of a basic block. If so, the fact that it is in *gen* takes precedence, because in *gen-kill* form, the *kill* set is applied before the *gen* set.

Example 9.10 : The *gen* set for the basic block

d_1: `a = 3`
d_2: `a = 4`

is $\{d_2\}$ since d_1 is not downwards exposed. The *kill* set contains both d_1 and d_2, since d_1 kills d_2 and vice versa. Nonetheless, since the subtraction of the *kill* set precedes the union operation with the *gen* set, the result of the transfer function for this block always includes definition d_2. □

Control-Flow Equations

Next, we consider the set of constraints derived from the control flow between basic blocks. Since a definition reaches a program point as long as there exists at least one path along which the definition reaches, $\text{OUT}[P] \subseteq \text{IN}[B]$ whenever there is a control-flow edge from P to B. However, since a definition cannot reach a point unless there is a path along which it reaches, $\text{IN}[B]$ needs to be no larger than the union of the reaching definitions of all the predecessor blocks. That is, it is safe to assume

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$$

We refer to union as the *meet operator* for reaching definitions. In any data-flow schema, the meet operator is the one we use to create a summary of the contributions from different paths at the confluence of those paths.

Iterative Algorithm for Reaching Definitions

We assume that every control-flow graph has two empty basic blocks, an ENTRY node, which represents the starting point of the graph, and an EXIT node to which all exits out of the graph go. Since no definitions reach the beginning of the graph, the transfer function for the ENTRY block is a simple constant function that returns $\emptyset$ as an answer. That is, $\text{OUT}[\text{ENTRY}] = \emptyset$.

The reaching definitions problem is defined by the following equations:

$$\text{OUT}[\text{ENTRY}] = \emptyset$$

and for all basic blocks B other than ENTRY,

$$\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$$

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P].$$

These equations can be solved using the following algorithm. The result of the algorithm is the *least fixedpoint* of the equations, i.e., the solution whose assigned values to the IN's and OUT's is contained in the corresponding values for any other solution to the equations. The result of the algorithm below is acceptable, since any definition in one of the sets IN or OUT surely must reach the point described. It is a desirable solution, since it does not include any definitions that we can be sure do not reach.

Algorithm 9.11: Reaching definitions.

INPUT: A flow graph for which $kill_B$ and gen_B have been computed for each block B.

OUTPUT: IN$[B]$ and OUT$[B]$, the set of definitions reaching the entry and exit of each block B of the flow graph.

METHOD: We use an iterative approach, in which we start with the "estimate" OUT$[B] = \emptyset$ for all B and converge to the desired values of IN and OUT. As we must iterate until the IN's (and hence the OUT's) converge, we could use a boolean variable *change* to record, on each pass through the blocks, whether any OUT has changed. However, in this and in similar algorithms described later, we assume that the exact mechanism for keeping track of changes is understood, and we elide those details.

The algorithm is sketched in Fig. 9.14. The first two lines initialize certain data-flow values.[4] Line (3) starts the loop in which we iterate until convergence, and the inner loop of lines (4) through (6) applies the data-flow equations to every block other than the entry. □

Intuitively, Algorithm 9.11 propagates definitions as far as they will go without being killed, thus simulating all possible executions of the program. Algorithm 9.11 will eventually halt, because for every B, OUT$[B]$ never shrinks; once a definition is added, it stays there forever. (See Exercise 9.2.6.) Since the set of all definitions is finite, eventually there must be a pass of the while-loop during which nothing is added to any OUT, and the algorithm then terminates. We are safe terminating then because if the OUT's have not changed, the IN's will

[4] The observant reader will notice that we could easily combine lines (1) and (2). However, in similar data-flow algorithms, it may be necessary to initialize the entry or exit node differently from the way we initialize the other nodes. Thus, we follow a pattern in all iterative algorithms of applying a "boundary condition" like line (1) separately from the initialization of line (2).

```
1)  OUT[ENTRY] = ∅;
2)  for (each basic block B other than ENTRY) OUT[B] = ∅;
3)  while (changes to any OUT occur)
4)          for (each basic block B other than ENTRY) {
5)                  IN[B] = ⋃_{P a predecessor of B} OUT[P];
6)                  OUT[B] = gen_B ∪ (IN[B] − kill_B);
            }
```

Figure 9.14: Iterative algorithm to compute reaching definitions

not change on the next pass. And, if the IN's do not change, the OUT's cannot, so on all subsequent passes there can be no changes.

The number of nodes in the flow graph is an upper bound on the number of times around the while-loop. The reason is that if a definition reaches a point, it can do so along a cycle-free path, and the number of nodes in a flow graph is an upper bound on the number of nodes in a cycle-free path. Each time around the while-loop, each definition progresses by at least one node along the path in question, and it often progresses by more than one node, depending on the order in which the nodes are visited.

In fact, if we properly order the blocks in the for-loop of line (4), there is empirical evidence that the average number of iterations of the while-loop is under 5 (see Section 9.6.7). Since sets of definitions can be represented by bit vectors, and the operations on these sets can be implemented by logical operations on the bit vectors, Algorithm 9.11 is surprisingly efficient in practice.

Example 9.12: We shall represent the seven definitions $d_1, d_2, \ldots, d_7$ in the flow graph of Fig. 9.13 by bit vectors, where bit i from the left represents definition d_i. The union of sets is computed by taking the logical OR of the corresponding bit vectors. The difference of two sets $S - T$ is computed by complementing the bit vector of T, and then taking the logical AND of that complement, with the bit vector for S.

Shown in the table of Fig. 9.15 are the values taken on by the IN and OUT sets in Algorithm 9.11. The initial values, indicated by a superscript 0, as in $\text{OUT}[B]^0$, are assigned, by the loop of line (2) of Fig. 9.14. They are each the empty set, represented by bit vector 000 0000. The values of subsequent passes of the algorithm are also indicated by superscripts, and labeled $\text{IN}[B]^1$ and $\text{OUT}[B]^1$ for the first pass and $\text{IN}[B]^2$ and $\text{OUT}[B]^2$ for the second.

Suppose the for-loop of lines (4) through (6) is executed with B taking on the values

$$B_1, B_2, B_3, B_4, \text{EXIT}$$

in that order. With $B = B_1$, since $\text{OUT}[\text{ENTRY}] = \emptyset$, $\text{IN}[B_1]^1$ is the empty set, and $\text{OUT}[B_1]^1$ is gen_{B_1}. This value differs from the previous value $\text{OUT}[B_1]^0$, so

Block B	OUT$[B]^0$	IN$[B]^1$	OUT$[B]^1$	IN$[B]^2$	OUT$[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

Figure 9.15: Computation of IN and OUT

we now know there is a change on the first round (and will proceed to a second round).

Then we consider $B = B_2$ and compute

$$\begin{aligned} \text{IN}[B_2]^1 &= \text{OUT}[B_1]^1 \cup \text{OUT}[B_4]^0 \\ &= 111\ 0000 + 000\ 0000 = 111\ 0000 \\ \text{OUT}[B_2]^1 &= gen_{B_2} \cup (\text{IN}[B_2]^1 - kill_{B_2}) \\ &= 000\ 1100 + (111\ 0000 - 110\ 0001) = 001\ 1100 \end{aligned}$$

This computation is summarized in Fig. 9.15. For instance, at the end of the first pass, OUT$[B_2]^1 = 001\ 1100$, reflecting the fact that d_4 and d_5 are generated in B_2, while d_3 reaches the beginning of B_2 and is not killed in B_2.

Notice that after the second round, OUT$[B_2]$ has changed to reflect the fact that d_6 also reaches the beginning of B_2 and is not killed by B_2. We did not learn that fact on the first pass, because the path from d_6 to the end of B_2, which is $B_3 \rightarrow B_4 \rightarrow B_2$, is not traversed in that order by a single pass. That is, by the time we learn that d_6 reaches the end of B_4, we have already computed IN$[B_2]$ and OUT$[B_2]$ on the first pass.

There are no changes in any of the OUT sets after the second pass. Thus, after a third pass, the algorithm terminates, with the IN's and OUT's as in the final two columns of Fig. 9.15. □

9.2.5 Live-Variable Analysis

Some code-improving transformations depend on information computed in the direction opposite to the flow of control in a program; we shall examine one such example now. In *live-variable analysis* we wish to know for variable x and point p whether the value of x at p could be used along some path in the flow graph starting at p. If so, we say x is *live* at p; otherwise, x is *dead* at p.

An important use for live-variable information is register allocation for basic blocks. Aspects of this issue were introduced in Sections 8.6 and 8.8. After a value is computed in a register, and presumably used within a block, it is not

necessary to store that value if it is dead at the end of the block. Also, if all registers are full and we need another register, we should favor using a register with a dead value, since that value does not have to be stored.

Here, we define the data-flow equations directly in terms of IN[B] and OUT[B], which represent the set of variables live at the points immediately before and after block B, respectively. These equations can also be derived by first defining the transfer functions of individual statements and composing them to create the transfer function of a basic block. Define

1. def_B as the set of variables *defined* (i.e., definitely assigned values) in B prior to any use of that variable in B, and
2. use_B as the set of variables whose values may be used in B prior to any definition of the variable.

Example 9.13: For instance, block B_2 in Fig. 9.13 definitely uses i. It also uses j before any redefinition of j, unless it is possible that i and j are aliases of one another. Assuming there are no aliases among the variables in Fig. 9.13, then $use_{B_2} = \{i, j\}$. Also, B_2 clearly defines i and j. Assuming there are no aliases, $def_{B_2} = \{i, j\}$, as well. □

As a consequence of the definitions, any variable in use_B must be considered live on entrance to block B, while definitions of variables in def_B definitely are dead at the beginning of B. In effect, membership in def_B "kills" any opportunity for a variable to be live because of paths that begin at B.

Thus, the equations relating *def* and *use* to the unknowns IN and OUT are defined as follows:

$$\text{IN}[\text{EXIT}] = \emptyset$$

and for all basic blocks B other than EXIT,

$$\begin{aligned} \text{IN}[B] &= use_B \cup (\text{OUT}[B] - def_B) \\ \text{OUT}[B] &= \bigcup_{S \text{ a successor of } B} \text{IN}[S] \end{aligned}$$

The first equation specifies the boundary condition, which is that no variables are live on exit from the program. The second equation says that a variable is live coming into a block if either it is used before redefinition in the block or it is live coming out of the block and is not redefined in the block. The third equation says that a variable is live coming out of a block if and only if it is live coming into one of its successors.

The relationship between the equations for liveness and the reaching-definitions equations should be noticed:

- Both sets of equations have union as the meet operator. The reason is that in each data-flow schema we propagate information along paths, and we care only about whether *any* path with desired properties exist, rather than whether something is true along *all* paths.
- However, information flow for liveness travels "backward," opposite to the direction of control flow, because in this problem we want to make sure that the use of a variable x at a point p is transmitted to all points prior to p in an execution path, so that we may know at the prior point that x will have its value used.

To solve a backward problem, instead of initializing OUT[ENTRY], we initialize IN[EXIT]. Sets IN and OUT have their roles interchanged, and *use* and *def* substitute for *gen* and *kill*, respectively. As for reaching definitions, the solution to the liveness equations is not necessarily unique, and we want the solution with the smallest sets of live variables. The algorithm used is essentially a backwards version of Algorithm 9.11.

Algorithm 9.14: Live-variable analysis.

INPUT: A flow graph with *def* and *use* computed for each block.

OUTPUT: IN[B] and OUT[B], the set of variables live on entry and exit of each block B of the flow graph.

METHOD: Execute the program in Fig. 9.16. □

```
IN[EXIT] = ∅;
for (each basic block B other than EXIT) IN[B] = ∅;
while (changes to any IN occur)
        for (each basic block B other than EXIT) {
                OUT[B] = ⋃_{S a successor of B} IN[S];
                IN[B] = use_B ∪ (OUT[B] − def_B);
        }
```

Figure 9.16: Iterative algorithm to compute live variables

9.2.6 Available Expressions

An expression $x + y$ is *available* at a point p if every path from the entry node to p evaluates $x + y$, and after the last such evaluation prior to reaching p, there are no subsequent assignments to x or y.[5] For the available-expressions data-flow schema we say that a block *kills* expression $x + y$ if it assigns (or may

[5] Note that, as usual in this chapter, we use the operator + as a generic operator, not necessarily standing for addition.

assign) x or y and does not subsequently recompute $x + y$. A block *generates* expression $x + y$ if it definitely evaluates $x + y$ and does not subsequently define x or y.

Note that the notion of "killing" or "generating" an available expression is not exactly the same as that for reaching definitions. Nevertheless, these notions of "kill" and "generate" behave essentially as they do for reaching definitions.

The primary use of available-expression information is for detecting global common subexpressions. For example, in Fig. 9.17(a), the expression $4 * i$ in block B_3 will be a common subexpression if $4 * i$ is available at the entry point of block B_3. It will be available if i is not assigned a new value in block B_2, or if, as in Fig. 9.17(b), $4 * i$ is recomputed after i is assigned in B_2.

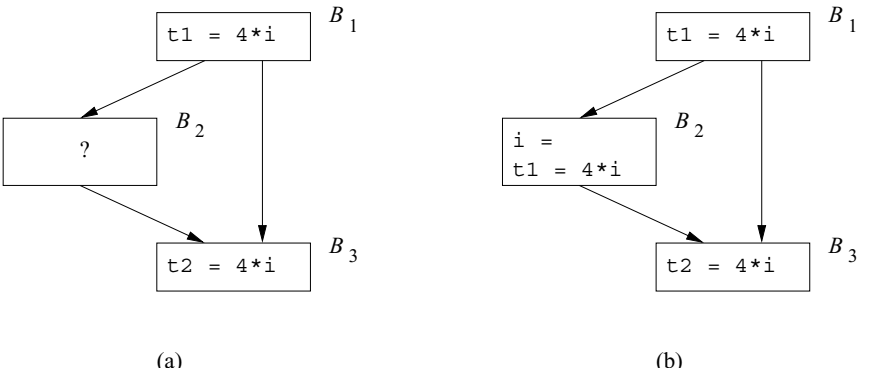


Figure 9.17: Potential common subexpressions across blocks

We can compute the set of generated expressions for each point in a block, working from beginning to end of the block. At the point prior to the block, no expressions are generated. If at point p set S of expressions is available, and q is the point after p, with statement `x = y+z` between them, then we form the set of expressions available at q by the following two steps.

1. Add to S the expression $y + z$.

2. Delete from S any expression involving variable x.

Note the steps must be done in the correct order, as x could be the same as y or z. After we reach the end of the block, S is the set of generated expressions for the block. The set of killed expressions is all expressions, say $y + z$, such that either y or z is defined in the block, and $y + z$ is not generated by the block.

Example 9.15 : Consider the four statements of Fig. 9.18. After the first, $b + c$ is available. After the second statement, $a - d$ becomes available, but $b + c$ is no longer available, because b has been redefined. The third statement does not make $b + c$ available again, because the value of c is immediately changed.

After the last statement, $a - d$ is no longer available, because d has changed. Thus no expressions are generated, and all expressions involving a, b, c, or d are killed. □

Statement	Available Expressions
	$\emptyset$
`a = b + c`	
	$\{b + c\}$
`b = a - d`	
	$\{a - d\}$
`c = b + c`	
	$\{a - d\}$
`d = a - d`	
	$\emptyset$

Figure 9.18: Computation of available expressions

We can find available expressions in a manner reminiscent of the way reaching definitions are computed. Suppose U is the "universal" set of all expressions appearing on the right of one or more statements of the program. For each block B, let IN$[B]$ be the set of expressions in U that are available at the point just before the beginning of B. Let OUT$[B]$ be the same for the point following the end of B. Define e_gen_B to be the expressions generated by B and e_kill_B to be the set of expressions in U killed in B. Note that IN, OUT, e_gen, and e_kill can all be represented by bit vectors. The following equations relate the unknowns IN and OUT to each other and the known quantities e_gen and e_kill:

$$\text{OUT}[\text{ENTRY}] = \emptyset$$

and for all basic blocks B other than ENTRY,

$$\text{OUT}[B] = e\_gen_B \cup (\text{IN}[B] - e\_kill_B)$$
$$\text{IN}[B] = \bigcap_{P \text{ a predecessor of } B} \text{OUT}[P].$$

The above equations look almost identical to the equations for reaching definitions. Like reaching definitions, the boundary condition is OUT[ENTRY] $= \emptyset$, because at the exit of the ENTRY node, there are no available expressions. The most important difference is that the meet operator is intersection rather than union. This operator is the proper one because an expression is available at the beginning of a block only if it is available at the end of *all* its predecessors. In contrast, a definition reaches the beginning of a block whenever it reaches the end of any one or more of its predecessors.

The use of $\cap$ rather than $\cup$ makes the available-expression equations behave differently from those of reaching definitions. While neither set has a unique solution, for reaching definitions, it is the solution with the smallest sets that corresponds to the definition of "reaching," and we obtained that solution by starting with the assumption that nothing reached anywhere, and building up to the solution. In that way, we never assumed that a definition d could reach a point p unless an actual path propagating d to p could be found. In contrast, for available expression equations we want the solution with the largest sets of available expressions, so we start with an approximation that is too large and work down.

It may not be obvious that by starting with the assumption "everything (i.e., the set U) is available everywhere except at the end of the entry block" and eliminating only those expressions for which we can discover a path along which it is not available, we do reach a set of truly available expressions. In the case of available expressions, it is conservative to produce a subset of the exact set of available expressions. The argument for subsets being conservative is that our intended use of the information is to replace the computation of an available expression by a previously computed value. Not knowing an expression is available only inhibits us from improving the code, while believing an expression is available when it is not could cause us to change what the program computes.

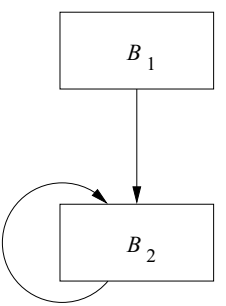


Figure 9.19: Initializing the OUT sets to $\emptyset$ is too restrictive.

Example 9.16: We shall concentrate on a single block, B_2 in Fig. 9.19, to illustrate the effect of the initial approximation of $\text{OUT}[B_2]$ on $\text{IN}[B_2]$. Let G and K abbreviate $e\_gen_{B_2}$ and $e\_kill_{B_2}$, respectively. The data-flow equations for block B_2 are

$$\text{IN}[B_2] = \text{OUT}[B_1] \cap \text{OUT}[B_2]$$

$$\text{OUT}[B_2] = G \cup (\text{IN}[B_2] - K)$$

These equations may be rewritten as recurrences, with I^j and O^j being the jth

approximations of IN$[B_2]$ and OUT$[B_2]$, respectively:

$$I^{j+1} = \text{OUT}[B_1] \cap O^j$$

$$O^{j+1} = G \cup (I^{j+1} - K)$$

Starting with $O^0 = \emptyset$, we get $I^1 = \text{OUT}[B_1] \cap O^0 = \emptyset$. However, if we start with $O^0 = U$, then we get $I^1 = \text{OUT}[B_1] \cap O^0 = \text{OUT}[B_1]$, as we should. Intuitively, the solution obtained starting with $O^0 = U$ is more desirable, because it correctly reflects the fact that expressions in OUT$[B_1]$ that are not killed by B_2 are available at the end of B_2. □

Algorithm 9.17: Available expressions.

INPUT: A flow graph with e_kill_B and e_gen_B computed for each block B. The initial block is B_1.

OUTPUT: IN$[B]$ and OUT$[B]$, the set of expressions available at the entry and exit of each block B of the flow graph.

METHOD: Execute the algorithm of Fig. 9.20. The explanation of the steps is similar to that for Fig. 9.14. □

```
OUT[ENTRY] = ∅;
for (each basic block B other than ENTRY) OUT[B] = U;
while (changes to any OUT occur)
        for (each basic block B other than ENTRY) {
                IN[B] = ⋂_{P a predecessor of B} OUT[P];
                OUT[B] = e_gen_B ∪ (IN[B] − e_kill_B);
        }
```

Figure 9.20: Iterative algorithm to compute available expressions

9.2.7 Summary

In this section, we have discussed three instances of data-flow problems: reaching definitions, live variables, and available expressions. As summarized in Fig. 9.21, the definition of each problem is given by the domain of the data-flow values, the direction of the data flow, the family of transfer functions, the boundary condition, and the meet operator. We denote the meet operator generically as $\wedge$.

The last row shows the initial values used in the iterative algorithm. These values are chosen so that the iterative algorithm will find the most precise solution to the equations. This choice is not strictly a part of the definition of

the data-flow problem, since it is an artifact needed for the iterative algorithm. There are other ways of solving the problem. For example, we saw how the transfer function of a basic block can be derived by composing the transfer functions of the individual statements in the block; a similar compositional approach may be used to compute a transfer function for the entire procedure, or transfer functions from the entry of the procedure to any program point. We shall discuss such an approach in Section 9.7.

	Reaching Definitions	Live Variables	Available Expressions
Domain	Sets of definitions	Sets of variables	Sets of expressions
Direction	Forwards	Backwards	Forwards
Transfer function	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e\_gen_B \cup (x - e\_kill_B)$
Boundary	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
Meet ($\wedge$)	$\cup$	$\cup$	$\cap$
Equations	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
Initialize	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$

Figure 9.21: Summary of three data-flow problems

9.2.8 Exercises for Section 9.2

Exercise 9.2.1: For the flow graph of Fig. 9.10 (see the exercises for Section 9.1), compute

a) The *gen* and *kill* sets for each block.

b) The IN and OUT sets for each block.

Exercise 9.2.2: For the flow graph of Fig. 9.10, compute the *e_gen*, *e_kill*, IN, and OUT sets for available expressions.

Exercise 9.2.3: For the flow graph of Fig. 9.10, compute the *def*, *use*, IN, and OUT sets for live variable analysis.

! **Exercise 9.2.4:** Suppose V is the set of complex numbers. Which of the following operations can serve as the meet operation for a semilattice on V?

a) Addition: $(a + ib) \wedge (c + id) = (a + c) + i(b + d)$.

b) Multiplication: $(a + ib) \wedge (c + id) = (ac - bd) + i(ad + bc)$.

Why the Available-Expressions Algorithm Works

We need to explain why starting all OUT's except that for the entry block with U, the set of all expressions, leads to a conservative solution to the data-flow equations; that is, all expressions found to be available really *are* available. First, because intersection is the meet operation in this data-flow schema, any reason that an expression $x + y$ is found not to be available at a point will propagate forward in the flow graph, along all possible paths, until $x + y$ is recomputed and becomes available again. Second, there are only two reasons $x + y$ could be unavailable:

1. $x + y$ is killed in block B because x or y is defined without a subsequent computation of $x + y$. In this case, the first time we apply the transfer function f_B, $x + y$ will be removed from OUT$[B]$.
2. $x + y$ is never computed along some path. Since $x + y$ is never in OUT[ENTRY], and it is never generated along the path in question, we can show by induction on the length of the path that $x + y$ is eventually removed from IN's and OUT's along that path.

Thus, after changes subside, the solution provided by the iterative algorithm of Fig. 9.20 will include only truly available expressions.

c) Componentwise minimum: $(a + ib) \wedge (c + id) = \min(a, c) + i \min(b, d)$.

d) Componentwise maximum: $(a + ib) \wedge (c + id) = \max(a, c) + i \max(b, d)$.

! **Exercise 9.2.5:** We claimed that if a block B consists of n statements, and the ith statement has gen and kill sets gen_i and $kill_i$, then the transfer function for block B has gen and kill sets gen_B and $kill_B$ given by

$$kill_B = kill_1 \cup kill_2 \cup \cdots \cup kill_n$$

$$\begin{aligned} gen_B \;= {} & gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \\ & \cdots \cup (gen_1 - kill_2 - kill_3 - \cdots - kill_n). \end{aligned}$$

Prove this claim by induction on n.

! **Exercise 9.2.6:** Prove by induction on the number of iterations of the for-loop of lines (4) through (6) of Algorithm 9.11 that none of the IN's or OUT's ever shrinks. That is, once a definition is placed in one of these sets on some round, it never disappears on a subsequent round.

! **Exercise 9.2.7:** Show the correctness of Algorithm 9.11. That is, show that

a) If definition d is put in IN[B] or OUT[B], then there is a path from d to the beginning or end of block B, respectively, along which the variable defined by d might not be redefined.

b) If definition d is not put in IN[B] or OUT[B], then there is no path from d to the beginning or end of block B, respectively, along which the variable defined by d might not be redefined.

! **Exercise 9.2.8:** Prove the following about Algorithm 9.14:

a) The IN's and OUT's never shrink.

b) If variable x is put in IN[B] or OUT[B], then there is a path from the beginning or end of block B, respectively, along which x might be used.

c) If variable x is not put in IN[B] or OUT[B], then there is no path from the beginning or end of block B, respectively, along which x might be used.

! **Exercise 9.2.9:** Prove the following about Algorithm 9.17:

a) The IN's and OUT's never grow; that is, successive values of these sets are subsets (not necessarily proper) of their previous values.

b) If expression e is removed from IN[B] or OUT[B], then there is a path from the entry of the flow graph to the beginning or end of block B, respectively, along which e is either never computed, or after its last computation, one of its arguments might be redefined.

c) If expression e remains in IN[B] or OUT[B], then along every path from the entry of the flow graph to the beginning or end of block B, respectively, e is computed, and after the last computation, no argument of e could be redefined.

! **Exercise 9.2.10:** The astute reader will notice that in Algorithm 9.11 we could have saved some time by initializing OUT[B] to gen_B for all blocks B. Likewise, in Algorithm 9.14 we could have initialized IN[B] to gen_B. We did not do so for uniformity in the treatment of the subject, as we shall see in Algorithm 9.25. However, is it possible to initialize OUT[B] to e_gen_B in Algorithm 9.17? Why or why not?

! **Exercise 9.2.11:** Our data-flow analyses so far do not take advantage of the semantics of conditionals. Suppose we find at the end of a basic block a test such as

```
if (x < 10) goto ...
```

How could we use our understanding of what the test $x < 10$ means to improve our knowledge of reaching definitions? Remember, "improve" here means that we eliminate certain reaching definitions that really cannot ever reach a certain program point.

9.3 Foundations of Data-Flow Analysis

Having shown several useful examples of the data-flow abstraction, we now study the family of data-flow schemas as a whole, abstractly. We shall answer several basic questions about data-flow algorithms formally:

1. Under what circumstances is the iterative algorithm used in data-flow analysis correct?
2. How precise is the solution obtained by the iterative algorithm?
3. Will the iterative algorithm converge?
4. What is the meaning of the solution to the equations?

In Section 9.2, we addressed each of the questions above informally when describing the reaching-definitions problem. Instead of answering the same questions for each subsequent problem from scratch, we relied on analogies with the problems we had already discussed to explain the new problems. Here we present a general approach that answers all these questions, once and for all, rigorously, and for a large family of data-flow problems. We first identify the properties desired of data-flow schemas and prove the implications of these properties on the correctness, precision, and convergence of the data-flow algorithm, as well as the meaning of the solution. Thus, to understand old algorithms or formulate new ones, we simply show that the proposed data-flow problem definitions have certain properties, and the answers to all the above difficult questions are available immediately.

The concept of having a common theoretical framework for a class of schemas also has practical implications. The framework helps us identify the reusable components of the algorithm in our software design. Not only is coding effort reduced, but programming errors are reduced by not having to recode similar details several times.

A *data-flow analysis framework* $(D, V, \wedge, F)$ consists of

1. A direction of the data flow D, which is either FORWARDS or BACKWARDS.
2. A semilattice (see Section 9.3.1 for the definition), which includes a *domain* of values V and a *meet operator* $\wedge$.
3. A family F of transfer functions from V to V. This family must include functions suitable for the boundary conditions, which are constant transfer functions for the special nodes ENTRY and EXIT in any flow graph.

9.3.1 Semilattices

A *semilattice* is a set V and a binary meet operator $\wedge$ such that for all x, y, and z in V:

1. $x \wedge x = x$ (meet is *idempotent*).
2. $x \wedge y = y \wedge x$ (meet is *commutative*).
3. $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ (meet is *associative*).

A semilattice has a *top* element, denoted $\top$, such that

$$\text{for all } x \text{ in } V,\ \top \wedge x = x.$$

Optionally, a semilattice may have a *bottom* element, denoted $\bot$, such that

$$\text{for all } x \text{ in } V,\ \bot \wedge x = \bot.$$

Partial Orders

As we shall see, the meet operator of a semilattice defines a partial order on the values of the domain. A relation $\leq$ is a *partial order* on a set V if for all x, y, and z in V:

1. $x \leq x$ (the partial order is *reflexive*).
2. If $x \leq y$ and $y \leq x$, then $x = y$ (the partial order is *antisymmetric*).
3. If $x \leq y$ and $y \leq z$, then $x \leq z$ (the partial order is *transitive*).

The pair $(V, \leq)$ is called a *poset*, or *partially ordered set*. It is also convenient to have a $<$ relation for a poset, defined as

$$x < y \text{ if and only if } (x \leq y) \text{ and } (x \neq y).$$

The Partial Order for a Semilattice

It is useful to define a partial order $\leq$ for a semilattice $(V, \wedge)$. For all x and y in V, we define

$$x \leq y \text{ if and only if } x \wedge y = x.$$

Because the meet operator $\wedge$ is idempotent, commutative, and associative, the $\leq$ order as defined is reflexive, antisymmetric, and transitive. To see why, observe that:

- Reflexivity: for all x, $x \leq x$. The proof is that $x \wedge x = x$ since meet is idempotent.
- Antisymmetry: if $x \leq y$ and $y \leq x$, then $x = y$. In proof, $x \leq y$ means $x \wedge y = x$ and $y \leq x$ means $y \wedge x = y$. By commutativity of $\wedge$, $x = (x \wedge y) = (y \wedge x) = y$.

- Transitivity: if $x \leq y$ and $y \leq z$, then $x \leq z$. In proof, $x \leq y$ and $y \leq z$ means that $x \wedge y = x$ and $y \wedge z = y$. Then $(x \wedge z) = \big((x \wedge y) \wedge z\big) = \big(x \wedge (y \wedge z)\big) = (x \wedge y) = x$, using associativity of meet. Since $x \wedge z = x$ has been shown, we have $x \leq z$, proving transitivity.

Example 9.18: The meet operators used in the examples in Section 9.2 are set union and set intersection. They are both idempotent, commutative, and associative. For set union, the top element is $\emptyset$ and the bottom element is U, the universal set, since for any subset x of U, $\emptyset \cup x = x$ and $U \cup x = U$. For set intersection, $\top$ is U and $\bot$ is $\emptyset$. V, the domain of values of the semilattice, is the set of all subsets of U, which is sometimes called the *power set* of U and denoted 2^U.

For all x and y in V, $x \cup y = x$ implies $x \supseteq y$; therefore, the partial order imposed by set union is $\supseteq$, set inclusion. Correspondingly, the partial order imposed by set intersection is $\subseteq$, set containment. That is, for set intersection, sets with fewer elements are considered to be smaller in the partial order. However, for set union, sets with *more* elements are considered to be smaller in the partial order. To say that sets larger in size are smaller in the partial order is counterintuitive; however, this situation is an unavoidable consequence of the definitions.[6]

As discussed in Section 9.2, there are usually many solutions to a set of data-flow equations, with the greatest solution (in the sense of the partial order $\leq$) being the most precise. For example, in reaching definitions, the most precise among all the solutions to the data-flow equations is the one with the smallest number of definitions, which corresponds to the greatest element in the partial order defined by the meet operation, union. In available expressions, the most precise solution is the one with the largest number of expressions. Again, it is the greatest solution in the partial order defined by intersection as the meet operation. □

Greatest Lower Bounds

There is another useful relationship between the meet operation and the partial ordering it imposes. Suppose $(V, \wedge)$ is a semilattice. A *greatest lower bound* (or *glb*) of domain elements x and y is an element g such that

1. $g \leq x$,
2. $g \leq y$, and
3. If z is any element such that $z \leq x$ and $z \leq y$, then $z \leq g$.

It turns out that the meet of x and y is their only greatest lower bound. To see why, let $g = x \wedge y$. Observe that:

[6] And if we defined the partial order to be $\geq$ instead of $\leq$, then the problem would surface when the meet was intersection, although not for union.

Joins, Lub's, and Lattices

In symmetry to the glb operation on elements of a poset, we may define the *least upper bound* (or *lub*) of elements x and y to be that element b such that $x \leq b$, $y \leq b$, and if z is any element such that $x \leq z$ and $y \leq z$, then $b \leq z$. One can show that there is at most one such element b if it exists.

In a true *lattice*, there are two operations on domain elements, the meet $\wedge$, which we have seen, and the operator *join*, denoted $\vee$, which gives the lub of two elements (which therefore must always exist in the lattice). We have been discussing only "semi" lattices, where only one of the meet and join operators exist. That is, our semilattices are *meet semilattices*. One could also speak of *join semilattices*, where only the join operator exists, and in fact some literature on program analysis does use the notation of join semilattices. Since the traditional data-flow literature speaks of meet semilattices, we shall also do so in this book.

- $g \leq x$ because $(x \wedge y) \wedge x = x \wedge y$. The proof involves simple uses of associativity, commutativity, and idempotence. That is,

$$\begin{aligned} g \wedge x &= \big((x \wedge y) \wedge x\big) = \big(x \wedge (y \wedge x)\big) = \\ &\big(x \wedge (x \wedge y)\big) = \big((x \wedge x) \wedge y\big) = \\ &(x \wedge y) = g \end{aligned}$$

- $g \leq y$ by a similar argument.
- Suppose z is any element such that $z \leq x$ and $z \leq y$. We claim $z \leq g$, and therefore, z cannot be a glb of x and y unless it is also g. In proof: $(z \wedge g) = \big(z \wedge (x \wedge y)\big) = \big((z \wedge x) \wedge y\big)$. Since $z \leq x$, we know $(z \wedge x) = z$, so $(z \wedge g) = (z \wedge y)$. Since $z \leq y$, we know $z \wedge y = z$, and therefore $z \wedge g = z$. We have proven $z \leq g$ and conclude $g = x \wedge y$ is the only glb of x and y.

Lattice Diagrams

It often helps to draw the domain V as a lattice diagram, which is a graph whose nodes are the elements of V, and whose edges are directed downward, from x to y if $y \leq x$. For example, Fig. 9.22 shows the set V for a reaching-definitions data-flow schema where there are three definitions: d_1, d_2, and d_3. Since $\leq$ is $\supseteq$, an edge is directed downward from any subset of these three definitions to each of its supersets. Since $\leq$ is transitive, we conventionally omit the edge from x

to y as long as there is another path from x to y left in the diagram. Thus, although $\{d_1, d_2, d_3\} \leq \{d_1\}$, we do not draw this edge since it is represented by the path through $\{d_1, d_2\}$, for example.

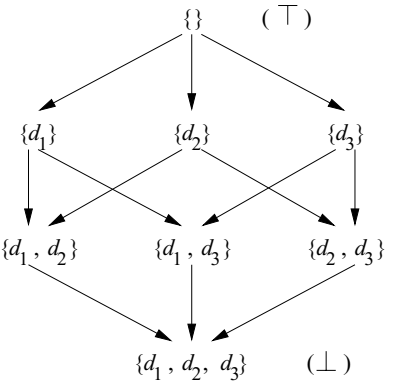


Figure 9.22: Lattice of subsets of definitions

It is also useful to note that we can read the meet off such diagrams. Since $x \wedge y$ is the glb, it is always the highest z for which there are paths downward to z from both x and y. For example, if x is $\{d_1\}$ and y is $\{d_2\}$, then z in Fig. 9.22 is $\{d_1, d_2\}$, which makes sense, because the meet operator is union. The top element will appear at the top of the lattice diagram; that is, there is a path downward from $\top$ to each element. Likewise, the bottom element will appear at the bottom, with a path downward from every element to $\bot$.

Product Lattices

While Fig. 9.22 involves only three definitions, the lattice diagram of a typical program can be quite large. The set of data-flow values is the power set of the definitions, which therefore contains 2^n elements if there are n definitions in the program. However, whether a definition reaches a program is independent of the reachability of the other definitions. We may thus express the lattice[7] of definitions in terms of a "product lattice," built from one simple lattice for each definition. That is, if there were only one definition d in the program, then the lattice would have two elements: $\{\}$, the empty set, which is the top element, and $\{d\}$, which is the bottom element.

Formally, we may build product lattices as follows. Suppose $(A, \wedge_A)$ and $(B, \wedge_B)$ are (semi)lattices. The *product lattice* for these two lattices is defined as follows:

1. The domain of the product lattice is $A \times B$.

[7] In this discussion and subsequently, we shall often drop the "semi," since lattices like the one under discussion do have a join or lub operator, even if we do not make use of it.

2. The meet $\wedge$ for the product lattice is defined as follows. If (a, b) and (a', b') are domain elements of the product lattice, then

$$(a, b) \wedge (a', b') = (a \wedge_A a', b \wedge_B b'). \tag{9.19}$$

It is simple to express the $\leq$ partial order for the product lattice in terms of the partial orders $\leq_A$ and $\leq_B$ for A and B

$$(a, b) \leq (a', b') \text{ if and only if } a \leq_A a' \text{ and } b \leq_B b'. \tag{9.20}$$

To see why (9.20) follows from (9.19), observe that

$$(a, b) \wedge (a', b') = (a \wedge_A a', b \wedge_B b').$$

So we might ask under what circumstances does $(a \wedge_A a', b \wedge_B b') = (a, b)$? That happens exactly when $a \wedge_A a' = a$ and $b \wedge_B b' = b$. But these two conditions are the same as $a \leq_A a'$ and $b \leq_B b'$.

The product of lattices is an associative operation, so one can show that the rules (9.19) and (9.20) extend to any number of lattices. That is, if we are given lattices $(A_i, \wedge_i)$ for $i = 1, 2, \ldots, k$, then the product of all k lattices, in this order, has domain $A_1 \times A_2 \times \cdots \times A_k$, a meet operator defined by

$$(a_1, a_2, \ldots, a_k) \wedge (b_1, b_2, \ldots, b_k) = (a_1 \wedge_1 b_1, a_2 \wedge_2 b_2, \ldots, a_k \wedge_k b_k)$$

and a partial order defined by

$$(a_1, a_2, \ldots, a_k) \leq (b_1, b_2, \ldots, b_k) \text{ if and only if } a_i \leq b_i \text{ for all } i.$$

Height of a Semilattice

We may learn something about the rate of convergence of a data-flow analysis algorithm by studying the "height" of the associated semilattice. An *ascending chain* in a poset $(V, \leq)$ is a sequence where $x_1 < x_2 < \ldots < x_n$. The *height* of a semilattice is the largest number of $<$ relations in any ascending chain; that is, the height is one less than the number of elements in the chain. For example, the height of the reaching definitions semilattice for a program with n definitions is n.

Showing convergence of an iterative data-flow algorithm is much easier if the semilattice has finite height. Clearly, a lattice consisting of a finite set of values will have a finite height; it is also possible for a lattice with an infinite number of values to have a finite height. The lattice used in the constant propagation algorithm is one such example that we shall examine closely in Section 9.4.

9.3.2 Transfer Functions

The family of transfer functions $F : V \to V$ in a data-flow framework has the following properties:

1. F has an identity function I, such that $I(x) = x$ for all x in V.

2. F is closed under composition; that is, for any two functions f and g in F, the function h defined by $h(x) = g\big(f(x)\big)$ is in F.

Example 9.21 : In reaching definitions, F has the identity, the function where *gen* and *kill* are both the empty set. Closure under composition was actually shown in Section 9.2.4; we repeat the argument succinctly here. Suppose we have two functions

$$f_1(x) = G_1 \cup (x - K_1) \text{ and } f_2(x) = G_2 \cup (x - K_2).$$

Then

$$f_2\big(f_1(x)\big) = G_2 \cup \Big(\big(G_1 \cup (x - K_1)\big) - K_2\Big).$$

The right side of the above is algebraically equivalent to

$$\big(G_2 \cup (G_1 - K_2)\big) \cup \big(x - (K_1 \cup K_2)\big).$$

If we let $K = K_1 \cup K_2$ and $G = G_2 \cup (G_1 - K_2)$, then we have shown that the composition of f_1 and f_2, which is $f(x) = G \cup (x - K)$, is of the form that makes it a member of F. If we consider available expressions, the same arguments used for reaching definitions also show that F has an identity and is closed under composition. □

Monotone Frameworks

To make an iterative algorithm for data-flow analysis work, we need for the data-flow framework to satisfy one more condition. We say that a framework is *monotone* if when we apply any transfer function f in F to two members of V, the first being no greater than the second, then the first result is no greater than the second result.

Formally, a data-flow framework $(D, F, V, \wedge)$ is *monotone* if

$$\text{For all } x \text{ and } y \text{ in } V \text{ and } f \text{ in } F,\ x \leq y \text{ implies } f(x) \leq f(y). \tag{9.22}$$

Equivalently, monotonicity can be defined as

$$\text{For all } x \text{ and } y \text{ in } V \text{ and } f \text{ in } F,\ f(x \wedge y) \leq f(x) \wedge f(y). \tag{9.23}$$

Equation (9.23) says that if we take the meet of two values and then apply f, the result is never greater than what is obtained by applying f to the values individually first and then "meeting" the results. Because the two definitions of monotonicity seem so different, they are both useful. We shall find one or the other more useful under different circumstances. Later, we sketch a proof to show that they are indeed equivalent.

We shall first assume (9.22) and show that (9.23) holds. Since $x \wedge y$ is the greatest lower bound of x and y, we know that

$$x \wedge y \leq x \text{ and } x \wedge y \leq y.$$

Thus, by (9.22),

$$f(x \wedge y) \leq f(x) \text{ and } f(x \wedge y) \leq f(y).$$

Since $f(x) \wedge f(y)$ is the greatest lower bound of $f(x)$ and $f(y)$, we have (9.23).

Conversely, let us assume (9.23) and prove (9.22). We suppose $x \leq y$ and use (9.23) to conclude $f(x) \leq f(y)$, thus proving (9.22). Equation (9.23) tells us

$$f(x \wedge y) \leq f(x) \wedge f(y).$$

But since $x \leq y$ is assumed, $x \wedge y = x$, by definition. Thus (9.23) says

$$f(x) \leq f(x) \wedge f(y).$$

Since $f(x) \wedge f(y)$ is the glb of $f(x)$ and $f(y)$, we know $f(x) \wedge f(y) \leq f(y)$. Thus

$$f(x) \leq f(x) \wedge f(y) \leq f(y)$$

and (9.23) implies (9.22).

Distributive Frameworks

Often, a framework obeys a condition stronger than (9.23), which we call the *distributivity condition*,

$$f(x \wedge y) = f(x) \wedge f(y)$$

for all x and y in V and f in F. Certainly, if $a = b$, then $a \wedge b = a$ by idempotence, so $a \leq b$. Thus, distributivity implies monotonicity, although the converse is not true.

Example 9.24: Let y and z be sets of definitions in the reaching-definitions framework. Let f be a function defined by $f(x) = G \cup (x - K)$ for some sets of definitions G and K. We can verify that the reaching-definitions framework satisfies the distributivity condition, by checking that

$$G \cup \big((y \cup z) - K\big) = \big(G \cup (y - K)\big) \cup \big(G \cup (z - K)\big).$$

While the equation above may appear formidable, consider first those definitions in G. These definitions are surely in the sets defined by both the left and right sides. Thus, we have only to consider definitions that are not in G. In that case, we can eliminate G everywhere, and verify the equality

$$(y \cup z) - K = (y - K) \cup (z - K).$$

The latter equality is easily checked using a Venn diagram. □

9.3.3 The Iterative Algorithm for General Frameworks

We can generalize Algorithm 9.11 to make it work for a large variety of data-flow problems.

Algorithm 9.25: Iterative solution to general data-flow frameworks.

INPUT: A data-flow framework with the following components:

1. A data-flow graph, with specially labeled ENTRY and EXIT nodes,
2. A direction of the data-flow D,
3. A set of values V,
4. A meet operator $\wedge$,
5. A set of functions F, where f_B in F is the transfer function for block B, and
6. A constant value v_{ENTRY} or v_{EXIT} in V, representing the boundary condition for forward and backward frameworks, respectively.

OUTPUT: Values in V for IN$[B]$ and OUT$[B]$ for each block B in the data-flow graph.

METHOD: The algorithms for solving forward and backward data-flow problems are shown in Fig. 9.23(a) and 9.23(b), respectively. As with the familiar iterative data-flow algorithms from Section 9.2, we compute IN and OUT for each block by successive approximation. □

It is possible to write the forward and backward versions of Algorithm 9.25 so that a function implementing the meet operation is a parameter, as is a function that implements the transfer function for each block. The flow graph itself and the boundary value are also parameters. In this way, the compiler implementor can avoid recoding the basic iterative algorithm for each data-flow framework used by the optimization phase of the compiler.

We can use the abstract framework discussed so far to prove a number of useful properties of the iterative algorithm:

1. If Algorithm 9.25 converges, the result is a solution to the data-flow equations.
2. If the framework is monotone, then the solution found is the maximum fixedpoint (MFP) of the data-flow equations. A *maximum fixedpoint* is a solution with the property that in any other solution, the values of IN$[B]$ and OUT$[B]$ are $\leq$ the corresponding values of the MFP.
3. If the semilattice of the framework is monotone and of finite height, then the algorithm is guaranteed to converge.

```
1)  OUT[ENTRY] = v_ENTRY;
2)  for (each basic block B other than ENTRY) OUT[B] = ⊤;
3)  while (changes to any OUT occur)
4)          for (each basic block B other than ENTRY) {
5)                  IN[B] = ⋀_{P a predecessor of B} OUT[P];
6)                  OUT[B] = f_B(IN[B]);
            }
```

(a) Iterative algorithm for a forward data-flow problem.

```
1)  IN[EXIT] = v_EXIT;
2)  for (each basic block B other than EXIT) IN[B] = ⊤;
3)  while (changes to any IN occur)
4)          for (each basic block B other than EXIT) {
5)                  OUT[B] = ⋀_{S a successor of B} IN[S];
6)                  IN[B] = f_B(OUT[B]);
            }
```

(b) Iterative algorithm for a backward data-flow problem.

Figure 9.23: Forward and backward versions of the iterative algorithm

We shall argue these points assuming that the framework is forward. The case of backwards frameworks is essentially the same. The first property is easy to show. If the equations are not satisfied by the time the while-loop ends, then there will be at least one change to an OUT (in the forward case) or IN (in the backward case), and we must go around the loop again.

To prove the second property, we first show that the values taken on by $\text{IN}[B]$ and $\text{OUT}[B]$ for any B can only decrease (in the sense of the $\leq$ relationship for lattices) as the algorithm iterates. This claim can be proven by induction.

BASIS: The base case is to show that the value of $\text{IN}[B]$ and $\text{OUT}[B]$ after the first iteration is not greater than the initialized value. This statement is trivial because $\text{IN}[B]$ and $\text{OUT}[B]$ for all blocks $B \neq \text{ENTRY}$ are initialized with $\top$.

INDUCTION: Assume that after the kth iteration, the values are all no greater than those after the $(k-1)$st iteration, and show the same for iteration $k+1$ compared with iteration k. Line (5) of Fig. 9.23(a) has

$$\text{IN}[B] = \bigwedge_{P \text{ a predecessor of } B} \text{OUT}[P].$$

Let us use the notation $\text{IN}[B]^i$ and $\text{OUT}[B]^i$ to denote the values of $\text{IN}[B]$ and $\text{OUT}[B]$ after iteration i. Assuming $\text{OUT}[P]^k \leq \text{OUT}[P]^{k-1}$, we know that $\text{IN}[B]^{k+1} \leq \text{IN}[B]^k$ because of the properties of the meet operator. Next, line (6)

says

$$\text{OUT}[B] = f_B(\text{IN}[B]).$$

Since $\text{IN}[B]^{k+1} \leq \text{IN}[B]^k$, we have $\text{OUT}[B]^{k+1} \leq \text{OUT}[B]^k$ by monotonicity.

Note that every change observed for values of $\text{IN}[B]$ and $\text{OUT}[B]$ is necessary to satisfy the equation. The meet operators return the greatest lower bound of their inputs, and the transfer functions return the only solution that is consistent with the block itself and its given input. Thus, if the iterative algorithm terminates, the result must have values that are at least as great as the corresponding values in any other solution; that is, the result of Algorithm 9.25 is the MFP of the equations.

Finally, consider the third point, where the data-flow framework has finite height. Since the values of every $\text{IN}[B]$ and $\text{OUT}[B]$ decrease with each change, and the algorithm stops if at some round nothing changes, the algorithm is guaranteed to converge after a number of rounds no greater than the product of the height of the framework and the number of nodes of the flow graph.

9.3.4 Meaning of a Data-Flow Solution

We now know that the solution found using the iterative algorithm is the maximum fixedpoint, but what does the result represent from a program-semantics point of view? To understand the solution of a data-flow framework $(D, F, V, \wedge)$, let us first describe what an ideal solution to the framework would be. We show that the ideal cannot be obtained in general, but that Algorithm 9.25 approximates the ideal conservatively.

The Ideal Solution

Without loss of generality, we shall assume for now that the data-flow framework of interest is a forward-flowing problem. Consider the entry point of a basic block B. The ideal solution begins by finding all the *possible* execution paths leading from the program entry to the beginning of B. A path is "possible" only if there is some computation of the program that follows exactly that path. The ideal solution would then compute the data-flow value at the end of each possible path and apply the meet operator to these values to find their greatest lower bound. Then no execution of the program can produce a smaller value for that program point. In addition, the bound is tight; there is no greater data-flow value that is a glb for the value computed along every possible path to B in the flow graph.

We now try to define the ideal solution more formally. For each block B in a flow graph, let f_B be the transfer function for B. Consider any path

$$P = \text{ENTRY} \rightarrow B_1 \rightarrow B_2 \rightarrow \ldots \rightarrow B_{k-1} \rightarrow B_k$$

from the initial node ENTRY to some block B_k. The program path may have cycles, so one basic block may appear several times on the path P. Define the

transfer function for P, f_P, to be the composition of $f_{B_1}, f_{B_2} \cdots, f_{B_{k-1}}$. Note that f_{B_k} is not part of the composition, reflecting the fact that this path is taken to reach the beginning of block B_k, not its end. The data-flow value created by executing this path is thus $f_P(v_{\text{ENTRY}})$, where v_{ENTRY} is the result of the constant transfer function representing the initial node ENTRY. The ideal result for block B is thus

$$\text{IDEAL}[B] = \bigwedge_{P, \text{ a possible path from ENTRY to } B} f_P(v_{\text{ENTRY}}).$$

We claim that, in terms of the lattice-theoretic partial order $\leq$ for the framework in question,

- Any answer that is greater than IDEAL is incorrect.
- Any value smaller than or equal to the ideal is conservative, i.e., safe.

Intuitively, the closer the value to the ideal the more precise it is.[8] To see why solutions must be $\leq$ the ideal solution, note that any solution greater than IDEAL for any block could be obtained by ignoring some execution path that the program could take, and we cannot be sure that there is not some effect along that path to invalidate any program improvement we might make based on the greater solution. Conversely, any solution less than IDEAL can be viewed as including certain paths that either do not exist in the flow graph, or that exist but that the program can never follow. This lesser solution will allow only transformations that are correct for all possible executions of the program, but may forbid some transformations that IDEAL would permit.

The Meet-Over-Paths Solution

However, as discussed in Section 9.1, finding all possible execution paths is undecidable. We must therefore approximate. In the data-flow abstraction, we assume that every path in the flow graph can be taken. Thus, we can define the meet-over-paths solution for B to be

$$\text{MOP}[B] = \bigwedge_{P, \text{ a path from ENTRY to } B} f_P(v_{\text{ENTRY}}).$$

Note that, as for IDEAL, the solution MOP$[B]$ gives values for IN$[B]$ in forward-flow frameworks. If we were to consider backward-flow frameworks, then we would think of MOP$[B]$ as a value for OUT$[B]$.

The paths considered in the MOP solution are a superset of all the paths that are possibly executed. Thus, the MOP solution meets together not only the data-flow values of all the executable paths, but also additional values associated

[8] Note that in forward problems, the value IDEAL$[B]$ is what we would like IN$[B]$ to be. In backward problems, which we do not discuss here, we would define IDEAL$[B]$ to be the ideal value of OUT$[B]$.

with the paths that cannot possibly be executed. Taking the meet of the ideal solution plus additional terms cannot create a solution larger than the ideal. Thus, for all B we have $\text{MOP}[B] \leq \text{IDEAL}[B]$, and we will simply say that $\text{MOP} \leq \text{IDEAL}$.

The Maximum Fixedpoint Versus the MOP Solution

Notice that in the MOP solution, the number of paths considered is still unbounded if the flow graph contains cycles. Thus, the MOP definition does not lend itself to a direct algorithm. The iterative algorithm certainly does not first find all the paths leading to a basic block before applying the meet operator. Rather,

1. The iterative algorithm visits basic blocks, not necessarily in the order of execution.

2. At each confluence point, the algorithm applies the meet operator to the data-flow values obtained so far. Some of these values used were introduced artificially in the initialization process, not representing the result of any execution from the beginning of the program.

So what is the relationship between the MOP solution and the solution MFP produced by Algorithm 9.25?

We first discuss the order in which the nodes are visited. In an iteration, we may visit a basic block before having visited its predecessors. If the predecessor is the ENTRY node, OUT[ENTRY] would have already been initialized with the proper, constant value. Otherwise, it has been initialized to $\top$, a value no smaller than the final answer. By monotonicity, the result obtained by using $\top$ as input is no smaller than the desired solution. In a sense, we can think of $\top$ as representing no information.

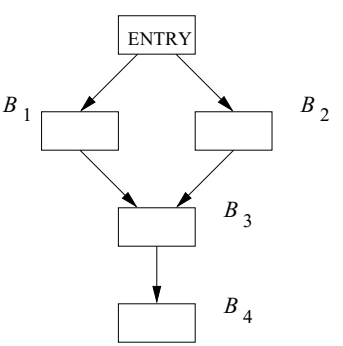


Figure 9.24: Flow graph illustrating the effect of early meet over paths

What is the effect of applying the meet operator early? Consider the simple example of Fig. 9.24, and suppose we are interested in the value of $\text{IN}[B_4]$. By

the definition of MOP,

$$\text{MOP}[B_4] = \big((f_{B_3} \circ f_{B_1}) \wedge (f_{B_3} \circ f_{B_2})\big)(v_{\text{ENTRY}})$$

In the iterative algorithm, if we visit the nodes in the order B_1, B_2, B_3, B_4, then

$$\text{IN}[B_4] = f_{B_3}\Big(\big(f_{B_1}(v_{\text{ENTRY}}) \wedge f_{B_2}(v_{\text{ENTRY}})\big)\Big)$$

While the meet operator is applied at the end in the definition of MOP, the iterative algorithm applies it early. The answer is the same only if the data-flow framework is distributive. If the data-flow framework is monotone but not distributive, we still have $\text{IN}[B_4] \leq \text{MOP}[B_4]$. Recall that in general a solution $\text{IN}[B]$ is safe (conservative) if $\text{IN}[B] \leq \text{IDEAL}[B]$ for all blocks B. Surely, $\text{MOP}[B] \leq \text{IDEAL}[B]$.

We now provide a quick sketch of why in general the MFP solution provided by the iterative algorithm is always safe. An easy induction on i shows that the values obtained after i iterations are smaller than or equal to the meet over all paths of length i or less. But the iterative algorithm terminates only if it arrives at the same answer as would be obtained by iterating an unbounded number of times. Thus, the result is no greater than the MOP solution. Since $\text{MOP} \leq \text{IDEAL}$ and $\text{MFP} \leq \text{MOP}$, we know that $\text{MFP} \leq \text{IDEAL}$, and therefore the solution MFP provided by the iterative algorithm is safe.

9.3.5 Exercises for Section 9.3

Exercise 9.3.1: Construct a lattice diagram for the product of three lattices, each based on a single definition d_i, for $i = 1, 2, 3$. How is your lattice diagram related to that in Fig. 9.22?

! **Exercise 9.3.2:** In Section 9.3.3 we argued that if the framework has finite height, then the iterative algorithm converges. Here is an example where the framework does not have finite height, and the iterative algorithm does not converge. Let the set of values V be the nonnegative real numbers, and let the meet operator be the minimum. There are three transfer functions:

i. The identity, $f_I(x) = x$.

ii. "half," that is, the function $f_H(x) = x/2$.

iii. "one." that is, the function $f_O(x) = 1$.

The set of transfer functions F is these three plus the functions formed by composing them in all possible ways.

a) Describe the set F.

b) What is the $\leq$ relationship for this framework?

c) Give an example of a flow graph with assigned transfer functions, such that Algorithm 9.25 does not converge.

d) Is this framework monotone? Is it distributive?

! **Exercise 9.3.3:** We argued that Algorithm 9.25 converges if the framework is monotone and of finite height. Here is an example of a framework that shows monotonicity is essential; finite height is not enough. The domain V is $\{1, 2\}$, the meet operator is min, and the set of functions F is only the identity (f_I) and the "switch" function ($f_S(x) = 3 - x$) that swaps 1 and 2.

a) Show that this framework is of finite height but not monotone.

b) Give an example of a flow graph and assignment of transfer functions so that Algorithm 9.25 does not converge.

! **Exercise 9.3.4:** Let $\text{MOP}_i[B]$ be the meet over all paths of length i or less from the entry to block B. Prove that after i iterations of Algorithm 9.25, $\text{IN}[B] \leq \text{MOP}_i[B]$. Also, show that as a consequence, if Algorithm 9.25 converges, then it converges to something that is $\leq$ the MOP solution.

! **Exercise 9.3.5:** Suppose the set F of functions for a framework are all of gen-kill form. That is, the domain V is the power set of some set, and $f(x) = G \cup (x - K)$ for some sets G and K. Prove that if the meet operator is either (a) union or (b) intersection, then the framework is distributive.

9.4 Constant Propagation

All the data-flow schemas discussed in Section 9.2 are actually simple examples of distributive frameworks with finite height. Thus, the iterative Algorithm 9.25 applies to them in either its forward or backward version and produces the MOP solution in each case. In this section, we shall examine in detail a useful data-flow framework with more interesting properties.

Recall that constant propagation, or "constant folding," replaces expressions that evaluate to the same constant every time they are executed, by that constant. The constant-propagation framework described below is different from all the data-flow problems discussed so far, in that

a) it has an unbounded set of possible data-flow values, even for a fixed flow graph, and

b) it is not distributive.

Constant propagation is a forward data-flow problem. The semilattice representing the data-flow values and the family of transfer functions are presented next.

9.4.1 Data-Flow Values for the Constant-Propagation Framework

The set of data-flow values is a product lattice, with one component for each variable in a program. The lattice for a single variable consists of the following:

1. All constants appropriate for the type of the variable.
2. The value NAC, which stands for not-a-constant. A variable is mapped to this value if it is determined not to have a constant value. The variable may have been assigned an input value, or derived from a variable that is not a constant, or assigned different constants along different paths that lead to the same program point.
3. The value UNDEF, which stands for undefined. A variable is assigned this value if nothing may yet be asserted; presumably, no definition of the variable has been discovered to reach the point in question.

Note that NAC and UNDEF are not the same; they are essentially opposites. NAC says we have seen so many ways a variable could be defined that we know it is not constant; UNDEF says we have seen so little about the variable that we cannot say anything at all.

The semilattice for a typical integer-valued variable is shown in Fig. 9.25. Here the top element is UNDEF, and the bottom element is NAC. That is, the greatest value in the partial order is UNDEF and the least is NAC. The constant values are unordered, but they are all less than UNDEF and greater than NAC. As discussed in Section 9.3.1, the meet of two values is their greatest lower bound. Thus, for all values v,

$$\text{UNDEF} \wedge v = v \text{ and } \text{NAC} \wedge v = \text{NAC}.$$

For any constant c,

$$c \wedge c = c$$

and given two distinct constants c_1 and c_2,

$$c_1 \wedge c_2 = \text{NAC}.$$

A data-flow value for this framework is a map from each variable in the program to one of the values in the constant semilattice. The value of a variable v in a map m is denoted by $m(v)$.

9.4.2 The Meet for the Constant-Propagation Framework

The semilattice of data-flow values is simply the product of the semilattices like Fig. 9.25, one for each variable. Thus, $m \leq m'$ if and only if for all variables v we have $m(v) \leq m'(v)$. Put another way, $m \wedge m' = m''$ if $m''(v) = m(v) \wedge m'(v)$ for all variables v.

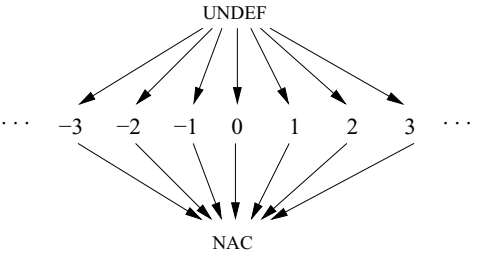


Figure 9.25: Semilattice representing the possible "values" of a single integer variable

9.4.3 Transfer Functions for the Constant-Propagation Framework

We assume in the following that a basic block contains only one statement. Transfer functions for basic blocks containing several statements can be constructed by composing the functions corresponding to individual statements. The set F consists of certain transfer functions that accept a map of variables to values in the constant lattice and return another such map.

F contains the identity function, which takes a map as input and returns the same map as output. F also contains the constant transfer function for the ENTRY node. This transfer function, given any input map, returns a map m_0, where $m_0(v) = \text{UNDEF}$, for all variables v. This boundary condition makes sense, because before executing any program statements there are no definitions for any variables.

In general, let f_s be the transfer function of statement s, and let m and m' represent data-flow values such that $m' = f_s(m)$. We shall describe f_s in terms of the relationship between m and m'.

1. If s is not an assignment statement, then f_s is simply the identity function.
2. If s is an assignment to variable x, then $m'(v) = m(v)$, for all variables $v \neq x$, and $m'(x)$ is defined as follows:
   (a) If the right-hand-side (RHS) of the statement s is a constant c, then $m'(x) = c$.
   (b) If the RHS is of the form $y + z$, then[9]

$$m'(x) = \begin{cases} m(y) + m(z) & \text{if } m(y) \text{ and } m(z) \text{ are constant values} \\ \text{NAC} & \text{if either } m(y) \text{ or } m(z) \text{ is NAC} \\ \text{UNDEF} & \text{otherwise} \end{cases}$$

   (c) If the RHS is any other expression (e.g. a function call or assignment through a pointer), then $m'(x) = \text{NAC}$.

[9] As usual, + represents a generic operator, not necessarily addition.

9.4.4 Monotonicity of the Constant-Propagation Framework

Let us show that the constant propagation framework is monotone. First, we can consider the effect of a function f_s on a single variable. In all but case 2(b), f_s either does not change the value of $m(x)$, or it changes the map to return a constant or NAC. In these cases, f_s must surely be monotone.

For case 2(b), the effect of f_s is tabulated in Fig 9.26. The first and second columns represent the possible input values of y and z; the last represents the output value of x. The values are ordered from the greatest to the smallest in each column or subcolumn. To show that the function is monotone, we check that for each possible input value of y, the value of x does not get bigger as the value of z gets smaller. For example, in the case where y has a constant value c_1, as the value of z varies from UNDEF to c_2 to NAC, the value of x varies from UNDEF, to $c_1 + c_2$, and then to NAC, respectively. We can repeat this procedure for all the possible values of y. Because of symmetry, we do not even need to repeat the procedure for the second operand before we conclude that the output value cannot get larger as the input gets smaller.

$m(y)$	$m(z)$	$m'(x)$
UNDEF	UNDEF	UNDEF
	c_2	UNDEF
	NAC	NAC
c_1	UNDEF	UNDEF
	c_2	$c_1 + c_2$
	NAC	NAC
NAC	UNDEF	NAC
	c_2	NAC
	NAC	NAC

Figure 9.26: The constant-propagation transfer function for `x = y+z`

9.4.5 Nondistributivity of the Constant-Propagation Framework

The constant-propagation framework as defined is monotone but not distributive. That is, the iterative solution MFP is safe but may be smaller than the MOP solution. An example will prove that the framework is not distributive.

Example 9.26 : In the program in Fig. 9.27, x and y are set to 2 and 3 in block B_1, and to 3 and 2, respectively, in block B_2. We know that regardless of which path is taken, the value of z at the end of block B_3 is 5. The iterative algorithm does not discover this fact, however. Rather, it applies the meet operator at the entry of B_3, getting NAC's as the values of x and y. Since adding two NAC's

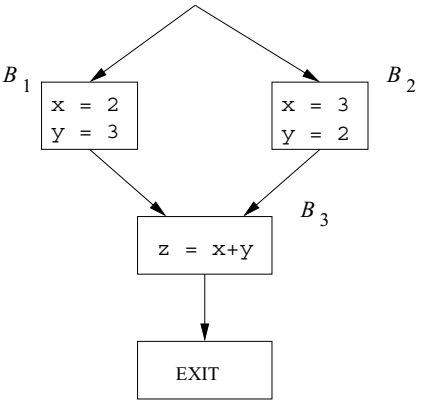


Figure 9.27: An example demonstrating that the constant propagation framework is not distributive

yields a NAC, the output produced by Algorithm 9.25 is that $z =$ NAC at the exit of the program. This result is safe, but imprecise. Algorithm 9.25 is imprecise because it does not keep track of the correlation that whenever x is 2, y is 3, and vice versa. It is possible, but significantly more expensive, to use a more complex framework that tracks all the possible equalities that hold among pairs of expressions involving the variables in the program; this approach is discussed in Exercise 9.4.2.

Theoretically, we can attribute this loss of precision to the nondistributivity of the constant propagation framework. Let f_1, f_2, and f_3 be the transfer functions representing blocks B_1, B_2 and B_3, respectively. As shown in Fig 9.28,

$$f_3\big(f_1(m_0) \wedge f_2(m_0)\big) < f_3\big(f_1(m_0)\big) \wedge f_3\big(f_2(m_0)\big)$$

rendering the framework nondistributive. □

m	$m(x)$	$m(y)$	$m(z)$
m_0	UNDEF	UNDEF	UNDEF
$f_1(m_0)$	2	3	UNDEF
$f_2(m_0)$	3	2	UNDEF
$f_1(m_0) \wedge f_2(m_0)$	NAC	NAC	UNDEF
$f_3(f_1(m_0) \wedge f_2(m_0))$	NAC	NAC	NAC
$f_3(f_1(m_0))$	2	3	5
$f_3(f_2(m_0))$	3	2	5
$f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$	NAC	NAC	5

Figure 9.28: Example of nondistributive transfer functions

9.4.6 Interpretation of the Results

The value UNDEF is used in the iterative algorithm for two purposes: to initialize the ENTRY node and to initialize the interior points of the program before the iterations. The meaning is slightly different in the two cases. The first says that variables are undefined at the beginning of the program execution; the second says that for lack of information at the beginning of the iterative process, we approximate the solution with the top element UNDEF. At the end of the iterative process, the variables at the exit of the ENTRY node will still hold the UNDEF value, since OUT[ENTRY] never changes.

It is possible that UNDEF's may show up at some other program points. When they do, it means that no definitions have been observed for that variable along any of the paths leading up to that program point. Notice that with the way we define the meet operator, as long as there exists a path that defines a variable reaching a program point, the variable will not have an UNDEF value. If all the definitions reaching a program point have the same constant value, the variable is considered a constant even though it may not be defined along some program path.

By assuming that the program is correct, the algorithm can find more constants than it otherwise would. That is, the algorithm conveniently chooses some values for those possibly undefined variables in order to make the program more efficient. This change is legal in most programming languages, since undefined variables are allowed to take on any value. If the language semantics requires that all undefined variables be given some specific value, then we must change our problem formulation accordingly. And if instead we are interested in finding possibly undefined variables in a program, we can formulate a different data-flow analysis to provide that result (see Exercise 9.4.1).

Example 9.27: In Fig. 9.29, the values of x are 10 and UNDEF at the exit of basic blocks B_2 and B_3, respectively. Since UNDEF $\wedge$ 10 = 10, the value of x is 10 on entry to block B_4. Thus, block B_5, where x is used, can be optimized by replacing x by 10. Had the path executed been $B_1 \rightarrow B_3 \rightarrow B_4 \rightarrow B_5$, the value of x reaching basic block B_5 would have been undefined. So, it appears incorrect to replace the use of x by 10.

However, if it is impossible for predicate Q to be false while Q' is true, then this execution path never occurs. While the programmer may be aware of that fact, it may well be beyond the capability of any data-flow analysis to determine. Thus, if we assume that the program is correct and that all the variables are defined before they are used, it is indeed correct that the value of x at the beginning of basic block B_5 can only be 10. And if the program is incorrect to begin with, then choosing 10 as the value of x cannot be worse than allowing x to assume some random value. □

9.4.7 Exercises for Section 9.4

! **Exercise 9.4.1:** Suppose we wish to detect all possibility of a variable being

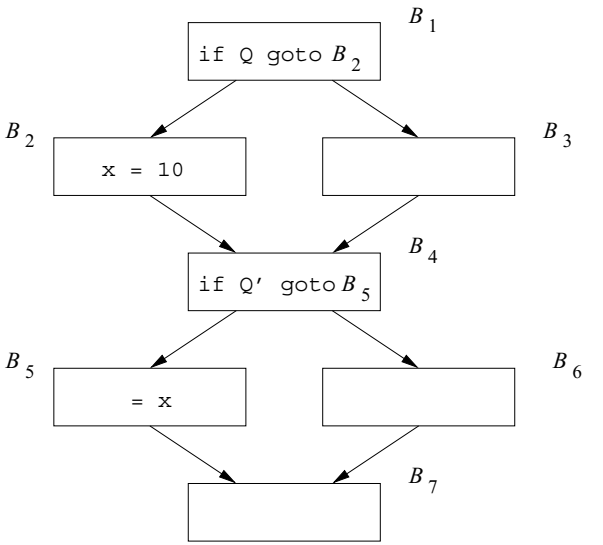


Figure 9.29: Meet of UNDEF and a constant

uninitialized along any path to a point where it is used. How would you modify the framework of this section to detect such situations?

!! **Exercise 9.4.2:** An interesting and powerful data-flow-analysis framework is obtained by imagining the domain V to be all possible partitions of expressions, so that two expressions are in the same class if and only if they are certain to have the same value along any path to the point in question. To avoid having to list an infinity of expressions, we can represent V by listing only the minimal pairs of equivalent expressions. For example, if we execute the statements

```
a = b
c = a + d
```

then the minimal set of equivalences is $\{a \equiv b, c \equiv a + d\}$. From these follow other equivalences, such as $c \equiv b + d$ and $a + e \equiv b + e$, but there is no need to list these explicitly.

a) What is the appropriate meet operator for this framework?

b) Give a data structure to represent domain values and an algorithm to implement the meet operator.

c) What are the appropriate functions to associate with statements? Explain the effect that a statement such as `a = b+c` should have on a partition of expressions (i.e., on a value in V).

d) Is this framework monotone? Distributive?

9.5 Partial-Redundancy Elimination

In this section, we consider in detail how to minimize the number of expression evaluations. That is, we want to consider all possible execution sequences in a flow graph, and look at the number of times an expression such as $x + y$ is evaluated. By moving around the places where $x + y$ is evaluated and keeping the result in a temporary variable when necessary, we often can reduce the number of evaluations of this expression along many of the execution paths, while not increasing that number along any path. Note that the number of different places in the flow graph where $x + y$ is evaluated may increase, but that is relatively unimportant, as long as the number of *evaluations* of the expression $x + y$ is reduced.

Applying the code transformation developed here improves the performance of the resulting code, since, as we shall see, an operation is never applied unless it absolutely has to be. Every optimizing compiler implements something like the transformation described here, even if it uses a less "aggressive" algorithm than the one of this section. However, there is another motivation for discussing the problem. Finding the right place or places in the flow graph at which to evaluate each expression requires four different kinds of data-flow analyses. Thus, the study of "partial-redundancy elimination," as minimizing the number of expression evaluations is called, will enhance our understanding of the role data-flow analysis plays in a compiler.

Redundancy in programs exists in several forms. As discussed in Section 9.1.4, it may exist in the form of common subexpressions, where several evaluations of the expression produce the same value. It may also exist in the form of a loop-invariant expression that evaluates to the same value in every iteration of the loop. Redundancy may also be partial, if it is found along some of the paths, but not necessarily along *all* paths. Common subexpressions and loop-invariant expressions can be viewed as special cases of partial redundancy; thus a single partial-redundancy-elimination algorithm can be devised to eliminate all the various forms of redundancy.

In the following, we first discuss the different forms of redundancy, in order to build up our intuition about the problem. We then describe the generalized redundancy-elimination problem, and finally we present the algorithm. This algorithm is particularly interesting, because it involves solving multiple data-flow problems, in both the forward and backward directions.

9.5.1 The Sources of Redundancy

Figure 9.30 illustrates the three forms of redundancy: common subexpressions, loop-invariant expressions, and partially redundant expressions. The figure shows the code both before and after each optimization.

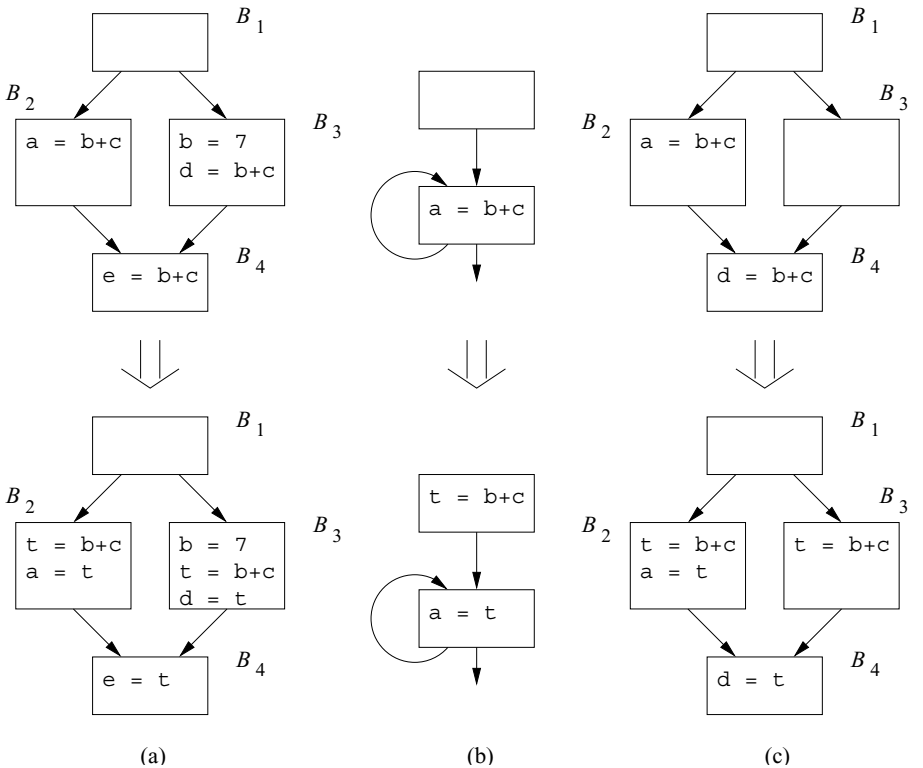


Figure 9.30: Examples of (a) global common subexpression, (b) loop-invariant code motion, (c) partial-redundancy elimination.

Global Common Subexpressions

In Fig. 9.30(a), the expression $b + c$ computed in block B_4 is redundant; it has already been evaluated by the time the flow of control reaches B_4 regardless of the path taken to get there. As we observe in this example, the value of the expression may be different on different paths. We can optimize the code by storing the result of the computations of $b + c$ in blocks B_2 and B_3 in the same temporary variable, say t, and then assigning the value of t to the variable e in block B_4, instead of reevaluating the expression. Had there been an assignment to either b or c after the last computation of $b + c$ but before block B_4, the expression in block B_4 would not be redundant.

Formally, we say that an expression $b + c$ is (fully) *redundant* at point p, if it is an available expression, in the sense of Section 9.2.6, at that point. That is, the expression $b + c$ has been computed along all paths reaching p, and the variables b and c were not redefined after the last expression was evaluated. The latter condition is necessary, because even though the expression $b + c$ is textually executed before reaching the point p, the value of $b + c$ computed at

Finding "Deep" Common Subexpressions

Using available-expressions analysis to identify redundant expressions only works for expressions that are textually identical. For example, an application of common-subexpression elimination will recognize that `t1` in the code fragment

```
t1 = b + c; a = t1 + d;
```

has the same value as does `t2` in

```
t2 = b + c; e = t2 + d;
```

as long as the variables b and c have not been redefined in between. It does not, however, recognize that a and e are also the same. It is possible to find such "deep" common subexpressions by re-applying common subexpression elimination until no new common subexpressions are found on one round. It is also possible to use the framework of Exercise 9.4.2 to catch deep common subexpressions.

point p would have been different, because the operands might have changed.

Loop-Invariant Expressions

Fig. 9.30(b) shows an example of a loop-invariant expression. The expression $b + c$ is loop invariant assuming neither the variable b nor c is redefined within the loop. We can optimize the program by replacing all the re-executions in a loop by a single calculation outside the loop. We assign the computation to a temporary variable, say t, and then replace the expression in the loop by t. There is one more point we need to consider when performing "code motion" optimizations such as this. We should not execute any instruction that would not have executed without the optimization. For example, if it is possible to exit the loop without executing the loop-invariant instruction at all, then we should not move the instruction out of the loop. There are two reasons.

1. If the instruction raises an exception, then executing it may throw an exception that would not have happened in the original program.

2. When the loop exits early, the "optimized" program takes more time than the original program.

To ensure that loop-invariant expressions in while-loops can be optimized, compilers typically represent the statement

```
while c {
    S;
}
```

in the same way as the statement

```
if c {
    repeat
       S;
    until not c;
}
```

In this way, loop-invariant expressions can be placed just prior to the repeat-until construct.

Unlike common-subexpression elimination, where a redundant expression computation is simply dropped, loop-invariant-expression elimination requires an expression from inside the loop to move outside the loop. Thus, this optimization is generally known as "loop-invariant code motion." Loop-invariant code motion may need to be repeated, because once a variable is determined to to have a loop-invariant value, expressions using that variable may also become loop-invariant.

Partially Redundant Expressions

An example of a partially redundant expression is shown in Fig. 9.30(c). The expression $b + c$ in block B_4 is redundant on the path $B_1 \rightarrow B_2 \rightarrow B_4$, but not on the path $B_1 \rightarrow B_3 \rightarrow B_4$. We can eliminate the redundancy on the former path by placing a computation of $b + c$ in block B_3. All the results of $b + c$ are written into a temporary variable t, and the calculation in block B_4 is replaced with t. Thus, like loop-invariant code motion, partial-redundancy elimination requires the placement of new expression computations.

9.5.2 Can All Redundancy Be Eliminated?

Is it possible to eliminate all redundant computations along every path? The answer is "no," unless we are allowed to change the flow graph by creating new blocks.

Example 9.28 : In the example shown in Fig. 9.31(a), the expression of $b + c$ is computed redundantly in block B_4 if the program follows the execution path $B_1 \rightarrow B_2 \rightarrow B_4$. However, we cannot simply move the computation of $b + c$ to block B_3, because doing so would create an extra computation of $b + c$ when the path $B_1 \rightarrow B_3 \rightarrow B_5$ is taken.

What we would like to do is to insert the computation of $b + c$ only along the edge from block B_3 to block B_4. We can do so by placing the instruction in a new block, say, B_6, and making the flow of control from B_3 go through B_6 before it reaches B_4. The transformation is shown in Fig. 9.31(b). □

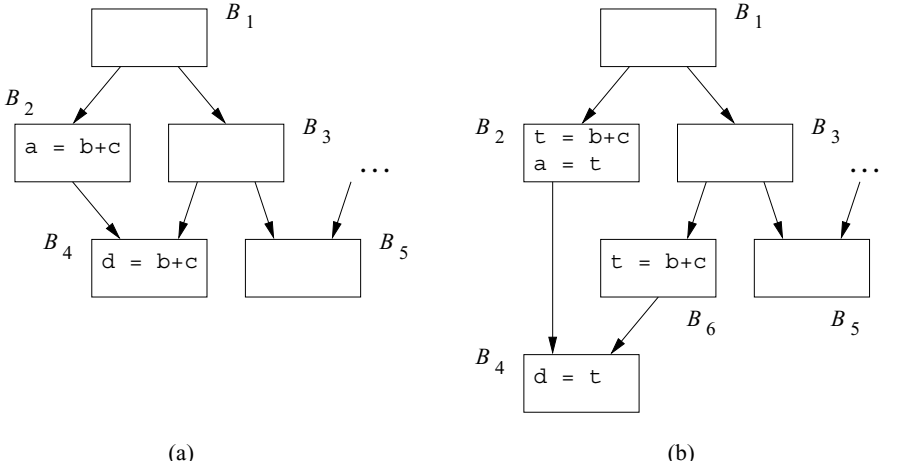


Figure 9.31: $B_3 \rightarrow B_4$ is a critical edge

We define a *critical edge* of a flow graph to be any edge leading from a node with more than one successor to a node with more than one predecessor. By introducing new blocks along critical edges, we can always find a block to accommodate the desired expression placement. For instance, the edge from B_3 to B_4 in Fig. 9.31(a) is critical, because B_3 has two successors, and B_4 has two predecessors.

Adding blocks may not be sufficient to allow the elimination of all redundant computations. As shown in Example 9.29, we may need to duplicate code so as to isolate the path where redundancy is found.

Example 9.29 : In the example shown in Figure 9.32(a), the expression of $b+c$ is computed redundantly along the path $B_1 \rightarrow B_2 \rightarrow B_4 \rightarrow B_6$. We would like to remove the redundant computation of $b + c$ from block B_6 in this path and compute the expression only along the path $B_1 \rightarrow B_3 \rightarrow B_4 \rightarrow B_6$. However, there is no single program point or edge in the source program that corresponds uniquely to the latter path. To create such a program point, we can duplicate the pair of blocks B_4 and B_6, with one pair reached through B_2 and the other reached through B_3, as shown in Figure 9.32(b). The result of $b + c$ is saved in variable t in block B_2, and moved to variable d in B_6', the copy of B_6 reached from B_2. □

Since the number of paths is exponential in the number of conditional branches in the program, eliminating all redundant expressions can greatly increase the size of the optimized code. We therefore restrict our discussion of redundancy-elimination techniques to those that may introduce additional blocks but that do not duplicate portions of the control flow graph.

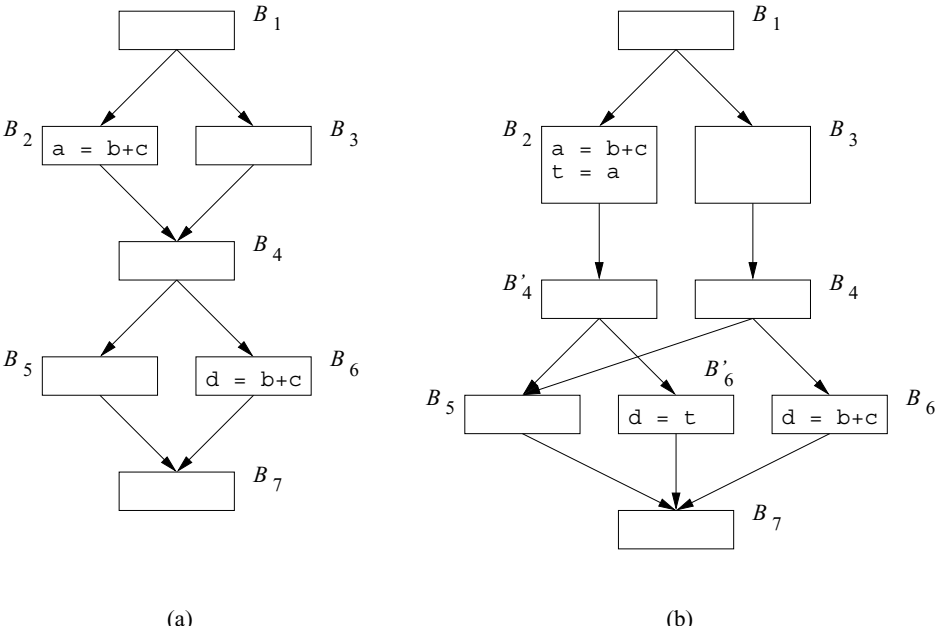


Figure 9.32: Code duplication to eliminate redundancies

9.5.3 The Lazy-Code-Motion Problem

It is desirable for programs optimized with a partial-redundancy-elimination algorithm to have the following properties:

1. All redundant computations of expressions that can be eliminated without code duplication are eliminated.

2. The optimized program does not perform any computation that is not in the original program execution.

3. Expressions are computed at the latest possible time.

The last property is important because the values of expressions found to be redundant are usually held in registers until they are used. Computing a value as late as possible minimizes its lifetime — the duration between the time the value is defined and the time it is last used, which in turn minimizes its usage of a register. We refer to the optimization of eliminating partial redundancy with the goal of delaying the computations as much as possible as *lazy code motion*.

To build up our intuition of the problem, we first discuss how to reason about partial redundancy of a single expression along a single path. For convenience, we assume for the rest of the discussion that every statement is a basic block of its own.

Full Redundancy

An expression e in block B is redundant if along all paths reaching B, e has been evaluated and the operands of e have not been redefined subsequently. Let S be the set of blocks, each containing expression e, that renders e in B redundant. The set of edges leaving the blocks in S must necessarily form a *cutset*, which if removed, disconnects block B from the entry of the program. Moreover, no operands of e are redefined along the paths that lead from the blocks in S to B.

Partial Redundancy

If an expression e in block B is only partially redundant, the lazy-code-motion algorithm attempts to render e fully redundant in B by placing additional copies of the expressions in the flow graph. If the attempt is successful, the optimized flow graph will also have a set of basic blocks S, each containing expression e, and whose outgoing edges are a cutset between the entry and B. Like the fully redundant case, no operands of e are redefined along the paths that lead from the blocks in S to B.

9.5.4 Anticipation of Expressions

There is an additional constraint imposed on inserted expressions to ensure that no extra operations are executed. Copies of an expression must be placed only at program points where the expression is *anticipated*. We say that an expression $b + c$ is *anticipated* at point p if all paths leading from the point p eventually compute the value of the expression $b + c$ from the values of b and c that are available at that point.

Let us now examine what it takes to eliminate partial redundancy along an acyclic path $B_1 \rightarrow B_2 \rightarrow \ldots \rightarrow B_n$. Suppose expression e is evaluated only in blocks B_1 and B_n, and that the operands of e are not redefined in blocks along the path. There are incoming edges that join the path and there are outgoing edges that exit the path. We see that e is *not* anticipated at the entry of block B_i if and only if there exists an outgoing edge leaving block B_j, $i \leq j < n$, that leads to an execution path that does not use the value of e. Thus, anticipation limits how early an expression can be inserted.

We can create a cutset that includes the edge $B_{i-1} \rightarrow B_i$ and that renders e redundant in B_n if e is either available or anticipated at the entry of B_i. If e is anticipated but not available at the entry of B_i, we must place a copy of the expression e along the incoming edge.

We have a choice of where to place the copies of the expression, since there are usually several cutsets in the flow graph that satisfy all the requirements. In the above, computation is introduced along the incoming edges to the path of interest and so the expression is computed as close to the use as possible, without introducing redundancy. Note that these introduced operations may themselves be partially redundant with other instances of the same expression

in the program. Such partial redundancy may be eliminated by moving these computations further up.

In summary, anticipation of expressions limits how early an expression can be placed; you cannot place an expression so early that it is not anticipated where you place it. The earlier an expression is placed, the more redundancy can be removed, and among all solutions that eliminate the same redundancies, the one that computes the expressions the latest minimizes the lifetimes of the registers holding the values of the expressions involved.

9.5.5 The Lazy-Code-Motion Algorithm

This discussion thus motivates a four-step algorithm. The first step uses anticipation to determine where expressions can be placed; the second step finds the *earliest* cutset, among those that eliminate as many redundant operations as possible without duplicating code and without introducing any unwanted computations. This step places the computations at program points where the values of their results are first anticipated. The third step then pushes the cutset down to the point where any further delay would alter the semantics of the program or introduce redundancy. The fourth and final step is a simple pass to clean up the code by removing assignments to temporary variables that are used only once. Each step is accomplished with a data-flow pass: the first and fourth are backward-flow problems, the second and third are forward-flow problems.

Algorithm Overview

1. Find all the expressions anticipated at each program point using a backward data-flow pass.

2. The second step places the computation where the values of the expressions are first anticipated along some path. After we have placed copies of an expression where the expression is first anticipated, the expression would be *available* at program point p if it has been anticipated along all paths reaching p. Availability can be solved using a forward data-flow pass. If we wish to place the expressions at the earliest possible positions, we can simply find those program points where the expressions are anticipated but are not available.

3. Executing an expression as soon as it is anticipated may produce a value long before it is used. An expression is *postponable* at a program point if the expression has been anticipated and has yet to be used along any path reaching the program point. Postponable expressions are found using a forward data-flow pass. We place expressions at those program points where they can no longer be postponed.

4. A simple, final backward data-flow pass is used to eliminate assignments to temporary variables that are used only once in the program.

Preprocessing Steps

We now present the full lazy-code-motion algorithm. To keep the algorithm simple, we assume that initially every statement is in a basic block of its own, and we only introduce new computations of expressions at the beginnings of blocks. To ensure that this simplification does not reduce the effectiveness of the technique, we insert a new block between the source and the destination of an edge if the destination has more than one predecessor. Doing so obviously also takes care of all critical edges in the program.

We abstract the semantics of each block B with two sets: e_use_B is the set of expressions computed in B and e_kill_B is the set of expressions killed, that is, the set of expressions any of whose operands are defined in B. Example 9.30 will be used throughout the discussion of the four data-flow analyses whose definitions are summarized in Fig. 9.34.

Example 9.30 : In the flow graph in Fig. 9.33(a), the expression $b + c$ appears three times. Because the block B_9 is part of a loop, the expression may be computed many times. The computation in block B_9 is not only loop invariant; it is also a redundant expression, since its value already has been used in block B_7. For this example, we need to compute $b+c$ only twice, once in block B_5 and once along the path after B_2 and before B_7. The lazy code motion algorithm will place the expression computations at the beginning of blocks B_4 and B_5. □

Anticipated Expressions

Recall that an expression $b + c$ is anticipated at a program point p if all paths leading from point p eventually compute the value of the expression $b + c$ from the values of b and c that are available at that point.

In Fig. 9.33(a), all the blocks anticipating $b+c$ on entry are shown as lightly shaded boxes. The expression $b + c$ is anticipated in blocks B_3, B_4, B_5, B_6, B_7, and B_9. It is not anticipated on entry to block B_2, because the value of c is recomputed within the block, and therefore the value of $b + c$ that would be computed at the beginning of B_2 is not used along any path. The expression $b+c$ is not anticipated on entry to B_1, because it is unnecessary along the branch from B_1 to B_2 (although it would be used along the path $B_1 \rightarrow B_5 \rightarrow B_6$). Similarly, the expression is not anticipated at the beginning of B_8, because of the branch from B_8 to B_{11}. The anticipation of an expression may oscillate along a path, as illustrated by $B_7 \rightarrow B_8 \rightarrow B_9$.

The data-flow equations for the anticipated-expressions problem are shown in Fig 9.34(a). The analysis is a backward pass. An anticipated expression at the exit of a block B is an anticipated expression on entry only if it is not in the e_kill_B set. Also a block B generates as new uses the set of e_use_B expressions. At the exit of the program, none of the expressions are anticipated. Since we are interested in finding expressions that are anticipated along every subsequent

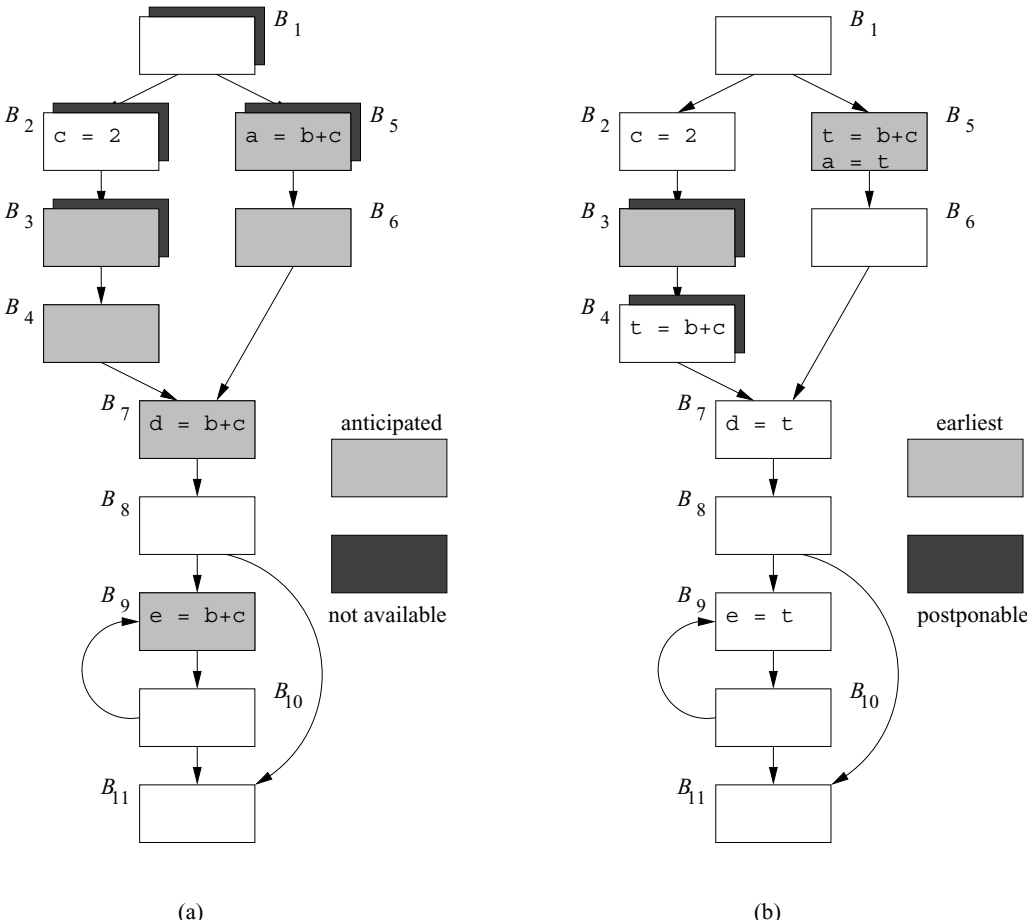


Figure 9.33: Flow graph of Example 9.30

path, the meet operator is set intersection. Consequently, the interior points must be initialized to the universal set U, as was discussed for the available-expressions problem in Section 9.2.6.

Available Expressions

At the end of this second step, copies of an expression will be placed at program points where the expression is first anticipated. If that is the case, an expression will be *available* at program point p if it is anticipated along all paths reaching p. This problem is similar to available-expressions described in Section 9.2.6. The transfer function used here is slightly different though. An expression is available on exit from a block if it is

	(a) Anticipated Expressions	(b) Available Expressions
Domain	Sets of expressions	Sets of expressions
Direction	Backwards	Forwards
Transfer function	$f_B(x) = e\_use_B \cup (x - e\_kill_B)$	$f_B(x) = (anticipated[B].in \cup x) - e\_kill_B$
Boundary	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
Meet ($\wedge$)	$\cap$	$\cap$
Equations	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
Initialization	$\text{IN}[B] = U$	$\text{OUT}[B] = U$

	(c) Postponable Expressions	(d) Used Expressions
Domain	Sets of expressions	Sets of expressions
Direction	Forwards	Backwards
Transfer function	$f_B(x) = (earliest[B] \cup x) - e\_use_B$	$f_B(x) = (e\_use_B \cup x) - latest[B]$
Boundary	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$
Meet ($\wedge$)	$\cap$	$\cup$
Equations	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$
Initialization	$\text{OUT}[B] = U$	$\text{IN}[B] = \emptyset$

$$earliest[B] = anticipated[B].in - available[B].in$$

$$latest[B] = (earliest[B] \cup postponable[B].in) \cap \Big(e\_use_B \cup \neg\big(\bigcap_{S,succ(B)} (earliest[S] \cup postponable[S].in)\big)\Big)$$

Figure 9.34: Four data-flow passes in partial-redundancy elimination

Completing the Square

Anticipated expressions (also called "very busy expressions" elsewhere) is a type of data-flow analysis we have not seen previously. While we have seen backwards-flowing frameworks such as live-variable analysis (Sect. 9.2.5), and we have seen frameworks where the meet is intersection such as available expressions (Sect. 9.2.6), this is the first example of a useful analysis that has both properties. Almost all analyses we use can be placed in one of four groups, depending on whether they flow forwards or backwards, and depending on whether they use union or intersection for the meet. Notice also that the union analyses always involve asking about whether there exists a path along which something is true, while the intersection analyses ask whether something is true along all paths.

1. Either
   (a) Available on entry, or
   (b) In the set of anticipated expressions upon entry (i.e., it *could* be made available if we chose to compute it here),

   and
2. Not killed in the block.

The data-flow equations for available expressions are shown in Fig 9.34(b). To avoid confusing the meaning of IN, we refer to the result of an earlier analysis by appending "$[B].in$" to the name of the earlier analysis.

With the earliest placement strategy, the set of expressions placed at block B, i.e., $earliest[B]$, is defined as the set of anticipated expressions that are not yet available. That is,

$$earliest[B] = anticipated[B].in - available[B].in.$$

Example 9.31 : The expression $b + c$ in the flow graph in Figure 9.35 is not anticipated at the entry of block B_3 but is anticipated at the entry of block B_4. It is, however, not necessary to compute the expression $b + c$ in block B_4, because the expression is already available due to block B_2. □

Example 9.32 : Shown with dark shadows in Fig. 9.33(a) are the blocks for which expression $b + c$ is not available; they are B_1, B_2, B_3, and B_5. The early-placement positions are represented by the lightly shaded boxes with dark shadows, and are thus blocks B_3 and B_5. Note, for instance, that $b + c$ is considered available on entry to B_4, because there is a path $B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow B_4$ along which $b + c$ is anticipated at least once — at B_3 in this case — and since the beginning of B_3, neither b nor c was recomputed. □

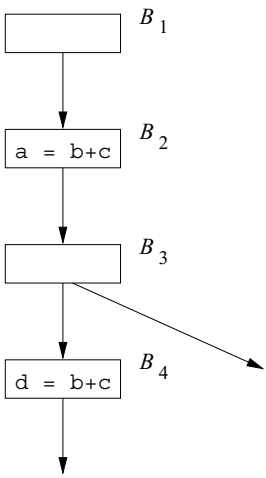


Figure 9.35: Flow graph for Example 9.31 illustrating the use of availability

Postponable Expressions

The third step postpones the computation of expressions as much as possible while preserving the original program semantics and minimizing redundancy. Example 9.33 illustrates the importance of this step.

Example 9.33 : In the flow graph shown in Figure 9.36, the expression $b + c$ is computed twice along the path $B_1 \rightarrow B_5 \rightarrow B_6 \rightarrow B_7$. The expression $b + c$ is anticipated even at the beginning of block B_1. If we compute the expression as soon as it is anticipated, we would have computed the expression $b + c$ in B_1. The result would have to be saved from the beginning, through the execution of the loop comprising blocks B_2 and B_3, until it is used in block B_7. Instead we can delay the computation of expression $b + c$ until the beginning of B_5 and until the flow of control is about to transition from B_4 to B_7. □

Formally, an expression $x + y$ is *postponable* to a program point p if an early placement of $x + y$ is encountered along every path from the entry node to p, and there is no subsequent use of $x + y$ after the last such placement.

Example 9.34 : Let us again consider expression $b + c$ in Fig. 9.33. The two earliest points for $b + c$ are B_3 and B_5; note that these are the two blocks that are both lightly and darkly shaded in Fig. 9.33(a), indicating that $b + c$ is both anticipated and not available for these blocks, and only these blocks. We cannot postpone $b + c$ from B_5 to B_6, because $b + c$ is used in B_5. We can postpone it from B_3 to B_4, however.

But we cannot postpone $b + c$ from B_4 to B_7. The reason is that, although $b + c$ is not used in B_4, placing its computation at B_7 instead would lead to a

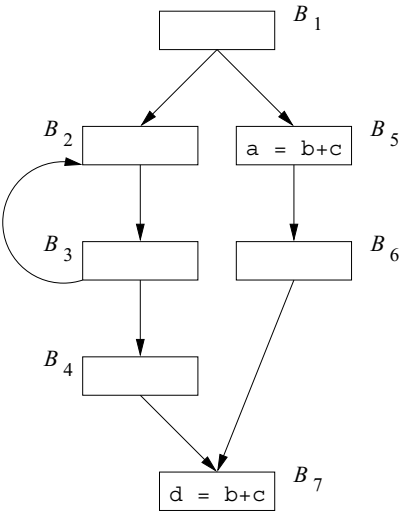


Figure 9.36: Flow graph for Example 9.33 to illustrate the need for postponing an expression

redundant computation of $b + c$ along the path $B_5 \rightarrow B_6 \rightarrow B_7$. As we shall see, B_4 is one of the latest places we can compute $b + c$. $\Box$

The data-flow equations for the postponable-expressions problem are shown in Fig 9.34(c). The analysis is a forward pass. We cannot "postpone" an expression to the entry of the program, so OUT[ENTRY] $= \emptyset$. An expression is postponable to the exit of block B if it is not used in the block, and either it is postponable to the entry of B or it is in $earliest[B]$. An expression is not postponable to the entry of a block unless all its predecessors include the expression in their *postponable* sets at their exits. Thus, the meet operator is set intersection, and the interior points must be initialized to the top element of the semilattice — the universal set.

Roughly speaking, an expression is placed at the *frontier* where an expression transitions from being postponable to not being postponable. More specifically, an expression e may be placed at the beginning of a block B only if the expression is in B's *earliest* or *postponable* set upon entry. In addition, B is in the postponement frontier of e if one of the following holds:

1. e is not in $postponable[B].out$. In other words, e is in e_use_B.
2. e cannot be postponed to one of its successors. In other words, there exists a successor of B such that e is not in the *earliest* or *postponable* set upon entry to that successor.

Expression e can be placed at the front of block B in either of the above scenarios because of the new blocks introduced by the preprocessing step in the algorithm.

Example 9.35 : Fig. 9.33(b) shows the result of the analysis. The light-shaded boxes represent the blocks whose *earliest* set includes $b + c$. The dark shadows indicate those that include $b + c$ in their *postponable* set. The latest placements of the expressions are thus the entries of blocks B_4 and B_5, since

1. $b + c$ is in the *postponable* set of B_4 but not B_7, and
2. B_5's *earliest* set includes $b + c$ and it uses $b + c$.

The expression is stored into the temporary variable t in blocks B_4 and B_5, and t is used in place of $b + c$ everywhere else, as shown in the figure. □

Used Expressions

Finally, a backward pass is used to determine if the temporary variables introduced are used beyond the block they are in. We say that an expression is *used* at point p if there exists a path leading from p that uses the expression before the value is reevaluated. This analysis is essentially liveness analysis (for expressions, rather than for variables).

The data-flow equations for the used expressions problem are shown in Fig 9.34(d). The analysis is a backward pass. A used expression at the exit of a block B is a used expression on entry only if it is not in the *latest* set. A block generates, as new uses, the set of expressions in e_use_B. At the exit of the program, none of the expressions are used. Since we are interested in finding expressions that are used by any subsequent path, the meet operator is set union. Thus, the interior points must be initialized with the top element of the semilattice — the empty set.

Putting it All Together

All the steps of the algorithm are summarized in Algorithm 9.36.

Algorithm 9.36 : Lazy code motion.

INPUT: A flow graph for which e_use_B and e_kill_B have been computed for each block B.

OUTPUT: A modified flow graph satisfying the four lazy code motion conditions in Section 9.5.3.

METHOD:

1. Insert an empty block along all edges entering a block with more than one predecessor.
2. Find $anticipated[B].in$ for all blocks B, as defined in Fig. 9.34(a).
3. Find $available[B].in$ for all blocks B as defined in Fig. 9.34(b).

4. Compute the earliest placements for all blocks B:

$$earliest[B] = anticipated[B].in - available[B].in$$

5. Find $postponable[B].in$ for all blocks B as defined in Fig. 9.34(c).

6. Compute the latest placements for all blocks B:

$$latest[B] = (earliest[B] \cup postponable[B].in) \cap \left(e\_use_B \cup \neg\left(\bigcap_{S \text{ in } succ(B)} (earliest[S] \cup postponable[S].in)\right)\right)$$

Note that $\neg$ denotes complementation with respect to the set of all expressions computed by the program.

7. Find $used[B].out$ for all blocks B, as defined in Fig. 9.34(d).

8. For each expression, say $x+y$, computed by the program, do the following:

   (a) Create a new temporary, say t, for $x + y$.

   (b) For all blocks B such that $x + y$ is in $latest[B] \cap used[B].out$, add `t = x+y` at the beginning of B.

   (c) For all blocks B such that $x + y$ is in

$$e\_use_B \cap (\neg latest[B] \cup used.out[B])$$

   replace every original $x + y$ by t.

□

Summary

Partial-redundancy elimination finds many different forms of redundant operations in one unified algorithm. This algorithm illustrates how multiple data-flow problems can be used to find optimal expression placement.

1. The placement constraints are provided by the anticipated-expressions analysis, which is a *backwards* data-flow analysis with a set-intersection meet operator, as it determines if expressions are used *subsequent* to each program point on *all* paths.

2. The earliest placement of an expression is given by program points where the expression is anticipated but is not available. Available expressions are found with a *forwards* data-flow analysis with a set-intersection meet operator that computes if an expression has been anticipated *before* each program point along *all* paths.

3. The latest placement of an expression is given by program points where an expression can no longer be postponed. Expressions are postponable at a program point if for *all* paths *reaching* the program point, no use of the expression has been encountered. Postponable expressions are found with a *forwards* data-flow analysis with a set-intersection meet operator.

4. Temporary assignments are eliminated unless they are used by *some* path *subsequently*. We find used expressions with a *backwards* data-flow analysis, this time with a set-union meet operator.

9.5.6 Exercises for Section 9.5

Exercise 9.5.1: For the flow graph in Fig. 9.37:

a) Compute *anticipated* for the beginning and end of each block.

b) Compute *available* for the beginning and end of each block.

c) Compute *earliest* for each block.

d) Compute *postponable* for the beginning and end of each block.

e) Compute *used* for the beginning and end of each block.

f) Compute *latest* for each block.

g) Introduce temporary variable t; show where it is computed and where it is used.

Exercise 9.5.2: Repeat Exercise 9.5.1 for the flow graph of Fig. 9.10 (see the exercises to Section 9.1). You may limit your analysis to the expressions $a + b$, $c - a$, and $b * d$.

!! **Exercise 9.5.3:** The concepts discussed in this section can also be applied to eliminate partially dead code. A definition of a variable is *partially dead* if the variable is live on some paths and not others. We can optimize the program execution by only performing the definition along paths where the variable is live. Unlike partial-redundancy elimination, where expressions are moved before the original, the new definitions are placed after the original. Develop an algorithm to move partially dead code, so expressions are evaluated only where they will eventually be used.

9.6 Loops in Flow Graphs

In our discussion so far, loops have not been handled differently; they have been treated just like any other kind of control flow. However, loops are important because programs spend most of their time executing them, and optimizations

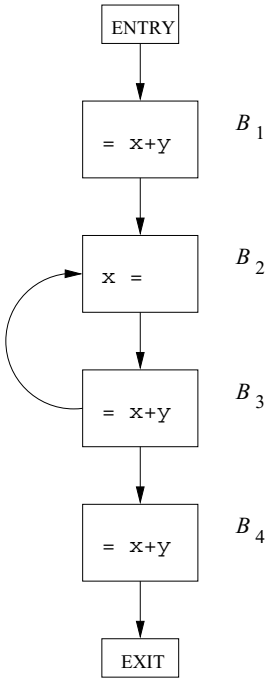


Figure 9.37: Flow graph for Exercise 9.5.1

that improve the performance of loops can have a significant impact. Thus, it is essential that we identify loops and treat them specially.

Loops also affect the running time of program analyses. If a program does not contain any loops, we can obtain the answers to data-flow problems by making just one pass through the program. For example, a forward data-flow problem can be solved by visiting all the nodes once, in topological order.

In this section, we introduce the following concepts: dominators, depth-first ordering, back edges, graph depth, and reducibility. Each of these is needed for our subsequent discussions on finding loops and the speed of convergence of iterative data-flow analysis.

9.6.1 Dominators

We say node d of a flow graph *dominates* node n, written d *dom* n, if every path from the entry node of the flow graph to n goes through d. Note that under this definition, every node dominates itself.

Example 9.37: Consider the flow graph of Fig. 9.38, with entry node 1. The entry node dominates every node (this statement is true for every flow graph). Node 2 dominates only itself, since control can reach any other node along a path that begins with $1 \rightarrow 3$. Node 3 dominates all but 1 and 2. Node 4 dominates

all but 1, 2 and 3, since all paths from 1 must begin with $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ or $1 \rightarrow 3 \rightarrow 4$. Nodes 5 and 6 dominate only themselves, since flow of control can skip around either by going through the other. Finally, 7 dominates 7, 8, 9, and 10; 8 dominates 8, 9, and 10; 9 and 10 dominate only themselves. □

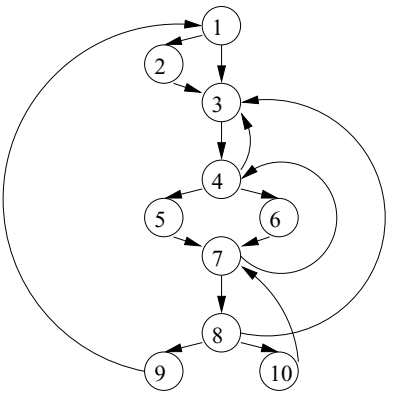


Figure 9.38: A flow graph

A useful way of presenting dominator information is in a tree, called the *dominator tree*, in which the entry node is the root, and each node d dominates only its descendants in the tree. For example, Fig. 9.39 shows the dominator tree for the flow graph of Fig. 9.38.

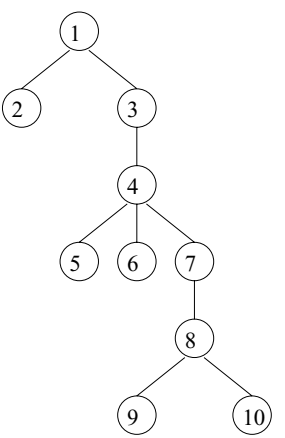


Figure 9.39: Dominator tree for flow graph of Fig. 9.38

The existence of dominator trees follows from a property of dominators: each node n has a unique *immediate dominator* m that is the last dominator of n on any path from the entry node to n. In terms of the *dom* relation, the

immediate dominator m has that property that if $d \neq n$ and d *dom* n, then d *dom* m.

We shall give a simple algorithm for computing the dominators of every node n in a flow graph, based on the principle that if $p_1, p_2, \ldots, p_k$ are all the predecessors of n, and $d \neq n$, then d *dom* n if and only if d *dom* p_i for each i. This problem can be formulated as a forward data-flow analysis. The data-flow values are sets of basic blocks. A node's set of dominators, other than itself, is the intersection of the dominators of all its predecessors; thus the meet operator is set intersection. The transfer function for block B simply adds B itself to the set of input nodes. The boundary condition is that the ENTRY node dominates itself. Finally, the initialization of the interior nodes is the universal set, that is, the set of all nodes.

Algorithm 9.38: Finding dominators.

INPUT: A flow graph G with set of nodes N, set of edges E and entry node ENTRY.

OUTPUT: $D(n)$, the set of nodes that dominate node n, for all nodes n in N.

METHOD: Find the solution to the data-flow problem whose parameters are shown in Fig. 9.40. The basic blocks are the nodes. $D(n) = \text{OUT}[n]$ for all n in N. □

Finding dominators using this data-flow algorithm is efficient. Nodes in the graph need to be visited only a few times, as we shall see in Section 9.6.7.

	Dominators
Domain	The power set of N
Direction	Forwards
Transfer function	$f_B(x) = x \cup \{B\}$
Boundary	$\text{OUT}[\text{ENTRY}] = \{\text{ENTRY}\}$
Meet ($\wedge$)	$\cap$
Equations	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
Initialization	$\text{OUT}[B] = N$

Figure 9.40: A data-flow algorithm for computing dominators

Example 9.39: Let us return to the flow graph of Fig. 9.38, and suppose the for-loop of lines (4) through (6) in Fig. 9.23 visits the nodes in numerical order. Let $D(n)$ be the set of nodes in $\text{OUT}[n]$. Since 1 is the entry node, $D(1)$ was assigned $\{1\}$ at line (1). Node 2 has only 1 for a predecessor, so

Properties of the *dom* Relation

A key observation about dominators is that if we take any acyclic path from the entry to node n, then all the dominators of n appear along this path, and moreover, they must appear *in the same order* along any such path. To see why, suppose there were one acyclic path P_1 to n along which dominators a and b appeared in that order and another path P_2 to n, along which b preceded a. Then we could follow P_1 to a and P_2 to n, thereby avoiding b altogether. Thus, b would not really dominate n.

This reasoning allows us to prove that *dom* is transitive: if a *dom* b and b *dom* c, then a *dom* c. Also, *dom* is antisymmetric: it is never possible that both a *dom* b and b *dom* a hold, if $a \neq b$. Moreover, if a and b are two dominators of n, then either a *dom* b or b *dom* a must hold. Finally, it follows that each node n except the entry must have a unique immediate dominator — the dominator that appears closest to n along any acyclic path from the entry to n.

$D(2) = \{2\} \cup D(1)$. Thus, $D(2)$ is set to $\{1, 2\}$. Then node 3, with predecessors 1, 2, 4, and 8, is considered. Since all the interior nodes are initialized with the universal set N,

$$D(3) = \{3\} \cup (\{1\} \cap \{1,2\} \cap \{1,2,\ldots,10\} \cap \{1,2,\ldots,10\}) = \{1,3\}$$

The remaining calculations are shown in Fig. 9.41. Since these values do not change in the second iteration through the outer loop of lines (3) through (6) in Fig. 9.23(a), they are the final answers to the dominator problem. □

$$\begin{aligned}
D(4) &= \{4\} \cup (D(3) \cap D(7)) = \{4\} \cup (\{1,3\} \cap \{1,2,\ldots,10\}) = \{1,3,4\} \\
D(5) &= \{5\} \cup D(4) = \{5\} \cup \{1,3,4\} = \{1,3,4,5\} \\
D(6) &= \{6\} \cup D(4) = \{6\} \cup \{1,3,4\} = \{1,3,4,6\} \\
D(7) &= \{7\} \cup (D(5) \cap D(6) \cap D(10)) \\
&= \{7\} \cup (\{1,3,4,5\} \cap \{1,3,4,6\} \cap \{1,2,\ldots,10\}) = \{1,3,4,7\} \\
D(8) &= \{8\} \cup D(7) = \{8\} \cup \{1,3,4,7\} = \{1,3,4,7,8\} \\
D(9) &= \{9\} \cup D(8) = \{9\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,9\} \\
D(10) &= \{10\} \cup D(8) = \{10\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,10\}
\end{aligned}$$

Figure 9.41: Completion of the dominator calculation for Example 9.39

9.6.2 Depth-First Ordering

As introduced in Section 2.3.4, a *depth-first search* of a graph visits all the nodes in the graph once, by starting at the entry node and visiting the nodes as far away from the entry node as quickly as possible. The route of the search in a depth-first search forms a *depth-first spanning tree* (DFST). Recall from Section 2.3.4 that a preorder traversal visits a node before visiting any of its children, which it then visits recursively in left-to-right order. Also, a postorder traversal visits a node's children, recursively in left-to-right order, before visiting the node itself.

There is one more variant ordering that is important for flow-graph analysis: a *depth-first ordering* is the reverse of a postorder traversal. That is, in a depth-first ordering, we visit a node, then traverse its rightmost child, the child to its left, and so on. However, before we build the tree for the flow graph, we have choices as to which successor of a node becomes the rightmost child in the tree, which node becomes the next child, and so on. Before we give the algorithm for depth-first ordering, let us consider an example.

Example 9.40 : One possible depth-first presentation of the flow graph in Fig. 9.38 is illustrated in Fig. 9.42. Solid edges form the tree; dashed edges are the other edges of the flow graph. A depth-first traversal of the tree is given by: $1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10$, then back to 8, then to 9. We go back to 8 once more, retreating to 7, 6, and 4, and then forward to 5. We retreat from 5 back to 4, then back to 3 and 1. From 1 we go to 2, then retreat from 2, back to 1, and we have traversed the entire tree.

The preorder sequence for the traversal is thus

$$1, 3, 4, 6, 7, 8, 10, 9, 5, 2.$$

The postorder sequence for the traversal of the tree in Fig. 9.42 is

$$10, 9, 8, 7, 6, 5, 4, 3, 2, 1.$$

The depth-first ordering, which is the reverse of the postorder sequence, is

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10.$$

□

We now give an algorithm that finds a depth-first spanning tree and a depth-first ordering of a graph. It is this algorithm that finds the DFST in Fig. 9.42 from Fig. 9.38.

Algorithm 9.41 : Depth-first spanning tree and depth-first ordering.

INPUT: A flow graph G.

OUTPUT: A DFST T of G and an ordering of the nodes of G.

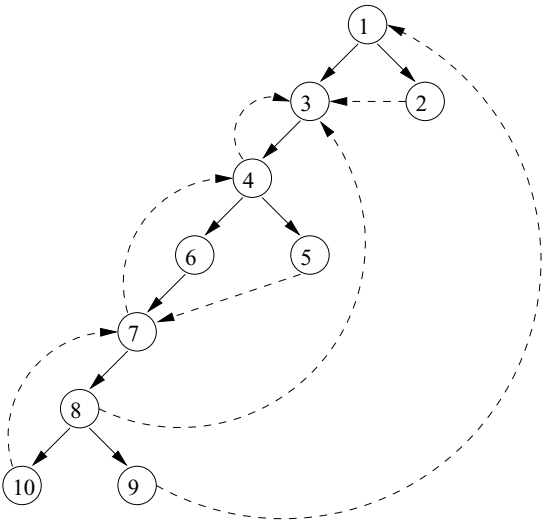


Figure 9.42: A depth-first presentation of the flow graph in Fig. 9.38

METHOD: We use the recursive procedure *search*(n) of Fig. 9.43. The algorithm initializes all nodes of G to "unvisited," then calls *search*(n_0), where n_0 is the entry. When it calls *search*(n), it first marks n "visited" to avoid adding n to the tree twice. It uses c to count from the number of nodes of G down to 1, assigning depth-first numbers *dfn*[n] to nodes n as we go. The set of edges T forms the depth-first spanning tree for G. □

Example 9.42: For the flow graph in Fig. 9.42, Algorithm 9.41 sets c to 10 and begins the search by calling *search*(1). The rest of the execution sequence is shown in Fig. 9.44. □

9.6.3 Edges in a Depth-First Spanning Tree

When we construct a DFST for a flow graph, the edges of the flow graph fall into three categories.

1. There are edges, called *advancing edges*, that go from a node m to a proper descendant of m in the tree. All edges in the DFST itself are advancing edges. There are no other advancing edges in Fig. 9.42, but, for example, if $4 \rightarrow 8$ were an edge, it would be in this category.

2. There are edges that go from a node m to an ancestor of m in the tree (possibly to m itself). These edges we shall term *retreating edges*. For example, $4 \rightarrow 3$, $7 \rightarrow 4$, $10 \rightarrow 7$, $8 \rightarrow 3$, and $9 \rightarrow 1$ are the retreating edges in Fig. 9.42.

```
void search(n) {
        mark n "visited";
        for (each successor s of n)
                if (s is "unvisited") {
                        add edge n → s to T;
                        search(s);
                }
        dfn[n] = c;
        c = c − 1;
}

main() {
        T = ∅; /* set of edges */
        for (each node n of G)
                mark n "unvisited";
        c = number of nodes of G;
        search(n0);
}
```

Figure 9.43: Depth-first search algorithm

3. There are edges $m \rightarrow n$ such that neither m nor n is an ancestor of the other in the DFST. Edges $2 \rightarrow 3$ and $5 \rightarrow 7$ are the only such examples in Fig. 9.42. We call these edges *cross edges*. An important property of cross edges is that if we draw the DFST so children of a node are drawn from left to right in the order in which they were added to the tree, then all cross edges travel from right to left.

It should be noted that $m \rightarrow n$ is a retreating edge if and only if $dfn[m] \geq dfn[n]$. To see why, note that if m is a descendant of n in the DFST, then $search(m)$ terminates before $search(n)$, so $dfn[m] \geq dfn[n]$. Conversely, if $dfn[m] \geq dfn[n]$, then $search(m)$ terminates before $search(n)$, or $m = n$. But $search(n)$ must have begun before $search(m)$ if there is an edge $m \rightarrow n$, or else the fact that n is a successor of m would have made n a descendant of m in the DFST. Thus the time $search(m)$ is active is a subinterval of the time $search(n)$ is active, from which it follows that n is an ancestor of m in the DFST.

9.6.4 Back Edges and Reducibility

A *back edge* is an edge $a \rightarrow b$ whose head b dominates its tail a. For any flow graph, every back edge is retreating, but not every retreating edge is a back edge. A flow graph is said to be *reducible* if all its retreating edges in any depth-first spanning tree are also back edges. In other words, if a graph is reducible, then all the DFST's have the same set of retreating edges, and

Call *search*(1)	Node 1 has two successors. Suppose $s = 3$ is considered first; add edge $1 \rightarrow 3$ to T.
Call *search*(3)	Add edge $3 \rightarrow 4$ to T.
Call *search*(4)	Node 4 has two successors, 4 and 6. Suppose $s = 6$ is considered first; add edge $4 \rightarrow 6$ to T.
Call *search*(6)	Add $6 \rightarrow 7$ to T.
Call *search*(7)	Node 7 has two successors, 4 and 8. But 4 is already marked "visited" by *search*(4), so do nothing when $s = 4$. For $s = 8$, add edge $7 \rightarrow 8$ to T.
Call *search*(8)	Node 8 has two successors, 9 and 10. Suppose $s = 10$ is considered first; add edge $8 \rightarrow 10$.
Call *search*(10)	10 has a successor, 7, but 7 is already marked "visited." Thus, *search*(10) completes by setting *dfn*[10] = 10 and $c = 9$.
Return to *search*(8)	Set $s = 9$ and add edge $8 \rightarrow 9$ to T.
Call *search*(9)	The only successor of 9, node 1, is already "visited," so set *dfn*[9] = 9 and $c = 8$.
Return to *search*(8)	The last successor of 8, node 3, is "visited," so do nothing for $s = 3$. At this point, all successors of 8 have been considered, so set *dfn*[8] = 8 and $c = 7$.
Return to *search*(7)	All of 7's successors have been considered, so set *dfn*[7] = 7 and $c = 6$.
Return to *search*(6)	Similarly, 6's successors have been considered, so set *dfn*[6] = 6 and $c = 5$.
Return to *search*(4)	Successor 3 of 4 has been "visited," but 5 has not, so add $4 \rightarrow 5$ to the tree.
Call *search*(5)	Successor 7 of 5 has been "visited," thus set *dfn*[5] = 5 and $c = 4$.
Return to *search*(4)	All successors of 4 have been considered, set *dfn*[4] = 4 and $c = 3$.
Return to *search*(3)	Set *dfn*[3] = 3 and $c = 2$.
Return to *search*(1)	2 has not been visited yet, so add $1 \rightarrow 2$ to T.
Call *search*(2)	Set *dfn*[2] = 2, $c = 1$.
Return to *search*(1)	Set *dfn*[1] = 1 and $c = 0$.

Figure 9.44: Execution of Algorithm 9.41 on the flow graph in Fig. 9.42

Why Are Back Edges Retreating Edges?

Suppose $a \rightarrow b$ is a back edge; i.e., its head dominates its tail. The sequence of calls of the function *search* in Fig. 9.43 that lead to node a must be a path in the flow graph. This path must, of course, include any dominator of a. It follows that a call to $search(b)$ must be open when $search(a)$ is called. Therefore b is already in the tree when a is added to the tree, and a is added as a descendant of b. Therefore, $a \rightarrow b$ must be a retreating edge.

those are exactly the back edges in the graph. If the graph is *nonreducible* (not reducible), however, all the back edges are retreating edges in any DFST, but each DFST may have additional retreating edges that are not back edges. These retreating edges may be different from one DFST to another. Thus, if we remove all the back edges of a flow graph and the remaining graph is cyclic, then the graph is nonreducible, and conversely.

Flow graphs that occur in practice are almost always reducible. Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible. Even programs written using goto statements often turn out to be reducible, as the programmer logically thinks in terms of loops and branches.

Example 9.43 : The flow graph of Fig. 9.38 is reducible. The retreating edges in the graph are all back edges; that is, their heads dominate their respective tails. □

Example 9.44 : Consider the flow graph of Fig. 9.45, whose initial node is 1. Node 1 dominates nodes 2 and 3, but 2 does not dominate 3, nor vice-versa. Thus, this flow graph has no back edges, since no head of any edge dominates its tail. There are two possible depth-first spanning trees, depending on whether we choose to call $search(2)$ or $search(3)$ first, from $search(1)$. In the first case, edge $3 \rightarrow 2$ is a retreating edge but not a back edge; in the second case, $2 \rightarrow 3$ is the retreating-but-not-back edge. Intuitively, the reason this flow graph is not reducible is that the cycle 2–3 can be entered at two different places, nodes 2 and 3. □

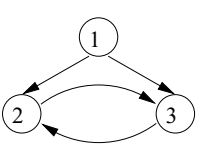


Figure 9.45: The canonical nonreducible flow graph

9.6.5 Depth of a Flow Graph

Given a depth-first spanning tree for the graph, the *depth* is the largest number of retreating edges on any cycle-free path. We can prove the depth is never greater than what one would intuitively call the depth of loop nesting in the flow graph. If a flow graph is reducible, we may replace "retreating" by "back" in the definition of "depth," since the retreating edges in any DFST are exactly the back edges. The notion of depth then becomes independent of the DFST actually chosen, and we may truly speak of the "depth of a flow graph," rather than the depth of a flow graph in connection with one of its depth-first spanning trees.

Example 9.45: In Fig. 9.42, the depth is 3, since there is a path

$$10 \rightarrow 7 \rightarrow 4 \rightarrow 3$$

with three retreating edges, but no cycle-free path with four or more retreating edges. It is a coincidence that the "deepest" path here has only retreating edges; in general we may have a mixture of retreating, advancing, and cross edges in a deepest path. □

9.6.6 Natural Loops

Loops can be specified in a source program in many different ways: they can be written as for-loops, while-loops, or repeat-loops; they can even be defined using labels and goto statements. From a program-analysis point of view, it does not matter how the loops appear in the source code. What matters is whether they have the properties that enable easy optimization. In particular, we care about whether a loop has a single-entry node; if it does, compiler analyses can assume certain initial conditions to hold at the beginning of each iteration through the loop. This opportunity motivates the need for the definition of a "natural loop."

A *natural loop* is defined by two essential properties.

1. It must have a single-entry node, called the *header*. This entry node dominates all nodes in the loop, or it would not be the sole entry to the loop.

2. There must be a back edge that enters the loop header. Otherwise, it is not possible for the flow of control to return to the header directly from the "loop"; i.e., there really is no loop.

Given a back edge $n \rightarrow d$, we define the *natural loop of the edge* to be d plus the set of nodes that can reach n without going through d. Node d is the header of the loop.

Algorithm 9.46: Constructing the natural loop of a back edge.

INPUT: A flow graph G and a back edge $n \rightarrow d$.

OUTPUT: The set *loop* consisting of all nodes in the natural loop of $n \rightarrow d$.

METHOD: Let *loop* be $\{n, d\}$. Mark d as "visited," so that the search does not reach beyond d. Perform a depth-first search on the reverse control-flow graph starting with node n. Insert all the nodes visited in this search into *loop*. This procedure finds all the nodes that reach n without going through d. □

Example 9.47: In Fig. 9.38, there are five back edges, those whose heads dominate their tails: $10 \rightarrow 7$, $7 \rightarrow 4$, $4 \rightarrow 3$, $8 \rightarrow 3$ and $9 \rightarrow 1$. Note that these are exactly the edges that one would think of as forming loops in the flow graph.

Back edge $10 \rightarrow 7$ has natural loop $\{7, 8, 10\}$, since 8 and 10 are the only nodes that can reach 10 without going through 7. Back edge $7 \rightarrow 4$ has a natural loop consisting of $\{4, 5, 6, 7, 8, 10\}$ and therefore contains the loop of $10 \rightarrow 7$. We thus assume the latter is an inner loop contained inside the former.

The natural loops of back edges $4 \rightarrow 3$ and $8 \rightarrow 3$ have the same header, node 3, and they also happen to have the same set of nodes: $\{3, 4, 5, 6, 7, 8, 10\}$. We shall therefore combine these two loops as one. This loop contains the two smaller loops discovered earlier.

Finally, the edge $9 \rightarrow 1$ has as its natural loop the entire flow graph, and therefore is the outermost loop. In this example, the four loops are nested within one another. It is typical, however, to have two loops, neither of which is a subset of the other. □

In reducible flow graphs, since all retreating edges are back edges, we can associate a natural loop with each retreating edge. That statement does not hold for nonreducible graphs. For instance, the nonreducible flow graph in Fig. 9.45 has a cycle consisting of nodes 2 and 3. Neither of the edges in the cycle is a back edge, so this cycle does not fit the definition of a natural loop. We do not identify the cycle as a natural loop, and it is not optimized as such. This situation is acceptable, because our loop analyses can be made simpler by assuming that all loops have single-entry nodes, and nonreducible programs are rare in practice anyway.

By considering only natural loops as "loops," we have the useful property that unless two loops have the same header, they are either disjoint or one is nested within the other. Thus, we have a natural notion of *innermost loops*: loops that contain no other loops.

When two natural loops have the same header, as in Fig. 9.46, it is hard to tell which is the inner loop. Thus, we shall assume that when two natural loops have the same header, and neither is properly contained within the other, they are combined and treated as a single loop.

Example 9.48: The natural loops of the back edges $3 \rightarrow 1$ and $4 \rightarrow 1$ in Fig. 9.46 are $\{1, 2, 3\}$ and $\{1, 2, 4\}$, respectively. We shall combine them into a single loop, $\{1, 2, 3, 4\}$.

However, were there another back edge $2 \rightarrow 1$ in Fig. 9.46, its natural loop would be $\{1, 2\}$, a third loop with header 1. This set of nodes is properly

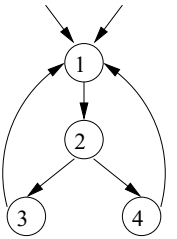


Figure 9.46: Two loops with the same header

contained within $\{1, 2, 3, 4\}$, so it would not be combined with the other natural loops, but rather treated as an inner loop, nested within. $\Box$

9.6.7 Speed of Convergence of Iterative Data-Flow Algorithms

We are now ready to discuss the speed of convergence of iterative algorithms. As discussed in Section 9.3.3, the maximum number of iterations the algorithm may take is the product of the height of the lattice and the number of nodes in the flow graph. For many data-flow analyses, it is possible to order the evaluation such that the algorithm converges in a much smaller number of iterations. The property of interest is whether all events of significance at a node will be propagated to that node along some acyclic path. Among the data-flow analyses discussed so far, reaching definitions, available expressions and live variables have this property, but constant propagation does not. More specifically:

- If a definition d is in IN$[B]$, then there is some acyclic path from the block containing d to B such that d is in the IN's and OUT's all along that path.
- If an expression $x + y$ is *not* available at the entrance to block B, then there is some acyclic path that demonstrates that either the path is from the entry node and includes no statement that kills or generates $x + y$, or the path is from a block that kills $x + y$ and along the path there is no subsequent generation of $x + y$.
- If x is live on exit from block B, then there is an acyclic path from B to a use of x, along which there are no definitions of x.

We should check that in each of these cases, paths with cycles add nothing. For example, if a use of x is reached from the end of block B along a path with a cycle, we can eliminate that cycle to find a shorter path along which the use of x is still reached from B.

In contrast, constant propagation does not have this property. Consider a simple program that has one loop containing a basic block with statements

```
L:    a = b
      b = c
      c = 1
      goto L
```

The first time the basic block is visited, c is found to have constant value 1, but both a and b are undefined. Visiting the basic block the second time, we find that b and c have constant values 1. It takes three visits of the basic block for the constant value 1 assigned to c to reach a.

If all useful information propagates along acyclic paths, we have an opportunity to tailor the order in which we visit nodes in iterative data-flow algorithms, so that after relatively few passes through the nodes we can be sure information has passed along all the acyclic paths.

Recall from Section 9.6.3 that if $a \rightarrow b$ is an edge, then the depth-first number of b is less than that of a only when the edge is a retreating edge. For forward data-flow problems, it is desirable to visit the nodes according to the depth-first ordering. Specifically, we modify the algorithm in Fig. 9.23(a) by replacing line (4), which visits the basic blocks in the flow graph with

for (each block B other than ENTRY, in depth-first order) {

Example 9.49 : Suppose we have a path along which a definition d propagates, such as

$$3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$$

where integers represent the depth-first numbers of the blocks along the path. Then the first time through the loop of lines (4) through (6) in the algorithm in Fig. 9.23(a), d will propagate from OUT[3] to IN[5] to OUT[5], and so on, up to OUT[35]. It will not reach IN[16] on that round, because as 16 precedes 35, we had already computed IN[16] by the time d was put in OUT[35]. However, the next time we run through the loop of lines (4) through (6), when we compute IN[16], d will be included because it is in OUT[35]. Definition d will also propagate to OUT[16], IN[23], and so on, up to OUT[45], where it must wait because IN[4] was already computed on this round. On the third pass, d travels to IN[4], OUT[4], IN[10], OUT[10], and IN[17], so after three passes we establish that d reaches block 17. □

It should not be hard to extract the general principle from this example. If we use depth-first order in Fig. 9.23(a), then the number of passes needed to propagate any reaching definition along any acyclic path is no more than one greater than the number of edges along that path that go from a higher numbered block to a lower numbered block. Those edges are exactly the retreating edges, so the number of passes needed is one plus the depth. Of course Algorithm 9.11 does not detect the fact that all definitions have reached wherever they can reach, until one more pass has yielded no changes. Therefore, the upper bound on the number of passes taken by that algorithm with depth-first

A Reason for Nonreducible Flow Graphs

There is one place where we cannot generally expect a flow graph to be reducible. If we reverse the edges of a program flow graph, as we did in Algorithm 9.46 to find natural loops, then we may not get a reducible flow graph. The intuitive reason is that, while typical programs have loops with single entries, those loops sometimes have several exits, which become entries when we reverse the edges.

block ordering is actually two plus the depth. A study[10] has shown that typical flow graphs have an average depth around 2.75. Thus, the algorithm converges very quickly.

In the case of backward-flow problems, like live variables, we visit the nodes in the reverse of the depth-first order. Thus, we may propagate a use of a variable in block 17 backwards along the path

$$3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$$

in one pass to IN[4], where we must wait for the next pass in order to reach OUT[45]. On the second pass it reaches IN[16], and on the third pass it goes from OUT[35] to OUT[3].

In general, one-plus-the-depth passes suffice to carry the use of a variable backward, along any acyclic path. However, we must choose the reverse of depth-first order to visit the nodes in a pass, because then, uses propagate along any decreasing sequence in a single pass.

The bound described so far is an upper bound on all problems where cyclic paths add no information to the analysis. In special problems such as dominators, the algorithm converges even faster. In the case where the input flow graph is reducible, the correct set of dominators for each node is obtained in the first iteration of a data-flow algorithm that visits the nodes in depth-first ordering. If we do not know that the input is reducible ahead of time, it takes an extra iteration to determine that convergence has occurred.

9.6.8 Exercises for Section 9.6

Exercise 9.6.1: For the flow graph of Fig. 9.10 (see the exercises for Section 9.1):

i. Compute the dominator relation.

ii. Find the immediate dominator of each node.

[10] D. E. Knuth, "An empirical study of FORTRAN programs," *Software — Practice and Experience* **1**:2 (1971), pp. 105–133.

iii. Construct the dominator tree.

iv. Find one depth-first ordering for the flow graph.

v. Indicate the advancing, retreating, cross, and tree edges for your answer to *iv*.

vi. Is the flow graph reducible?

vii. Compute the depth of the flow graph.

viii. Find the natural loops of the flow graph.

Exercise 9.6.2: Repeat Exercise 9.6.1 on the following flow graphs:

a) Fig. 9.3.

b) Fig. 8.9.

c) Your flow graph from Exercise 8.4.1.

d) Your flow graph from Exercise 8.4.2.

! **Exercise 9.6.3:** Prove the following about the *dom* relation:

a) If a *dom* b and b *dom* c, then a *dom* c (*transitivity*).

b) It is never possible that both a *dom* b and b *dom* a hold, if $a \neq b$ (*anti-symmetry*).

c) If a and b are two dominators of n, then either a *dom* b or b *dom* a must hold.

d) Each node n except the entry has a unique *immediate dominator* — the dominator that appears closest to n along any acyclic path from the entry to n.

! **Exercise 9.6.4:** Figure 9.42 is one depth-first presentation of the flow graph of Fig. 9.38. How many other depth-first presentations of this flow graph are there? Remember, order of children matters in distinguishing depth-first presentations.

!! **Exercise 9.6.5:** Prove that a flow graph is reducible if and only if when we remove all the back edges (those whose heads dominate their tails), the resulting flow graph is acyclic.

! **Exercise 9.6.6:** A *complete flow graph* on n nodes has arcs $i \rightarrow j$ between any two nodes i and j (in both directions). For what values of n is this graph reducible?

! **Exercise 9.6.7:** A *complete, acyclic flow graph* on n nodes $1, 2, \ldots, n$ has arcs $i \rightarrow j$ for all nodes i and j such that $i < j$. Node 1 is the entry.

a) For what values of n is this graph reducible?

b) Does your answer to (a) change if you add self-loops $i \to i$ for all nodes i?

! **Exercise 9.6.8:** The natural loop of a back edge $n \to h$ was defined to be h plus the set of nodes that can reach n without going through h. Show that h dominates all the nodes in the natural loop of $n \to h$.

!! **Exercise 9.6.9:** We claimed that the flow graph of Fig. 9.45 is nonreducible. If the arcs were replaced by paths of disjoint sets of nodes (except for the endpoints, of course), then the flow graph would still be nonreducible. In fact, node 1 need not be the entry; it can be any node reachable from the entry along a path whose intermediate nodes are not part of any of the four explicitly shown paths. Prove the converse: that every nonreducible flow graph has a subgraph like Fig. 9.45, but with arcs possibly replaced by node-disjoint paths and node 1 being any node reachable from the entry by a path that is node-disjoint from the four other paths.

!! **Exercise 9.6.10:** Show that every depth-first presentation for every nonreducible flow graph has a retreating edge that is not a back edge.

!! **Exercise 9.6.11:** Show that if the following condition

$$f(a) \wedge g(a) \wedge a \leq f\big(g(a)\big)$$

holds for all functions f and g, and value a, then the general iterative algorithm, Algorithm 9.25, with iteration following a depth-first ordering, converges within 2-plus-the-depth passes.

! **Exercise 9.6.12:** Find a nonreducible flow graph with two different DFST's that have different depths.

! **Exercise 9.6.13:** Prove the following:

a) If a definition d is in IN$[B]$, then there is some acyclic path from the block containing d to B such that d is in the IN's and OUT's all along that path.

b) If an expression $x + y$ is *not* available at the entrance to block B, then there is some acyclic path that demonstrates that fact; either the path is from the entry node and includes no statement that kills or generates $x+y$, or the path is from a block that kills $x+y$ and along the path there is no subsequent generation of $x + y$.

c) If x is live on exit from block B, then there is an acyclic path from B to a use of x, along which there are no definitions of x.

9.7 Region-Based Analysis

The iterative data-flow analysis algorithm we have discussed so far is just one approach to solving data-flow problems. Here we discuss another approach called *region-based analysis*. Recall that in the iterative-analysis approach, we create transfer functions for basic blocks, then find the fixedpoint solution by repeated passes over the blocks. Instead of creating transfer functions just for individual blocks, a region-based analysis finds transfer functions that summarize the execution of progressively larger regions of the program. Ultimately, transfer functions for entire procedures are constructed and then applied, to get the desired data-flow values directly.

While a data-flow framework using an iterative algorithm is specified by a semilattice of data-flow values and a family of transfer functions closed under composition, region-based analysis requires more elements. A region-based framework includes both a semilattice of data-flow values and a semilattice of transfer functions that must possess a meet operator, a composition operator, and a closure operator. We shall see what all these elements entail in Section 9.7.4.

A region-based analysis is particularly useful for data-flow problems where paths that have cycles may change the data-flow values. The closure operator allows the effect of a loop to be summarized more effectively than does iterative analysis. The technique is also useful for interprocedural analysis, where transfer functions associated with a procedure call may be treated like the transfer functions associated with basic blocks.

For simplicity, we shall consider only forward data-flow problems in this section. We first illustrate how region-based analysis works by using the familiar example of reaching definitions. In Section 9.8 we show a more compelling use of this technique, when we study the analysis of induction variables.

9.7.1 Regions

In region-based analysis, a program is viewed as a hierarchy of *regions*, which are (roughly) portions of a flow graph that have only one point of entry. We should find this concept of viewing code as a hierarchy of regions intuitive, because a block-structured procedure is naturally organized as a hierarchy of regions. Each statement in a block-structured program is a region, as control flow can only enter at the beginning of a statement. Each level of statement nesting corresponds to a level in the region hierarchy.

Formally, a *region* of a flow graph is a collection of nodes N and edges E such that

1. There is a header h in N that dominates all the nodes in N.

2. If some node m can reach a node n in N without going through h, then m is also in N.

3. E is the set of all the control flow edges between nodes n_1 and n_2 in N, except (possibly) for some that enter h.

Example 9.50 : Clearly a natural loop is a region, but a region does not necessarily have a back edge and need not contain any cycles. For example, in Fig. 9.47, nodes B_1 and B_2, together with the edge $B_1 \rightarrow B_2$, form a region; so do nodes B_1, B_2, and B_3 with edges $B_1 \rightarrow B_2$, $B_2 \rightarrow B_3$, and $B_1 \rightarrow B_3$.

However, the subgraph with nodes B_2 and B_3 with edge $B_2 \rightarrow B_3$ does not form a region, because control may enter the subgraph at both nodes B_2 and B_3. More precisely, neither B_2 nor B_3 dominates the other, so condition (1) for a region is violated. Even if we picked, say, B_2 to be the "header," we would violate condition (2), since we can reach B_3 from B_1 without going through B_2, and B_1 is not in the "region." $\Box$

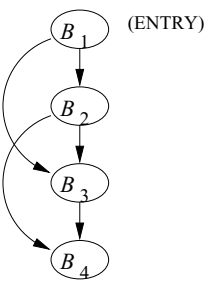


Figure 9.47: Examples of regions

9.7.2 Region Hierarchies for Reducible Flow Graphs

In what follows, we shall assume the flow graph is reducible. If occasionally we must deal with nonreducible flow graphs, then we can use a technique called "node splitting" that will be discussed in Section 9.7.6.

To construct a hierarchy of regions, we identify the natural loops. Recall from Section 9.6.6 that in a reducible flow graph, any two natural loops are either disjoint or one is nested within the other. The process of "parsing" a reducible flow graph into its hierarchy of loops begins with every block as a region by itself. We call these regions *leaf regions*. Then, we order the natural loops from the inside out, i.e., starting with the innermost loops. To process a loop, we replace the entire loop by a node in two steps:

1. First, the *body* of the loop L (all nodes and edges except the back edges to the header) is replaced by a node representing a region R. Edges to the header of L now enter the node for R. An edge from any exit of loop L is replaced by an edge from R to the same destination. However, if the edge is a back edge, then it becomes a loop on R. We call R a *body region*.

2. Next, we construct a region R' that represents the entire natural loop L. We call R' a *loop region*. The only difference between R and R' is that the latter includes the back edges to the header of loop L. Put another way, when R' replaces R in the flow graph, all we have to do is remove the edge from R to itself.

We proceed this way, reducing larger and larger loops to single nodes, first with a looping edge and then without. Since loops of a reducible flow graph are nested or disjoint, the loop region's node can represent all the nodes of the natural loop in the series of flow graphs that are constructed by this reduction process.

Eventually, all natural loops are reduced to single nodes. At that point, the flow graph may be reduced to a single node, or there may be several nodes remaining, with no loops; i.e., the reduced flow graph is an acyclic graph of more than one node. In the former case we are done constructing the region hierarchy, while in the latter case, we construct one more body region for the entire flow graph.

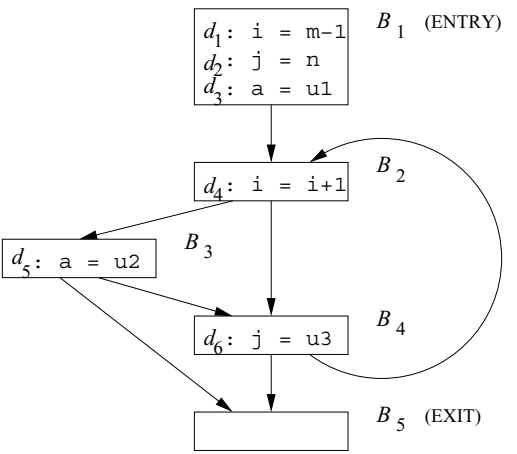
Example 9.51 : Consider the control flow graph in Fig. 9.48(a). There is one back edge in this flow graph, which leads from B_4 to B_2. The hierarchy of regions is shown in Fig. 9.48(b); the edges shown are the edges in the region flow graphs. There are altogether 8 regions:

1. Regions $R_1, \ldots, R_5$ are leaf regions representing blocks B_1 through B_5, respectively. Every block is also an exit block in its region.

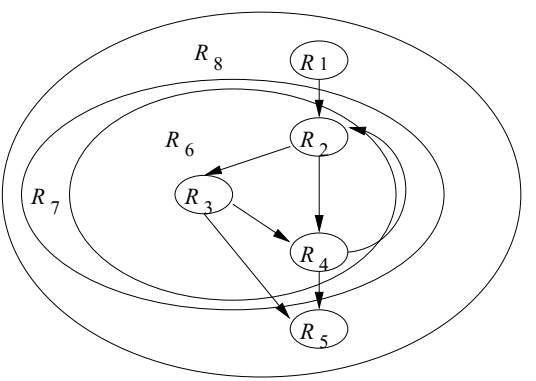
2. Body region R_6 represents the body of the only loop in the flow graph; it consists of regions R_2, R_3, and R_4 and three interregion edges: $B_2 \rightarrow B_3$, $B_2 \rightarrow B_4$, and $B_3 \rightarrow B_4$. It has two exit blocks, B_3 and B_4, since they both have outgoing edges not contained in the region. Figure 9.49(a) shows the flow graph with R_6 reduced to a single node. Notice that although the edges $R_3 \rightarrow R_5$ and $R_4 \rightarrow R_5$ have both been replaced by edge $R_6 \rightarrow R_5$, it is important to remember that the latter edge represents the two former edges, since we shall have to propagate transfer functions across this edge eventually, and we need to know that what comes out of both blocks B_3 and B_4 will reach the header of R_5.

3. Loop region R_7 represents the entire natural loop. It includes one subregion, R_6, and one back edge $B_4 \rightarrow B_2$. It has also two exit nodes, again B_3 and B_4. Figure 9.49(b) shows the flow graph after the entire natural loop is reduced to R_7.

4. Finally, body region R_8 is the top region. It includes three regions, R_1, R_7, R_5 and three interregion edges, $B_1 \rightarrow B_2$, $B_3 \rightarrow B_5$, and $B_4 \rightarrow B_5$. When we reduce the flow graph to R_8, it becomes a single node. Since there are no back edges to its header, B_1, there is no need for a final step reducing this body region to a loop region.



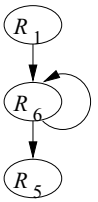
(a)



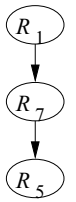
(b)

Figure 9.48: (a) An example flow graph for the reaching definitions problem and (b) Its region hierarchy

□



(a) After reducing to a body region



(b) After reducing to a loop region

Figure 9.49: Steps in the reduction of the flow graph of Fig. 9.48 to a single region

To summarize the process of decomposing reducible flow graphs hierarchically, we offer the following algorithm.

Algorithm 9.52: Constructing a bottom-up order of regions of a reducible flow graph.

INPUT: A reducible flow graph G.

OUTPUT: A list of regions of G that can be used in region-based data-flow problems.

METHOD:

1. Begin the list with all the leaf regions consisting of single blocks of G, in any order.
2. Repeatedly choose a natural loop L such that if there are any natural loops contained within L, then these loops have had their body and loop regions added to the list already. Add first the region consisting of the body of L (i.e., L without the back edges to the header of L), and then the loop region of L.
3. If the entire flow graph is not itself a natural loop, add at the end of the list the region consisting of the entire flow graph.

□

9.7.3 Overview of a Region-Based Analysis

For each region R, and for each subregion R' within R, we compute a transfer function $f_{R,\text{IN}[R']}$ that summarizes the effect of executing all possible paths

Where "Reducible" Comes From

We now see why reducible flow graphs were given that name. While we shall not prove this fact, the definition of "reducible flow graph" used in this book, involving the back edges of the graph, is equivalent to several definitions in which we mechanically reduce the flow graph to a single node. The process of collapsing natural loops described in Section 9.7.2 is one of them. Another interesting definition is that the reducible flow graphs are all and only those graphs that can be reduced to a single node by the following two transformations:

T_1: Remove an edge from a node to itself.

T_2: If node n has a single predecessor m, and n is not the entry of the flow graph, combine m and n.

leading from the entry of R to the entry of R', while staying within R. We say that a block B within R is an *exit block* of region R if it has an outgoing edge to some block outside R. We also compute a transfer function for each exit block B of R, denoted $f_{R,\mathrm{OUT}[B]}$, that summarizes the effect of executing all possible paths within R, leading from the entry of R to the exit of B.

We then proceed up the region hierarchy, computing transfer functions for progressively larger regions. We begin with regions that are single blocks, where $f_{B,\mathrm{IN}[B]}$ is just the identity function and $f_{B,\mathrm{OUT}[B]}$ is the transfer function for the block B itself. As we move up the hierarchy,

- If R is a body region, then the edges belonging to R form an acyclic graph on the subregions of R. We may proceed to compute the transfer functions in a topological order of the subregions.

- If R is a loop region, then we only need to account for the effect of the back edges to the header of R.

Eventually, we reach the top of the hierarchy and compute the transfer functions for region R_n that is the entire flow graph. How we perform each of these computations will be seen in Algorithm 9.53.

The next step is to compute the data-flow values at the entry and exit of each block. We process the regions in the reverse order, starting with region R_n and working our way down the hierarchy. For each region, we compute the data-flow values at the entry. For region R_n, we apply $f_{R_n,\mathrm{IN}[R]}(\mathrm{IN}[\mathrm{ENTRY}])$ to get the data-flow values at the entry of the subregions R in R_n. We repeat until we reach the basic blocks at the leaves of the region hierarchy.

9.7.4 Necessary Assumptions About Transfer Functions

In order for region-based analysis to work, we need to make certain assumptions about properties of the set of transfer functions in the framework. Specifically, we need three primitive operations on transfer functions: composition, meet and closure; only the first is required for data-flow frameworks that use the iterative algorithm.

Composition

The transfer function of a sequence of nodes can be derived by composing the functions representing the individual nodes. Let f_1 and f_2 be transfer functions of nodes n_1 and n_2. The effect of executing n_1 followed by n_2 is represented by $f_2 \circ f_1$. Function composition has been discussed in Section 9.2.2, and an example using reaching definitions was shown in Section 9.2.4. To review, let gen_i and $kill_i$ be the *gen* and *kill* sets for f_i. Then:

$$\begin{aligned} f_2 \circ f_1(x) &= gen_2 \cup \Big(\big(gen_1 \cup (x - kill_1) \big) - kill_2 \Big) \\ &= \big(gen_2 \cup (gen_1 - kill_2) \big) \cup \big(x - (kill_1 \cup kill_2) \big) \end{aligned}$$

Thus, the *gen* and *kill* sets for $f_2 \circ f_1$ are $gen_2 \cup (gen_1 - kill_2)$ and $kill_1 \cup kill_2$, respectively. The same idea works for any transfer function of the gen-kill form. Other transfer functions may also be closed, but we have to consider each case separately.

Meet

Here, the transfer functions themselves are values of a semilattice with a meet operator $\wedge_f$. The meet of two transfer functions f_1 and f_2, $f_1 \wedge_f f_2$, is defined by $(f_1 \wedge_f f_2)(x) = f_1(x) \wedge f_2(x)$, where $\wedge$ is the meet operator for data-flow values. The meet operator on transfer functions is used to combine the effect of alternative paths of execution with the same end points. Where it is not ambiguous, from now on, we shall refer to the meet operator of transfer functions also as $\wedge$. For the reaching-definitions framework, we have

$$\begin{aligned} (f_1 \wedge f_2)(x) &= f_1(x) \wedge f_2(x) \\ &= \big(gen_1 \cup (x - kill_1) \big) \cup \big(gen_2 \cup (x - kill_2) \big) \\ &= (gen_1 \cup gen_2) \cup \big(x - (kill_1 \cap kill_2) \big) \end{aligned}$$

That is, the *gen* and *kill* sets for $f_1 \wedge f_2$ are $gen_1 \cup gen_2$ and $kill_1 \cap kill_2$, respectively. Again, the same argument applies to any set of gen-kill transfer functions.

Closure

If f represents the transfer function of a cycle, then f^n represents the effect of going around the cycle n times. In the case where the number of iterations is not known, we have to assume that the loop may be executed 0 or more times. We represent the transfer function of such a loop by f^*, the *closure* of f, which is defined by

$$f^* = \bigwedge_{n \geq 0} f^n.$$

Note that f^0 must be the identity transfer function, since it represents the effect of going zero times around the loop, i.e., starting at the entry and not moving. If we let I represent the identity transfer function, then we can write

$$f^* = I \wedge (\bigwedge_{n > 0} f^n).$$

Suppose the transfer function f in a reaching definitions framework has a *gen* set and a *kill* set. Then,

$$\begin{aligned} f^2(x) &= f\big(f(x)\big) \\ &= gen \cup \Big(\big(gen \cup (x - kill)\big) - kill\Big) \\ &= gen \cup (x - kill) \\ f^3(x) &= f\big(f^2(x)\big) \\ &= gen \cup (x - kill) \end{aligned}$$

and so on: any $f^n(x)$ is $gen \cup (x - kill)$. That is, going around a loop doesn't affect the transfer function, if it is of the *gen-kill* form. Thus,

$$\begin{aligned} f^*(x) &= I \wedge f^1(x) \wedge f^2(x) \wedge \ldots \\ &= x \cup \big(gen \cup (x - kill)\big) \\ &= gen \cup x \end{aligned}$$

That is, the *gen* and *kill* sets for f^* are *gen* and $\emptyset$, respectively. Intuitively, since we might not go around a loop at all, anything in x will reach the entry to the loop. In all subsequent iterations, the reaching definitions include those in the *gen* set.

9.7.5 An Algorithm for Region-Based Analysis

The following algorithm solves a forward data-flow-analysis problem on a reducible flow graph, according to some framework that satisfies the assumptions of Section 9.7.4. Recall that $f_{R,\text{IN}[R']}$ and $f_{R,\text{OUT}[B]}$ refer to transfer functions that transform data-flow values at the entry to region R into the correct value at the entry of subregion R' and the exit of the exit block B, respectively.

Algorithm 9.53: Region-based analysis.

INPUT: A data-flow framework with the properties outlined in Section 9.7.4 and a reducible flow graph G.

OUTPUT: Data-flow values $\text{IN}[B]$ for each block B of G.

METHOD:

1. Use Algorithm 9.52 to construct the bottom-up sequence of regions of G, say $R_1, R_2, \ldots, R_n$, where R_n is the topmost region.

2. Perform the bottom-up analysis to compute the transfer functions summarizing the effect of executing a region. For each region $R_1, R_2, \ldots, R_n$, in the bottom-up order, do the following:

   (a) If R is a leaf region corresponding to block B, let $f_{R,\text{IN}[B]} = I$, and $f_{R,\text{OUT}[B]} = f_B$, the transfer function associated with block B.

   (b) If R is a body region, perform the computation of Fig. 9.50(a).

   (c) If R is a loop region, perform the computation of Fig. 9.50(b).

3. Perform the top-down pass to find the data-flow values at the beginning of each region.

   (a) $\text{IN}[R_n] = \text{IN}[\text{ENTRY}]$.

   (b) For each region R in $\{R_1, \ldots R_{n-1}\}$, in the top-down order, compute $\text{IN}[R] = f_{R',\text{IN}[R]}(\text{IN}[R'])$, where R' is the immediate enclosing region of R.

Let us first look at the details of how the bottom-up analysis works. In line (1) of Fig. 9.50(a) we visit the subregions of a body region, in some topological order. Line (2) computes the transfer function representing all the possible paths from the header of R to the header of S; then in lines (3) and (4) we compute the transfer functions representing all the possible paths from the header of R to the exits of R — that is, to the exits of all blocks that have successors outside S. Notice that all the predecessors B' in R must be in regions that precede S in the topological order constructed at line (1). Thus, $f_{R,\text{OUT}[B']}$ will have been computed already, in line (4) of a previous iteration through the outer loop.

For loop regions, we perform the steps of lines (1) through (4) in Fig. 9.50(b) Line (2) computes the effect of going around the loop body region S zero or more times. Lines (3) and (4) compute the effect at the exits of the loop after one or more iterations.

In the top-down pass of the algorithm, step 3(a) first assigns the boundary condition to the input of the top-most region. Then if R is immediately contained in R', we can simply apply the transfer function $f_{R',\text{IN}[R]}$ to the data-flow value $\text{IN}[R']$ to compute $\text{IN}[R]$. □

1) **for** (each subregion S immediately contained in R, in topological order) {
2) $f_{R,\text{IN}[S]} = \bigwedge_{\text{predecessors } B \text{ in } R \text{ of the header of } S} f_{R,\text{OUT}[B]}$;
/* if S is the header of region R, then $f_{R,\text{IN}[S]}$ is the meet over nothing, which is the identity function */
3) **for** (each exit block B in S)
4) $f_{R,\text{OUT}[B]} = f_{S,\text{OUT}[B]} \circ f_{R,\text{IN}[S]}$;
}

(a) Constructing transfer functions for a body region R

1) **let** S be the body region immediately nested within R; that is, S is R without back edges from R to the header of R;
2) $f_{R,\text{IN}[S]} = \left(\bigwedge_{\text{predecessors } B \text{ in } R \text{ of the header of } S} f_{S,\text{OUT}[B]}\right)^*$;
3) **for** (each exit block B in R)
4) $f_{R,\text{OUT}[B]} = f_{S,\text{OUT}[B]} \circ f_{R,\text{IN}[S]}$;

(b) Constructing transfer functions for a loop region R'

Figure 9.50: Details of region-based data-flow computations

Example 9.54: Let us apply Algorithm 9.53 to find reaching definitions in the flow graph in Fig. 9.48(a). Step 1 constructs the bottom-up order in which the regions are visited; this order will be the numerical order of their subscripts, $R_1, R_2, \ldots, R_n$.

The values of the *gen* and *kill* sets for the five blocks are summarized below:

B	B_1	B_2	B_3	B_4	B_5
gen_B	$\{d_1, d_2, d_3\}$	$\{d_4\}$	$\{d_5\}$	$\{d_6\}$	$\emptyset$
$kill_B$	$\{d_4, d_5, d_6\}$	$\{d_1\}$	$\{d_3\}$	$\{d_2\}$	$\emptyset$

Remember the simplified rules for *gen-kill* transfer functions, from Section 9.7.4:

- To take the meet of transfer functions, take the union of the *gen*'s and the intersection of the *kill*'s.
- To compose transfer functions, take the union of both the *gen*'s and the *kill*'s. However, as an exception, an expression that is generated by the first function, not generated by the second, but killed by the second is *not* in the *gen* of the result.
- To take the closure of a transfer function, retain its *gen* and replace the *kill* by $\emptyset$.

The first five regions $R_1, \ldots, R_5$ are blocks $B_1, \ldots, B_5$, respectively. For $1 \leq i \leq 5$, $f_{R_i,\text{IN}[B_i]}$ is the identity function, and $f_{R_i,\text{OUT}[B_i]}$ is the transfer function for block B_i:

$$f_{B_i,\text{OUT}[B_i]}(x) = (x - kill_{B_i}) \cup gen_{B_i}.$$

	Transfer Function		*gen*	*kill*
R_6	$f_{R_6,\text{IN}[R_2]}$	$= I$	$\emptyset$	$\emptyset$
	$f_{R_6,\text{OUT}[B_2]}$	$= f_{R_2,\text{OUT}[B_2]} \circ f_{R_6,\text{IN}[R_2]}$	$\{d_4\}$	$\{d_1\}$
	$f_{R_6,\text{IN}[R_3]}$	$= f_{R_6,\text{OUT}[B_2]}$	$\{d_4\}$	$\{d_1\}$
	$f_{R_6,\text{OUT}[B_3]}$	$= f_{R_3,\text{OUT}[B_3]} \circ f_{R_6,\text{IN}[R_3]}$	$\{d_4, d_5\}$	$\{d_1, d_3\}$
	$f_{R_6,\text{IN}[R_4]}$	$= f_{R_6,\text{OUT}[B_2]} \wedge f_{R_6,\text{OUT}[B_3]}$	$\{d_4, d_5\}$	$\{d_1\}$
	$f_{R_6,\text{OUT}[B_4]}$	$= f_{R_4,\text{OUT}[B_4]} \circ f_{R_6,\text{IN}[R_4]}$	$\{d_4, d_5, d_6\}$	$\{d_1, d_2\}$
R_7	$f_{R_7,\text{IN}[R_6]}$	$= f^*_{R_6,\text{OUT}[B_4]}$	$\{d_4, d_5, d_6\}$	$\emptyset$
	$f_{R_7,\text{OUT}[B_3]}$	$= f_{R_6,\text{OUT}[B_3]} \circ f_{R_7,\text{IN}[R_6]}$	$\{d_4, d_5, d_6\}$	$\{d_1, d_3\}$
	$f_{R_7,\text{OUT}[B_4]}$	$= f_{R_6,\text{OUT}[B_4]} \circ f_{R_7,\text{IN}[R_6]}$	$\{d_4, d_5, d_6\}$	$\{d_1, d_2\}$
R_8	$f_{R_8,\text{IN}[R_1]}$	$= I$	$\emptyset$	$\emptyset$
	$f_{R_8,\text{OUT}[B_1]}$	$= f_{R_1,\text{OUT}[B_1]}$	$\{d_1, d_2, d_3\}$	$\{d_4, d_5, d_6\}$
	$f_{R_8,\text{IN}[R_7]}$	$= f_{R_8,\text{OUT}[B_1]}$	$\{d_1, d_2, d_3\}$	$\{d_4, d_5, d_6\}$
	$f_{R_8,\text{OUT}[B_3]}$	$= f_{R_7,\text{OUT}[B_3]} \circ f_{R_8,\text{IN}[R_7]}$	$\{d_2, d_4, d_5, d_6\}$	$\{d_1, d_3\}$
	$f_{R_8,\text{OUT}[B_4]}$	$= f_{R_7,\text{OUT}[B_4]} \circ f_{R_8,\text{IN}[R_7]}$	$\{d_3, d_4, d_5, d_6\}$	$\{d_1, d_2\}$
	$f_{R_8,\text{IN}[R_5]}$	$= f_{R_8,\text{OUT}[B_3]} \wedge f_{R_8,\text{OUT}[B_4]}$	$\{d_2, d_3, d_4, d_5, d_6\}$	$\{d_1\}$
	$f_{R_8,\text{OUT}[B_5]}$	$= f_{R_5,\text{OUT}[B_5]} \circ f_{R_8,\text{IN}[R_5]}$	$\{d_2, d_3, d_4, d_5, d_6\}$	$\{d_1\}$

Figure 9.51: Computing transfer functions for the flow graph in Fig. 9.48(a), using region-based analysis

The rest of the transfer functions constructed in Step 2 of Algorithm 9.53 are summarized in Fig. 9.51. Region R_6, consisting of regions R_2, R_3, and R_4,

represents the loop body and thus does not include the back edge $B_4 \rightarrow B_2$. The order of processing these regions will be the only topological order: R_2, R_3, R_4. First, R_2 has no predecessors within R_6; remember that the edge $B_4 \rightarrow B_2$ goes outside R_6. Thus, $f_{R_6,\text{IN}[B_2]}$ is the identity function,[11] and $f_{R_6,\text{OUT}[B_2]}$ is the transfer function for block B_2 itself.

The header of region B_3 has one predecessor within R_6, namely R_2. The transfer function to its entry is simply the transfer function to the exit of B_2, $f_{R_6,\text{OUT}[B_2]}$, which has already been computed. We compose this function with the transfer function of B_3 within its own region to compute the transfer function to the exit of B_3.

Last, for the transfer function to the entry of R_4, we must compute

$$f_{R_6,\text{OUT}[B_2]} \wedge f_{R_6,\text{OUT}[B_3]}$$

because both B_2 and B_3 are predecessors of B_4, the header of R_4. This transfer function is composed with the transfer function $f_{R_4,\text{OUT}[B_4]}$ to get the desired function $f_{R_6,\text{OUT}[B_4]}$. Notice, for example, that d_3 is not killed in this transfer function, because the path $B_2 \rightarrow B_4$ does not redefine variable a.

Now, consider loop region R_7. It contains only one subregion R_6 which represents its loop body. Since there is only one back edge, $B_4 \rightarrow B_2$, to the header of R_6, the transfer function representing the execution of the loop body 0 or more times is just $f^*_{R_6,\text{OUT}[B_4]}$: the *gen* set is $\{d_4, d_5, d_6\}$ and the *kill* set is $\emptyset$. There are two exits out of region R_7, blocks B_3 and B_4. Thus, this transfer function is composed with each of the transfer functions of R_6 to get the corresponding transfer functions of R_7. Notice, for instance, how d_6 is in the *gen* set for $f_{R_7,\text{OUT}[B_3]}$ because of paths like $B_2 \rightarrow B_4 \rightarrow B_2 \rightarrow B_3$, or even $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2 \rightarrow B_3$.

Finally, consider R_8, the entire flow graph. Its subregions are R_1, R_7, and R_5, which we shall consider in that topological order. As before, the transfer function $f_{R_8,\text{IN}[B_1]}$ is simply the identity function, and the transfer function $f_{R_8,\text{OUT}[B_1]}$ is just $f_{R_1,\text{OUT}[B_1]}$, which in turn is f_{B_1}.

The header of R_7, which is B_2, has only one predecessor, B_1, so the transfer function to its entry is simply the transfer function out of B_1 in region R_8. We compose $f_{R_8,\text{OUT}[B_1]}$ with the transfer functions to the exits of B_3 and B_4 within R_7 to obtain their corresponding transfer functions within R_8. Lastly, we consider R_5. Its header, B_5, has two predecessors within R_8, namely B_3 and B_4. Therefore, we compute $f_{R_8,\text{OUT}[B_3]} \wedge f_{R_8,\text{OUT}[B_4]}$ to get $f_{R_8,\text{IN}[B_5]}$. Since the transfer function of block B_5 is the identity function, $f_{R_8,\text{OUT}[B_5]} = f_{R_8,\text{IN}[B_5]}$.

Step 3 computes the actual reaching definitions from the transfer functions. In step 3(a), $\text{IN}[R_8] = \emptyset$ since there are no reaching definitions at the beginning of the program. Figure 9.52 shows how step 3(b) computes the rest of the data-flow values. The step starts with the subregions of R_8. Since the transfer function from the start of R_8 to the start of each of its subregion has been

[11] Strictly speaking, we mean $f_{R_6,\text{IN}[R_2]}$, but when a region like R_2 is a single block, it is often clearer if we use the block name rather than the region name in this context.

computed, a single application of the transfer function finds the data-flow value at the start each subregion. We repeat the steps until we get the data-flow values of the leaf regions, which are simply the individual basic blocks. Note that the data-flow values shown in Figure 9.52 are exactly what we would get had we applied iterative data-flow analysis to the same flow graph, as must be the case, of course. □

$$
\begin{array}{lcll}
\text{IN}[R_8] & = & \emptyset & \\
\text{IN}[R_1] & = & f_{R_8,\text{IN}[R_1]}(\text{IN}[R_8]) & = \emptyset \\
\text{IN}[R_7] & = & f_{R_8,\text{IN}[R_7]}(\text{IN}[R_8]) & = \{d_1, d_2, d_3\} \\
\text{IN}[R_5] & = & f_{R_8,\text{IN}[R_5]}(\text{IN}[R_8]) & = \{d_2, d_3, d_4, d_5, d_6\} \\
\text{IN}[R_6] & = & f_{R_7,\text{IN}[R_6]}(\text{IN}[R_7]) & = \{d_1, d_2, d_3, d_4, d_5, d_6\} \\
\text{IN}[R_4] & = & f_{R_6,\text{IN}[R_4]}(\text{IN}[R_6]) & = \{d_2, d_3, d_4, d_5, d_6\} \\
\text{IN}[R_3] & = & f_{R_6,\text{IN}[R_3]}(\text{IN}[R_6]) & = \{d_2, d_3, d_4, d_5, d_6\} \\
\text{IN}[R_2] & = & f_{R_6,\text{IN}[R_2]}(\text{IN}[R_6]) & = \{d_1, d_2, d_3, d_4, d_5, d_6\}
\end{array}
$$

Figure 9.52: Final steps of region-based flow analysis

9.7.6 Handling Nonreducible Flow Graphs

If nonreducible flow graphs are expected to be common for the programs to be processed by a compiler or other program-processing software, then we recommend using an iterative rather than a hierarchy-based approach to data-flow analysis. However, if we need only to be prepared for the occasional nonreducible flow graph, then the following "node-splitting " technique is adequate.

Construct regions from natural loops to the extent possible. If the flow graph is nonreducible, we shall find that the resulting graph of regions has cycles, but no back edges, so we cannot parse the graph any further. A typical situation is suggested in Fig. 9.53(a), which has the same structure as the nonreducible flow graph of Fig. 9.45, but the nodes in Fig. 9.53 may actually be complex regions, as suggested by the smaller nodes within.

We pick some region R that has more than one predecessor and is not the header of the entire flow graph. If R has k predecessors, make k copies of the entire flow graph R, and connect each predecessor of R's header to a different copy of R. Remember that only the header of a region could possibly have a predecessor outside that region. It turns out, although we shall not prove it, that such node splitting results in a reduction by at least one in the number of regions, after new back edges are identified and their regions constructed. The resulting graph may still not be reducible, but by alternating a splitting phase with a phase where new natural loops are identified and collapsed to regions, we eventually are left with a single region; i.e., the flow graph has been reduced.

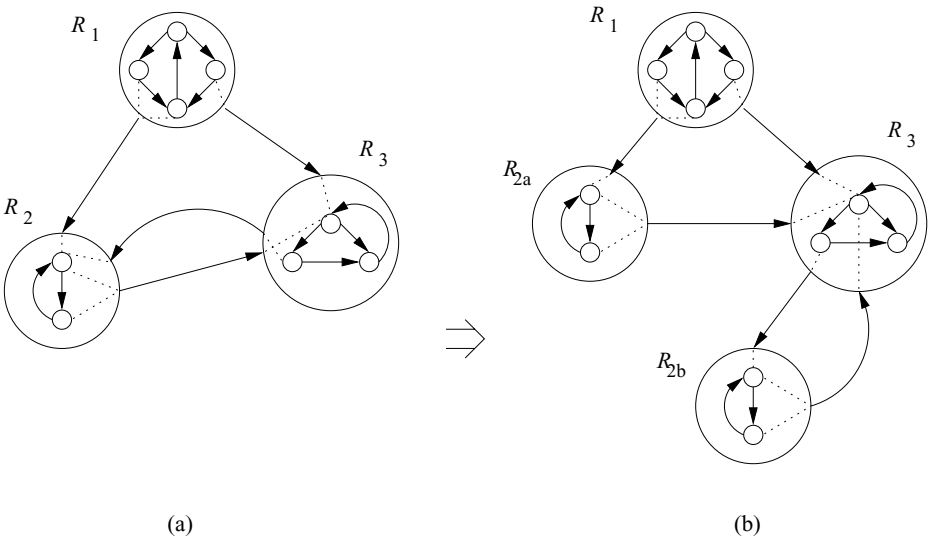


Figure 9.53: Duplicating a region to make a nonreducible flow graph become reducible

Example 9.55 : The splitting shown in Fig. 9.53(b) has turned the edge $R_{2b} \rightarrow R_3$ into a back edge, since R_3 now dominates R_{2b}. These two regions may thus be combined into one. The resulting three regions — R_1, R_{2a} and the new region — form an acyclic graph, and therefore may be combined into a single body region. We thus have reduced the entire flow graph to a single region. In general, additional splits may be necessary, and in the worst case, the total number of basic blocks could become exponential in the number of blocks in the original flow graph. □

We must also think about how the result of the data-flow analysis on the split flow graph relates to the answer we desire for the original flow graph. There are two approaches we might consider.

1. Splitting regions may be beneficial for the optimization process, and we can simply revise the flow graph to have copies of certain blocks. Since each duplicated block is entered along only a subset of the paths that reached the original, the data-flow values at these duplicated blocks will tend to contain more specific information than was available at the original. For instance, fewer definitions may reach each of the duplicated blocks that reach the original block.

2. If we wish to retain the original flow graph, with no splitting, then after analyzing the split flow graph, we look at each split block B, and its corresponding set of blocks $B_1, B_2, \ldots, B_k$. We may compute $\text{IN}[B] = \text{IN}[B_1] \wedge \text{IN}[B_2] \wedge \cdots \wedge \text{IN}[B_k]$, and similarly for the OUT's.

9.7.7 Exercises for Section 9.7

Exercise 9.7.1: For the flow graph of Fig. 9.10 (see the exercises for Section 9.1):

i. Find all the possible regions. You may, however, omit from the list the regions consisting of a single node and no edges.

ii. Give the set of nested regions constructed by Algorithm 9.52.

iii. Give a T_1-T_2 reduction of the flow graph as described in the box on "Where 'Reducible' Comes From" in Section 9.7.2.

Exercise 9.7.2: Repeat Exercise 9.7.1 on the following flow graphs:

a) Fig. 9.3.

b) Fig. 8.9.

c) Your flow graph from Exercise 8.4.1.

d) Your flow graph from Exercise 8.4.2.

Exercise 9.7.3: Prove that every natural loop is a region.

!! **Exercise 9.7.4:** Show that a flow graph is reducible if and only it can be transformed to a single node using:

a) The operations T_1 and T_2 described in the box in Section 9.7.2.

b) The region definition introduced in Section 9.7.2.

! **Exercise 9.7.5:** Show that when you apply node splitting to a nonreducible flow graph, and then perform T_1-T_2 reduction on the resulting split graph, you wind up with strictly fewer nodes than you started with.

! **Exercise 9.7.6:** What happens if you apply node-splitting and T_1-T_2 reduction alternately, to reduce a complete directed graph of n nodes?

9.8 Symbolic Analysis

We shall use symbolic analysis in this section to illustrate the use of region-based analysis. In this analysis, we track the values of variables in programs symbolically as expressions of input variables and other variables, which we call *reference variables*. Expressing variables in terms of the same set of reference variables draws out their relationships. Symbolic analysis can be used for a range of purposes such as optimization, parallelization, and analyses for program understanding.

```
1)  x = input();
2)  y = x-1;
3)  z = y-1;
4)  A[x] = 10;
5)  A[y] = 11;
6)  if (z > x)
7)        z = x;
```

Figure 9.54: An example program motivating symbolic analysis

Example 9.56: Consider the simple example program in Fig. 9.54. Here, we use x as the sole reference variable. Symbolic analysis will find that y has the value $x-1$ and z has the value $x-2$ after their respective assignment statements in lines (2) and (3). This information is useful, for example, in determining that the two assignments in lines (4) and (5) write to different memory locations and can thus be executed in parallel. Furthermore, we can tell that the condition $z > x$ is never true, thus allowing the optimizer to remove the conditional statement in lines (6) and (7) all together. □

9.8.1 Affine Expressions of Reference Variables

Since we cannot create succinct and closed-form symbolic expressions for all values computed, we choose an abstract domain and approximate the computations with the most precise expressions within the domain. We have already seen an example of this strategy before: constant propagation. In constant propagation, our abstract domain consists of the constants, an UNDEF symbol if we have not yet determined if the value is a constant, and a special NAC symbol that is used whenever a variable is found not to be a constant.

The symbolic analysis we present here expresses values as *affine* expressions of reference variables whenever possible. An expression is affine with respect to variables $v_1, v_2, \ldots, v_n$ if it can be expressed as $c_0 + c_1 v_1 + \cdots + c_n v_n$, where $c_0, c_1, \ldots, c_n$ are constants. Such expressions are informally known as linear expressions. Strictly speaking, an affine expression is linear only if c_0 is zero. We are interested in affine expressions because they are often used to index arrays in loops—such information is useful for optimizations and parallelization. Much more will be said about this topic in Chapter 11.

Induction Variables

Instead of using program variables as reference variables, an affine expression can also be written in terms of the count of iterations through the loop. Variables whose values can be expressed as $c_1 i + c_0$, where i is the count of iterations through the closest enclosing loop, are known as *induction variables*.

Example 9.57: Consider the code fragment

```
for (m = 10; m < 20; m++)
    { x = m*3; A[x] = 0; }
```

Suppose we introduce for the loop a variable, say i, representing the number of iterations executed. The value i is 0 in the first iteration of the loop, 1 in the second, and so on. We can express variable m as an affine expression of i, namely $m = i + 10$. Variable x, which is $3m$, takes on values $30, 33, \ldots, 57$ during successive iterations of the loop. Thus, x has the affine expression $x = 30 + 3i$. We conclude that both m and x are induction variables of this loop. □

Expressing variables as affine expressions of loop indexes makes the series of values being computed explicit and enables several transformations. The series of values taken on by an induction variable can be computed with additions rather than multiplications. This transformation is known as "strength reduction" and was introduced in Sections 8.7 and 9.1. For instance, we can eliminate the multiplication `x=m*3` from the loop of Example 9.57 by rewriting the loop as

```
x = 27;
for (m = 10; m < 20; m++)
    { x = x+3; A[x] = 0; }
```

In addition, notice that the locations assigned 0 in that loop, $\&A + 30$, $\&A + 33, \ldots, \&A + 57$, are also affine expressions of the loop index. In fact, this series of integers is the only one that needs to be computed. We do not need both m and x; for instance, the code above can be replaced by:

```
for (x = &A+30; x <= &A+57; x = x+3)
    *x = 0;
```

Besides speeding up the computation, symbolic analysis is also useful for parallelization. When the array indexes in a loop are affine expressions of loop indexes, we can reason about relations of data accessed across the iterations. For example, we can tell that the locations written are different in each iteration and therefore all the iterations in the loop can be executed in parallel on different processors. Such information is used in Chapters 10 and 11 to extract parallelism from sequential programs.

Other Reference Variables

If a variable is not a linear function of the reference variables already chosen, we have the option of treating its value as reference for future operations. For example, consider the code fragment:

```
a = f();
b = a + 10;
c = a + 11;
```

While the value held by a after the function call cannot itself be expressed as a linear function of any reference variables, it can be used as reference for subsequent statements. For example, using a as a reference variable, we can discover that c is one larger than b at the end of the program.

```
1)  a = 0;
2)  for (f = 100; f < 200; f++) {
3)        a = a + 1;
4)        b = 10 * a;
5)        c = 0;
6)        for (g = 10; g < 20; g++) {
7)              d = b + c;
8)              c = c + 1;
          }
    }
```

Figure 9.55: Source code for Example 9.58

Example 9.58 : Our running example for this section is based on the source code shown in Fig. 9.55. The inner and outer loops are easy to understand, since f and g are not modified except as required by the for-loops. It is thus possible to replace f and g by reference variables i and j that count the number of iterations of the outer and inner loops, respectively. That is, we can let $f = i + 99$ and $g = j + 9$, and substitute for f and g throughout. When translating to intermediate code, we can take advantage of the fact that each loop iterates at least once, and so postpone the test for $i \leq 100$ and $j \leq 10$ to the ends of the loops. Figure 9.56 shows the flow graph for the code of Fig. 9.55, after introducing i and j and treating the for-loops as if they were repeat-loops.

It turns out that a, b, c, and d are all induction variables. The sequences of values assigned to the variables in each line of the code are shown in Figure 9.57. As we shall see, it is possible to discover the affine expressions for these variables, in terms of the reference variables i and j. That is, at line (4) $a = i$, at line (7) $d = 10i + j - 1$, and at line (8), $c = j$. □

9.8.2 Data-Flow Problem Formulation

This analysis finds affine expressions of reference variables introduced (1) to count the number of iterations executed in each loop, and (2) to hold values at the entry of regions where necessary. This analysis also finds induction variables, loop invariants, as well as constants, as degenerate affine expressions. Note that this analysis cannot find all constants because it only tracks affine expressions of reference variables.

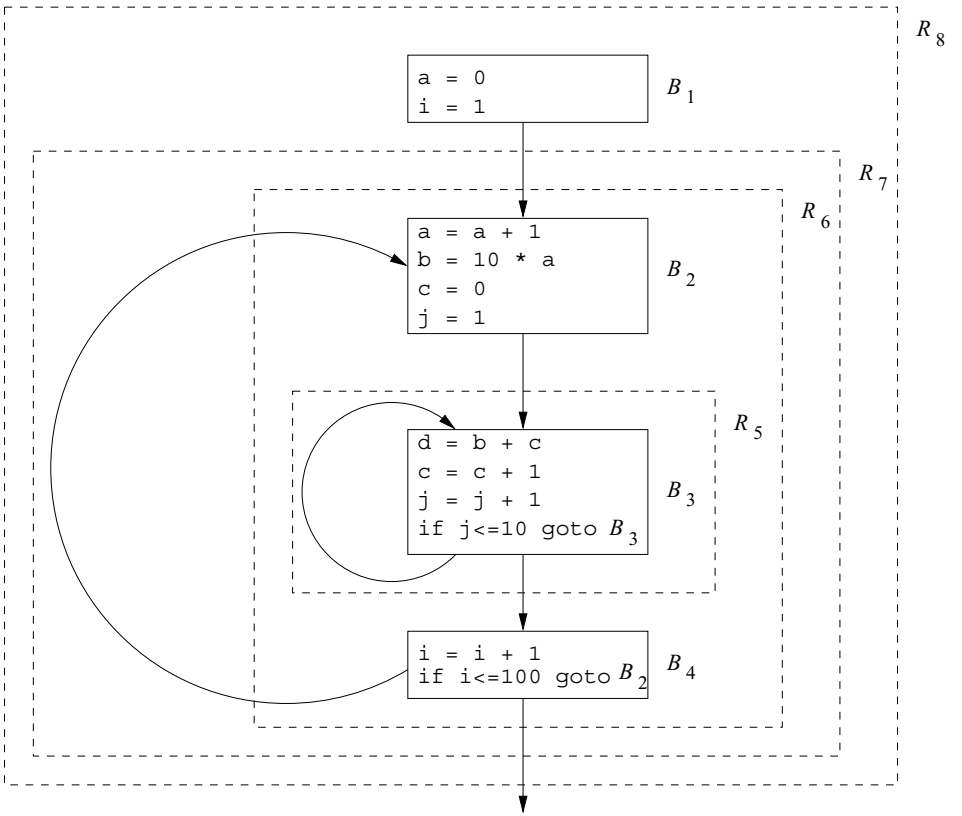


Figure 9.56: Flow graph and its region hierarchy for Example 9.58

Data-Flow Values: Symbolic Maps

The domain of data-flow values for this analysis is symbolic maps, which are functions that map each variable in the program to a value. The value is either an affine function of reference values, or the special symbol NAA to represent a non-affine expression. If there is only one variable, the bottom value of the semilattice is a map that sends the variable to NAA. The semilattice for n variables is simply the product of the individual semilattices. We use m_{NAA} to denote the bottom of the semilattice which maps all variables to NAA. We can define the symbolic map that sends all variables to an unknown value to be the top data-flow value, as we did for constant propagation. However, we do not need top values in region-based analysis.

Example 9.59 : The symbolic maps associated with each block for the code in Example 9.58 are shown in Figure 9.58. We shall see later how these maps are discovered; they are the result of doing region-based data-flow analysis on the flow graph of Fig. 9.56.

line	var	$i = 1$ $j = 1, \ldots, 10$	$i = 2$ $j = 1, \ldots, 10$	$1 \leq i \leq 100$ $j = 1, \ldots, 10$	$i = 100$ $j = 1, \ldots, 10$
3	a	1	2	i	100
4	b	10	20	$10i$	1000
7	d	$10, \ldots, 19$	$20, \ldots, 29$	$10i, \ldots, 10i + 9$	$1000, \ldots, 1009$
8	c	$1, \ldots, 10$	$1, \ldots, 10$	$1, \ldots, 10$	$1, \ldots, 10$

Figure 9.57: Sequence of values seen in program points in Example 9.58.

m	$m(a)$	$m(b)$	$m(c)$	$m(d)$
IN$[B_1]$	NAA	NAA	NAA	NAA
OUT$[B_1]$	0	NAA	NAA	NAA
IN$[B_2]$	$i - 1$	NAA	NAA	NAA
OUT$[B_2]$	i	$10i$	0	NAA
IN$[B_3]$	i	$10i$	$j - 1$	NAA
OUT$[B_3]$	i	$10i$	j	$10i + j - 1$
IN$[B_4]$	i	$10i$	j	$10i + j - 1$
OUT$[B_4]$	$i - 1$	$10i - 10$	j	$10i + j - 11$

Figure 9.58: Symbolic maps of the program in Example 9.58.

The symbolic map associated with the entry of the program is m_{NAA}. At the exit of B_1, the value of a is set to 0. Upon entry to block B_2, a has value 0 in the first iteration and increments by one in each subsequent iteration of the outer loop. Thus a has value $i - 1$ at the entry of the ith iteration and value i at the end. The symbolic map at the entry of B_2 maps variables b, c, d to NAA, because the variables have unknown values on entry to the outer loop. Their values depend on the number of iterations of the outer loop, so far. The symbolic map on exit from B_2 reflects the assignment statements to a, b, and c in that block. The rest of the symbolic maps can be deduced in a similar manner. Once we have established the validity of the maps in Fig. 9.58, we can replace each of the assignments to a, b, c, and d in Fig. 9.55 by the appropriate affine expressions. That is, we can replace Fig. 9.55 by the code in Fig. 9.59. □

Transfer Function of a Statement

The transfer functions in this data-flow problem send symbolic maps to symbolic maps. To compute the transfer function of an assignment statement, we interpret the semantics of the statement and determine if the assigned variable can be expressed as an affine expression of the values on the right of the

```
1)  a = 0;
2)  for (i = 1; i <= 100; i++) {
3)        a = i;
4)        b = 10*i;
5)        c = 0;
6)        for (j = 1; j <= 10; j++) {
7)              d = 10*i + j -1;
8)              c = j;
          }
    }
```

Figure 9.59: The code of Fig. 9.55 with assignments replaced by affine expressions of the reference variables i and j

Cautions Regarding Transfer Functions on Value Maps

A subtlety in the way we define the transfer functions on symbolic maps is that we have options regarding how the effects of a computation are expressed. When m is the map for the input of a transfer function, $m(x)$ is really just "whatever value variable x happens to have on entry". We try very hard to express the result of the transfer function as an affine expression of reference variables used by the input map.

You should observe the proper interpretation of expressions like $f(m)(x)$, where f is a transfer function, m a map, and x a variable. As is conventional in mathematics, we apply functions from the left, meaning that we first compute $f(m)$, which is a map. Since a map is a function, we may then apply it to variable x to produce a value.

assignment. The values of all other variables remain unchanged.

The transfer function of statement s, denoted f_s, is defined as follows:

1. If s is not an assignment statement, then f_s is the identity function.
2. If s is an assignment statement to variable x, then

$$f_s(m)(x) = \begin{cases} m(v) & \text{for all variables } v \neq x \\ c_0 + c_1 m(y) + c_2 m(z) & \text{if } x \text{ is assigned } c_0 + c_1 y + c_2 z, \\ & (c_1 = 0, \text{ or } m(y) \neq \text{NAA}), \text{ and} \\ & (c_2 = 0, \text{ or } m(z) \neq \text{NAA}) \\ \text{NAA} & \text{otherwise.} \end{cases}$$

The expression $c_0 + c_1 m(y) + c_2 m(z)$ is intended to represent all the possible forms of expressions involving arbitrary variables y and z that may appear on the right side of an assignment to x and that give x a value that is an affine transformation on prior values of variables. These expressions are: c_0, $c_0 + y$, $c_0 - y$, $y + z$, $x - y$, $c_1 * y$, and $y/(1/c_1)$. Note that in many cases, one or more of c_0, c_1, and c_2 are 0.

Example 9.60 : If the assignment is `x=y+z`, then $c_0 = 0$ and $c_1 = c_2 = 1$. If the assignment is `x=y/5`, then $c_0 = c_2 = 0$, and $c_1 = 1/5$. □

Composition of Transfer Functions

To compute $f_2 \circ f_1$, where f_1 and f_2 are defined in terms of input map m, we substitute the value of $m(v_i)$ in the definition of f_2 with the definition of $f_1(m)(v_i)$. We replace all operations on NAA values with NAA. That is,

1. If $f_2(m)(v) = \text{NAA}$, then $(f_2 \circ f_1)(m)(v) = \text{NAA}$.
2. If $f_2(m)(v) = c_0 + \sum_i c_i m(v_i)$, then

$$(f_2 \circ f_1)(m)(v)$$

$$= \begin{cases} \text{NAA}, & \text{if } f_1(m)(v_i) = \text{NAA for some } i \neq 0, c_i \neq 0 \\ c_0 + \sum_i c_i f_1(m)(v_i) & \text{otherwise} \end{cases}$$

Example 9.61 : The transfer functions of the blocks in Example 9.58 can be computed by composing the transfer functions of their constituent statements. These transfer functions are defined in Fig. 9.60. □

f	$f(m)(a)$	$f(m)(b)$	$f(m)(c)$	$f(m)(d)$
f_{B_1}	0	$m(b)$	$m(c)$	$m(d)$
f_{B_2}	$m(a)+1$	$10m(a)+10$	0	$m(d)$
f_{B_3}	$m(a)$	$m(b)$	$m(c)+1$	$m(b)+m(c)$
f_{B_4}	$m(a)$	$m(b)$	$m(c)$	$m(d)$

Figure 9.60: Transfer Functions of Example 9.58

Solution to Data-Flow Problem

We use the notation $\text{IN}_{i,j}[B_3]$ and $\text{OUT}_{i,j}[B_3]$ to refer to the input and output data-flow values of block B_3 in iteration j of the inner loop and iteration i of the outer loop. For the other blocks, we use $\text{IN}_i[B_k]$ and $\text{OUT}_i[B_k]$ to refer to these values in the ith iteration of the outer loop. Also, we can see that

$$
\begin{array}{rcll}
\text{OUT}[B_k] & = & f_B(\text{IN}[B_k]), & \text{for all } B_k \\
\text{OUT}[B_1] & \geq & \text{IN}_1[B_2] & \\
\text{OUT}_i[B_2] & \geq & \text{IN}_{i,1}[B_3], & 1 \leq i \leq 10 \\
\text{OUT}_{i,j-1}[B_3] & \geq & \text{IN}_{i,j}[B_3], & 1 \leq i \leq 100,\ 2 \leq j \leq 10 \\
\text{OUT}_{i,10}[B_3] & \geq & \text{IN}_i[B_4], & 2 \leq i \leq 100 \\
\text{OUT}_{i-1}[B_4] & \geq & \text{IN}_i[B_2], & 1 \leq i \leq 100
\end{array}
$$

Figure 9.61: Constraints satisfied on each iteration of the nested loops

the symbolic maps shown in Fig. 9.58 satisfy the constraints imposed by the transfer functions, listed in Fig. 9.61.

The first constraint says that the output map of a basic block is obtained by applying the block's transfer function to the input map. The rest of the constraints say that the output map of a basic block must be greater than or equal to the input map of a successor block in the execution.

Note that our iterative data-flow algorithm cannot produce the above solution because it lacks the concept of expressing data-flow values in terms of the number of iterations executed. Region-based analysis can be used to find such solutions, as we shall see in the next section.

9.8.3 Region-Based Symbolic Analysis

We can extend the region-based analysis described in Section 9.7 to find expressions of variables in the ith iteration of a loop. A region-based symbolic analysis has a bottom-up pass and a top-down pass, like other region-based algorithms. The bottom-up pass summarizes the effect of a region with a transfer function that sends a symbolic map at the entry to an output symbolic map at the exit. In the top-down pass, values of symbolic maps are propagated down to the inner regions.

The difference lies in how we handle loops. In Section 9.7, the effect of a loop is summarized with a closure operator. Given a loop with body f, its closure f^* is defined as an infinite meet of all possible numbers of applications of f. However, to find induction variables, we need to determine if a value of a variable is an affine function of the number of iterations executed so far. The symbolic map must be parameterized by the number of the iteration being executed. Furthermore, whenever we know the total number of iterations executed in a loop, we can use that number to find the values of induction variables after the loop. For instance, in Example 9.58, we claimed that a has the value of i after executing the ith iteration. Since the loop has 100 iterations, the value of a must be 100 at the end of the loop.

In what follows, we first define the primitive operators: meet and composition of transfer functions for symbolic analysis. Then show how we use them to perform region-based analysis of induction variables.

Meet of Transfer Functions

When computing the meet of two functions, the value of a variable is NAA unless the two functions map the variable to the same value and the value is not NAA. Thus,

$$(f_1 \wedge f_2)(m)(v) = \begin{cases} f_1(m)(v) & \text{if } f_1(m)(v) = f_2(m)(v) \\ \text{NAA} & \text{otherwise} \end{cases}$$

Parameterized Function Compositions

To express a variable as an affine function of a loop index, we need to compute the effect of composing a function some given number of times. If the effect of one iteration is summarized by transfer function f, then the effect of executing i iterations, for some $i \geq 0$, is denoted f^i. Note that when $i = 0$, $f^i = f^0 = I$, the identify function.

Variables in the program are divided into four categories:

1. If $f(m)(x) = m(x) + c$, where c is a constant, then $f^i(m)(x) = m(x) + ci$ for every value of $i \geq 0$. We say that x is a *basic induction variable* of the loop whose body is represented by the transfer function f.

2. If $f(m)(x) = m(x)$, then $f^i(m)(x) = m(x)$ for all $i \geq 0$. The variable x is not modified and it remains unchanged at the end of any number of iterations through the loop with transfer function f. We say that x is a *symbolic constant* in the loop.

3. If $f(m)(x) = c_0 + c_1 m(x_1) + \cdots + c_n m(x_n)$, where each x_k is either a basic induction variable or a symbolic constant, then for $i > 0$,

$$f^i(m)(x) = c_0 + c_1 f^i(m)(x_1) + \cdots + c_n f^i(m)(x_n).$$

   We say that x is also an induction variable, though not a basic one. Note that the formula above does not apply if $i = 0$.

4. In all other cases, $f^i(m)(x) = \text{NAA}$.

To find the effect of executing a fixed number of iterations, we simply replace i above by that number. In the case where the number of iterations is unknown, the value at the start of the last iteration is given by f^*. In this case, the only variables whose values can still be expressed in the affine form are the loop-invariant variables.

$$f^*(m)(v) = \begin{cases} m(v) & \text{if } f(m)(v) = m(v) \\ \text{NAA} & \text{otherwise} \end{cases}$$

Example 9.62: For the innermost loop in Example 9.58, the effect of executing i iterations, $i > 0$, is summarized by $f^i_{B_3}$. From the definition of f_{B_3}, we see that a and b are symbolic constants, c is a basic induction variable as it is

incremented by one every iteration. d is an induction variable because it is an affine function the symbolic constant b and basic induction variable c. Thus,

$$f^i_{B_3}(m)(v) = \begin{cases} m(a) & \text{if } v = a \\ m(b) & \text{if } v = b \\ m(c) + i & \text{if } v = c \\ m(b) + m(c) + i & \text{if } v = d. \end{cases}$$

If we could not tell how many times the loop of block B_3 iterated, then we could not use f^i and would have to use f^* to express the conditions at the end of the loop. In this case, we would have

$$f^*_{B_3}(m)(v) = \begin{cases} m(a) & \text{if } v = a \\ m(b) & \text{if } v = b \\ \text{NAA} & \text{if } v = c \\ \text{NAA} & \text{if } v = d. \end{cases}$$

□

A Region-Based Algorithm

Algorithm 9.63: Region-based symbolic analysis.

INPUT: A reducible flow graph G.

OUTPUT: Symbolic maps $\text{IN}[B]$ for each block B of G.

METHOD: We make the following modifications to Algorithm 9.53.

1. We change how we construct the transfer function for a loop region. In the original algorithm we use the $f_{R,\text{IN}[S]}$ transfer function to map the symbolic map at the entry of loop region R to a symbolic map at the entry of loop body S after executing an unknown number of iterations. It is defined to be the closure of the transfer function representing all paths leading back to the entry of the loop, as shown in Fig. 9.50(b). Here we define $f_{R,i,\text{IN}[S]}$ to represent the effect of execution from the start of the loop region to the entry of the ith iteration. Thus,

$$f_{R,i,\text{IN}[S]} = \Big(\bigwedge_{\text{predecessors } B \text{ in } R \text{ of the header of } S} f_{S,\text{OUT}[B]} \Big)^{i-1}$$

2. If the number of iterations of a region is known, the summary of the region is computed by replacing i with the actual count.

3. In the top-down pass, we compute $f_{R,i,\text{IN}[B]}$ to find the symbolic map associated with the entry of the ith iteration of a loop.

4. In the case where the input value of a variable $m(v)$ is used on the right-hand-side of a symbolic map in region R, and $m(v) =$ NAA upon entry to the region, we introduce a new reference variable t, add assignment `t = v` to the beginning of region R, and all references of $m(v)$ are replaced by t. If we did not introduce a reference variable at this point, the NAA value held by v would penetrate into inner loops.

□

$$
\begin{aligned}
f_{R_5,j,\mathrm{IN}[B_3]} &= f_{B_3}^{j-1} \\
f_{R_5,j,\mathrm{OUT}[B_3]} &= f_{B_3}^{j}
\end{aligned}
$$

$$
\begin{aligned}
f_{R_6,\mathrm{IN}[B_2]} &= I \\
f_{R_6,\mathrm{IN}[R_5]} &= f_{B_2} \\
f_{R_6,\mathrm{OUT}[B_4]} &= I \circ f_{R_5,10,\mathrm{OUT}[B_3]} \circ f_{B_2}
\end{aligned}
$$

$$
\begin{aligned}
f_{R_7,i,\mathrm{IN}[R_6]} &= f_{R_6,\mathrm{OUT}[B_4]}^{i-1} \\
f_{R_7,i,\mathrm{OUT}[B_4]} &= f_{R_6,\mathrm{OUT}[B_4]}^{i}
\end{aligned}
$$

$$
\begin{aligned}
f_{R_8,\mathrm{IN}[B_1]} &= I \\
f_{R_8,\mathrm{IN}[R_7]} &= f_{B_1} \\
f_{R_8,\mathrm{OUT}[B_4]} &= f_{R_7,100,\mathrm{OUT}[B_4]} \circ f_{B_1}
\end{aligned}
$$

Figure 9.62: Transfer function relations in the bottom-up pass for Example 9.58.

Example 9.64: For Example 9.58, we show how the transfer functions for the program are computed in the bottom-up pass in Fig. 9.62. Region R_5 is the inner loop, with body B_5. The transfer function representing the path from the entry of region R_5 to the beginning of the jth iteration, $j \geq 1$, is $f_{B_3}^{j-1}$. The transfer function representing the path to the end of the jth iteration, $j \geq 1$, is $f_{B_3}^{j}$.

Region R_6 consists of blocks B_2 and B_4, with loop region R_5 in the middle. The transfer functions from the entry of B_2 and R_5 can be computed in the same way as in the original algorithm. Transfer function $f_{R_6,\mathrm{OUT}[B_3]}$ represents the composition of block B_2 and the entire execution of the inner loop, since f_{B_4} is the identity function. Since the inner loop is known to iterate 10 times, we can replace j by 10 to summarize the effect of the inner loop precisely. The

f	$f(m)(a)$	$f(m)(b)$	$f(m)(c)$	$f(m)(d)$
$f_{R_5,j,\text{IN}[B_3]}$	$m(a)$	$m(b)$	$m(c)+j-1$	NAA
$f_{R_5,j,\text{OUT}[B_3]}$	$m(a)$	$m(b)$	$m(c)+j$	$m(b)+m(c)+j-1$
$f_{R_6,\text{IN}[B_2]}$	$m(a)$	$m(b)$	$m(c)$	$m(d)$
$f_{R_6,\text{IN}[R_5]}$	$m(a)+1$	$10m(a)+10$	0	$m(d)$
$f_{R_6,\text{OUT}[B_4]}$	$m(a)+1$	$10m(a)+10$	10	$10m(a)+9$
$f_{R_7,i,\text{IN}[R_6]}$	$m(a)+i-1$	NAA	NAA	NAA
$f_{R_7,i,\text{OUT}[B_4]}$	$m(a)+i$	$10m(a)+10i$	10	$10m(a)+10i+9$
$f_{R_8,\text{IN}[B_1]}$	$m(a)$	$m(b)$	$m(c)$	$m(d)$
$f_{R_8,\text{IN}[R_7]}$	0	$m(b)$	$m(c)$	$m(d)$
$f_{R_8,\text{OUT}[B_4]}$	100	1000	10	1009

Figure 9.63: Transfer functions computed in the bottom-up pass for Example 9.58

rest of the transfer functions can be computed in a similar manner. The actual transfer functions computed are shown in Fig. 9.63.

The symbolic map at the entry of the program is simply m_{NAA}. We use the top-down pass to compute the symbolic map to the entry to successively nested regions until we find all the symbolic maps for every basic block. We start by computing the data-flow values for block B_1 in region R_8:

$$\begin{aligned} \text{IN}[B_1] &= m_{\text{NAA}} \\ \text{OUT}[B_1] &= f_{B_1}(\text{IN}[B_1]) \end{aligned}$$

Descending down to regions R_7 and R_6, we get

$$\begin{aligned} \text{IN}_i[B_2] &= f_{R_7,i,\text{IN}[R_6]}(\text{OUT}[B_1]) \\ \text{OUT}_i[B_2] &= f_{B_2}(\text{IN}_i[B_2]) \end{aligned}$$

Finally, in region R_5, we get

$$\begin{aligned} \text{IN}_{i,j}[B_3] &= f_{R_5,j,\text{IN}[B_3]}(\text{OUT}_i[B_2]) \\ \text{OUT}_{i,j}[B_3] &= f_{B_3}(\text{IN}_{i,j}[B_3]) \end{aligned}$$

Not surprisingly, these equations produce the results we showed in Fig. 9.58. □

Example 9.58 shows a simple program where every variable used in the symbolic map has an affine expression. We use Example 9.65 to illustrate why and how we introduce reference variables in Algorithm 9.63.

```
1)  for (i = 1; i < n; i++} {
2)      a = input();
3)      for (j = 1; j < 10; j++) {
4)        a = a - 1;
5)        b = j + a;
6)        a = a + 1;
        }
     }
```

(a) A loop where a fluctuates.

```
    for (i = 1; i < n; i++} {
        a = input();
        t = a;
        for (j = 1; j < 10; j++) {
          a = t - 1;
          b = t - 1 + j;
          a = t;
        }
    }
```

(b) A reference variable t makes b an induction variable.

Figure 9.64: The need to introduce reference variables

Example 9.65: Consider the simple example in Fig. 9.64(a). Let f_j be the transfer function summarizing the effect of executing j iterations of the inner loop. Even though the value of a may fluctuate during the execution of the loop, we see that b is an induction variable based on the value of a on entry of the loop; that is, $f_j(m)(b) = m(a) - 1 + j$. Because a is assigned an input value, the symbolic map upon entry to the inner loop maps a to NAA. We introduce a new reference variable t to save the value of a upon entry, and perform the substitutions as in Fig. 9.64(b). □

9.8.4 Exercises for Section 9.8

Exercise 9.8.1: For the flow graph of Fig. 9.10 (see the exercises for Section 9.1), give the transfer functions for

a) Block B_2.

b) Block B_4.

c) Block B_5.

Exercise 9.8.2: Consider the inner loop of Fig. 9.10, consisting of blocks B_3 and B_4. If i represents the number of times around the loop, and f is the transfer function for the loop body (i.e., excluding the edge from B_4 to B_3) from the entry of the loop (i.e., the beginning of B_3) to the exit from B_4, then what is f^i? Remember that f takes as argument a map m, and m assigns a value to each of variables a, b, d, and e. We denote these values $m(a)$, and so on, although we do not know their values.

! **Exercise 9.8.3:** Now consider the outer loop of Fig. 9.10, consisting of blocks B_2, B_3, B_4, and B_5. Let g be the transfer function for the loop body, from the entry of the loop at B_2 to its exit at B_5. Let i measure the number of iterations of the inner loop of B_3 and B_4 (which count of iterations we cannot know), and let j measure the number of iterations of the outer loop (which we also cannot know). What is g^j?

9.9 Summary of Chapter 9

- *Global Common Subexpressions*: An important optimization is finding computations of the same expression in two different basic blocks. If one precedes the other, we can store the result the first time it is computed and use the stored result on subsequent occurrences.

- *Copy Propagation*: A copy statement, $u = v$, assigns one variable v to another, u. In some circumstances, we can replace all uses of u by v, thus eliminating both the assignment and u.

- *Code Motion*: Another optimization is to move a computation outside the loop in which it appears. This change is only correct if the computation produces the same value each time around the loop.

- *Induction Variables*: Many loops have induction variables, variables that take on a linear sequence of values each time around the loop. Some of these are used only to count iterations, and they often can be eliminated, thus reducing the time it takes to go around the loop.

- *Data-Flow Analysis*: A data-flow analysis schema defines a value at each point in the program. Statements of the program have associated transfer functions that relate the value before the statement to the value after. Statements with more than one predecessor must have their value defined by combining the values at the predecessors, using a meet (or confluence) operator.

- *Data-Flow Analysis on Basic Blocks*: Because the propagation of data-flow values within a block is usually quite simple, data-flow equations are generally set up to have two variables for each block, called IN and OUT, that represent the data-flow values at the beginning and end of the

block, respectively. The transfer functions for the statements in a block are composed to get the transfer function for the block as a whole.

- *Reaching Definitions*: The reaching-definitions data-flow framework has values that are sets of statements in the program that define values for one or more variables. The transfer function for a block kills definitions of variables that are definitely redefined in the block and adds ("generates") those definitions of variables that occur within the block. The confluence operator is union, since definitions reach a point if they reach any predecessor of that point.

- *Live Variables*: Another important data-flow framework computes the variables that are live (will be used before redefinition) at each point. The framework is similar to reaching definitions, except that the transfer function runs backward. A variable is live at the beginning of a block if it is either used before definition in the block or is live at the end of the block and not redefined in the block.

- *Available Expressions*: To discover global common subexpressions, we determine the available expressions at each point — expressions that have been computed and neither of the expression's arguments were redefined after the last computation. The data-flow framework is similar to reaching definitions, but the confluence operator is intersection rather than union.

- *Abstraction of Data-Flow Problems*: Common data-flow problems, such as those already mentioned, can be expressed in a common mathematical structure. The values are members of a semilattice, whose meet is the confluence operator. Transfer functions map lattice elements to lattice elements. The set of allowed transfer functions must be closed under composition and include the identity function.

- *Monotone Frameworks*: A semilattice has a $\leq$ relation defined by $a \leq b$ if and only if $a \wedge b = a$. Monotone frameworks have the property that each transfer function preserves the $\leq$ relationship; that is, $a \leq b$ implies $f(a) \leq f(b)$, for all lattice elements a and b and transfer function f.

- *Distributive Frameworks*: These frameworks satisfy the condition that $f(a \wedge b) = f(a) \wedge f(b)$, for all lattice elements a and b and transfer function f. It can be shown that the distributive condition implies the monotone condition.

- *Iterative Solution to Abstract Frameworks*: All monotone data-flow frameworks can be solved by an iterative algorithm, in which the IN and OUT values for each block are initialized appropriately (depending on the framework), and new values for these variables are repeatedly computed by applying the transfer and confluence operations. This solution is always safe (optimizations that it suggests will not change what the

program does), but the solution is certain to be the best possible only if the framework is distributive.

- *The Constant Propagation Framework*: While the basic frameworks such as reaching definitions are distributive, there are interesting monotone-but-not-distributive frameworks as well. One involves propagating constants by using a semilattice whose elements are mappings from the program variables to constants, plus two special values that represent "no information" and "definitely not a constant."

- *Partial-Redundancy Elimination*: Many useful optimizations, such as code motion and global common-subexpression elimination, can be generalized to a single problem called partial-redundancy elimination. Expressions that are needed, but are available along only some of the paths to a point, are computed only along the paths where they are not available. The correct application of this idea requires the solution to a sequence of four different data-flow problems plus other operations.

- *Dominators*: A node in a flow graph dominates another if every path to the latter must go through the former. A proper dominator is a dominator other than the node itself. Each node except the entry node has an immediate dominator — that one of its proper dominators that is dominated by all the other proper dominators.

- *Depth-First Ordering of Flow Graphs*: If we perform a depth-first search of a flow graph, starting at its entry, we produce a depth-first spanning tree. The depth-first order of the nodes is the reverse of a postorder traversal of this tree.

- *Classification of Edges*: When we construct a depth-first spanning tree, all the edges of the flow graph can be divided into three groups: advancing edges (those that go from ancestor to proper descendant), retreating edges (those from descendant to ancestor) and cross edges (others). An important property is that all the cross edges go from right to left in the tree. Another important property is that of these edges, only the retreating edges have a head lower than its tail in the depth-first order (reverse postorder).

- *Back Edges*: A back edge is one whose head dominates its tail. Every back edge is a retreating edge, regardless of which depth-first spanning tree for its flow graph is chosen.

- *Reducible Flow Graphs*: If every retreating edge is a back edge, regardless of which depth-first spanning tree is chosen, then the flow graph is said to be reducible. The vast majority of flow graphs are reducible; those whose only control-flow statements are the usual loop-forming and branching statements are certainly reducible.

- *Natural Loops*: A natural loop is a set of nodes with a header node that dominates all the nodes in the set and has at least one back edge entering that node. Given any back edge, we can construct its natural loop by taking the head of the edge plus all nodes that can reach the tail of the edge without going through the head. Two natural loops with different headers are either disjoint or one is completely contained in the other; this fact lets us talk about a hierarchy of nested loops, as long as "loops" are taken to be natural loops.

- *Depth-First Order Makes the Iterative Algorithm Efficient*: The iterative algorithm requires few passes, as long as propagation of information along acyclic paths is sufficient; i.e., cycles add nothing. If we visit nodes in depth-first order, any data-flow framework that propagates information forward, e.g., reaching definitions, will converge in no more than 2 plus the largest number of retreating edges on any acyclic path. The same holds for backward-propagating frameworks, like live variables, if we visit in the reverse of depth-first order (i.e., in postorder).

- *Regions*: Regions are sets of nodes and edges with a header h that dominates all nodes in the region. The predecessors of any node other than h in the region must also be in the region. The edges of the region are all that go between nodes of the region, with the possible exception of some or all that enter the header.

- *Regions and Reducible Flow Graphs*: Reducible flow graphs can be parsed into a hierarchy of regions. These regions are either loop regions, which include all the edges into the header, or body regions that have no edges into the header.

- *Region-Based Data-Flow Analysis*: An alternative to the iterative approach to data-flow analysis is to work up and down the region hierarchy, computing transfer functions from the header of each region to each node in that region.

- *Region-Based Induction Variable Detection*: An important application of region-based analysis is in a data-flow framework that tries to compute formulas for each variable in a loop region whose value is an affine (linear) function of the number of times around the loop.

9.10 References for Chapter 9

Two early compilers that did extensive code optimization are Alpha [7] and Fortran H [16]. The fundamental treatise on techniques for loop optimization (e.g., code motion) is [1], although earlier versions of some of these ideas appear in [8]. An informally distributed book [4] was influential in disseminating code-optimization ideas.

The first description of the iterative algorithm for data-flow analysis is from the unpublished technical report of Vyssotsky and Wegner [20]. The scientific study of data-flow analysis is said to begin with a pair of papers by Allen [2] and Cocke [3].

The lattice-theoretic abstraction described here is based on the work of Kildall [13]. These frameworks assumed distributivity, which many frameworks do not satisfy. After a number of such frameworks came to light, the monotonicity condition was embedded in the model by [5] and [11].

Partial-redundancy elimination was pioneered by [17]. The lazy-code-motion algorithm described in this chapter is based on [14].

Dominators were first used in the compiler described in [13]. However, the idea dates back to [18].

The notion of reducible flow graphs comes from [2]. The structure of these flow graphs, as presented here, is from [9] and [10]. [12] and [15] first connected reducibility of flow graphs to the common nested control-flow structures, which explains why this class of flow graphs is so common.

The definition of reducibility by T_1-T_2 reduction, as used in region-based analysis, is from [19]. The region-based approach was first used in a compiler described in [21].

The static single-assignment (SSA) form of intermediate representation introduced in Section 6.2.4 incorporates both data flow and control flow into its representation. SSA facilitates the implementation of many optimizing transformations from a common framework [6].

1. Allen, F. E., "Program optimization," *Annual Review in Automatic Programming* **5** (1969), pp. 239–307.

2. Allen, F. E., "Control flow analysis," *ACM Sigplan Notices* **5**:7 (1970), pp. 1–19.

3. Cocke, J., "Global common subexpression elimination," *ACM SIGPLAN Notices* **5**:7 (1970), pp. 20–24.

4. Cocke, J. and J. T. Schwartz, *Programming Languages and Their Compilers: Preliminary Notes*, Courant Institute of Mathematical Sciences, New York Univ., New York, 1970.

5. Cousot, P. and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," *Fourth ACM Symposium on Principles of Programming Languages* (1977), pp. 238–252.

6. Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems* **13**:4 (1991), pp. 451–490.

7. Ershov, A. P., "Alpha — an automatic programming system of high efficiency," *J. ACM* **13**:1 (1966), pp. 17–24.

8. Gear, C. W., "High speed compilation of efficient object code," *Comm. ACM* **8**:8 (1965), pp. 483–488.

9. Hecht, M. S. and J. D. Ullman, "Flow graph reducibility," *SIAM J. Computing* **1** (1972), pp. 188–202.

10. Hecht, M. S. and J. D. Ullman, "Characterizations of reducible flow graphs," *J. ACM* **21** (1974), pp. 367–375.

11. Kam, J. B. and J. D. Ullman, "Monotone data flow analysis frameworks," *Acta Informatica* **7**:3 (1977), pp. 305–318.

12. Kasami, T., W. W. Peterson, and N. Tokura, "On the capabilities of while, repeat, and exit statements," *Comm. ACM* **16**:8 (1973), pp. 503–512.

13. Kildall, G., "A unified approach to global program optimization," *ACM Symposium on Principles of Programming Languages* (1973), pp. 194–206.

14. Knoop, J., "Lazy code motion," *Proc. ACM SIGPLAN 1992 conference on Programming Language Design and Implementation*, pp. 224–234.

15. Kosaraju, S. R., "Analysis of structured programs," *J. Computer and System Sciences* **9**:3 (1974), pp. 232–255.

16. Lowry, E. S. and C. W. Medlock, "Object code optimization," *Comm. ACM* **12**:1 (1969), pp. 13–22.

17. Morel, E. and C. Renvoise, "Global optimization by suppression of partial redundancies," *Comm. ACM* **22** (1979), pp. 96–103.

18. Prosser, R. T., "Application of boolean matrices to the analysis of flow diagrams," *AFIPS Eastern Joint Computer Conference* (1959), Spartan Books, Baltimore MD, pp. 133–138.

19. Ullman, J. D., "Fast algorithms for the elimination of common subexpressions," *Acta Informatica* **2** (1973), pp. 191–213.

20. Vyssotsky, V. and P. Wegner, "A graph theoretical Fortran source language analyzer," unpublished technical report, Bell Laboratories, Murray Hill NJ, 1963.

21. Wulf, W. A., R. K. Johnson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*, Elsevier, New York, 1975.

This page intentionally left blank