

UNIT-VI

Machine Independent Optimization. The principle sources of Optimization, peep hole Optimization, Introduction to Data flow Analysis.

UNIT 6

MACHINE INDEPENDENT OPTIMIZATION

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."

Optimizations are classified into two categories.

1. Machine independent optimizations:

Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine

2. Machine dependant optimizations:

Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

The Principal Sources of Optimization

A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels.

Function-Preserving Transformations: There are a number of ways in which a compiler can improve a program without changing the function it computes.

: Common sub expression elimination

Copy propagation,

Dead-code elimination

Constant folding

Common Sub expressions elimination:

An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

- For example

t1: = 4*i

t2: = a [t1]

t3: = 4*j

t4: = 4*i

t5: = n

t6: = b [t4] +t5

The above code can be optimized using the common sub-expression elimination as

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5
```

The common sub expression $t4: = 4*i$ is eliminated as its computation is already in $t1$ and the value of i is not been changed from definition to use.

Copy Propagation:

Assignments of the form $f := g$ called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f , whenever possible after the copy statement $f := g$. Copy propagation means use of one variable instead of another.

- For example:

```
x=Pi;
A=x*r*r;
```

The optimization using copy propagation can be done as follows: $A=Pi*r*r$;
Here the variable x is eliminated

Dead-Code Eliminations:

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.

Example:

```
i=0;
if(i==1)
{
a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

Constant folding:

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,

```
a=3.14157/2 can be replaced by
a=1.570
```

Loop Optimizations:

In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

1. Code motion, which moves code outside a loop;
2. Induction-variable elimination, which we apply to replace variables from inner loop.
- 3.Reduction in strength, which replaces expensive operation by a cheaper one, such as a multiplication by an addition.

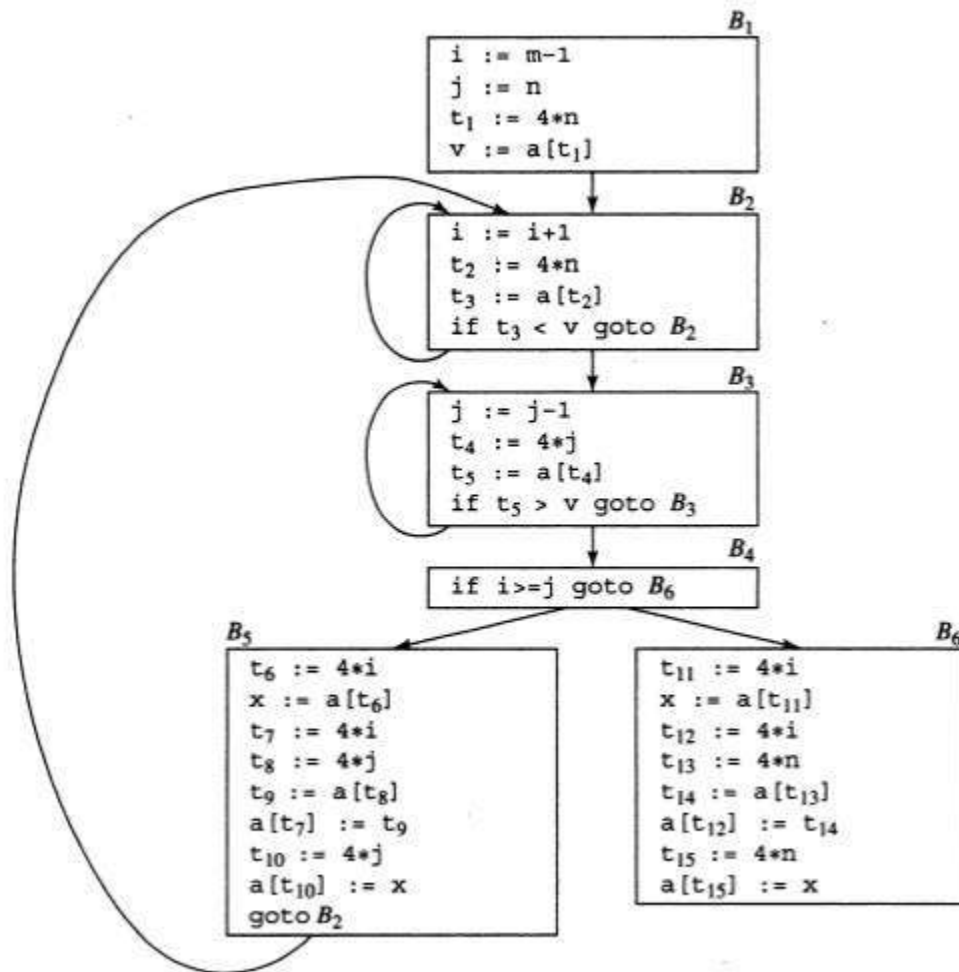


Fig. 5.2 Flow graph

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 places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2)
```

Code motion will result in the equivalent of

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

Induction Variables :

Loops are usually processed inside out. For example consider the loop around B3. Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because $4*j$ is assigned to t4. Such identifiers are called induction variables.

When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig.5.3 we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.

However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

Example:

As the relationship $t4:=4*j$ surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, 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 B3 for the first time. Since we must maintain the relationship $t4=4*j$ on entry to the block B3, we place an initialization of t4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in Fig.5.3.

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction

Reduction In Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x^2 is invariably cheaper to implement as $x*x$ than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

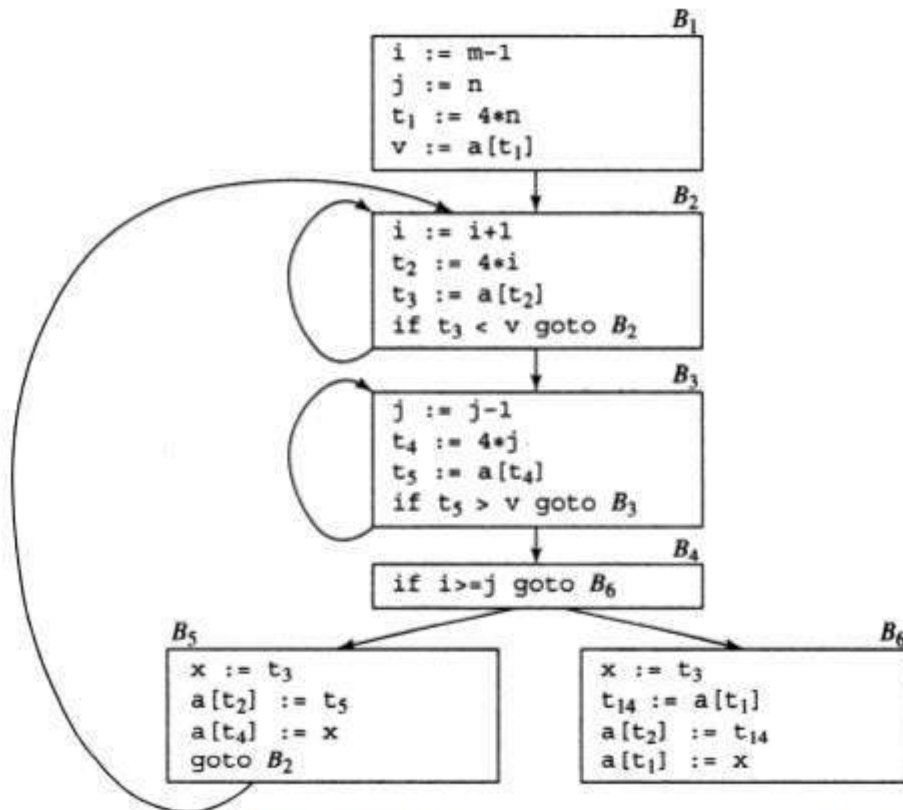


Fig. 5.3 B5 and B6 after common subexpression elimination

Fig. 5.3 B5 and B6 after common subexpression elimination

PEEPHOLE OPTIMIZATION

A statement-by-statement code-generations strategy often produces target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying “optimizing” transformations to the target program.

A simple but effective technique for improving the target code is peephole optimization, A method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

The peephole is a small, moving window on the target program.

Characteristics of peephole optimizations:

- Redundant-instructions elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms
- Unreachable code

Redundant-instructions elimination

see the instructions sequence

- (1) MOV R0,a
- (2) MOV a,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1. In C, the source code might look like:

```
#define debug 0
....

If ( debug ) {
    Print debugging information
}
```

In the intermediate representations the if-statement may be translated as:

If debug =1 goto L1 goto L2

L1: print debugging information L2: (a)

One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of debug; (a) can be replaced by:

```
    If debug ≠1 goto L2
    Print debugging information
L2: ..... (b)
```

```
    If debug ≠0 goto L2
    Print debugging information
L2: ..... (c)
```

As the argument of the statement of (c) evaluates to a constant true it can be replaced

By goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L1

....

L1: gotoL2 (d)

by the sequence

goto L2

....

L1: goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

if a < b goto L1

....

L1: goto L2 (e)

can be replaced by

If a < b goto L2

....

L1: goto L2

Ø Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1

L1: if a < b goto L2 (f) L3:

may be replaced by

CSE Dept.,Sir CRR COE.

**If a < b goto L2
goto L3**

.....

L3:

While the number of instructions in (e) and (f) is the same, we sometimes skip the unconditional jump in (f), but never in (e). Thus (f) is superior to (e) in execution time

Algebraic Simplification:

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

$x := x + 0$ or
 $x := x * 1$

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Reduction in Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, x^2 is invariably cheaper to implement as $x*x$ than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

$X^2 \rightarrow X*X$

Use of Machine Idioms:

The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like $i := i + 1$.

$i := i + 1 \rightarrow i++$

Introduction to Data flow Analysis.

1 The Data-Flow Abstraction

2 The Data-Flow Analysis Schema

3 Data-Flow Schemas on Basic Blocks

4 Reaching Definitions

5 Live-Variable Analysis

6 Available Expressions

"Data-flow analysis" refers to a body of techniques that derive information about the flow of data along program execution paths.

1. The Data-Flow Abstraction

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. 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.

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_i to point p_n to be a sequence of points p_i, p_2, \dots, p_n such that for each $i = 1, 2, \dots, n-1$, 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.

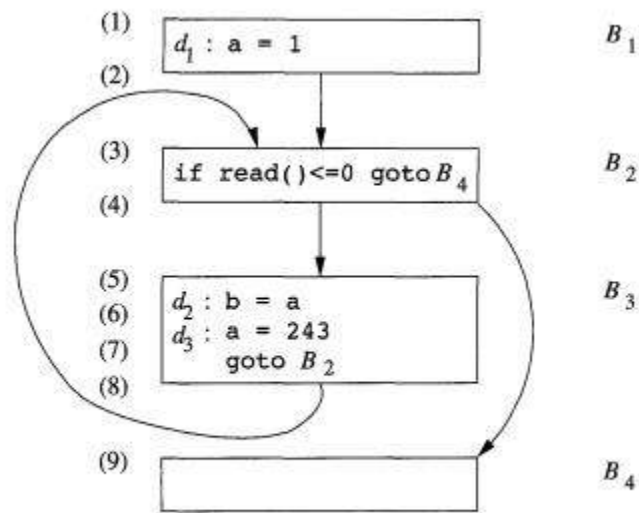


Figure 9.12: Example program illustrating the data-flow abstraction

. 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 $\{^1, ^3\}$. 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.

2. The Data-Flow Analysis Schema

, 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.

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 $f_s(a)$, takes the data-flow value before the statement and produces a new data-flow value after the statement. That is,

$$OUT[s] = f_s(IN[s]).$$

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

$$IN[s] = f_s(OUT[s]).$$

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, \dots, 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,

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

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.

3. Data-Flow Schemas on Basic Blocks

While a data-flow schema 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 $m[B]$ and $OUT[B]$, respectively. The constraints involving $m[B]$ and $OUT[B]$ can be derived from those involving $w[s]$ and $OUT[s]$ for the various statements s in B as follows.

Suppose block B consists of statements s_1, \dots, s_n , in that order. If s_1 is the first statement of basic block B , then $m[B] = IN[s_1]$. Similarly, if s_n is the last statement of basic block B , then $OUT[B] = OUT[s_n]$. 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 $f_B = f_{s_n} \circ \dots \circ f_{s_2} \circ f_{s_1}$. The relationship between the beginning and end of the block is

$$OUT[B] = f_B(IN[B]).$$

The constraints due to control flow between basic blocks can easily be rewritten by substituting $IN[B]$ and $OUT[B]$ for $IN[s_i]$ 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

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

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,

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

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.

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

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.

Transfer Equations for Reaching Definitions

Start by examining the details of a single statement. Consider a definition

$d: u = v + w$

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 re-maining incoming definitions unaffected. The transfer function of definition d thus can be expressed as

$$f_d(x) = \text{gen}_d \cup (x - \text{kill}_d) \quad (9.1)$$

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

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 $f_1(x) = \text{gen}_1 \cup (x - \text{kill}_1)$ and $f_2(x) = \text{gen}_2 \cup (x - \text{kill}_2)$. Then

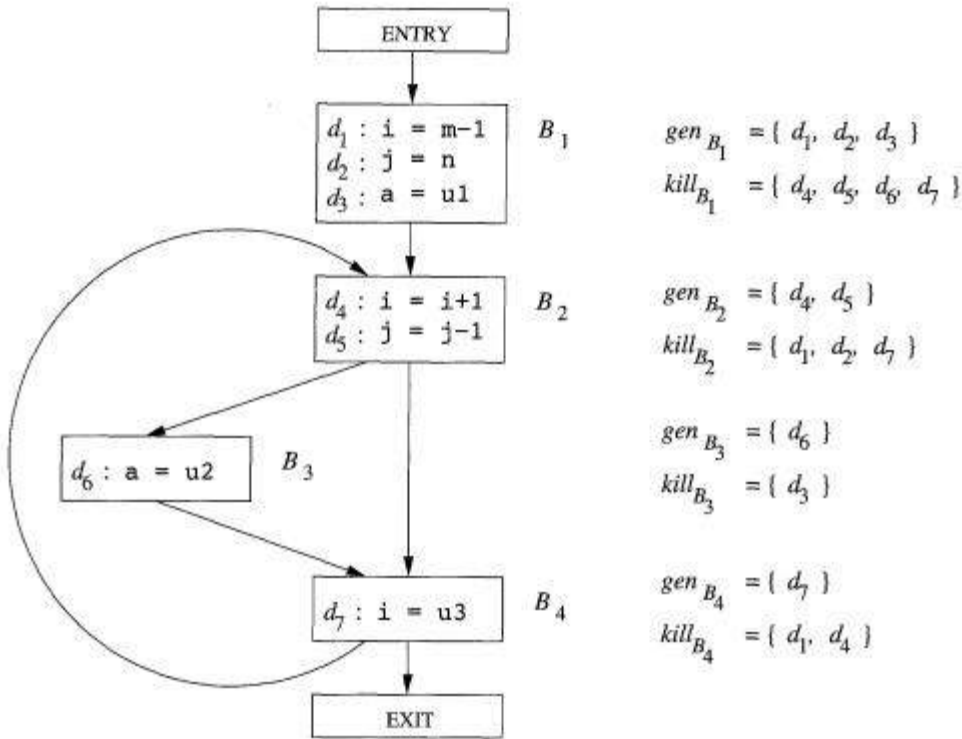


Figure 9.13: Flow graph for illustrating reaching definitions

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

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 $i = 1, 2, \dots, n$. Then the transfer function for block B may be written as:

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

where

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

and

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

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

CSE Dept.,Sir CRR COE.

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 {d2} since d1 is not downwards exposed. The kill set contains both d1 and d2, since d1 kills d2 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 d2.

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, $OUT[P] \subseteq IN[B]$ 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, $OUT[P]$ needs to be no larger than the union of the reaching definitions of all the predecessor blocks. That is, it is safe to assume

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

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 0 as an answer. That is, $OUT[ENTRY] = \emptyset$.

The reaching definitions problem is defined by the following equations:

$$OUT[ENTRY] = \emptyset$$

and for all basic blocks B other than ENTRY,

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

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

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 *kills* 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" $\text{OUT}[B] = \emptyset$ 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. •

Algorithm 9.11 propagates definitions as far as they will go with-out 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

```

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] =  $\bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]$ ;
6)          OUT[B] = genB ∪ (IN[B] - killB);
      }
```

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 (5), 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, \dots, d_j 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 $OUT[S]_0$, 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 $IN[I]_1$ and $OUT[S]_1$ for the first pass and $IN[B]_2$ and $OUT[S]_2$ 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 $OUT[ENTRY] = 0$, $IN[B_1] - \text{Pow}(1)$ is the empty set, and $OUT[P_1]_1$ is $\text{gen}[B_1]$. This value differs from the previous value $OUT[S]_0$, 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

$$\begin{aligned}
 IN[B_2]_1 &= OUT[B_1]_1 \cup OUT[B_4]_0 \\
 &= 111\ 0000 + 000\ 0000 = 111\ 0000 \\
 OUT[B_2]_1 &= \text{gen}[B_2] \cup (IN[B_2]_1 - \text{kill}[B_2]) \\
 &= 000\ 1100 + (111\ 0000 - 110\ 0001) = 001\ 1100
 \end{aligned}$$

CSE Dept., Sir CRR COE.

This computation is summarized in Fig. 9.15. For instance, at the end of the first pass, $OUT[5, 2, 1] = 001\ 1100$, 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.

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. $defB$ as the set of variables defined (i.e., definitely assigned values) in B prior to any use of that variable in B , and $useB$ 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 $uses_2 = \{i, j\}$. Also, B_2 clearly defines i and j . Assuming there are no aliases, $defB_2 =$ as well.

As a consequence of the definitions, any variable in $useB$ must be considered live on entrance to block B , while definitions of variables in $defB$ definitely are dead at the beginning of B . In effect, membership in $defB$ "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:

$$IN[EXIT] = \emptyset$$

and for all basic blocks B other than EXIT,

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

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-defin-itions 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: $m[B]$ and $OUT[B]$, the set of variables live on entry and exit of each block B of the flow graph.

```

IN[EXIT] =  $\emptyset$ ;
for (each basic block  $B$  other than EXIT) IN[ $B$ ] =  $\emptyset$ ;
while (changes to any IN occur)
    for (each basic block  $B$  other than EXIT) {
        OUT[ $B$ ] =  $\bigcup_{S \text{ a successor of } B} \text{IN}[S]$ ;
        IN[ $B$ ] = use $_B \cup (\text{OUT}[B] - \text{def}_B)$ ;
    }

```

Figure 9.16: Iterative algorithm to compute live variables

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 assign) to x or y . Note that, as usual in this chapter, we use the operator $+$ as a generic operator, not necessarily standing for addition.

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 .

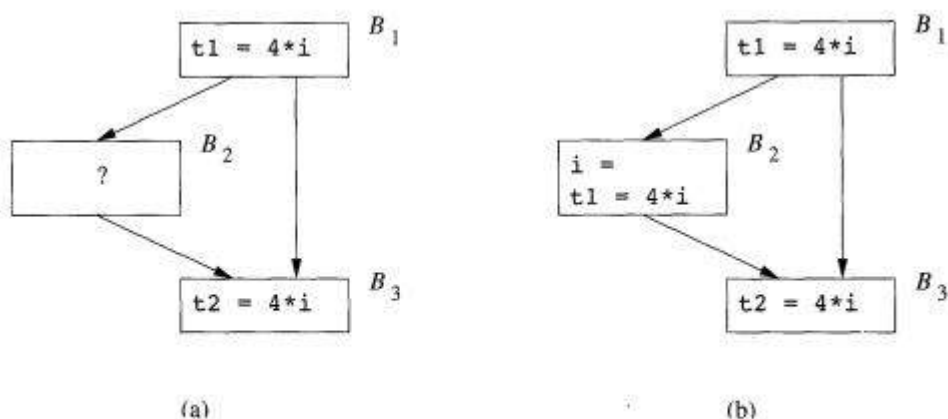


Figure 9.17: Potential common subexpressions across blocks

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.

Add to S the expression $y + z$.

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 $egenB$ to be the expressions generated by B and $ekills$ to be the set of expressions in U killed in B . Note that IN , OUT , $egen$, and $ekill$ can all be represented by bit vectors. The following equations relate the unknowns

CSE Dept.,Sir CRR COE.

IN and OUT to each other and the known quantities e_gen and e_kill :

$$OUT[ENTRY] = \emptyset$$

and for all basic blocks B other than ENTRY,

$$OUT[B] = e_gen_B \cup (IN[B] - e_kill_B)$$

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

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.

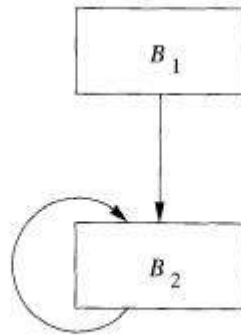


Figure 9.19: Initializing the OUT sets to \emptyset is too restrictive.

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 $OUT[B_2]$ on $IN[B_2]$ - Let G and K abbreviate $e.genB_2$ and $e.killB_2$, respectively. The data-flow equations for block B_2 are

$$IN[B_2] = OUT[B_1] \cap OUT[B_2]$$

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

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

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

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

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

Starting with $O^0 = \emptyset$, we get $I^1 = OUT[B_1] \cap O^0 = \emptyset$. However, if we start with $O^0 = U$, then we get $I^1 = OUT[B_1] \cap O^0 = OUT[B_1]$, 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 . \square

Algorithm 9.17 : Available expressions.

INPUT: A flow graph with e-kills and e.gens 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.

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

Figure 9.20: Iterative algorithm to compute available expressions

Figure 9.20: Iterative algorithm to compute available expressions