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.

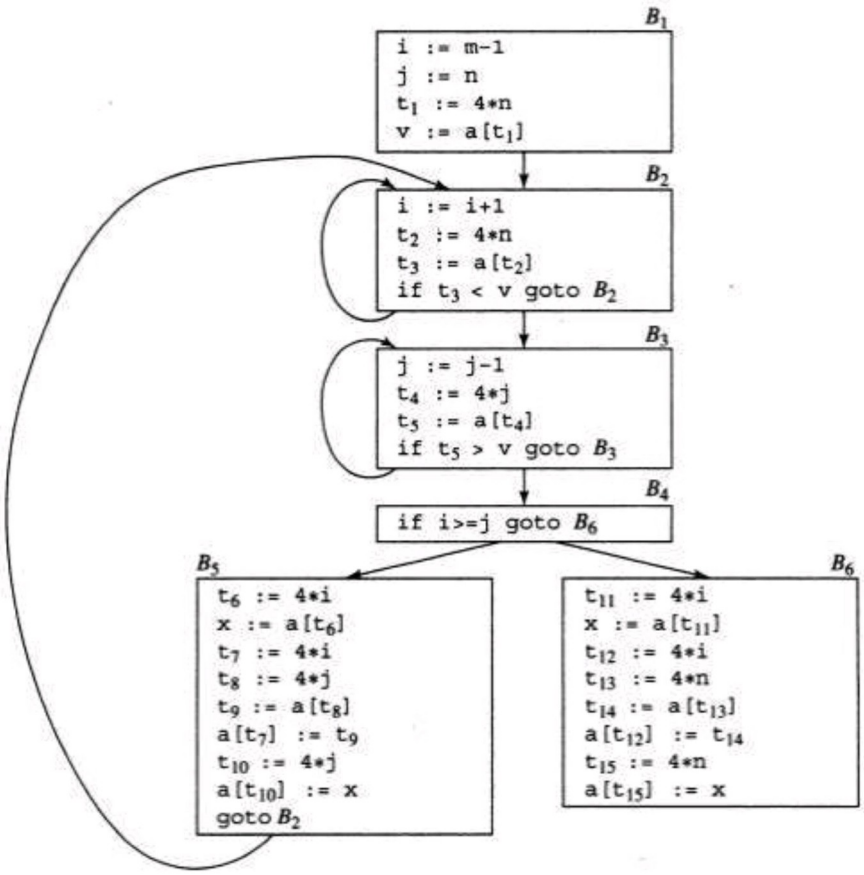


Fig. 5.2 Flow graph

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 initializations 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.

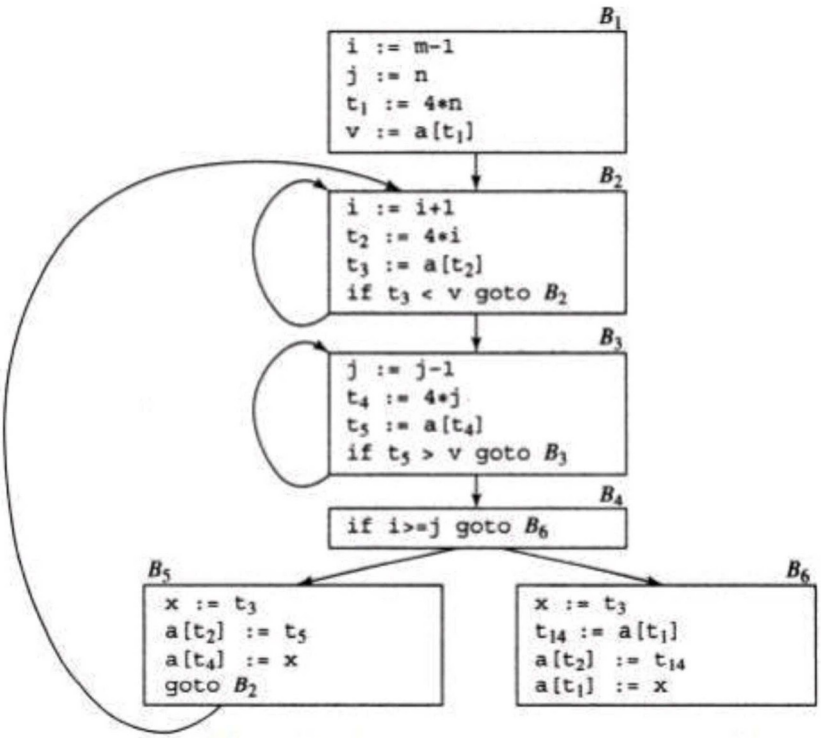


Fig. 5.3 B5 and B6 after common subexpression elimination

Fig. 5.3 B5 and B6 after common subexpression elimination

i:=i-1 → i- -

Introduction to Date flow Analysis.

"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 *B1* to block $B2_2$, then the program point after the last statement of *B1* 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 pi to point pn to be a sequence of points pi,p2,... ,pn such that for each i = 1,2, ... ,n - 1, either

1. Pi is the point immediately preceding a statement and $pi_{+}i$ is the point immediately following that same statement, or

2.pi is the end of some block and p_{i+1} is the beginning of a successor block.

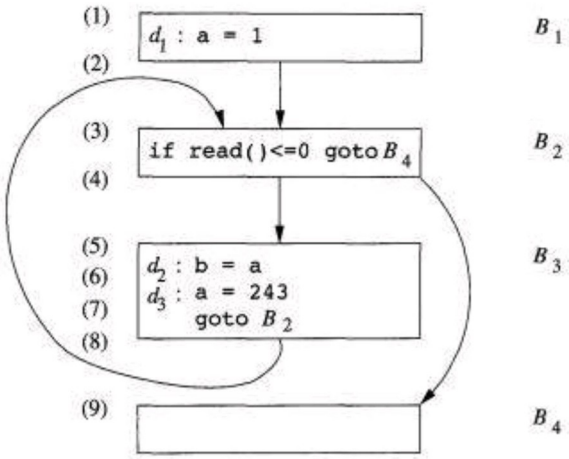


Figure 9.12: Example program illustrating the data-flow abstraction

. 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(a), 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(a) for statement 8 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 s1, s_2, • • •,s_n in that order, then the control-flow value out of ***Si*** is the same as the control-flow value into ***Si+i.*** 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.

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 0 U T [S] , respectively. The constraints involving *m[B]* and 0UT[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 , . . . , sn, in that order. If si is the first statement of basic block B, then m[B] = I N [S I] , Similarly, if sn is the last statement of basic block B, then OUT[S] = OUT[s,,] . The transfer function of a basic block B, which we denote fB, can be derived by composing the transfer functions of the statements in the block. That is, let fa. be the transfer function of statement st. Then of statement si. Then fB = f,sn, o . . . o f,s2, o fsl. . 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[SI**] and **OUT[sn],** 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 analy-sis, the equations are similar, but with the roles of the IN's and OUT's reversed. That is,

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

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

$$f_d(x) = gen_d \cup (x - kill_d) \qquad (9.1)$$

where gend = {d}, the set of definitions generated by the statement, and killd 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 fi(x) = gen1 U (x - kill1) and f2(x) = gen2 U (x — kill2). Then

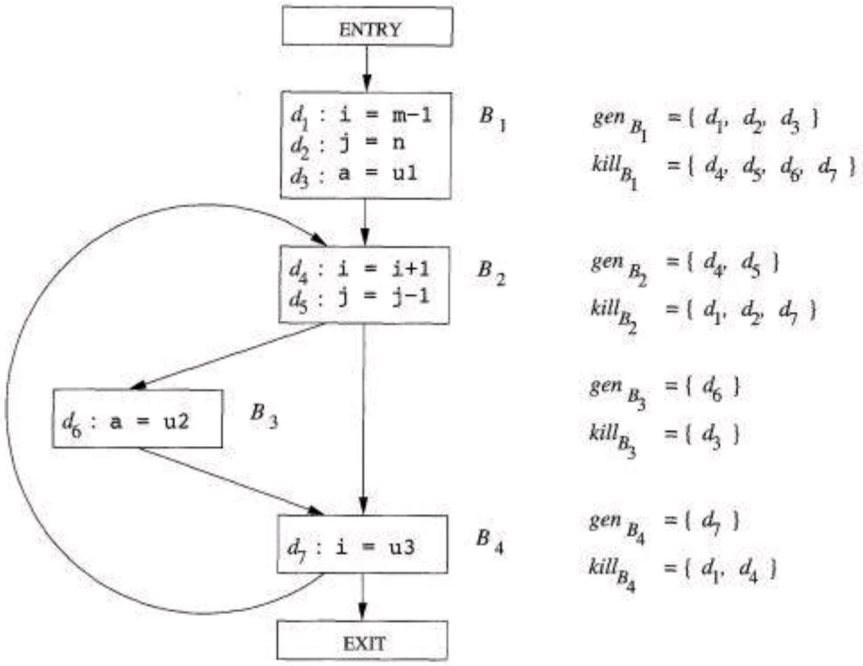


Figure 9.13: Flow graph for illustrating reaching definitions

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

This rule extends to a block consisting of any number of statements. Suppose block *B* has *n* statements, with transfer functions *fi(x) = geni* U *(x — kilh)* for i = 1,2, ... , 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 . 1 0 : 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, **O U T** [P] **C** *m[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, *w[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 0 as an answer. That is, O U T [E N T R Y] = 0.

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.

A l g o r i t h m 9 . 1 1 : Reaching definitions.

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

OUTPUT: **I N [B]** and O U T **[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[JB] = 0 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. Algo-rithm 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] = ⋃P a predecessor of B 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 . 1 2 : We shall represent the seven definitions d1, d2, • • • ,d>j in the flow graph of Fig. 9.13 by bit vectors, where bit i from the left represents definition d{. 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 OUTfS]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 OUTfS]1 for the first pass and m[Bf 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 = B1, since OUT [ENTRY] = 0, [IN B1]-Pow(1) is the empty set, and OUT[P1]1 is genBl. This value differs from the previous value OUT[Si]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 [5 2] 1 = 001 1100, reflecting the fact that d4 and d5 are generated in B2, while d3 reaches the beginning of B2 and is not killed in B2.

Notice that after the second round, OUT [B2] has changed to reflect the fact that d& also reaches the beginning of B2 and is not killed by B2. We did not learn that fact on the first pass, because the path from d6 to the end of B2, which is B3 -» B4 -> B2, is not traversed in that order by a single pass. That is, by the time we learn that d$ reaches the end of B4, we have already computed IN[B2] 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 [5] and OUTpB], 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 . 1 3 : For instance, block B2 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 uses2 = {i,j}- Also, B2 clearly defines i and j. Assuming there are no aliases, defB2 = 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:

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

and for all basic blocks B other than EXIT,

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

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 O U T [E N T R Y] , we initialize I N [EXIT] . Sets I N and O U T 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 so-lution with the smallest sets of live variables. The algorithm used is essentially a backwards version of Algorithm 9.11.

Algorithm 9 . 1 4 : Live-variable analysis.

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

OUTPUT: m[B] and O U T [£] , the set of variables live on entry and exit of each block B of the flow graph.

```
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

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 N o te 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 Bs will be a common subexpression if 4 * i is available at the entry point of block B3. It will be available if i is not assigned a new value in block B2, or if, as in Fig. 9.17(b), 4 * i is recomputed after i is assigned in B2.

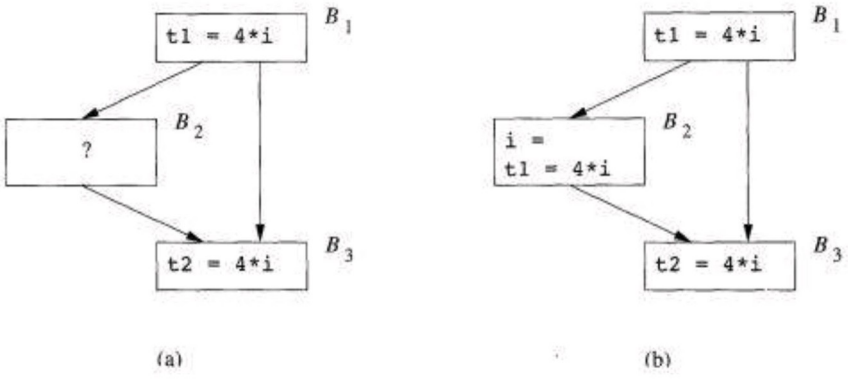


Figure 9.17: Potential common subexpressions across blocks

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.

E x a m p l e 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 reach-ing 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.genB*** to be the expressions generated by *B* and *eJnills* to be the set of expressions in *U* killed in *B*. Note that **I N , O U T ,** e_#en, and *eJkill* 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,

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

T he above equations look almost identical to the equations for reaching definitions. Like reaching definitions, the boundary condition is OUT [ENTRY] = 0, because at the exit of the E N T R Y 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 D rather than U 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 expres-sion 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.

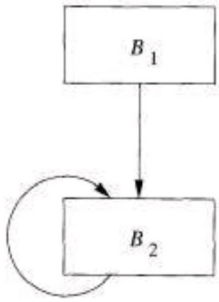


Figure 9.19: Initializing the OUT sets to $\emptyset$ is too restrictive.

Example 9 . 1 6 : We shall concentrate on a single block, B2 in Fig. 9.19, to illustrate the effect of the initial approximation of OUT[B2] on IN [B 2] - Let G and K abbreviate e.genB2 and e-killB2, respectively. The data-flow equations for block B2 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 $\text{IN}[B_2]$ and $\text{OUT}[B_2]$, 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 $\text{OUT}[B_1]$ that are not killed by B_2 are available at the end of B_2. □

Algorithm 9 . 1 7 : Available expressions.

INPUT: A flow graph with e-kills and e.gens computed for each block B. The initial block is B1.

OUTPUT: IN [5] and O U T [5] , the set of expressions available at the entry and exit of each block B of the flow graph.

```
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

Figure 9.20: Iterative algorithm to compute available expressions

Introduction to Data Flow Analysis

It is the analysis of flow of data in control flow graph, i.e., the analysis that determines the information regarding the definition and use of data in program. With the help of this analysis, optimization can be done. In general, its process in which values are computed using data flow analysis. The data flow property represents information that can be used for optimization.

Data flow analysis is a technique used in compiler design to analyze how data flows through a program. It involves tracking the values of variables and expressions as they are computed and used throughout the program, with the goal of identifying opportunities for optimization and identifying potential errors.

The basic idea behind data flow analysis is to model the program as a graph, where the nodes represent program statements and the edges represent data flow dependencies between the statements. The data flow information is then propagated through the graph, using a set of rules and equations to compute the values of variables and expressions at each point in the program.

Some of the common types of data flow analysis performed by compilers include:

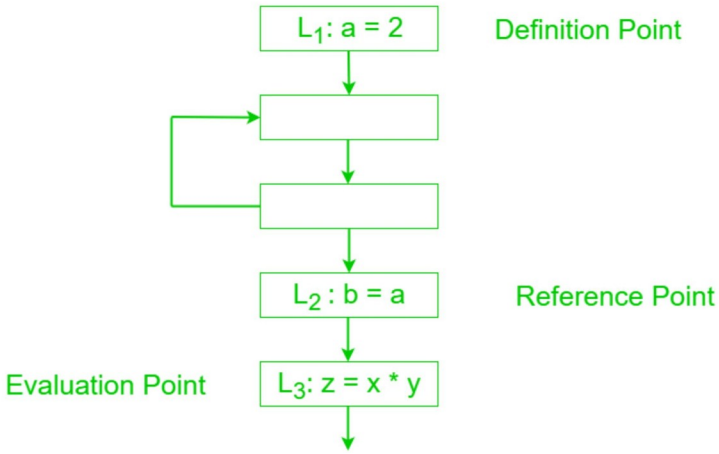
1. **Reaching Definitions Analysis:** This analysis tracks the definition of a variable or expression and determines the points in the program where the definition "reaches" a particular use of the variable or expression. This information can be used to identify variables that can be safely optimized or eliminated.
2. **Live Variable Analysis:** This analysis determines the points in the program where a variable or expression is "live", meaning that its value is still needed for some future computation. This information can be used to identify variables that can be safely removed or optimized.
3. **Available Expressions Analysis:** This analysis determines the points in the program where a particular expression is "available", meaning that its value has already been computed and can be reused. This information can be used to identify opportunities for common subexpression elimination and other optimization techniques.
4. **Constant Propagation Analysis:** This analysis tracks the values of constants and determines the points in the program where a particular constant value is used. This information can be used to identify opportunities for constant folding and other optimization techniques.

Data flow analysis can have a few advantages in compiler design, including:

1. **Improved code quality:** By identifying opportunities for optimization and eliminating potential errors, data flow analysis can help improve the quality and efficiency of the compiled code.
2. **Better error detection:** By tracking the flow of data through the program, data flow analysis can help identify potential errors and bugs that might otherwise go unnoticed.
3. **Increased understanding of program behavior:** By modeling the program as a graph and tracking the flow of data, data flow analysis can help programmers better understand how the program works and how it can be improved.

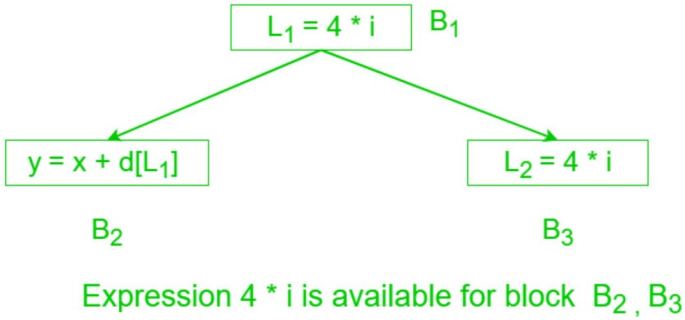
Basic Terminologies –

- **Definition Point:** a point in a program containing some definition.
- **Reference Point:** a point in a program containing a reference to a data item.
- **Evaluation Point:** a point in a program containing evaluation of expression.



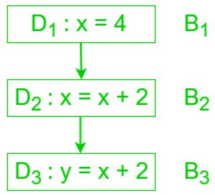
Data Flow Properties –

- **Available Expression –** A expression is said to be available at a program point x if along paths its reaching to x. A Expression is available at its evaluation point. An expression a+b is said to be available if none of the operands gets modified before their use.
  Example –



- **Advantage –**
  It is used to eliminate common sub expressions.
- **Reaching Definition –** A definition D is reaches a point x if there is path from D to x in which D is not killed, i.e., not redefined.

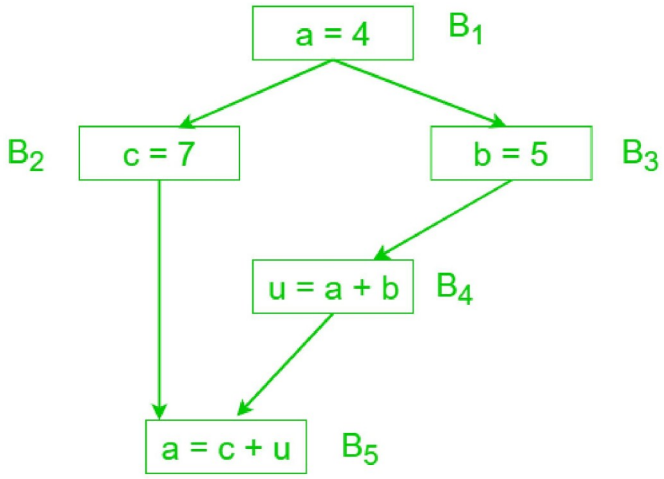
Example –



D_1 is reaching definition for B_2 but not for B_3 since it is killed by D_2

It is used in constant and variable propagation.

- **Live variable –** A variable is said to be live at some point p if from p to end the variable is used before it is redefined else it becomes dead.
  Example –



a is live at block B_1 , B_3 , B_4 but killed at B_5

- **Advantage –**
  1. It is useful for register allocation.
  2. It is used in dead code elimination.
- **Busy Expression –** An expression is busy along a path if its evaluation exists along that path and none of its operand definition exists before its evaluation along the path.
  Advantage –
  It is used for performing code movement optimization.

Features:

Identifying dependencies: Data flow analysis can identify dependencies between different parts of a program, such as variables that are read or modified by multiple statements.
Detecting dead code: By tracking how variables are used, data flow analysis can detect code that is never executed, such as statements that assign values to variables that are never used.
Optimizing code: Data flow analysis can be used to optimize code by identifying opportunities for common subexpression elimination, constant folding, and other optimization techniques.
Detecting errors: Data flow analysis can detect errors in a program, such as uninitialized variables, by tracking how variables are used throughout the program.
Handling complex control flow: Data flow analysis can handle complex control flow structures, such as loops and conditionals, by tracking how data is used within those structures.
Intraprocedural analysis: Data flow analysis can be performed across multiple functions in a program, allowing it to analyse how data flows between different parts of the program.
Scalability: Data flow analysis can be scaled to large programs, allowing it to analyse programs with many thousands or even millions of lines of code.

FOUNDATIONS OF DATA FLOW ANALYSIS:

The foundations of data flow analysis in compiler design follow basics concepts:

1.Lattices

2.Reaching Definition

3.Control Flow Graph

4.Iterative Algorithms

Lattices- These are partially ordered sets. The sets represent the state of the program during data flow analysis. They perform two operations - join operation and meet operation. The set elements' lower bound is calculated in the join operation. In meet operation, there is a merge of two lattices.

Reaching Definition- It determines which definition may reach a specific code point. It tracks the information flow in a code. It tracks the information flow in a code. A definition is said to reach a point such that there has been no variable redefinition. Reaching definitions track the variables and the values they hold at different points.

Control Flow Graph- It is a graphical representation of the control flow in compiler design. The control flow refers to the computations during the execution of a program. Control flow graphs are process-oriented directed graphs. The control flow graph has two blocks: the entry and exit. It helps to analyse the flow of computations and identify potential performance loopholes.

Iterative Algorithms- Iterative algorithms are used to solve data flow analysis problems. The iterative algorithms used are for reaching problems and for available expressions. Iterative algorithms for reaching problems assume that all definitions reach all points. It then iteratively solves and updates the reachable points for different definitions. For available expressions, it assumes that all expressions are available at all points. It then iteratively solves and updates the point of use of the expressions.

Lattices, Reaching Definition, and Control Flow are all important. They provide foundations of data flow analysis in compiler design. All three of them help in code optimization and data flow tracking.

CONSTANT PROPAGATION:

Constant Propagation is one of the local code optimization technique in Compiler Design. It can be defined as the process of replacing the constant value of variables in the expression. In simpler words, we can say that if some value is assigned a known constant, than we can simply replace the that value by constant. Constants assigned to a variable can be propagated through the flow graph and can be replaced when the variable is used.

Constant propagation is executed using reaching definition analysis results in compilers, which means that if reaching definition of all variables have same assignment which assigns a same constant to the variable, then the variable has a constant value and can be substituted with the constant.

Suppose we are using pi variable and assign it value of 22/7

pi = 22/7 = 3.14

In the above code the compiler has to first perform division operation, which is an expensive operation and then assign the computed result 3.14 to the variable pi. Now if anytime we have to use this constant value of pi, then the compiler again has to look – up for the value and again perform division operation and then assign it to pi and then use it. This is not a good idea when we can directly assign the value 3.14 to pi variable, thus reducing the time needed for code to run.

Also, Constant propagation reduces the number of cases where values are directly copied from one location or variable to another, in order to simply allocate their value to another variable. For an example :

Consider the following pseudocode :

a = 30

b = 20 - a /2

c = b * (30 / a + 2) - a

We can see that in the first expression value of a have assigned a constant value that is 30. Now, when the compiler comes to execute the second expression it encounters a, so it goes up to the first expression to look for the value of a and then assign the value of 30 to a again, and then it executes the second expression. Now it comes to the third expression and encounters b and a again, and then it needs to evaluate the first and second expression again in order to compute the value of c. Thus, a needs to be propagated 3 times This procedure is very time consuming.

We can instead , rewrite the same code as :

a = 30

b = 20 - 30/2

c = b * (30 / 30 + 2) - 30

This updated code is faster as compared to the previous code as the compiler does not need to again and again go back to the previous expressions looking up and copying the value of a

the current expressions. This saves a lot of time and thus, reducing time complexity and perform operations more efficiently.

Note that this constant propagation technique behavior depends on compiler like few compilers perform constant propagation operations within the basic blocks; while a few compilers perform constant propagation operations in more complex control flow.

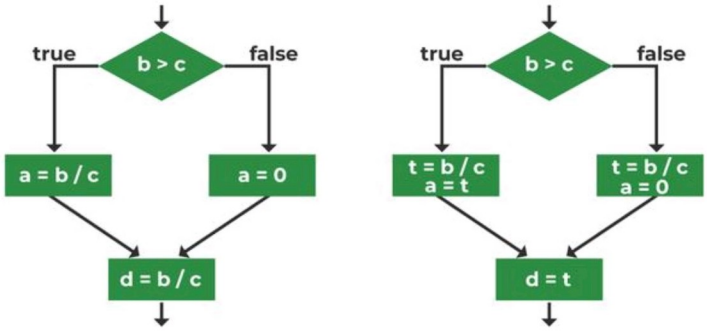
Partial-Redundancy Elimination

Code optimization is one of the phases of compilation. This phase is often known as Machine-Independent Code Optimization. In this phase, several techniques and procedures are implied to reduce the code's runtime. The code that is being optimized here is also of different types, such as dead code, redundant code, partially redundant code, etc.

In this article, we will discuss redundancy in code. A redundant piece of code contains expressions or statements that repeat themselves or produce the same results throughout the execution flow of the code. Similarly, a partially redundant code has redundancy in one or more execution flows of the code but not necessarily in all the paths of execution.

These redundancies may exist in various forms, such as in common sub-expressions and loop-invariant expressions, etc.

For example, consider figure 1a, here the expression "*b / c*" is evaluated twice along one flow path, i.e., when the condition $b > c$ is true even though there is no change in the values of variables *b* and *c*. Hence, this particular piece of code is partially redundant. This redundancy can be eliminated if we compute the expression "*b / c"* once and store it in a variable, say *t*. Then we can use the variable *t* in our code whenever we need the value of the expression "*b / c*".



Example of Partial redundant Code

Can we eliminate all redundancies?

In the earlier example, we eliminated the redundancy in the code by duplicating the common expression and moving it to blocks of other paths of the code. However, it is not always possible to implement this technique in codes with tens or hundreds of execution paths. For example, in figure 2 below, the expression *b / c* is redundant in the path "1 ⇒ 2 ⇒ 3". But we cannot duplicate and move the expression to block 3, as it was done in the previous example,

l create an unnecessary computation in the paths other than “1 ⇒ 2 ⇒ 3” or “1 ⇒ 3 ⇒ 4”.

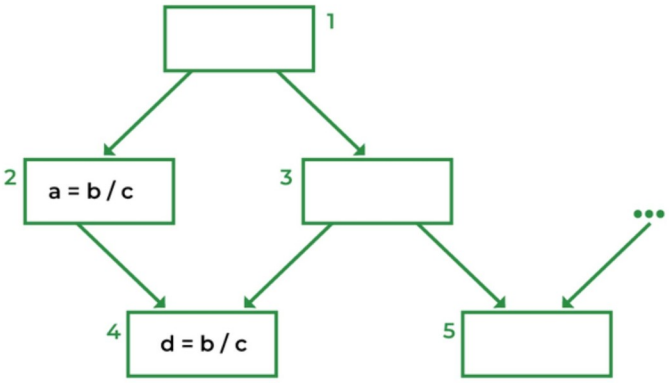


Figure 1

Although this issue can be solved if it is allowed to create new blocks in the execution path of the code as in figure 3. However, this technique is also inadequate to eliminate all the redundancies in the code because we might need to duplicate the expressions to separate the paths where redundancy occurs.

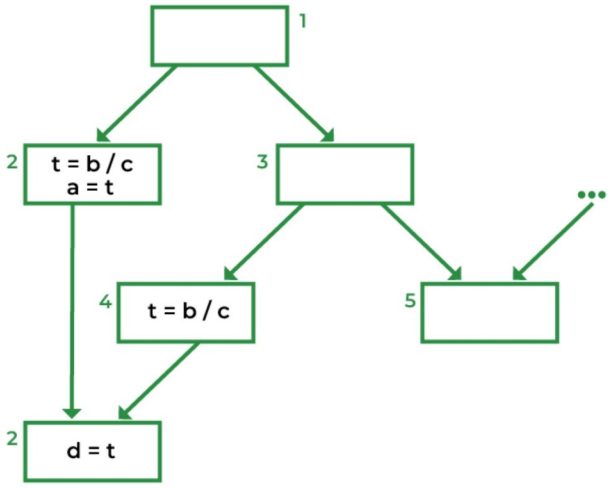


Figure 2

Thus, we can only eliminate the common expressions that do not generate any duplication in code.

The Lazy-Code-Motion Algorithm

After a code is optimized for eliminating redundancy, they are expected to have the following characteristics:

expressions that do not generate any duplication in code are eliminated.
2. The optimized code has not introduced any new computations to the previous code.
3. The computation of expressions is done at the latest possible time.

The Lazy code motion is the optimization of partial redundancy to perform the computations at the latest possible time in the code. This property is important as the common expression's values are stored in registers until their last use, hence, the latest computation of these expressions results in their smaller lifetime.

The intuition of Partial Redundancy of a single expression:

Consider an expression *E* which is redundant in block A, which means that *E* has been computed in all the execution paths that reach block A. Now, in this case, there must exist a set of blocks, say B, among which all blocks have the expression *E*, hence making it redundant in block A. The set of outgoing edges from B in the flowgraph necessarily forms a cut-set that can disconnect block A from the entry of the code, if removed.

In case of partial redundancy, the lazy code-motion algorithm tries to make copies of the expression *E* in the other execution paths and make it fully redundant. If optimization is successful, the optimized code's flowgraph will also contain the cut-set between block A and the entry.

Anticipation of Expressions:

To ensure there are no extra computations in the code, copies of partial redundant code should be inserted at points where the expression is anticipated. An expression *E* is said to be anticipated at a point *P* if the values being used in *E* are available at that point and all the execution paths leading from *P* evaluate the value of *E*.

Algorithm:

Lazy-Code Motion:

Step 1: Find all the anticipated expressions using a backward data-flow pass at each point in the code.

Step 2: Place the expressions where their values are anticipated along some other execution path.

Step 3: Find the postponable expressions using a forward data-flow pass and place at a point in the code where they cannot be postponed further.

Postponable expressions are those which are anticipated at some point in the code but they are not used for a long span of flow of code.

Step 4: Remove all the temporary variables assignment that are used only once in the complete code.

LOOPS IN FLOW GRAPH

A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

Dominators:

In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by d dom n. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

Example:

*In the flow graph below,
*Initial node,node1 dominates every node. *node 2 dominates itself
*node 3 dominates all but 1 and 2. *node 4 dominates all but 1,2 and 3.
*node 5 and 6 dominates only themselves,since flow of control can skip around either by goin through the other.
*node 7 dominates 7,8 ,9 and 10. *node 8 dominates 8,9 and 10.
*node 9 and 10 dominates only themselves.

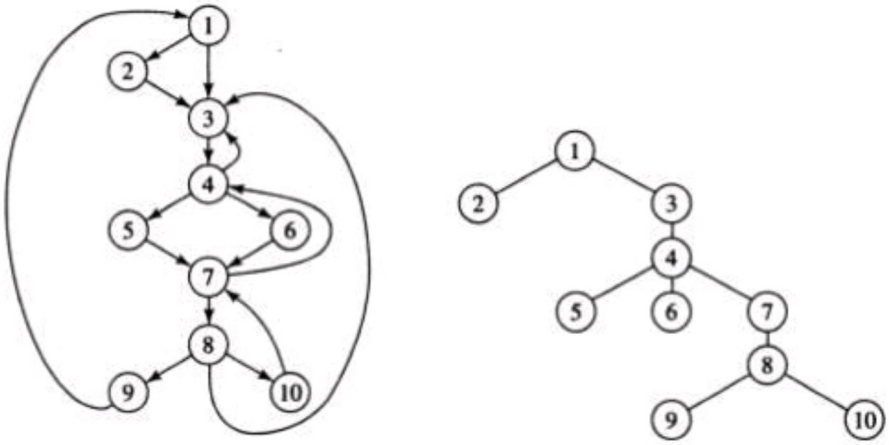


Fig. 5.3(a) Flow graph (b) Dominator tree

The way of presenting dominator information is in a tree, called the dominator tree, in which

- The initial node is the root.
- The parent of each other node is its immediate dominator.
- Each node d dominates only its descendents in the tree.

The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n. In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d dom m.

***

D(1)={1}
D(2)={1,2}
D(3)={1,3}
D(4)={1,3,4}
D(5)={1,3,4,5}
D(6)={1,3,4,6}
D(7)={1,3,4,7}
D(8)={1,3,4,7,8}
D(9)={1,3,4,7,8,9}
D(10)={1,3,4,7,8,10}

Natural Loops:

One application of dominator information is in determining the loops of a flow graph suitable for improvement. There are two essential properties of loops:

- Ø A loop must have a single entrypoint, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
- Ø There must be at least one way to iterate the loop(i.e.)at least one path back to the headerOne way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of

edges are called as back edges.

Example:

In the above graph,
7→4 4 DOM 7
10 →7 7 DOM 10
4→3
8→3
9 →1

The above edges will form loop in flow graph. Given a back edge n → 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: Constructing the natural loop of a back edge.

Input: A flow graph G and a back edge n→d.

Output: The set loop consisting of all nodes in the natural loop n→d.

Method: Beginning with node n, we consider each node m*d that we know is in loop, to make sure that m's predecessors are also placed in loop. Each node in loop, except for d, is placed once
on stack, so its predecessors will be examined. Note that because d is put in the loop initially, we never examine its predecessors, and thus find only those nodes that reach n without going through d.

```
Procedure insert(m);
if m is not in loop then begin loop := loop U {m};
push m onto stack end;
stack : = empty;
loop : = {d}; insert(n);
while stack is not empty do begin
pop m, the first element of stack, off stack;
        for each predecessor p of m do insert(p)
end
```

Inner loops:

If we use the natural loops as "the loops", then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.

When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

Pre-Headers:

Several transformations require us to move statements "before the header". Therefore begin treatment of a loop L by creating a new block, called the preheader. The pre-header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header. Edges from inside loop L to the header are not changed. Initially the pre-header is empty, but transformations on L may place statements in it.

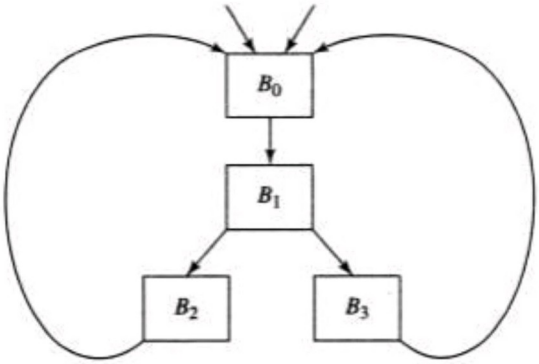


Fig. 5.4 Two loops with the same header

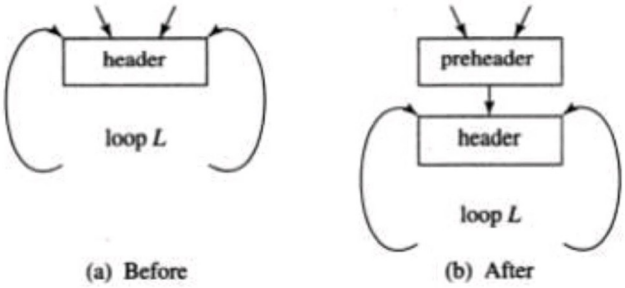


Fig. 5.5 Introduction of the preheader

Reducible flow graphs:

Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently. 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.

The most important properties of reducible flow graphs are that

1. There are no umps into the middle of loops from outside;
2. The only entry to a loop is through its header

Definition:

A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, forward edges and back edges, with the following properties.

1. The forward edges from an acyclic graph in which every node can be reached from initial node of G.
2. The back edges consist only of edges where heads dominate theirs tails.

Example: The above flow graph is reducible. If we know the relation DOM for a flow graph, we can find and remove all the back edges. The remaining edges are forward edges. If the forward edges form an acyclic graph, then we can say the flow graph reducible. In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph is acyclic.