

UNIT - V

Machine-Independent Optimization: The Principal Sources of Optimization, Introduction to Data-Flow Analysis, Foundations of Data-Flow Analysis, Constant Propagation, Partial-Redundancy Elimination, Loops in Flow Graphs.

5.1 The Principal Sources of Optimization

Code optimization is used to improve the intermediate code so that the output of the program could run faster and takes less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution without wasting resources. It tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

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. Local transformations are usually performed first.

Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes.

Function preserving transformations examples:

1. Common sub expression elimination
2. Copy propagation,
3. Dead-code elimination
4. Constant folding

The other transformations come up primarily when global optimizations are performed. Frequently, a program will include several calculations of the offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

1. Common-Subexpression 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.

2. 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. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x .

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.

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

Example:

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

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

4. 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
 $a=1.570$ there by eliminating a division operation.

5. 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:

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

6. Code Motion(or Frequency Reduction):

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop.

Example 1

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

Code motion will result in the equivalent of

```
t= limit-2;
```

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

Example2 :

Before optimization:	After optimization:
<pre>while(i<100) { a = Sin(x)/Cos(x) + i; i++; }</pre>	<pre>t = Sin(x)/Cos(x); while(i<100) { a = t + i; i++; }</pre>

7. Induction-variable elimination, which we apply to replace variables from inner loop.

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.

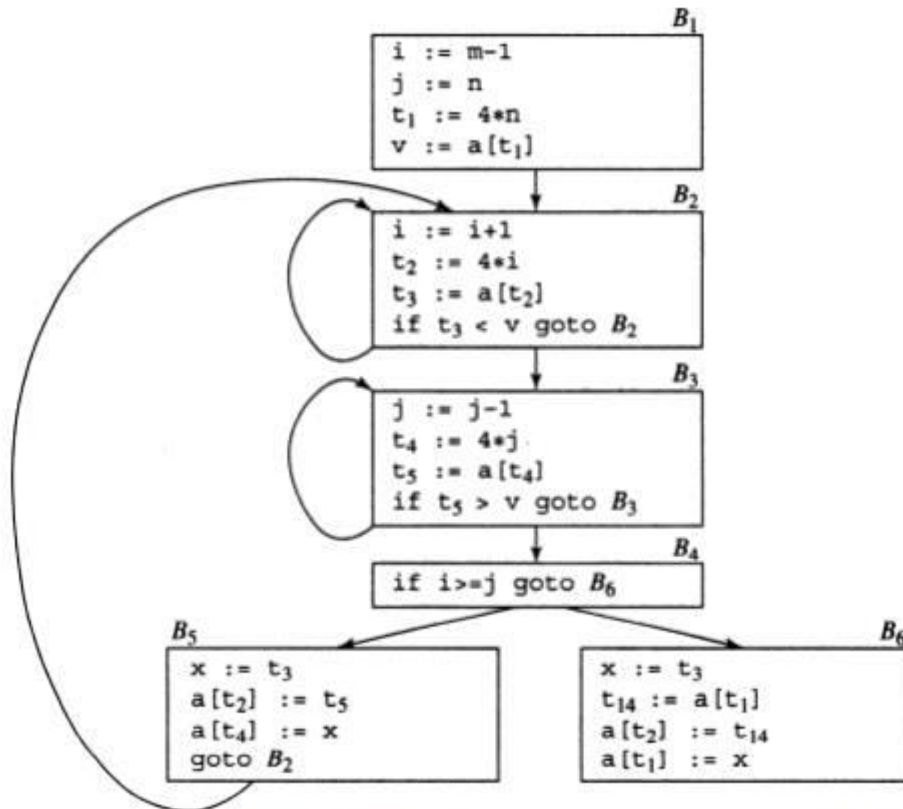


Fig. 5.3 B5 and B6 after common subexpression elimination

8. Reduction in Strength: The strength of certain operators is higher than other operators. For example, strength of $*$ is higher than $+$. Usually, compiler takes more time for higher strength operators and execution speed is less. Replacement of higher strength operator by lower strength operator is called a strength reduction technique

Optimization can be done by applying strength reduction technique where higher strength can be replaced by lower strength operators.

Example:

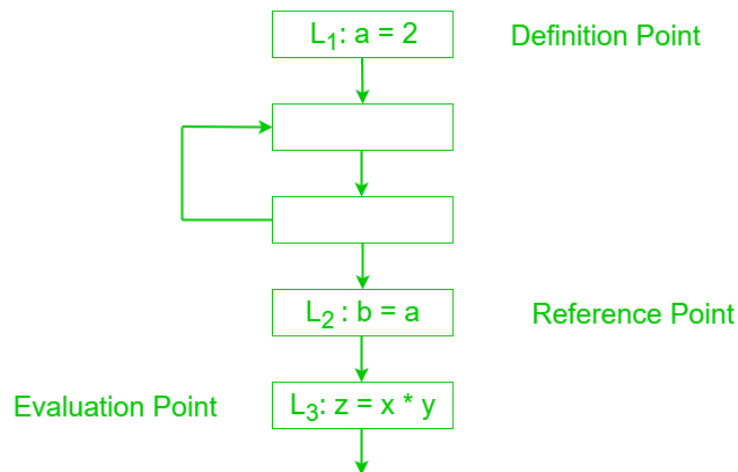
Before optimization:	After optimization:
<pre> for (i=1;i<=10;i++) { sum = i * 7; printf("%d", sum); } </pre>	<pre> temp = 7; for(i=1;i<=10;i++) { temp = temp + 7; sum = temp; printf("%d", sum) } </pre>

5.2 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.

Basic Terminologies –

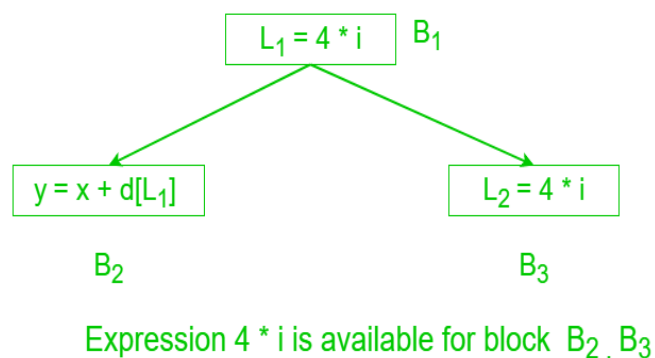
- **Definition Point:** a point in a program containing some definition.
- **Reference Point:** a point in a program containing a reference to a data item.
- **Evaluation Point:** a point in a program containing evaluation of expression.



Data Flow Properties –

1. **Available Expression** – An expression is said to be available at a program point x if along paths its reaching to x . An 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

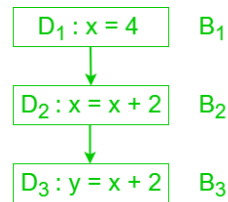


Advantage

It is used to eliminate common sub expressions.

2. Reaching Definition – A definition D reaches a point x if there is a path from D to x in which D is not killed, i.e., not redefined.

Example



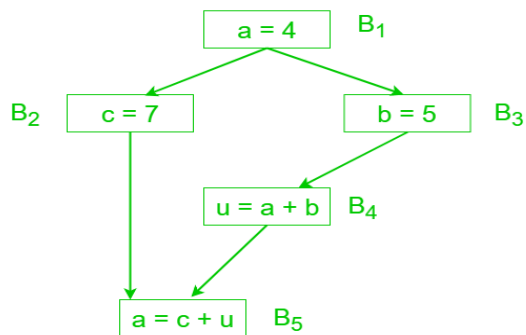
D₁ is reaching definition for B₂ but not for B₃ since it is killed by D₂

Advantage

It is used in constant and variable propagation.

3. 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 B₁, B₃, B₄ but killed at B₅

Advantage –

1. It is useful for register allocation.
2. It is used in dead code elimination.

4. 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.

5.3 PARTIAL-REDUNDANCY ELIMINATION

The expression or sub-expression that has been appeared and computed before and appears again during the computation of the code is the common sub-expression. Elimination of that sub-expression is known as **Common sub-expression elimination**.

The advantage of this elimination method is to make the *computation faster and better* by avoiding the re-computation of the expression. In addition, it utilizes memory efficiently.

Elimination of global common sub expressions:

- The available expressions data-flow problem discussed in the last section allows us to determine if an expression at point p in a flow graph is a common sub-expression. The following algorithm formalizes the intuitive ideas presented for eliminating common sub-expressions.

ALGORITHM: Global common sub expression elimination.

INPUT: A flow graph with available expression information. **OUTPUT:** A revised flow graph.

METHOD: For every statement s of the form $x := y+z$ such that y+z is available at the beginning of block and neither y nor z is defined prior to statement s in that block, do the following.

1. To discover the evaluations of y+z that reach s's block, we follow flow graph edges, searching backward from s's block. However, we do not go through any block that evaluates y+z. The last evaluation of y+z in each block encountered is an evaluation of y+z that reaches s.
2. Create new variable u.
3. Replace each statement w: $=y+z$ found in (1) by
 - a. $u := y + z$
 - b. $w := u$
4. Replace statement s by $x:=u$.

Some remarks about this algorithm are in order:

The search in step(1) of the algorithm for the evaluations of y+z that reach statement s can also be formulated as a data-flow analysis problem. However, it does not make sense to solve it for all expressions y+z and all statements or blocks because too much irrelevant information is gathered.

1. Not all changes made by algorithm are improvements. We might number of different evaluations reaching s found in step (1), probably to one.
2. Algorithm will miss the fact that $a*z$ and $c*z$ must have the same value in
$$\begin{array}{l} a := x+y \quad c := x+y \\ \text{vs} \\ b := a*z \quad d := c*z \end{array}$$

Because this simple approach to common sub expressions considers only the literal expressions themselves, rather than the values computed by expressions.

5.4 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 $d \text{ dom } n$. 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, node 1 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 going 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.

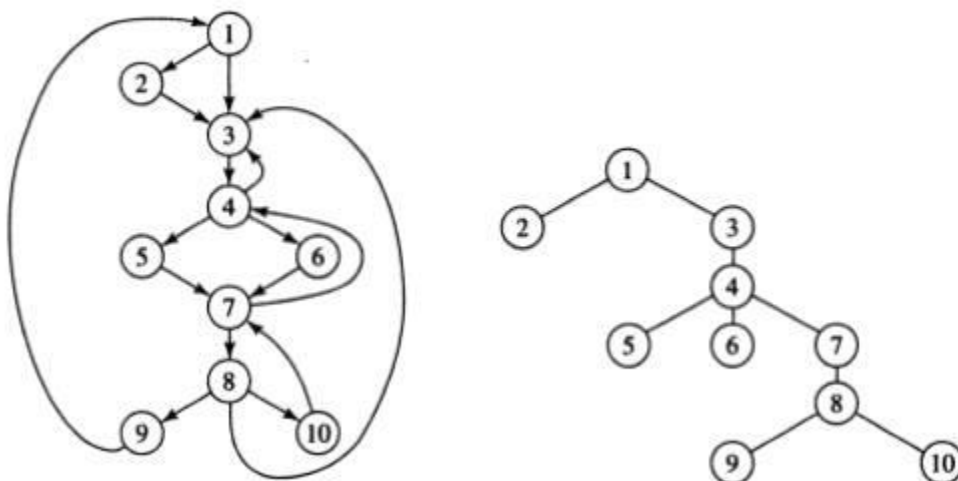


Fig. 5.3(a) Flow graph (b) Dominator tree

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 $d \neq n$ and $d \text{ dom } n$, then $d \text{ dom } m$.

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

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:
- 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.
- 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 $a \rightarrow b$ 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,

$7 \rightarrow 4$ 4 DOM 7

$10 \rightarrow 7$ 7 DOM 10

$4 \rightarrow 3$

$8 \rightarrow 3$

$9 \rightarrow 1$

The above edges will form loop in flow graph. Given a back edge $n \rightarrow d$, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d . Node d is the header of the loop.

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 disjoint 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.

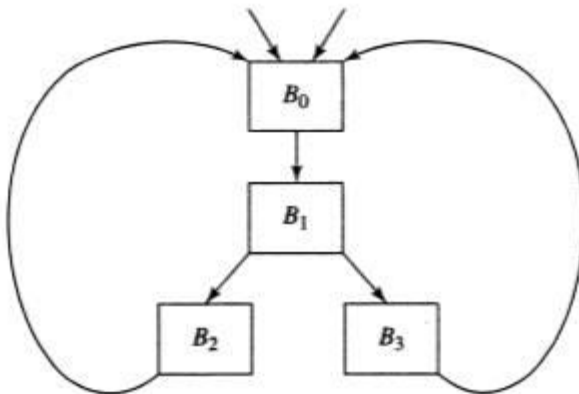


Fig. 5.4 Two loops with the same header

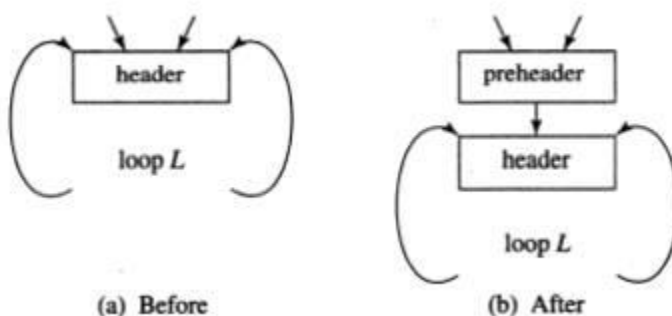


Fig. 5.5 Introduction of the preheader

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 jumps 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 form 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 their 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 is reducible. In the above example remove the five back edges $4 \rightarrow 3$, $7 \rightarrow 4$, $8 \rightarrow 3$, $9 \rightarrow 1$ and $10 \rightarrow 7$ whose heads dominate their tails, the remaining graph is acyclic.

5.5 Copy propagation:

Various algorithms introduce copy statements such as $x := \text{copies}$ may also be generated directly by the intermediate code generator, although most of these involve temporaries local to one block and can be removed by the dag construction. We may substitute y for x in all these places, provided the following conditions are met every such use u of x .

1. Statement s must be the only definition of x reaching u .
2. On every path from s to including paths that go through u several times, there are no assignments to y .

Condition (1) can be checked using ud -changing information. We shall set up a new data-flow analysis problem in which $in[B]$ is the set of copies $s: x := y$ such that every path from initial node to the beginning of B contains the statement s , and subsequent to the last occurrence of s , there are no assignments to y .

ALGORITHM: Copy propagation.

INPUT: a flow graph G, with ud-chains giving the definitions reaching block B, and with $c_in[B]$ representing the solution to equations that is the set of copies $x:=y$ that reach block B along every path, with no assignment to x or y following the last occurrence of $x:=y$ on the path. We also need ud-chains giving the uses of each definition.

OUTPUT: A revised flow graph.

METHOD: For each copy $s : x:=y$ do the following:

1. Determine those uses of x that are reached by this definition of namely, $s: x:=y$.
2. Determine whether for every use of x found in (1) , s is in $c_in[B]$, where B is the block of this particular use, and moreover, no definitions of x or y occur prior to this use of x within B. Recall that if s is in $c_in[B]$ then s is the only definition of x that reaches B.
3. If s meets the conditions of (2), then remove s and replace all u by y.

5.6 INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS

In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph. A compiler could take advantage of “reaching definitions” , such as knowing where a variable like debug was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.

Data-flow information can be collected by setting up and solving systems of equations of the form :

$$\text{out } [S] = \text{gen } [S] \cup (\text{in } [S] - \text{kill } [S])$$

This equation can be read as “ the information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement.” Such equations are called data-flow equation.

1. The details of how data-flow equations are set and solved depend on three factors. The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along

with flow of control and defining $out[S]$ in terms of $in[S]$, we need to proceed backwards and define $in[S]$ in terms of $out[S]$.

2. Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write $out[s]$ we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.
3. There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Points and Paths:

Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.

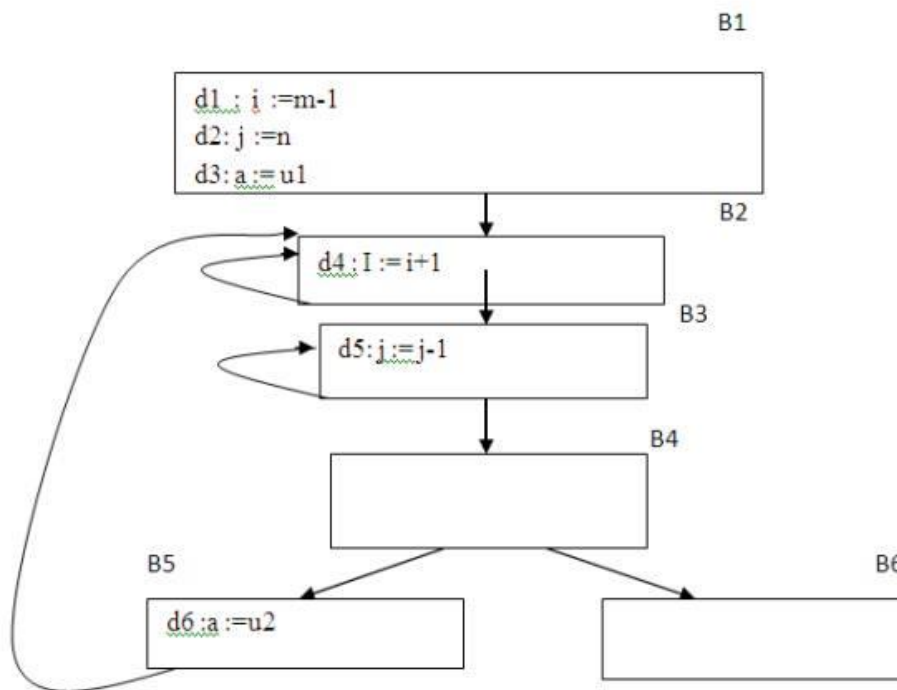


Fig. A flow graph

Now let us take a global view and consider all the points in all the blocks. A path from p_1 to p_n is a sequence of points p_1, p_2, \dots, p_n such that for each i between 1 and $n-1$, either

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

Reaching definitions:

A definition of variable x is a statement that assigns, or may assign, a value to x . The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x . These statements certainly define a value for x , and they are referred to as unambiguous definitions of x . There are certain kinds of statements that may define a value for x ; they are called ambiguous definitions.

The most usual forms of ambiguous definitions of x are:

1. A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure.
2. An assignment through a pointer that could refer to x . For example, the assignment $*q:=y$ is a definition of x if it is possible that q points to x . we must assume that an assignment through a pointer is a definition of every variable.

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. Thus a point can be reached by an unambiguous definition and an ambiguous definition of the appearing later along one path.

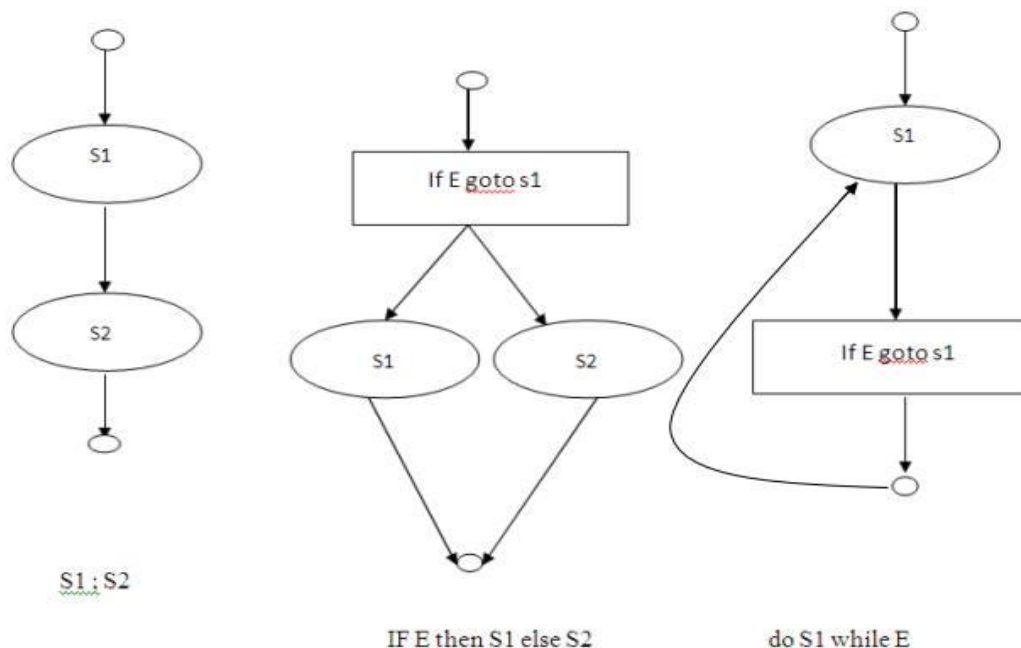


Fig. 5.7 Some structured control constructs

Data-flow analysis of structured programs:

Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from

when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

$S \rightarrow id := E \mid S; S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{do } S \text{ while } E$
 $E \rightarrow id + id \mid id$

Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.

We define a portion of a flow graph called a region to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header. The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

We say that the beginning points of the dummy blocks at the statement's region are the beginning and end points, respective equations are inductive, or syntax-directed, definition of the sets $\text{in}[S]$, $\text{out}[S]$, $\text{gen}[S]$, and $\text{kill}[S]$ for all statements S . $\text{gen}[S]$ is the set of definitions "generated" by S while $\text{kill}[S]$ is the set of definitions that never reach the end of S .

- Consider the following data-flow equations for reaching definitions :

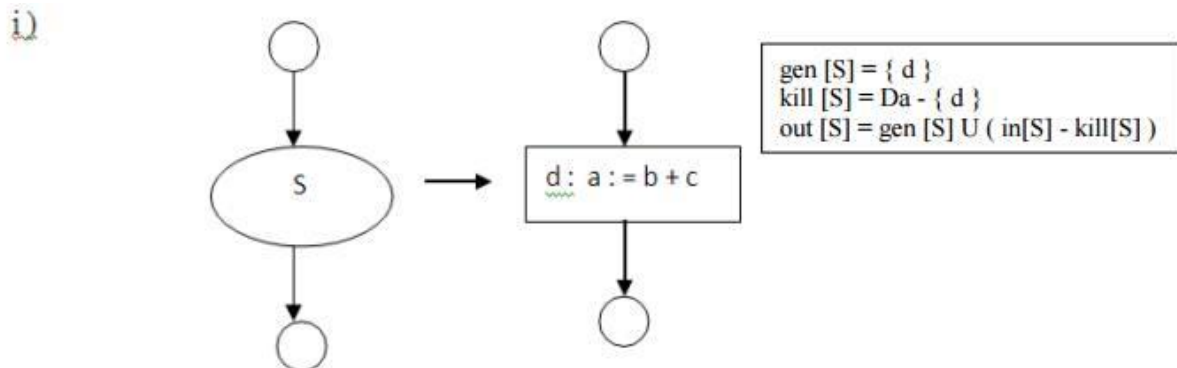


Fig. 5.8 (a) Data flow equations for reaching definitions

Observe the rules for a single assignment of variable a . Surely that assignment is a definition of a , say d . Thus

$$\text{gen}[S] = \{ d \}$$

On the other hand, d "kills" all other definitions of a , so we write

$$\text{Kill}[S] = \text{Da} - \{d\}$$

Where, Da is the set of all definitions in the program for variable a.

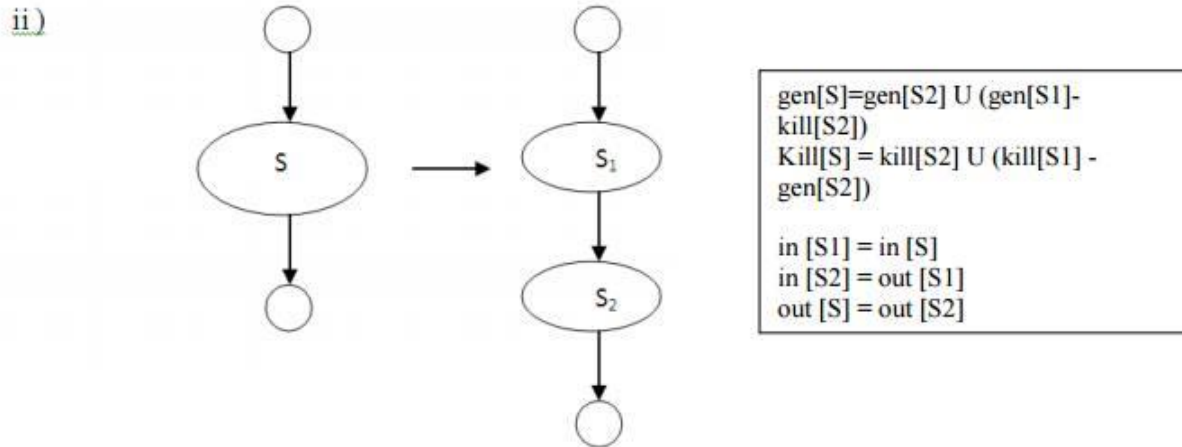


Fig. 5.8 (b) Data flow equations for reaching definitions

Under what circumstances is definition d generated by $S=S_1; S_2$? First of all, if it is generated by S_2 , then it is surely generated by S. if d is generated by S_1 , it will reach the end of S provided it is not killed by S_2 . Thus, we write

$$\text{gen}[S] = \text{gen}[S_2] \cup (\text{gen}[S_1] - \text{kill}[S_2])$$

Similar reasoning applies to the killing of a definition, so we have

$$\text{Kill}[S] = \text{kill}[S_2] \cup (\text{kill}[S_1] - \text{gen}[S_2])$$

Conservative estimation of data-flow information:

There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are “uninterpreted”; that is, there exists inputs to the program that make their branches go either way.

We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input. When we compare the computed gen with the “true” gen we discover that the true gen is always a subset of the computed gen. on the other hand, the true kill is always a superset of the computed kill.

These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.

Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.

Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached. Decreasing kill can only increase the set of definitions reaching any given point.

Computation of in and out:

Many data-flow problems can be solved by synthesized translation to compute gen and kill. It can be used, for example, to determine computations. However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in. we intend that $in[S]$ be the set of definitions reaching the beginning of S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.

The set $out[S]$ is defined similarly for the end of s. it is important to note the distinction between $out[S]$ and $gen[S]$. The latter is the set of definitions that reach the end of S without following paths outside S. Assuming we know $in[S]$ we compute out by equation, that is

$$Out[S] = gen[S] \cup (in[S] - kill[S])$$

Considering cascade of two statements S1; S2, as in the second case. We start by observing $in[S1]=in[S]$. Then, we recursively compute $out[S1]$, which gives us $in[S2]$, since a definition reaches the beginning of S2 if and only if it reaches the end of S1. Now we can compute $out[S2]$, and this set is equal to $out[S]$.

Consider the if-statement. we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S1 or S2 exactly when it reaches the beginning of S. That is,

$$in[S1] = in[S2] = in[S]$$

If a definition reaches the end of S if and only if it reaches the end of one or both substatements; i.e.,

$$\text{out}[S] = \text{out}[S1] \cup \text{out}[S2]$$

Representation of sets:

Sets of definitions, such as $\text{gen}[S]$ and $\text{kill}[S]$, can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set.

The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.

A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming languages. The difference $A-B$ of sets A and B can be implemented complement of B and then using logical and to compute $A \cap \text{complement of } B$.

Local reaching definitions:

Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.

Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

Use-definition chains:

It is often convenient to store the reaching definition information as "use-definition chains" or "ud-chains", which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable a in block B is preceded by no unambiguous definition of a, then ud-chain for that use of a is the set of definitions in $\text{in}[B]$ that are definitions of a. In addition, if there are ambiguous definitions of a, then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a.

Evaluation order:

The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after all uses of it have occurred. Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

General control flow:

Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax directed manner. When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.

Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval-based methods. However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

5.7 DETECTION OF LOOP-INVARIANT COMPUTATIONS:

Ud-chains can be used to detect those computations in a loop that are loop-invariant, that is, whose value does not change as long as control stays within the loop. Loop is a region consisting of set of blocks with a header that dominates all the other blocks, so the only way to enter the loop is through the header.

If an assignment $x := y+z$ is at a position in the loop where all possible definitions of y and z are outside the loop, then $y+z$ is loop-invariant because its value will be the same each time $x:=y+z$ is encountered. Having recognized that value of x will not change, consider $v:= x+w$, where w could only have been defined outside the loop, then $x+w$ is also loop-invariant.

ALGORITHM: Detection of loop-invariant computations.

INPUT: A loop L consisting of a set of basic blocks, each block containing sequence of three-address statements. We assume ud-chains are available for the individual statements.

OUTPUT: the set of three-address statements that compute the same value each time executed, from the time control enters the loop L until control next leaves L.

METHOD: we shall give a rather informal specification of the algorithm, trusting that the principles will be clear.

1. Mark “invariant” those statements whose operands are all either constant or have all their reaching definitions outside L.
2. Repeat step (3) until at some repetition no new statements are marked “invariant”.
3. Mark “invariant” all those statements not previously so marked all of whose operands either are constant, have all their reaching definitions outside L, or have exactly one reaching definition, and that definition is a statement in L marked invariant.

5.8 ELIMINATION OF CODE MOTION:

Having found the invariant statements within a loop, we can apply to some of them an optimization known as code motion, in which the statements are moved to pre-header of the loop. The following three conditions ensure that code motion does not change what the program computes. Consider $s: x = y + z$.

1. The block containing s dominates all exit nodes of the loop, where an exit of a loop is a node with a successor not in the loop.
2. There is no other statement in the loop that assigns to x . Again, if x is a temporary assigned only once, this condition is surely satisfied and need not be changed.
3. No use of x in the loop is reached by any definition of x other than will be satisfied, normally, if x is temporary.

ALGORITHM: Code motion.

INPUT: A loop L with ud-chaining information and dominator information.

OUTPUT: A revised version of the loop with a pre-header and some statements moved to the pre-header.

METHOD:

1. Use loop-invariant computation algorithm to find loop-invariant statements.
2. For each statement s defining x found in step(1), check:
 - i) That it is in a block that dominates all exits of L,

- ii) That x is not defined elsewhere in L , and
 - iii) That all uses in L of x can only be reached by the definition of x in statement s .
3. Move, in the order found by loop-invariant algorithm, each statement s found in (1) and meeting conditions (2i), (2ii), (2iii), to a newly created pre-header, provided any operands of s that are defined in loop L have previously had their definition statements moved to the pre-header.

To understand why no change to what the program computes can occur, condition (2i) and (2ii) of this algorithm assure that the value of x computed at s must be the value of x after any exit block of L . When we move s to a pre-header, s will still be the definition of x that reaches the end of any exit block of L . Condition (2iii) assures that any uses of x within L did, and will continue to, use the value of x computed by s .

5.9 ELIMINATION OF INDUCTION VARIABLE:

A variable x is called an induction variable of a loop L if every time the variable x changes values, it is incremented or decremented by some constant. Often, an induction variable is incremented by the same constant each time around the loop, as in a loop headed by `for i := 1 to 10`. However, our methods deal with variables that are incremented or decremented zero, one, two, or more times as we go around a loop. The number of changes to an induction variable may even differ at different iterations.

A common situation is one in which an induction variable, say i , indexes an array, and some other induction variable, say t , whose value is a linear function of i , is the actual offset used to access the array. Often, the only use made of i is in the test for loop termination. We can then get rid of i by replacing its test by one on t . We shall look for basic induction variables, which are those variables i whose only assignments within loop L are of the form $i := i + c$ or $i - c$, where c is a constant.

ALGORITHM: Elimination of Induction variable

INPUT: A loop L with reaching definition information, loop-in information and live variable information.

OUTPUT: A revised loop. **METHOD:**

1. Consider each basic induction variable i whose only uses are to compute other induction variables in its family and in conditional branches. Take some j in i 's family, preferably one such that c and d in its triple are as simple as possible and modify each test that i appears in to use j instead. We assume in the following that c is positive. A test of the form 'if i rel op x goto B ', where x is not an induction variable, is replaced by

```

a.  r := c*x      /* r := x if c is 1. */
b.  r := r+d      /* omit if d is 0 */
c.  if j relop r goto B

```

where, r is a new temporary. The case ‘if x relop i goto B’ is handled analogously. If there are two induction variables i1 and i2 in the test if i1 relop i2 goto B, then we check if both i1 and i2 can be replaced. The easy case is when we have j1 with triple and j2 with triple, and c1=c2 and d1=d2. Then, i1 relop i2 is equivalent to j1 relop j2.

2. Now, consider each induction variable j for which a statement j:=s was introduced. First check that there can be no assignment to s between the introduced statement j:=s and any use of j. In the usual situation, j is used in the block in which it is defined, simplifying this check; otherwise, reaching definitions information, plus some graph analysis is needed to implement the check. Then replace all uses of j by uses of s and delete statement j:=s.