

CODE IMPROVING TRANSFORMATIONS

Algorithms for performing the code improving transformations rely on data-flow information. Here we consider common sub-expression elimination, copy propagation and transformations for moving loop invariant computations out of loops and for eliminating induction variables. Global transformations are not substitute for local transformations; both must be performed.

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:

1. 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.
2. Not all changes made by algorithm are improvements. We might number of different evaluations reaching s found in step (1), probably tooone.
3. Algorithm will miss the fact that $a*z$ and $c*z$ must have the same value in

$a := x+y$ $c := x+y$
vs
 $b := a*z$ $d := c*z$

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

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 $s: x:=y$.
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 .

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. A loop is a region consisting of a 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 a 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.

Performing 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 statements.
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 .

Alternative code motion strategies:

The condition (1) can be relaxed if we are willing to take the risk that we may actually increase the running time of the program a bit; of course, we never change what the program computes. The relaxed version of code motion condition (1) is that we may move a statement s assigning x only if:

1'. The block containing s either dominates all exits of the loop, or x is not used outside the loop. For example, if x is a temporary variable, we can be sure that the value will be used only in its own block.

If code motion algorithm is modified to use condition (1'), occasionally the running time will increase, but we can expect to do reasonably well on the average. The modified algorithm may move to pre-header certain computations that may not be executed in the loop. Not only does this risk slowing down the program significant an error in certain circumstances.

Even if none of the conditions of (2i), (2ii), (2iii) of code motion algorithm are met by an assignment $x := y+z$, we can still take the computation $y+z$ outside a loop. Create a new temporary t , and set $t := y+z$ in the pre-header. Then replace $x := y+z$ by $x := t$ in the loop. In many cases we can propagate out the copy statement $x := t$.

Maintaining data-flow information after code motion:

The transformations of code motion algorithm do not change ud-chaining information, since by condition (2i), (2ii), and (2iii), all uses of the variable assigned by a moved statement that were reached by s are still reached by s from its new position. Definitions of variables used by s are either outside L , in which case they reach the pre-header, or they are inside L , in which case by step (3) they were moved to pre-header ahead of s .

If the ud-chains are represented by lists of pointers to pointers to statements, we can maintain ud-chains when we move statement s by simply changing the pointer to s when we move it. That is, we create for each statement s pointer ps , which always points to s . We put the pointer on each ud-chain containing s . Then, no matter where we move s , we have only to change ps , regardless of how many ud-chains s is on.

The dominator information is changed slightly by code motion. The pre-header is now the immediate dominator of the header, and the immediate dominator of the pre-header is the node that formerly was the immediate dominator of the header. That is, the pre-header is inserted into the dominator tree as the parent of the header.

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 $i := i - c$, 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 into use j instead. We assume in the following that c is positive. A test of the form 'if i relop x goto B ', where x is not an induction variable, is replaced by

- a. $r := c * x$ $/* r := x \text{ if } c \text{ is } 1. */$
- b. $r := r + d$ $/* \text{ omit if } d \text{ is } 0. */$
- 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 i_1 and i_2 in the test if i_1 relop i_2 goto B , then we check if both i_1 and i_2 can be replaced. The easy case is when we have j_1 with triple and j_2 with triple, and $c_1 = c_2$ and $d_1 = d_2$. Then, i_1 relop i_2 is equivalent to j_1 relop j_2 .

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 $j := s$.

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:

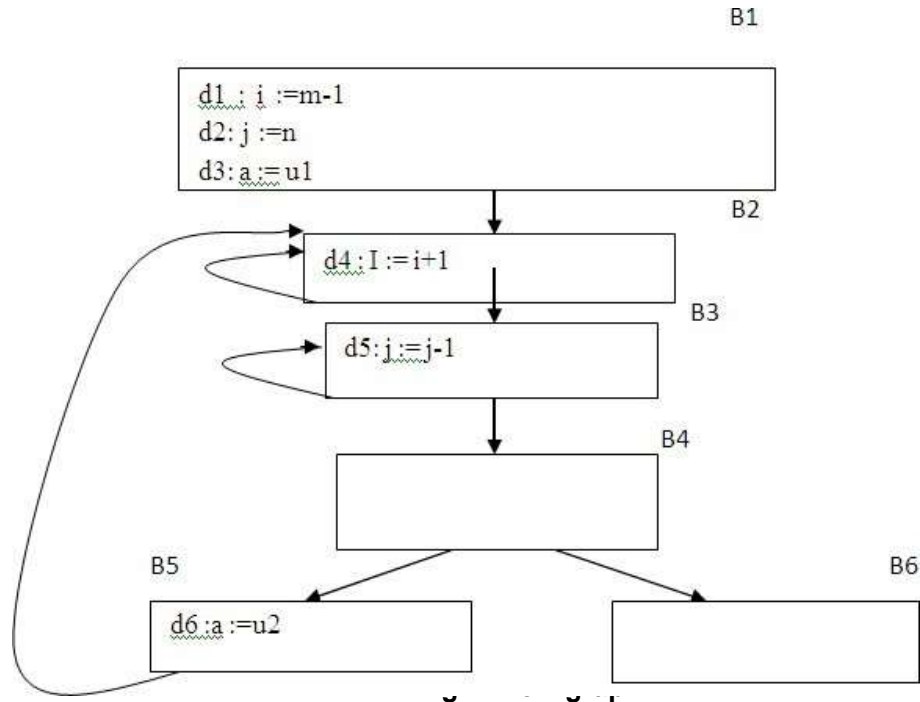
$$\text{out [S]} = \text{gen [S]} \cup (\text{in [S]} - \text{kill [S]})$$

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 equations.

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 endpoints.
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.



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.

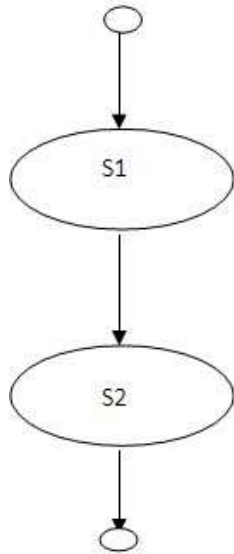
Reaching definitions:

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.

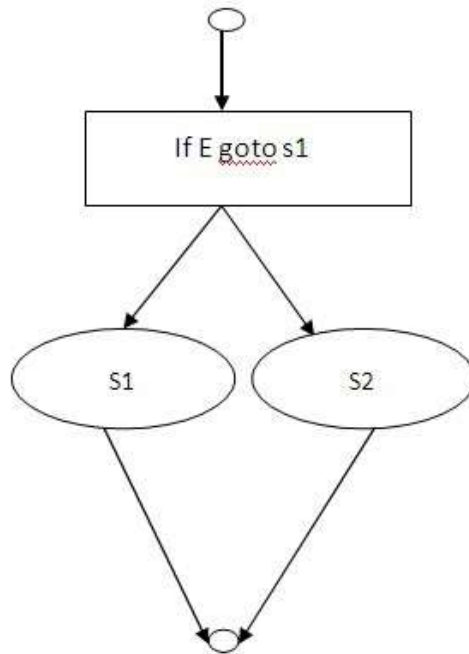
The most usual forms of ambiguous definitions of x are:

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.

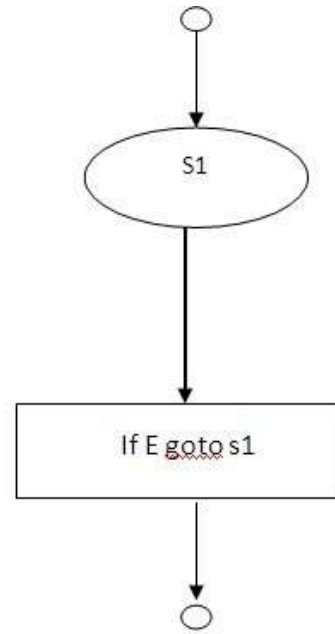
We say a definition d reaches a point p if there is a path from the point immediately following d to p , such that d is not "killed" along that path. Thus a point can be reached by an unambiguous definition and an ambiguous definition of the appearing later along one path.



S1; S2



IF E then S1 else S2



do S1 while E

m

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.

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

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 $in[S]$, $out[S]$, $gen[S]$, and $kill[S]$ for all statements S . $gen[S]$ is the set of definitions "generated" by S while $kill[S]$ is the set of definitions that never reach the end of S .

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

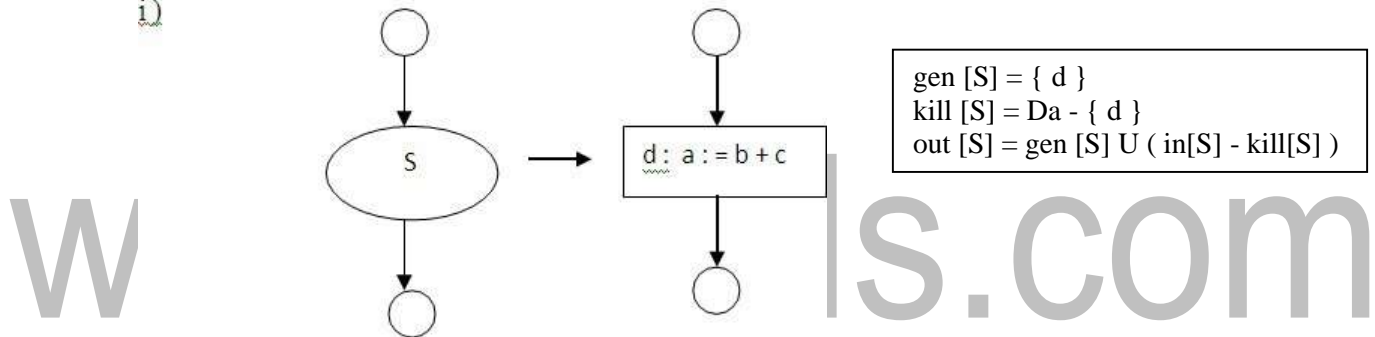


Fig. Data flow equations for reaching definitions

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

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

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

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

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

ii)

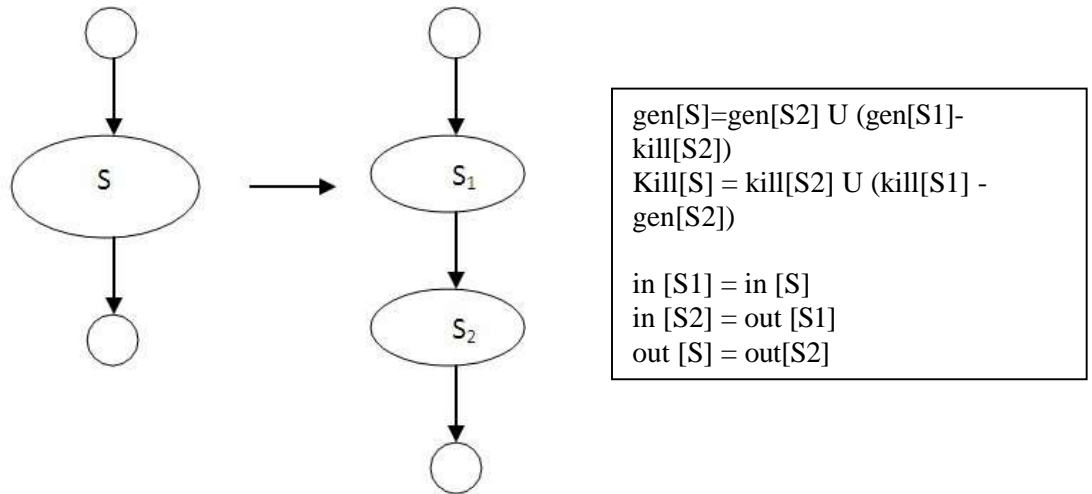


Fig. Data flow equations for reaching definitions

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 a cascade of two statements $S_1; S_2$, as in the second case. We start by observing $in[S_1] = in[S]$. Then, we recursively compute $out[S_1]$, which gives us $in[S_2]$, since a definition reaches the beginning of S_2 if and only if it reaches the end of S_1 . Now we can compute $out[S_2]$, 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 S_1 or S_2 exactly when it reaches the beginning of S . That is,

$$in[S_1] = in[S_2] = in[S]$$

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

$$out[S] = out[S_1] \cup out[S_2]$$

Representation of sets:

Sets of definitions, such as $gen[S]$ and $kill[S]$, 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 l if and only if the definition numbered l 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 by computing the complement of B and then using logical and to compute $A \cap \bar{B}$.

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 on 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.

OPTIMIZATION OF BASICBLOCKS

There are two types of basic block optimizations. They are :

- ☐ Structure-Preserving Transformations
- ☐ Algebraic Transformations

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- ☐ Common sub-expression elimination
- ☐ Dead code elimination
- ☐ Renaming of temporary variables
- ☐ Interchange of two independent adjacent statements.

Common sub-expression elimination:

Common subexpressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced. Example: $a = b + c$; $b = a - d$; $c = a - d$; $d = a - d$

The 2nd and 4th statements compute the same

expression: $b + c$ and $a - d$. Basic block can be transformed

to

$a = b + c$; $b = a - d$; $c = a - d$; $d = b$

Dead code elimination:

It is possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program - once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

Renaming of temporary variables:

A statement $t = b + c$ where t is a temporary name can be changed to $u = b + c$ where u is another temporary name, and change all uses of t to u . In this a basic block is transformed to its equivalent block called normal-form block.

Interchange of two independent adjacent statements:

- Two statements $t1 = b + c$ $t2 = x + y$

can be interchanged or reordered in its computation in the basic block when value of $t1$ does not affect the value of $t2$.

Algebraic Transformations:

Algebraic identities represent another important class of optimizations on

basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength. Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression $2*3.14$ would be replaced by 6.28 .

The relational operators \leq , \geq , $<$, $>$, $+$ and $=$ sometimes generate unexpected common sub expressions. Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

a :=b+c

e :=c+d+b

the following intermediate code may be generated: $a:=b+c; t:=c+d; e:=t+b$

Example:

$x:=x+0$ can be removed

$x:=y**2$ can be replaced by a cheaper statement $x:=y*y$

The compiler writer should examine the language specification carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate $x*y-x*z$ as $x*(y-z)$ but it may not evaluate $a+(b-c)$ as $(a+b)-c$.

PEEPHOLE OPTIMIZATION

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

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

The peephole is a small, moving window on the target program. The code in the peephole need not be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

Characteristics of peephole optimizations:

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

Redundant Loads And Stores:

If we see the instructions sequence

- (1) MOVR0,a
- (2) MOVa,R0

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

Unreachable Code:

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

```
#define debug 0
....
if ( debug ) {
    Print debugging information
```

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

```
If debug =1 goto L1 goto L2
   L1:  print debugging
   information L2: ..... (a)
```

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

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

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

☐ As the argument of the statement of (c) evaluates to a constant true it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

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

www.binils.com

```
   goto L1
   ....
L1:  goto L2                               (d)
```

by the sequence

```
   goto L2
   ....
L1:  goto L2
```

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

```
   if a < b goto L1
   ....
L1: goto L2                               (e)
```

can be replaced by

```
   If a < b goto L2
```


auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like $i := i + 1$.

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

$i := i - 1 \rightarrow i--$

www.binils.com

PRINCIPAL SOURCES OF OPTIMISATION

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:

- ☒ Common sub expression elimination
- ☒ Copy propagation,
- ☒ Dead-code elimination
- ☒ 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.

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 subexpression $4 \cdot i$ 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. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x .

- 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. 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.

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,

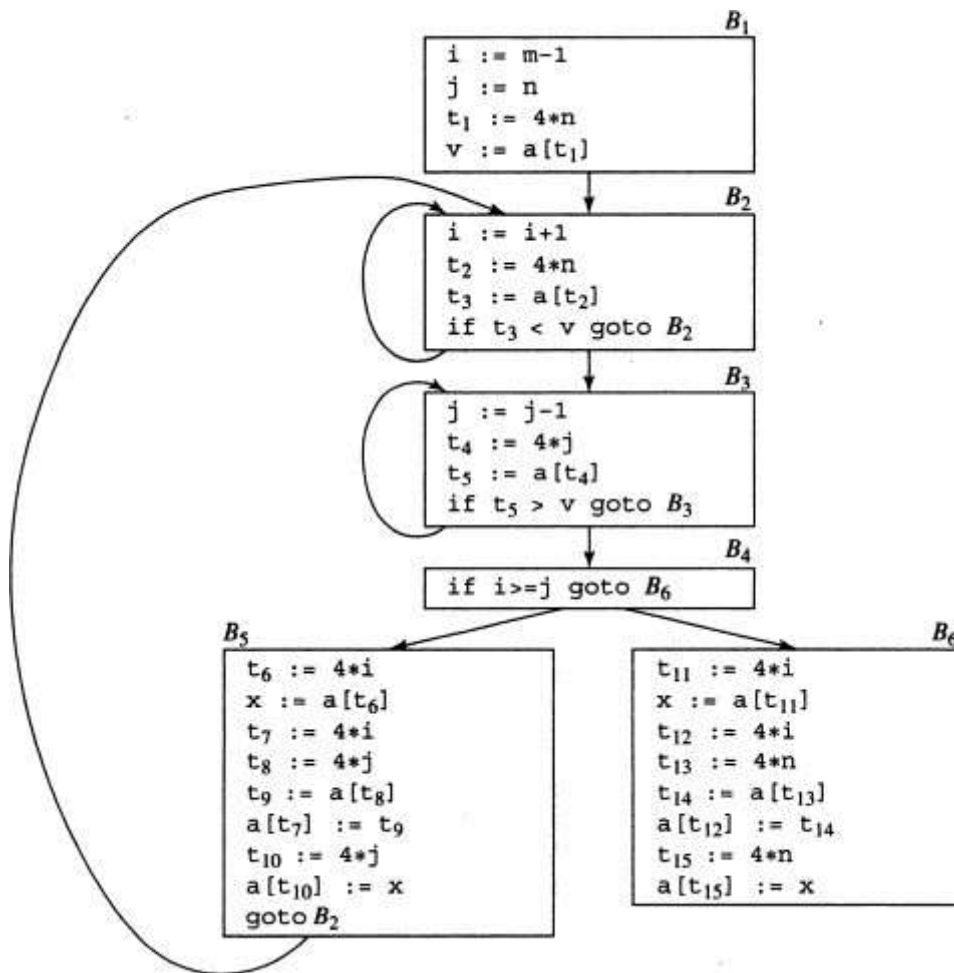
$a=3.14157/2$ can be replaced by
 $a=1.570$ there by eliminating a division operation.

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 loop;
- ☐ Induction-variable elimination, which we apply to replace variables from innerloop.
- ☐ Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.



.com

Code Motion:

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. For example, evaluation of $limit-2$ is a loop-invariant computation in the following while-statement:

```
while(i<=limit-2)/* statement does not change limit*/Code
motion will result in the equivalent of
t= limit-2;
while(i<=t) /* statement does not change limit or t*/
```

Induction Variables :

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

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

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

Example:

As the relationship $t4:=4*j$ surely holds after such an assignment to $t4$ in Fig. and $t4$ is not changed elsewhere in the inner loop around B3, it follows that just after the statement $j:=j-1$ the relationship $t4:=4*j-4$ must hold. We may therefore replace the assignment $t4:=4*j$ by $t4:=t4-4$. The only problem is that $t4$ does not have a value when we enter block B3 for the first time. Since we must maintain the relationship $t4=4*j$ on entry to the block B3, we place an initialization of $t4$ at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in Fig.5.3.

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

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.

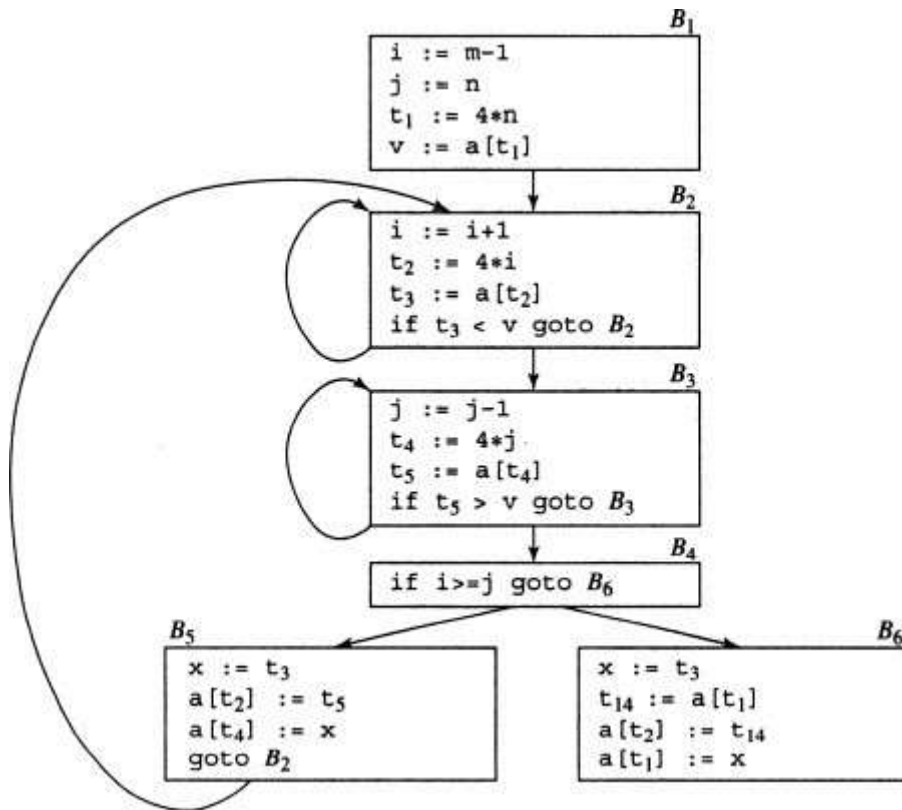


Fig. B5 and B6 after common sub expression elimination

www.binils.com