Compiler Theory:
Code Optimization

Marc Moreno Maza
University of Western Ontario
CS447a – Fall 2004

December 1, 2004

1 Code Optimization

<u>Code Optimization.</u> The target code generated by compilers is still not as good as code produced by an expert assembly language programmer.

- Compilers need to perform *code improving transformations* in order to reduce the running time of the target code.
- Such transformations are called optimization though very rarely will the code be *optimal* in the formal sense of the word.
- The transformations performed by an optimizing compiler should satisfy the following properties:
  - the transformation should preserve the meaning (semantics) of programs,
  - the transformation should, on average, speed up programs by a measurable amount,
  - though the size of programs is not generally an issue in itself, it should remain reasonable.
  - the amount of compiler effort put into optimizing a section of code should be relative to
    * the number of times the code will be run

* to the entire program

In general code optimization problems

- are NP-complete and
- thus we cannot expect to find a global optimum in a reasonable amount of time.
- Hence the strategy is mainly based on heuristics, that is
- identify *hot areas* and put a lot of resources into optimizing these.

Some examples of hot areas

- loops and nested loops
- optimize or eliminate procedure calls
- recognize common subexpressions
- propagate constants
- and lots lots more

2 Introduction to three-address code optimization

2.1 Code analysis.

<u>Optimizations to three-address code</u> (TAC) are portable to different back-ends. They rely on some simple concepts

- basic blocks
- control flow graphs

<u>Basic blocks.</u> A *basic block* is a TAC sequence where flow of control

- enters at the beginning and
- leaves at the end.

```
L3:  t5 := a * b
     t6 := t5 + c
     d := t2 * t2
     if d = 0 goto L4
```

<u>METHOD TO RECOGNIZE BASIC BLOCKS.</u>

1. Statements that start a basic block are identified by searching for
   - first statement of any function
   - any labeled statement that is the target of a jump (conditional or unconditional)
   - any statement following a jump (conditional or unconditional)
2. for each statement starting a basic block, the block consists of all statements up to (but excluding) the start of the next basic block or the end of the program.

<u>MANIPULATING TAC IN A BASIC BLOCK</u>

- implies that we need to worry only about effects caused to the values of variables
  - at the start of a block and
  - at its end
- is generally referred to as *local optimization.*

<u>CONTROL FLOW GRAPH.</u> Basic Blocks can be organized into a directed graph: the *Control Flow Graph* (CFG)

- whose *nodes* are the basic blocks and
- such that there is an *edge* that goes from basic block `A` to basic block `B` if it is possible for control to *flow* from `A` to `B`.

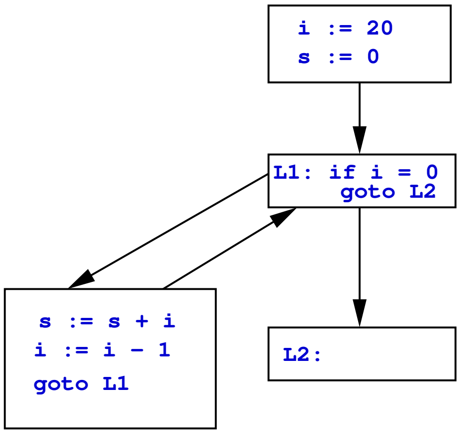

Figure 1: A control flow graph.

```
     i := 20
     s := 20
L1:  if i=0 goto L2
     s := s + i
     i := i - 1
     goto L1
L2:
```

<u>LIVE VARIABLE ANALYSIS.</u> We say that the TAC statement

`x := y op z`

- *references* `y` and `z` and
- *defines* `x`.

A name in a basic block B is said to be *live after* B if it is referenced further in the program after B, otherwise it is said to be *dead* in B.

Warning! In a basic block B, any piece of code of the form `x := y op z` where `x` is a variable dead in B is **not** necessarily a useless piece of code. Indeed `x` may be a temporary used to build a value propagated after B.

Algorithm 1 computes the dead names of a block B that are *never* referenced for computing the right values of a live variable. We will call such names *useless*.

Determining the life time and the use of variables in a program is called *live variable analysis.* We can say that a basic block *computes a set of expressions*: the values of variables that are live when we exit the block

2.2 An overview of some basic optimization techniques

<u>Determining useless names of a basic block.</u> In the following algorithm, for a basic block B we assume that $\mathsf{lives}(B)$ is the set of the variables appearing in B that are referred in a basic block B' following B in the CFG.

Algorithm 1

Input: A basic block B consisting of TAC statements $s_1, \ldots, s_\ell$ in this order.

Output: The set of the useless names in B.

$D := \emptyset$
$L := \mathsf{lives}(B)$
for $i := \ell..1$ **step** -1 **repeat**
  let s_i be `x := y op z`
  if `x` $\notin L$
    then $D := D \bigcup \{$ `x` $\}$
    else $L := L \bigcup \{$ `y`, `z` $\}$
return $D \setminus L$

<u>Dead Code Elimination</u> consists of the following transformations

- Removal of any TAC statement `x := y op z` such that `x` is not referenced further in the program.
- Replacement of any jump to a dead code line by a jump to whatever follow this line.

Dead code arises for a number of reasons.

- copy propagation is a common one.

<u>COMMON SUBEXPRESSION ELIMINATION</u> Consider the following TAC.

```
t1 := 4 - 2
t2 := t1
t3 := a * t2
t4 := t3 * t1
t5 := t4 + b
t6 := t3 * t1
t7 := t6 + b
c := t5 * t7
```

Note that the values of `t1` and `t3` cannot change between the two computations of `t4` and `t6`. Hence the line

```
t6 := t3 * t1
```

can be replaced by

```
t6 := t4
```

Of course this is a fairly trivial case. But a compiler could recognize much more complex common subexpressions.

<u>COPY PROPAGATION.</u>After a statement `x := y` we know that `x` and `y` have the same value, we can replace all occurrences of `x` with `y` between this assignment and the next definition of `y`.

Copy propagation makes further subexpressions elimination possible.

<u>ARITHMETIC TRANSFORMATIONS.</u> If possible, arithmetic expressions should be computed at compile time.

- This is called *Constant Folding.*
- Constant folding can also be used to eliminate conditional branches.

Example 1 On most machines

- division is much more expensive than multiplication (8 times as slow is typical)
- and multiplication is typically 1-2 times as slow as addition.

For example the code

```
t3:= t1 / y
t4:= t2 / y
```

would be better executed as

```
t5 := 1/y
t3 := t1 * t5
t4 := t2 * t5
```

Example 2 Another common example is the *fused multiply-add*. On many architectures there is an instruction

```
fma R1,R2,R3,R4
```

which assigns

```
R1:=R2+R3*R4
```

and runs in the time of a single multiply. If we can re-order code to recognize such opportunities we can significantly improve common loops.

PACKING TEMPORARIES. We can replace two distinct temporaries by a single one if they are not simultaneously live. For example the code

```
t4 := a + a
t5 := t4 + b
c := t5 * t5
```

would be better executed as

```
t1 := a + a
t1 := t1 + b
c := t1 * t1
```

One of the main difficulties is deciding in which order to apply these optimizations, and how many times to apply them ...

2.3 Data-flow Analysis and Global Optimization

<u>Data-flow analysis</u> provides tools to get increased information about the program such that

- we can extend optimization across multiple basic blocks and such that
- we use new optimizations applicable to groups of basic blocks forming loops.

Global data-flow analysis can provide the following types of information

liveness of variables

available expressions: if we have two identical expressions in separate blocks, we need to know if the variables they use hold the same value

reaching definitions: for any TAC statement `x:= y op z`, we need to know where the operands `y` and `z` were defined.

copy statements: in order to extend copy propagation to more than a single basic block, one needs to know which copy statements remain unchanged across blocks

<u>Global optimization</u> uses transformations applicable to groups of basic blocks, usually groups of blocks forming loops.

Global analysis and global optimization are studied in the next sections.

3 Data-flow analysis: identifying loops

<u>A loop L in a control flow graph G</u> is a subgraph satisfying the following properties.

- There exists a path from any node of L to any other node of L.
- There is only one node h of L such that there exists a node n of G which is not a node of L and (n, h) is an edge of G. This node h is called the *entry point* or the *header* of the loop.

A loop that contains no other loop is called an *inner loop*. To identify loops we need the following two notions.

IN A CONTROL FLOW GRAPH G, A DOMINATOR for a node n is a node d such that any path from the entry of the CFG to n goes through d. In this case we also say that d *dominates* n.

Algorithm 2

Input: G a CFG with set of nodes N, set of edges E and initial node n_0.

Output: for every $n \in N$, the set $D(n)$ of dominators of n.

$D(n_0) := \{n_0\}$
for $n \in N \setminus \{n_0\}$ **repeat**
  $D(n) := N$
while changes to any $D(n)$ occur **repeat**
  $D(n) := \{n\} \bigcup \bigcap_{p \in N \ | \ (p,n) \in E} D(p)$

- This algorithm terminates since $\Sigma_{n \in N} \ | \ D(n) \ |$ cannot decrease infinitely many times.
- The correctness of the algorithm can be made easily by induction on the maximal length of an elementary path from n to n_0.

Observe that the header h of a loop L dominates all other nodes of L. Otherwise it cannot be the only entry point.

A BACK EDGE IN A CFG G is an edge (p, d) of G such that d dominates p. To each back edge one can associate a loop as follows.

THE NATURAL LOOP OF A BACK EDGE (n, d) is a loop of its CFG computed by the following algorithm.

Algorithm 3

Input: G a CFG and (n, d) a back edge of G

Output: the set of nodes N_L of a maximal loop L for which (n, d) is an edge.

$N_L := \{n, d\}$
stack := *empty*
push(n, *stack*)
while $stack \neq [\,]$ **repeat**
  $m :=$ top(*stack*)
  pop(*stack*)
  for each predecessor p of m **repeat**
    if $p \notin N_L$ **repeat**
      $N_L := N_L \cup \{p\}$
      push(p, *stack*)
return N_L

Example 3 Consider the following TAC fragment.

```
    s:=0
    i:=0
    n:=10
L1: t1 := a-b
    ifz t1 goto L2
    t2 := i*4
    s := s+t2
    goto L3
L2: s := s+i
L3: i := i+1
    t3 := n-i
    ifnz t3 goto L1
    t4 := a-b
```

Its control flow graph is shown on Figure 2. There is one back edge: (B_5, B_2). Its natural loop consists of the nodes B_2, B_3, B_4, B_5. Occasionally it is difficult to determine when a loop is nested.

<u>Loop optimizations</u> often involve

- moving code out of the loop or

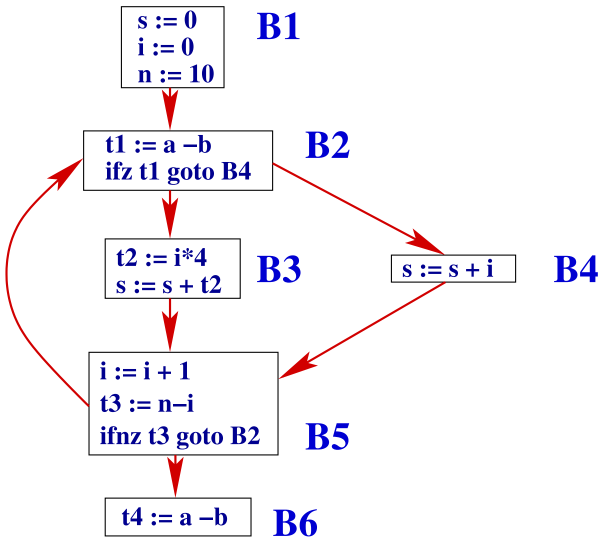

Figure 2: A control flow graph with two loops.

- pre-computing values for use in the loop

To do so, we need a basic block in which we insert such code

- before any loop optimization we create a new block which is inserted immediately before the header
- such a block is called a *pre-header*

<u>Reducible control flow graphs.</u> A control flow graph G is said to be *reducible* if the removal of its back edges leads to an acyclic graph where each node can be reached from the initial node of G. In this case, all the edges of this acyclic graph are called *forward edges*. Figure 3 shows a control flow graph which is not reducible. Indeed it has no loops (in the sense of the above definition) and it is not acyclic.

Figure 4 sketches a simple algorithm for determining whether a control flow graph is non-reducible. When no more *reductions* are possible, the initial CFG is reducible if and only if the reduced CFG consists only of a single node. In the first situation, one can *contract* the nodes 1 and 2 since it is allowed to arrive at 2 only from 1. In the second case, one is allowed to remove the loop (in the sense of graph theory) around the node.

One can show the following properties about reducible CFG

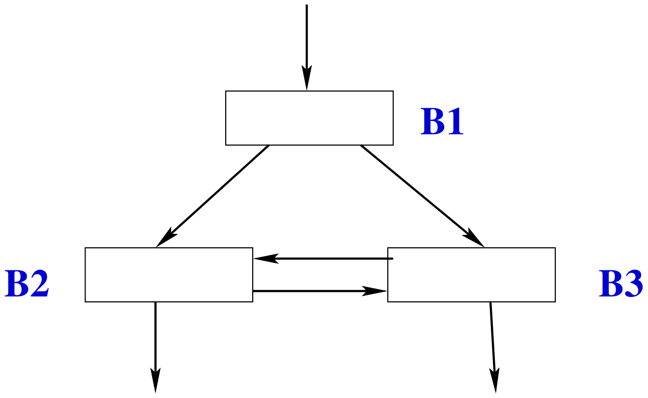

Figure 3: A control flow graph which is not reducible.

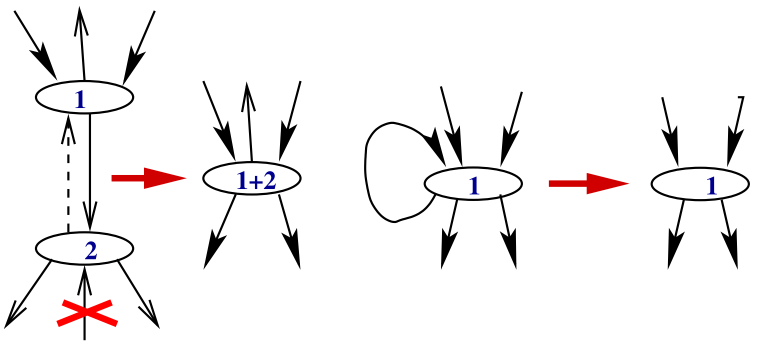

Figure 4: Determining whether a control flow graph is reducible or not.

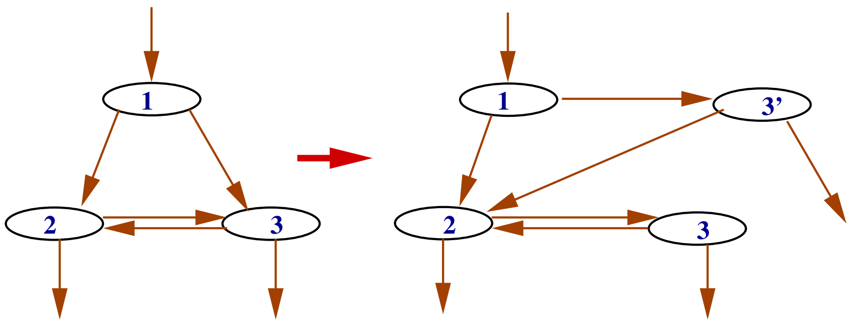

Figure 5: Dealing with improper regions by node-splitting.

- Non-reducible flow graphs arise basically only from unstructured use of goto-statements (jumps into the middle of a loop from the outside of the loop, without using the header).
- The key property of reducible control flow graphs is that any set of nodes that intuitively appears as a loop, contains a back edge.
- Thus we only need to identify the back edges to find all loops in a reducible control flow graph.

Figure 5 sketches a technique of *node-splitting* for dealing with the *improper regions* of non-reducible graphs.

4 Data-flow analysis: reaching definitions

Recall that we consider a TAC program $\mathcal{P}$. We assume that it is generated from a high level language where statements are

- assignments,
- alternatives,
- **while** loops.

So our high level program may be generated by a grammar of the form

$$
\begin{array}{lcl}
S & \longmapsto & \mathbf{id :=}\ \mathrm{E} \\
S & \longmapsto & S\ \mathbf{;}\ S \\
S & \longmapsto & \mathbf{if}\ E\ \mathbf{then}\ S\ \mathbf{else}\ S \\
S & \longmapsto & \mathbf{while}\ E\ \mathbf{do}\ S \\
E & \longmapsto & E + E \\
E & \longmapsto & E - E \\
E & \longmapsto & E = E \\
E & \longmapsto & E < E \\
E & \longmapsto & \mathbf{id} \\
E & \longmapsto & \mathbf{num} \\
E & \longmapsto & \mathbf{bool}
\end{array}
$$

Hence we can assume that the control flow graph G of $\mathcal{P}$ is reducible. Moreover we assume that

- the assignments of our TAC are of one of the forms
  - `x := y`,
  - `x := unop y` where `unop` is a unary arithmetic operation,
  - `x := y binop z` where `binop` is a binary arithmetic operation.
- and all other statements are conditional or unconditional jumps. (Hence we do not consider pointers nor function calls.)

Finally we assume that the alternatives and **while** loops are translated using the translation schemes of the previous chapter. This implies that whatever is E the control will flow to the same point after executing each of the statements.

- **if** E **then** S_1 **else** S_2
- **while** E **do** S

<u>Points.</u> Within a basic block B, a *point* denotes

- either the beginning of B (before its first statement)
- or the end of a statement.

Hence each basic block consisting of ℓ statements possesses $\ell + 1$ points. We will use the following notations.

- The first point of B is denoted by $first(B)$.
- The last point of B is denoted by $last(B)$.

PATH. A sequence of points $p_1, \ldots, p_n$ is a *path* if for every $i = 1, \ldots, n-1$

- either p_i and p_{i+1} are in the same basic bloc **and** p_i immediately precedes p_{i+1}.
- or p_i is the last point of a block B **and** p_{i+1} is the first point a block B' such that there is an edge from B to B' in G.

VARIABLE DEFINITION. A *definition* of a variable `x` is a statement that assigns (or may assign) `x`. Observations.

- Because of our assumptions regarding TAC a statement assign or does not assign a variable.
- In a more general theory, with procedure calls or pointers, a statement may assign a variable. In this case we would consider the notions of *unambiguous definitions* and *ambiguous definitions*.

A VARIABLE DEFINITION d REACHES A POINT p if there is a path from the point immediately following d to p such that d **is not modified** (*killed*) along this path.

- It is also frequent to say that a definition d is *active* at a point p instead of saying that d reaches p.
- Observe that some points may be reached by any definition.

```
if a = b then a := 2
else if a = b then a := 4
```

From now

- we **number** each TAC statement and
- we will refer to a variable definition by the number of the statement where this definition is made.

Region. A subgraph R of the control flow graph G is a *region* of G if there exists a unique node h of R which dominates all other nodes of R. Observations.

- This unique point h is called the *header* of the region.
- A loop is a special case of a region that is strongly connected.

Theorem 1 *The subgraph of the CFG corresponding to the translation of a statement S of our high level language is a region denoted by $region(S)$.*

Moreover, by introducing empty blocks, one may assume that for any statement S of our high level language control can flow to only one outside block when it leaves $region(S)$.

A statement region is a subgraph of the CFG of the form $region(S)$. Because of the previous definition and theorem, it makes sense to talk about the *first point* and the *last point* of a region, denoted respectively by $entry(S)$ and $exit(S)$.

Data-flow sets for statement regions. For each statement S of our high level program, we associate four sets to $region(S)$. Each of them

- is a subset of the variable definitions of the TAC program $\mathcal{P}$ and
- can be viewed as an attribute of the statement S.

These sets are defined as follows.

$in(S)$**:** A definition d (from anywhere in $\mathcal{P}$) belongs to $in(S)$ if d is active at $entry(S)$.

$out(S)$**:** A definition d (from anywhere in $\mathcal{P}$) belongs to $out(S)$ if d is active at $exit(S)$.

$gen(S)$**:** A definition d belongs to $gen(S)$ if d is made in S and there exists a path from to d to $exit(S)$ which uses points all inside S.

$kill(S)$**:** A definition d (from anywhere in $\mathcal{P}$) belongs to $kill(S)$ if **for every path** from $entry(S)$ to $exit(S)$ it is killed by definition d' with $d \neq d'$.

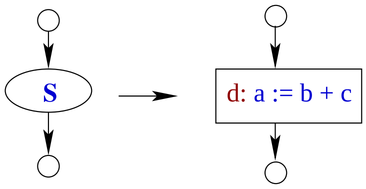

Figure 6: Data-flow equations for an assignment.

Observe that these definitions extend **naturally to basic blocks since a basic block can be viewed as a sequence of statements of the high level language.**

DATA-FLOW EQUATIONS. We give below equations relating the data-flow sets.

- It is important to realize that these equations are an approximation of what will really happen when the programs runs. Indeed some *killing* statements may in fact not kill like in

```
x := 0
y := y + x
```

- A central principle of code improving transformations is that we should make *conservative* decisions
  - if in doubt we should add a statement to a *generated set*
  - if in doubt, we should not add any definition to a *killed set*.

DATA-FLOW EQUATIONS FOR AN ASSIGNMENT.

$$\begin{aligned} gen(S) &= \{d\} \\ kill(S) &= D_a \setminus \{d\} \\ out(S) &= gen(S) \bigcup \, (in(S) \setminus kill(S)) \end{aligned} \tag{1}$$

where D_a denotes the set of all definitions of a in the TAC program $\mathcal{P}$. Figure 6 illustrates the fact that $gen(S)$, $kill(S)$, $out(S)$ can be computed from $in(S)$.

DATA-FLOW EQUATIONS FOR A SEQUENCE OF STATEMENTS.

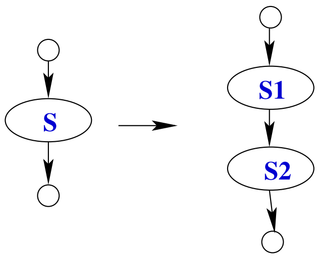

Figure 7: Data-flow equations for a sequence of statements.

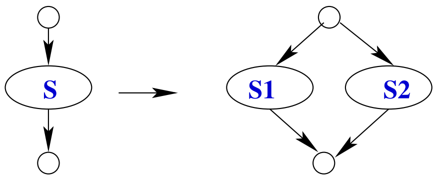

Figure 8: Data-flow equations for an alternative.

$$\begin{aligned}
gen(S) &= gen(S_2) \bigcup \; (gen(S_1) \setminus kill(S_2)) \\
kill(S) &= kill(S_2) \bigcup \; (kill(S_1) \setminus gen(S_2)) \\
in(S_1) &= in(S) \\
in(S_2) &= out(S_1) \\
out(S) &= out(S_2)
\end{aligned} \tag{2}$$

Figure 7 illustrates the fact that $gen(S)$, $kill(S)$, $out(S)$ are synthesized attributes whereas $in(S)$ is an inherited attribute.

Data-flow equations for an alternative.

$$\begin{aligned}
gen(S) &= gen(S_1) \bigcup gen(S_2) \\
kill(S) &= kill(S_1) \bigcap kill(S_2) \\
in(S_1) &= in(S) \\
in(S_2) &= in(S) \\
out(S) &= out(S_2) \bigcup out(S_1)
\end{aligned} \tag{3}$$

Data-flow equations for a **while** loop.

$$\begin{aligned}
gen(S) &= gen(S_1) \\
kill(S) &= kill(S_1) \\
in(S_1) &= in(S) \bigcup gen(S_1) \\
out(S) &= out(S_1)
\end{aligned} \tag{4}$$

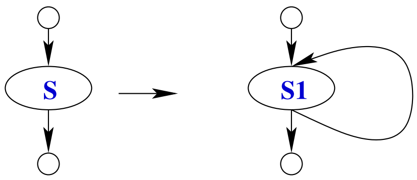

Figure 9: Data-flow equations for a **while** loop.

Observe that syntax-directed definition associated with the above rules is not L-attributed. Indeed, the inherited attribute $in(S_1)$ depends on the synthesized attribute $gen(S_1)$.

<u>Computing the data-flow sets.</u>

- The synthesized attributes $gen(S)$ and $kill(S)$ can be computed during a bottom-up parsing
- However the inherited attributes $in(S)$ cannot be computed by a depth-first traversal of the parse tree because of the rule

$$in(S_1) \ = \ in(S) \bigcup gen(S_1) \tag{5}$$

  as mentioned above.
- To compute the inherited attributes $in(S)$ we use a *completion algorithm* as for the FOLLOW sets or the *dominator* sets.
- This is done by Algorithm 4 which assumes that the sets $kill(B)$ and $gen(B)$ are known for every basic bloc B.

Algorithm 4

Input: $kill(B)$ and $gen(B)$ for every basic bloc B.

Output: $in(B)$ and $out(B)$ for every basic bloc B.

for each B **repeat**
  $out(B) := gen(B)$
while changes to any $out(B)$ occur **repeat**
  $in(B) := \bigcup_{B' \in \mathsf{pred}(B)} out(B')$
  $out(B) := gen(B) \bigcup (in(B) \setminus kill(B))$

Observations.

- Since each $\mid out(B) \mid$ is bounded (by the number of lines of the TAC program) and since there is a finite number of basic blocks, the algorithm has to stop.
- When it stops, the sets $in(B)$ **cannot** further increase since the sets $out(B)$ are themselves saturated.
- One can show (not really difficult) that this algorithm has a quadratic time complexity in the number of lines of the $\mathcal{P}$ program.
- This algorithm can be implemented efficiently by coding each data-flow set by an inetger (regarded as a bit vector) and using bit vector operation from the machine processor.
- Algorithm 5 detects loop invariant computations.

Algorithm 5

Input: A loop L (consisting of basic blocks for which the data-flow sets are known)

Output: The set of the TAC that computes the same value each time executed, from the time control enters the loop header until control next leaves L.

```
mark INVARIANT statemens with constant operands
mark INVARIANT statemens whose reaching definition are all outside L
while new statements are marked INVARIANT repeat
  mark INVARIANT statements with a unique reaching definition
      which is inside L and
      which is marked INVARIANT
```

Note that a loop invariant is not necessarily a TAC which can be moved into a pre-header. Indeed, in a given loop we may have two statemens with constant operands that define the same variable. For instance, `x := 2` and `x:= 5`.

5 Data-flow analysis: copy propagation.

Many optimizations and user code introduce copy statements of the form

`s:  x := y`

It is sometimes possible to eliminate such copy statements.

<u>Copy propagation conditions.</u> Assume that for a statement `u` where `x` is used

- statement `s` is the only definition of `x` reaching `u`
- on every path from `s` to `u` there are no assignments to `y`

Then we may substitute `y` for `x` in `u`.

Observe that

- we know how to check the first condition,
- but not the second one ...

We shall set up

- a new data-flow analysis together with
- new data-flow equations to solve this problem.

<u>Copy reaching a point.</u> Let `s:  x := y` be a copy statement and p be a point of the TAC program $\mathcal{P}$. We say that the copy `s:  x := y` reaches p if every path γ from the initial node of $\mathcal{P}$ to p

- contains `s:  x := y`
- and subsequent to the last occurrence of `s:  x := y` on γ there is no assignment to `y` or `x`.

This implies that the statement `s:  x := y` reaches the point p.

<u>Copy propagation data-flow sets.</u> For a basic block B

- $in_C(B)$ is the set of the copies (from anywhere in $\mathcal{P}$) reaching $entry(B)$.
- $out_C(B)$ is the set of the copies (from anywhere in $\mathcal{P}$) reaching $exit(B)$.

- $gen_C(B)$ is the set of the copies defined in B and reaching $exit(B)$.
- A copy statement `s: x := y` is killed in B if
  - `s: x := y` $\notin B$ and
  - `x` or `y` are assigned in B.
- $kill_C(B)$ is the set of the copy statements (from anywhere in $\mathcal{P}$) that are killed in within block B by a redefinition of `y` or `x`.

Observe that

- $in_C(B)$ can contain **only one** copy statement with a given `x` on the left.

Observe that for every basic block B

- the data-flow sets $gen_C(B)$ and $kill_C(B)$ can be computed easily,
- the definitions of $gen_C(B)$ and $kill_C(B)$ essentially deals with graph theory and will not detect copy statements that are never hit (= evaluated),
- so we may apply the following precautionary principles
  - we add a copy statement to $gen_C(B)$ only if we are sure that it is hit,
  - we add a copy statement to $kill_C(B)$ even if it is doubtful that it will be killed.

<u>DATA-FLOW EQUATIONS</u> For every basic block B the data-flow sets $in_C(B)$ and $out_C(B)$ satisfy the following equations

$$\begin{aligned} in_C(B) &= \bigcap_{B' \in \mathsf{pred}(B)} out_C(B'), \\ out_C(B) &= gen_C(B) \bigcup \left(in_C(B) \setminus kill_C(B)\right). \end{aligned} \tag{6}$$

<u>COMPUTING THE DATA-FLOW SETS FOR THE COPY PROPAGATION PROBLEM</u> Assume that the set C of all copy statements in $\mathcal{P}$ has been computed. Then, for each basis block B one can compute $gen_C(B)$ and $kill_C(B)$. Now set

$$in_C(B_1) = \emptyset \quad \text{and} \quad out_C(B_1) = gen_C(B_1) \tag{7}$$

where B_1 is the first block Then one can set up a completion algorithm initializing every for basic block $B \neq B_1$

$$in_C(B) \ = \ C \tag{8}$$

and using the *updating* equations

$$\begin{array}{rcl} out_C(B) & = & gen_C(B) \bigcup \ (in_C(B) \setminus kill_C(B)) \,. \\ in_C(B) & = & \bigcap_{\ B' \ \in \ \mathsf{pred}(B)} out_C(B'), \end{array} \tag{9}$$

6 Data-flow analysis: available expressions.

<u>In performing common subexpression elimination</u> we wish tofind in the program two expressions that will always yield the same value, such that one can be replaced by the other.

<u>An expression is available</u> `b op c` at a point p of the program if

- all routes (= pathes) from the start of the program to that point p must pass through the evaluation of `b op c`, and
- after the (last) evaluation of `b op c` there are **no redefinitions** of `b` or `c`.

This means that at point p we can replace `b op c` by the result of its last evaluation.

<u>Data-flow equations.</u>

- An expression is *generated* by a statement where it is evaluated. Similarly, an expression is *killed* by a statement that redefines `b` or `c`.
- Therefore, we can again think of *killing* and *generating* available expressions.
- The set $gen_E(B)$ of the expressions generated by a basic block B can be computed (or defined) as follows.
    1. Initially, we set $gen_E(B) = \emptyset$.
    2. Going through the statements of B in sequence,
        - for a statement `a := b op c` add `b op c` to $gen_E(B)$ and

- remove from $gen_E(B)$ any expression containing `a`.

- Let E be the set of all the expressions in the entire program $\mathcal{P}$. The set $kill_E(B)$ of the expressions killed by a basic block B can be computed (or defined) as follows.
  1. Initially, we set $kill_E(B) = \emptyset$.
  2. Going through the statements of B in sequence, for a statement `a := b op c` add to $kill_E(B)$ any expression in E containing `a`.
- Precautionary principles:
  - we only add an expression to $gen_E(B)$ if we are sure that it will be evaluated
  - if there is a possibility that the expression is killed, we add it to $kill_E(B)$.
- For a basic block B we define
  - $in_E(B)$ as the set of the expressions available at $entry(B)$,
  - $out_E(B)$ as the set of the expressions available at $exit(B)$.

<u>DATA-FLOW EQUATIONS.</u> For every basic block B the data-flow sets $in_E(B)$ and $out_E(B)$ satisfy the following equations

$$\begin{aligned} in_E(B) &= \bigcap_{B' \in \mathsf{pred}(B)} out_E(B'), \\ out_E(B) &= gen_E(B) \bigcup \left(in_E(B) \setminus kill_E(B)\right). \end{aligned} \tag{10}$$

<u>COMPUTING THE DATA-FLOW SETS.</u> We assume that

- the set E of all expressions in the entire program $\mathcal{P}$ has been computed,
- the data-flow sets $kill_E(B)$ and $gen_E(B)$ have been computed for every basic block B.

The iterative algorithm for determining available expressions is now basically the same as for copy statements.

1. For the first basic block B_1 we define

$$in_E(B_1) = \emptyset \quad \text{and} \quad out_E(B_1) = gen_E(B_1). \tag{11}$$

2. For each basic block $B \neq B_1$ we define

$$out_E(B) = E \setminus kill_E(B) \tag{12}$$

3. We update the data-flow sets $in_E(B)$ and $out_E(B)$ (for every basic block $B \neq B_1$) with the data-flow equations until there are no more changes in these sets.

7 Global Live Variable Analysis

We return to the analysis of live varaibles. Our earlier analysis and optimization were restricted to a basic block, We recall the following definition.

A VARIABLE IS *live* AT A POINT p of the program if it is referenced further in the program (which can decided by considering the control flow graph) otherwise it is said to be dead.

DATA-FLOW SETS. We assume that the control flow graph is reducible. For a basic block B

- $use_L(B)$ is the set of the variables which are used in B before they are (re-)dedefined in B. Thus, such variables rely on some earlier definitions (in the control flow graph).
- $def_L(B)$ is the set of the variables which are defined in B before they are used anywhere in the program. This can be computed by using a topogical sort of the control flow graph (after its back edges have been removed).
- $bottom_L(B)$ is the set of variables that are live at $exit(B)$.
- $top_L(B)$ is the set of variables that are live at $entry(B)$.

The safety principle requires that we should assume that variables remain live if there is any doubt.

COMPUTING THE DATA-FLOW SETS. Note that the information to be computed is opposite to the flow of control

The following iterative algorithm propagates liveness information backwards around the control flow graph.

- For each basic block B we set

$$top_L(B) = use_L(B) \ \text{ and } \ bottom_L(B) = \emptyset \tag{13}$$

- For each basic block B

$$\begin{array}{rcl} bottom_L(B) & = & \bigcup_{B' \in \mathsf{pred}(B)} top_L(B') \\ top_L(B) & = & (bottom_L(B) \setminus def_L(B)) \bigcup use_L(B) \end{array} \tag{14}$$

- Repeat previous step until none of the sets change.

8 Global Copy Propagation and Dead Code Elimination

<u>Key idea.</u> With the data-flow sets $in_R(B)$ and $in_C(B)$ computed in the previous sections we can consider global copy propagation and dead-code elimination across basic blocks.

- Consider abasic block B and a statement S of B which references a variable `b`.
- If a copy statement `b := c` reaches statement S, then it belongs to $in_R(B)$ (where $in_R(B)$ denotes the set of the definitions that reach B).
- If `b := c` also belongs to $in_C(B)$ then we know that this is the **only definition** of `b` reaching B.
- In addition, if there is no other definition of `b` or `c` in block B before S (which is easily checked) then we can replace `b` by `c` in statement S.
- Finally, if all references to `b` that are reached by `b := c` can be replaced by `c`, then the assigned `b := c` becomes *dead code* and can be removed.

<u>Algorithm.</u> For each copy statement `S: b := c` in the program do

1. Find all uses of `b` that are reached by `S'`. That is a references to `b` in a statement `S` of block B such that no redefinition of `b` or `c` appear between $entry(B)$ and `S'`.
2. For each such use of `b` where, furthermore, $S \in in_C(B)$, replace `b` by `c`.

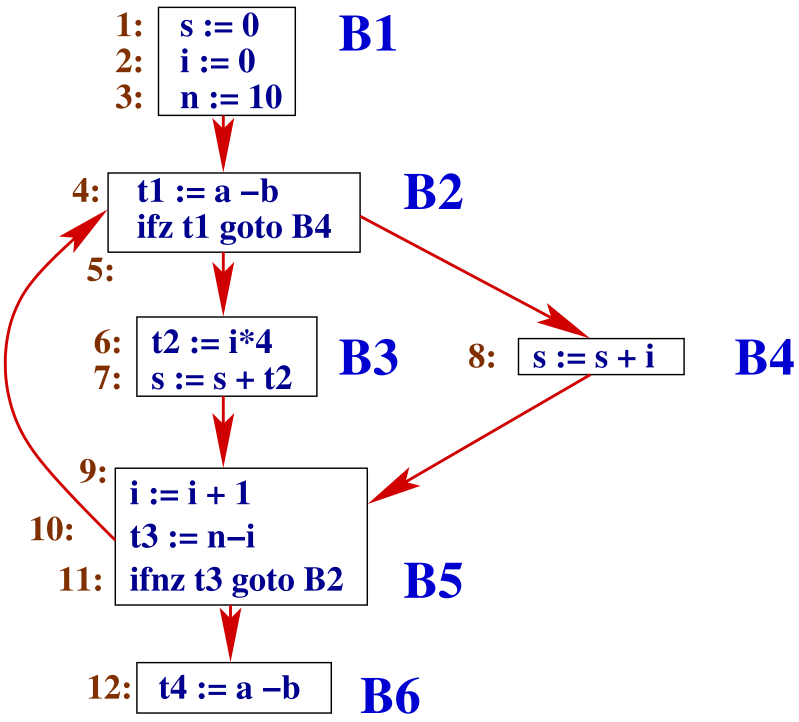

Figure 10: A control flow graph.

3. if we can do this replacement in all the above cases, then remove the copy statement `S: b := c`.

<u>EXERCISE.</u> Apply the previous algorithm to Figure 10.

- Copy statement `1` belongs to $in_R(B_3)$ and $in_R(B_4)$ and reaches references of `s` there. However, `1` does not belong to neither $in_C(B_3)$ nor $in_C(B_4)$. So, the replacement cannot be done.

- Copy statement 2 belongs to $in_R(B_3)$, $in_R(B_4)$ and $in_R(B_5)$ and reaches the reference of `i` there. But `2` does not belong to any of $in_C(B_3)$, $in_C(B_4)$ or $in_C(B_5)$.

- Copy statement `3` belongs to $in_R(B_5)$ and $in_C(B_5)$ and reaches the reference of `n` in statement `10`. So, the replacement can be done.

- After these modifications we have toupdate the dataflow information for the modified flow graph.

9 Global Recycling of Available Expressions

<u>Key idea.</u>

- We look for statements `a:=b op c` where
  - `b op c` is in the list of available expressions at the start of the block where `a:=b op c` appears, and
  - `b,c` have not been defined earlier in the block.
- We look for an earlier evaluation of a statement `d:=b op c`
  - However, we cannot just use the value of `d` because the available expression may actually have come along a completely different route.
  - This problem can be solved by storing the value of each occurrence of theavailable expression (in different parts of the program) into the same temporary variable `t` and then use the value of the temporary.
- This process generates new copy statements, but these can usually be eliminated using further copy propagation and deadcode elimination.

<u>Algorithm.</u>

1. Find all statements `S: a := b op c` occurring in a basic block B such that
   - `b op c` is available at $Entry(B)$, and
   - B contains no earlier definition for `b` or `c`.
2. For each statement `S` found in the previous step,
   (a) allocate a temporary variable `t`, and
   (b) from block B search backwards along all possible paths the previous statement `d := b op c`, and immediately after it insert the statement `t := d`.
   (c) Replace the statement `S` by `a := t`.

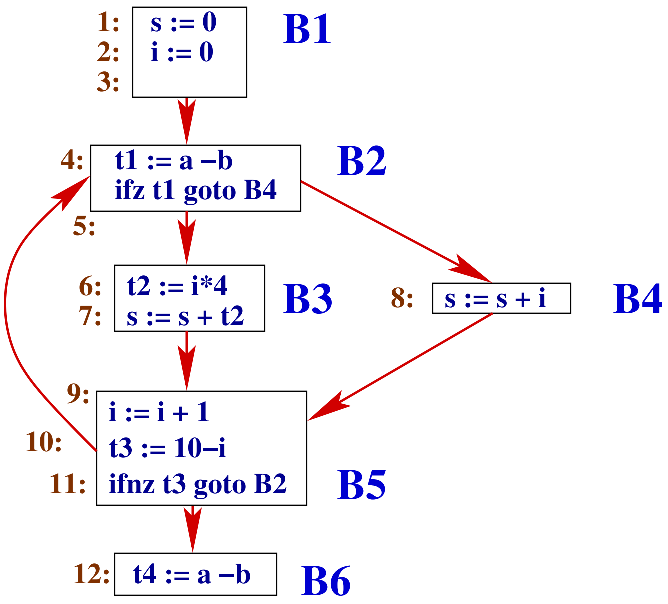

Figure 11: A control flow graph.

<u>Exercise.</u> Apply the previous algorithm to Figure 11.

- The statement `t4 := a-b` is found in stage (i) of the algorithm.
- Searching backwards, the unique previous evaluation in in statment `4`.
- So, after Statement `4`, we insert `t5 := t1` and statement `12` is replaced by `t4 := t5`.
- After new copy propagation and dead code elimination, line `12` is eliminated.

After applying these transformations, we obtain the control flow graph of Figure 12. Obviously, one would like to move out the first two lines of the basic block B_2. This will be achieved using *Invariant Code Motion* to be described now.

10 Invariant Code Motion

<u>The general method</u> can be described as follows.

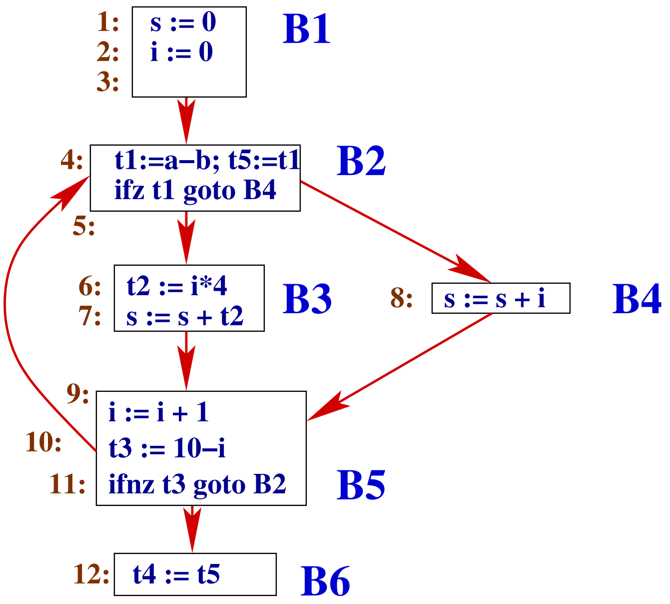

Figure 12: A control flow graph.

1. A statement `a : = b op c` in block B of a loop is marked as invariant if all definitions of `b` and `c` that are in $in_R(B)$ are outside the loop, or are invariant statements in the loop. Observe that
   - this condition is always satisfied if `b,c` are constants.
   - if `a := b op c` is invariant and if `a` and `c` do not have other definitions in the loop, then `a := a op c` is invariant too.
2. Repeat previous step until no more invariant statements are found.

Remarks.

- We still need to be careful in choosing which invariant statements can be moved to the pre-header.
- The pre-header is always executed whereas the same is not true for all statements in the loop.
- The invariant statement `a : = b op c` moved to the pre-header need tosatisfy the following requirements.

1. The statement is always executed in the loop. That is, it dominates all the blocks. (Otherwise, different blocks in the loop may need different values of `a`.)
2. The statement has to be the only definition of `a` belonging to $in_R(B)$ for any block B in the loop. (`a := 2` and `a := 5` can be both invariant in the same loop but none can be moved out.)

11 Induction Variables

<u>A BASIC INDUCTION VARIABLE</u> of a loop is a variable `b` defined in a statement of the form `b := b + n` or `b := b - n` where

- `n` is a constant, and
- `b` has no other definitions in the loop.

Knowing the induction variables is very useful. For instance, they may allow us to replace multiplication by addition (often by a constant)

<u>REPLACING MULTIPLICATIONS BY ADDITIONS</u> can be performed by the following algorithms.

- Find every variable `c` in the loop
  - with a single definition
  - of the form `c := b*m`
  - where `m` is a constant and `b` is a basic induction variable.
- After each occurrence of `b:= b + n` (or `b := b - n`) add a statement `t := t +p`, where `p` is theconstant `n*m`
- Replace statement `c := b*m` by `c := t`.
- Add a statement `t := b*m` into the pre-header (initialization) after possible definition of `b`.

12 A complete exercise

Consider the following three-address code program.

```
L0  n := 10
    s := 0
    i := 0
L1  if i > n goto L6
    j := i
    m := n
    p := m
    f := 1
L2  if p = 1 goto L3
    f := f * p
    p := p - 1
    goto L2
L3  a := f
    p := m - j
    f := 1
L4  if p = 1 goto L5
    f := f * p
    p := p - 1
    goto L4
L5  b := f
    c := a quo b
    s := s + c
    i := i - 1
    goto L1
L6
```

1. Give the flow graph of this program.

2. Identify the back edges and the loops.

3. For each basic block B compute the data-flow sets $gen(B)$ and $kill(B)$ of the generated and killed variable definitions.

4. Deduce from this study a significant optimization for the above code.