Control Flow Graphs

From Matthew S. Hetch.

Flow Analysis of Computer Programs.

North Holland. 1977.

Control Flow Graphs

We will now discuss flow graphs. These are used for global optimizations (as opposed to optimizations local to basic block).

The nodes of a flow graph are basic blocks.

There is an initial node, s, in every flow graph. The node s corresponds to the basic block whose leader is the first statement. An *artificial* single entry node can be created if there are multiple entries (add an arc from the artifical entry to every entry node)

There is an arc from node n_1 to node n_2 if the basic blocks associated to n_2, B_2, can immediately follow at execution time the basic block associated with n_1, B_1.

That is, there should be an arc from node n_1 to node n_2 if either

- There is a conditional or inconditional jump from B_1 to (the first statement of) B_2, or
- B_2 immediately follows B_1 in the original order of the program and B_1 does not end in an inconditional jump.

Definition. A flow graph is a triple $G=(N,A,s)$, where (N,A) is a (finite) directed graph, and there is a path from the intial node, $s \in N$, to every node.

Any node unreachable from s can be deleted without loss of generality.

An exit node in a flow graph has no successors.

Flow graphs are usually *sparse*. That is, $|A| = O(|N|)$. In fact, if only binary branching is allowed $|A| \leq 2|N|$.

Example of Control Flow Graph

A possible translation of the source code:

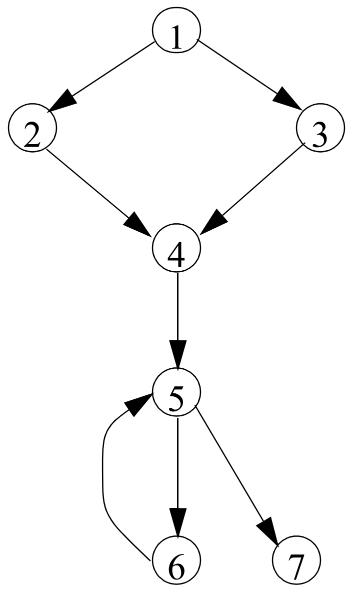
```
x=a+1
b=x+3
if b<0  then
            a=3
            b=a+5
        else
            t=b
            y=a+2
end if
for i=1 to n do
    m=m+i
    a(m)=b+i
end for
```

into low-level code is:

```
    x=a+1
    b=x+3                      1
    if b>=o then go to 3
    ------------------------------
    a=3
    b=a+5                      2
    go to 4
    ------------------------------
3   t=b                        3
    y=a+2
    ------------------------------
4   i=1                        4
    ------------------------------
2   if i>n go to 7             5
    ------------------------------
    m=m+i
    a(m)=b+i                   6
    i=i+1
    go to 2
    ------------------------------
7
```

Where the basic blocks are framed and numbered.

The control flow graph has the following form:



Relations

A relation **R** from set A to set B is any subset of A×B: $R \subseteq A \times B$. We use a**R**b to denote $(a,b) \in \mathbf{R}$ and a~~**R**~~b to denote $(a,b) \notin \mathbf{R}$

R is

- reflexive iff $(\forall x \in N)[x\mathbf{R}x]$
- antisymmetric iff $(\forall x,y \in N)[x\mathbf{R}y \wedge y\mathbf{R}x \rightarrow x=y]$
- asymmetric iff $(\forall x,y \in N)[x\mathbf{R}y \rightarrow y \not\mathbf{R} x]$
- transitive iff $(\forall x,y,z \in N)[x\mathbf{R}y \wedge y\mathbf{R}z \rightarrow x\mathbf{R}z]$

Dominance in Control Flow Graphs

Definition 1. A node x in a flow graph G dominates node y (could be the same as x) iff every path in G from s to y contains x.

DOM(x) denotes the set of dominators of x.

Definition 2. x properly dominates y if x dominates y and $x \neq y$.

Definition 3. x directly dominates y if x properly dominates y and any other node z that properly dominates y also properly dominates x.

Lemma 1. **DOM**(s) = $\{s\}$.

Lemma 2. The dominance relation of a flow graph G is a partial ordering. That is dominance is reflexive, antisymmetric, and transitive relation.

Proof:

It is reflexive because for any node x, x dominates x.

It is antisymmetric. If x dominates y, then y cannot dominate x. Assume otherwise. Then in every path from s to y, x has to appear before any occurrence of y (because x dominates y) and y before any occurrence of x (because y dominates x). Not possible.

It is transitive. If x dominates y and y dominates z then in every path from s to z, y has to appear before z and x before y.

Lemma 3. The initial node s of a flow graph G dominates all nodes of G.

Lemma 4. The dominators of a node form a chain.

Proof. Assume that two nodes, *x* and *y,* dominate node *z*. Then there is a path from *s* to *z* where both *x* and *y* appear only once (if not "cut and paste" the path). Assume that *x* appears first. Then if *x* does not dominate *y* there is a path from *s* to *y* that does not include *x* contradicting the assumption that *x* dominates *z*.

Lemma 5. Every node except *s* has a unique direct dominator.

Proof. The dominators form a chain. The last node in the chain is the direct dominator (which clearly always exist and is unique).

Lemma 6. A graph of dominators can be created from the nodes *N* of *G.* There is an arc from *x* to *y* iff *x* directly dominates *y.* This graph is a tree.

Algorithm DOM: Finding Dominators in A Flow Graph

*Input:*A Flow Graph $G = (N,A.s)$.

*Output:*The sets DOM(x) for each $x \varepsilon N$.

```
DOM($s$) := {$s$}
**forall** $n$ in $N$ - {$s$} **do** DOM($n$) := $N$ **od**
**while** changes to any DOM($n$) occur **do**
        **forall** $n$ in $N$ - {$s$} **do**
                DOM($n$) := $\{n\} \cup \bigcap_{p \text{ a preceсessor of } n} \text{DOM}(p)$
        **od**
**od**
```

Intervals

The notion of loop in the flow graph is introduced through the concept of interval.

Notice that notions such as cycle and strongly connected component are not appropriate. The former is too fine and the latter too coarse. With cycles loops are not necessarily properly nested or disjoint; and with strongly connected components, there is no nesting.

Definition 4. The interval with node *h* as header, denoted I(h), is the subset of nodes of *G* obtained as follows:

```
I($h$) := {$h$}
while ∃ node $m$ such that $m \notin I(h)$ and $m \neq s$ and all arcs
                              entering $m$ leave nodes in I($h$) do
    I($h$) := I($h$) + {$m$}
od
```

I(*h*) is unique and does not depend on the order of selection in the while loop.

Lemma 7. The subgraph generated by I(*h*) is a flow graph.

Lemma 8.

(a) Every arc entering a node of the interval I(*h*) from the outside enters the header *h*.

(b) *h* dominates every node in I(*h*)

(c) every cycle in I(*h*) includes *h*

Proof.

(a) Consider a node $m \neq h$ that is also an entry node of I(h). Then m could not have been added to I(h)

(b) Consider a node m in I(h) not dominated by h. Then m could not have been added to I(h).

(c) Suppose there is a cycle in I(h) that does not include h. Then no node in the cycle could have been added to I(h), because before any such node could be added the preceeding node in the cycle would have to be added.

Algorithm INT: Partitioning a Flow Graph Into Intervals

Input:

1. A Flow Graph Represented by successor lists.

Output:

A set L of disjoint intervals whose union is *N*.

Intermediate:

A set of potential header nodes, H.

```
H := {$s$}
L:= $\varnothing$
**while** H $\neq \varnothing$ **do**
        $h$ from H
        Compute I($h$)
        L := L $\cup$ {I($h$)}
        H := H $\cup$ { nodes with predecessors in I($h$) but that are
                        not in H or in one of the intervals in L}
**od**
```

Theorem 1. The preceeding algorithm partitions the set of nodes of G.

Proof. If a node is added to an interval it is not added again. Also, since all nodes are accessible from *s,* every node is added to H or to one interval.

Reducibility

Definition 5. If G is a flow graph, then the derived flow graph of G, I(G) is:

(a) The nodes of I(G) are the intervals of G

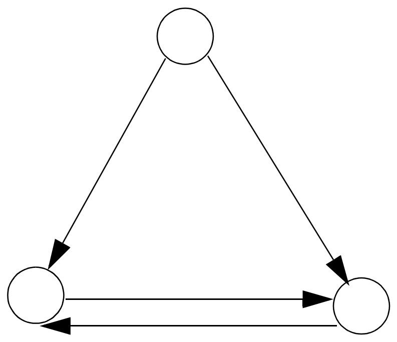
(b) The initial node of I(G) is I(s)

(c) There is an arc from node I(h) to I(k), $k \neq h$, in I(G) if there is any arc from a node in I(h) to node k in G.

Definition 6. The sequence $G=G_0, G_1, \ldots, G_k$ is called the derived sequence for G iff $G_{i+1} = I(G_i)$ for $0 \leq i < k$, $G_{k-1} \neq G_k$, $I(G_k) = G_k$. G_k is called the limit flow graph of G.

Definition 7. A flow graph is *reducible* iff its limit flow graph is a single node with no arc. Otherwise it is called *irreducible*.

Example of irreducible fow graph:



Definition 8. Interval order for a reducible flow graph is defined recursively as follows:

1. If I(G) is a single node, then an interval order is an order in which nodes may be added to the lone interval G.
2. If G is reducible and I(G) is not a single node, then an interval order is formed by:

   (a) Find an interval order for I(G)

   (b) In the order of (a) substitute for each node of I(G) the nodes of G that make up the corresponding interval, themselves in interval order.

Lemma 9. Interval order topsorts the dominance relation of a reducible flow graph G.

Proof Let G=G_0, G_1, ...,G_k be the derived sequence of G.

We show by induction that if a node *x* properly dominates *y*, then *x* precedes *y* in any interval order of the nodes of G_i .

For i=k it is vacuosly true because there is only one node in G_k.

Assume true for G_{i+1} and consider two nodes *x* and *y* in G_i.

If *x* and *y* are in the same interval of G_i, then clearly *x* precedes *y* in any interval order (*y* will not be added to the interval until *x* has been added).

If *x* and *y* are in different intervals of G_i, then they will be in different nodes, say *X* and *Y*, of G_{i+1}. *X* has to dominate *Y* otherwise there would be in G_i a path to *y* that does not include *x* (contradicting the assumtion that *x* dominates *y*). By inductive hypothesis *X* precedes *Y* in any interval order of G_{i+1}.

Depth-First Spanning Trees

Definition 9. A depth-first spanning tree of a flow graph G is an ordered spanning tree grown by the DFS algorithm.

Algorithm DFS: Depth First Search computation of spanning tree

Input: A Flow Graph Represented by successor lists. Nodes are numbered from 1 to n.

Output: A Depth First Spanning tree, T, and a numbering of the nodes indicating the reverse of the order in which each node was visited by the algorithm.

```
DFS($x$):
    Mark $x$ “visited”
    **while** SUC($x$) $\neq \emptyset$ **do**
        $y$ from SUC($x$)   /* SUC(x) is an ordered list
                               (left-to-right) of nodes    */
        **if** $y$ is marked “unvisited” **then**
            add (x,y) to T
            DFS(y)
        **fi**
    **od**
    order(x) = i
    i=i-1
/**** Main Program Follows ****/
T=$\emptyset$
mark all nodes of G as “unvisited”
i=number of nodes of G
DFS($s$)
```

Definition 10. The arcs in G that are not in its depth first spanning tree fall into three categories:

(a) Arcs that go from ancestors to descendants are called *forward arcs* (advancing arcs in AHU).

(b) From descendants to ancestors or from a node to itself are called *back arcs* (retreating arcs in AHU) .

(c) The other arcs are called *cross arcs*.

Observations:

1. Arc (x,y) is a back arc iff order(x) $\geq$ order(y).
2. Cross arcs go from right to left
3. Every cycle of G contains at least one back arc.

Proof of Observation 1.

Want to prove that $x \rightarrow y$ is a back edge $\Leftrightarrow$ order(x) $\geq$ order(y)

$\Rightarrow$ x is a desdendant of y is the DFST. Therefore DFS(x) terminates before DFS(y) and, as a result, order(x) $\geq$ order(y)

$\Leftarrow$order(x) $\geq$ order(y) then we have to consequences:

1. either x=y or DFS(x) terminates before DFS(y).

2. DFS(y) must have begun before DFS(x), otherwise, the arc $x \rightarrow y$ would have been included in the DFST.

DFS(x) is active during a subinterval of the time that DFS(y) is active. Therefore, y is an ancestor of x.

Graph Transformations T1 and T2

We now introduce two graph transformations T1 and T2. Reducibility by successive application of these two transformations is equivalent to reducibility by intervals.

The transformation T1 removes self arcs (i.e. arcs of the form (x,x)). The transformation T2 replaces two nodes x and y connected by an arc, and x only predecessor of y, with a single node z. Predecessors of x become predecessors of z and successors of x or y become successors of z. (x comsumes y).

We denote the application of one of these transformations on a graph G as either $G \Rightarrow_{Ti} G'$ with i = 1, 2 or $G \Rightarrow G'$.

Theorem 2.

$\Rightarrow$ can only be applied a finite number of times.

$\Rightarrow^\circ$ is a function. That is $P \Rightarrow^\circ Q$ and $P \Rightarrow^\circ R$ implies $Q = R$.

Note: $\Rightarrow^\circ$ is the completion of $\Rightarrow$. That is, $P \Rightarrow^\circ Q$ means that there is a sequence of graphs $G_1, G_2, \ldots, G_k$ such that $P = G_1 \Rightarrow G_2 \Rightarrow \ldots \Rightarrow G_k = Q$, and there is no $R \neq Q$such that $Q \Rightarrow R$.

Lemma 10. If G is a flow graph, then G $\Rightarrow^*$ I(G).

($\Rightarrow^*$ is the reflexive and transitive clousure of $\Rightarrow$. That is P $\Rightarrow^*$ Q means that either P=Q or there are G_1, G_2, ..., G_k such that P = $G_1 \Rightarrow G_2 \Rightarrow ... \Rightarrow G_k$ = Q)

Proof It is sufficient to show that each interval of G can be transformed using the T1 and T2 transformations.

For each interval

1. Delete self loops from header using T1
2. Apply T2 in the order in which the nodes are included in the interval. If a self loop is generated, remove it applying T1.

Corollary 1. If a flow graph G is reducible, then G $\Rightarrow^*$ 0 (the trivial graph)

Proof Apply Lemma 10 iteratively.

Theorem 3. If G $\Rightarrow^\circ$ 0, then G is reducible.

Proof Assume G $\Rightarrow^\circ$ 0, but I°(G) = G' $\neq$ 0.

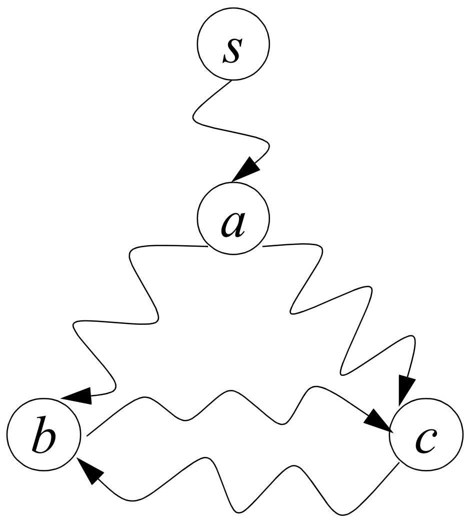
Clearly G $\Rightarrow^*$ G' by Lemma 10.

Also, G' $\Rightarrow^\circ$ 0. The reason is that if G' $\Rightarrow^\circ$ G" then G $\Rightarrow^\circ$ G" Now, because $\Rightarrow^\circ$ is a function and G$\Rightarrow^\circ$ 0, we have G" = 0.

We assume that I(G') = G'. But since G' $\Rightarrow^\circ$ 0, either T1 or T2 can be applied to G'. If T1 can be applied is because there is a self loop that is also removed by the I transformation. Therefore I(G') $\neq$ G'.

If T2 can be applied to (x,y), then y should be in I(x) and therefore I(G') $\neq$ G'.

The (*) graph:



The lines represent disjoint paths, and nodes *s* and *a* could be the same.

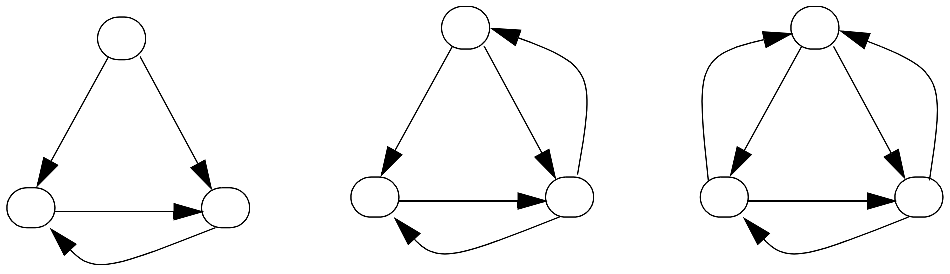
Lemma 11. The absence of a subgraph (*) in a flow graph is preserved by the application of T1 and T2.

Proof T1 and T2 do not create new paths or make two paths disjoint. Therefore, their application cannot create a (*) subgraph.

Theorem 4. If a flow graph is irreducible then it contains a (*) subgraph.

Proof By induction

BASIS: (n=3) For three nodes the graph is irreducible



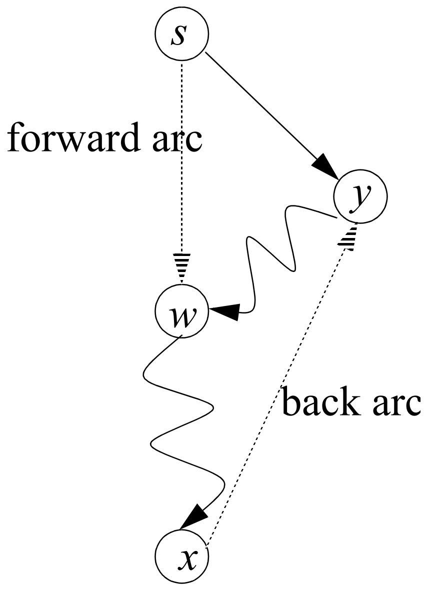
INDUCTIVE STEP: (n>3) Now assume that the theorem is true for a graph with n-1 or fewer nodes. Consider a graph G with n nodes. By the previous Lemma we may assume that neither T1 nor T2 is applicable to G (otherwise transform G⇒°G' and if G'can be shown to include (*), then we will also have shown that G has (*)).

Let T be the the Depth First Spanning Tree of G.

Let *y* be the rightmost child of the root of T. At least two arcs enter *y* (otherwise T2 could be applied). One of them is a back arc (*x*,*y*) [cannot be forward arc because the only predecessor is the root. Cannot be a cross arc because they run from right to left]

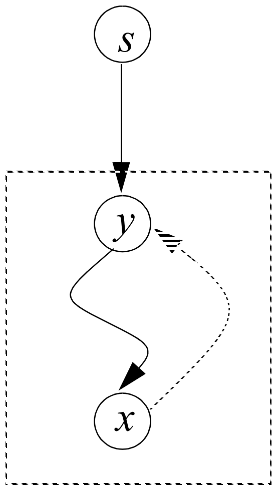
Two cases

1. *y* does not dominate *x*. Then there is a path from *s* to *x*



Whether w = x or not we have the subgraph (*).

2. *y* dominates *x*. Then we have the ireducible flow graph in the dotted



square which includes all nodes (other than s) from where x can be reached (recall that T1 and T2 cannot be applied to the original graph) with n-1 or fewer nodes. By induction it contains a (*) subgraph.

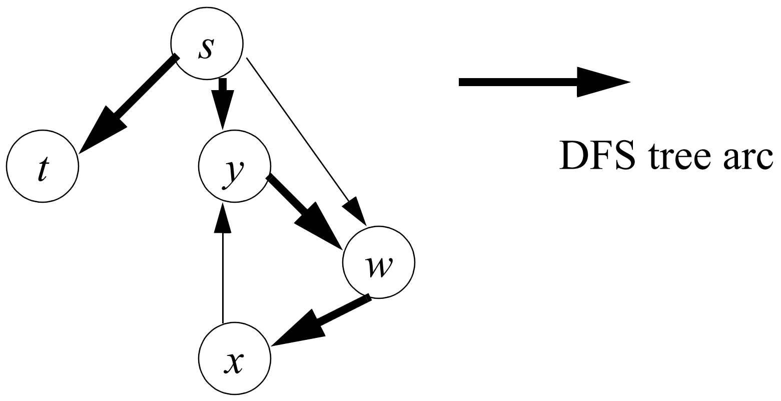
Corollary If G is a nontrivial limit flow graph, then it has a (*) subgraph in which *s*=*a*, and the paths from *a* to *b* and from *a* to *c* are each a single arc.

Question:

Is case 1 possible? In other words, wouldn't (*s*,*w*) be a DFST arc?

Answer:

Not necessarily:



Here, *w* is the rightmost child of *s* in G, but not a child of *s* in the DFST.

Theorem 5. If a flow graph contains a (*) subgraph then it is irreducible

Proof by induction.

BASIS (n=3): Same as Theorem 4.

INDUCTIVE STEP: Assume that hypothesis is true for n-1 nodes. Assume hypthesis false for n nodes. That is there is a graph G with n nodes that contains a (*) subgraph and is reducible.

T1 and T2 can be applied to G to get the trivial graph. The (*) subgraph will not be removed when T2 is applied. But now we have a n-1 node subgraph that is reducible and contains a (*) subgraph. A contradiction.

Regions and Other Concepts

Definition 11. Let G=(N,A,s) be a flow graph, let $N_1 \subseteq N$, let $A_1 \subseteq A$, and let h be in N_1. R=(N_1,A_1,h) is called a region of G with header h iff in every path (x_1, ..., x_k), where $x_1=s$ and x_k is in N_1, there is some i ≤k such that

(a) $x_i = h$

(b) x_{i+1}, ..., x_k are in N_1

(c) (x_i,x_{i+1}), (x_{i+1},x_{i+2}), ..., (x_{k-1}, x_k) are in A_1.

That is access to every node in the region is through the header only.

Lemma 12. A region of the flow graph is a subflowgraph

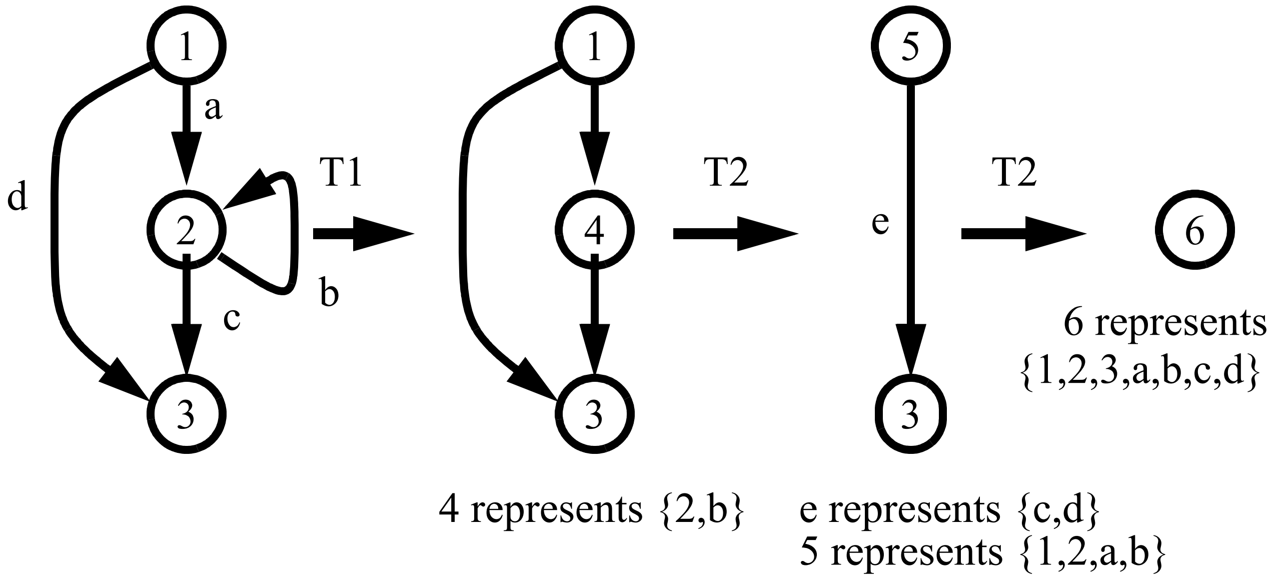
Lemma 13. The header of a region dominates all nodes in the region.

Definition 12. We say that each node and arc in the original flow graph represents itself.

If T1 is applied to node w with arc (w,w), then the resulting node represents what node w and arc (w,w) represented.

If T2 is applied to x and y with arc (x,y) eliminated, then the resulting node z represents what x, y and (x,y) represented. In addition, if two arcs (x,u) and (y,u) are replaced by a single arc (z,u), the (z,u) represents what (x,u) and (y,u) represented.

Example:



Lemma 14. In a flow graph, if region R results from region R’ consuming region R”, then the header h of R’ dominates all nodes in R”.

Theorem 6. As we reduce a flow graph G by T1 and T2, at all times the following conditions are true:

1. A node represents a region of G.
2. An edge from *x* to *y* represents a set of edges. Each such edge is from some node in the region represented by *x* to the header of the region represented by *y*.
3. Each node and edge of G is represented by exactly one node or edge of the current graph.

Proof The theorem holds trivially for G itself. Every node is a region by itself, and every edge represents only itself.

Whenever T1 is applied, we add a self arc to a node representing a region. Adding the self arc does not change the fact that the node is a region.

Assume now that T2 is applied to consume node *y* by node *x*. Let *x* and *y* represent regions X and Y respectively. Also, let A be the set of arcs represented by (*x*,*y*). We claim that X, Y and A together form a region whose header is the header of X. All we need is to prove that the header of X dominates every node in Y. But this is true because T2 was applied and therefore all arcs entering Y come from X.

Definition 13. A parse π of a reducible flow graph *G*=(*N,A,s*) is a sequence of objects of the form (T1,*u,v,S*) or (T2,*u,v,w,S*), where *u, v* and *w* are nodes and *S* is a set of arcs. We define the parse of a reducible flow graph recursively as follows:

1. The trivial flow graph has only the empty sequence as its parse.
2. If *G'* (which may not be the origianl flow graph in a seqence of reductions) is reduced to *G"* by an application of T1 to node *u*, and the resulting node is named *v* in *G"*, then (T1*,u,v,S*) followed by a parse of *G"* is a parse of *G'*, where *S* is the set of arcs represented by the arc (*u,u*) eliminated from *G'*.
3. If *G'* is reduced to *G"* by an application of T2 to nodes *u* and *v* (with *u* consuming *v*), and the resulting node is called *w*, then (T2,*u,v,w,S*) followed by a parse of *G"* is a parse of G', where *S* is the set of arcs represented by the arc (*u,v*) in *G'*.
4. In both (2) and (3) above, "representation in *G'* carries over to *G"*. That is, whatever an object represents in *G'* is also represented by that object in *G"*, except for those changes in representation caused by the particular transformation (T1 or T2) currently being applied.

Example: The parse of the previous flow graph is:

(T1,2,4,{b}) (T2,1,4,5,{a}) (T2,5,3,6,{c,d})

Definition 14. Let $G=(N,A,s)$ be a reducible flow graph and let π be a parse of G. We say that an arc in A is a *back* arc with respect to π if it appears in set S of an object (T1,u,v,S) of π and a *forward* arc (not to be confused with the forward arcs of a DFST. This is the forward arc of Definition 7') with respect to π otherwise. Let $B(G)$ be the set of arcs in A that are back arcs in every parse of G.

Definition 15. A *DAG of a flow graph* $G=(N,A,s)$ is an acyclic flow graph $D=(N,A',s)$ such that A' is a subset of A and for any arc e in $A-A'$, $(N,A'\cup\{e\},s)$ is not a *DAG*. That is, D is a maximal acyclic subflowgraph.

Theorem 7. Let G=(N,A,s) be a RFG and let π be a parse of G. Arc (x,y) is a back arc iff y dominates x.

Theorem 8. A flow graph is reducible iff its DAG is unique.

Corollary The DAG of a reducible flowgraph is any DFST of G plus its forward and cross arcs. Since this DAG is unique, the arcs of the DFST, its cross arcs and forward arcs together must be the same as the forward arcs of a parse of G.

Corollary. The back arcs of a parse of a reducible flow graph are exactly the back arcs of any DFST for G.

Corollary = **Definition 7'**. A flow graph is reducible iff we can partition the edges into two disjoint groups, often called the *forward* edges (not to be confused with the forward edges in a DFST) and the back edges, with the following two properties:

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

Node Splitting

How can an irreducible flow graph be transformed to an equivalent reducible flow graph?

First, let us assume that the nodes of a flow graph have (not necessarily distinct) labels. If P=(x_1, ..., x_k) is a path in a flow graph, then we define labels(P) to be the string of labels of these nodes: (label(x_1),..., label(x_k)).

We say that two flow graphs G_1 and G_2 are equivalent iff, for each path P in G_1, there is a path Q in G_2 such that labels(P)=labels(Q) and conversely.

Let G=(N,A,*s*) be a flow graph. Let *x* ($x \neq s$) be a node with no self loop, predecessors w_1, ..., w_p ($p \geq 2$), and successors y_1, ..., y_t ($t \geq 0$). We define G split x to be the flow graph resulting from the followin gprocedure:

1. Delete the arcs from the w_is to x and those from *x* to the y_is.
2. Add p copies of *x* (x_1, ..., x_p) with label(x_i) = label(*x*). Add arcs (w_i,x_i) and arcs from every x_i to all y_js.

Theorem 9. Let S denote the splitting of a node. Any flow graph can be transformed into the trivial flow graph by a transformation represented by the regular expression T°(ST°)*. That is, first apply T°, then apply S followed by T° zero or more times.

An Algorithm for Structured Flow Graphs

Objective: To generate "structured" code from an arbitrary flow graph. The source program may include `goto`s and the flow graph may be irreducible.

Restriction: Code duplication is not allowed.

Target Language:

1. Straight-line code
2. `stop`
3. `go to` L
4. `if (p) then` $\{S_1\}$ `else` $\{S_1\}$
5. `repeat` $\{S\}$
6. `break`$\{i\}$
7. `next`$\{i\}$

Some requirements:

1. repeat statements must reflect iterations in the program.

An example of undesirable outcome:

```
      repeat
              {   s=1
                  stop
              }
```

Another undesirable result:

```
      repeat
              {if (p) then
                      {   <code segment>
                          stop
                      }
                      else
                      {   x=f(x)
                      }
              }
```

A better translation of this last segment:

```
      repeat
              {   if (p) then {break(1)}
                          else {x=f(x) }
              }
      <code segment>
      stop
```

2. Each `if then else` should reflect branching and merging of flow of control in the program.
   - A `go to` statement should not jump into the middle of a `then` or `else` clause.
   - A statement should appear ithin a clause if it can be reached only from the clause and is within the innermost `repeat` containing the clause.

Steps of the algorithm to structure a flow graph

1. Build DFST.
   Generate a list L ordered by DFST number.
   Back arcs will go “up” in L and forward and cross arcs will go “down”
2. Insert repeats at the head of back arcs.
3. Generate code without `goto`s, `break`s, or `next`s.
4. Insert `goto`s (all will go “down” because `goto`s will only be generated for forward and cross arcs), `break`s and `next`s.

HEAD(), FOLLOW(), and REACH()

A `repeat` node **p** is the head of all loops and cycles which include **p** but no nodes preceding **p** in the list **L**.

Let **HEAD(q)** be the `repeat` node immediately enclosing **q**.

If there are no enclosing loops, **HEAD(q)** is undefined. If **HEAD(q)** and **HEAD(p)** are both undefined then **HEAD(q)=HEAD(p).**

FOLLOW(p):

- **p** is `if`:
  {**q** | **q** is entered by two or more forward or cross arcs, **p=IDOM(q),** and **HEAD(p)=HEAD(q)**}
- **p** is `repeat`: {**q** | **IDOM(q)** is nested within **p** and **HEAD(q)=HEAD(p)**}
- **p** is slc: { **q** | **p=IDOM(q)** and **HEAD(q)=HEAD(p)}**

Where **IDOM(q)** is the immediate dominator of **q**.

REACH(p):

Set of all nodes q entered by arcs from p or from nodes correspoding to to statements *nested* within p. Here nested has the intuitive meaning for `repeat` and statements.

Intermediate Representations

- IR design is an art not a science.
- Most important consideration is how appropriate is a representation for certain classes of analyses and optimizations.

  For example, dependence analysis requires knowledge of subscript expressions. Such information is not easily obtained from machine-language (low level) representations.

  Also, register allocation requires code that includes load and stores. Thus, representation using a stack machine are not appropriate.
- Three levels discussed in the book:

  High-Level IL: Usually in teh form of an AST.

  Medium-Level IL: Machine-language like, but machine independent.

  Low-Level IL: Very close to a particular machine although not identical.
- Alternative