Very Busy Expressions

This is an interesting variant of available expression analysis.

An expression is *very busy* at a point if it is *guaranteed* that the expression will be computed at some time in the future.

Thus starting at the point in question, the expression must be reached before its value changes.

Very busy expression analysis is a backward flow analysis, since it propagates information about future evaluations backward to "earlier" points in the computation.

The meet lattice is:

T (Expression is Very Busy)

|

F (Expression is Not Very Busy)

As initial values, at the end of all exit nodes, nothing is very busy. Hence, for a given expression,

VeryBusyOut(b_{last}) = F

The transfer function for e_1 in block b is defined as:

If e_1 is computed in b before any of its operands

Then VeryBusyIn(b) = T

Elsif any of e_1's operands are changed before e_1 is computed

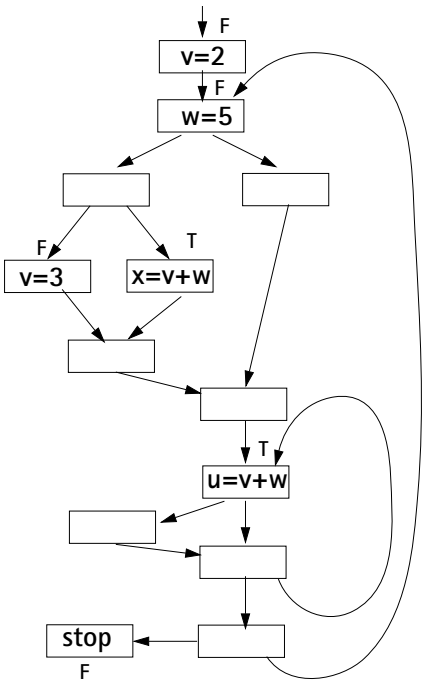
Then VeryBusyIn(b) = F

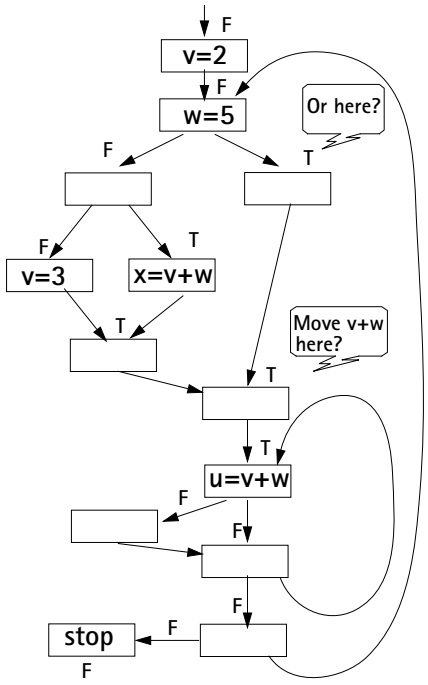
Else VeryBusyIn(b) = VeryBusyOut(b)

The meet operation (to combine solutions) is:

$$\text{VeryBusyOut}(b) = \bigwedge_{s \in \text{Succ}(b)} \text{VeryBusyIn}(s)$$

Example: e_1=v+w





Identifying Identical Expressions

We can hash expressions, based on hash values assigned to operands and operators. This makes recognizing potentially redundant expressions straightforward.

For example, if H(a) = 10, H(b) = 21 and H(+) = 5, then (using a simple product hash),
H(a+b) = 10×21×5 Mod TableSize

Effects of Aliasing and Calls

When looking for assignments to operands, we must consider the effects of pointers, formal parameters and calls.

An assignment through a pointer (e.g, `*p = val`) *kills* all expressions dependent on variables `p` might point too. Similarly, an assignment to a formal parameter kills all expressions dependent on variables the formal might be bound to.

A call kills all expressions dependent on a variable changeable during the call.

Lacking careful alias analysis, pointers, formal parameters and calls can kill all (or most) expressions.

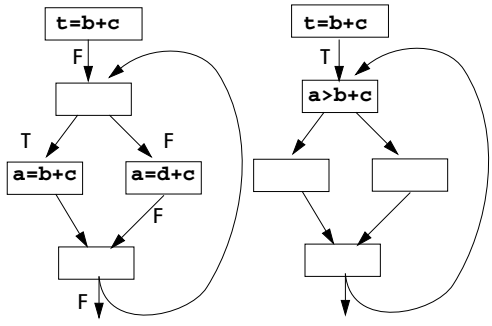
Very Busy Expressions and Loop Invariants

Very busy expressions are ideal candidates for invariant loop motion.

If an expression, invariant in a loop, is also very busy, we know it must be used in the future, and hence evaluation outside the loop must be worthwhile.

```
for (...) {
  if (...)
      a=b+c;
 else a=d+c;}
```

```
for (...) {
  if (a>b+c)
      x=1;
 else x=0;}
```



b+c is not very busy at loop entrance

b+c is very busy at loop entrance

Reaching Definitions

We have seen reaching definition analysis formulated as a set-valued problem. It can also be formulated on a per-definition basis.

That is, we ask "What blocks does a particular definition to v reach?"

This is a boolean-valued, forward flow data flow problem.

Initially, DefIn(b_0) = false.

For basic block b:

DefOut(b) =
  If the definition being analyzed is the last definition to v in b
  Then True
  Elsif any other definition to v occurs in b
  Then False
  Else DefIn(b)

The meet operation (to combine solutions) is:

$$\text{DefIn(b)} = \underset{p \in \text{Pred(b)}}{\text{OR}} \text{DefOut(p)}$$

To get all reaching definition, we do a series of single definition analyses.

Live Variable Analysis

This is a boolean-valued, backward flow data flow problem.

Initially, LiveOut(b_{last}) = false.

For basic block b:

LiveIn(b) =
  If the variable is used before it is defined in b
  Then True
  Elsif it is defined before it is used in b
  Then False
  Else LiveOut(b)

The meet operation (to combine solutions) is:

$$\text{LiveOut(b)} = \underset{s \in \text{Succ(b)}}{\text{OR}} \text{LiveIn(s)}$$

Bit Vectoring Data Flow Problems

The four data flow problems we have just reviewed all fit within a *single* framework.

Their solution values are Booleans (bits).

The meet operation is And or OR.

The transfer function is of the general form

$Out(b) = (In(b) - Kill_b) \cup Gen_b$

or

$In(b) = (Out(b) - Kill_b) \cup Gen_b$

where $Kill_b$ is true if a value is "killed" within b and Gen_b is true if a value is "generated" within b.

In Boolean terms:

$Out(b) = (In(b) \text{ AND Not } Kill_b) \text{ OR } Gen_b$

or

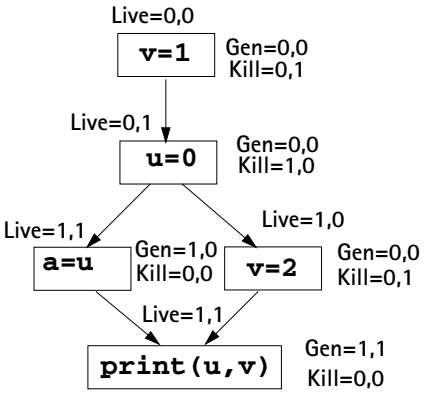
$In(b) = (Out(b) \text{ AND Not } Kill_b) \text{ OR } Gen_b$

An advantage of a bit vectoring data flow problem is that we can do a series of data flow problems "in parallel" using a bit vector.

Hence using ordinary word-level ANDs, ORs, and NOTs, we can solve 32 (or 64) problems simultaneously.

Example

Do live variable analysis for u and v, using a 2 bit vector:



We expect no variable to be live at the start of b_0. (Why?)

Reading Assignment

- Read pages 31-62 of "Automatic Program Optimization," by Ron Cytron. (Linked from the class Web page.)

Depth-First Spanning Trees

Sometimes we want to "cover" the nodes of a control flow graph with an acyclic structure.

This allows us to visit nodes once, without worrying about cycles or infinite loops.

Also, a careful visitation order can approximate forward control flow (very useful in solving forward data flow problems).

A Depth-First Spanning Tree (DFST) is a tree structure that covers the nodes of a control flow graph, with the start node serving as root of the DFST.

Building a DFST

We will visit CFG nodes in depth-first order, keeping arcs if the visited node hasn't be reached before.

To create a DFST, T, from a CFG, G:

1. T ← empty tree
2. Mark all nodes in G as "unvisited."
3. Call DF(start node)

DF (node) {

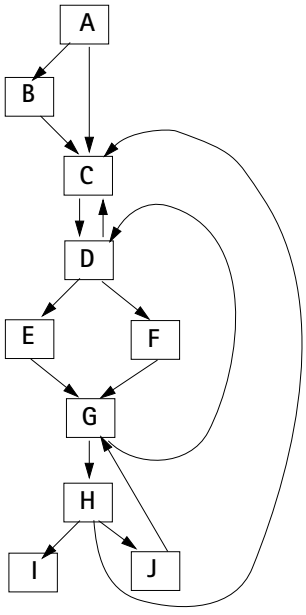
1. Mark node as visited.
2. For each successor, s, of node in G:

   If s is unvisited

   (a) Add node → s to T

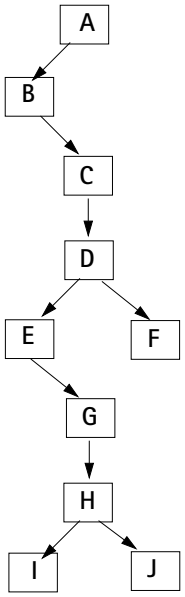
   (b) Call DF(s)

Example



Visit order is A, B, C, D, E, G, H, I, J, F

The DFST is



Categorizing Arcs using a DFST

Arcs in a CFG can be categorized by examining the corresponding DFST.

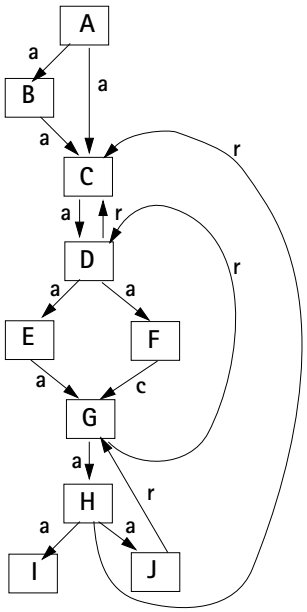
An arc A→B in a CFG is

(a) An *Advancing Edge* if B is a proper descendent of A in the DFST.

(b) A *Retreating Edge* if B is an ancestor of A in the DFST. (This includes the A→A case.)

(c) A *Cross Edge* if B is neither a descendent nor an ancestor of A in the DFST.

Example



Depth-First Order

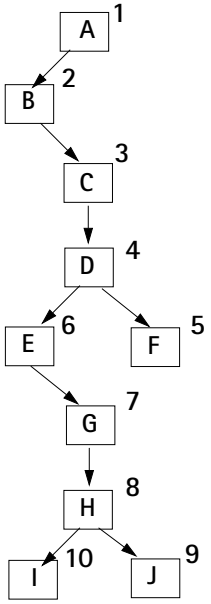
Once we have a DFST, we can label nodes with a *Depth-First Ordering* (DFO).

Let i = the number of nodes in a CFG (= the number of nodes in its DFST).

```
DFO(node) {
  For (each successor s of node) do
      DFO(s);
  Mark node with i;
  i--;
}
```


Example

The number of nodes = 10.



Application of Depth-First Ordering

- *Retreating edges* (a necessary component of loops) are easy to identify:
  a→b is a retreating edge if and only if dfo(b) ≤ dfo(a)
- A depth-first ordering in an excellent *visit order* for solving forward data flow problems. We want to visit nodes in essentially topological order, so that all predecessors of a node are visited (and evaluated) before the node itself is.

Dominators

A CFG node M *dominates* N (M dom N) if and only if *all* paths from the start node to N *must* pass through M.

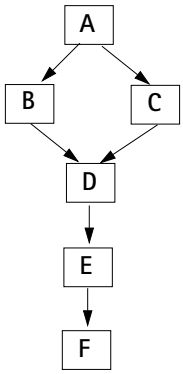
A node trivially dominates itself.

Thus (N dom N) is always true.

A CFG node M *strictly dominates* N (M sdom N) if and only if (M dom N) and M ≠ N.
A node can't strictly dominates itself.

Thus (N sdom N) is never true.

A CFG node may have many dominators.



Node F is dominated by F, E, D and A.

Immediate Dominators

If a CFG node has more than one dominator (which is common), there is always a unique "closest" dominator called its *immediate dominator*.

(M idom N) if and only if
(M sdom N) and
(P sdom N) ⇒ (P dom M)

To see that an immediate dominator always exists (except for the start node) and is unique, assume that node N is strictly dominated by M_1, M_2, ..., M_p, P ≥ 2.

By definition, M_1, ..., M_p must appear on *all* paths to N, including acyclic paths.

Look at the relative ordering among M_1 to M_p on some arbitrary acyclic path from the start node to N.

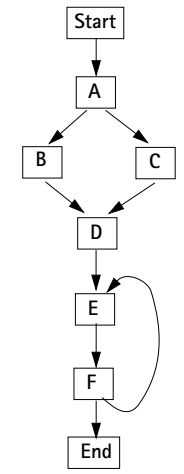
Assume that M_i is "last" on that path (and hence "nearest" to N).

If, on some other acyclic path, $M_j \neq M_i$ is last, then we can shorten this second path by going directly from M_i to N without touching any more of the M_1 to M_p nodes.

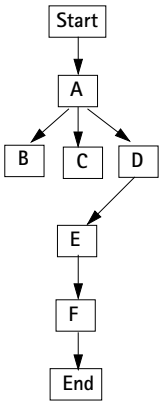
But, this totally removes M_j from the path, contradicting the assumption that (M_j sdom N).

Dominator Trees

Using immediate dominators, we can create a *dominator tree* in which A→B in the dominator tree if and only if (A idom B).

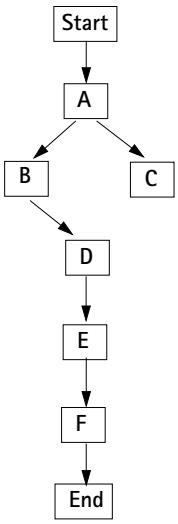


Control Flow Graph

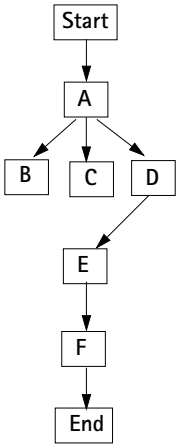


Dominator Tree

Note that the Dominator Tree of a CFG and its DFST are distinct trees (though they have the same nodes).



Depth-First Spanning Tree



Dominator Tree

A Dominator Tree is a compact and convenient representation of both the dom and idom relations.

A node in a Dominator Tree dominates all its descendents in the tree, and immediately dominates all its children.