Reading Assignment

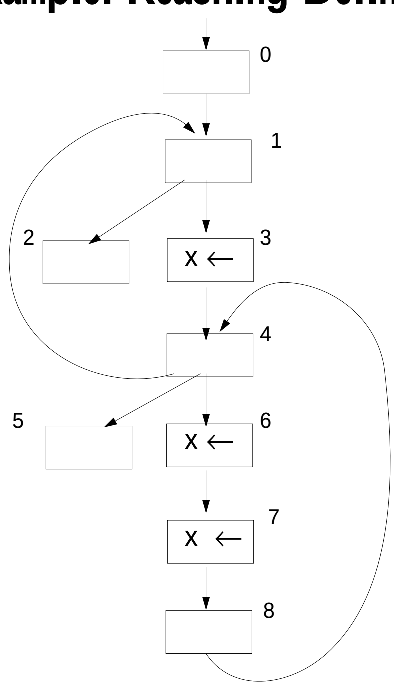
- Read pages 63-end of “Automatic Program Optimization,” by Ron Cytron. (Linked from the class Web page.)

Iterative Solution of Data Flow Problems

This algorithm will use DFO numbering to determine the order in which blocks are visited for evaluation. We iterate over the nodes until convergence.


```
EvalDF{
  For (all n ∈ CFG) {
     soln(n) = T
     ReEval(n) = true }
  Repeat
     LoopAgain = false
     For (all n ∈ CFG in DFO order){
         If (ReEval(n)) {
              ReEval(n) = false
              OldSoln = soln(n)
              In =      ∧      soln(p)
                    p ∈ Pred(n)
              soln(n) = f_n(In)
              If (soln(n) ≠ OldSoln) {
                 For (all s ∈ Succ(n)) {
                    ReEval(s) = true
                    LoopAgain = LoopAgain OR
                           IsBackEdge(n,s)
     } }     } }
  Until (! LoopAgain)
}
```


Example: Reaching Definitions



We'll do this as a set-valued problem (though it really is just three bit-valued analyses, since each analysis is independent).

L is the power set of Basic Blocks

$\wedge$ is set union

T is ϕ; $\perp$ is the set of all blocks

$a \leq b \equiv b \subseteq a$

$f_3(in) = \{3\}$

$f_6(in) = \{6\}$

$f_7(in) = \{7\}$

For all other blocks, $f_b(in) = in$

We'll track soln and ReEval across multiple passes

	0	1	2	3	4	5	6	7	8	Loop-Again
Initial	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	ϕ	**true**
	true	**true**	**true**	**true**	**true**	**true**	**true**	**true**	**true**	
Pass 1	ϕ	ϕ	ϕ	{3}	{3}	{3}	{6}	{7}	{7}	**true**
	false	**true**	**false**	**false**	**true**	**false**	**false**	**false**	**false**	
Pass 2	ϕ	{3}	{3}	{3}	{3,7}	{3,7}	{6}	{7}	{7}	**true**
	false	**true**	**false**	**false**	**false**	**false**	**false**	**false**	**false**	
Pass 3	ϕ	{3,7}	{3,7}	{3}	{3,7}	{3,7}	{6}	{7}	{7}	**false**
	false	**false**	**false**	**false**	**false**	**false**	**false**	**false**	**false**	

Properties of Iterative Data Flow Analysis

- If the height of the lattice (the maximum distance from T to $\perp$) is finite, then termination is *guaranteed*.
  Why?
  Recall that transfer functions are assumed monotone ($a \leq b \Rightarrow f(a) \leq f(b)$).
  Also, $\wedge$ has the property that
  $a \wedge b \leq a$ and $a \wedge b \leq b$.
  At each iteration, some solution value must change, else we halt. If something changes it must "move down" the lattice (we start at T). If the lattice has finite height, each block's value can change only a bounded number of times. Hence termination is guaranteed.

- If the iterative data flow algorithm terminates, a valid solution *must* have been computed. (This is because data flow values flow forward, and any change along a backedge forces another iteration.)

How Many Iterations are Needed?

Can we bound the number of iterations needed to compute a data flow solution?

In our example, 3 passes were needed, but why?

In an "ideal" CFG, with no loops or backedges, only 1 pass is needed.

With backedges, it can take several passes for a value computed in one block to reach a block that depends upon the value.

Let p be the maximum number of backedges in any acyclic path in the CFG.

Then (p+1) passes suffice to propagate a data flow value to any other block that uses it.

Recall that any block's value can change only a bounded number of times. In fact, the height of the lattice (maximum distance from top to bottom) is that bound.

Thus the maximum number of passes in our iterative data flow evaluator =

(p+1) * Height of Lattice

In our example, p = 2 and lattice height really was 1 (we did 3 independent bit valued problems).

So passes needed = (2+1)*1 = 3.

Rapid Data Flow Frameworks

We still have the concern that it may take many passes to traverse a solution lattice that has a significant height.

Many data flow problems are *rapid*. For rapid data flow problems, extra passes to feed back values along cyclic paths aren't needed.

For a data flow problem to be rapid we require that:

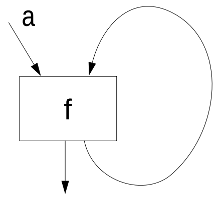
$(\forall a \in A)(\forall f \in F) \quad a \wedge f(T) \leq f(a)$

This is an odd requirement that states that using f(T) as a very crude approximation to a value computed by F is OK when joined using the $\wedge$ operator. In effect the term "a" rather than f(T) is dominant).

(Recall that $a \wedge f(a) \leq f(a)$ always holds.)

How does the Rapid Data Flow Property Help?

Consider a direct feedback loop (the idea holds for indirect loops too):



a is an input from outside the loop.

Our concern is how often we'll need to reevaluate f, as new values are computed and fed back into f.

Initially, we'll use T to model the value on the backedge.

Iteration 1: Input = $a \wedge T = a$
Output = $f(a)$
Iteration 2: Input = $a \wedge f(a)$
Output = $f(a \wedge f(a))$
Iteration 3: Input = $a \wedge f(a \wedge f(a))$

Now we'll exploit the rapid data flow property: $b \wedge f(T) \leq f(b)$

Let $b \equiv a \wedge f(a)$

Then $a \wedge f(a) \wedge f(T) \leq f(a \wedge f(a))$ (*)

Note that $x \leq y \Rightarrow a \wedge x \leq a \wedge y$ (**)

To prove this, recall that

(1) $p \wedge q = p \Rightarrow p \leq q$

(2) $x \leq y \Rightarrow x \wedge y = x$

Thus $(a \wedge x) \wedge (a \wedge y) = a \wedge (x \wedge y) = (a \wedge x)$ (by 2) $\Rightarrow (a \wedge x) \leq (a \wedge y)$ (by 1).

From (*) and (**) we get

$a \wedge a \wedge f(a) \wedge f(T) \leq f(a \wedge f(a)) \wedge a$ (***)

Now $a \leq T \Rightarrow f(a) \leq f(T) \Rightarrow$

$f(a) \wedge f(T) = f(a)$.

Using this on (***) we get

$a \wedge f(a) \leq f(a \wedge f(a)) \wedge a$

That is, $Input_2 \leq Input_3$

Note too that

$a \wedge f(a) \leq a \Rightarrow f(a \wedge f(a)) \leq f(a) \Rightarrow$
$a \wedge f(a \wedge f(a)) \leq a \wedge f(a)$

That is, $Input_3 \leq Input_2$

Thus we conclude $Input_2 = Input_3$, which means we can stop after two passes *independent* of lattice height!

(One initial visit plus one reevaluation via the backedge.)

Many Important Data Flow Problems are Rapid

Consider reaching definitions, done as sets. We may have many definitions to the same variable, so the height of the lattice may be large.

L is the power set of Basic Blocks

$\wedge$ is set union

T is ϕ; $\perp$ is the set of all blocks

$a \leq b \equiv a \supseteq b$

$f_b(in) = (In - Kill_b) \cup Gen_b$

where Gen_b is the last definition to a variable in b,

$Kill_b$ is all defs to a variable except the last one in b,

$Kill_b$ is empty if there is no def to a variable in b.

The Rapid Data Flow Property is

$a \wedge f(\top) \le f(a)$

In terms of Reaching Definitions this is

$a \cup f(\phi) \supseteq f(a) \equiv$

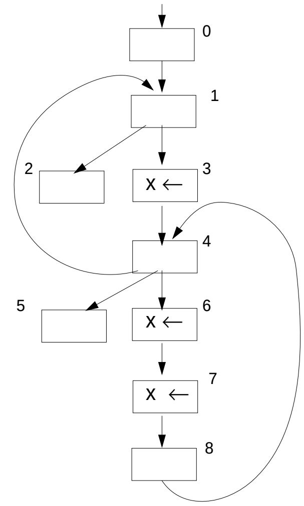
$a \cup (\phi - Kill) \cup Gen \supseteq (a - Kill) \cup Gen$

Simplifying,

$a \cup Gen \supseteq (a - Kill) \cup Gen$

which always holds.

Recall



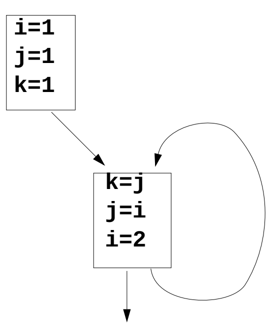
Here it took two passes to transmit the def in b7 to b1, so we expect 3 passes to evaluate *independent* of the lattice height.

Constant Propagation isn't Rapid

We require that

$a \wedge f(\top) \le f(a)$

Consider



Look at the transfer function for the second (bottom) block.

f(t) = t' where

t'(v) = case(v){

k: t(j);

j: t(i);

i: 2; }

Let $a = (\perp,1,1)$.

$f(\top) = (2,\top,\top)$

$a \wedge f(\top) = (\perp,1,1) \wedge (2,\top,\top) = (\perp,1,1)$

$f(a) = f(\perp,1,1) = (2,\perp,1)$.

Now $(\perp,1,1)$ is not $\le (2,\perp,1)$

so this problem isn't rapid.

Let's follow the iterations:

Pass 1: In = $(1,1,1) \wedge (T,T,T) = (1,1,1)$

Out = $(2,1,1)$

Pass 2: In = $(1,1,1) \wedge (2,1,1) = (\perp,1,1)$

Out = $(2,\perp,1)$

Pass 3: In = $(1,1,1) \wedge (2,\perp,1) = (\perp,\perp,1)$

Out = $(2,\perp,\perp)$

This took 3 passes. In general, if we had N variables, we could require N passes, with each pass resolving the constant status of one variable.

How Good Is Iterative Data Flow Analysis?

A single execution of a program will follow some path $b_0, b_{i_1}, b_{i_2}, \ldots, b_{i_n}$.

The Data Flow solution along this path is

$f_{i_n}(\ldots f_{i_2}(f_{i_1}(f_0(T)))\ldots) \equiv f(b_0, b_1, \ldots, b_{i_n})$

The best possible static data flow solution at some block b is computed over all possible paths from b_0 to b.

Let P_b = The set of all paths from b_0 to b.

$$MOP(b) = \bigwedge_{p \in P_b} f(p)$$

Any particular path p_i from b_0 to b is included in P_b.

Thus $MOP(b) \wedge f(p_i) = MOP(b) \le f(p_i)$.

This means MOP(b) is *always* a safe approximation to the "true" solution $f(p_i)$.

If we have the distributive property for transfer functions,

$f(a \wedge b) = f(a) \wedge f(b)$

then our iterative algorithm *always* computes the MOP solution, the best static solution possible.

To prove this, note that for trivial path of length 1, containing only the start block, b_0, the algorithm computes $f_0(T)$ which is $MOP(b_0)$ (trivially).

Now assume that the iterative algorithm for paths of length n or less to block c *does* compute MOP(c).

We'll show that for paths to block b of length n+1, MOP(b) is computed.

Let P be the set of all paths to b of length n+1 or less.

The paths in P end with b.

MOP(b) = $f_b(f(P_1)) \wedge f_b(f(P_2) \wedge \ldots$

where P_1, P_2, ... are the prefixes (of length n or less) of paths in P with b removed.

Using the distributive property,

$f_b(f(P_1)) \wedge f_b(f(P_2) \wedge \ldots =$

$f_b(f(P_1) \wedge f(P_2) \wedge \ldots)$.

But note that $f(P_1) \wedge f(P_2) \wedge \ldots$ is just the input to f_b in our iterative algorithm, which then applies f_b.

Thus MOP(b) for paths of length n+1 is computed.

For data flow problems that aren't distributive (like constant propagation), the iterative solution is ≤ the MOP solution.

This means that the solution is a safe approximation, but perhaps not as "sharp" as we might wish.