Data Flow Frameworks Revisited

Recall that a Data Flow problem is characterized as:

(a) A Control Flow Graph

(b) A Lattice of Data Flow values

(c) A Meet operator to join solutions from Predecessors or Successors

(d) A Transfer Function
$Out = f_b(In)$ or $In = f_b(Out)$

Value Lattice

The lattice of values is usually a *meet semilattice* defined by:

A: a set of values

T and $\perp$ ("top" and "bottom"): distinguished values in the lattice

$\leq$: A reflexive partial order relating values in the lattice

$\wedge$: An associative and commutative meet operator on lattice values

Lattice Axioms

The following axioms apply to the lattice defined by A, T, $\perp$, $\leq$ and $\wedge$:

$a \leq b \iff a \wedge b = a$

$a \wedge a = a$

$(a \wedge b) \leq a$

$(a \wedge b) \leq b$

$(a \wedge T) = a$

$(a \wedge \perp) = \perp$

Monotone Transfer Function

Transfer Functions, f_b:L $\rightarrow$ L (where L is the Data Flow Lattice) are normally required to be monotone.

That is $x \leq y \Rightarrow f_b(x) \leq f_b(y)$.

This rule states that a "worse" input can't produce a "better" output.

Monotone transfer functions allow us to guarantee that data flow solutions are stable.

If we had $f_b(T) = \perp$ and $f_b(\perp)=T$, then solutions might oscillate between T and $\perp$ indefinitely.

Since $\perp \leq T$, $f_b(\perp)$ should be $\leq f_b(T)$. But $f_b(\perp) = T$ which is not $\leq f_b(T) = \perp$. Thus f_b isn't monotone.

Dominators fit the Data Flow Framework

Given a set of Basic Blocks, N, we have:

A is 2^N (all subsets of Basic Blocks).

T is N.

$\perp$ is ϕ.

$a \le b \equiv a \subseteq b$.

$f_Z(in) = In \cup \{Z\}$

$\wedge$ is $\cap$ (set intersection).

The required axioms are satisfied:

$a \subseteq b \Leftrightarrow a \cap b = a$

$a \cap a = a$

$(a \cap b) \subseteq a$

$(a \cap b) \subseteq b$

$(a \cap N) = a$

$(a \cap \phi) = \phi$

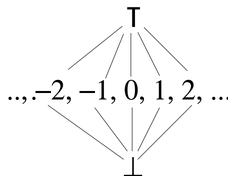
Also f_Z is monotone since

$a \subseteq b \Rightarrow a \cup \{Z\} \subseteq b \cup \{Z\} \Rightarrow f_Z(a) \subseteq f_Z(b)$

Constant Propagation

We can model Constant Propagation as a Data Flow Problem. For each scalar integer variable, we will determine whether it is known to hold a particular constant value at a particular basic block.

The value lattice is



T represents a variable holding a constant, whose value is not yet known.

i represents a variable holding a known constant value.

$\perp$ represents a variable whose value is non-constant.

This analysis is complicated by the fact that variables interact, so we can't just do a series of independent one variable analyses.

Instead, the solution lattice will contain functions (or vectors) that map each variable in the program to its constant status (T, $\perp$, or some integer).

Let V be the set of all variables in a program.

Let $t : V \rightarrow N \cup \{T,\perp\}$

t is the set of all total mappings from V (the set of variables) to $N \cup \{T,\perp\}$ (the lattice of "constant status" values).

For example, $t_1=(T,6,\perp)$ is a mapping for three variables (call them A, B and C) into their constant status. t_1 says A is considered a constant, with value as yet undetermined. B holds the value 6, and C is non-constant.

We can create a lattice composed of t functions:

$t_T(V) = T\ (\forall V)\ (t_T=(T,T,T, ...)$

$t_\perp(V) = \perp\ (\forall V)\ (t_\perp=(\perp,\perp,\perp, ...)$

$t_a \le t_b \Leftrightarrow \forall v\ t_a(v) \le t_b(v)$

Thus $(1,\perp) \le (T,3)$
since $1 \le T$ and $\perp \le 3$.

The meet operator $\wedge$ is applied *componentwise*:

$t_a \wedge t_b = t_c$
where $\forall v\ t_c(v) = t_a(v) \wedge t_b(b)$

Thus $(1,\perp) \wedge (T,3) = (1,\perp)$
since $1 \wedge T = 1$ and $\perp \wedge 3 = \perp$.

The lattice axioms hold:

$t_a \le t_b \Leftrightarrow t_a \wedge t_b = t_a$ (since this axiom holds for each component)

$t_a \wedge t_a = t_a$ (trivially holds)

$(t_a \wedge t_b) \le t_a$ (per variable def of $\wedge$)

$(t_a \wedge t_b) \le t_b$ (per variable def of $\wedge$)

$(t_a \wedge t_T) = t_a$ (true for all components)

$(t_a \wedge t_\perp) = t_\perp$ (true for all components)

The Transfer Function

Constant propagation is a forward flow problem, so $Cout = f_b(Cin)$

Cin is a function, t(v), that maps variables to T, $\perp$, or an integer value

$f_b(t(v))$ is defined as:

(1) Initially, let $t'(v)=t(v)\ (\forall v)$

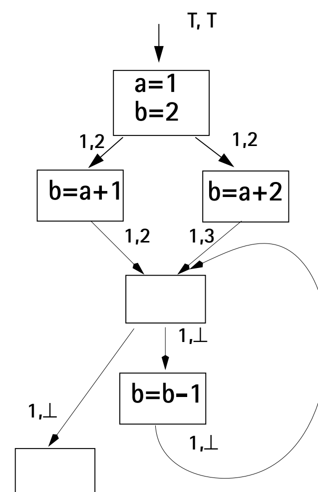
(2) For each assignment statement
$v = e(w_1,w_2,...,w_n)$
in b, in order of execution, do:
If any $t'(w_i) = \perp$ ($1 \le i \le n$)
Then set $t'(v) = \perp$ (strictness)
Elsif any $t'(w_i) = T$ ($1 \le i \le n$)
Then set $t'(v) = T$ (delay eval of v)
Else $t'(v) = e(t'(w_1),t'(w_2),...)$

(3) $Cout = t'(v)$

Note that in valid programs, we don't use uninitialized variables, so variables mapped to T should only occur prior to initialization.

Initially, all variables are mapped to T, indicating that initially their constant status is unknown.

Example



Distributive Functions

From the properties of $\wedge$ and f's monotone property, we can show that

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

To see this note that

$a \wedge b \le a,\ a \wedge b \le b \Rightarrow$

$f(a \wedge b) \le f(a),\ f(a \wedge b) \le f(b)$ (*)

Now we can establish that

$x \le y,\ x \le z \Rightarrow x \le y \wedge z$ (**)

To see that (**) holds, note that

$x \le y \Rightarrow x \wedge y = x$

$x \le z \Rightarrow x \wedge z = x$

$(y \wedge z) \wedge x \le y \wedge z$

$(y \wedge z) \wedge x = (y \wedge z) \wedge (x \wedge x) =$

$(y \wedge x) \wedge (z \wedge x) = x \wedge x = x$

Thus $x \le y \wedge z$, establishing (**).

Now substituting $f(a \wedge b)$ for x, f(a) for y and f(b) for z in (**) and using (*) we get

$f(a \wedge b) \le f(a) \wedge f(b)$.

Many Data Flow problems have flow equations that satisfy the *distributive property*:

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

For example, in our formulation of dominators:

$Out = f_b(In) = In \cup \{b\}$

where

$$In = \bigcap_{p \in Pred(b)} Out(p)$$

In this case, $\wedge$ = $\cap$.

Now $f_b(S_1 \cap S_2) = (S_1 \cap S_2) \cup \{b\}$

Also, $f_b(S_1) \cap f_b(S_2)$ =

$(S_1 \cup \{b\}) \cap (S_2 \cup \{b\})$ =

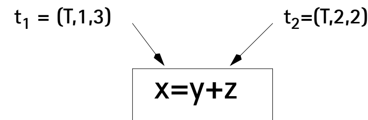
$(S_1 \cap S_2) \cup \{b\}$

So dominators are distributive.

Not all Data Flow Problems are Distributive

Constant propagation is *not* distributive.

Consider the following (with variables (x,y,z)):



Now f(t)=t' where

t'(y) = t(y), t'(z) = t(z),

t'(x) = if t(y)=$\perp$ or t(z) = $\perp$
 then $\perp$
 elseif t(y)=T or t(z) =T
 then T
 else t(y)+t(z)

Now $f(t_1 \wedge t_2) = f(T,\perp,\perp) = (\perp,\perp,\perp)$

$f(t_1) = (4,1,3)$

$f(t_2) = (4,2,2)$

$f(t_1) \wedge f(t_2) = (4,\perp,\perp) \geq (\perp,\perp,\perp)$

Why does it Matter if a Data Flow Problem isn't Distributive?

Consider actual program execution paths from b_0 to (say) b_k.

One path might be $b_0, b_{i_1}, b_{i_2}, \ldots, b_{i_n}$ where $b_{i_n} = b_k$.

At b_k the Data Flow information we want is

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

On a different path to b_k, say $b_0, b_{j_1}, b_{j_2}, \ldots, b_{j_m}$, where $b_{j_m} = b_k$

the Data Flow result we get is $f_{j_m}(\ldots f_{j_2}(f_{j_1}(f_0(T)))\ldots) \equiv$
 $f(b_0, b_{j_1}, \ldots, b_{j_m})$.

Since we can't know at compile time which path will be taken, we must *combine* all possible paths:

$$\bigwedge_{p \in \text{all paths to } b_k} f(p)$$

This is the *meet over all paths* (MOP) solution. It is the *best possible* static solution. (Why?)

As we shall see, the meet over all paths solution can be computed efficiently, using standard Data Flow techniques, if the problem is Distributive.

Other, non-distributive problems (like Constant Propagation) can't be solved as precisely.

Explicitly computing and meeting all paths is prohibitively expensive.

Conditional Constant Propagation

We can extend our Constant Propagation Analysis to determine that some paths in a CFG aren't executable. This is *Conditional Constant Propagation*.

Consider

```
i = 1;
if (i > 0)
     j = 1;
else j = 2;
```

Conditional Constant Propagation can determine that the else part of the if is unreachable, and hence `j` must be 1.

The idea behind Conditional Constant Propagation is simple. Initially, we mark all edges out of conditionals as "not reachable."

Starting at b_0, we propagate constant information *only* along edges considered reachable.

When a boolean expression $b(v_1,v_2,...)$ controls a conditional branch, we evaluate $b(v_1,v_2,...)$ using the t(v) mapping that identifies the "constant status" of variables.

If $t(v_i)=T$ for any v_i, we consider all out edges unreachable (for now).

Otherwise, we evaluate $b(v_1,v_2,...)$ using t(v), getting true, false or $\bot$.

Note that the short-circuit properties of boolean operators may yield true or false even if $t(v_i)=\bot$ for some v_i.

If $b(v_1,v_2,...)$ is true or false, we mark only one out edge as reachable.

Otherwise, if $b(v_1,v_2,...)$ evaluates to $\bot$, we mark all out edges as reachable.

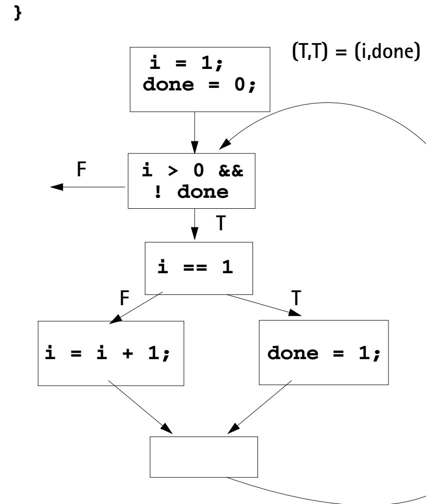
We propagate constant information only along reachable edges.

Example

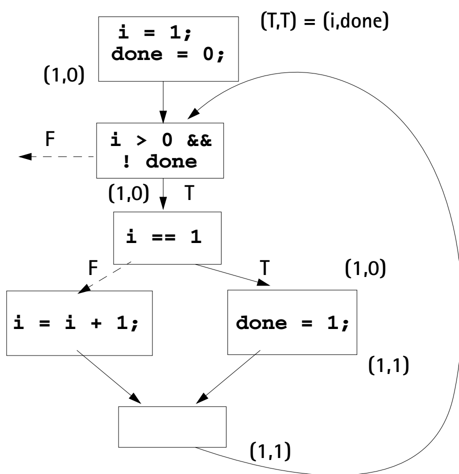
```
i = 1;
done = 0;
while ( i > 0 && ! done) {
  if (i == 1)
       done = 1;
  else i = i + 1;
```



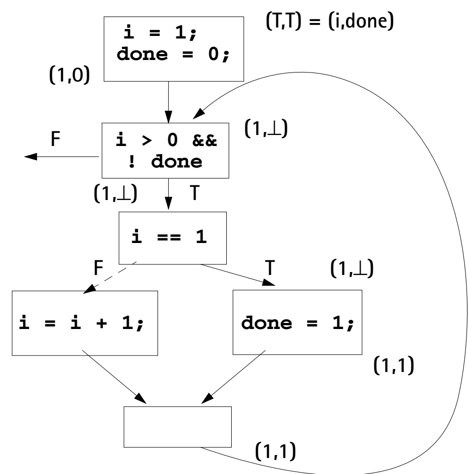
```
}
```



Pass 1:



Pass 2:



Reading Assignment

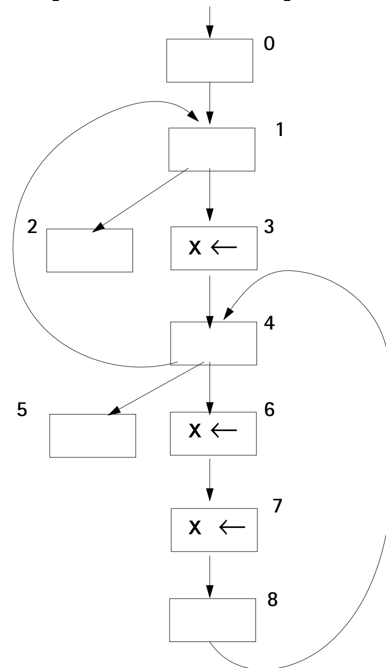
- Read pages 63–end of "Automatic Program Optimization," by Ron Cytron. (Linked from the class Web page.)

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

$a \le b \equiv b \subseteq a$

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

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

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

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 ($a \le b \Rightarrow f(a) \le f(b)$).
  Also, $\wedge$ has the property that
  $a \wedge b \le a$ and $a \wedge b \le b$.
  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.