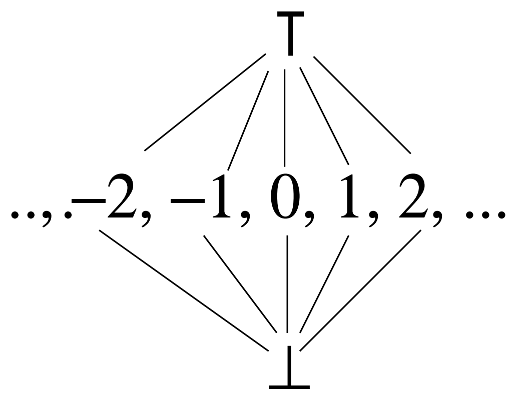
Constant Propagation

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 ($\top$, $\perp$, 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:

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

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

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

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

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,⊥, or an integer value

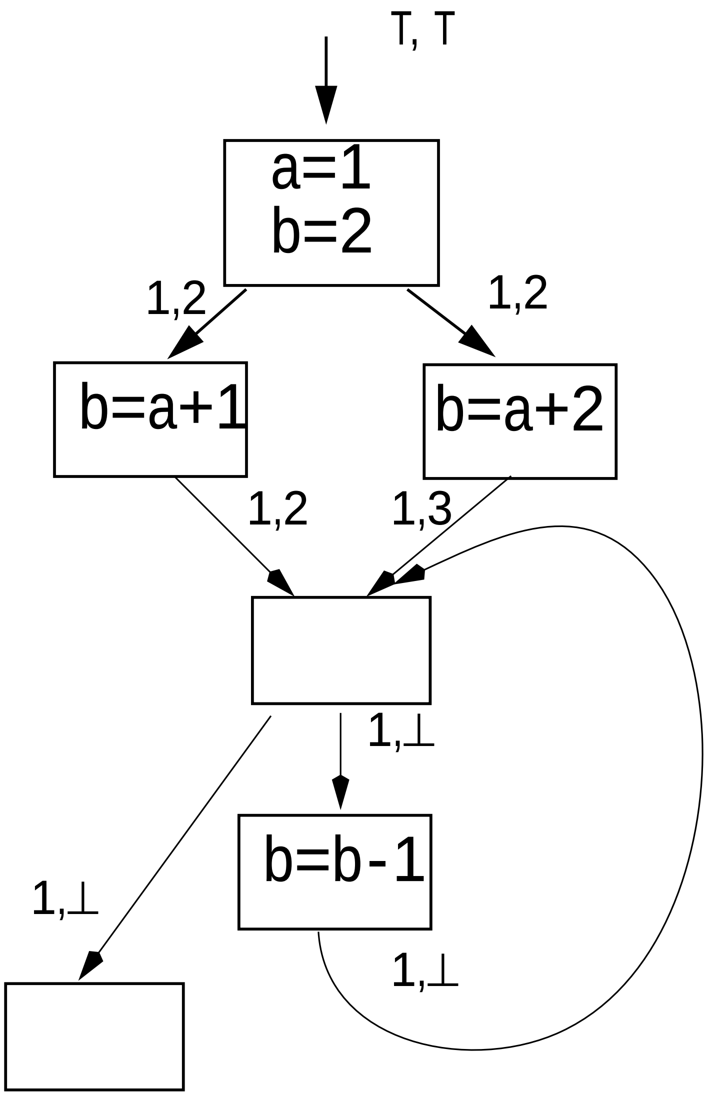
f_b(t(v)) is defined as:

```
(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'($w_i$) = ⊥ ( 1≤i≤n )
    Then set t'(v) = ⊥ (strictness)
    Elsif any t'($w_i$) = T (1≤i≤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

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

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

Now we can establish that

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

To see that (**) holds, note that

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

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

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

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

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

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:

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

where

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

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

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

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

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

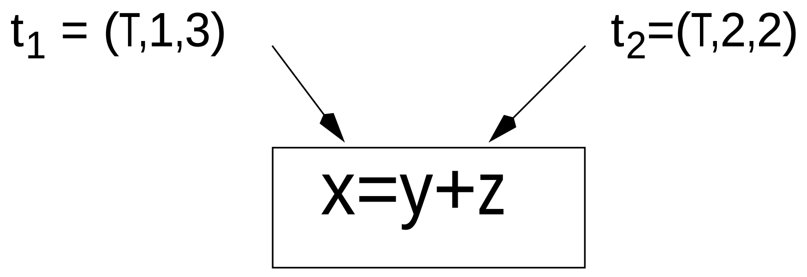
$(S_1 \cap S_2) \cup \{b\}$

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)=⊥ or t(z) = ⊥
            then ⊥
            elseif t(y)=T or t(z) =T
            then T
            else t(y)+t(z)
```

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

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

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

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

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 $b_0,b_{i_1},b_{i_2},...,b_{i_n}$ where $b_{i_n}=b_k$.

At b_k the Data Flow information we want is

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

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

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

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 $\perp$.

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

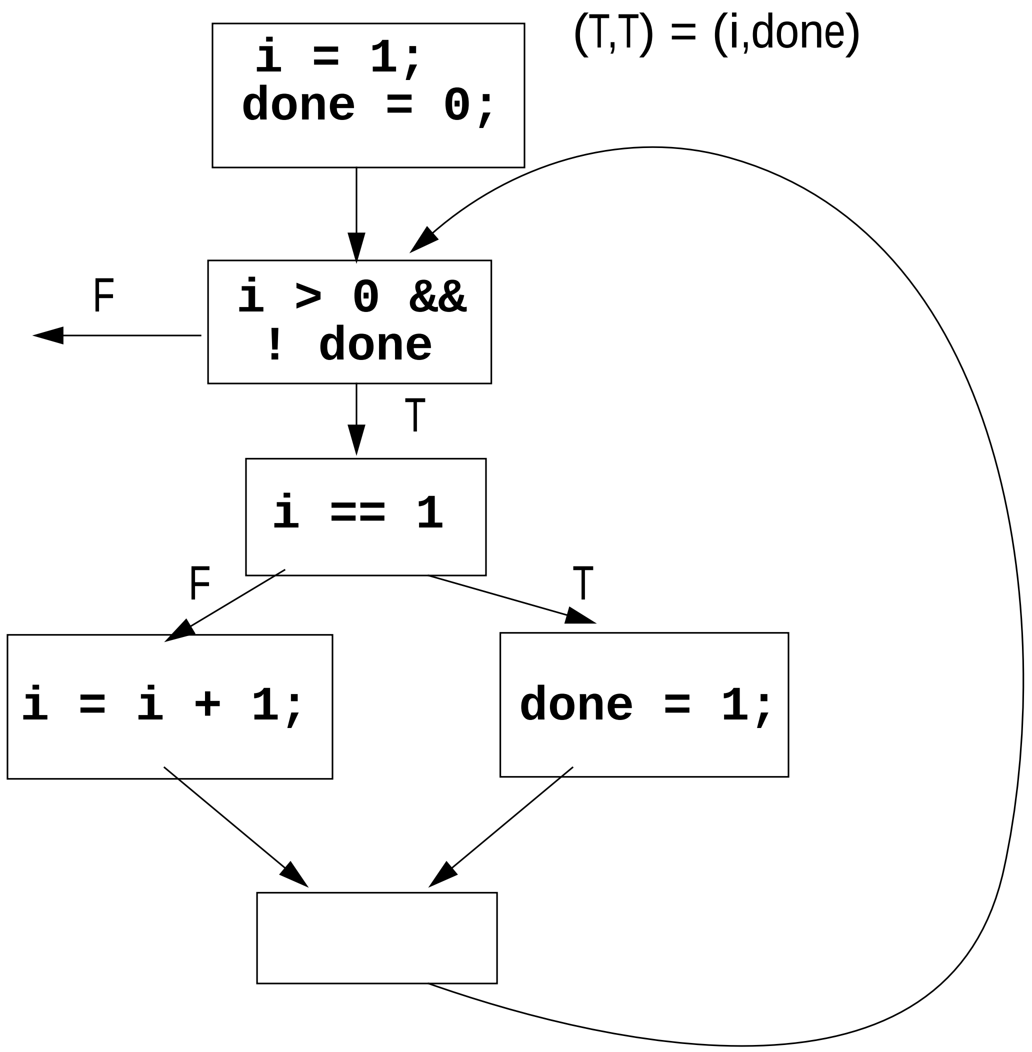
If $b(v_1,v_2,...)$ is true or false, we mark only one out edge as reachable.

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

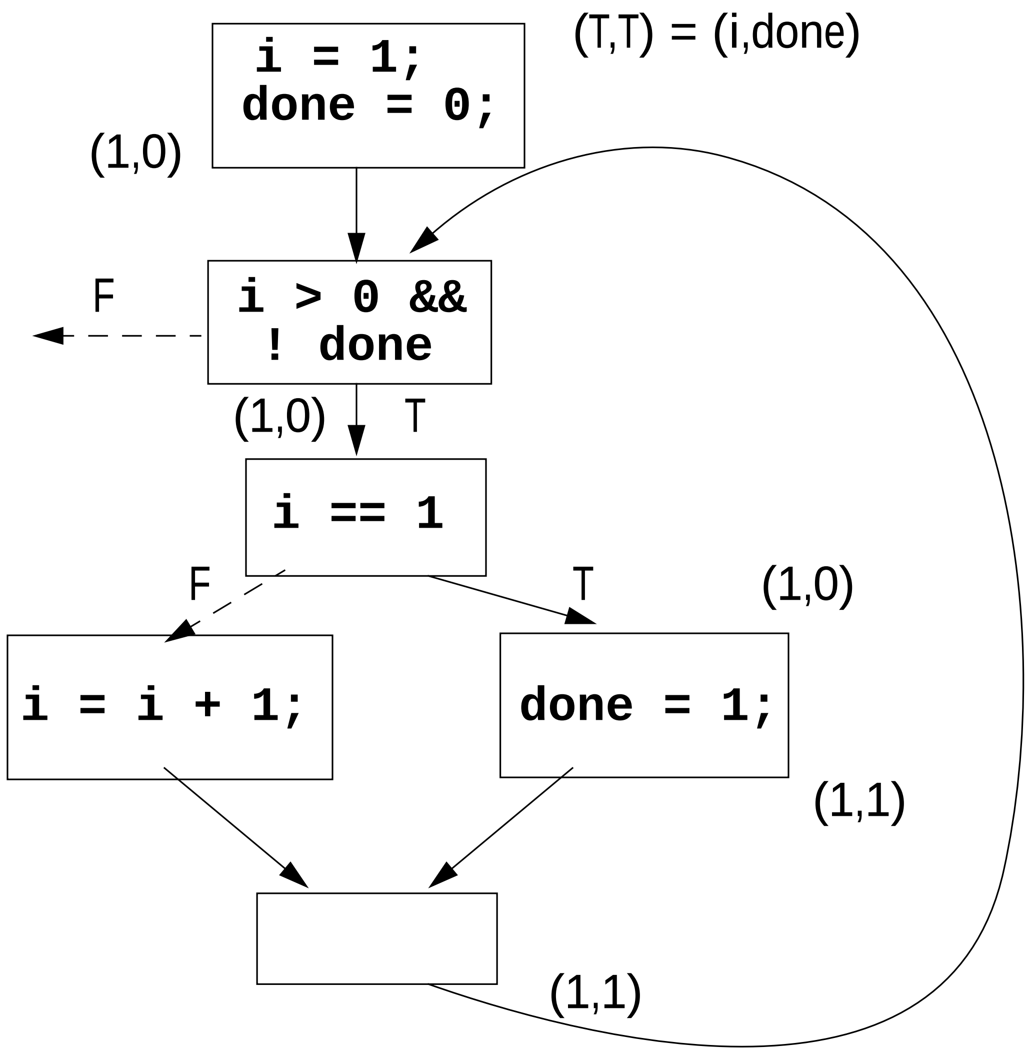
We propagate constant information only along reachable edges.

Example

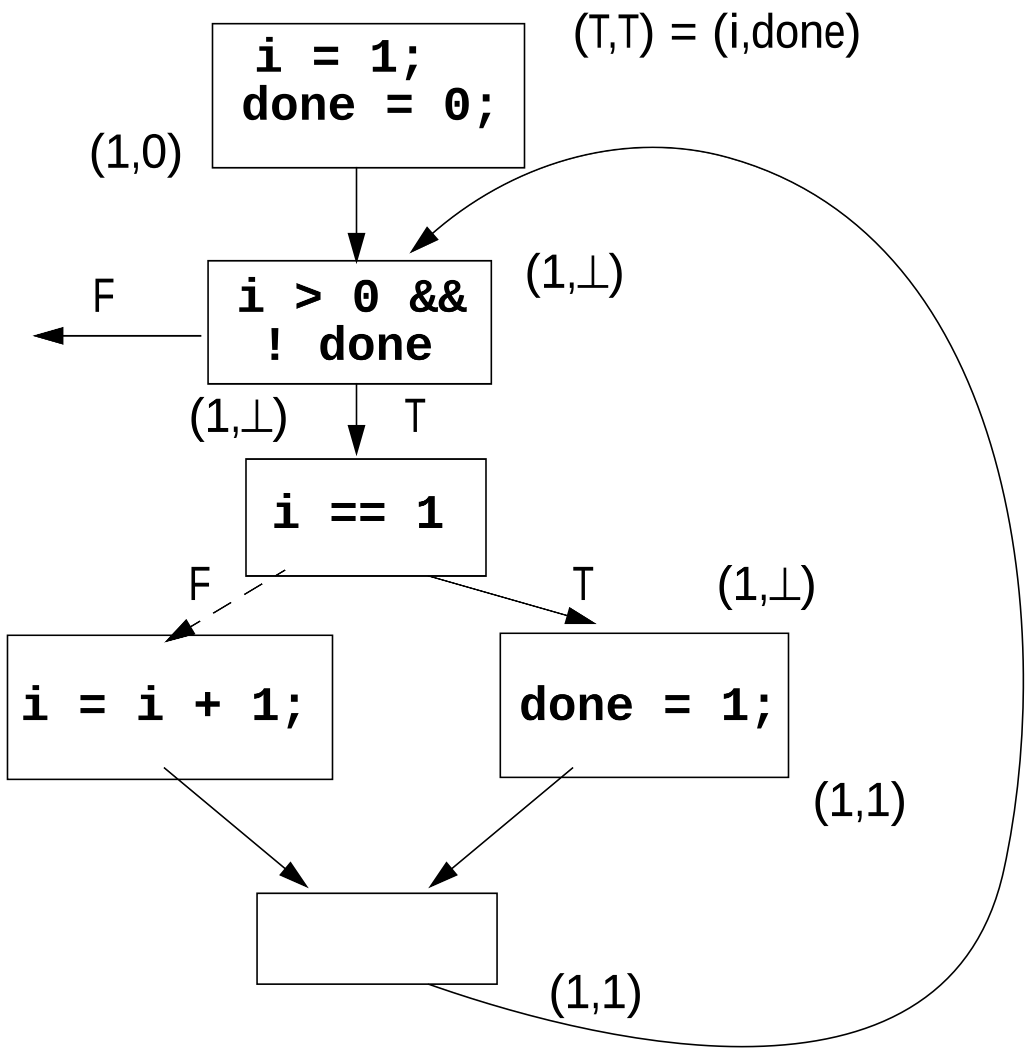
```
i = 1;
done = 0;
while ( i > 0 && ! done) {
  if (i == 1)
       done = 1;
  else i = i + 1;
}
```



Pass 1:



Pass 2:


(T,T) = (i,done)
i = 1;
done = 0;
(1,0)
(1,⊥)
F
i > 0 &&
! done
(1,⊥)
T
i == 1
F
T
(1,⊥)
i = i + 1;
done = 1;
(1,1)
(1,1)