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 \( \phi \).
- \( a \leq b \iff a \subseteq b \).
- \( f_Z(\text{in}) = \text{In} \cup \{Z\} \)
- \( \wedge \) is \( \cap \) (set intersection).

The required axioms are satisfied:
- \( a \subseteq b \iff a \cap b = a \)
- \( a \cap a = a \)
- \( (a \cap b) \subseteq a \)
- \( (a \cap b) \subseteq b \)
- \( (a \cap N) = a \)
- \( (a \cap \phi) = \phi \)

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

\[
\begin{align*}
T & \quad \perp \\
\ldots, -2, -1, 0, 1, 2, \ldots
\end{align*}
\]

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 \( t : V \rightarrow N \cup \{T, \bot\} \)

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

For example, \( t_1 = (T, 6, \bot) \) 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:

\[
\begin{align*}
t_T(V) &= T \quad (\forall V) \quad (t_T = (T, T, T, \ldots)) \\
t_\bot(V) &= \bot \quad (\forall V) \quad (t_\bot = (\bot, \bot, \bot, \ldots))
\end{align*}
\]

The lattice axioms hold:

\( t_a \leq t_b \iff \forall v \quad t_a(v) \leq t_b(v) \)

Thus \((1, \bot) \leq (T, 3)\)

since \(1 \leq T\) and \(\bot \leq 3\).

The meet operator \( \wedge \) is applied componentwise:

\[
(1, \bot) \wedge (T, 3) = (1, \bot)
\]

since \(1 \wedge T = 1\) and \(\bot \wedge 3 = \bot\).

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 \( T, \bot, \) or an integer value \( f_b(t(v)) \) is defined as:

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

(2) For each assignment statement \( v = e(w_1, w_2, \ldots, w_n) \) in \( b \), in order of execution, do:

\[
\begin{align*}
&\text{If any } t'(w_i) = \bot \quad (1 \leq i \leq n) \\
&\quad \text{Then set } t'(v) = \bot \quad \text{(strictness)} \\
&\text{Elseif any } t'(w_i) = T \quad (1 \leq i \leq n) \\
&\quad \text{Then set } t'(v) = T \quad \text{(delay eval of } v) \\
&\quad \text{Else } t'(v) = e(t'(w_1), t'(w_2), \ldots)
\end{align*}
\]

(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.

Distributive Functions

From the properties of $\land$ and $f$’s monotone property, we can show $f(a \land b) \leq f(a) \land f(b)$
To see this note that $a \land b \leq a, a \land b \leq b$ \Rightarrow
$f(a \land b) \leq f(a), f(a \land b) \leq f(b) \quad (*)$
Now we can establish that $x \leq y, x \leq z \Rightarrow x \leq y \land z \quad (**)$
To see that (**) holds, note that $x \leq y \Rightarrow x \land y = x$
$x \leq z \Rightarrow x \land z = x$
$(y \land z) \land x \leq y \land z$
$(y \land z) \land x = (y \land z) \land (x \land x) =$
$(y \land x) \land (z \land x) = x \land x = x$
Thus $x \leq y \land z$, establishing (**).

Example

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

Many Data Flow problems have flow equations that satisfy the distributive property:
$f(a \land b) = f(a) \land f(b)$
For example, in our formulation of dominators:
Out = $f_b$(In) = In U {b}
where

$$\text{In} = \bigcap_{p \in \text{Pred}(b)} \text{Out}(p)$$
In this case, $\land = \cap$.

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 $t''(y) = t(y), t''(z) = t(z), t''(x) = \text{if } t(y) = \bot \text{ or } t(z) = \bot \text{ then } \bot$

elseif $t(y) = T \text{ or } t(z) = T \text{ then } T$

else $t(y) + t(z)$

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

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 $f_{j_m}(...f_{j_2}(f_{j_1}(f_0(T)))...) = f(b_0, b_{j_1}, ..., b_{j_m})$. 
Since we can’t know at compile time which path will be taken, we must combine all possible paths:

\[ \bigwedge_{p \text{ all paths to } b_k} f(p) \]

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

\[
i = 1;
\]

\[
\text{if } (i > 0) \quad j = 1;
\]

\[
\text{else } j = 2;
\]

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

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

Pass 1:

```
(T,T) = (i,done)
```

Pass 2:

```
(T,T) = (i,done)
```