Dominance Frontiers

Dominators and postdominators tell us which basic block must be executed prior to, of after, a block N.

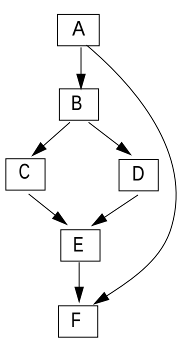
It is interesting to consider blocks “just before” or “just after” blocks we’re dominated by, or blocks we dominate.

The Dominance Frontier of a basic block N, DF(N), is the set of all blocks that are immediate successors to blocks dominated by N, but which aren’t themselves strictly dominated by N.

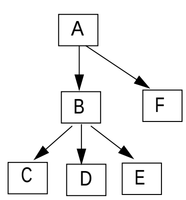
DF(N) =
$\{Z \mid M \rightarrow Z$ & (N dom M) &
$\neg$(N sdom Z)$\}$

The dominance frontier of N is the set of blocks that are not dominated N and which are “first reached” on paths from N.

Example



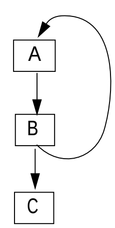
Control Flow Graph



Dominator Tree

Block	A	B	C	D	E	F
Dominance Frontier	ϕ	{F}	{E}	{E}	{F}	ϕ

A block can be in its own Dominance Frontier:



Here, DF(A) = {A}

Why? Reconsider the definition:

DF(N) =
$\{Z \mid M \rightarrow Z$ & (N dom M) &
$\neg$(N sdom Z)$\}$

Now B is dominated by A and B$\rightarrow$A.

Moreover, A does not *strictly* dominate itself. So, it meets the definition.

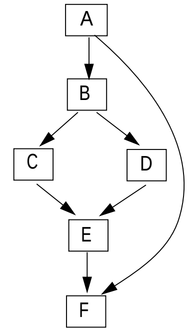
Postdominance Frontiers

The Postdominance Frontier of a basic block N, PDF(N), is the set of all blocks that are immediate predecessors to blocks postdominated by N, but which aren't themselves postdominated by N.

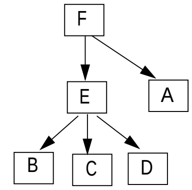
PDF(N) =
$\{Z \mid Z \rightarrow M \ \& \ (N \text{ pdom } M) \ \& \ \neg(N \text{ pdom } Z)\}$

The postdominance frontier of N is the set of blocks closest to N where a choice was made of whether to reach N or not.

Example



Control Flow Graph



Postominator Tree

Block	A	B	C	D	E	F
Postdominance Frontier	ϕ	{A}	{B}	{B}	{A}	ϕ

Control Dependence

Since CFGs model flow of control, it is useful to identify those basic blocks whose execution is controlled by a branch decision made by a predecessor.

We say Y is *control dependent* on X if, reaching X, choosing one out arc will force Y to be reached, while choosing another arc out of X allows Y to be avoided.

Formally, Y is control dependent on X if and only if,

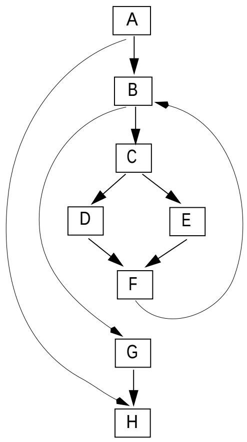
(a) Y postdominates a successor of X.

(b) Y does not postdominate all successors of X.

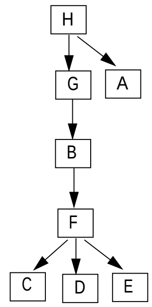
X is the most recent block where a choice was made to reach Y or not.

Control Dependence Graph

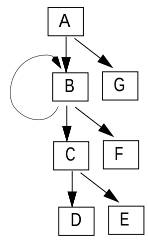
We can build a *Control Dependence Graph* that shows (in graphical form) all Control Dependence relations.
(A Block *can be* Control Dependent on itself.)



Control Flow Graph



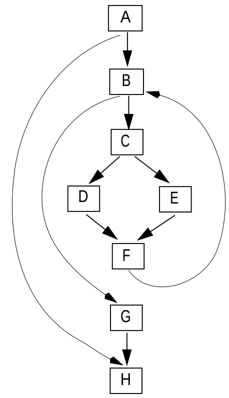
Postominator Tree



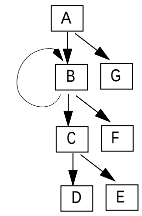
Control Dependence Graph

What happened to H in the CD Graph?

Let’s reconsider the CD Graph:



Control Flow Graph



Control Dependence Graph

Blocks C and F, as well as D and E, seem to have the same control dependence relations with their parent. But this isn’t so!

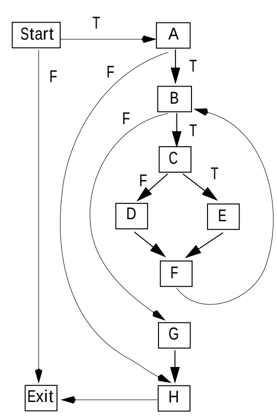
C and F *are* control equivalent, but D and E are *mutually exclusive*!

Improving the Representation of Control Dependence

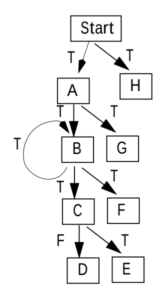
We can label arcs in the CFG and the CD Graph with the condition (T or F or some switch value) that caused the arc to be selected for execution.

This labeling then shows the conditions that lead to the execution of a given block.

To allow the exit block to appear in the CD Graph, we can also add “artificial” start and exit blocks, linked together.



Control Flow Graph



Control Dependence Graph

Now C and F have the same Control Dependence relations—they are part of the same extended basic block.

But D and E aren’t identically control dependent. Similarly, A and H are control equivalent, as are B and G.

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 ⊥ (“top” and “bottom”): distinguished values in the lattice

≤: A reflexive partial order relating values in the lattice

∧: An associative and commutative meet operator on lattice values

Lattice Axioms

The following axioms apply to the lattice defined by A, T, ⊥, ≤ and ∧:

$a \le b \Leftrightarrow a \wedge b = a$

$a \wedge a = a$

$(a \wedge b) \le a$

$(a \wedge b) \le b$

$(a \wedge T) = a$

$(a \wedge \bot) = \bot$

Monotone Transfer Function

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

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

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 $f_b(T) = \bot$ and $f_b(\bot)=T$, then solutions might oscillate between T and ⊥ indefinitely.

Since $\bot \le T$, $f_b(\bot)$ should be $\le f_b(T)$. But $f_b(\bot) = T$ which is not $\le f_b(T) = \bot$. 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:

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

$a \cap a = a$

$(a \cap b) \subseteq a$

$(a \cap b) \subseteq b$

$(a \cap N) = a$

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

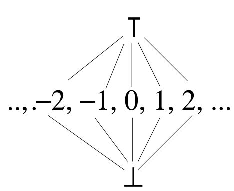
Also f_Z is monotone since

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

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 $t'(v)=t(v)\ (\forall v)$

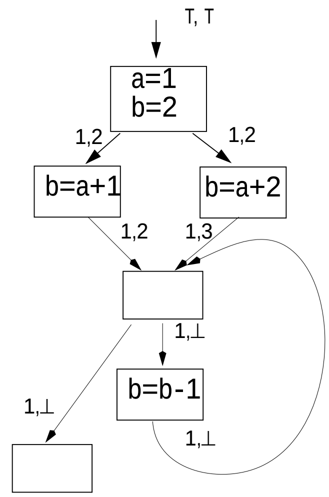
(2) For each assignment statement
$v = e(w_1,w_2,...,w_n)$
in b, in order of execution, do:
If any $t'(w_i) = \perp$ ($1 \le i \le n$)
Then set $t'(v) = \perp$ (strictness)
Elsif any $t'(w_i) = T$ ($1 \le i \le n$)
Then set $t'(v) = T$ (delay eval of v)
Else $t'(v) = e(t'(w_1),t'(w_2),...)$

(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.