Computing Dominators

Dominators can be computed as a Set-valued Forward Data Flow Problem.

If a node N dominates all of node M's predecessors, then N appears on all paths to M. Hence (N dom M).

Similarly, if M *doesn't* dominate all of M's predecessors, then there is a path to M that doesn't include M. Hence $\neg$(N dom M).

These observations give us a "data flow equation" for dominator sets:

$$dom(N) = \{N\} \cup \bigcap_{M \in Pred(N)} dom(M)$$

The analysis domain is the lattice of all subsets of nodes. Top is the set of all nodes; bottom is the empty set. The ordering relation is subset.

The meet operation is intersection.

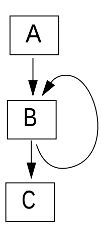
The Initial Condition is that $DomIn(b_0) = \phi$

$$DomIn(b) = \bigcap_{c \in Pred(b)} DomOut(c)$$

$$DomOut(b) = DomIn(b) \cup \{b\}$$

Loops Require Care

Loops in the Control Flow Graph induce circularities in the Data Flow equations for Dominators. In



we have the rule dom(B) =
DomOut(B) =
DomIn(B) U {B} =
{B} U (<u>DomOut(B)</u> $\cap$ DomOut(A))

If we choose DomOut(B) = ϕ initially, we get DomOut(B) =
{B} U (ϕ $\cap$ DomOut(A)) = {B}
which is *wrong*.

Instead, we should use the Universal Set (of all nodes) which is the identity for $\cap$.

Then we get DomOut(B) =
{B} U ({all nodes} $\cap$ DomOut(A)) =
{B} U DomOut(A)
which is correct.

A Worklist Algorithm for Dominators

The data flow equations we have developed for dominators can be evaluated using a simple Worklist Algorithm.

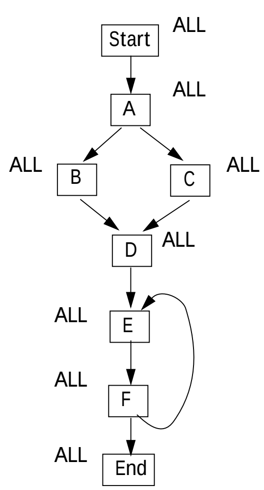
Initially, each node's dominator set is set to the set of all nodes. We add the start node to our worklist.

For each node on the worklist, we reevaluate its dominator set. If the set changes, the updated dominator set is used, and all the node's successors are added to the worklist (so that the updated dominator set can be propagated).

The algorithm terminates when the worklist becomes empty, indicating that a stable solution has been found.

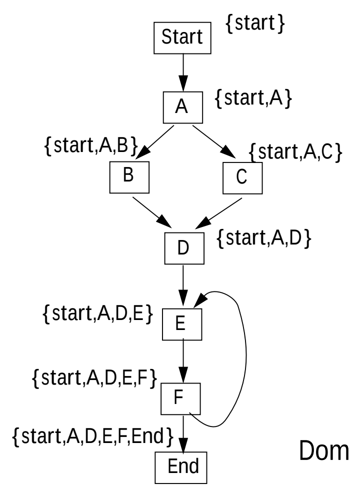
Compute Dominators(){
  For (each $n \in$ NodeSet)
      Dom(n) = NodeSet
  WorkList = {StartNode}
  While (WorkList $\neq \phi$) {
    Remove any node Y from WorkList
    New = $\{Y\} \cup \bigcap_{X \in Pred(Y)} Dom(X)$
    If New $\neq$ Dom(Y) {
      Dom(Y) = New
      For (each $Z \in$ Succ(Y))
          WorkList = WorkList U {Z}
}}}

Example

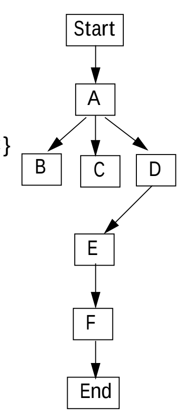


Initially the WorkList = {Start}.

Be careful when Pred(Node) = ϕ.



Control Flow Graph

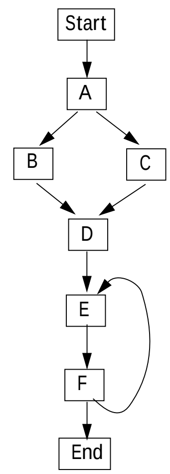


Dominator Tree

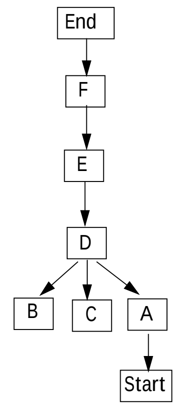
Postdominance

A block Z *postdominates* a block Y (Z pdom Y) if and only if all paths from Y to an exit block must pass through Z. Notions of immediate postdominance and a postdominator tree carry over.

Note that if a CFG has a single exit node, then postdominance is equivalent to dominance if flow is reversed (going from the exit node to the start node).



Control Flow Graph



Postdominator Tree

Dominance Frontiers

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

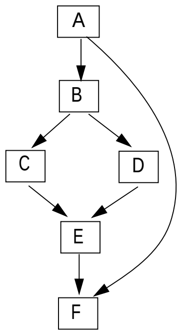
It is interesting to consider blocks "just before" or "just after" blocks we're dominated by, or blocks we dominate.

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.

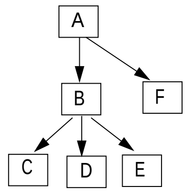
$$DF(N) = \{Z \mid M \rightarrow Z \ \& \ (N \text{ dom } M) \ \& \ \neg(N \text{ sdom } Z)\}$$

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



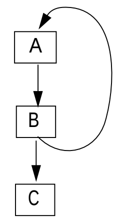
Control Flow Graph



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 | M→Z & (N dom M) &
¬(N sdom Z)}

Now B is dominated by A and B→A.

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

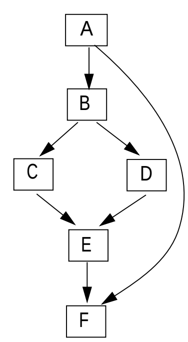
Postdominance Frontiers

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.

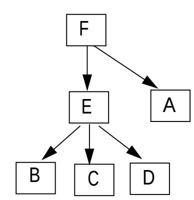
PDF(N) =
{Z | Z→M & (N pdom M) &
¬(N pdom Z)}

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.

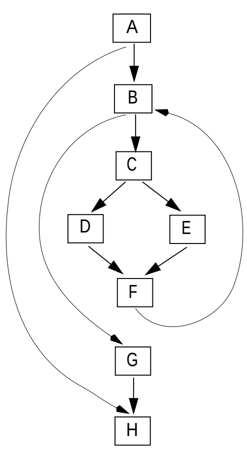
(b) Y does not postdominate all successors of X.

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

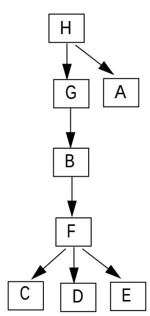
Control Dependence Graph

We can build a *Control Dependence Graph* that shows (in graphical form) all Control Dependence relations.

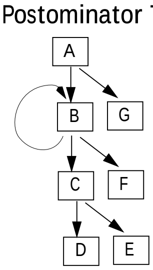
(A Block *can be* Control Dependent on itself.)



Control Flow Graph



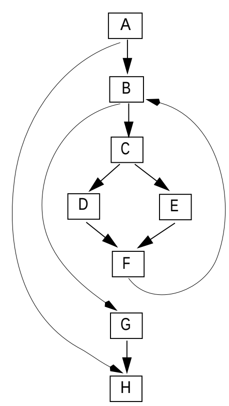
Postominator Tree



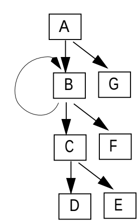
Control Dependence Graph

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!

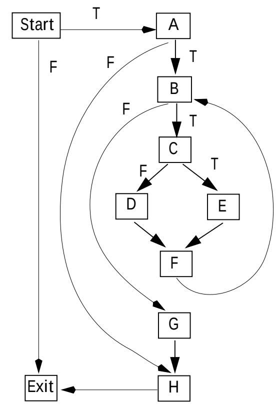
C and F *are* control equivalent, but D and E are *mutually exclusive*!

Improving the Representation of Control Dependence

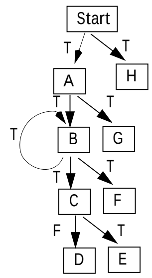
We can label arcs in the CFG and the CD Graph with the condition (T or F or some switch value) that caused the arc to be selected for execution.

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 $\perp$ (“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 $\wedge$:

$a \leq b \Leftrightarrow a \wedge b = a$

$a \wedge a = a$

$(a \wedge b) \leq a$

$(a \wedge b) \leq b$

$(a \wedge T) = a$

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

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.

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

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

$\wedge$ 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

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