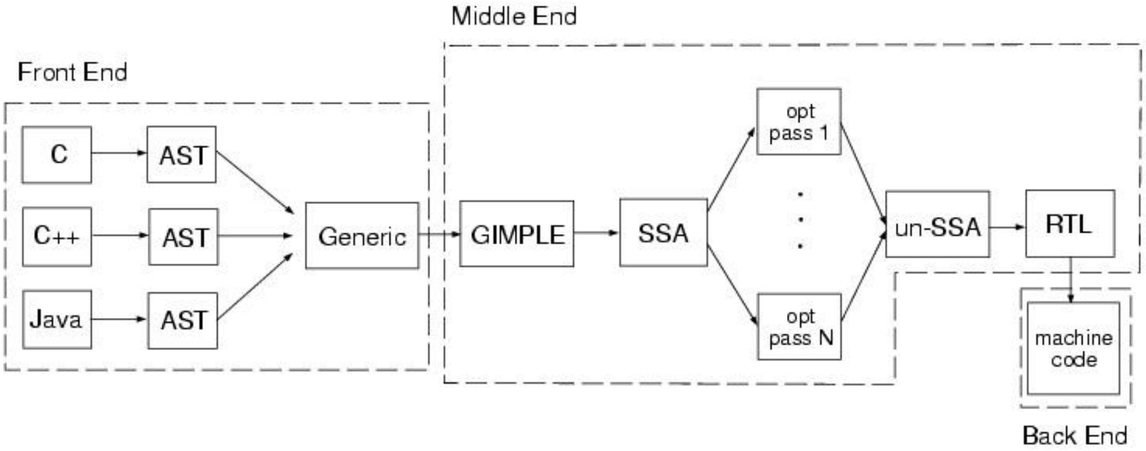
An Overview of GCC Architecture (source: wikipedia)



Control-Flow Analysis and Loop Detection

Last time

- Lattice-theoretic framework for data-flow analysis

Today

- Control-flow analysis
- Loops
- Identifying loops using dominators
- Converting to SSA using dominators
- Dominators and PA2

Context

Data-flow

- Flow of data values from defs to uses
- Could alternatively be represented as a data dependence

Control-flow

- Sequencing of operations
- Could alternatively be represented as a control dependence
- *e.g.,* Evaluation of then-code and else-code depends on if-test

Why study control flow analysis?

Finding Loops

- most computation time is spent in loops
- to optimize them, we need to find them

Loop Optimizations

- Loop-invariant code hoisting
- Induction variable elimination
- Array bounds check removal
- Loop unrolling
- Parallelization
- ...

Identifying structured control flow

- can be used to speed up data-flow analysis

Representing Control-Flow

High-level representation

– Control flow is implicit in an AST

Low-level representation:

– Use a **Control-flow graph**
  – Nodes represent statements
  – Edges represent explicit flow of control

Other options

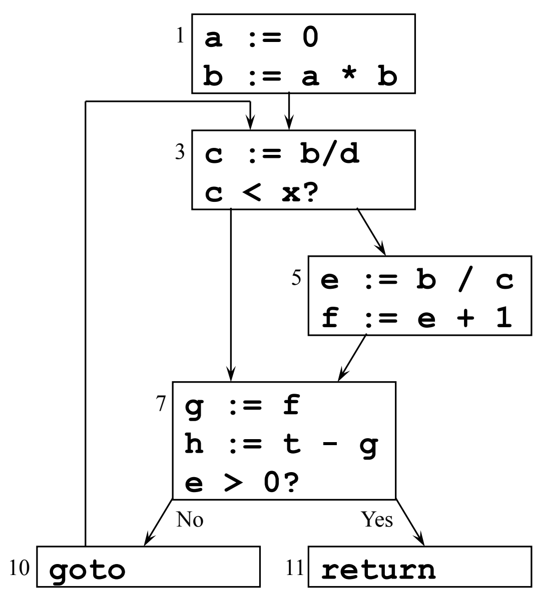
– Control dependences in program dependence graph (PDG) [Ferrante87]
– Dependences on explicit state in value dependence graph (VDG) [Weise 94]

What Is Control-Flow Analysis?

Control-flow analysis discovers the flow of control within a procedure
(*e.g.,* builds a CFG, identifies loops)

Example

```
    a := 0
    b := a * b
L1: c := b/d
    if c < x goto L2
    e := b / c
    f := e + 1
L2: g := f
    h := t - g
    if e > 0 goto L3
goto L1
L3: return
```



Loop Concepts

Loop: Strongly connected subgraph of CFG with a single entry point (header)

Loop entry edge: Source not in loop & target in loop

Loop exit edge: Source in loop & target not in loop

Loop header node: Target of loop entry edge. Dominates all nodes in loop.

Back edge: Target is loop header & source is in the loop

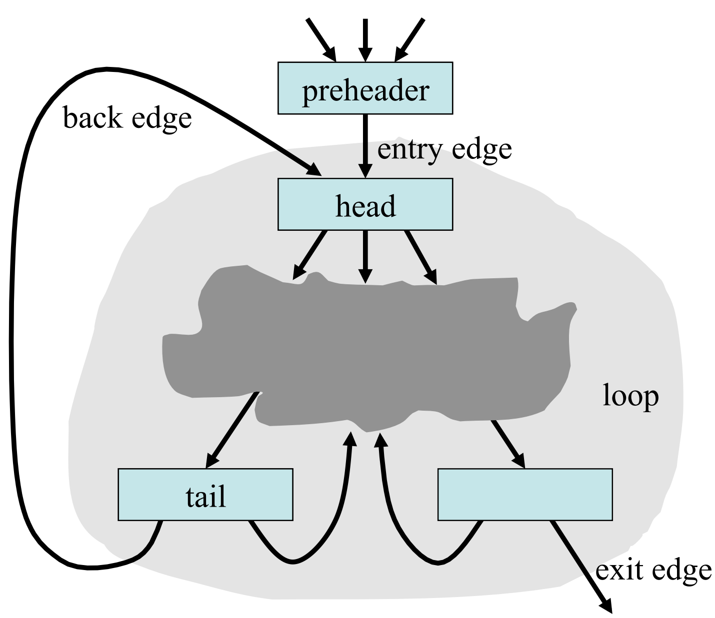
Natural loop: Associated with each back edge. Nodes dominated by header and with path to back edge without going through header

Loop tail node: Source of back edge

Loop preheader node: Single node that's source of the loop entry edge

Nested loop: Loop whose header is inside another loop

Picturing Loop Terminology



The Value of Preheader Nodes

Not all loops have preheaders

– Sometimes it is useful to create them

Without preheader node

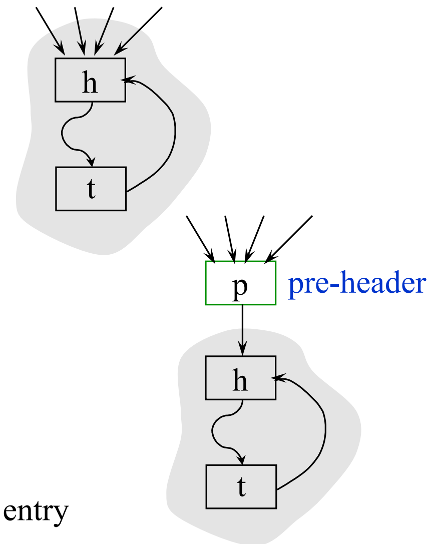
– There can be multiple entry edges

With single preheader node

– There is only one entry edge

Useful when moving code outside the loop

– Don’t have to replicate code for multiple entry edges



Identifying Loops

Why?

– Most execution time spent in loops, so optimizing loops will often give most benefit

Many approaches

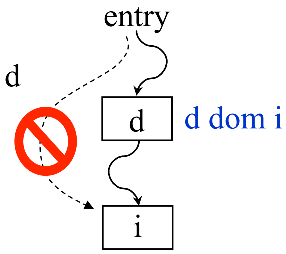
– Interval analysis
  – Exploit the natural hierarchical structure of programs
  – Decompose the program into nested regions called intervals
– Structural analysis: a generalization of interval analysis
– Identify **dominators** to discover loops

We’ll focus on the dominator-based approach

Dominator Terminology

Dominators

d **dom** i if all paths from entry to node i include d

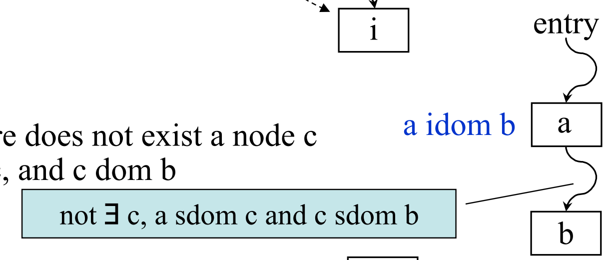


Strict dominators

d **sdom** i if d dom i and d ≠ i

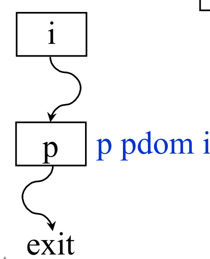
Immediate dominators

a **idom** b if a **sdom** b and there does not exist a node c such that c ≠ a, c ≠ b, a dom c, and c dom b



Post dominators

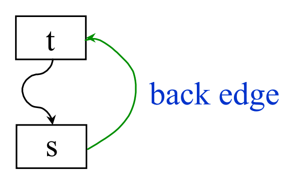
p **pdom** i if every possible path from i to exit includes p (p dom i in the flow graph whose arcs are reversed and entry and exit are interchanged)



Identifying Natural Loops with Dominators

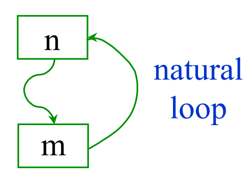
Back edges

A **back edge** of a natural loop is one whose target dominates its source



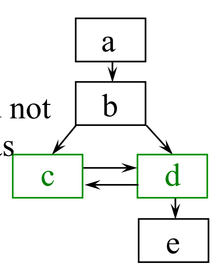
Natural loop

The **natural loop** of a back edge (m→n), where n dominates m, is the set of nodes x such that n dominates x and there is a path from x to m not containing n

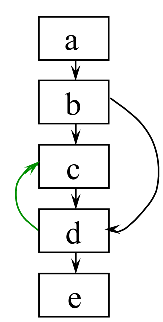


Example

SCC with c and d not a loop because has two entry points



The target, c, of the edge (d→c) does not dominate its source, d, so (d→c) does not define a natural loop



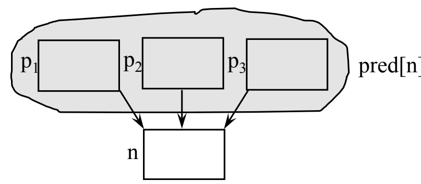
Computing Dominators

Input: Set of nodes N (in CFG) and an entry node s
Output: Dom[i] = set of all nodes that dominate node i

Dom[s] = {s}
for each n $\in$ N – {s}
    Dom[n] = N
repeat
    change = false
    for each n $\in$ N – {s}
        D = {n} $\cup$ ($\cap_{p \in pred(n)}$ Dom[p])
        if D $\neq$ Dom[n]
            change = true
            Dom[n] = D
until !change

Key Idea

If a node dominates all predecessors of node n, then it also dominates node n

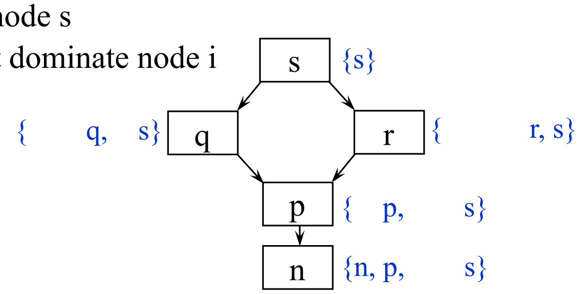


$x \in Dom(p_1) \wedge x \in Dom(p_2) \wedge x \in Dom(p_3) \Rightarrow x \in Dom(n)$

Computing Dominators (example)

Input: Set of nodes N and an entry node s
Output: Dom[i] = set of all nodes that dominate node i

Dom[s] = {s}
for each n $\in$ N – {s}
    Dom[n] = N
repeat
    change = false
    for each n $\in$ N – {s}
        D = {n} $\cup$ ($\cap_{p \in pred(n)}$ Dom[p])
        if D $\neq$ Dom[n]
            change = true
            Dom[n] = D
until !change



Initially

Dom[s] = {s}
Dom[q] = {n, p, q, r, s}. . .

Finally

Dom[q] = {q, s}
Dom[r] = {r, s}
Dom[p] = {p, s}
Dom[n] = {n, p, s}

Recall SSA, Another use of dominator information

Advantage

- Allow analyses and transformations to be simpler & more efficient/effective

Disadvantage

- May not be “executable” (requires extra translations to and from)
- May be expensive (in terms of time or space)

Process

Static Single Assignment (SSA) Form

Idea

- Each variable has only one static definition
- Makes it easier to reason about values instead of variables
- Similar to the notion of functional programming

Transformation to SSA

- Rename each definition
- Rename all uses reached by that assignment

Example

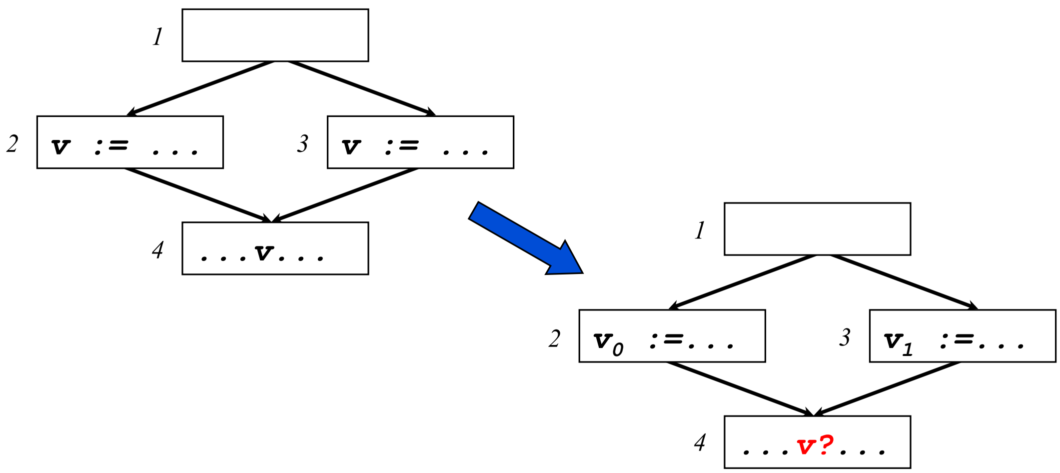
	→	
`v := ...`		v_0 `:= ...`
`... := ... v ...`		`... := ...` v_0 `...`
`v := ...`		v_1 `:= ...`
`... := ... v ...`		`... := ...` v_1 `...`

What do we do when there’s control flow?

SSA and Control Flow

Problem

– A use may be reached by several definitions

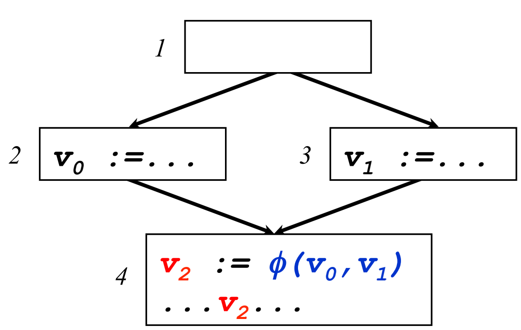


SSA and Control Flow (cont)

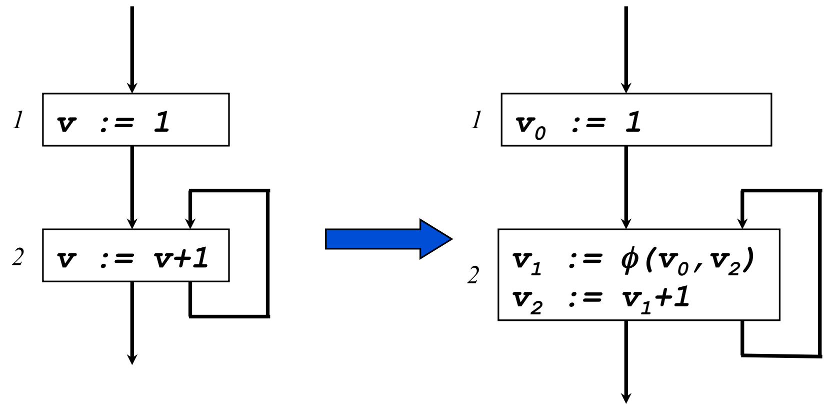
Merging Definitions

– ϕ-functions merge multiple reaching definitions

Example



Another Example



Transformation to SSA Form

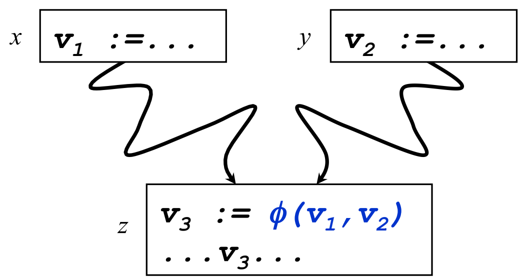
Two steps

- Insert ϕ-functions
- Rename variables

Where Do We Place ϕ-Functions?

Basic Rule

– If two distinct (non-null) paths x→z and y→z converge at node z, and nodes x and y contain definitions of variable v, then a ϕ-function for v is inserted at z

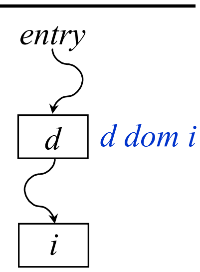


Machinery for Placing ϕ-Functions

Recall Dominators

– d **dom** i if all paths from entry to node i include d

– d **sdom** i if d dom i and d≠i



Dominance Frontiers

– The **dominance frontier** of a node d is the set of nodes that are "just barely" not dominated by d; i.e., the set of nodes n, such that

  – d dominates a predecessor p of n, and

  – d does **not** strictly dominate n

– DF(d) = {n | ∃p∈pred(n), d dom p and d !sdom n}

Notational Convenience

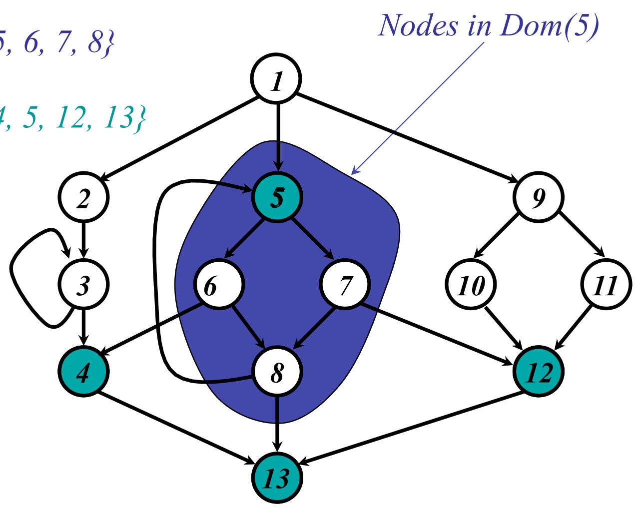
– $DF(S) = \bigcup_{n \in S} DF(n)$

Dominance Frontier Example

DF(d) = {n | ∃p∈pred(n), d dom p and d !sdom n}

Dom(5) = {5, 6, 7, 8}

DF(5) = {4, 5, 12, 13}



What's significant about the Dominance Frontier?

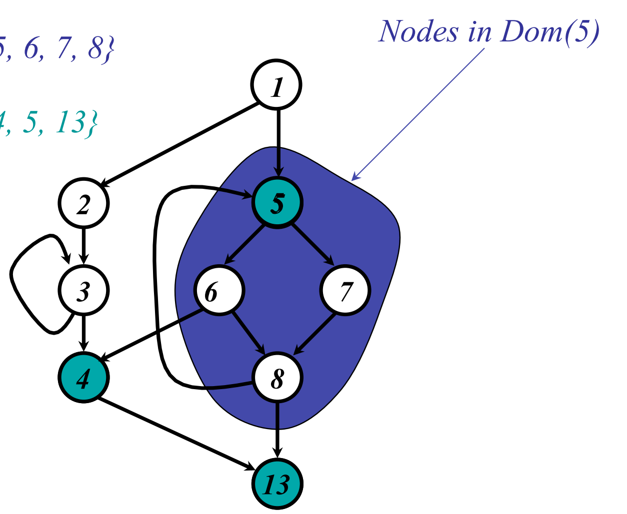
In SSA form, definitions must dominate uses

Dominance Frontier Example II

DF(d) = {n | ∃p∈pred(n), d dom p and d !sdom n}

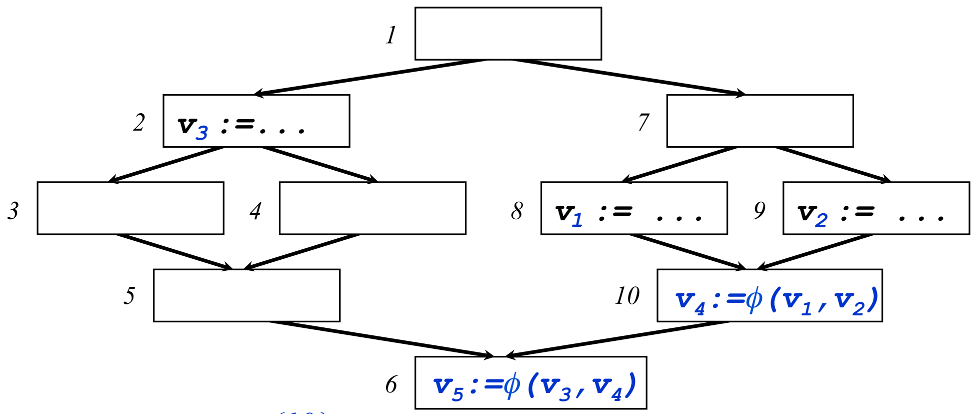
Dom(5) = {5, 6, 7, 8}

DF(5) = {4, 5, 13}



In this graph, node 4 is the first point of convergence between the entry and node 5, so do we need a ϕ- function at node 13?

SSA Exercise



DF(8) = {10}

DF(9) = {10}

DF(2) = {6} DF(d) = {n | ∃p∈pred(n), d dom p and d !sdom n}

DF({8,9}) = {10}

DF(10) = {6}

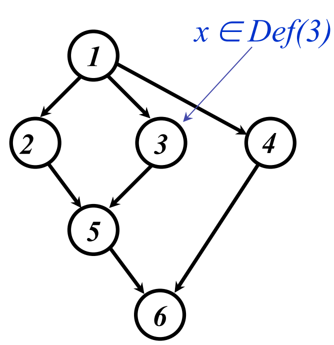
DF({2,8,9,6,10}) = {6,10}

See http://www.hipersoft.rice.edu/grads/publications/dom14.pdf for a more thorough description of DF.

Dominance Frontiers Revisited

Suppose that node 3 defines variable x

DF(3) = {5}



Do we need to insert a φ- function for x anywhere else?

Yes. At node 6. Why?

Dominance Frontiers and SSA

Let

– $DF_1(S) = DF(S)$

– $DF_{i+1}(S) = DF(S \cup DF_i(S))$

Iterated Dominance Frontier

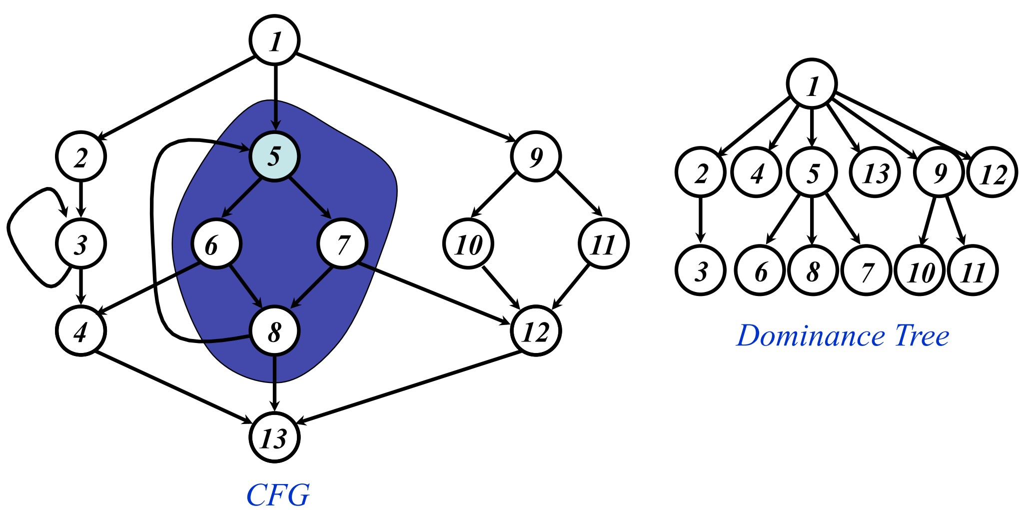
– $DF_\infty(S)$

Theorem

– If S is the set of CFG nodes that define variable v, then $DF_\infty(S)$ is the set of nodes that require ϕ-functions for v

Dominance Tree Example

The dominance tree shows the dominance relation



Inserting Phi Nodes

Calculate the dominator tree

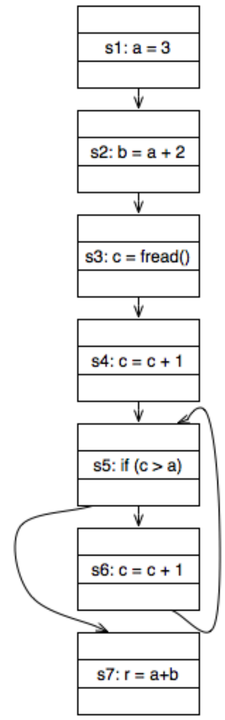
– a lot of research has gone into calculating this quickly

Computing dominance frontier from dominator tree

– $DF_{local}[n]$= successors of n (in CFG) that are not strictly dominated by n

– $DF_{up}[n]$= nodes in the dominance frontier of n that are not strictly dominated by n's immediate dominator

– $DF[n] = DF_{local}[n] \cup \bigcup_{c \in children[n]} DF_{up}[c]$



Algorithm for Inserting ϕ-Functions

for each variable v
    WorkList ← ∅
    EverOnWorkList ← ∅
    AlreadyHasPhiFunc ← ∅
    for each node n containing an assignment to v   *Put all defs of v on the worklist*
        WorkList ← WorkList ∪ {n}
    EverOnWorkList ← WorkList
    while WorkList ≠ ∅
        Remove some node n for WorkList
        for each d ∈ DF(n)
            if d ∉ AlreadyHasPhiFunc   *Insert at most one ϕ function per node*
                Insert a ϕ-function for v at d
                AlreadyHasPhiFunc ← AlreadyHasPhiFunc ∪ {d}
                if d ∉ EverOnWorkList   *Process each node at most once*
                    WorkList ← WorkList ∪ {d}
                    EverOnWorkList ← EverOnWorkList ∪ {d}

Transformation to SSA Form

Two steps

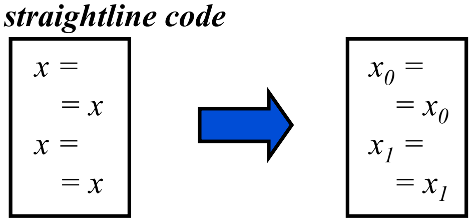
- Insert ϕ-functions
- Rename variables

Variable Renaming

Basic idea

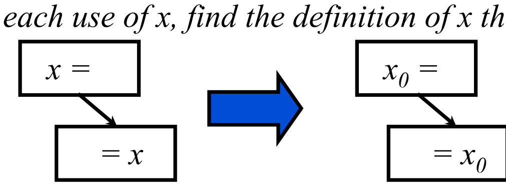
- When we see a variable on the LHS, create a new name for it
- When we see a variable on the RHS, use appropriate subscript

Easy for straightline code



Use a stack when there's control flow

- *For each use of x, find the definition of x that dominates it*



Traverse the dominance tree

Variable Renaming (cont)

Data Structures

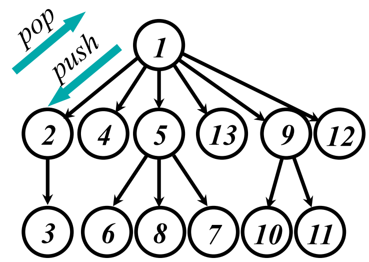
– Stacks[v] ∀v
  Holds the subscript of most recent definition of variable v, initially empty

– Counters[v] ∀v
  Holds the current number of assignments to variable v; initially 0

Auxiliary Routine

procedure GenName(variable v)

```
i := Counters[v]
push i onto Stacks[v]
Counters[v] := i + 1
```



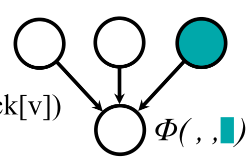
Use the Dominance Tree to remember the most recent definition of each variable

Variable Renaming Algorithm

Call Rename(entry-node)

procedure Rename(block b)
  if b previously visited return

  for each statement s in b (in order)
    for each variable v ∈ RHS(s) (except for ϕ-functions)
      replace v by v_i, where i = Top(Stacks[v])
    for each variable v ∈ LHS(s)
      GenName(v) and replace v with v_i, where i=Top(Stack[v])
  for each s ∈ succ(b) (in CFG)
    j ← position in s' s ϕ-function corresponding to block b
    for each ϕ-function p in s
      replace the j^{th} operand of RHS(p) by v_i, where i = Top(Stack[v])



  for each s ∈ child(b) (in DT)
    Rename(s)

Recurse using Depth First Search

  for each ϕ-function or statement t in b
    for each v_i ∈ LHS(t)
      Pop(Stack[v])

Unwind stack when done with this node

Transformation from SSA Form

Proposal

– Restore original variable names (*i.e.*, drop subscripts)
– Delete all ϕ-functions

Complications (the proposal doesn't work!)

–*What if versions get out of order? (simultaneously live ranges)*

x_0 =
x_1 =
= x_0
= x_1

Alternative

–*Perform dead code elimination (to prune ϕ-functions)*
–*Replace ϕ-functions with copies in predecessors*
–*Rely on register allocation coalescing to remove unnecessary copies*

PA2 and Dominators

Why might you be getting 'Instruction does not dominate all uses!' error?

Next Time

Reading

- Advanced Compiler Optimizations for Supercomputers by Padua and Wolfe

Lecture

- Dependencies in loops
- Parallelization and Performance Optimization of Applications