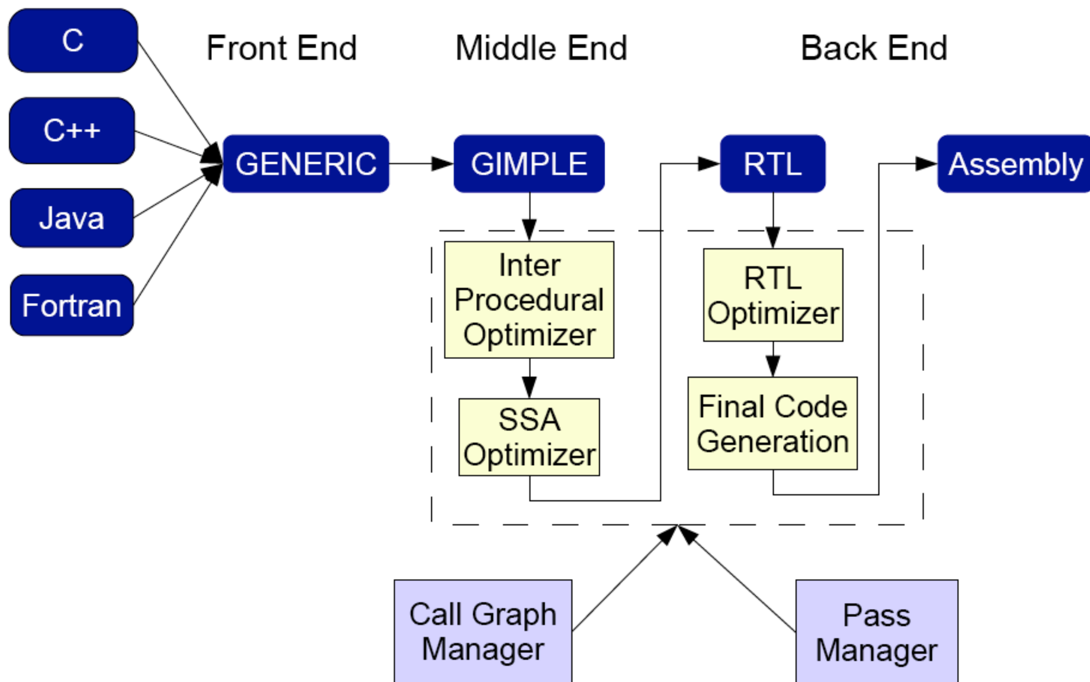
GCC Compiler Overview



From **GCC - An Architectural Overview**, D. Novillo, September 2006.

GCC Overview cont...

SSA Optimizers

- vectorization
- loop optimizations
- scalar optimizations: CCP, DCE, DSE, FRE, PRE, VRP, SRA
- field-sensitive, points-to alias analysis

RTL Optimizers

- RTL has infinite registers
- register allocation
- scheduling, SW pipelining, CSE, ...

From **GCC - An Architectural Overview**, D. Novillo, September 2006.

LLVM Compiler Infrastructure

Source: "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation" by Lattner and Adve

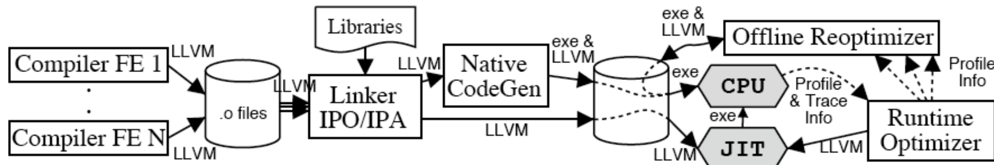


Figure 4: *LLVM system architecture diagram*

Goals

- lifelong analysis and optimization
- modular compiler components

IR is low level, control-flow graph and SSA

- language-independent types: any-bit ints, float, double, pointers, arrays, structures, functions

-interprocedural analysis and transformation

Status

-Apple is paying main developer (Chris Lattner) to continue development

-built OpenGL JIT compiler with it in two weeks

Static Single Assignment Form

Last Time

– Induction variable detection and elimination and strength reduction

Today

– Program representations

– Static single assignment (SSA) form

  – Program representation for sparse data-flow

– Conversion to and from SSA

Next Time

– Applications of SSA

Data Dependence

Definition

– Data dependences are constraints on the order in which statements may be executed

We say statement s_2 depends on s_1

– **Flow (true) dependence**: s_1 writes memory that s_2 later reads (RAW)
– **Anti-dependence**: s_1 reads memory that s_2 later writes (WAR)
– **Output dependences**: s_1 writes memory that s_2 later writes (WAW)
– **Input dependences**: s_1 reads memory that s_2 later reads (RAR)

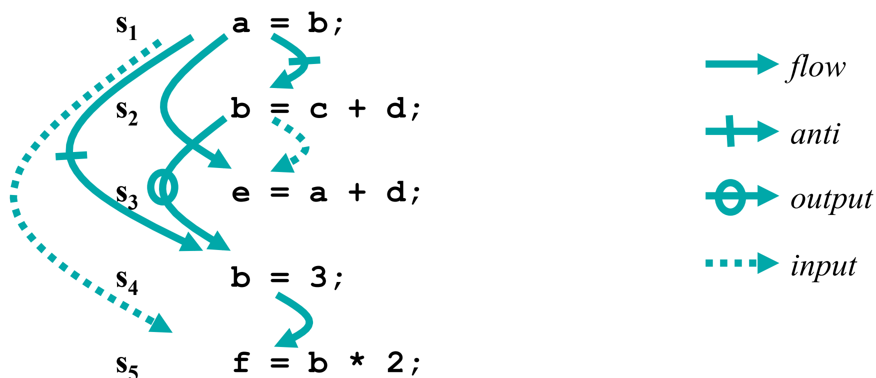
True dependences

– Flow dependences represent actual flow of data

False dependences

– Anti- and output dependences reflect reuse of memory, not actual data flow; can often be eliminated

Example



Representing Data Dependences

Implicitly

– Using variable defs and uses
– Pros: simple
– Cons: hides data dependence (analyses must find this info)

Def-use chains (du chains)

– Link each def to its uses
– Pros: explicit; therefore fast
– Cons: must be computed and updated, space consuming

Alternate representations

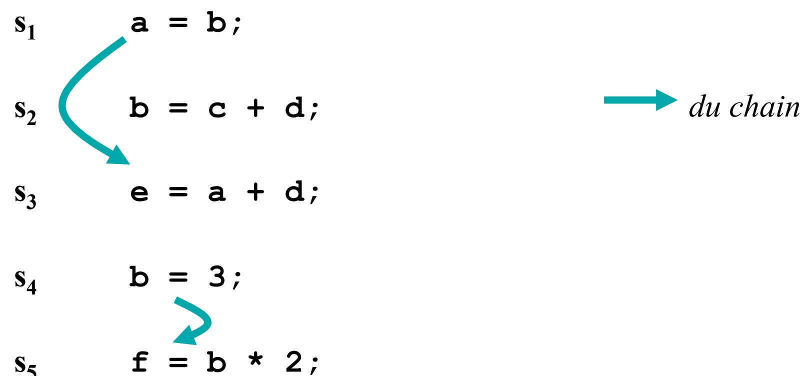
– *e.g.,* Static single assignment form (SSA), Program Dependence Graph (PDG), dependence flow graphs (DFG), value dependence graphs (VDG),

DU Chains

Definition

– du chains link each def to its uses

Example



UD Chains

Definition

– ud chains link each use to its defs

Example

```
s1    a = b;
s2    b = c + d;
s3    e = a + d;
s4    b = 3;
s5    f = b * 2;
```

ud chain

Role of Alternate Program Representations

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

Original Code (RTL)
Optimized Code (RTL)
SSA Code1
T1
SSA Code2
T2
SSA Code3

Static Single Assignment (SSA) Form

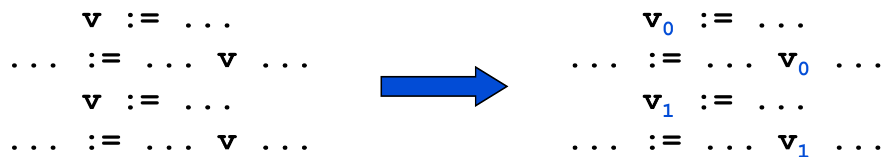
Idea

- 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

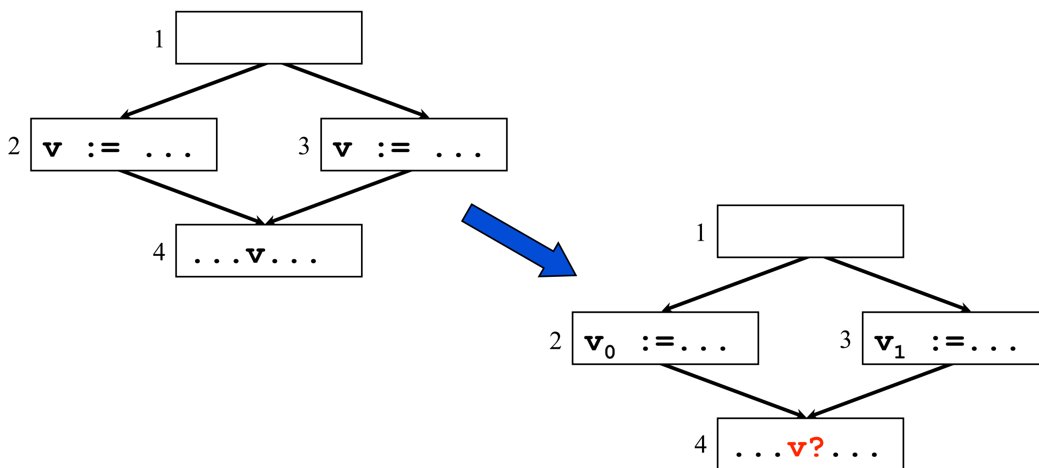


What do we do when there's control flow?

SSA and Control Flow

Problem

- A use may be reached by several definitions

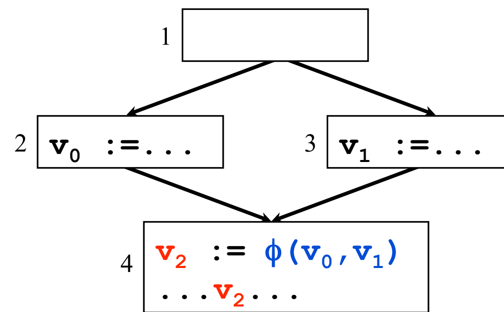


SSA and Control Flow (cont)

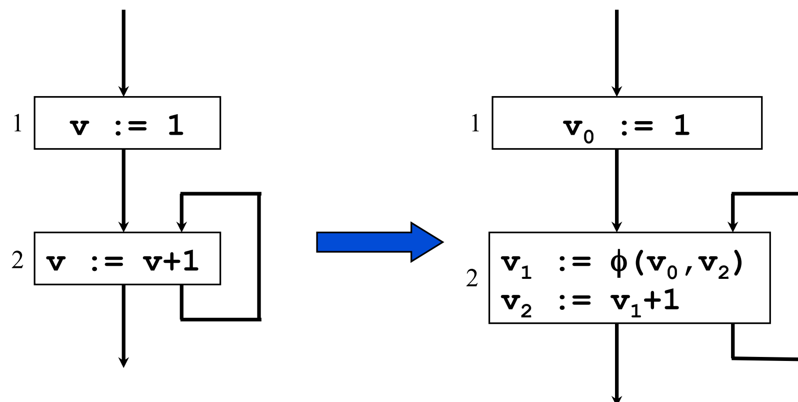
Merging Definitions

– ϕ-functions merge multiple reaching definitions

Example



Another Example



SSA vs. ud/du Chains

SSA form is more constrained

Advantages of SSA

- More compact
- Some analyses become simpler when each use has only one def
- Value merging is explicit
- Easier to update and manipulate?

Furthermore

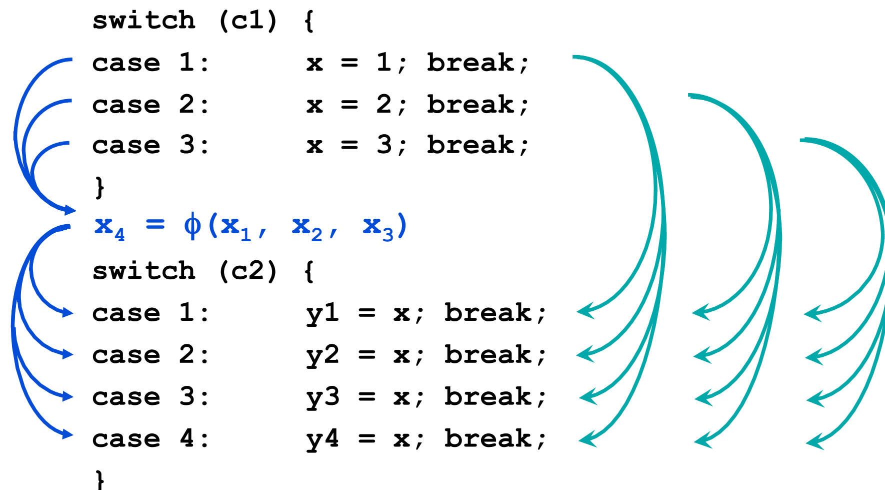
- Eliminates false dependences (simplifying context)

```
for (i=0; i<n; i++)
     A[i] = i;
for (i=0; i<n; i++)
     print(foo(i));
```

Unrelated uses of **i** are given different variable names

SSA vs. ud/du Chains (cont)

Worst case du-chains?



m defs and n uses leads to m×n du chains

Transformation to SSA Form

Two steps

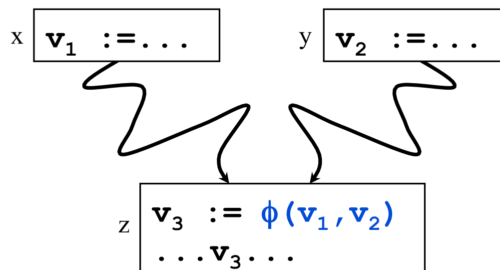
– Insert ϕ-functions

– 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



Approaches to Placing ϕ-Functions

Minimal

– As few as possible subject to the basic rule

Briggs-Minimal

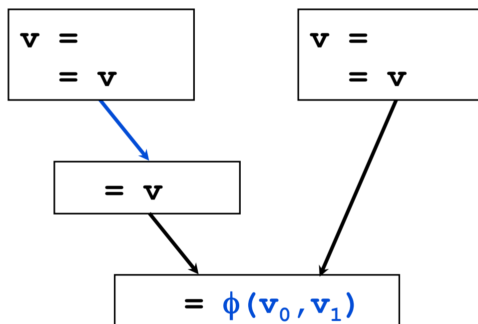
– Same as minimal, except v must be live across some edge of the CFG

Pruned

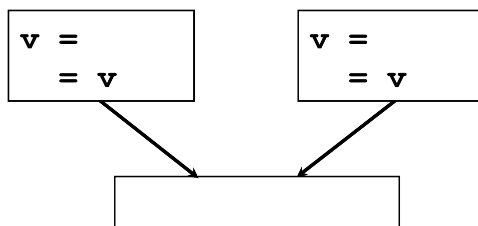
– Same as minimal, except dead ϕ-functions are not inserted

What's the difference between Briggs Minimal and Pruned SSA?

Briggs Minimal vs. Pruned



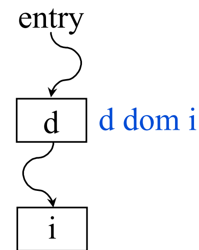
Briggs Minimal will add a ϕ function because v is live across the blue edge, but Pruned SSA will not because the ϕ function is dead



Neither Briggs Minimal nor Pruned SSA will place a ϕ function in this case because v is not live across any CFG edge

Why would we ever use Briggs Minimal instead of Pruned SSA?

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

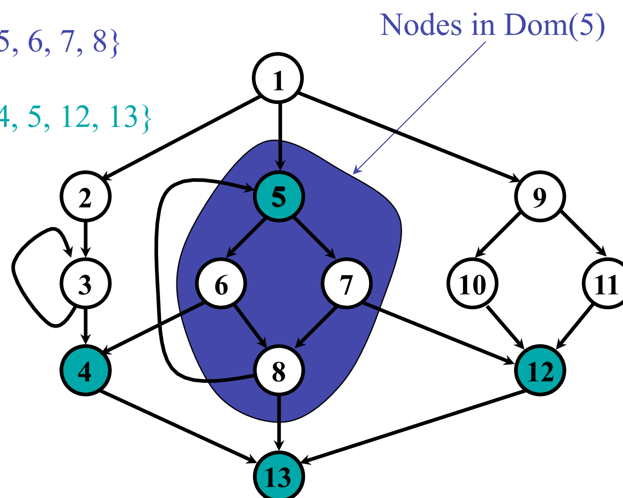
- $DF(S) = \bigcup_{n\in S} DF(n)$

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?

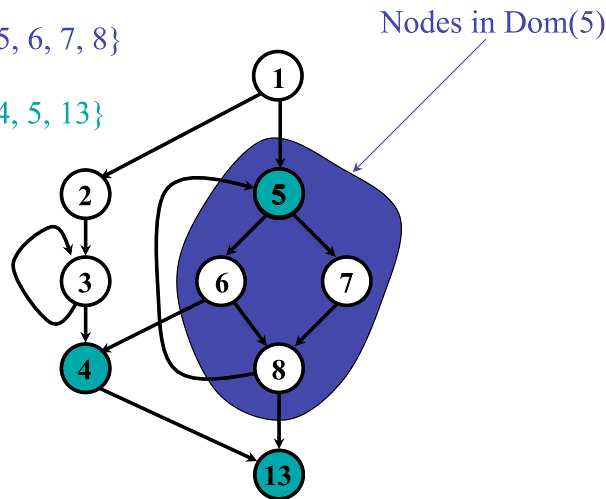
In SSA form, definitions must dominate uses

Dominance Frontier Example II

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

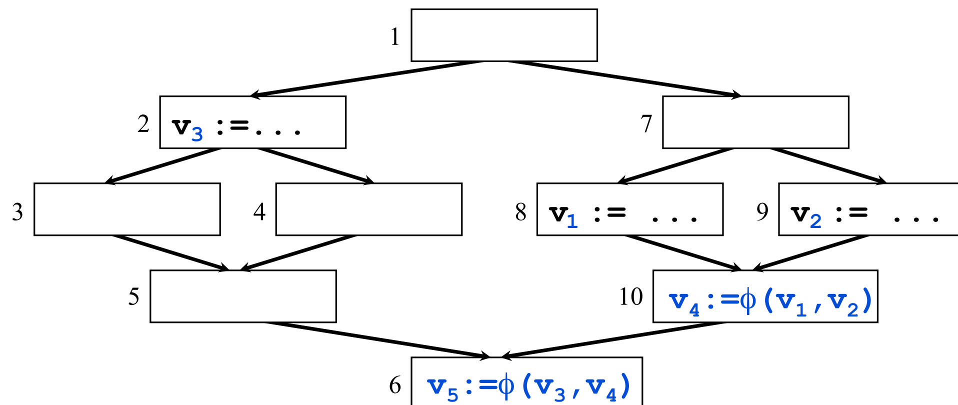
Dom(5) = {5, 6, 7, 8}

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}

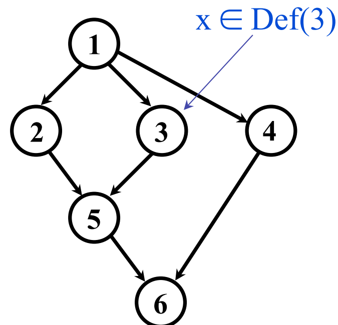
DF({2,8,9,6,10}) = {6,10}

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

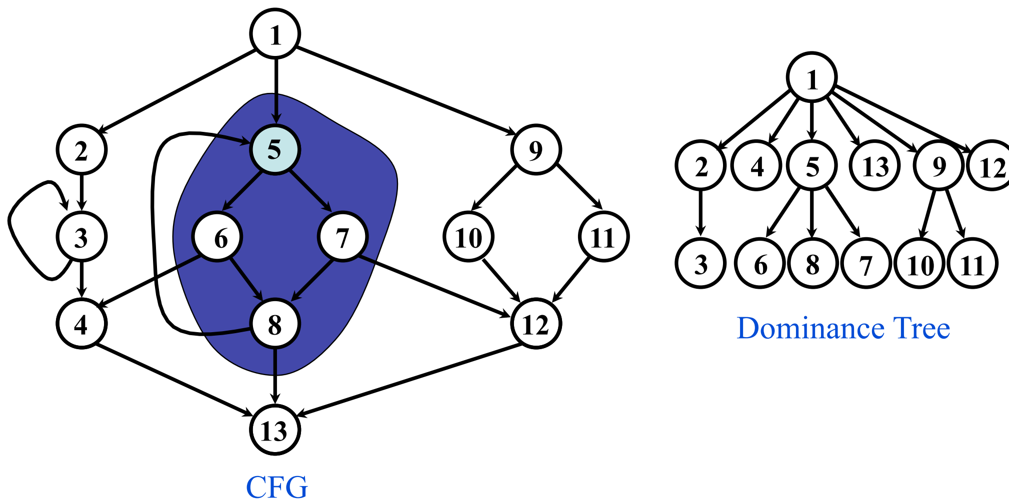
- $DF_\infty(S)$

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

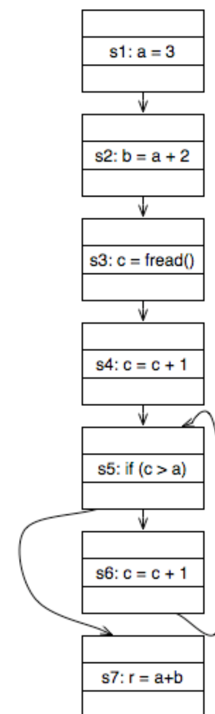
– a lot of research has gone into calculating this quickly

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

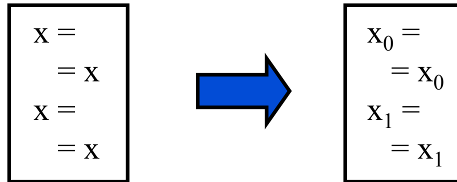
- Insert ϕ-functions
- Rename variables

Variable Renaming

Basic idea

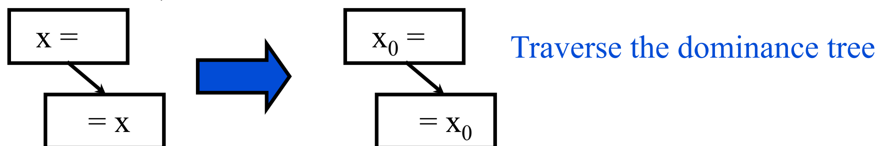
- When we see a variable on the LHS, create a new name for it
- When we see a variable on the RHS, use appropriate subscript

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



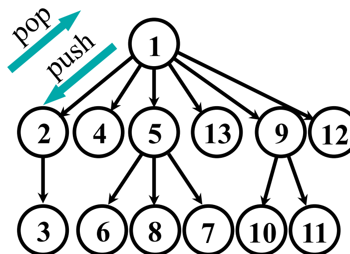
Variable Renaming (cont)

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



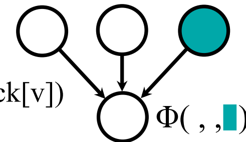
Use the Dominance Tree to remember the most recent definition of each variable

Variable Renaming Algorithm

procedure Rename(block b)
  if b previously visited return

Call Rename(entry-node)

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



  for each $s \in$ child(b) (in DT)
    Rename(s)

} Recurse using Depth First Search

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

} Unwind stack when done with this node

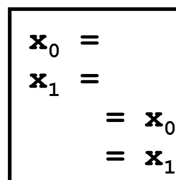
Transformation from SSA Form

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)



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

Concepts

Data dependences

- Three kinds of data dependences
- du-chains

Alternate representations

SSA form

Conversion to SSA form

- ϕ-function placement
- variable renaming

Next Time

Assignments

- Read Alpern and Zadeck paper on value numbering

Lecture

- Using SSA to apply optimizations