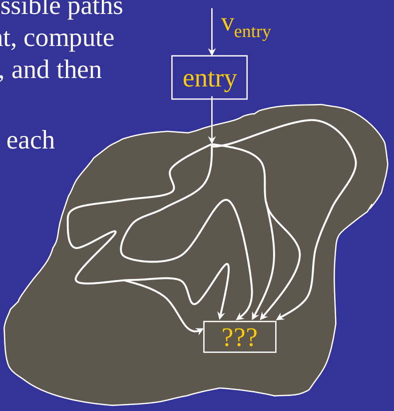
Recall the MOP Solution

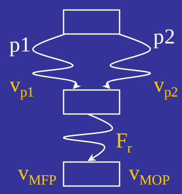
Goal

- For a forward problem, consider all possible paths from the entry to a given program point, compute the flow values at the end of each path, and then meet these values together
- Meet-over-all-paths (MOP) solution at each program point
- $\sqcap_{\text{all paths n1, n2, ..., ni}} (f_{ni}(\ldots f_{n2}(f_{n1}(v_{entry}))))$



Legality

"Is v_{MFP} legal?" ≡ "Is $v_{MFP} \sqsubseteq v_{MOP}$?"



Look at Merges

- $v_{MOP} = F_r(v_{p1}) \sqcap F_r(v_{p2})$
- $v_{MFP} = F_r(v_{p1} \sqcap v_{p2})$
- $v_{MFP} \sqsubseteq v_{MOP} \equiv F_r(v_{p1} \sqcap v_{p2}) \sqsubseteq F_r(v_{p1}) \sqcap F_r(v_{p2})$

Observation

$\forall x,y \in V$

$$f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y) \quad \Leftrightarrow \quad x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

$\therefore$ v_{MFP} legal when F_r (the flow functions) are monotonic

Monotonicity

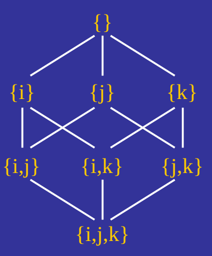
Monotonicity: $(\forall x,y \in V)[x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)]$

- If the flow function f is applied to two members of V, the result of applying f to the "lesser" of the two members will be under the result of applying f to the "greater" of the two
- Giving a flow function more conservative inputs leads to more conservative outputs (never more optimistic outputs)

Why else is monotonicity important?

For monotonic F over domain V

- The maximum number of times F can be applied to self w/o reaching a fixed point is height(V) – 1
- IDFA is guaranteed to terminate if the flow functions are monotonic and the lattice has finite height



Efficiency

Parameters

- n: Number of nodes in the CFG
- k: Height of lattice
- t: Time to execute one flow function

Complexity

- O(nkt)

Example

- Reaching definitions?

Accuracy

Distributivity

- $f(u \sqcap v) = f(u) \sqcap f(v)$
- $v_{MFP} \sqsubseteq v_{MOP} \equiv F_r(v_{p1} \sqcap v_{p2}) \sqsubseteq F_r(v_{p1}) \sqcap F_r(v_{p2})$
- If the flow functions are distributive, MFP = MOP

Examples

- Liveness?
- Reaching constants?

$f(u \sqcap v) = f(\{x=2,y=3\} \sqcap \{x=3,y=2\})$

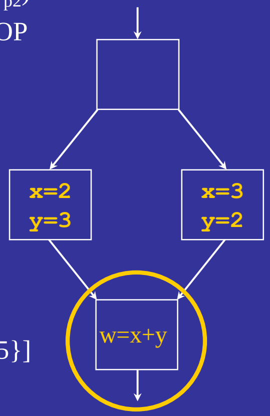
$= f(\varnothing) = \varnothing$

$f(u) \sqcap f(v) = f(\{x=2,y=3\}) \sqcap f(\{x=3,y=2\})$

$= [\{x=2,y=3,w=5\} \sqcap \{x=3,y=2,w=5\}]$

$= \{w=5\}$

$\Rightarrow$ MFP $\neq$ MOP



Concepts

Lattices

- Conservative approximation
- Optimistic (initial guess)
- Data-flow analysis frameworks
- Tuples of lattices

Data-flow analysis

- Fixed point
- Meet-over-all-paths (MOP)
- Maximum fixed point (MFP)
- Legal/safe (monotonic)
- Efficient
- Accurate (distributive)

Static Single Assignment Form

Last Time

- Lattice theoretic framework for data-flow analysis

Today

- Program representations
- Static single assignment (SSA) form
  - Program representation for sparse data-flow
- Conversion to and from SSA

Next Time

- Reuse optimizations

February 11, 2015 Static Single Assignment Form 8

Data Dependence

Definition

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

Types of dependences

- **Flow dependence**: s_1 writes memory that s_2 later reads (RAW)
  s1: `x = 17`
  s2: `print (x)`
- **Anti-dependence**: s_1 reads memory that s_2 later writes (WAR)
  s1: `print (x)`
  s2: `x = 18`
- **Output dependences**: s_1 writes memory that s_2 later writes (WAW)
  s1: `x = 19`
  s2: `x = 20`

February 11, 2015 Static Single Assignment Form 9

Data Dependence (cont)

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

s1: `print (x)` → s1: `print (x`$_1$`)` →
s2: `x = 18` → s2: `x`$_2$` = 18`

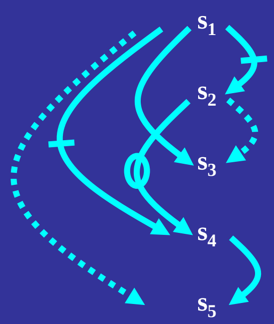
Other dependences

– **Input dependences:** s_1 reads memory that s_2 later reads (RAR)

s1: `y = x + 1`
s2: `print (x)`

Example

Identify the dependences




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

- flow (RAW)
- anti (WAR)
- output (WAW)
- input (RAR)

Representing Data Dependences

Implicitly

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

Def-use chains (du chains)

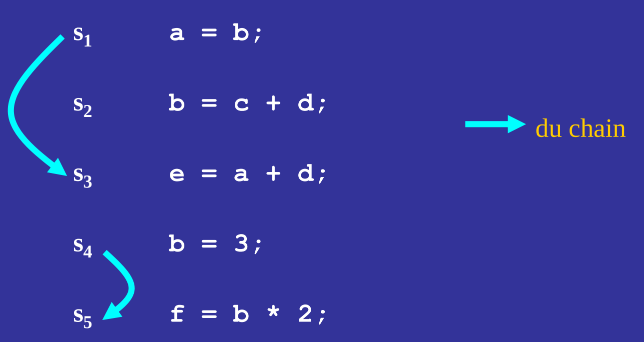
- Link each def to its uses
- Pros: explicit; therefore fast
- Cons: must be computed and updated, consumes space

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

February 11, 2015 Static Single Assignment Form 14

Representing Data Dependences (cont)

Implicitly

– Use 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, consumes space

Alternate representations

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

February 11, 2015 Static Single Assignment Form 15

Role of Alternate Program Representations

Process

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

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)

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 definition

Example

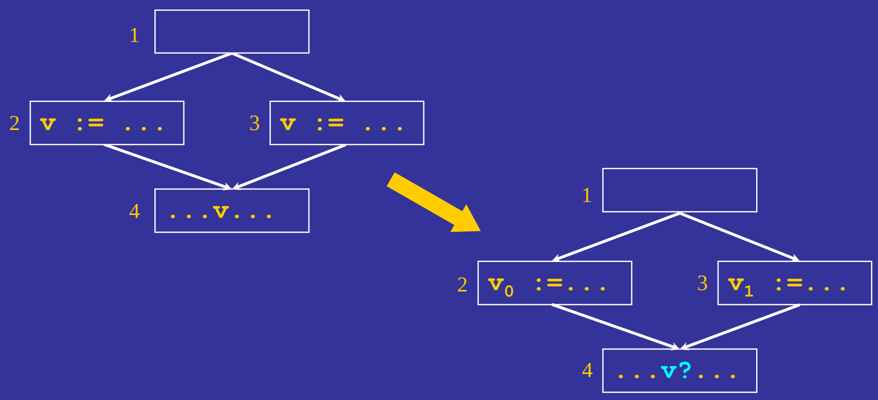
```
    v := ...                    v0 := ...
... := ... v ...    ==>    ... := ... v0 ...
    v := ...                    v1 := ...
... := ... v ...           ... := ... v1 ...
```

What do we do when there’s control flow?

SSA and Control Flow

Problem

– A use may be reached by several definitions



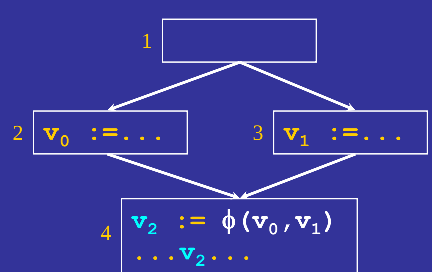
February 11, 2015 Static Single Assignment Form 18

SSA and Control Flow (cont)

Merging Definitions

– ϕ-functions merge multiple reaching definitions

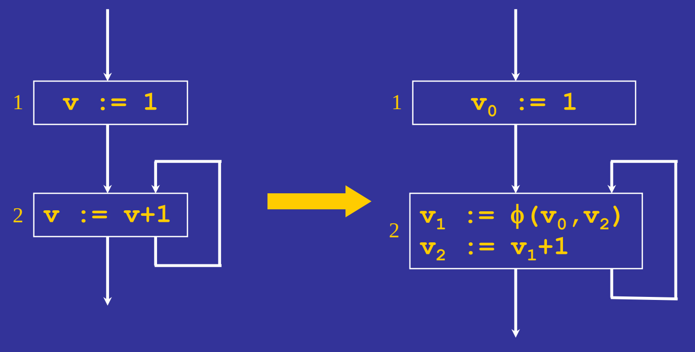
Example



February 11, 2015 Static Single Assignment Form 19

Exercise

Q: How do we transform the following code to SSA form?



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?

```
switch (c1) {
case 1:     x = 1; break;
case 2:     x = 2; break;
case 3:     x = 3; break;
}
x4 = φ(x1, x2, x3)
switch (c2) {
case 1:     y1 = x; break;
case 2:     y2 = x; break;
case 3:     y3 = x; break;
case 4:     y4 = x; break;
}
```

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

Transformation to SSA Form

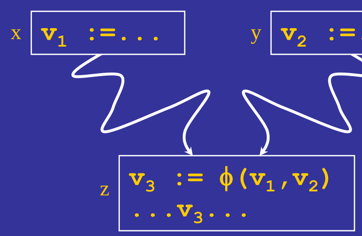
Two steps

- 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 we insert a ϕ-function for v at z



Approaches to Placing ϕ-Functions

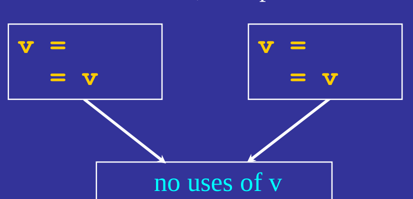
Minimal

– As few as possible subject to the basic rule

– How is this sub-optimal?

Briggs-Minimal

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



Briggs Minimal will not place a ϕ function in this case because v is not live across any CFG edge.

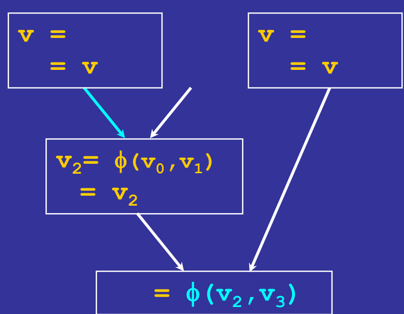
Exploits the short lifetimes of many temporary variables

Can we do better than Briggs-Minimal?

Approaches to Placing ϕ-Functions (cont)

Pruned

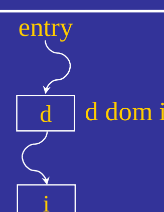
- Same as minimal, except does not insert dead ϕ-functions
- What's the difference between Pruned and Briggs-Minimal?



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 (assuming that this is the entire CFG)

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
- $DF(d) = \{n \mid \exists p \in pred(n),\ d \text{ dom } p \text{ and } d \text{ !sdom } n\}$

Notational Convenience

- $DF(S) = \cup_{s \in S} DF(s)$

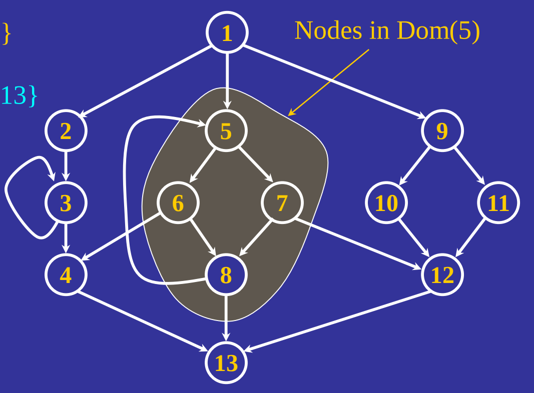
What is the significance of the dominance frontier?

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}



Where shall we place ϕ functions?

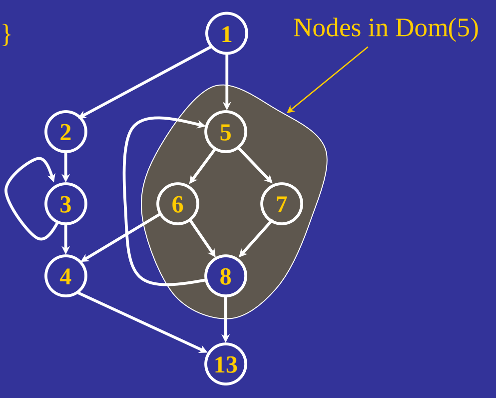
In SSA form, definitions must dominate uses

Dominance Frontier Example II

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

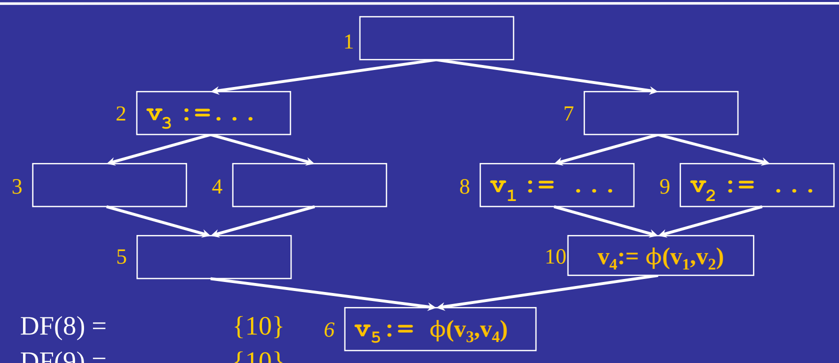
Dom(5) = {5, 6, 7, 8}

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



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



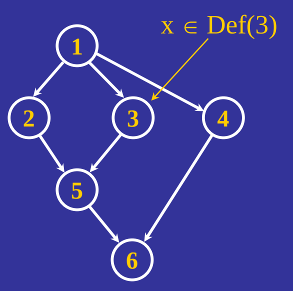
DF(8) = {10}
DF(9) = {10}
DF(2) = {6}
DF({8,9}) = {10}
DF(10) = {6}
DF({2,8,9,10}) = {6,10}

DF(d) = {n | $\exists p \in pred(n)$, d dom p and d !sdom n}

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

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 from 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}
```

Next Time

Lecture
- Will start at 2:15pm
- Data-flow analysis and SSA

Reading
- Csmith paper due Sunday February 15th at 5:00pm