

CSC D70: Compiler Optimization Dataflow-2 and Loops

Prof. Gennady Pekhimenko

University of Toronto

Winter 2018

*The content of this lecture is adapted from the lectures of
Todd Mowry and Phillip Gibbons*

Refreshing from Last Lecture

- Reaching definitions
- Live variables

Framework

	Reaching Definitions	Live Variables
Domain	Sets of definitions	Sets of variables
Direction	forward: $out[b] = f_b(in[b])$ $in[b] = \wedge out[pred(b)]$	backward: $in[b] = f_b(out[b])$ $out[b] = \wedge in[succ(b)]$
Transfer function	$f_b(x) = Gen_b \cup (x - Kill_b)$	$f_b(x) = Use_b \cup (x - Def_b)$
Meet Operation (\wedge)	\cup	\cup
Boundary Condition	$out[entry] = \emptyset$	$in[exit] = \emptyset$
Initial interior points	$out[b] = \emptyset$	$in[b] = \emptyset$

Other examples (e.g., Available expressions), defined in ALSU 9.2.6

Questions

- **Correctness**
 - equations are satisfied, if the program terminates.
- **Precision: how good is the answer?**
 - is the answer ONLY a union of all possible executions?
- **Convergence: will the analysis terminate?**
 - or, will there always be some nodes that change?
- **Speed: how fast is the convergence?**
 - how many times will we visit each node?

Foundations of Data Flow Analysis

- 1. Meet operator**
- 2. Transfer functions**
- 3. Correctness, Precision, Convergence**
- 4. Efficiency**

- Reference: ALSU pp. 613-631
- Background: Hecht and Ullman, Kildall, Allen and Cocke[76]
- Marlowe & Ryder, Properties of data flow frameworks: a unified model. Rutgers tech report, Apr. 1988

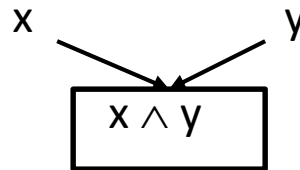
A Unified Framework

- **Data flow problems are defined by**
 - Domain of values: V
 - Meet operator ($V \wedge V \rightarrow V$), initial value
 - A set of transfer functions ($V \rightarrow V$)
- **Usefulness of unified framework**
 - To answer questions such as **correctness, precision, convergence, speed of convergence** for a family of problems
 - If meet operators and transfer functions have properties X , then we know Y about the above.
 - Reuse code

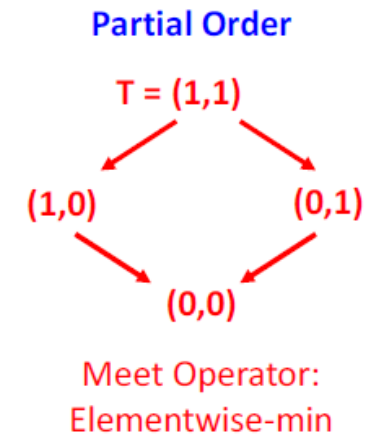
Meet Operator

- **Properties of the meet operator**

- **commutative:** $x \wedge y = y \wedge x$



- **idempotent:** $x \wedge x = x$
- **associative:** $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
- there is a **Top** element **T** such that $x \wedge T = x$

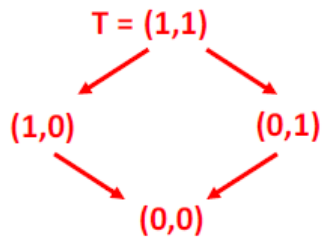


- **Meet operator defines a partial ordering on values**

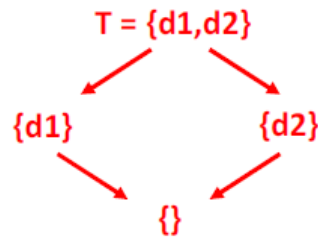
- $x \leq y$ if and only if $x \wedge y = x$ ($y \rightarrow x$ in diagram)
 - **Transitivity:** if $x \leq y$ and $y \leq z$ then $x \leq z$
 - **Antisymmetry:** if $x \leq y$ and $y \leq x$ then $x = y$
 - **Reflexivity:** $x \leq x$

Partial Order

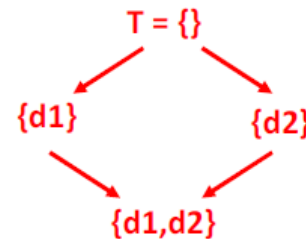
- Example: let $V = \{x \mid \text{such that } x \subseteq \{d_1, d_2\}\}$, $\wedge = \cap$



Meet Operator:
Elementwise-min



Meet Operator:
Intersection



Meet Operator:
Union

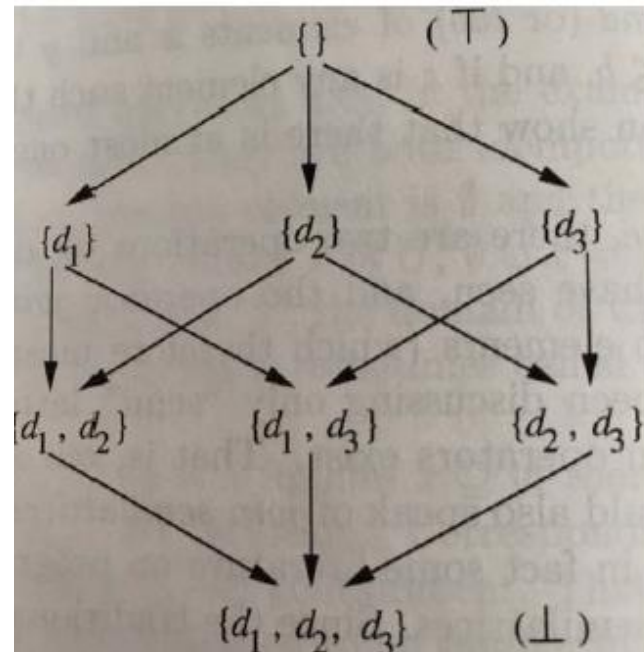
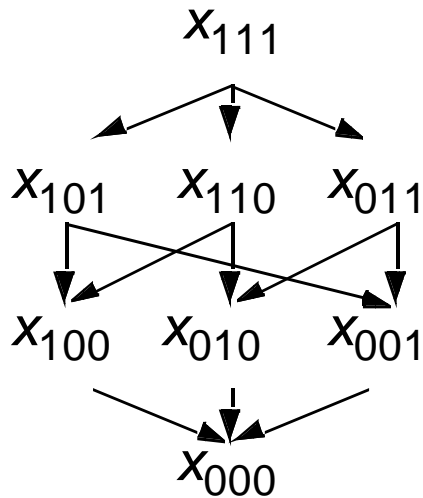
- Top and Bottom elements
 - Top T such that: $x \wedge T = x$
 - Bottom \perp such that: $x \wedge \perp = \perp$
- Values and meet operator in a data flow problem define a semi-lattice:
 - there exists a T , but not necessarily a \perp .
- x, y are ordered: $x \leq y$ then $x \wedge y = x$ ($y \rightarrow x$ in diagram)
- what if x and y are not ordered?
 - $x \wedge y \leq x$, $x \wedge y \leq y$, and if $w \leq x$, $w \leq y$, then $w \leq x \wedge y$

One vs. All Variables/Definitions

- Lattice for each variable: e.g. intersection



- Lattice for three variables:



Descending Chain

- **Definition**

- The **height** of a lattice is the largest number of **> relations** that will fit in a descending chain.

$$x_0 > x_1 > x_2 > \dots$$

- **Height of values in reaching definitions?**

Height n – number of definitions

- **Important property: finite descending chain**

- **Can an infinite lattice have a finite descending chain?**

yes

- **Example: Constant Propagation/Folding**

- To determine if a variable is a constant

- **Data values**

- undef, ... -1, 0, 1, 2, ..., not-a-constant

Transfer Functions

- **Basic Properties** $f: V \rightarrow V$
 - Has an identity function
 - There exists an f such that $f(x) = x$, for all x .
 - Closed under composition
 - if $f_1, f_2 \in F$, then $f_1 \cdot f_2 \in F$

Monotonicity

- A framework (F, V, \wedge) is **monotone** if and only if
 - $x \leq y$ implies $f(x) \leq f(y)$
 - i.e. a “smaller or equal” input to the same function will always give a “smaller or equal” output
- **Equivalently**, a framework (F, V, \wedge) is **monotone** if and only if
 - $f(x \wedge y) \leq f(x) \wedge f(y)$
 - i.e. merge input, then apply f is **small than or equal to** apply the transfer function individually and then merge the result

Example

- **Reaching definitions: $f(x) = \text{Gen} \cup (x - \text{Kill})$, $\wedge = \cup$**

- Definition 1:

- $x_1 \leq x_2, \text{Gen} \cup (x_1 - \text{Kill}) \leq \text{Gen} \cup (x_2 - \text{Kill})$

- Definition 2:

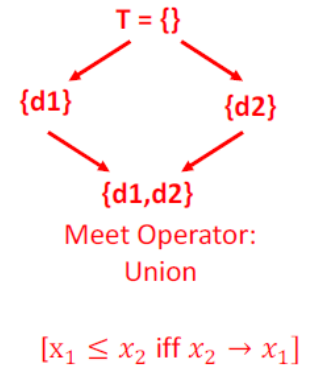
- $(\text{Gen} \cup (x_1 - \text{Kill})) \cup (\text{Gen} \cup (x_2 - \text{Kill}))$
 $= (\text{Gen} \cup ((x_1 \cup x_2) - \text{Kill}))$

- **Note: Monotone framework does not mean that $f(x) \leq x$**

- e.g., reaching definition for two definitions in program
- suppose: $f_x: \text{Gen}_x = \{d_1, d_2\}; \text{Kill}_x = \{\}$

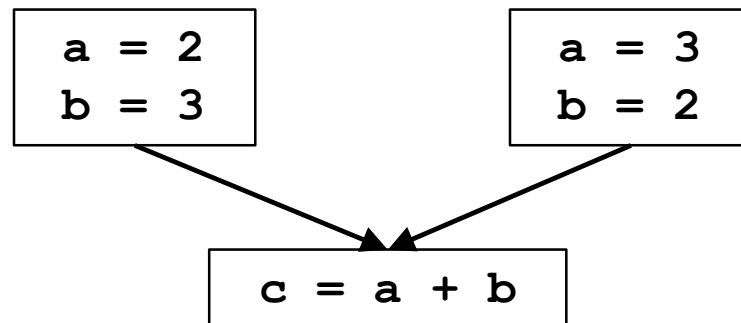
- **If $\text{input}(\text{second iteration}) \leq \text{input}(\text{first iteration})$**

- $\text{result}(\text{second iteration}) \leq \text{result}(\text{first iteration})$



Distributivity

- A framework (F, V, \wedge) is **distributive** if and only if
 - $f(x \wedge y) = f(x) \wedge f(y)$
 - i.e. merge input, then apply f is **equal to** apply the transfer function individually then merge result
- Example: Constant Propagation is NOT distributive



Data Flow Analysis

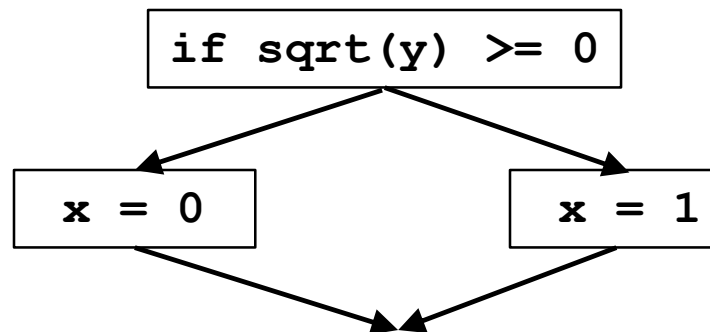
- **Definition**

- Let $f_1, \dots, f_m : \in F$, where f_i is the transfer function for node i
 - $f_p = f_{n_k} \cdot \dots \cdot f_{n_1}$, where p is a path through nodes n_1, \dots, n_k
 - $f_p = \text{identify function}$, if p is an empty path

- **Ideal data flow answer:**

- For each node n :

$\bigwedge f_{p_i}(T)$, for all possibly executed paths p_i reaching n .

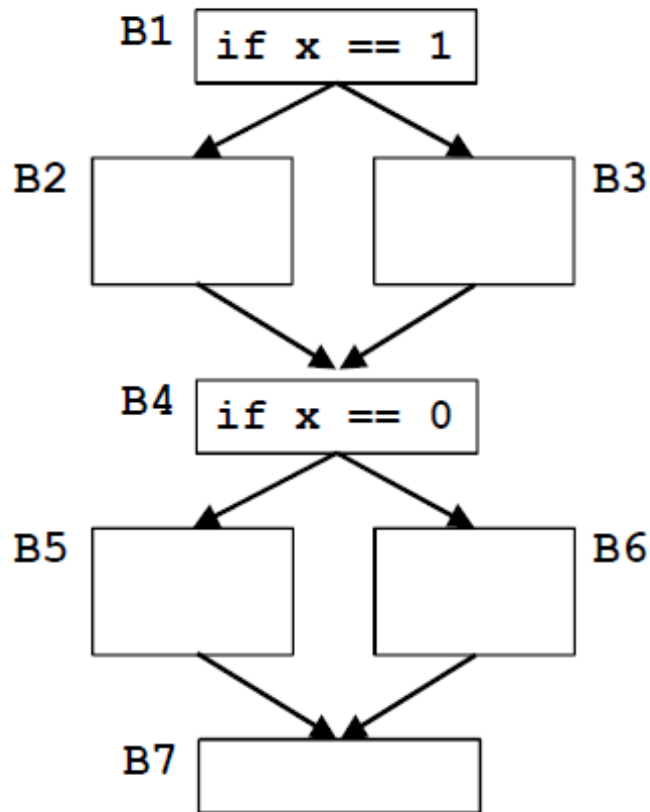


- But determining all possibly executed paths is **undecidable**

Meet-Over-Paths (MOP)

- Error in the conservative direction
- **Meet-Over-Paths (MOP):**
 - For each node n :
$$\text{MOP}(n) = \bigwedge f_{p_i}(T), \text{ for all paths } p_i \text{ reaching } n$$
 - a path exists as long there is an edge in the code
 - consider more paths than necessary
 - $\text{MOP} = \text{Perfect-Solution} \wedge \text{Solution-to-Unexecuted-Paths}$
 - $\text{MOP} \leq \text{Perfect-Solution}$
 - Potentially more constrained, solution is small
 - hence *conservative*
 - It is not **safe** to be $>$ Perfect-Solution!
- **Desirable solution: as close to MOP as possible**

MOP Example



Assume: B2 & B3 do not update x

Ideal: Considers only 2 paths
B1-B2-B4-B6-B7 (i.e., x=1)
B1-B3-B4-B5-B7 (i.e., x=0)

MOP: Also considers unexecuted paths
B1-B2-B4-B5-B7
B1-B3-B4-B6-B7

Solving Data Flow Equations

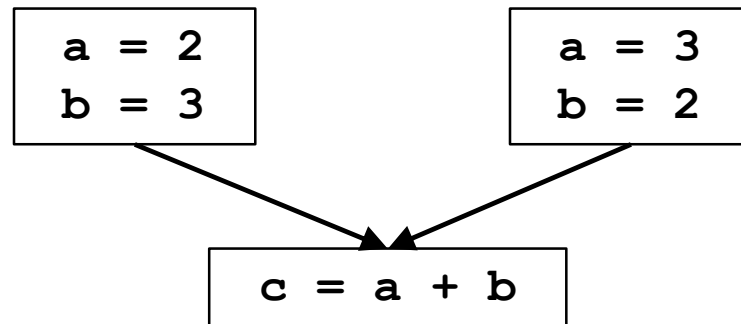
- **Example: Reaching definitions**
 - $\text{out}[\text{entry}] = \{\}$
 - $\text{Values} = \{\text{subsets of definitions}\}$
 - **Meet operator:** \cup
 - $\text{in}[b] = \cup \text{out}[p]$, for all predecessors p of b
 - **Transfer functions:** $\text{out}[b] = \text{gen}_b \cup (\text{in}[b] - \text{kill}_b)$
- **Any solution satisfying equations = Fixed Point Solution (FP)**
- **Iterative algorithm**
 - initializes $\text{out}[b]$ to $\{\}$
 - if converges, then it computes **Maximum Fixed Point (MFP)**:
 - **MFP** is the **largest of all solutions to equations**
- **Properties:**
 - $\text{FP} \leq \text{MFP} \leq \text{MOP} \leq \text{Perfect-solution}$
 - FP, MFP are safe
 - $\text{in}(b) \leq \text{MOP}(b)$

Partial Correctness of Algorithm

- If data flow framework is **monotone**, then if the algorithm converges, $IN[b] \leq MOP[b]$
- **Proof: Induction on path lengths**
 - Define $IN[entry] = OUT[entry]$
and transfer function of entry = Identity function
 - Base case: path of length 0
 - Proper initialization of $IN[entry]$
 - If true for path of length k , $p_k = (n_1, \dots, n_k)$, then true for path of length $k+1$: $p_{k+1} = (n_1, \dots, n_{k+1})$
 - Assume: $IN[n_k] \leq f_{n_{k-1}}(f_{n_{k-2}}(\dots f_{n_1}(IN[entry])))$
 - $IN[n_{k+1}] = OUT[n_k] \wedge \dots$
 - $\leq OUT[n_k]$
 - $\leq f_{n_k}(IN[n_k])$
 - $\leq f_{n_{k-1}}(f_{n_{k-2}}(\dots f_{n_1}(IN[entry])))$

Precision

- If data flow framework is **distributive**, then if the algorithm converges, **$IN[b] = MOP[b]$**



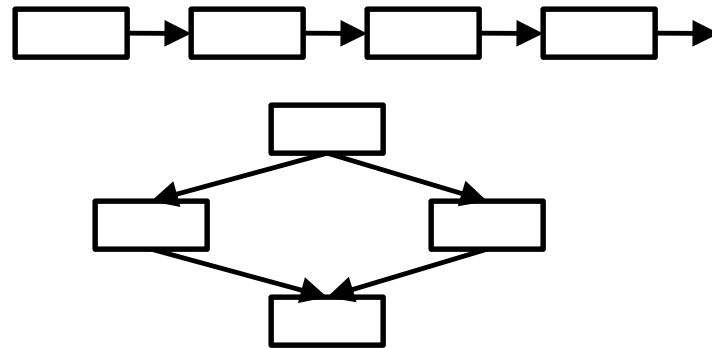
- Monotone but not distributive: behaves as if there are additional paths

Additional Property to Guarantee Convergence

- Data flow framework (**monotone**) converges if there is a **finite descending chain**
- For each variable $IN[b]$, $OUT[b]$, consider the sequence of values set to each variable **across iterations**:
 - if sequence for $in[b]$ is monotonically decreasing
 - sequence for $out[b]$ is monotonically decreasing
 - ($out[b]$ initialized to T)
 - if sequence for $out[b]$ is monotonically decreasing
 - sequence of $in[b]$ is monotonically decreasing

Speed of Convergence

- Speed of convergence depends on order of node visits



- Reverse “direction” for backward flow problems

Reverse Postorder

- Step 1: depth-first post order

```
main() {  
    count = 1;  
    Visit(root);  
}  
Visit(n) {  
    for each successor s that has not been  
visited  
        Visit(s);  
    PostOrder(n) = count;  
    count = count+1;  
}
```

- Step 2: reverse order

```
For each node i  
    rPostOrder = NumNodes - PostOrder(i)
```

Depth-First Iterative Algorithm (forward)

```
input: control flow graph CFG = (N, E, Entry, Exit)

/* Initialize */
  out[entry] = init_value
  For all nodes i
    out[i] =  $\perp$ 
  Change = True

/* iterate */
  While Change {
    Change = False
    For each node i in rPostOrder {
      in[i] =  $\wedge$ (out[p]), for all predecessors p of i
      oldout = out[i]
      out[i] =  $f_i$ (in[i])
      if oldout  $\neq$  out[i]
        Change = True
    }
  }
```


Speed of Convergence

- **If cycles do not add information**
 - information can flow in one pass down a series of nodes of increasing order number:
 - e.g., 1 -> 4 -> 5 -> 7 -> 2 -> 4 ...
 - passes determined by **number of back edges in the path**
 - essentially the nesting depth of the graph
 - **Number of iterations = number of back edges in any acyclic path + 2**
 - (2 are necessary even if there are no cycles)
- **What is the depth?**
 - corresponds to depth of intervals for “reducible” graphs
 - in real programs: average of 2.75

A Check List for Data Flow Problems

- **Semi-lattice**
 - set of values
 - meet operator
 - top, bottom
 - finite descending chain?
- **Transfer functions**
 - function of each basic block
 - monotone
 - distributive?
- **Algorithm**
 - initialization step (entry/exit, other nodes)
 - visit order: rPostOrder
 - depth of the graph

Conclusions

- Dataflow analysis examples
 - Reaching definitions
 - Live variables
- Dataflow formation definition
 - Meet operator
 - Transfer functions
 - Correctness, Precision, Convergence
 - Efficiency

CSC D70: Compiler Optimization Loops

Prof. Gennady Pekhimenko

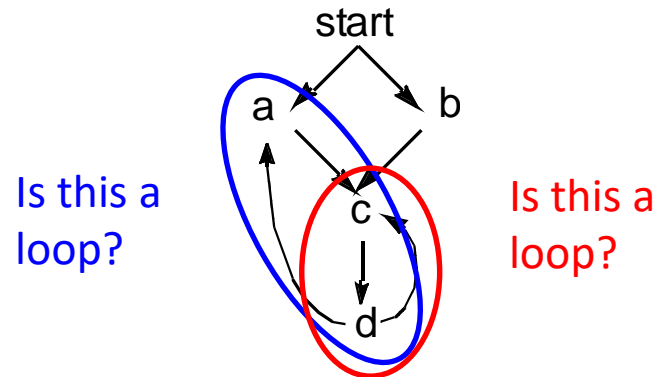
University of Toronto

Winter 2018

*The content of this lecture is adapted from the lectures of
Todd Mowry and Phillip Gibbons*

What is a Loop?

- **Goals:**
 - Define a loop in graph-theoretic terms (control flow graph)
 - Not sensitive to input syntax
 - A uniform treatment for all loops: DO, while, goto's
- **Not every cycle is a “loop” from an optimization perspective**

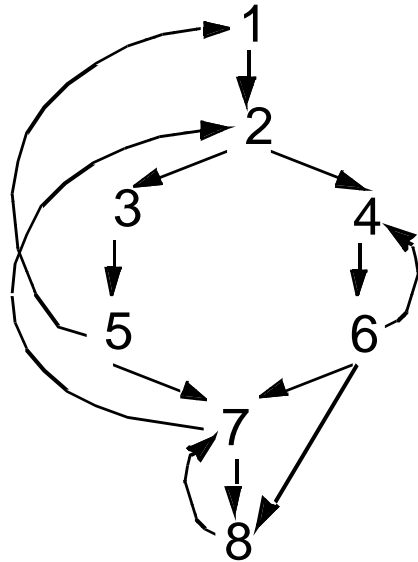


- **Intuitive properties of a loop**
 - single entry point
 - edges must form at least a cycle

Formal Definitions

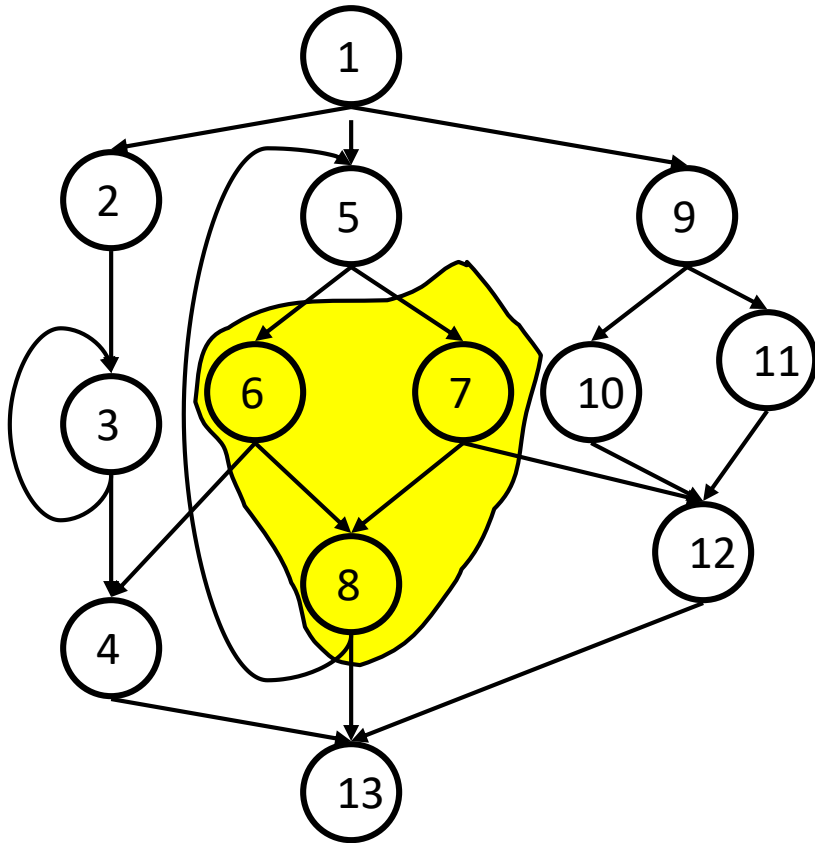
- **Dominators**

- Node d **dominates** node n in a graph ($d \text{ dom } n$) if every path from the start node to n goes through d

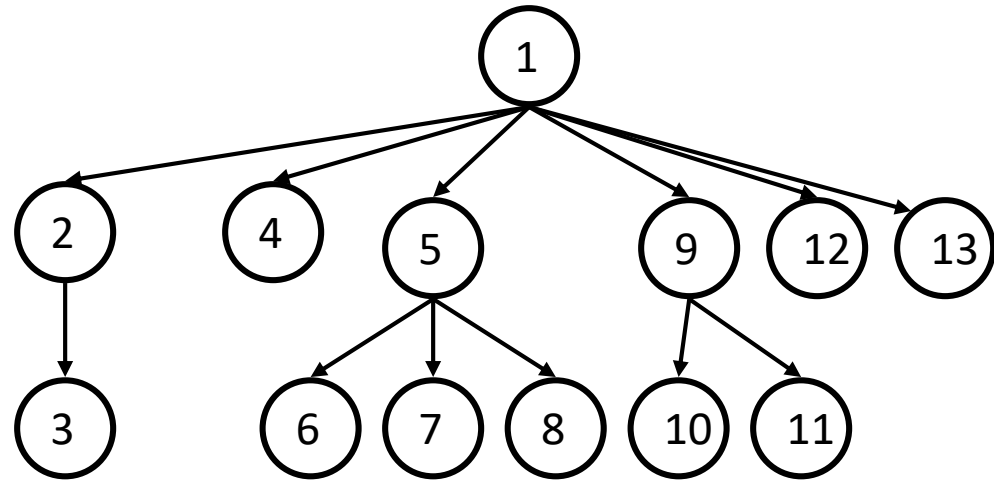


- Dominators can be organized as a **tree**
 - $a \rightarrow b$ in the **dominator tree** iff a immediately dominates b

Dominance



CFG



D-Tree

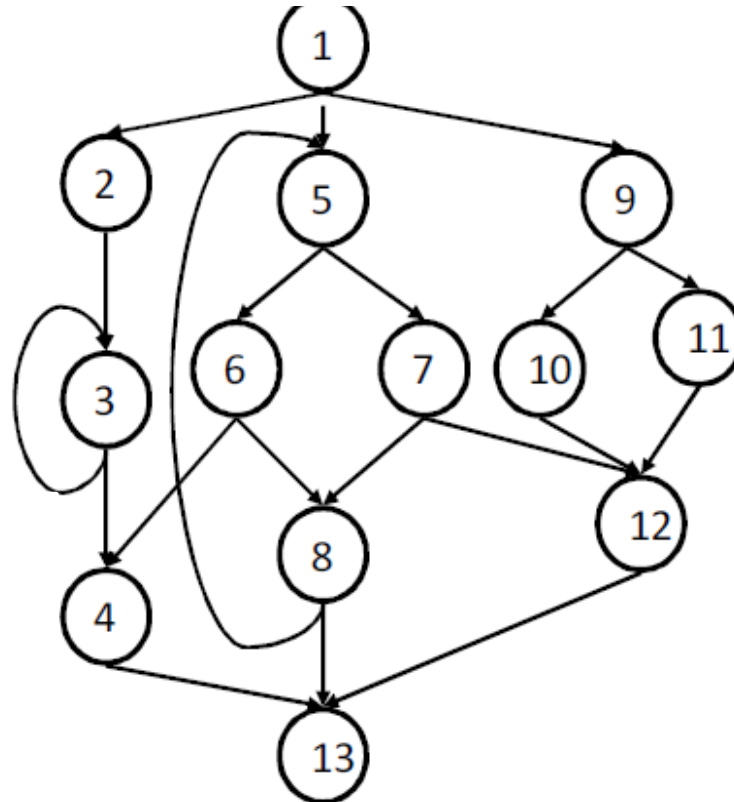
x strictly dominates w (x sdom w) iff x dom w AND $x \neq w$

Natural Loops

- **Definitions**

- Single entry-point: **header**
 - a header **dominates all nodes in the loop**
- A **back edge** is an arc whose **head dominates its tail** (tail -> head)
 - a back edge **must be a part of at least one loop**
- The **natural loop of a back edge** is the **smallest set** of nodes that **includes the head and tail of the back edge**, and has **no predecessors outside the set**, except for the predecessors of the header.

Natural Loops - Example



Algorithm to Find Natural Loops

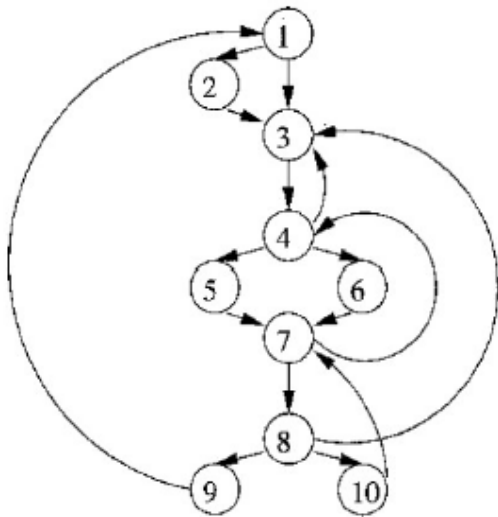
- Find the dominator relations in a flow graph
- Identify the back edges
- Find the natural loop associated with the back edge

1. Finding Dominators

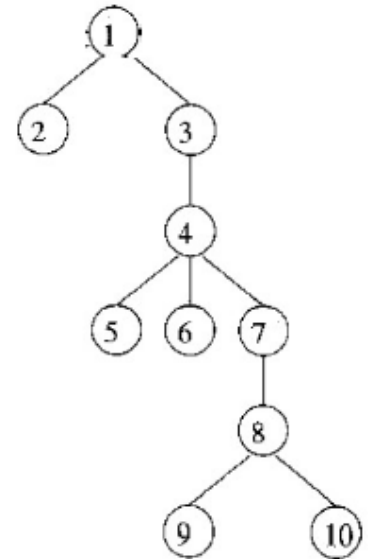
- **Definition**
 - Node d dominates node n in a graph ($d \text{ dom } n$) if every path from the start node to n goes through d
- **Formulated as MOP problem:**
 - node d lies on all possible paths reaching node $n \Rightarrow d \text{ dom } n$
 - Direction:
 - Values:
 - Meet operator:
 - Top:
 - Bottom:
 - Boundary condition: start/entry node =
 - Initialization for internal nodes
 - Finite descending chain?
 - Transfer function:
- **Speed:**
 - With reverse postorder, most flow graphs (reducible flow graphs) converge in 1 pass

Example

$$\text{OUT}[b] = \{b\} \cup \left(\bigcap_{p=\text{pre}(b)} \text{OUT}[p] \right)$$



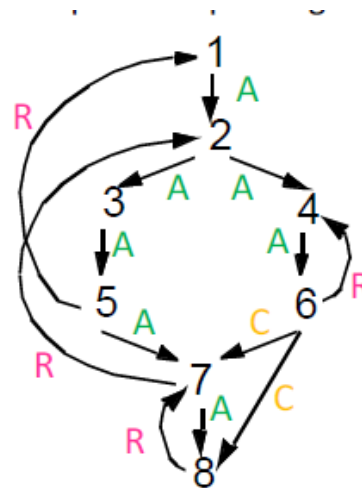
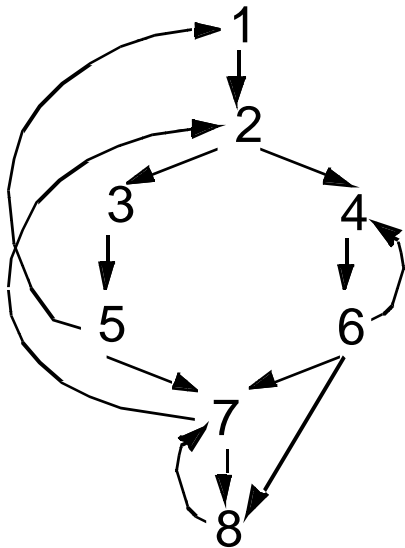
- OUT[1] = {1}
 - OUT[2] = {1,2}
 - OUT[3] = {1,3}
 - OUT[4] = {1,3,4}
 - OUT[5] = {1,3,4,5}
 - OUT[6] = {1,3,4,6}
 - OUT[7] = {1,3,4,7}
 - OUT[8] = {1,3,4,7,8}
 - OUT[9] = {1,3,4,7,8,9}
 - OUT[10] = {1,3,4,7,8,10}
- (No change in second iteration)



2. Finding Back Edges

- **Depth-first spanning tree**

- Edges traversed in a depth-first search of the flow graph form a depth-first spanning tree

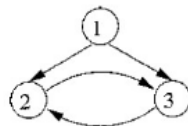


- **Categorizing edges in graph**

- Advancing (A) edges: from ancestor to proper descendant
- Cross (C) edges: from right to left
- Retreating (R) edges: from descendant to ancestor (not necessarily proper)

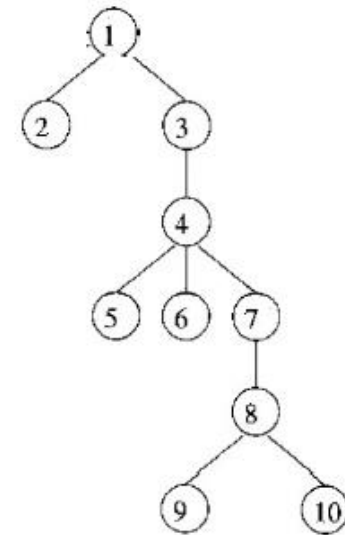
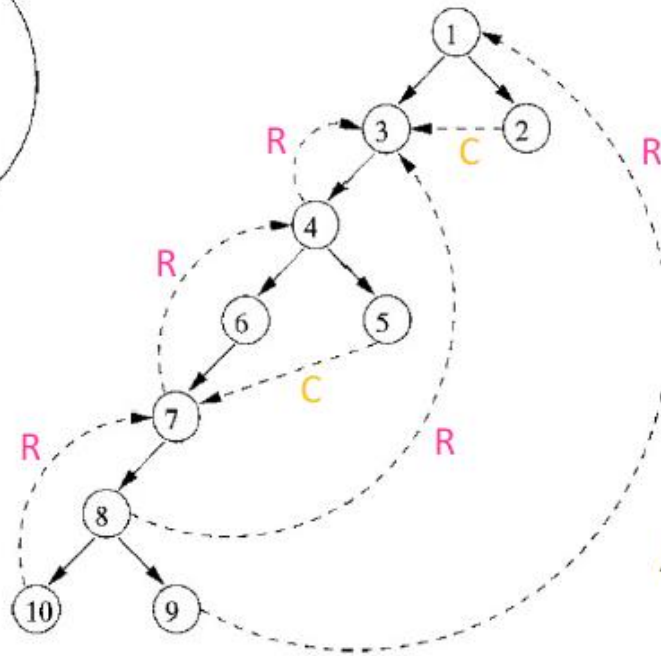
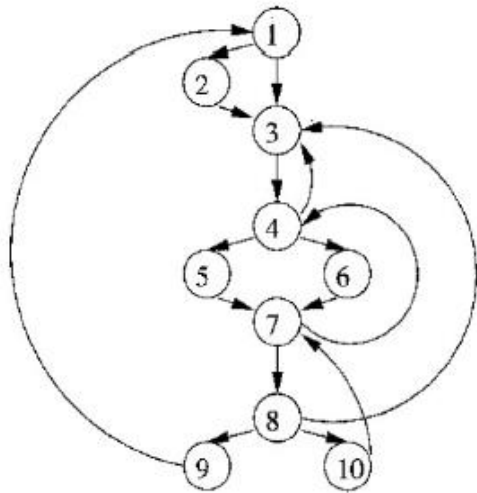
Back Edges

- **Definition**
 - **Back edge**: $t \rightarrow h$, h dominates t
- **Relationships between graph edges and back edges**
- **Algorithm**
 - Perform a depth first search
 - For each retreating edge $t \rightarrow h$, check if h is in t 's dominator list
- **Most programs (all structured code, and most GOTO programs) have **reducible** flow graphs**
 - retreating edges = back edges



A nonreducible flow graph

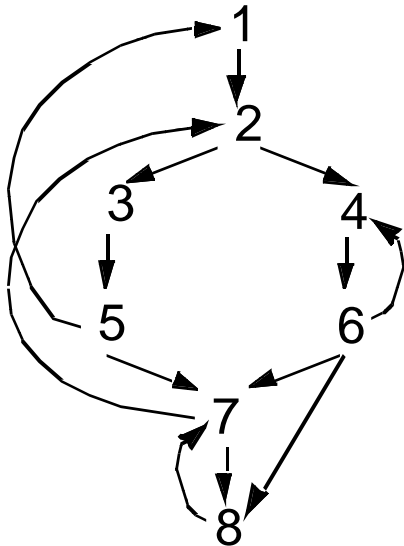
Examples



All the retreating edges
are back edges

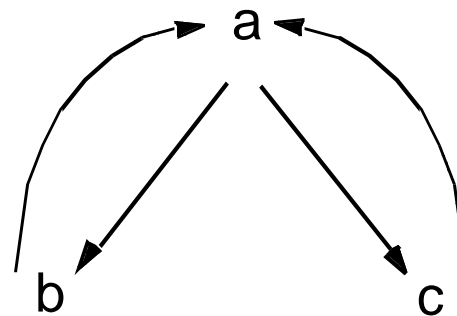
3. Constructing Natural Loops

- The **natural loop of a back edge** is the smallest set of nodes that includes the head and tail of the back edge, and has no predecessors outside the set, except for the predecessors of the header.
- **Algorithm**
 - delete h from the flow graph
 - find those nodes that can reach t
(those nodes plus h form the natural loop of $t \rightarrow h$)



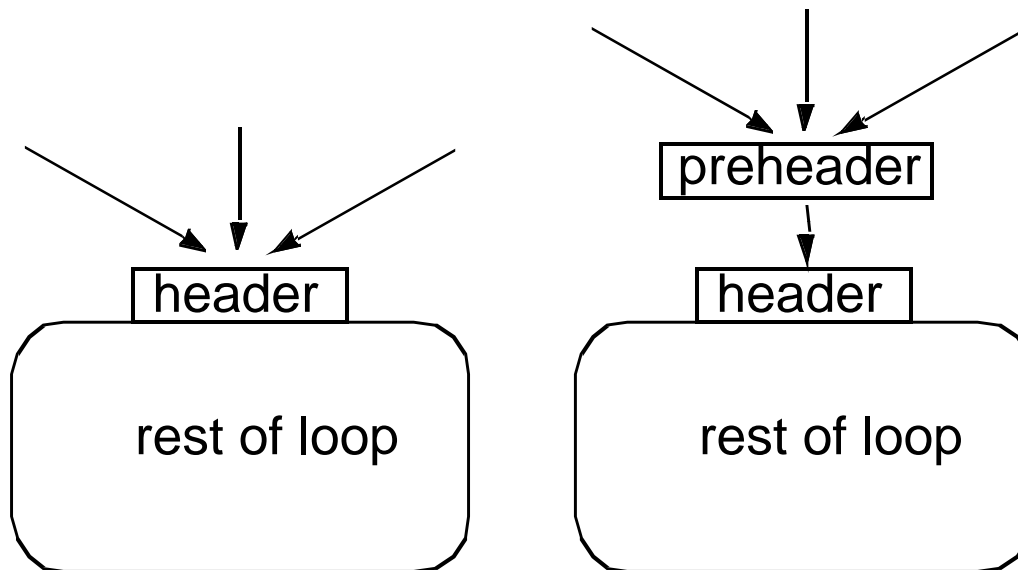
Inner Loops

- **If two loops do not have the same header:**
 - they are either disjoint, or
 - one is entirely contained (nested within) the other
 - inner loop: one that contains no other loop.
- **If two loops share the same header:**
 - Hard to tell which is the inner loop
 - Combine as one



Preheader

- Optimizations often require code to be executed once before the loop
- Create a preheader basic block for every loop



Finding Loops: Summary

- **Define loops in graph theoretic terms**
- **Definitions and algorithms for:**
 - Dominators
 - Back edges
 - Natural loops

CSC D70: Compiler Optimization Dataflow-2 and Loops

Prof. Gennady Pekhimenko

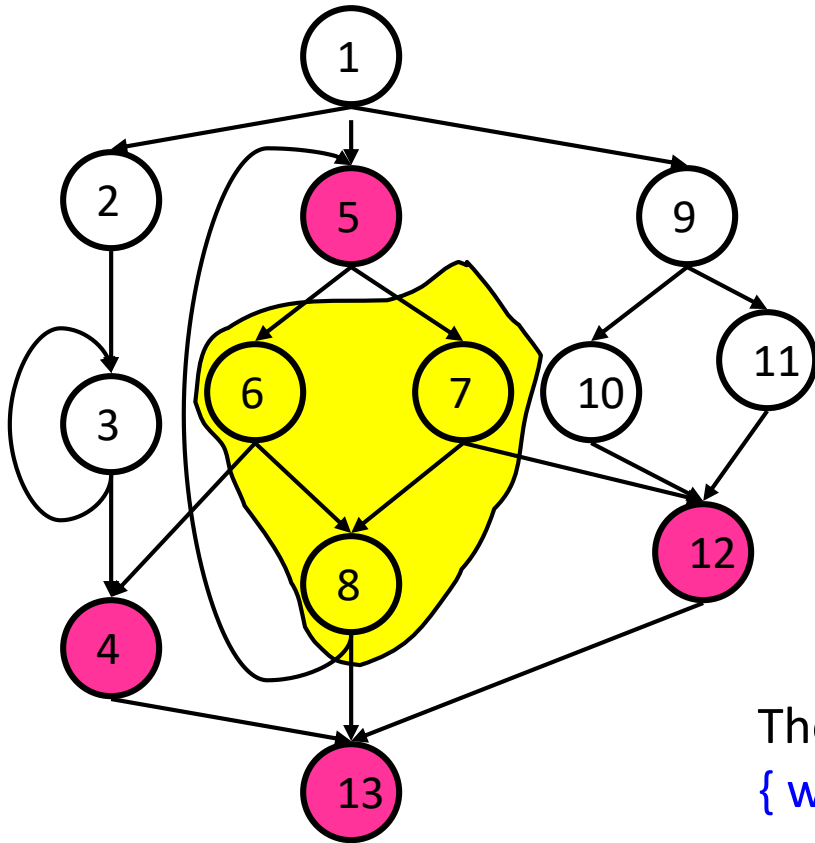
University of Toronto

Winter 2018

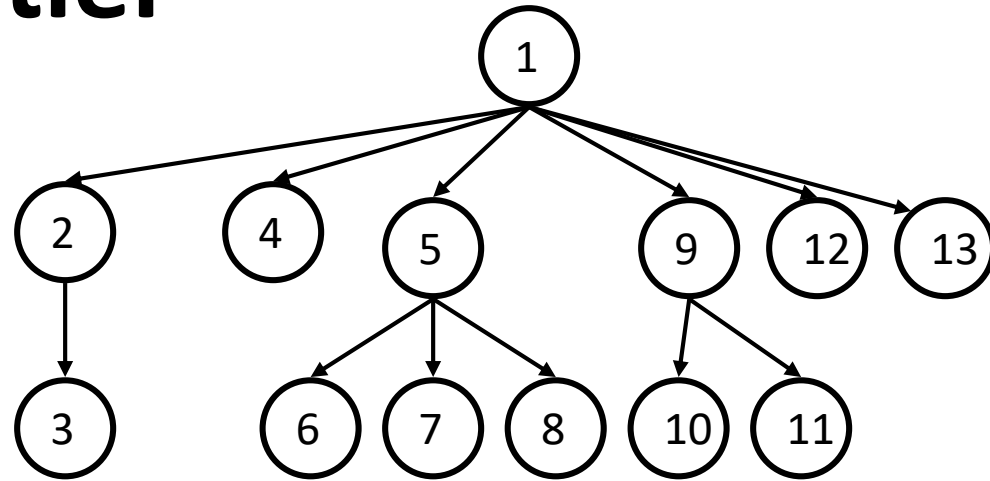
*The content of this lecture is adapted from the lectures of
Todd Mowry and Phillip Gibbons*

Backup Slides

Dominance Frontier



CFG



D-Tree

The **Dominance Frontier** of a node $x = \{ w \mid x \text{ dom pred}(w) \text{ AND } \neg(x \text{ sdom } w) \}$

x strictly dominates w ($x \text{ sdom } w$) iff $x \text{ dom } w$ AND $x \neq w$

Dominance Frontier and Path Convergence

