CSC D70:
Compiler Optimization
Dataflow Analysis-2

Prof. Gennady Pekhimenko
University of Toronto
Winter 2020

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

Framework

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

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

Foundations of Data Flow Analysis

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

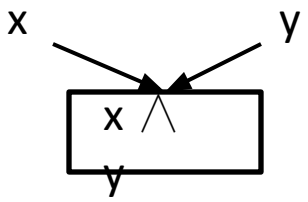
- Reference: ALSU pp. 613-631
- Background: Hecht and Ullman, Kildall, Allen and Cocke[76]
- Marlowe & Ryder, Properties of data flow frameworks: a unified model. Rutgers tech report, Apr. 1988

A Unified Framework

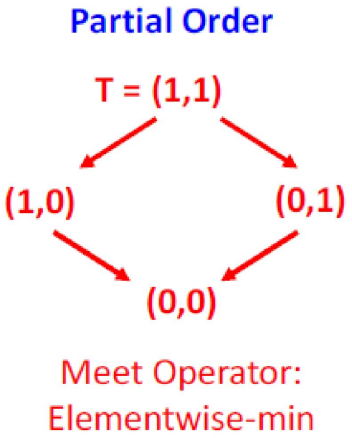
- **Data flow problems are defined by**
  - Domain of values: V
  - Meet operator (V ∧ V → V), initial value
  - A set of transfer functions (V → V)
- **Usefulness of unified framework**
  - To answer questions such as correctness, precision, convergence, speed of convergence for a family of problems
    - 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: $x \wedge y = y \wedge x$



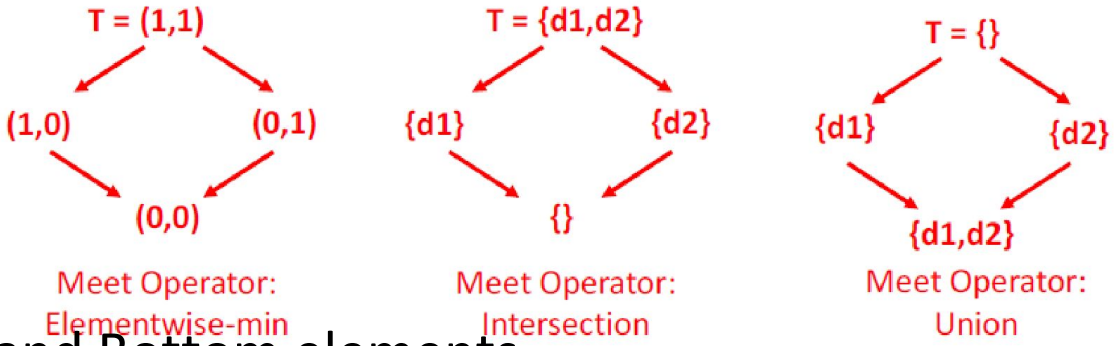
  - idempotent: $x \wedge x = x$
  - associative: $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
  - there is a Top element T such that $x \wedge T = x$



- **Meet operator defines a partial ordering on values**
  - $x \leq y$ if and only if $x \wedge y = x$ (y -> 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 $x = y$
    - Reflexitivity: $x \leq x$

Partial Order

- Example: let **V** = {x | such that x ⊆ { $\mathbf{d_1}$, $\mathbf{d_2}$}}, ⋀ = ∩

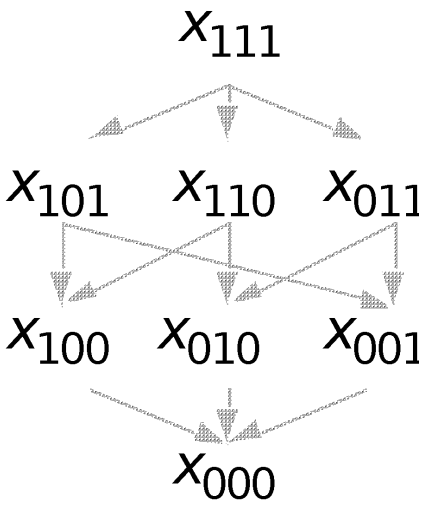


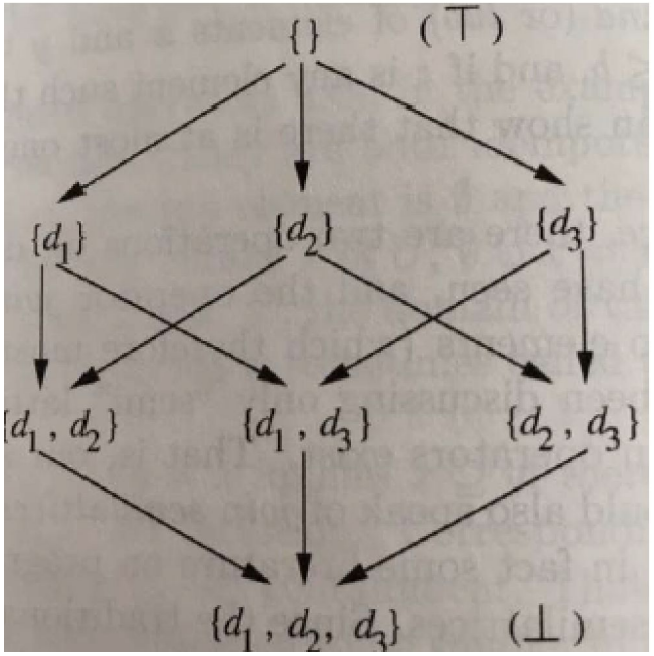
- Top and Bottom elements
  - Top T such that: **x** ⋀ T = **x**
  - Bottom ⊥ such that: **x** ⋀ ⊥ = ⊥
- Values and meet operator in a data flow problem define a semi-lattice:
  - there exists a T, but not necessarily a ⊥.
- x, y are ordered: x ≤ y then x ⋀ y = x (y -> x in diagram)
- what if x and y are not ordered?
  - x ⋀ y ≤ x, x ⋀ y ≤ y, and if w ≤ x, w ≤ y, then w ≤ x ⋀ 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.

$$x_0 > x_1 > x_2 > \ldots$$

- **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 → 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 $f_1 \cdot f_2 \in F$

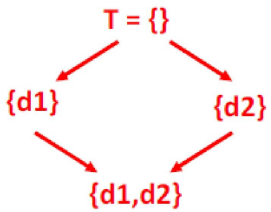
Monotonicity

- 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) = Gen ∪ (x - Kill), ∧ = ∪**
  - Definition 1:
    - $x_1 \leq x_2$, Gen ∪ (x_1 - Kill) ≤ Gen ∪ (x_2 - Kill)
  - Definition 2:
    - (Gen ∪ (x_1 - Kill)) ∪ (Gen ∪ (x_2 - Kill))

      = (Gen ∪ ((x_1 ∪ x_2) - Kill))



Meet Operator: Union

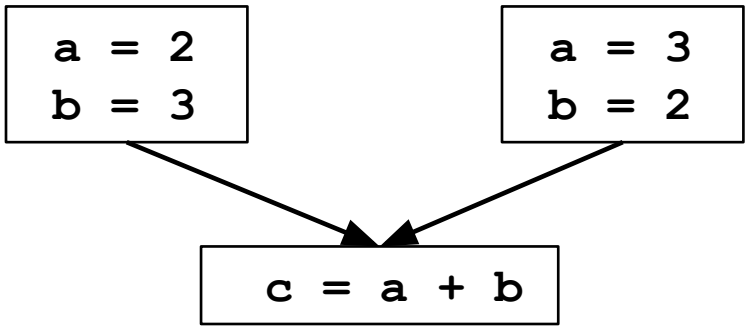
$[x_1 \leq x_2 \text{ iff } x_2 \rightarrow x_1]$

- **Note: Monotone framework does not mean that f(x) ≤ x**
  - e.g., reaching definition for two definitions in program
  - suppose: f_x: Gen_x = {d_1, d_2} ; $Kill_x$ = {}

- **If input(second iteration) ≤ input(first iteration)**
  - result(second iteration) ≤ result(first iteration)

Distributivity

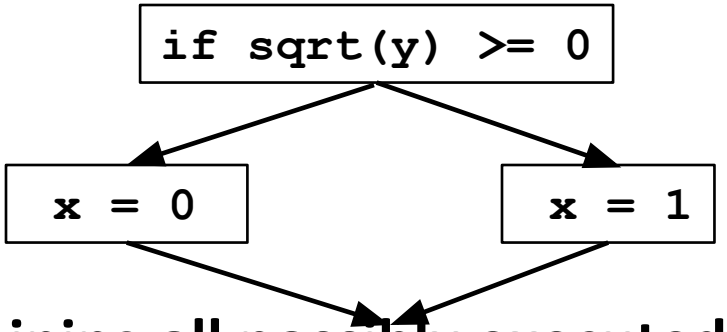
- A framework ($F, V, \wedge$) is **distributive** if and only if
  - $f(x \wedge y) = f(x) \wedge f(y)$
  - i.e. merge input, then apply f is **equal to** apply the transfer function individually then merge result
- Example: Constant Propagation is NOT distributive



Data Flow Analysis

- **Definition**
  - Let $f_1, \ldots, f_m : \in F$, where f_i is the transfer function for node i
    - $f_p = f_{nk} \cdot \ldots \cdot f_{n1}$, where p is a path through nodes $n_1, \ldots, n_k$
    - f_p = identify function, if p is an empty path
- **Ideal data flow answer:**
  - For each node n:

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



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

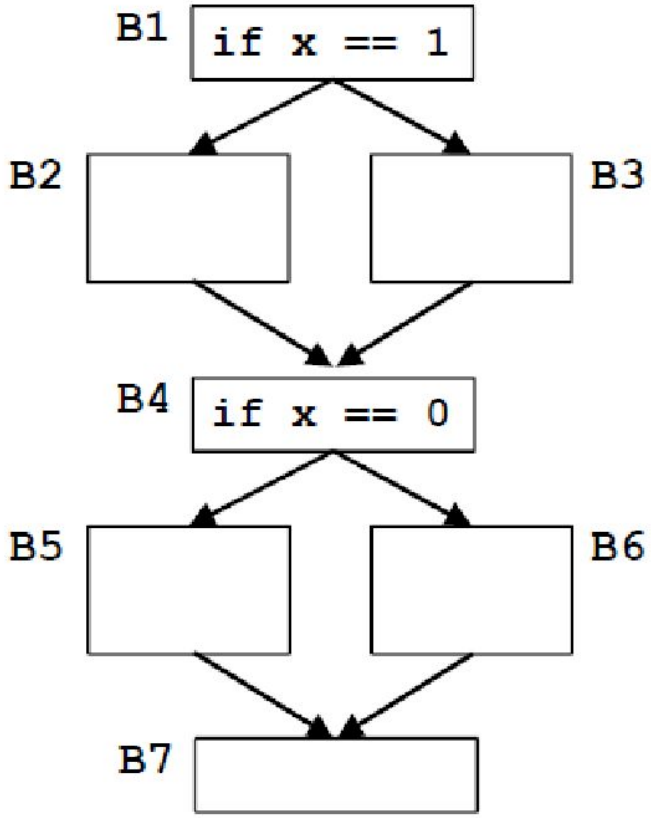
Meet-Over-Paths (MOP)

- **Error in the conservative direction**
- **Meet-Over-Paths (MOP):**
  - For each node *n:*

    MOP(*n*) = $\bigwedge f_{pi}(\top)$, for all paths p_i reaching *n*

  - a path exists as long there is an edge in the code
  - consider more paths than necessary
  - MOP = Perfect-Solution $\wedge$ Solution-to-Unexecuted-Paths
  - MOP ≤ 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**
  - out[entry] = {}
  - Values = {subsets of definitions}
  - Meet operator: $\cup$
    - in[b] = $\cup$ out[*p*], for all predecessors *p* of b
  - Transfer functions: out[b] = $gen_b \cup (in[b] - kill_b)$
- **Any solution satisfying equations = Fixed Point Solution (FP)**
- **Iterative algorithm**
  - initializes out[b] to {}
  - if converges, then it computes Maximum Fixed Point (MFP):
    - MFP is the largest of all solutions to equations
- **Properties:**
  - FP ≤ MFP ≤ MOP ≤ Perfect-solution
  - FP, MFP are safe
  - in(b) ≤ MOP(b)

Partial Correctness of Algorithm

- **If data flow framework is monotone, then if the algorithm converges, IN[b] ≤ 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, $p_k = (n_1, ..., n_k)$, then true for path of length k+1: $p_{k+1} = (n_1, ..., n_{k+1})$
    - Assume: $IN[n_k] \leq f_{nk-1}(f_{nk-2}(... f_{n1}(IN[entry])))$
    - $IN[n_{k+1}] = OUT[n_k] \wedge ...$

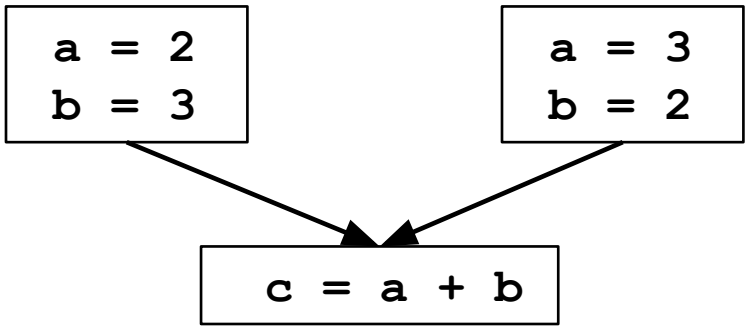
      $\leq OUT[n_k]$

      $\leq f_{nk}(IN[n_k])$

      $\leq f_{nk-1}(f_{nk-2}(... f_{n1}(IN[entry])))$

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**

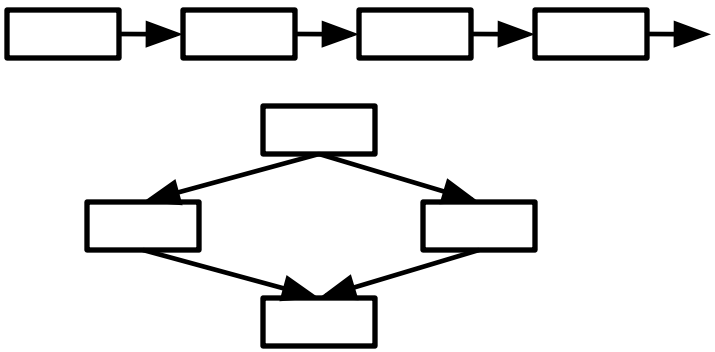
- For each variable IN[b], OUT[b], consider the sequence of values set to each variable across iterations:

  - 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] = ⊤
    Change = True
/* iterate */

    While Change {
       Change = False
       For each node i in rPostOrder {
          in[i] = ∧(out[p]), for all predecessors p of i
          oldout = out[i]
           out[i] = f_i(in[i])
          if oldout ≠ 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

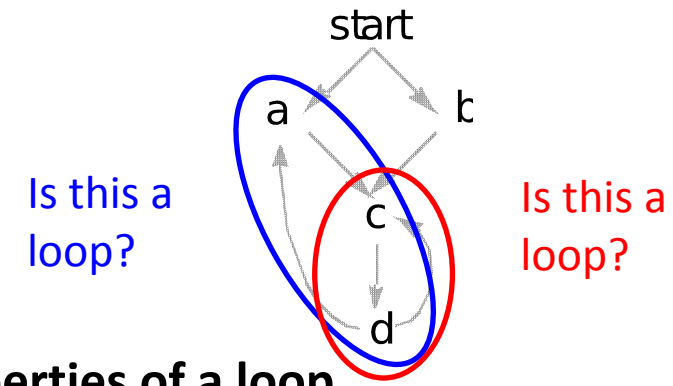
CSC D70: Compiler Optimization Loops

Prof. Gennady Pekhimenko
University of Toronto
Winter 2020

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

What is a Loop?

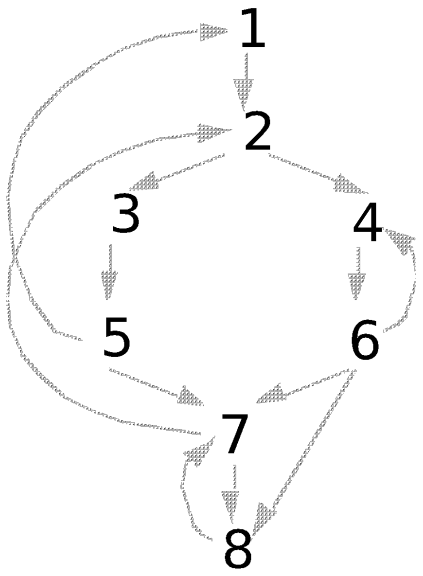
- **Goals:**
  - Define a loop in graph-theoretic terms (control flow graph)
  - Not sensitive to input syntax
  - A uniform treatment for all loops: DO, while, goto's
- **Not every cycle is a "loop" from an optimization perspective**



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

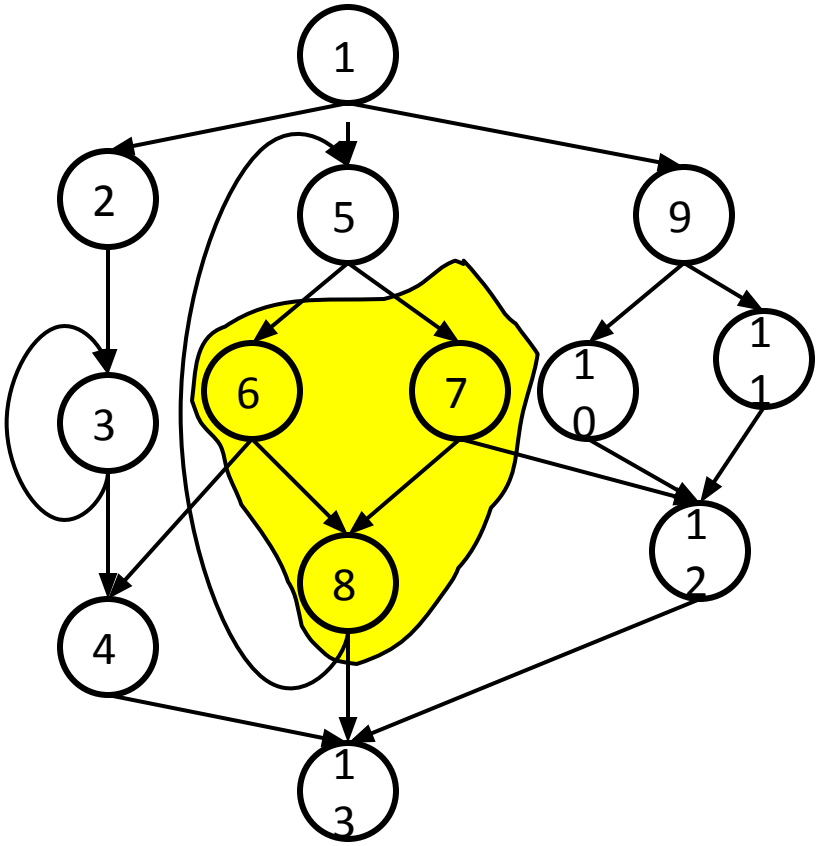
Formal Definitions

- **Dominators**
  - Node *d* dominates node *n* in a graph (*d* dom *n*) if every path from the start node to *n* goes through *d*

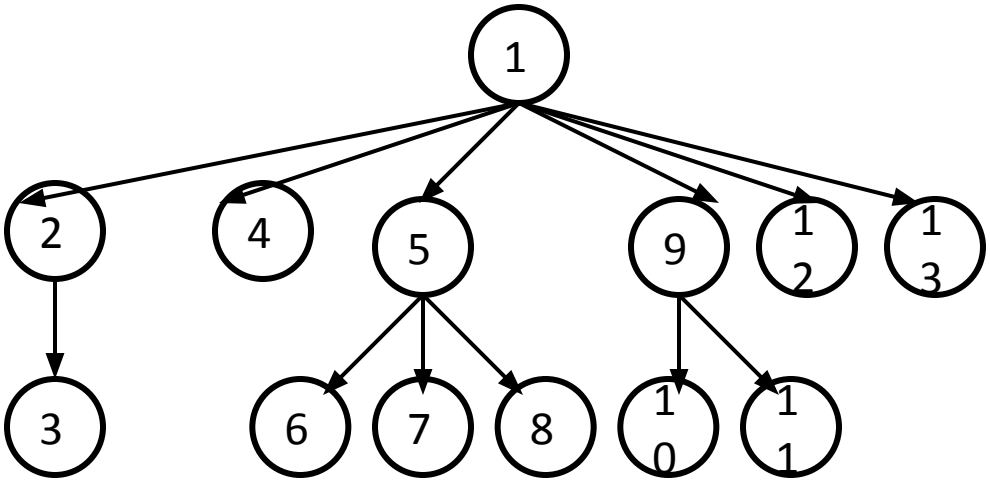


  - Dominators can be organized as a tree
    - *a* ->*b* in the dominator tree iff *a* immediately dominates *b*

Dominance



CFG



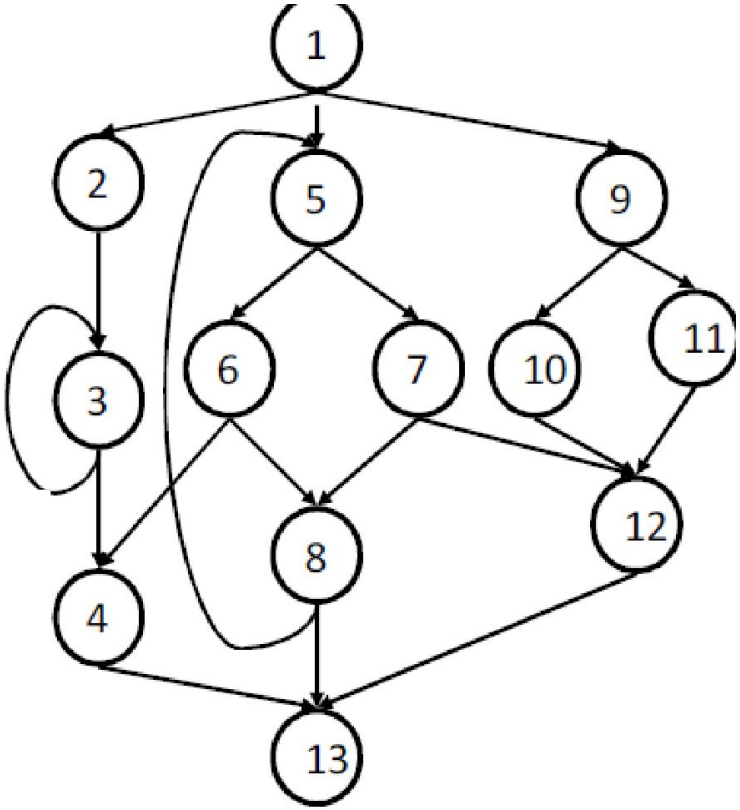
D-Tree

x strictly dominates w (x sdom w) iff x dom w AND x ≠ 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

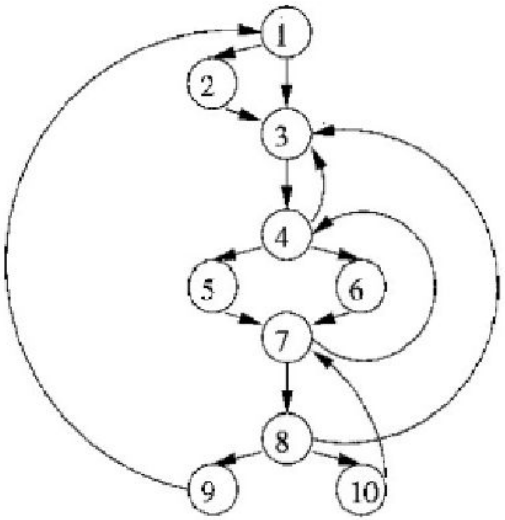
- Find the dominator relations in a flow graph
- Identify the back edges
- Find the natural loop associated with the back edge

1. Finding Dominators

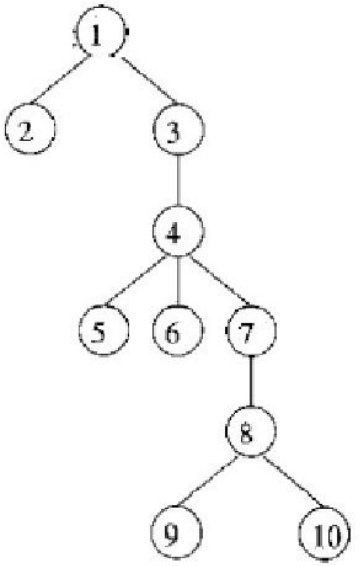
- **Definition**
  - Node *d* dominates node *n* in a graph (*d* dom *n*)
    if every path from the start node to *n* goes through *d*
- **Formulated as MOP problem:**
  - node *d* lies on all possible paths reaching node *n* ⇒ *d* dom *n*
    - Direction:
    - Values:
    - Meet operator:
    - Top:
    - 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

$$OUT[b]=\{b\} \cup (\cap_{\{p=pre\ (b)\}} OUT[p])$$

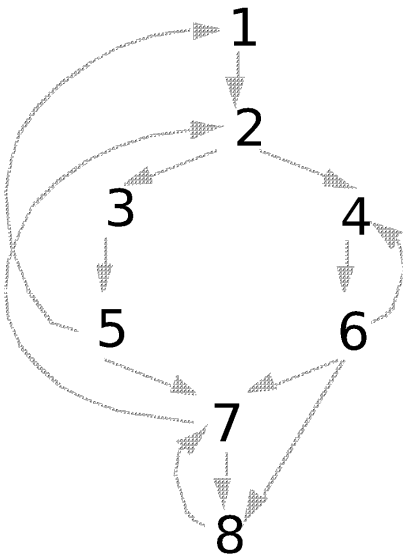


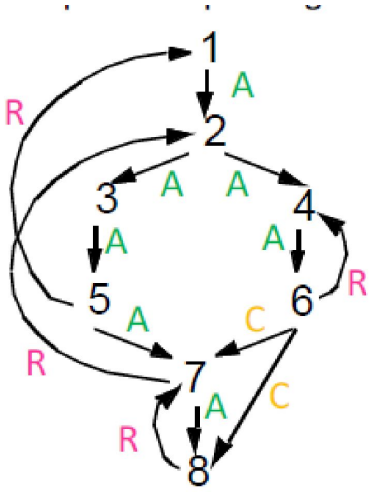
OUT[1] = {1}
OUT[2] = {1,2}
OUT[3] = {1,3}
OUT[4] = {1,3,4}
OUT[5] = {1,3,4,5}
OUT[6] = {1,3,4,6}
OUT[7] = {1,3,4,7}
OUT[8] = {1,3,4,7,8}
OUT[9] = {1,3,4,7,8,9}
OUT[10] = {1,3,4,7,8,10}
(No change in second iteration)



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

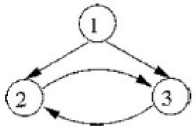




- **Categorizing edges in graph**
  - Advancing (A) edges: from ancestor to proper descendant
  - Cross (C) edges: from right to left
  - Retreating (R) edges: from descendant to ancestor (not necessarily proper)

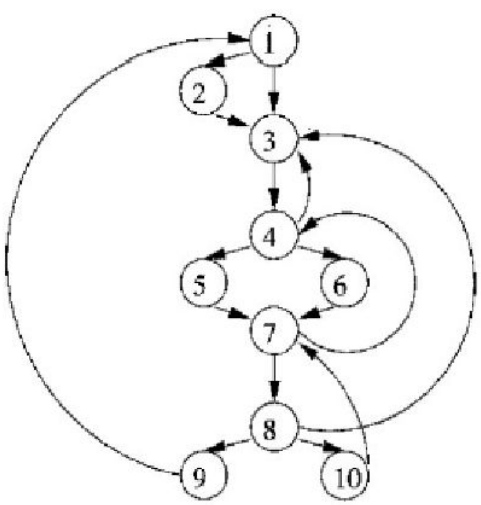
Back Edges

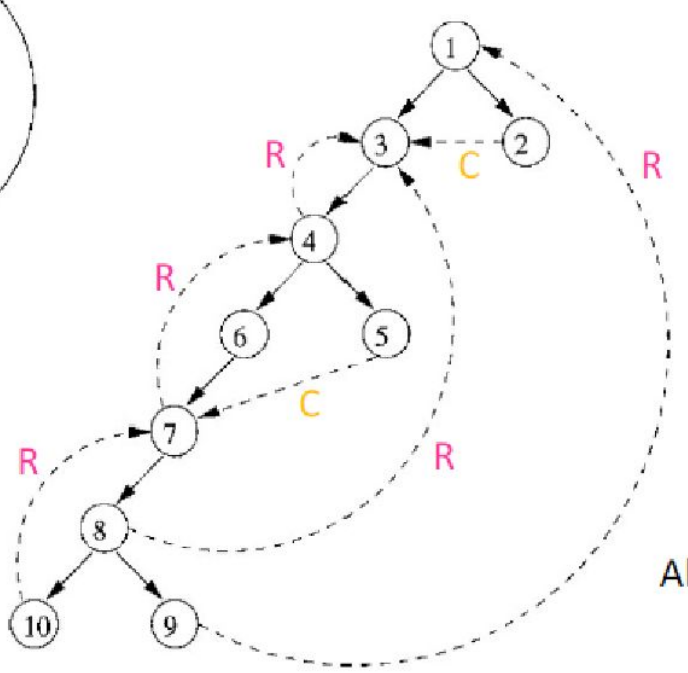
- **Definition**
  - **Back edge**: t->h, h dominates t
- **Relationships between graph edges and back edges**
- **Algorithm**
  - Perform a depth first search
  - For each retreating edge t->h, check if h is in t's dominator list
- **Most programs (all structured code, and most GOTO programs) have reducible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

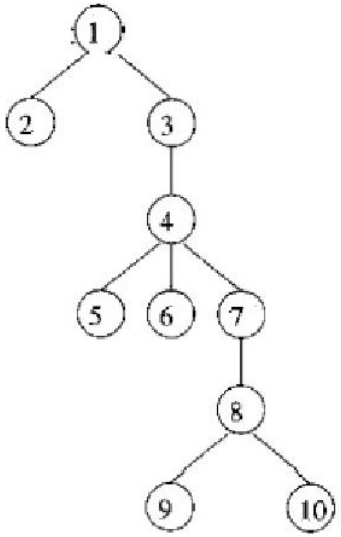


A nonreducible flow graph

Examples



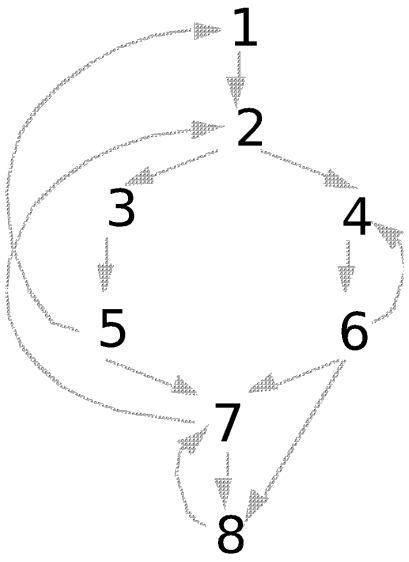




All the retreating edges are back edges

3. Constructing Natural Loops

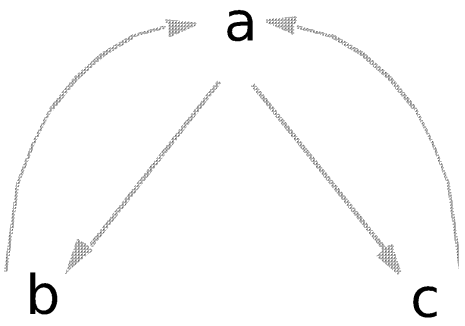
- The **natural loop of a back edge** is the smallest set of nodes that includes the head and tail of the back edge, and has no predecessors outside the set, except for the predecessors of the header.
- **Algorithm**
    - delete *h* from the flow graph
    - find those nodes that can reach *t*
    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



Inner Loops

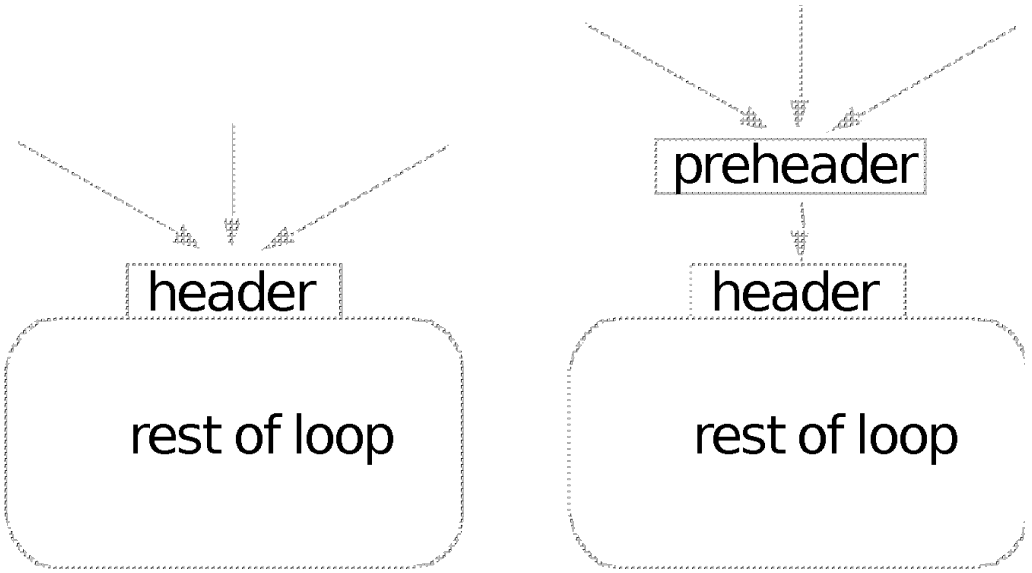
- **If two loops do not have the same header:**
  - they are either disjoint, or
  - one is entirely contained (nested within) the other
    - inner loop: one that contains no other loop.

- **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

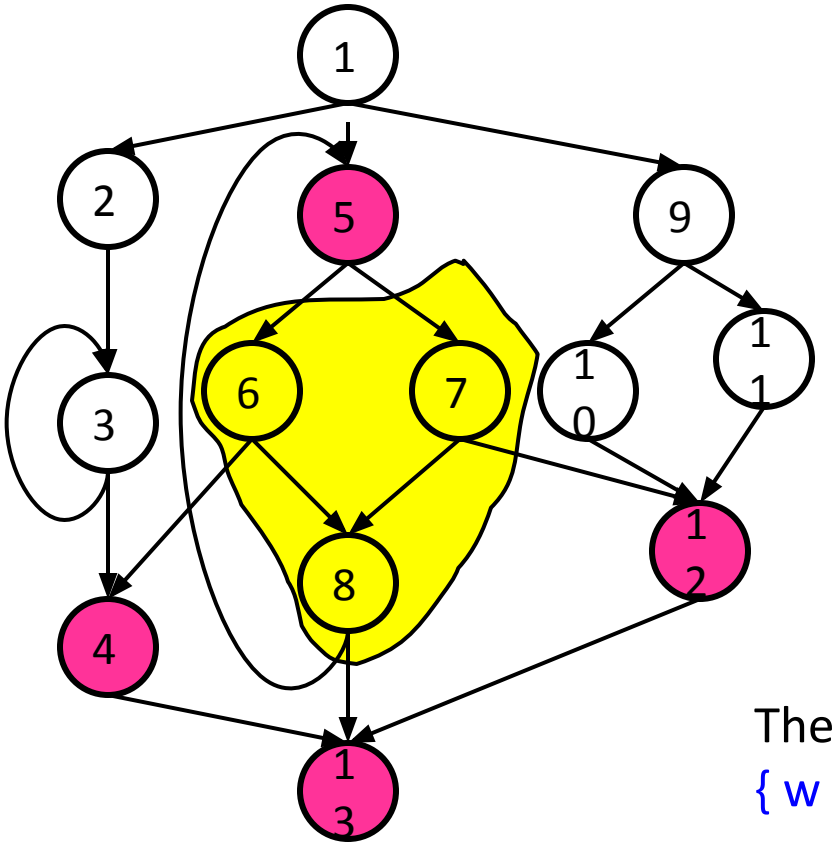
CSC D70: Compiler Optimization Dataflow-2 and Loops

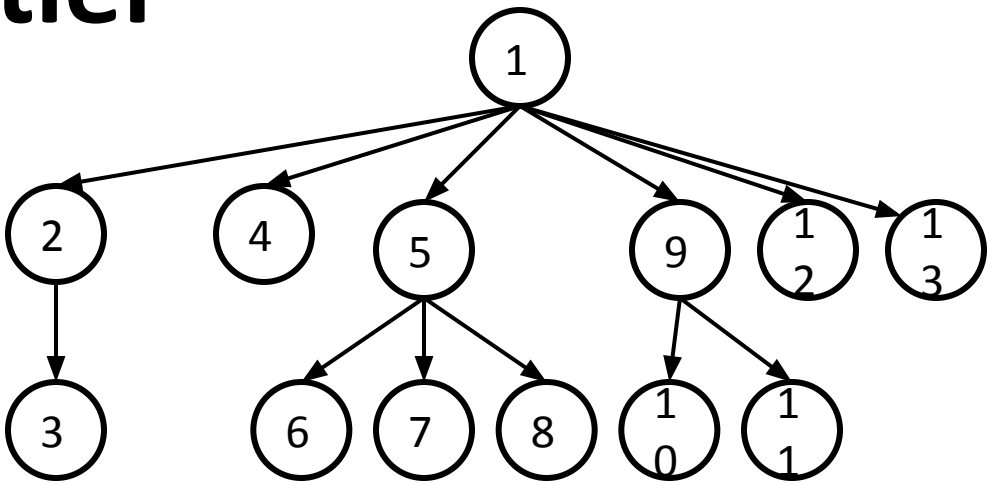
Prof. Gennady Pekhimenko
University of Toronto
Winter 2020

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

Backup Slides

Dominance Frontier





The Dominance Frontier of a node x =
{ w | x dom pred(w) AND !(x sdom w)}

CFG D-Tree

x strictly dominates w (x sdom w) iff x dom w AND x ≠ w

Dominance Frontier and Path Convergence