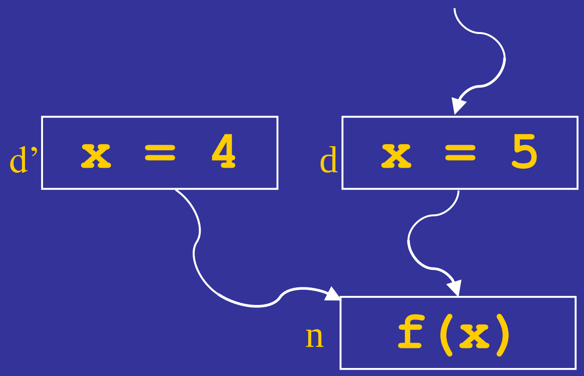
Reaching Definitions: Must or May Analysis?

Consider constant propagation



We need to know if d' might reach node n

Improving Iterative DFA Algorithm

Problem

- If any node's in[] or out[] set changes after an iteration, our algorithm computes all of the equations again, even though many of the equations may not be affected by the change

How can we do better?

Solution

- Use a **work-list** algorithm, which keeps track of those nodes whose out[] sets must be recalculated
- If node n is recomputed **and** its out[] set is found to change, all successors of n are added to the work list
- (For a backwards problem, substitute in[] for out[] and predecessor for successor.)

Work-List Algorithm for IDFA

Algorithm

for each node n

in[n] = υ; out[n] = υ

worklist = {entry node}

while worklist not empty

Remove some node n from worklist

out' = out[n]

in[n] = $\bigcap_{p \in pred[n]}$ out[p]

out[n] = gen[n] $\cup$ (in[n] – kill[n])

if out[n] $\neq$ out'

for each s $\in$ succ[n]

if s $\notin$ worklist, add s to worklist

Is this a forwards or backwards analysis? Is it a must or may analsyis?

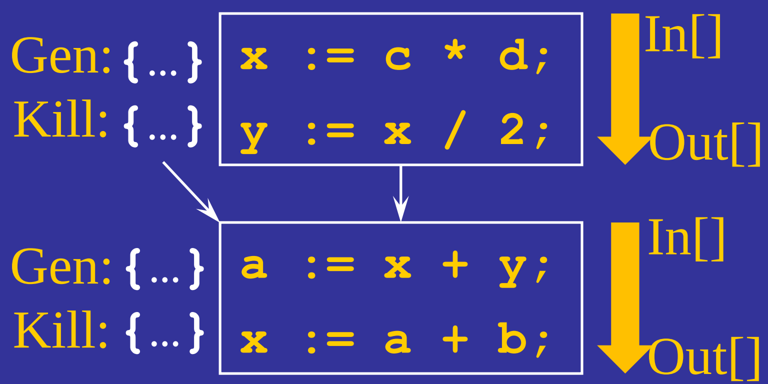
Improving Iterative DFA Algorithm (cont)

Problem

– CFG is bloated when each statement is represented by a node

Solution

– Perform IDFA on CFG of basic blocks



Approach

(1) Build CFG of basic blocks

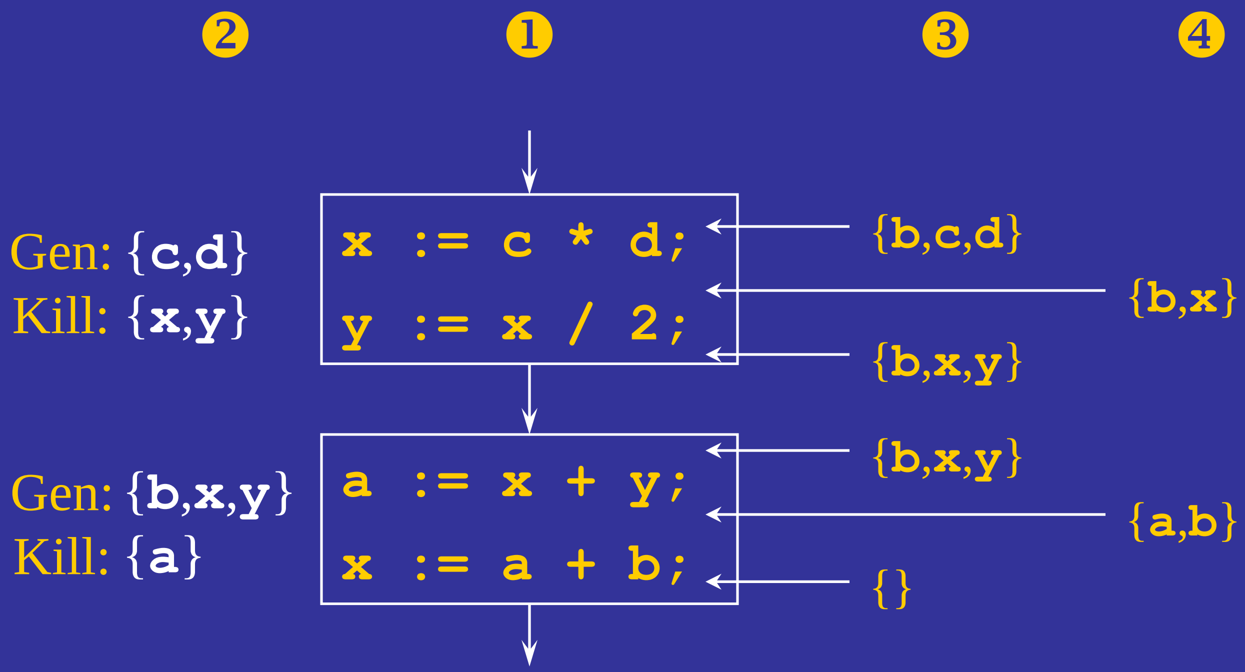
(2) Perform local data-flow analysis within each basic block to summarize Gen and Kill information for each node

(3) Perform global analysis on the smaller CFG

(4) Propagate global information inside of basic block: push information throughout the basic block from the entrance to the exit (or from the exit to the entrance if it's a backwards problem)

Example

Liveness



Reality Check!

Some definitions and uses are ambiguous

- We can't tell whether or what variable is involved
  e.g., `*p = x; /* what variable are we assigning?! */`
- Unambiguous assignments are called **strong updates**
- Ambiguous assignments are called **weak updates**

Solutions

- Be conservative
  - For liveness analysis, if we see `print (*p);`
    what should `*p` refer to?
  - For liveness analysis, if we see `*p = 4;`
    what should `*p` refer to?
- Compute a more precise answer:
  - Pointer analysis (more in a few weeks)

Concepts

Many data-flow analyses have the same character

Computed in the same way

Distinguished by

- Flow values (initial guess, type)
- May/must
- Direction
- Gen
- Kill
- Merge

Complication

- Ambiguous references (strong/weak updates)

Next Time

Lecture

- Lattice theoretic foundation for data-flow analysis

Lattice-Theoretic Framework for Data-Flow Analysis

Last time

– Generalizing data-flow analysis

Today

– Introduce lattice-theoretic framework for data-flow analysis

Today's Lecture

Goals

– Provide a single formal model that describes all data-flow analyses

– Formalize the notions of **safe**, **conservative**, and **optimistic**

– Place bounds on time complexity of data-flow analysis

Approach

– Define **domain** of program properties (**flow values**) computed by data-flow analysis, and organize the domain of elements as a **lattice**

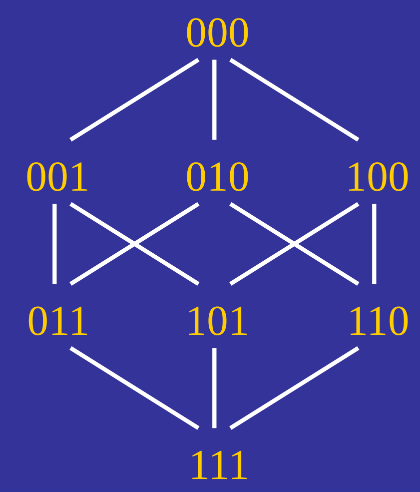
– Define **flow functions** and a **merge function** over this domain using lattice operations

– Exploit lattice theory in achieving goals

Lattices

Define lattice L = (V, ⊓)

– V is a set of elements of the lattice
– ⊓ (**meet** or **greatest lower bound**) is a binary relation over the elements of V



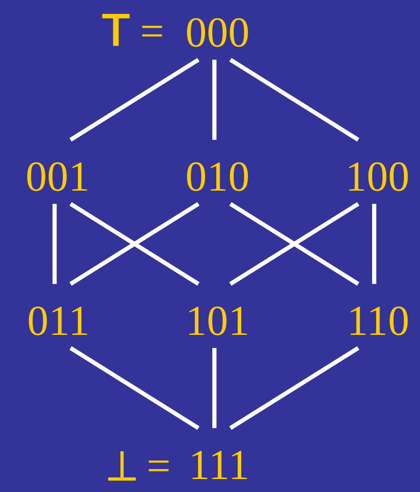
Properties of ⊓

– $x,y \in V \Rightarrow x \sqcap y \in V$ (closure)
– $x,y \in V \Rightarrow x \sqcap y = y \sqcap x$ (commutativity)
– $x,y,z \in V \Rightarrow (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$ (associativity)

Semi-Lattices

– Technically, these are semi-lattices
– A full lattice would also define a join function that allows us to move up the lattice

Lattices (cont)



Under ($\sqsubseteq$)

– Imposes a partial order on V
– $x \sqsubseteq y \Leftrightarrow x \sqcap y = x$

Top (T)

– A unique greatest element of V (if it exists)
– $\forall x \in V - \{T\}, x \sqsubset T$

Bottom ($\perp$)

– A unique least element of V (if it exists)
– $\forall x \in V - \{\perp\}, \perp \sqsubset x$

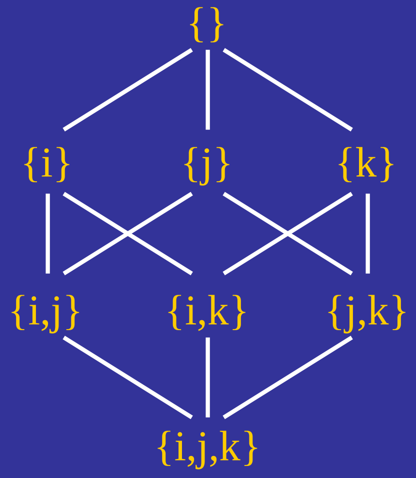
Height of lattice L

– The longest path through the partial order from greatest to least element (top to bottom)

Data-Flow Analysis via Lattices

Relationship

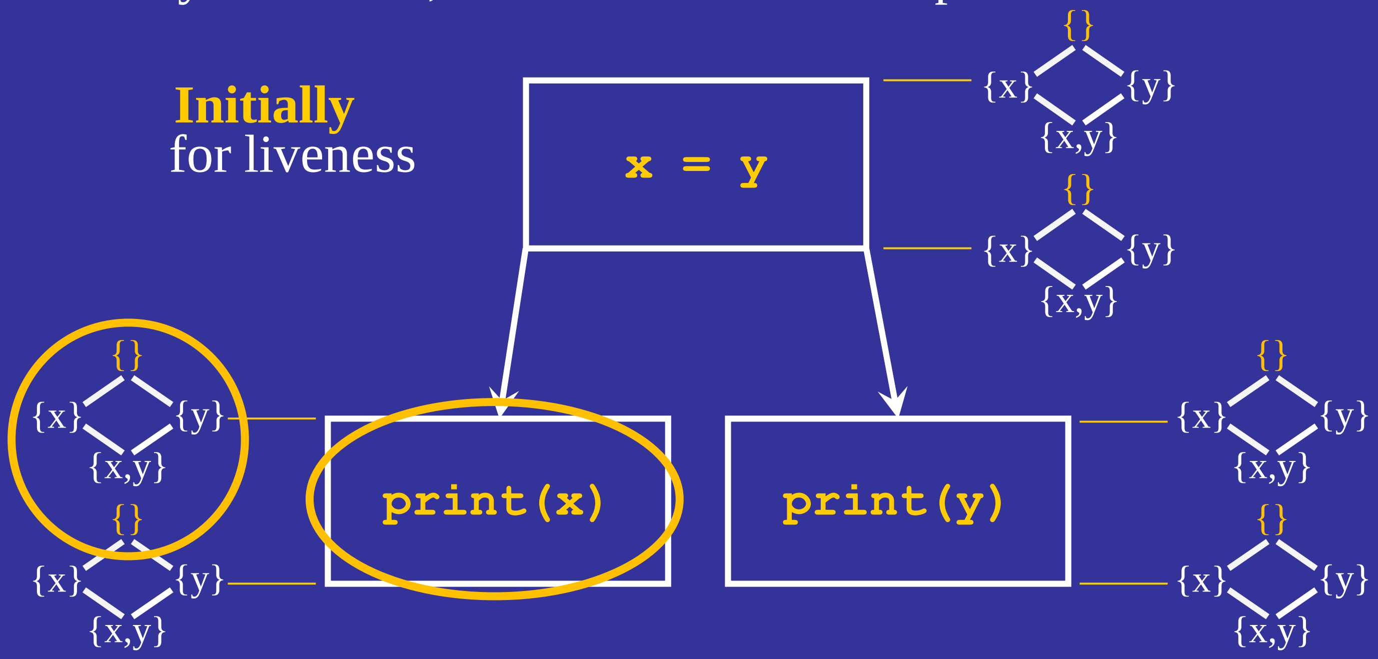
- Elements of the lattice (V) represent flow values (in[] and out[] sets)
  - *e.g.*, Sets of live variables for liveness
- $\top$ represents best-case information (initial flow value)
  - *e.g.*, Empty set
- $\bot$ represents worst-case information
  - *e.g.*, Universal set
- $\sqcap$ (meet) merges flow values
  - *e.g.*, Set union
- If $x \sqsubseteq y$, then x is a conservative approximation of y
  - *e.g.*, Superset



Data-Flow Analysis and the Lattice

Imagine a lattice at every program point

- The lattice element represents an in[] set or an out[] set
- As the analysis iterates, the flow value at each point moves down the lattice



When does the iteration stop?

Data-Flow Analysis Frameworks

Data-flow analysis framework

- A set of **flow values** (V)
- A binary **meet operator** ($\sqcap$)

A lattice

- A set of **flow functions** (F) (also known as **transfer functions**)

Flow Functions

- F = {f: V$\rightarrow$V}
  f describes how each node in CFG affects the flow values
- Flow functions map program behavior onto lattices

Visualizing DFA Frameworks as Lattices

Example: Liveness analysis with 3 variables
$S = \{v1, v2, v3\}$

– V: $\mathbf{2^S =}$ $\{\{v1,v2,v3\},$
$\{v1,v2\},\{v1,v3\},\{v2,v3\},$
$\{v1\},\{v2\},\{v3\}, \varnothing\}$

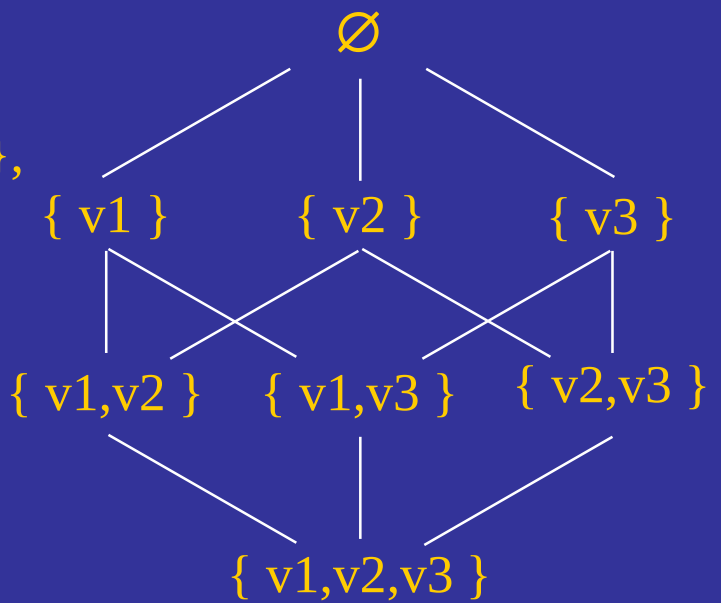
– Meet ($\sqcap$): $\cup$

– $\sqsubseteq$: $\supseteq$

– Top(T): $\varnothing$

– Bottom ($\perp$): υ

– F: $\{f_n(X) = Gen_n \cup (X - Kill_n), \forall n\}$



Inferior solutions are lower on the lattice
More conservative solutions are lower on the lattice

More Examples

Reaching definitions

- V: **2^S (S = set of all defs)**
- $\sqcap$: $\cup$
  - $\sqsubseteq$: $\supseteq$
  - Top($\top$): $\varnothing$
  - Bottom ($\bot$): υ
- F: . . .

Reaching Constants

- V: **$2^{v \times c}$, variables v and constants c**
- $\sqcap$: $\cap$
  - $\sqsubseteq$: $\subseteq$
  - Top($\top$): υ
  - Bottom ($\bot$): $\varnothing$
- F: . . .

Tuples of Lattices

Problem

- Simple analyses may require very complex lattices (*e.g.*, Reaching constants)

Solution

- Use a tuple of lattices, one per variable

$L = (V, \sqcap) \equiv (L_T = (V_T, \sqcap_T))^N$

- $V = (V_T)^N$
- Meet ($\sqcap$): point-wise application of $\sqcap_T$
- $(\ldots, v_i, \ldots) \sqsubseteq (\ldots, u_i, \ldots) \equiv v_i \sqsubseteq_T u_i, \forall i$
- Top ($\top$): tuple of tops ($\top_T$)
- Bottom ($\bot$): tuple of bottoms ($\bot_T$)
- Height $(L) = N \times \text{height}(L_T)$

Tuples of Lattices Example

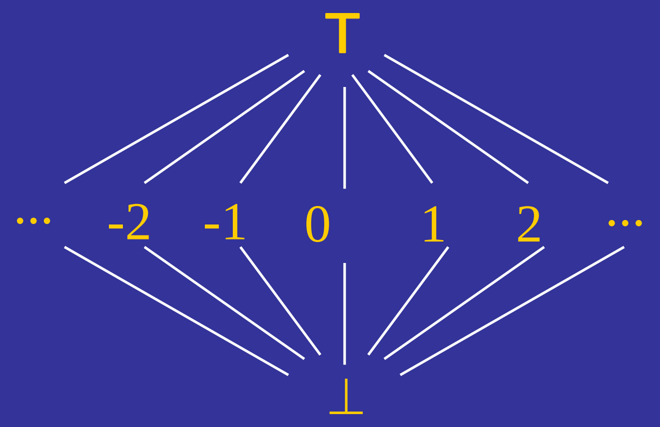
Reaching constants (previously)

– $P = v \times c$, for variables v & constants c

– V: 2^P

Alternatively

– $V = c \cup \{\top, \perp\}$



The whole problem is a tuple of lattices, one lattice for each variable

Examples of Lattice Domains

Two-point lattice ($\top$ and $\perp$)

- Examples?
- Implementation?

Set of incomparable values (and $\top$ and $\perp$)

- Examples?

Powerset lattice (2^S)

- $\top = \emptyset$ and $\perp = S$, or vice versa
- Isomorphic to tuple of two-point lattices

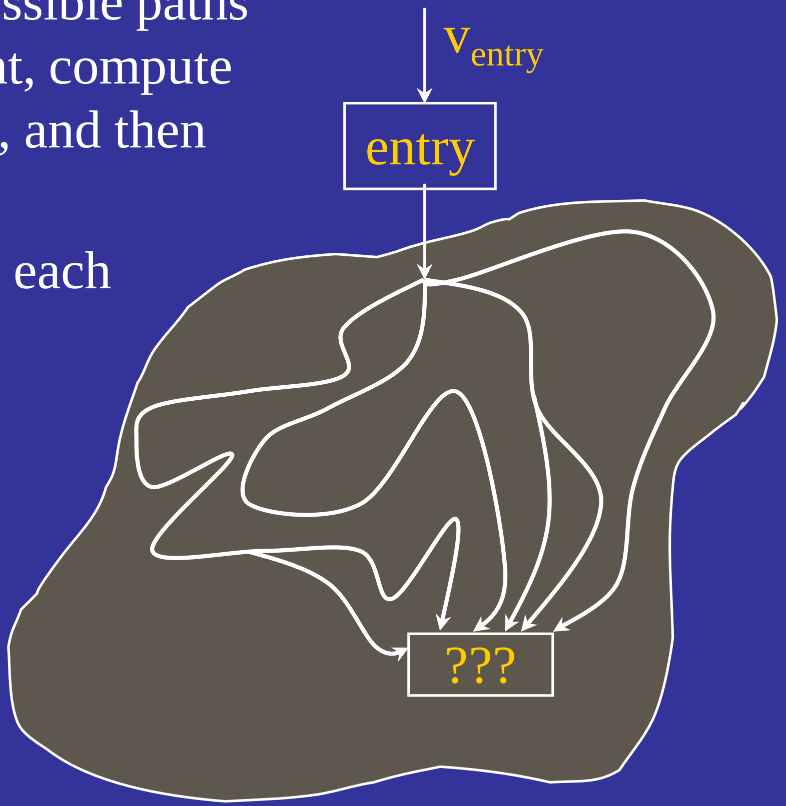
Solving Data-Flow Analyses

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

– $\sqcap_{\text{all paths n1, n2, ..., ni}} (f_{ni}(\ldots f_{n2}(f_{n1}(v_{entry}))))$



Problems with this goal?

Solving Data-Flow Analyses (cont)

Problems

- Loops result in an infinite number of paths
- Statements following merge must be analyzed for all preceding paths
  - Exponential blow-up

Solution

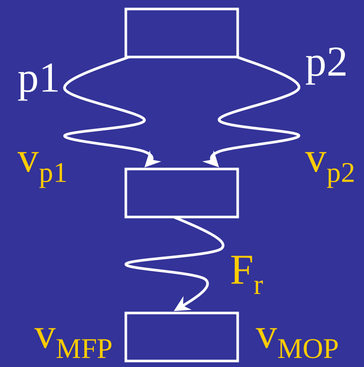
- Compute meets early (at merge points) rather than at the end
- Maximum fixed-point (MFP)

Questions

- Is this solution legal?
- Is this solution efficient?
- Is this solution accurate?

Legality

"Is v_{MFP} legal?" $\equiv$ "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 (really, the flow functions) are monotonic

Reading Assignments

Written responses

- Your reading responses can discuss any of a variety of topics, including the following:
  - You can ask questions about aspects of the paper that you do not understand
  - You can criticize or praise aspects of the paper, including its goals, assumptions, approach, methodology, evaluation, or presentation
  - You can pose questions or suggestions for improving upon or extending the work
  - You can draw connections with previously read papers, previously discussed topics, or previously submitted programming assignments
- Your response does not have to be long (though it might be), but we do hope that it's thoughtful
- Submit your responses using Canvas

Next Time

Assignments

– Assignment 2 is due Friday February 13th at 5:00pm

Reading

– "Finding and Understanding Bugs in C Compilers"
– The reading response is due 5:00pm on Sunday February 15th

Lecture

– Program representations (static single assignment)