Generalizing Data-flow Analysis

Announcements

- Assignment 1 peer review assignments are out and due Thursday Feb 9
- Final revised submission for Assignment 1 is due February 16th
- Assignment 2 is out, and it is strongly recommended that you start working on it.

Today

- Abstracting data-flow analysis
  What's common among the different analyses?
- Lattices for representing the in and out sets in data-flow analysis.
- Tuples of lattices
- Why iterative solutions to data-flow analysis converge

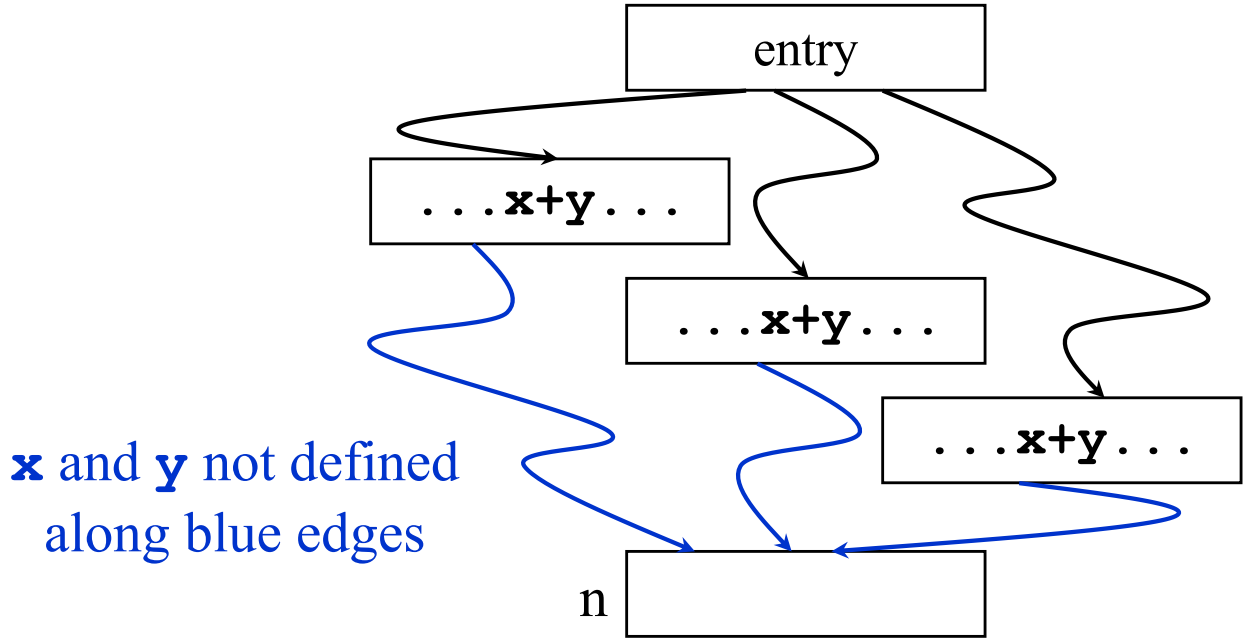
Aspects of Data-flow Analysis

Must or may Information	guaranteed or possible
Direction	forward or backward
Flow values	variables, definitions, ...
Initial guess	universal or empty set
Initial in or out set	universal or empty set
Conservative value	universal or empty set
Kill	due to semantics of stmt what is removed from set
Gen	due to semantics of stmt what is added to set
Merge	how sets from two control paths compose

Available Expressions

Definition

– An expression, **`x+y`**, is **available** at node n if every path from the entry node to n evaluates **`x+y`**, and there are no definitions of **`x`** or **`y`** after the last evaluation



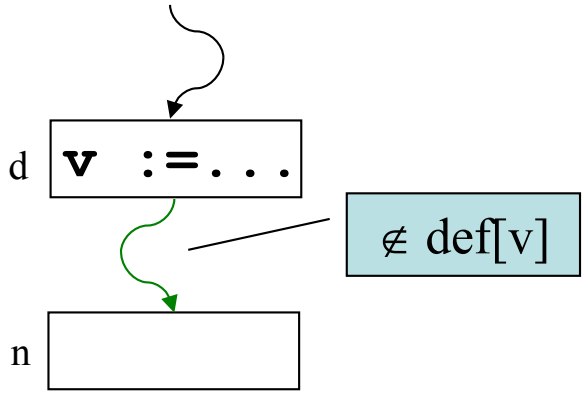
Defining Available Expressions Analysis

Must or may Information?	Must
Direction?	Forward
Flow values?	Sets of expressions
Initial guess?	Universal set
First in set?	Empty set
Conservative value?	Universal set
Kill?	Set of expressions killed by statement s
Gen?	Set of expressions evaluated by s
Merge?	Intersection

Reaching Definitions

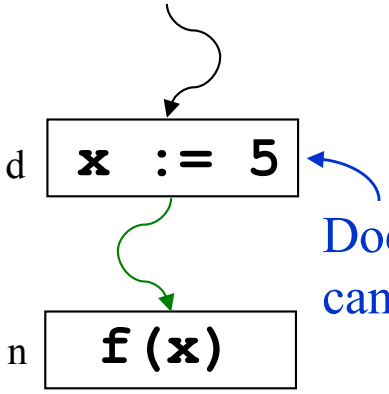
Definition

– A definition (statement) d of a variable **v reaches** node n if there is a path from d to n such that **v** is not redefined along that path



Uses of reaching definitions

– Build use/def chains
– Constant propagation
– Loop invariant code motion



Does this def of **x** reach n?
can we replace n with **f(5)** ?

```
1   a = . . .;
2   b = . . .;
3   for (. . .) {
4       x = a + b;
5       . . .
6   }
```

Reaching definitions of **a** and **b**

To determine whether it's legal to move statement 4 out of the loop, we need to ensure that there are no reaching definitions of **a** or **b** inside the loop

Reaching Constants (aka Constant Propagation)

Goal

- Compute value of each variable at each program point (if possible)

Flow values

- Set of (variable,constant) pairs

Merge function

- Intersection

Data-flow equations

- Effect of node n **`x = c`**
  - kill[n] = {(x,k)| ∀k}
  - gen[n] = {(x,c)}
- Effect of node n **`x = y + z`**
  - kill[n] = {(x,k)| ∀k}
  - gen[n] = {(x,c) | c=valy+valz, (y, valy) ∈ in[n], (z, valz) ∈ in[n]}

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
  - Sometimes we assume that it could be everything
    e.g., Defining `*p` (generating reaching definitions)
  - Sometimes we assume that it is nothing
    e.g., Defining `*p` (killing reaching definitions)
- Try to figure it out: alias/pointer analysis (more later)

Side Effects

What happens at function calls?

- For example, the call foo(&x) might use or define
  - any local or heap variable x that has been passed by address/reference
  - any global variable

Solution

- How do we handle this for liveness used for register allocation?
- In general
  - Be conservative: assume all globals and all vars passed by address/reference may be used and/or modified
  - Or Figure it out: calculate side effects (example of an interprocedural analysis)

Concepts

Data-flow analyses are distinguished by

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

Complication

- Ambiguous references (strong/weak updates)
- Side effects

Context for Lattice-Theoretic Framework

Goals

- Provide a single formal model that describes all data-flow analyses
- Formalize the notions of "correct," "conservative," and "optimistic"
- Correctness proof for IDFA (iterative data-flow analysis)
- Place bounds on time complexity of data-flow analysis

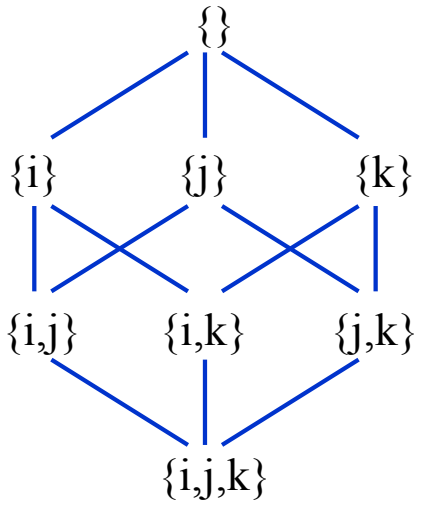
Approach

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



Properties of ⊓

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

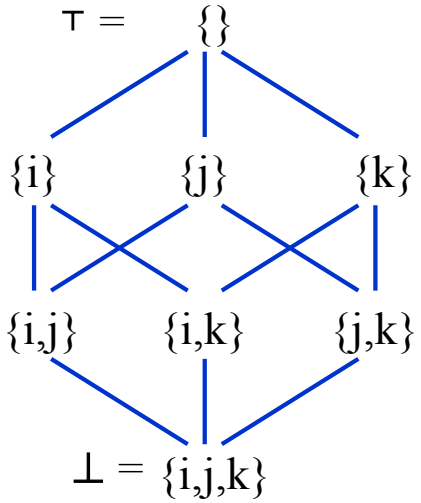
Lattices (cont)

Under ($\sqsubseteq$)

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

Top ($\top$)

- A unique "greatest" element of V (if it exists)
- $\forall x \in V - \{\top\}, x \sqsubseteq \top$



Bottom ($\bot$)

- A unique "least" element of V (if it exists)
- $\forall x \in V - \{\bot\}, \bot \sqsubseteq x$

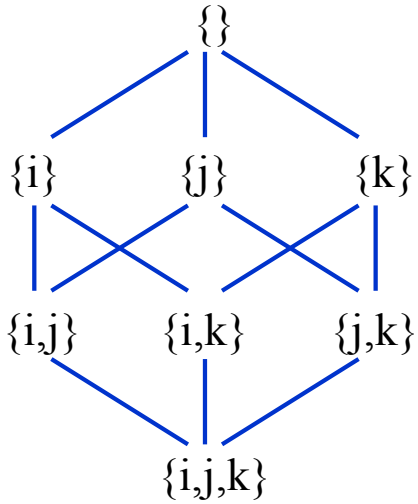
Height of lattice L

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

Data-Flow Analysis via Lattices

Relationship

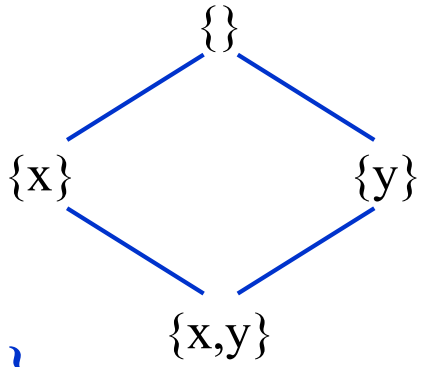
- Elements of the lattice (V) represent flow values (in[] and out[] sets)
  - *e.g.*, Sets of live variables for liveness
- $\top$ represents "best-case" information (initial flow value)
  - *e.g.*, Empty set
- $\bot$ represents "worst-case" information
  - *e.g.*, Universal set
- $\sqcap$ (meet) merges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



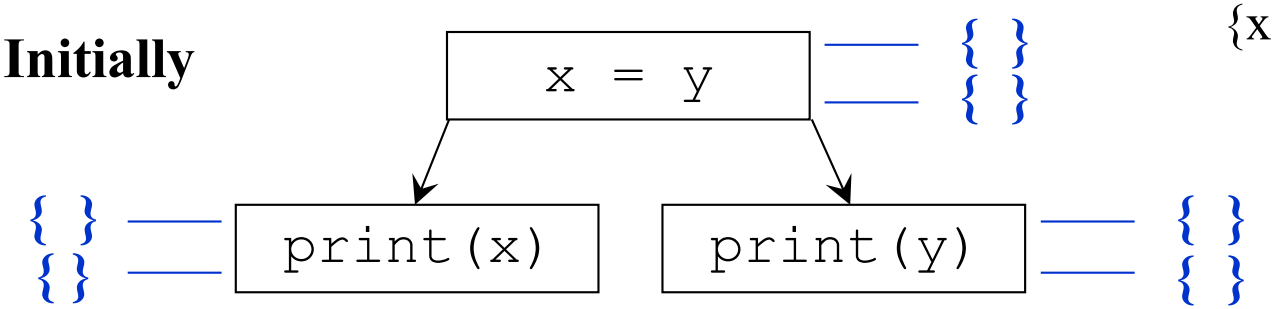
Data-Flow Analysis via Lattices (cont)

Remember what these flow values represent

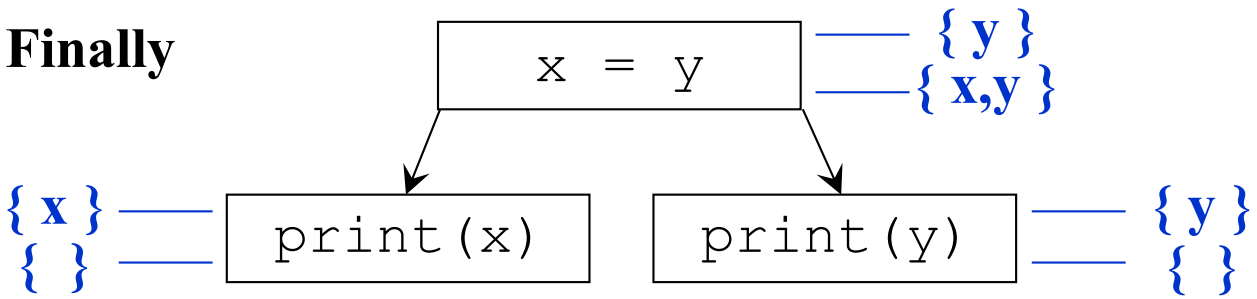
– At each program point a lattice element represents an in[] set or an out[] set



Initially



Finally



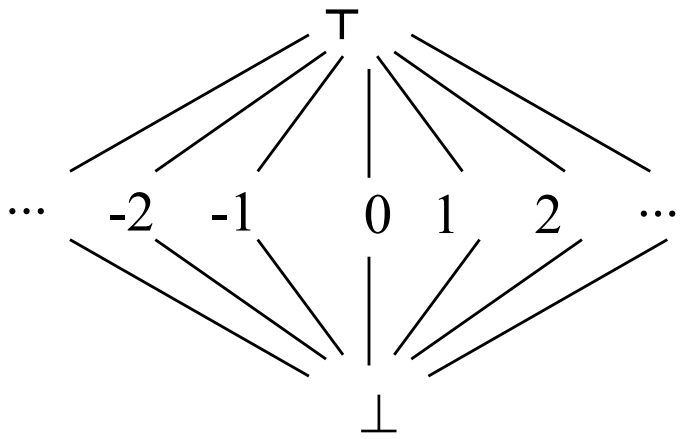
Typical Lattices in Dataflow Analysis

Powerset lattice: set of all subsets of a set U

- meet operator ($\sqcap$) is union ($\cup$) or intersection ($\cap$)
- Partial ordering ($\sqsubseteq$) is $\supseteq$ **or** $\subseteq$
- Bottom ($\bot$) and Top ($\top$) are U and $\varnothing$, or vice versa
- Height = | U | (infinite if U is infinite)

Set of unordered values plus top and bottom

- Example: Reaching constants domain for a particular variable
- Height = 2 (width may be infinite)



Two-point lattice: top and bottom

- Represents a boolean property

Data-Flow Analysis Frameworks

Data-flow analysis framework

- A set of **flow values** (V)
- A binary **meet operator** ($\sqcap$)
- A set of **flow functions** (F) (also known as **transfer functions**)

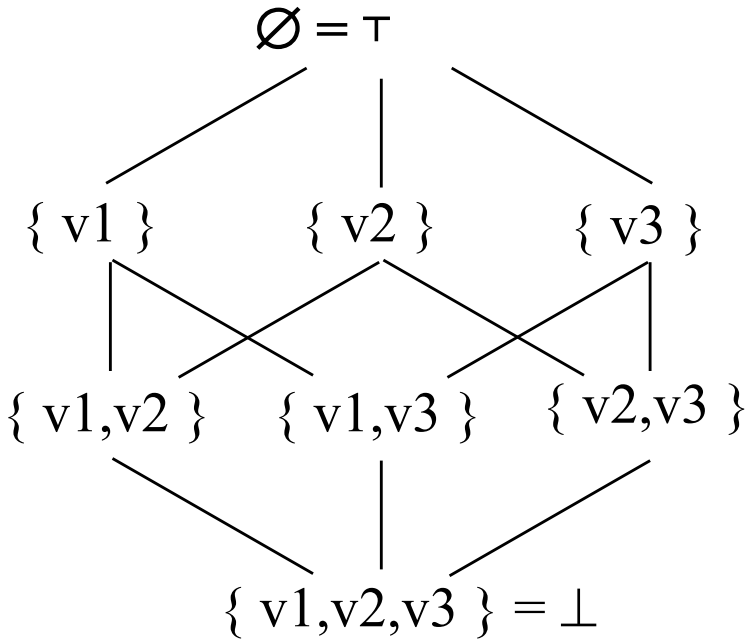
Flow Functions

- $F = \{f: V \rightarrow V\}$
  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
U = {v1, v2, v3}

- V: $\mathbf{2^U} = \{\{v1,v2,v3\}, \{v1,v2\},\{v1,v3\},\{v2,v3\}, \{v1\},\{v2\},\{v3\}, \varnothing\}$
- Meet ($\sqcap$): $\cup$
- $\sqsubseteq$: $\supseteq$
- Top($\top$): $\varnothing$
- Bottom ($\bot$): $\mathbf{U}$
- 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

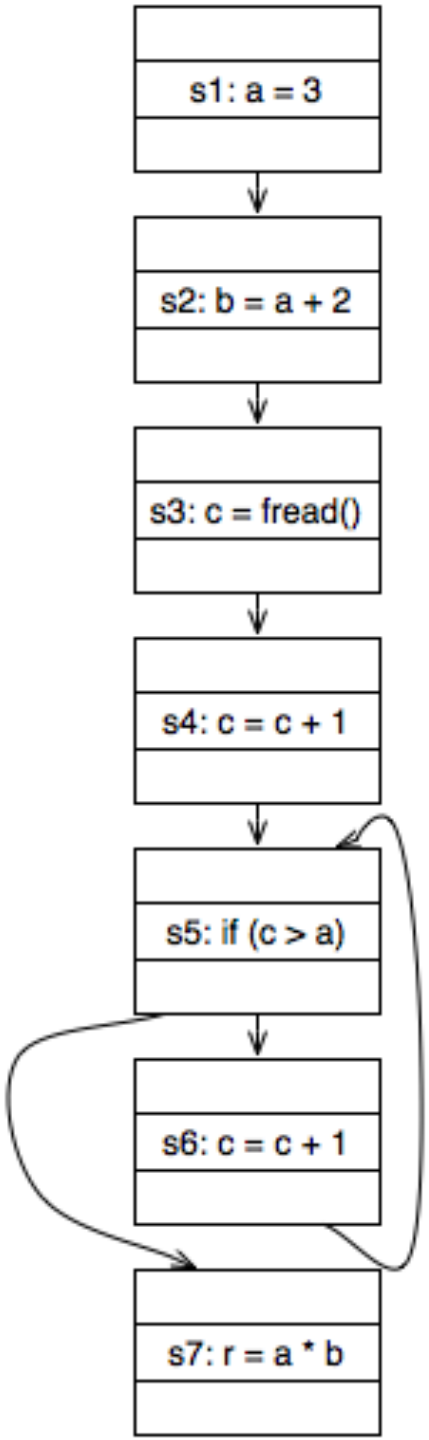
Lattice Example

What are the data-flow sets for liveness?

What is the meet operation for liveness?

What partial order does the meet operation induce?

What is the liveness lattice for this example?



Computing Reaching Definitions

Assumption

- At most one definition per node
- We can refer to definitions by their node "number"

Gen[n]: Definitions that are generated by node n (at most one)

Kill[n]: Definitions that are killed by node n

Defining Gen and Kill for various statement types

statement	Gen[s]	Kill[s]
s: t = b op c	{s}	def[t] – {s}
s: t = M[b]	{s}	def[t] – {s}
s: M[a] = b	{}	{}
s: if a op b goto L	{}	{}

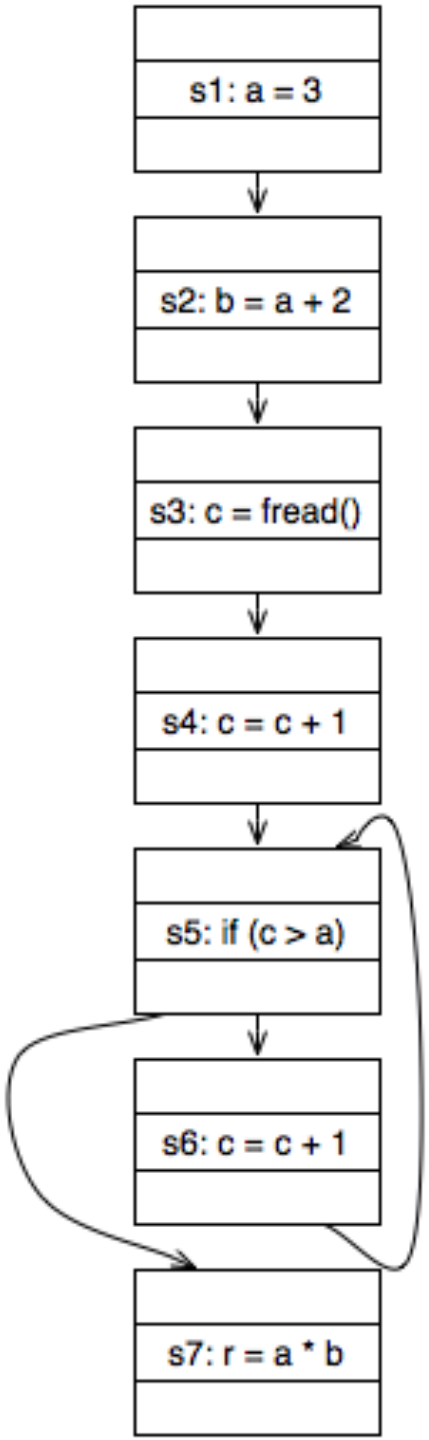
statement	Gen[s]	Kill[s]
s: goto L	{}	{}
s: L:	{}	{}
s: f(a,…)	{}	{}
s: t=f(a, …)	{s}	def[t] – {s}

Reaching Defs Example

What is the lattice?

What is the initial guess?

What is the meet operation?



Another Example

Reaching definitions

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

Reaching Constants (aka Constant Propagation)

Goal

- Compute value of each variable at each program point (if possible)

Flow values

- Set of (variable, constant) pairs

Merge function

- Intersection

Data-flow equations

- Effect of node n **`x = c`**
  - kill[n] = {(x,d)| ∀d}
  - gen[n] = {(x,c)}
- Effect of node n **`x = y + z`**
  - kill[n] = {(x,c)| ∀c}
  - gen[n] = {(x,c) | c=val(y)+valz, (y, valy) ∈ in[n], (z, valz) ∈ in[n]}

Tuples of Lattices

Problem

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

Possible Solutions for reaching constants

– Tuple of lattices, (variable, constant) tuples

– Tuple of lattices, one entry in tuple per variable

$\mathbf{L = (V, \sqcap) \equiv (L_i = (V_i, \sqcap_i))^N}$

– $V = V_1 \times V_2 \times \ldots \times V_N$

– Meet ($\sqcap$): point-wise application of $\sqcap_T$

– $(\ldots, v_i, \ldots) \sqsubseteq (\ldots, u_i, \ldots) \equiv v_i \sqsubseteq u_i, \forall\ i$

– Top ($\top$): tuple of tops $(\top_i)^N$

– Bottom ($\bot$): tuple of bottoms $(\bot_i)^N$

– Height (L) = height(L_1) * height(L_2) * … * height(L_N)

Equivalence of Power Set Lattices and Tuple of two-point lattices (bitvectors)

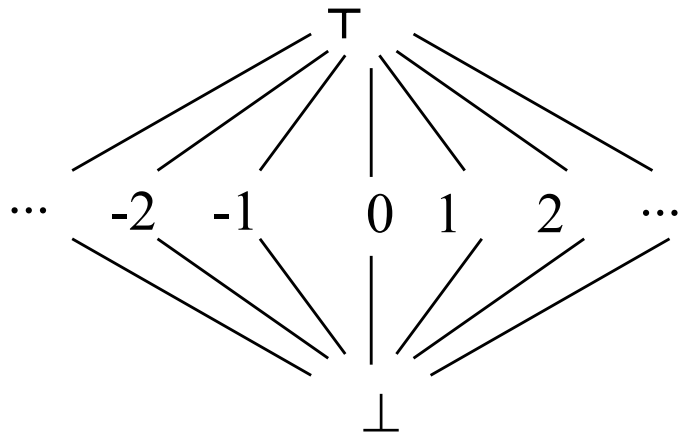
Tuples of Lattices Example

Reaching constants (previously)

- P = v×c, for variables v & constants c
- V: 2^P

Alternatively

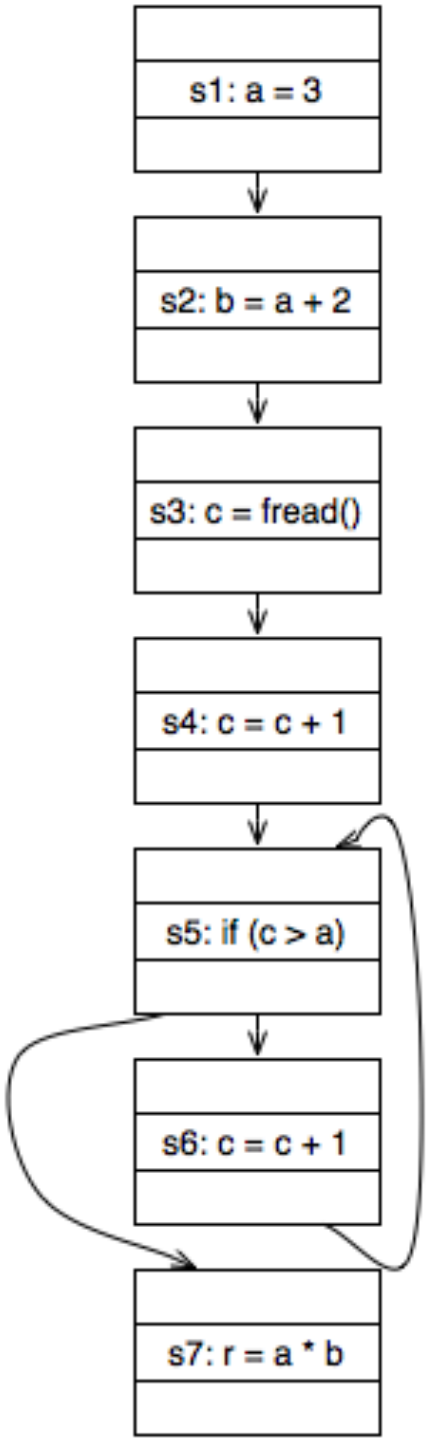
- V = c ∪ {⊤, ⊥}



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

Tuple of Lattices example

For reaching constants, how big is the tuple with entry per variable for this example?



Reaching Constants, Various Ways to do Tuple of Lattices

Reaching Constants

- V: $2^{(C, C, \ldots, C)}$
- $\sqcap$: $\cap$
- $\sqsubseteq$: $\subseteq$
- Top($\top$): **U**
- Bottom ($\bot$): $\varnothing$
- F: . . .

Reaching Constants

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

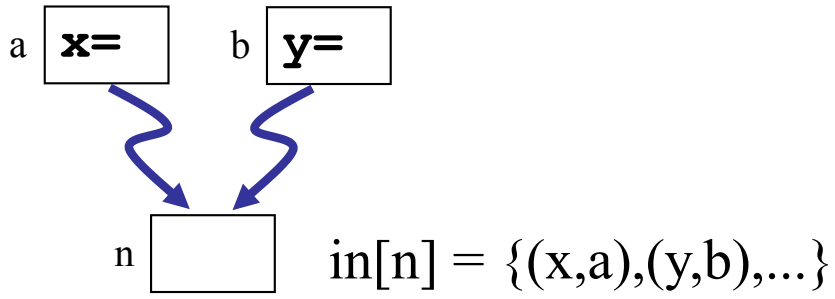
A Better Formulation of Reaching Definitions

Problem

- Reaching definitions gives you a set of definitions (nodes)
- Doesn't tell you what variable is defined
- Expensive to find definitions of variable v

Solution

- Reformulate to include variable
  e.g., Use a set of (var, def) pairs



in[n] = {(x,a),(y,b),...}

Concepts

Lattices

- Conservative approximation
- Optimistic (initial guess)
- Data-flow analysis frameworks
  - Initial in and/or out set
- Tuples of lattices

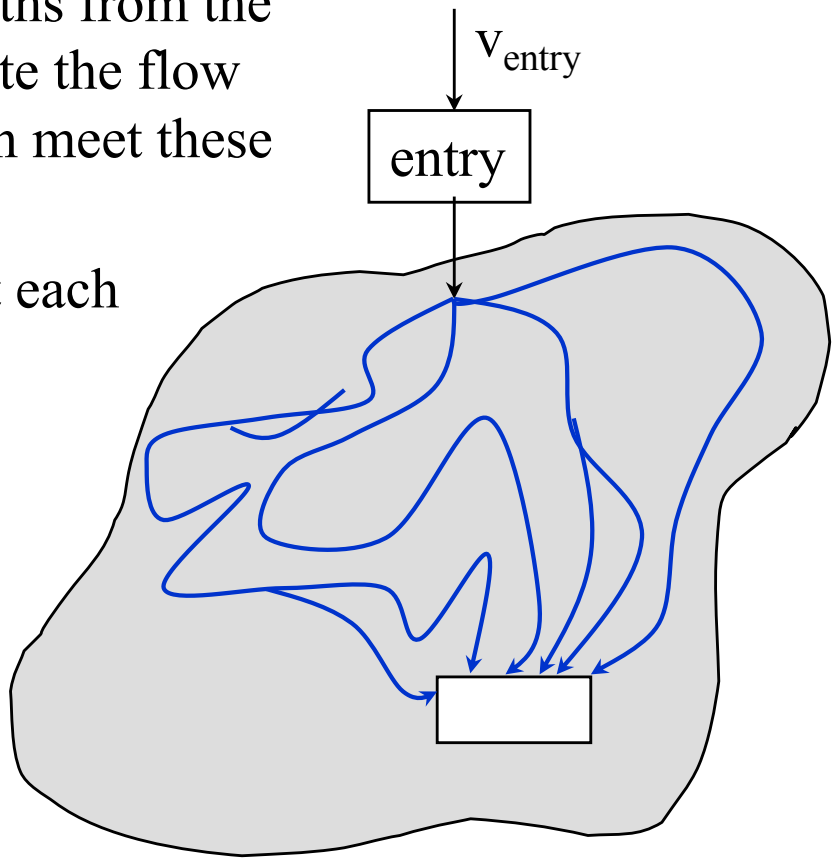
Next

- Iterative dataflow analysis, how do we know it works?

Solving Data-Flow Analyses

Goal

- For a forward problem, consider all 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
- $\sqcap_{\text{all paths } n1, n2, \ldots, ni} (f_{ni}(\ldots f_{n2}(f_{n1}(v_{entry}))))$



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

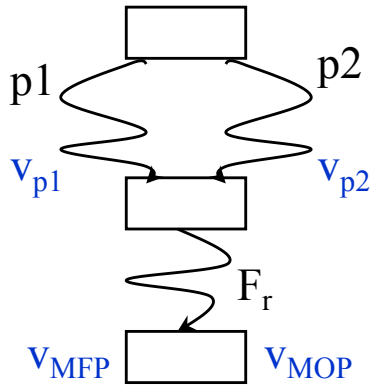
- Compute meets early (at merge points) rather than at the end
- Maximum fixed-point (MFP)

Questions

- Is this correct?
- Is this efficient?
- Is this accurate?

Correctness

"Is v_{MFP} correct?" ≡ "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

$\forall x,y \in V$

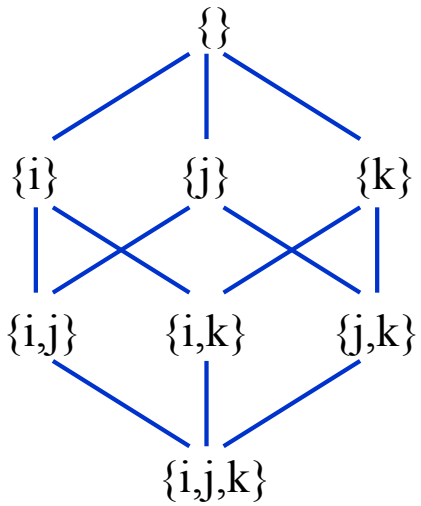
$f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y) \Leftrightarrow x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$

$\therefore$ v_{MFP} correct when F_r (really, 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

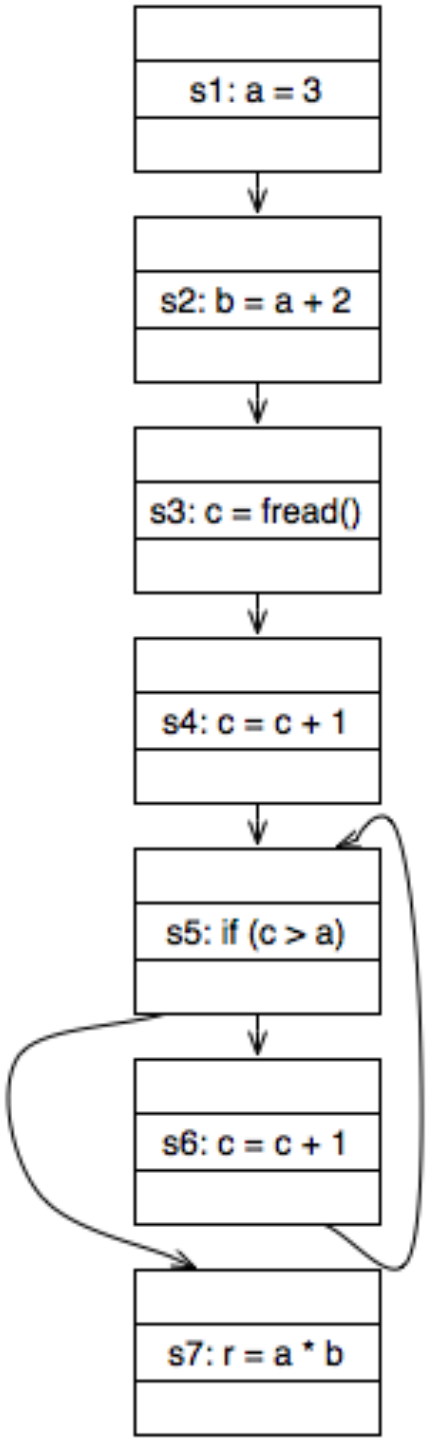
- Reaching definitions?

Reaching Defs Example

What is the height of the lattice?

How many passes over the nodes are necessary?

What if we visit the nodes in a non-optimal order?



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

- Reaching definitions?
- Reaching constants?

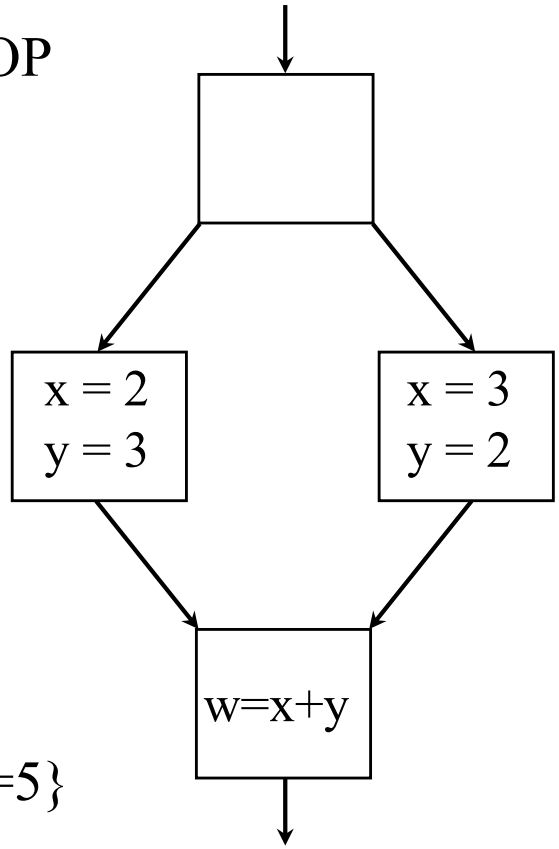
$f(u \sqcap v) = f(\{x=2,y=3\} \sqcap \{x=3,y=2\})$

$= f(\varnothing) = \varnothing$

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

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

$\Rightarrow$ MFP $\neq$ MOP



Concepts

Lattices

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

Lattice Theoretic framework for common subexpression elimination

Data-flow analysis

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