Generalizing Data-flow Analysis

Announcements

- PA1 grades have been posted

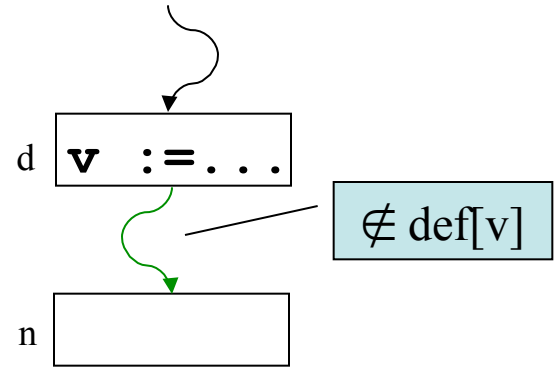
Today

- Other types of data-flow analysis
  - Reaching definitions, available expressions, reaching constants
- Abstracting data-flow analysis
  What's common among the different analyses?

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

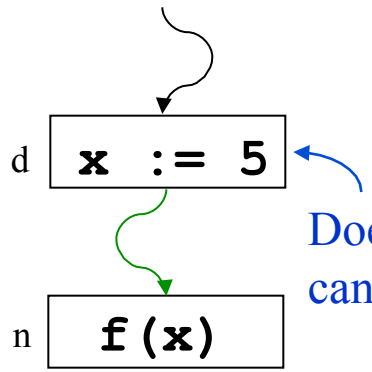


Uses of reaching definitions

– 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

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}

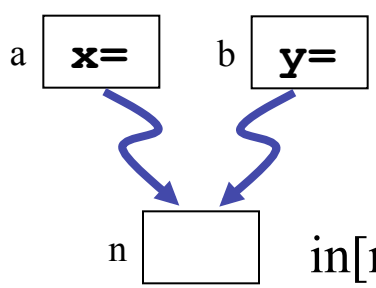
A Better Formulation of Reaching Definitions

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

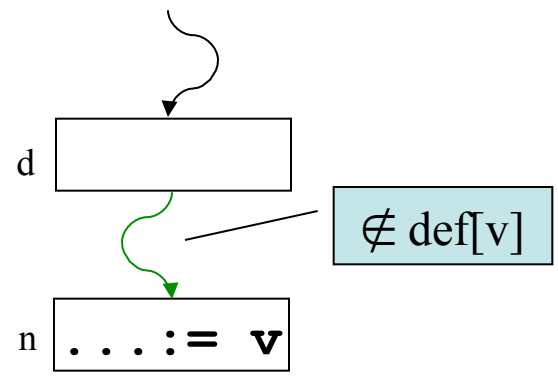


in[n] = {(x,a),(y,b),...}

Recall Liveness Analysis

Definition

– A variable is **live** at a particular point in the program if its value at that point will be used in the future (**dead**, otherwise).



Uses of Liveness

– Register allocation

– Dead-code elimination

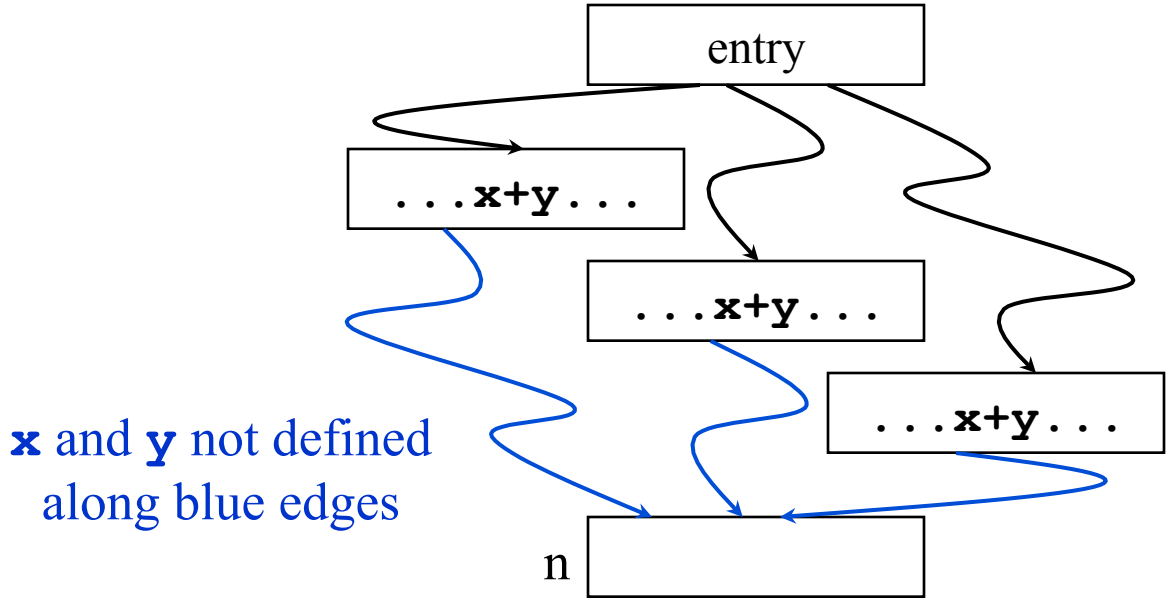
```
1   a = . . .;
2   b = . . .;
3   ...
4   x = f(b);
```

If **a** is not live out of statement 1 then statement 1 is dead code.

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



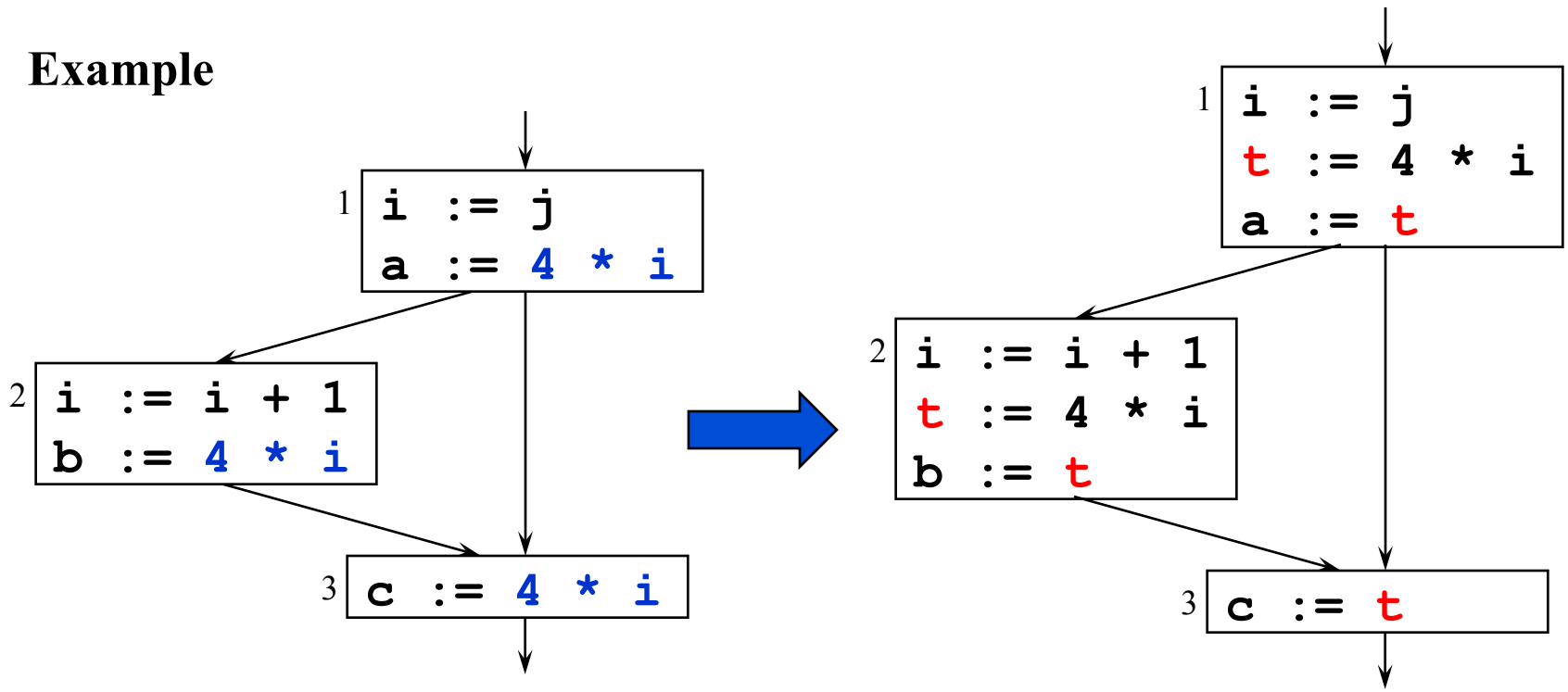
Available Expressions for CSE

How is this information useful?

Common Subexpression Elimination (CSE)

– If an expression is available at a point where it is evaluated, it need not be recomputed

Example



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
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

Must vs. May Information

Must information

- Implies a guarantee

May information

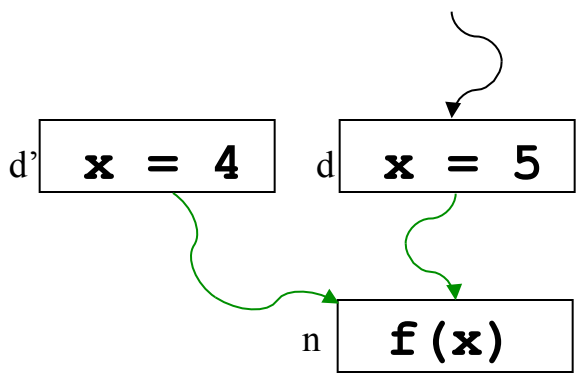
- Identifies possibilities

Liveness? Available expressions?

	May	Must
safe	overly large set	overly small set
desired information	small set	large set
Gen	add everything that might be true	add only facts that are guaranteed to be true
Kill	remove only facts that are guaranteed to be true	remove everything that might be false
merge	union	intersection
initial guess	empty set	universal set, except start or end

Reaching Definitions: Must or May Analysis?

Consider uses of reaching definitions

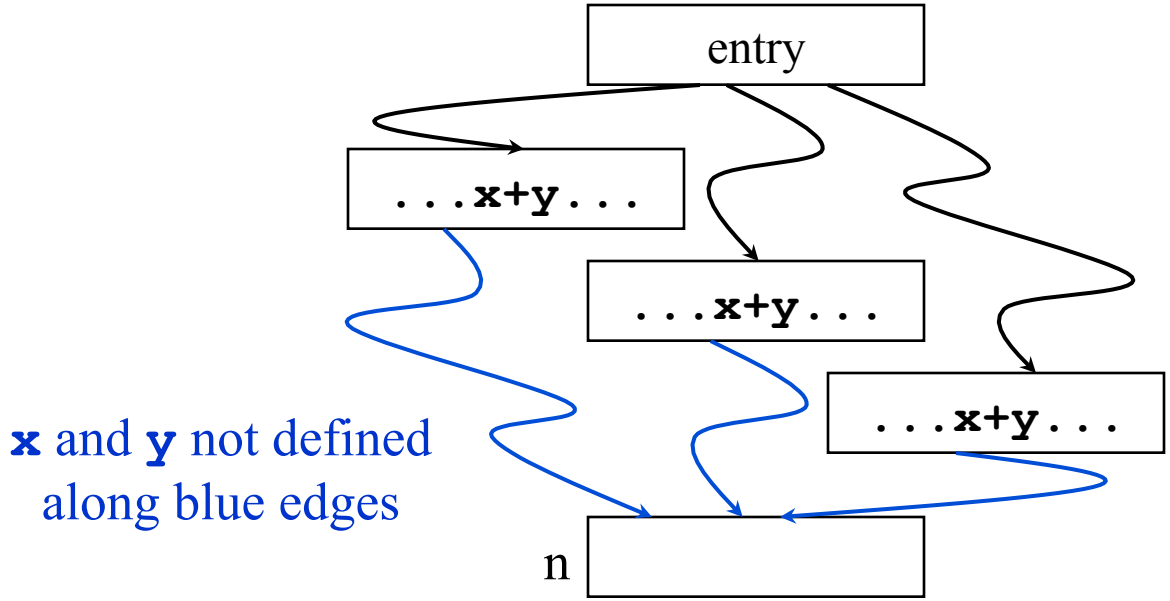


We need to know if d' might reach node n

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
Kill?	Set of expressions killed by statement s
Gen?	Set of expressions evaluated by s
Merge?	Intersection

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

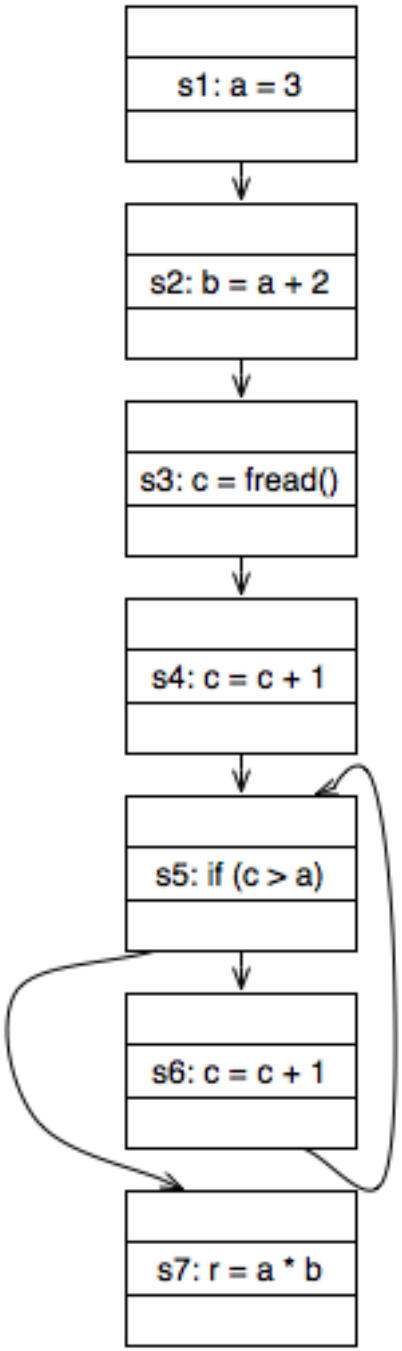
- Effect of node n `x = c`
  - kill[n] = {(x,k)| ∀k}
  - gen[n] = {(x,c)}
- Effect of node n `x = y + z`
  - kill[n] = {(x,k)| ∀k}
  - gen[n] = {(x,c) | c=valy+valz, (y, valy) ∈ in[n], (z, valz) ∈ in[n]}

Reaching Constants Example

Must or may info?

Direction?

Initial guess?



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

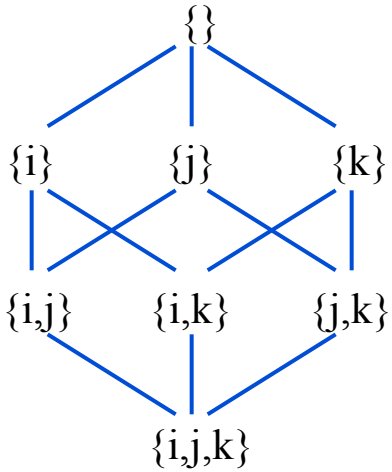
Approach

- 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 $L = (V, \sqcap)$

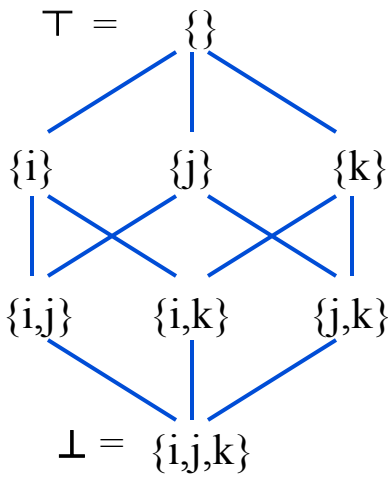
- V is a set of elements of the lattice
- $\sqcap$ is a binary relation over the elements of V (**meet** or **greatest lower bound**)



Properties of $\sqcap$

- $x,y \in V \Rightarrow x \sqcap y \in V$ (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 (⊑)

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

Top (⊤)

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

Bottom (⊥)

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

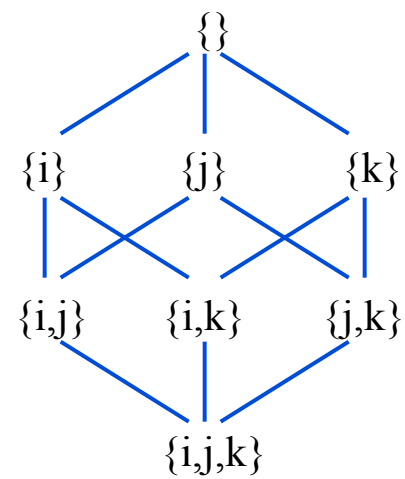
Height of lattice L

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

Data-Flow Analysis via Lattices

Relationship

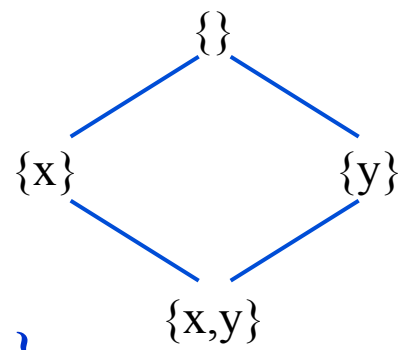
- Elements of the lattice (V) represent flow values (in[] and out[] sets)
  - *e.g.*, Sets of live variables for liveness
- $\top$ represents "best-case" information (initial flow value)
  - *e.g.*, Empty set
- $\bot$ represents "worst-case" information
  - *e.g.*, Universal set
- $\sqcap$ (meet) merges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



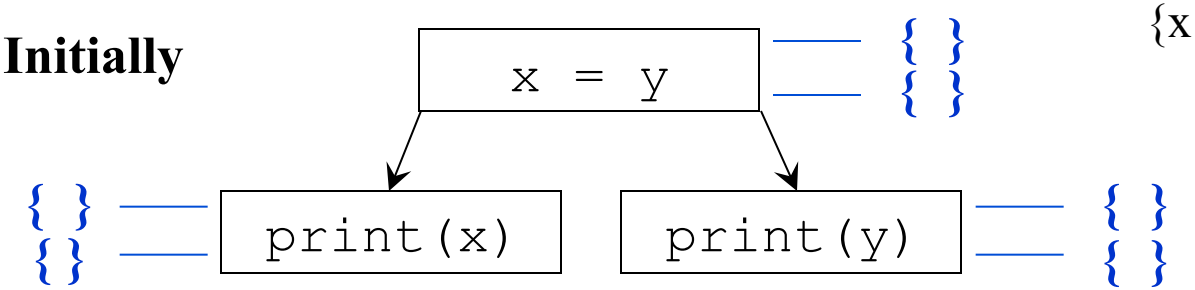
Data-Flow Analysis via Lattices (cont)

Remember what these flow values represent

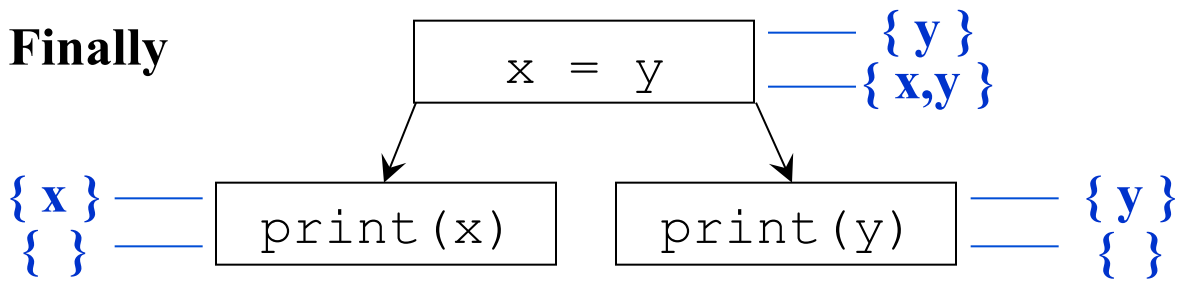
– At each program point a lattice element represents an in[] set or an out[] set



Initially



Finally



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**)

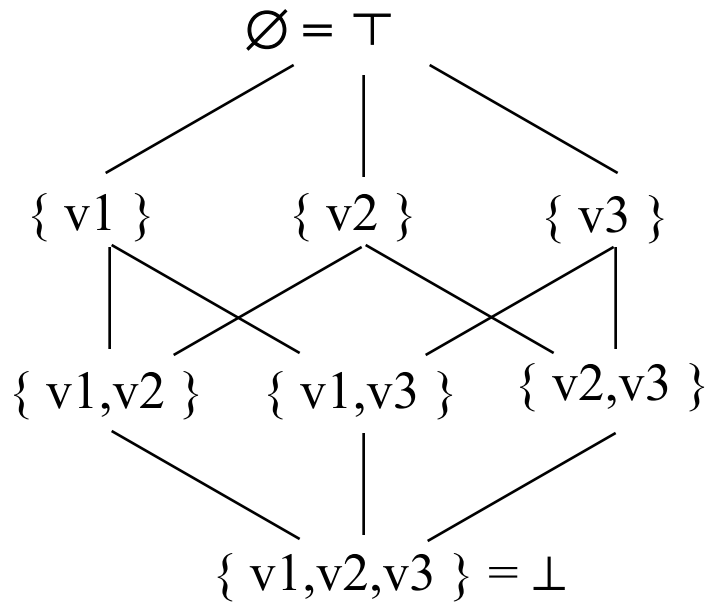
Flow Functions

- $F = \{f: V \rightarrow V\}$
  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: $2^U =$ {{v1,v2,v3},
  {v1,v2},{v1,v3},{v2,v3},
  {v1},{v2},{v3}, ∅}
- Meet (⊓): ∪
- ⊑: ⊇
- Top(⊤): ∅
- Bottom (⊥): **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

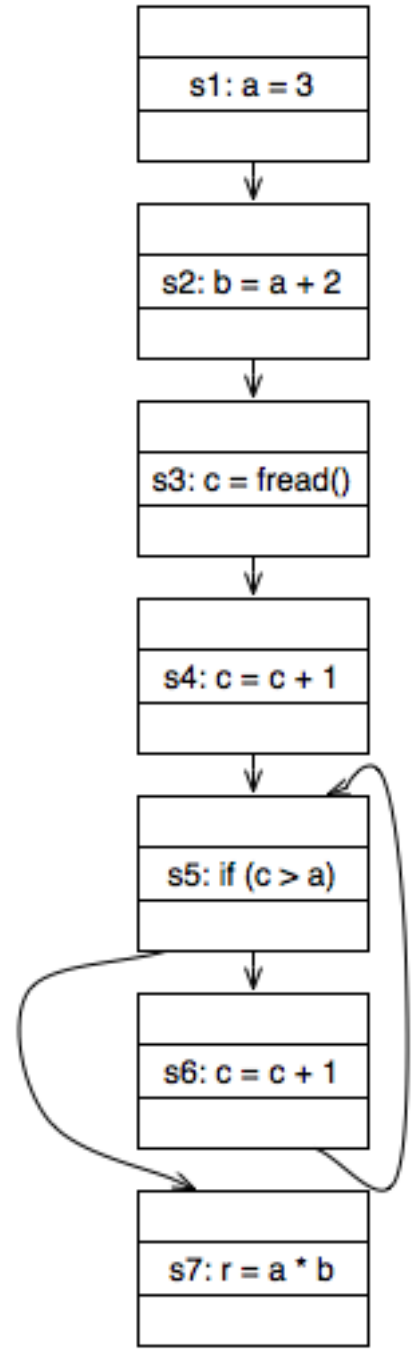
Lattice Example

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	{}	{}

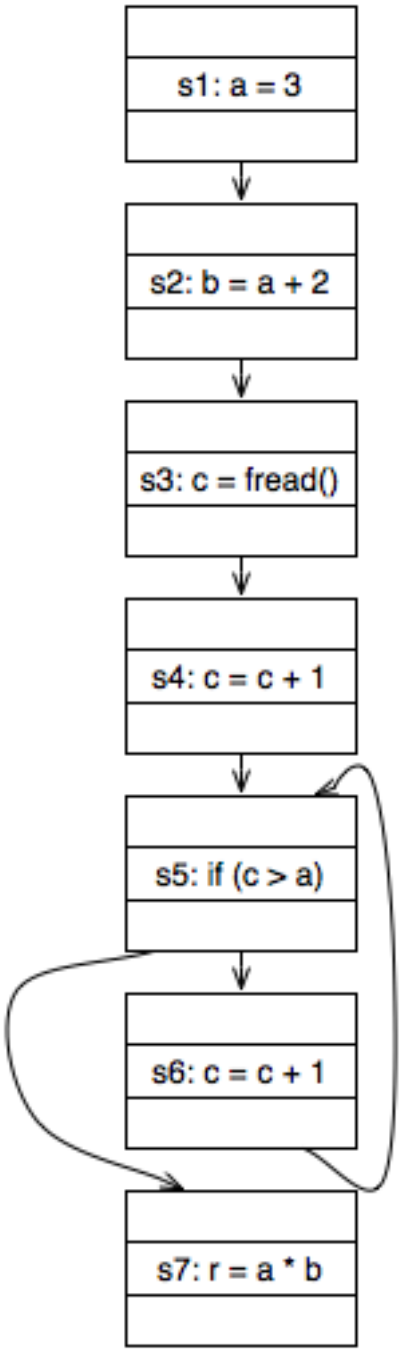
statement	Gen[s]	Kill[s]
s: goto L	{}	{}
s: L:	{}	{}
s: f(a,…)	{}	{}
s: t=f(a, …)	{s}	def[t] – {s}

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 ($\perp$): **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

- Use of tuple of lattices, (variable, constant) tuples
- Use a tuple of lattices, one entry in tuple per variable

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

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

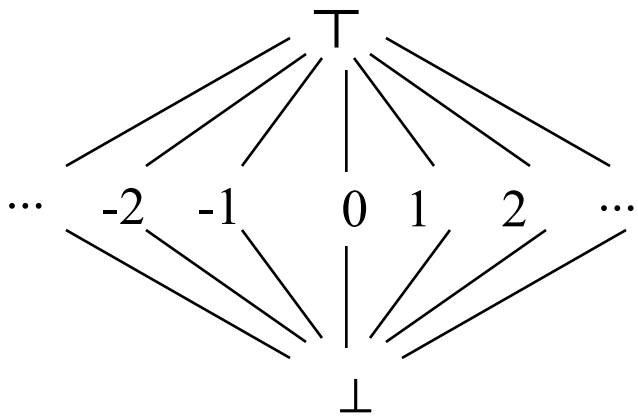
Tuples of Lattices Example

Reaching constants (previously)

- $P = v \times c$, for variables v & constants c
- $V: 2^P$

Alternatively

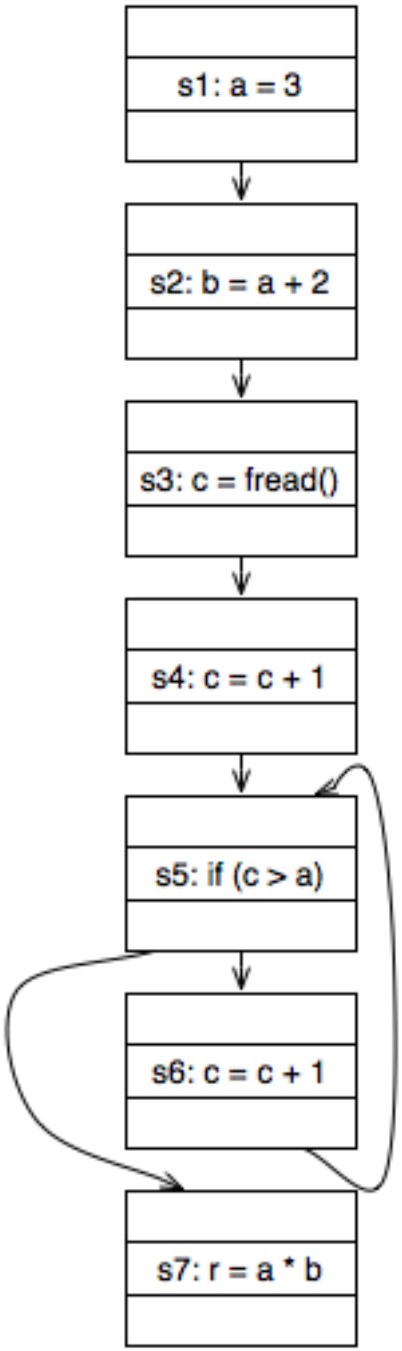
- $V = c \cup \{\top, \bot\}$



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 ($\perp$): $\varnothing$
- F: . . .

Reaching Constants

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

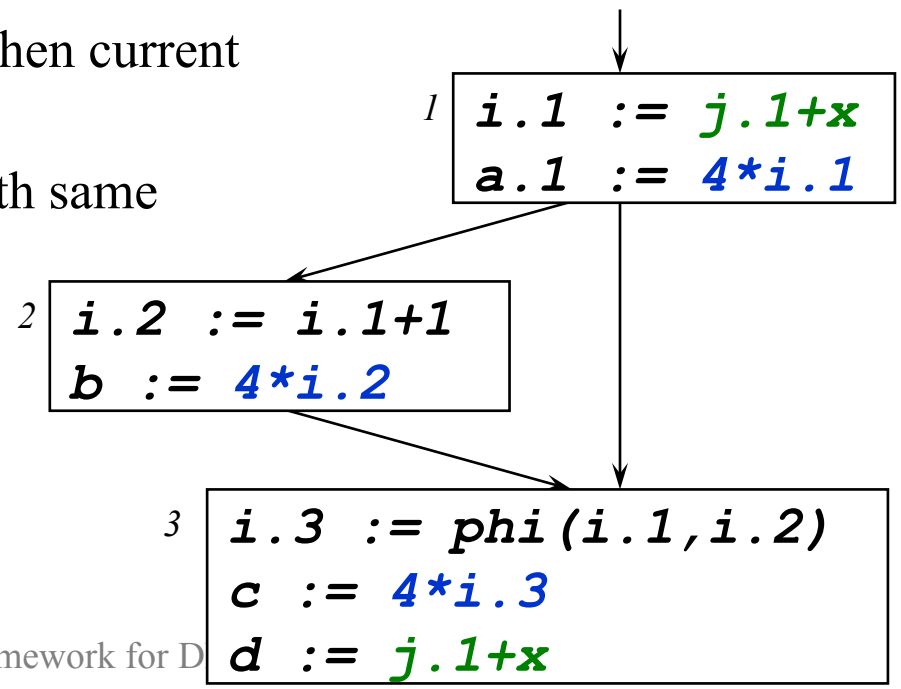
CSE in SSA and LLVM (PA2)

Available Expressions: Possible approach

- Going to miss out on some possibilities.
- Representation for expressions, define operator<
- If gen'ed expressions not in avail_in set
  - Map each gen'ed expression to instruction pointer that creates it
  - Put expression in avail_out set
- If gen'ed expression is in avail_in set, then current instruction is unnecessary
- Special set intersection, expressions with same define are equivalent

Lattice Theoretic Framework Approach

- Effect of instruction n: **`x = e`**
  - kill[n] = {(e, d)| ∀d}
  - gen[n] = {(x, n)}



PA2: Common Subexpression Elimination

Available Expressions (non-SSA)

- V: 2^S **(S = set of all expressions)**
- $\sqcap$: $\cap$
- $\sqsubseteq$: $\subseteq$
- Top($\top$): **U**
- Bottom ($\perp$): $\varnothing$
- F: . . .
- Start value: $\varnothing$

Reaching Constants

- V: $2^{S \times I}$**, expressions S and instructions I**
- $\sqcap$: $\cap$
- $\sqsubseteq$: $\subseteq$
- Top($\top$): **U**
- Bottom ($\perp$): $\varnothing$
- F: . . .
- Start value: $\varnothing$