Dataflow Analysis
Lattices & Solvers

15-411/15-611 Compiler Design

Seth Copen Goldstein

September 29, 2020

Dataflow Analysis

- A framework for proving facts about program
  - Reasons about lots of little facts
  - Little or no interaction between facts
  - Based on all paths through program
- Solve with iterative solver:
  - How do we know it terminates?
  - How do we know whether solution is precise? (or even correct?)

Recall: Data Flow Equations

- Let s be a statement
  - Succ(s) = {immediate successors of s}
  - Pred(s) = {immediate predecessors of s}
  - In(s) program point just before executing s
  - Out(s) program point just after executing s
- Transfer functions (for forward, must):

$$In(s) = \bigcap_{s' \in \mathrm{pred}(s)} Out(s')$$

$$Out(s) = Gen(s) \cup (In(s) - Kill(s))$$

- Gen(s) set of facts made true by s
- Kill(s) set of facts invalidated by s

Recall: Worklist algorithm (forward)

```
Initialize: in[B] = out[b] = Universe
Initialize: in[entry] = ∅
Work queue, W = all Blocks in topological order
while (|W| != 0) {
    remove b from W
    temp = out[b]
    compute In[b]
    compute Out[b]
    if (temp != out[b]) W = W ∪ succ(b)
}
```

Some Unidirectional Dataflow Analysis

	Union (may)	intersection (must)
Forward	Reaching definitions	Available expressions
Backward	Live variables	very busy expressions

Available Expressions

- X+Y is "available" at statement S if
  - x+y is computed along every path from the start to S AND
  - neither x nor y is modified after the last evaluation of x+y

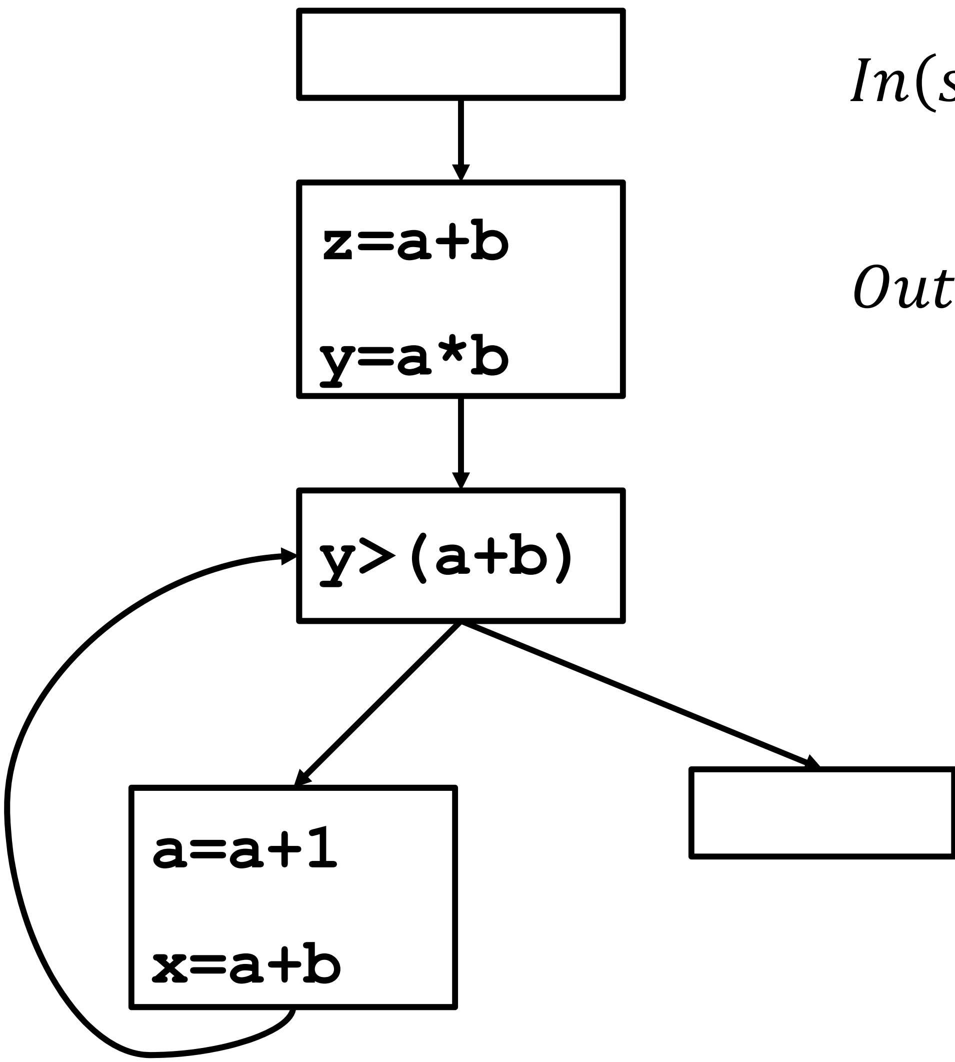
a <- b+c

b <- a-d

c <- b+c ⟵ b+c Not available, since b redefine

d <- a-d ⟵ a-d is available

Available Expressions



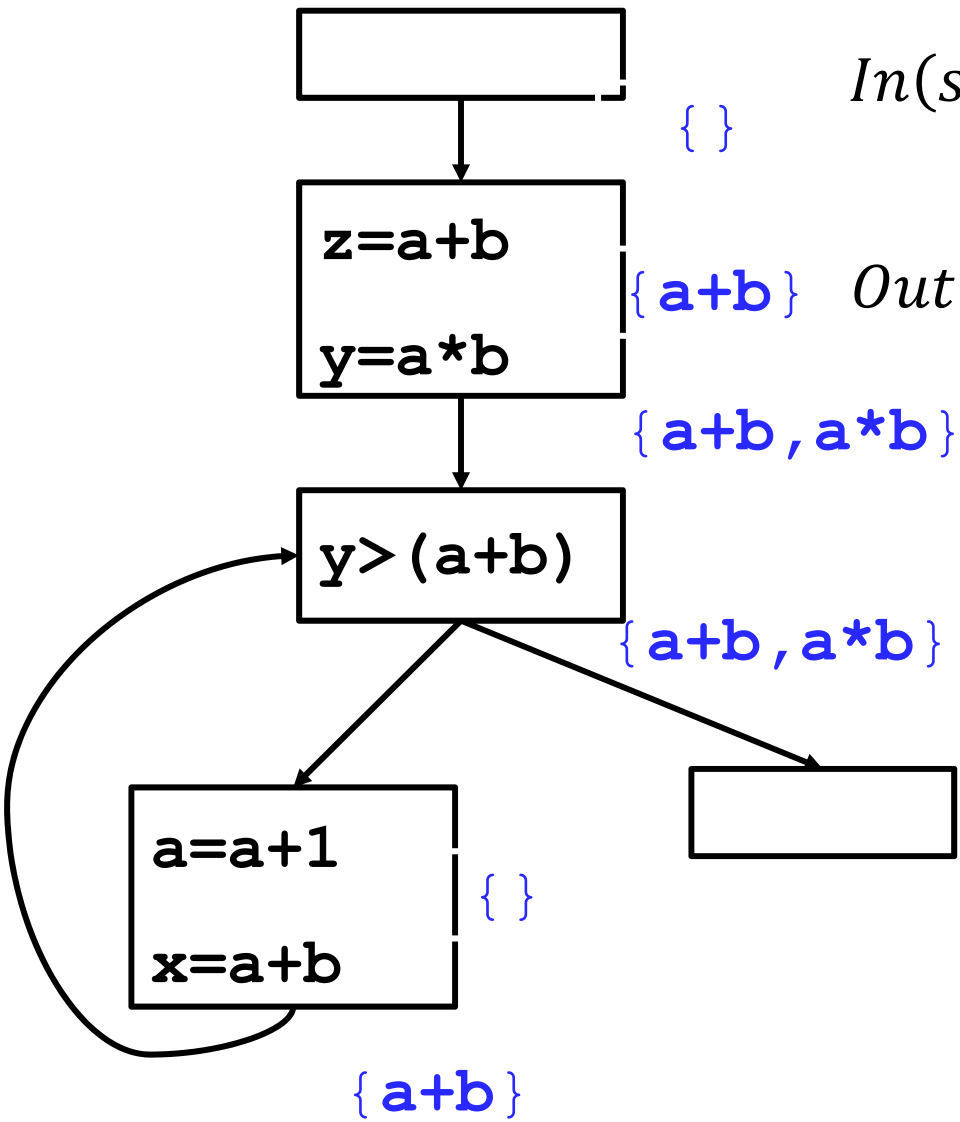
$$In(s) = \bigcap_{s' \in \mathrm{pred}(s)} Out(s')$$

$$Out(s) = \big(In(s) \cup Gen(s)\big) - Kill(s)$$

For x= a⊕b:
- Gen = {a⊕b}
- Kill = {All expressions using x}

Initialize all but entry to universe of expressions

Available Expressions



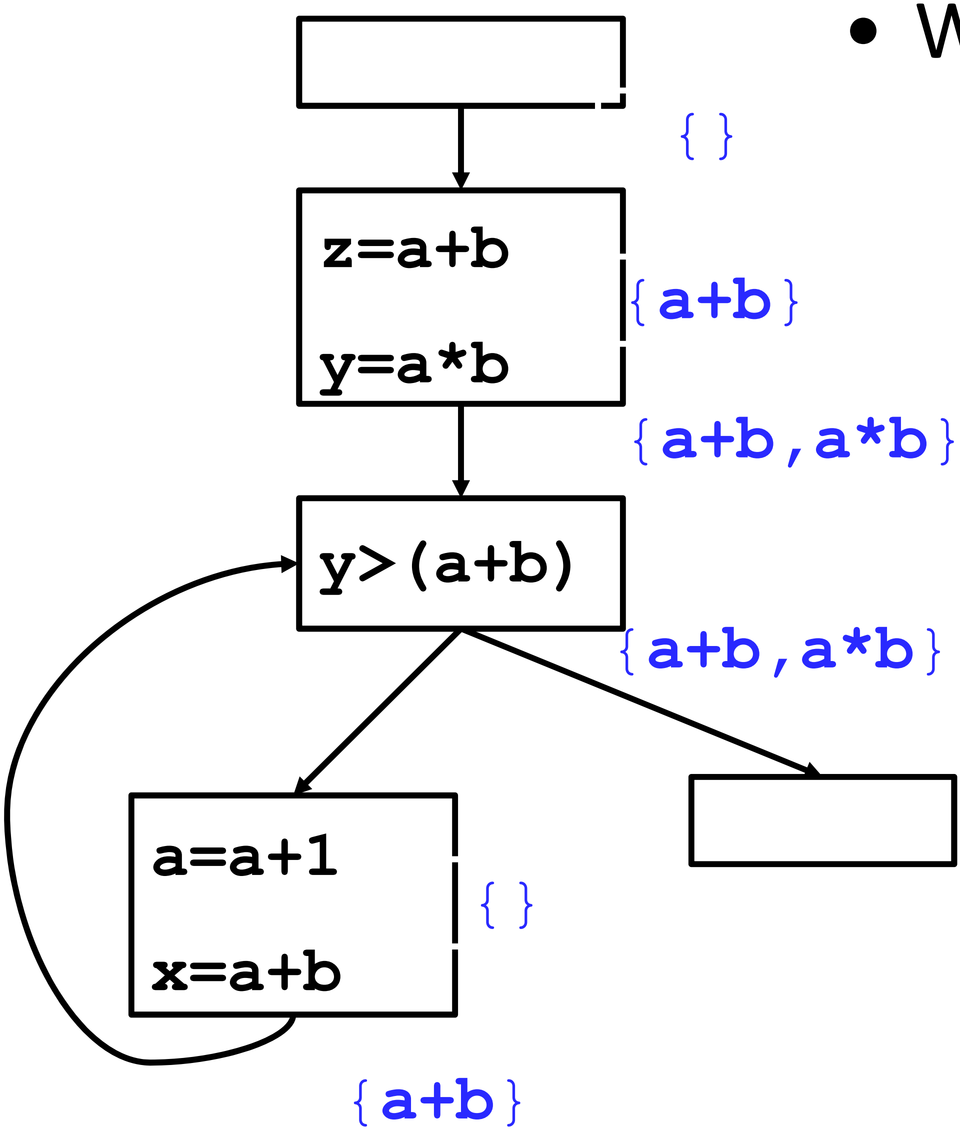
$$In(s) = \bigcap_{s' \in \text{pred}(s)} Out(s')$$

$$Out(s) = \big(In(s) \cup Gen(s)\big) - Kill(s)$$

For x= a⊕b:
- Gen = {a⊕b}
- Kill = {All expressions using x}

Initialize all but entry to universe of expressions

Available Expressions

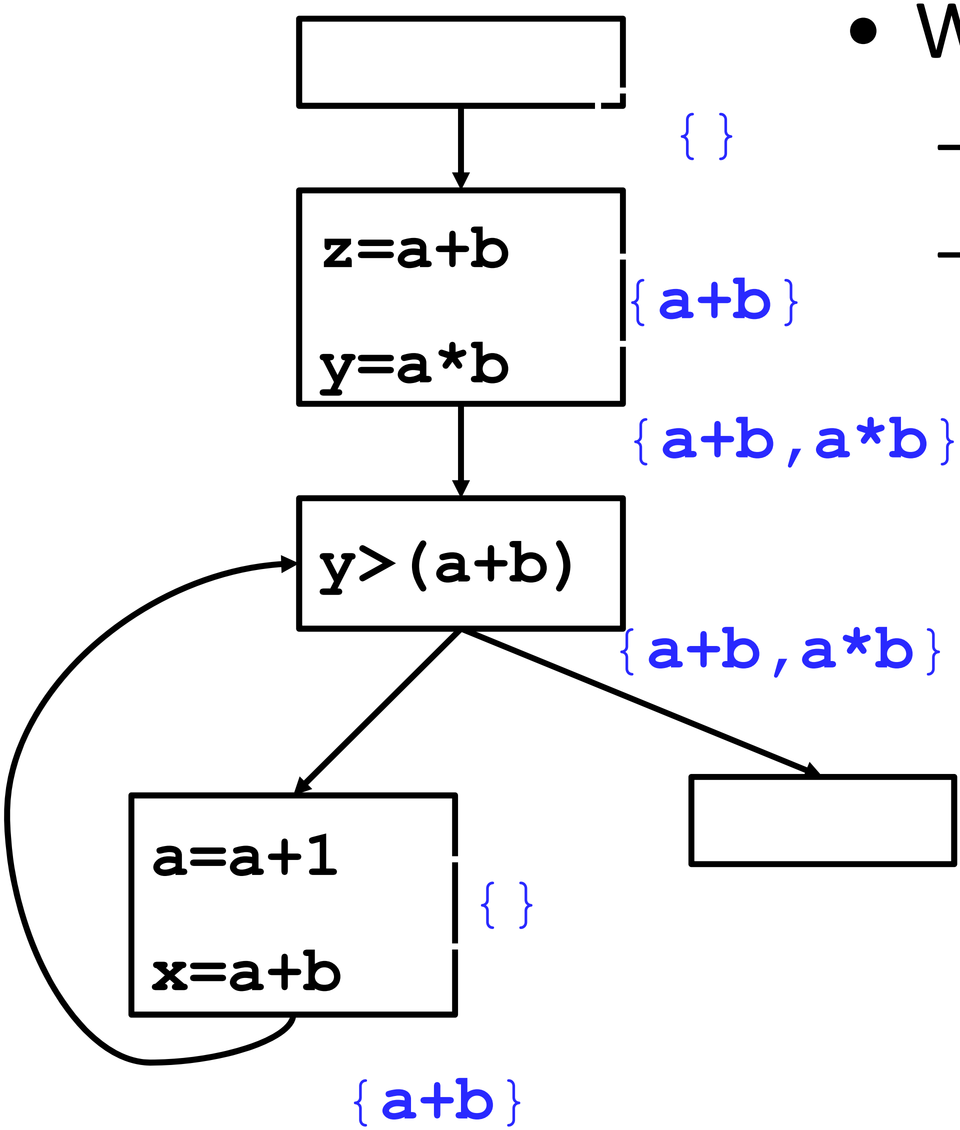


- Why Does this terminate?

$$In(s) = \bigcap_{s' \in \text{pred}(s)} Out(s')$$

$$Out(s) = \left(In(s) \cup Gen(s)\right) - Kill(s)$$

Available Expressions



- Why Does this terminate?
  - In(s) never grows
  - Out(s) never grows

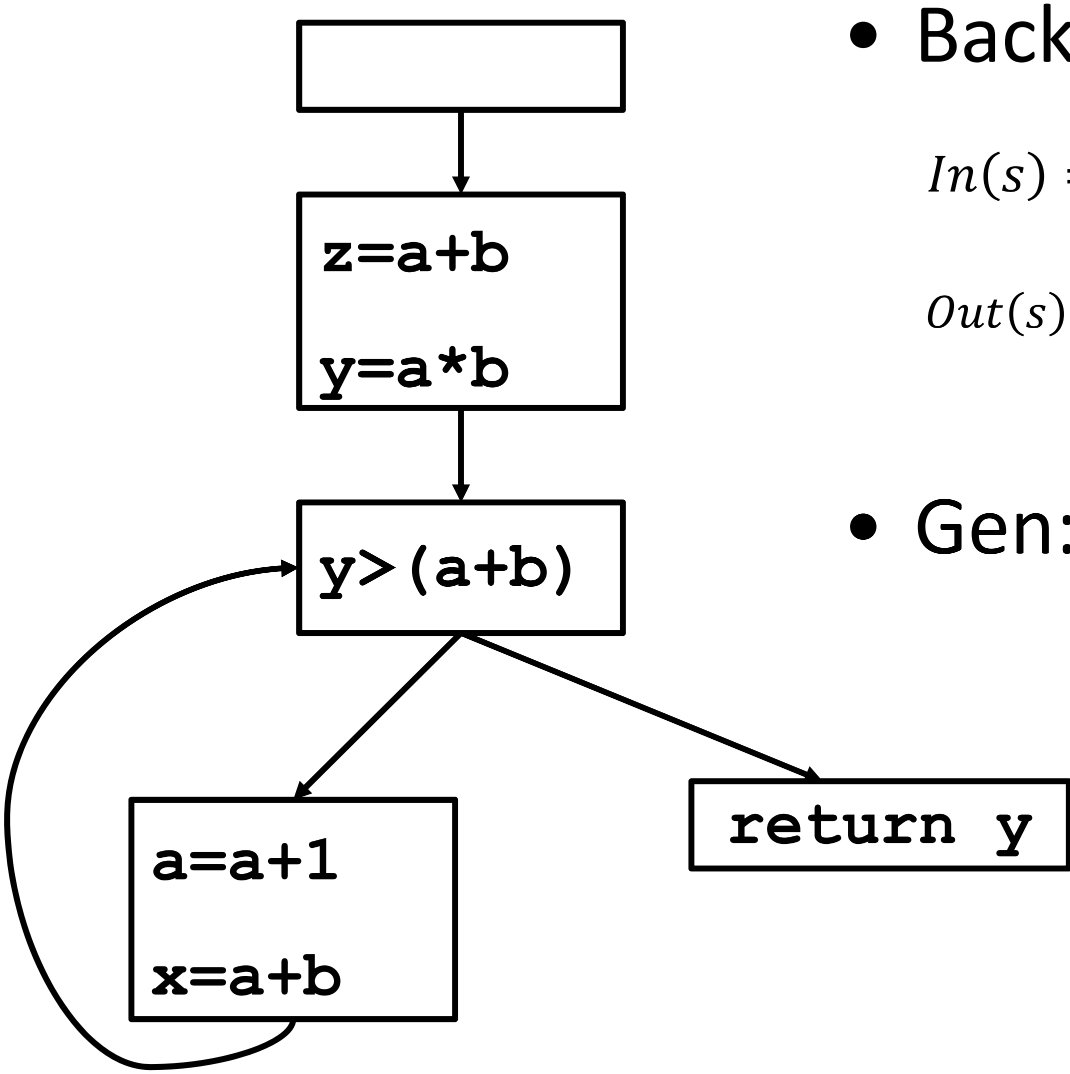
$$In(s) = \bigcap_{s' \in \text{pred}(s)} Out(s')$$

$$Out(s) = \big(In(s) \cup Gen(s)\big) - Kill(s)$$

Liveness as a dataflow problem

- This is a backwards analysis
  - A variable is live out if used by a successor
  - Gen: For a use: indicate it is live coming into s
  - Kill: Defining a variable v in s makes it dead before s (unless s uses v to define v)
  - Lattice is just live (top) and dead (bottom)
- Values are variables
- In[n] = variables live before n
  $= (out[n] - kill[n]) \cup gen[n]$
- Out[n] = variables live after n
  $= \bigcup_{s \in succ(n)} In[s]$

Liveness



- Backward, May

$$In(s) = \big(Out(s) - kill(s)\big) \cup Gen(s)$$

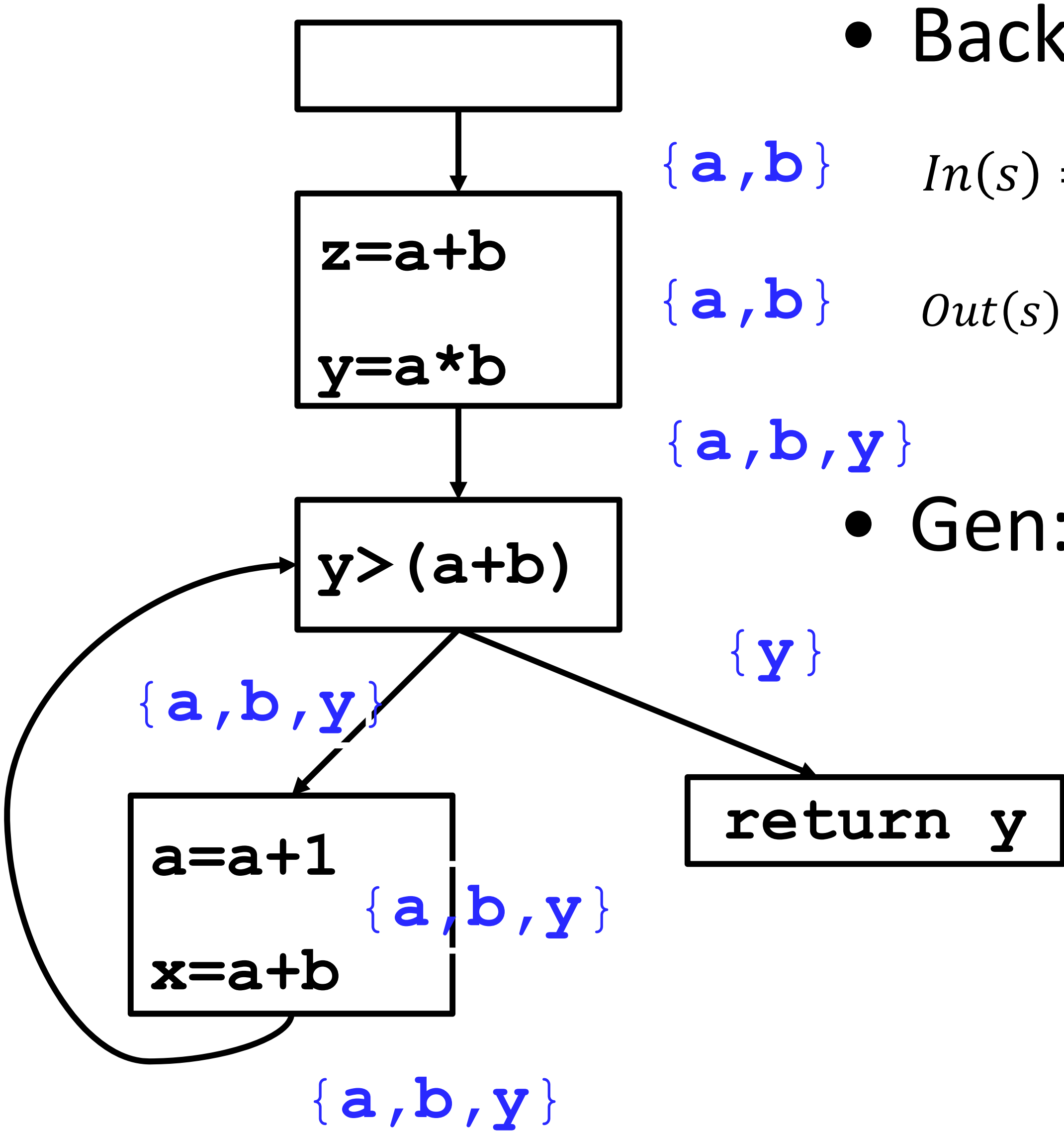
$$Out(s) = \bigcup_{s' \in \mathrm{succ}(s)} In(s')$$

- Gen:

For x= a⊕b:
Gen = {a,b}
Kill = {x}

Initialize all to empty set

Liveness



- Backward, May

$$In(s) = \left(Out(s) - kill(s)\right) \cup Gen(s)$$

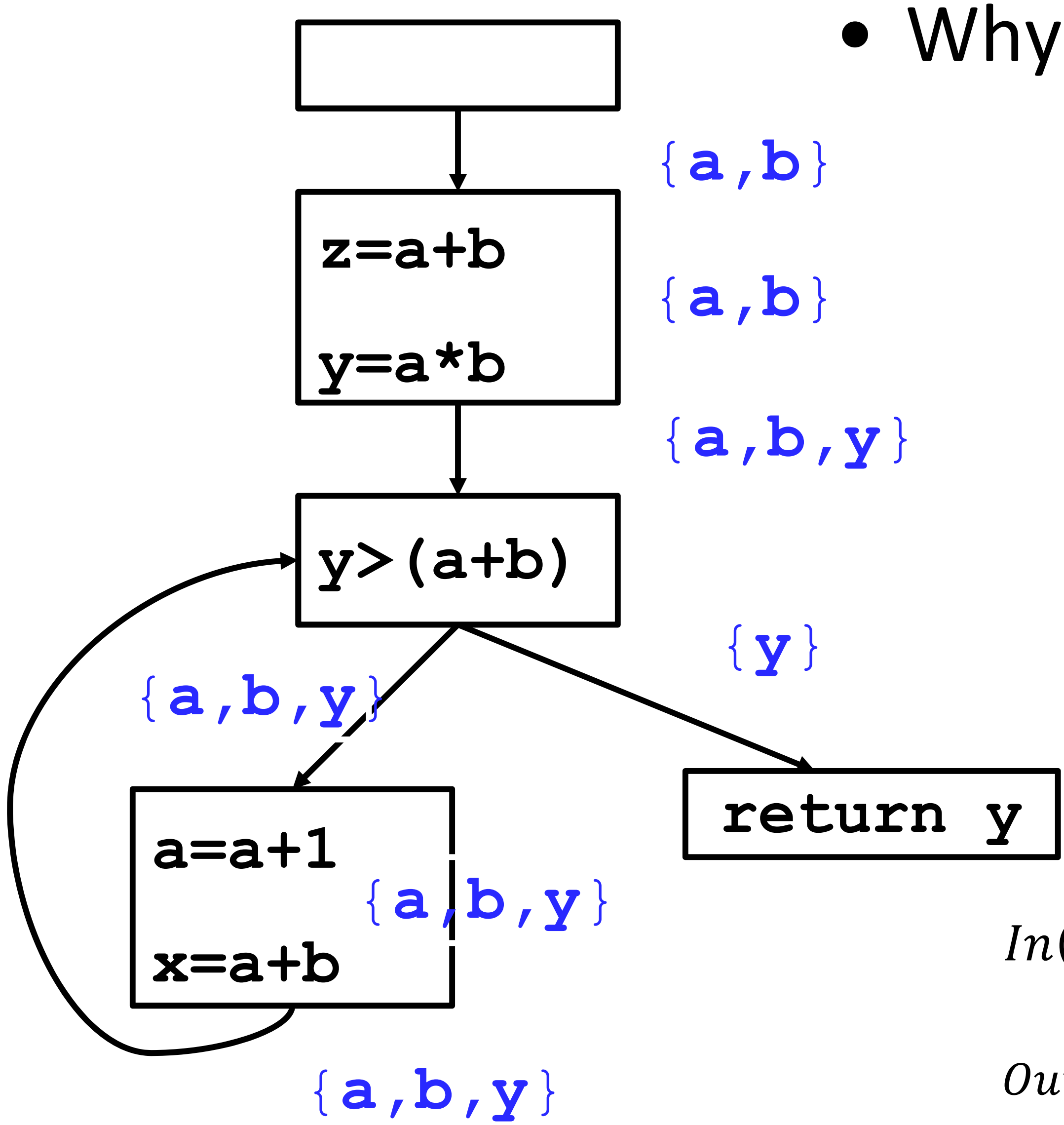
$$Out(s) = \bigcup_{s' \in \mathrm{succ}(s)} In(s')$$

- Gen:

For x= a⊕b:
  Gen = {a,b}
  Kill = {x}

Initialize all to empty set

Liveness

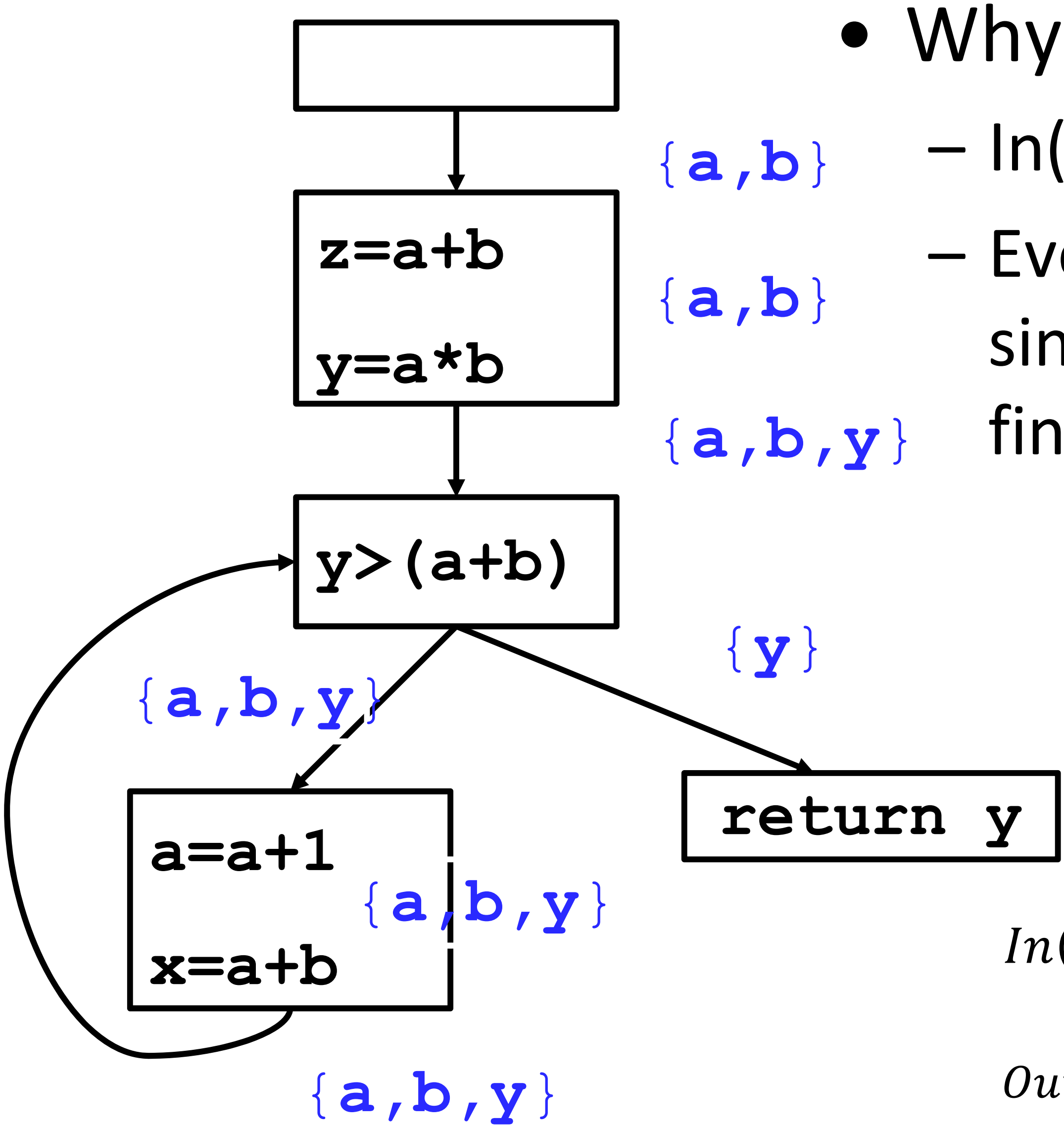


- Why does this terminate?

$$In(s) = \big(Out(s) - kill(s)\big) \cup Gen(s)$$

$$Out(s) = \bigcup_{s' \in \mathrm{succ}(s)} In(s')$$

Liveness



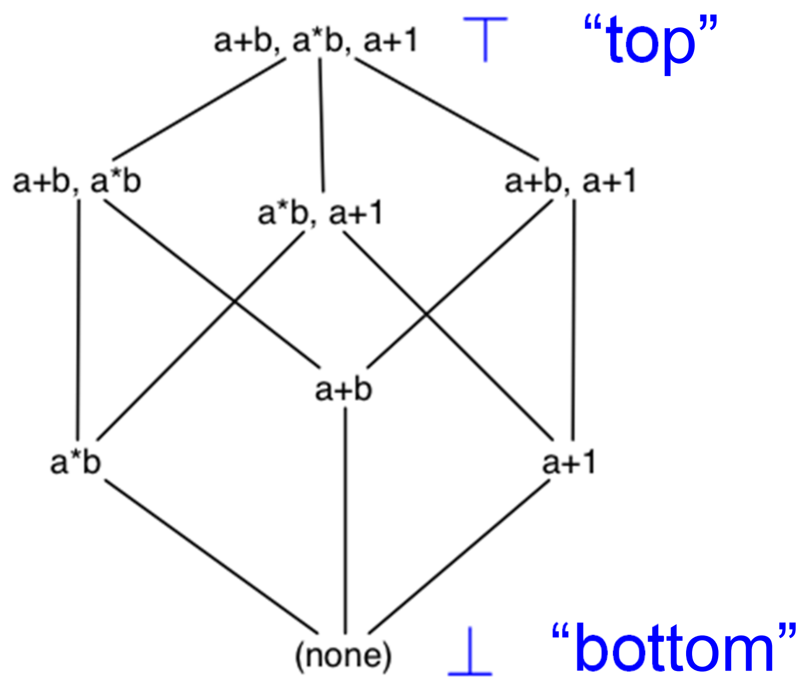
- Why does this terminate?
  - In(s) & Out(s) never shrink
  - Eventually reach fixed point since number of variables is finite.

$$In(s) = \big(Out(s) - kill(s)\big) \cup Gen(s)$$

$$Out(s) = \bigcup_{s' \in \mathrm{succ}(s)} In(s')$$

Data Flow Facts and lattices

- Typically, data flow facts form a lattice
- Example, Available expressions



Lattices

- All our dataflow analyses map program points to elements of a *lattice*.
- A *complete lattice* L = (S, ≤, ∨, ∧, ⊥, T) is formed by:
  - A set S
  - A partial order ≤ between elements of S.
  - A least element ⊥
  - A greatest element T
  - A join operator ∨
  - A meet operator ∧

Least Upper Bound & Join

- If $L = (S, \leq, \vee, \wedge, \perp, T)$ is a complete lattice, and $e_1 \in S$ and $e_2 \in S$, then
  least upper bound of $\{e_1, e_2\} \equiv e_{lub} = (e_2 \vee e_1) \in S$

Least Upper Bound & Join

- If $L = (S, \leq, \vee, \wedge, \bot, T)$ is a complete lattice, and $e_1 \in S$ and $e_2 \in S$, then
  least upper bound of $\{e_1, e_2\} \equiv e_{lub} = (e_2 \vee e_1) \in S$
- $\vee$ is the "join" operator
- e_{lub}, the least upper bound, has the properties:
  - $e_1 \leq e_{lub}$ and $e_2 \leq e_{lub}$
  - For all $e' \in S$, if $e_1 \leq e'$ and $e_2 \leq e'$, then $e_{lub} \leq e'$

Least Upper Bound & Join

- If $L = (S, \leq, \vee, \wedge, \bot, T)$ is a complete lattice, and $e_1 \in S$ and $e_2 \in S$, then
  least upper bound of $\{e_1, e_2\} \equiv e_{lub} = (e_2 \vee e_1) \in S$
- $\vee$ is the "join" operator
- e_{lub}, the least upper bound, has the properties:
  - $e_1 \leq e_{lub}$ and $e_2 \leq e_{lub}$
  - For all $e' \in S$, if $e_1 \leq e'$ and $e_2 \leq e'$, then $e_{lub} \leq e'$
- least upper bound of $S' \subseteq S$, is pairwise lub of all elements of S'
- For L to be a lattice, for all $S' \subseteq S$, $lub(S') \in S$

Greatest Lower Bound & Meet

- If $L = (S, \leq, \vee, \wedge, \perp, T)$ is a complete lattice, and $e_1 \in S$ and $e_2 \in S$, then
  greatest lower bound of $\{e_1, e_2\} \equiv e_{glb} = (e_2 \wedge e_1) \in S$
- $\wedge$ is the "meet" operator
- e_{glb}, the greatest lower bound, has the properties:
  - $e_{glb} \leq e_1$ and $e_{glb} \leq e_2$
  - For all $e' \in S$, if $e_1 \leq e'$ and $e_2 \leq e'$, then $e' \leq e_{glb}$

Greatest Lower Bound & Meet

- If $L = (S, \leq, \vee, \wedge, \perp, T)$ is a complete lattice, and $e_1 \in S$ and $e_2 \in S$, then
  greatest lower bound of $\{e_1, e_2\} \equiv e_{glb} = (e_2 \wedge e_1) \in S$
- $\wedge$ is the "meet" operator
- e_{glb}, the greatest lower bound, has the properties:
  - $e_{glb} \leq e_1$ and $e_{glb} \leq e_2$
  - For all $e' \in S$, if $e_1 \leq e'$ and $e_2 \leq e'$, then $e' \leq e_{glb}$
- greatest lower bound of $S' \subseteq S$, is pairwise glb of all elements of S'
- For L to be a lattice, for all $S' \subseteq S$, $glb(S') \in S$

Properties of join (and meet)

- Join is idempotent: $x \vee x = x$
- Join is commutative: $y \vee x = x \vee y$
- Join is associative: $x \vee (y \vee z) = (x \vee y) \vee z$
- Join has a multiplicative one:

  for all x in S, $(\bot \vee x) = x$
- Join has a multiplicative zero:

  for all x in S, $(T \vee x) = T$

Properties of join (and meet)

- Join is idempotent: $x \vee x = x$
- Join is commutative: $y \vee x = x \vee y$
- Join is associative: $x \vee (y \vee z) = (x \vee y) \vee z$
- Join has a multiplicative one:

  for all $x \in S$, $(\bot \vee x) = x$
- Join has a multiplicative zero:

  for all $x \in S$, $(T \vee x) = T$

Properties of join (and meet)

- Join is idempotent: $x \vee x = x$
- Join is commutative: $y \vee x = x \vee y$
- Join is associative: $x \vee (y \vee z) = (x \vee y) \vee z$
- Join has a multiplicative one:
  for all $x \in S$, $(\bot \vee x) = x$
- Join has a multiplicative zero:
  for all $x \in S$, $(T \vee x) = T$

- Similarly for meet, but:
  - multiplicative one is T, i.e., for all $x \in S$, $(T \wedge x) = T$
  - multiplicative zero is $\bot$, i.e., for all $x \in S$, $(\bot \wedge x) = T$

Semilattices

- Notice the dataflow analysis we looked at have either the join or meet operator, e.g.,
  - available expressions uses meet: $\wedge$ is intersection
  - liveness uses join: $\vee$ is union
- If only one of meet or join are defined, we call it a semilattice.

Partial Order

- A partial order is a pair (S, ≤) such that:
  - ≤ $\subseteq$ S×S
  - ≤ is reflexive, i.e.,

    x ≤ x
  - ≤ is anti-symmetric, i.e.,

    x ≤ y and y ≤ x implies x=y
  - ≤ is transitive, i.e.,

    x ≤ y and x ≤ z implies x ≤z

Partial Order, $\vee$, $\wedge$, and Semi-Lattice

- Join, least upper bound, on a semi-lattice defines a partial order:

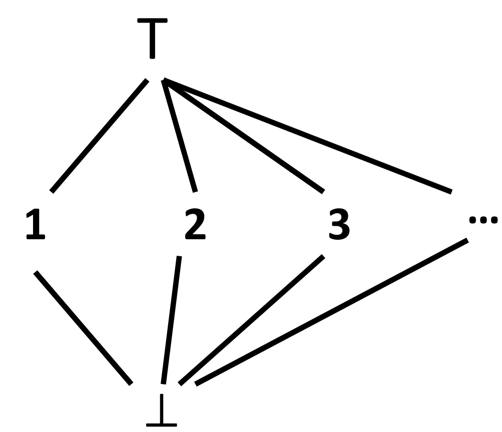
  $$x \leq y \text{ iff } x \vee y = y$$

- Meet, greatest lower bound, on a semi-lattice defines a partial order:

  $$x \leq y \text{ iff } x \wedge y = x$$

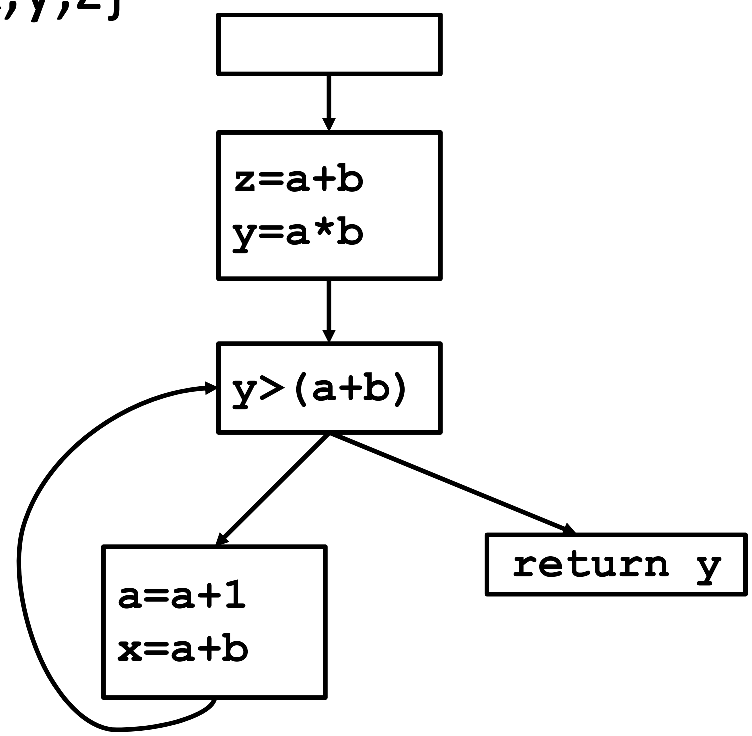
Useful Lattices

- $(2^S, \subseteq)$ forms a lattice for any set S.
  - 2^S is the power set of S (set of all subsets)
- If $(S, \leq)$ is a lattice, so is $(S, \geq)$
  - i.e., lattices can be flipped
- A lattice for constant propagation



Semilattice of Liveness

- L=({a,b,x,y,z},⊆,∪, {},{a,b,x,y,z})
  - Only define Join, ∪
  - Least Element, ⊥, {}
  - Greatest Element, T, {a,b,x,y,z}
  - x ≤ y means x ⊆ y



- more generally,

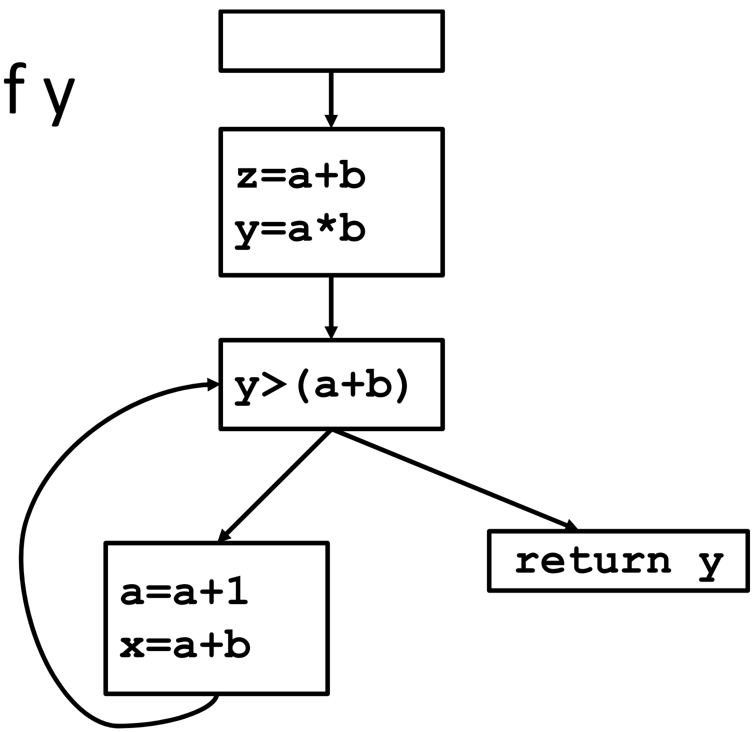
  L=(2^S, ⊆,∪, {},S)

$L=(2^S, \subseteq, \cup, \{\}, S)$

- Join operator must have the property:
  - $x \leq y$ iff $x \vee y = y$
  - Or, in our case, Is it true that: $x \subseteq y$ iff $x \cup y = y$?
- Is $\{\}$ $\perp$, or in our case: is $\{\} \subseteq x$, for all $x \in S$?
- is S T, or in our case is $x \subseteq T$, for all $x \in S$?

Semilattice of Available Expressions

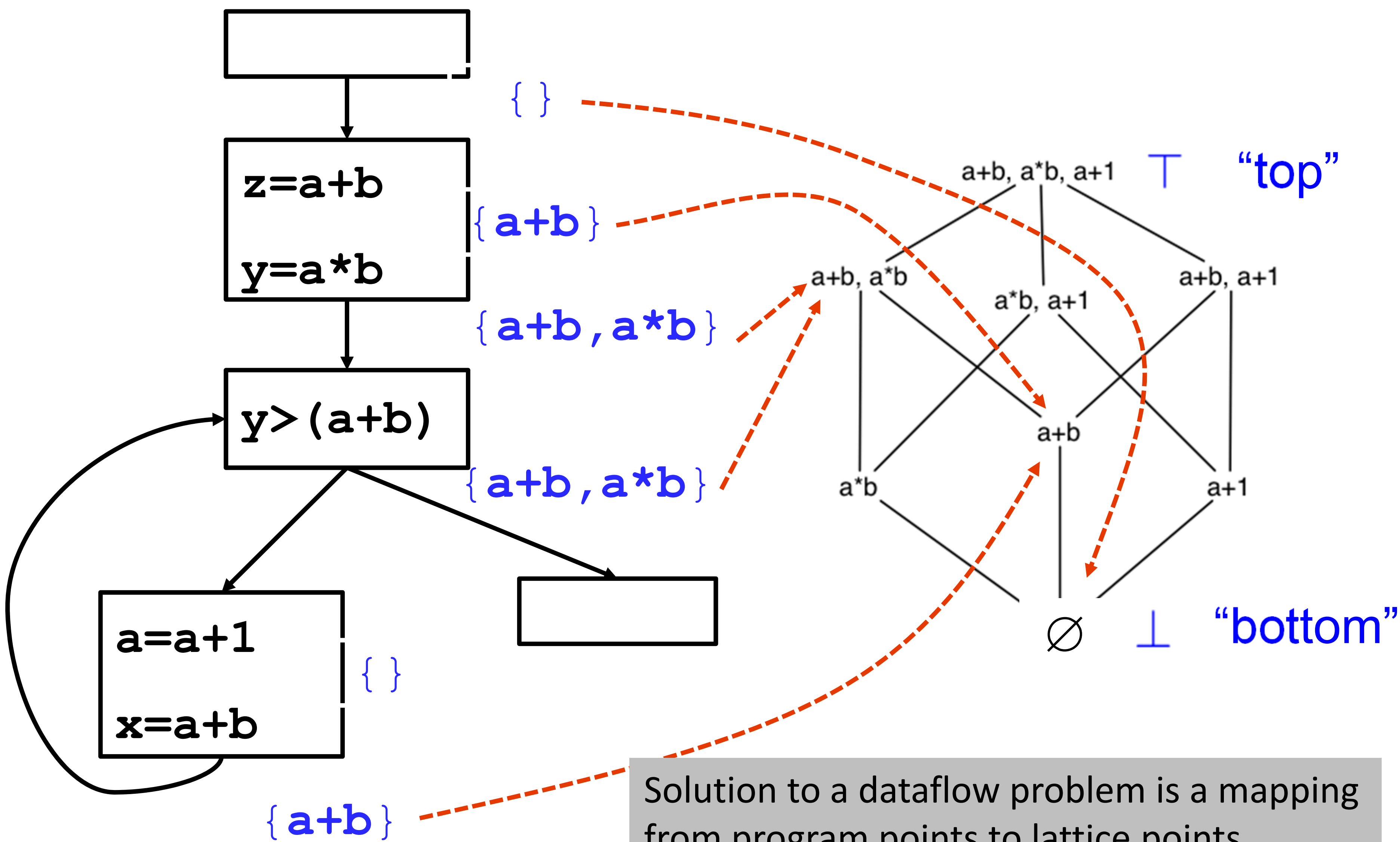
- $L=(\{a+b,a*b,a+1\},\supseteq,\cap, \{a+b,a*b,a+1\},\{\})$
  - Only define Meet, $\cap$
  - Least Element, $\perp$, $\{a+b,a*b,a+1\}$
  - Greatest Element, T, $\{\}$
  - $x \leq y$ means x is superset of y



- In general:

  $L=(2^S, \supseteq,\cap, S,\{\})$

Available Expressions



Solution to a dataflow problem is a mapping from program points to lattice points

Monotonicity & Termination

- A function f on a partial order is monotonic if

$$x \leq y \text{ implies } f(x) \leq f(y)$$

- We call f a transfer function

Monotonicity for Available Expressions

- A function f on a partial order is monotonic if

  x ≤ y implies f(x) ≤ f(y)

For x= a⊕b:
  Gen = {a⊕b}
  Kill = {All expressions using x}

$$In(s) = \bigcap_{s' \in \mathrm{pred}(s)} Out(s)$$

$$Out(s) = Gen(s) \cup \left(In(s) - Kill(s)\right)$$

$$Out(s) = f_s\left(\bigcap_{s' \in \mathrm{pred}(s)} Out(s')\right)$$

Termination

- Algorithm terminates because:
  - The lattice has finite height
  - The operations to compute In and Out are monotonic
  - On every iteration either:
    - W gets smaller, or
    - out(s) decreases for some s, i.e., we move down lattice

```
Initialize: in[s] = out[s] = Universe
Initialize: in[entry] = ∅
Work queue, W = all Blocks
while (|W| != 0) {
    remove s from W
    temp = out[s]
    compute In[s]
    compute Out[s]
    if (temp != out[s]) W = W ∪ succ(s)
}
```

Lattices (P, ≤)

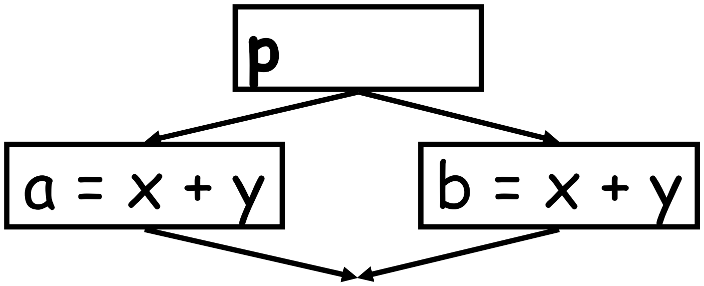
- Available expressions
  - P = sets of expressions
  - $S1 \wedge S2 = S1 \cap S2$
  - Top = set of all expressions
- Reaching Definitions
  - P = sets of definitions (assignment statements)
  - $S1 \wedge S2 = S1 \cup S2$
  - Top = empty set

Fixpoints

- We always start with Top
  - Every expression is available, no definitions reach this point
  - Most optimistic assumption
  - Strongest possible hypothesis (i.e., true of fewest number of states)
- Revise as we encounter contradictions
  - Always move down in the lattice (with meet)
- Result: A greatest fixpoint

Very Busy Expressions

- A Backward, Must data flow analysis
- An expression e is *very busy at point p* if On every path from p, e is evaluated before the value of e is changed
- Optimization
  - Can hoist very busy expression computation



Lattices (P, ≤), cont'd

- Live variables
  - P = sets of variables
  - S1 ∧ S2 = S1 ∪ S2
  - Top = empty set
- Very busy expressions
  - P = sets of expressions
  - S1 ∧ S2 = S1 ∩ S2
  - Top = set of all expressions

Lattices (P, ≤), cont'd

- Live variables
  - P = sets of variables
  - S1 ∧ S2 = S1 ∪ S2
  - Top = empty set
- Very busy expressions
  - P = sets of expressions
  - S1 ∧ S2 = S1 ∩ S2
  - Top = set of all expressions

Could have defined this as a semilattice using join, but dataflow tradition starts with top and uses meet to compute a greatest fixed point. (as compared to tradition for denotational semantics, uses meet and computes least fixed point)

Forward vs. Backward

```
Out(s) = Top  for all s
W := { all statements }
repeat
    Take s from W
    temp := f_s(∧ s′ ∈ pred(s) Out(s′))
    if (temp != Out(s)) {
     Out(s) := temp
     W := W ∪ succ(s)
    }
until W = ∅
```

```
In(s) = Top  for all s
W := { all statements }
repeat
    Take s from W
    temp := f_s(∧ s′ ∈ succ(s) In(s′))
    if (temp != In(s)) {
     In(s) := temp
     W := W ∪ pred(s)
    }
until W = ∅
```

Termination Revisited

- How many times can we apply this step:

  $temp := f_s(\sqcap_{s' \in pred(s)} Out(s'))$

  if (temp != Out(s)) { … }

Claim: Out(s) only shrinks

- Proof: Out(s) starts out as top
  - So temp must be ≤ than Top after first step

- Assume Out(s') shrinks for all predecessors s' of s

- Then $\sqcap_{s' \in pred(s)}$ Out(s') shrinks

- Since f_s monotonic, $f_s(\sqcap_{s' \in pred(s)} Out(s'))$ shrinks

Termination Revisited (cont'd)

- A *descending chain* in a lattice is a sequence
  - $x0 \sqsupseteq x1 \sqsupseteq x2 \sqsupseteq \ldots$
- The *height* of a lattice is the length of the longest descending chain in the lattice
- Then, dataflow must terminate in O(nk) time
  - n = # of statements in program
  - k = height of lattice
  - assumes meet operation takes O(1) time

Order Matters

- Acyclic
- Cycles, nesting depth

Order Matters

- Assume forward data flow problem
  - Let $G = (V, E)$ be the CFG
  - Let k be the height of the lattice

- If G acyclic, visit in topological order
  - Visit head before tail of edge
- Running time $O(|E|)$
  - No matter what size the lattice

Order Matters — Cycles

- If G has cycles, visit in reverse postorder
  - Order from depth-first search
- Let Q = max # back edges on cycle-free path
  - Nesting depth
  - Back edge is from node to ancestor on DFS tree
- Then if $\forall x, f(x) \leq x$ (sufficient, but not necessary)
  - Running time is $O((Q + 1) |E|)$
    - Note direction of depends on top vs. bottom

Distributive Data Flow Problems

- By monotonicity, we also have

$$f(x \sqcap y) \leq f(x) \sqcap f(y)$$

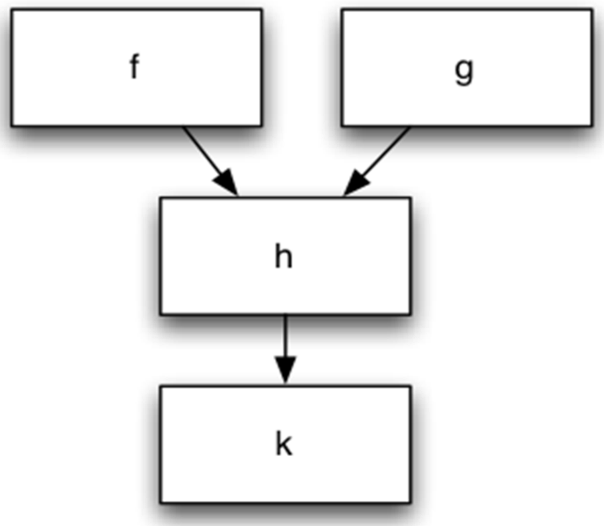
- A function f is distributive if

$$f(x \sqcap y) = f(x) \sqcap f(y)$$

Benefit of Distributivity

- Joins lose no information

$$k(h(f(\top) \sqcap g(\top))) =$$
$$k(h(f(\top)) \sqcap h(g(\top))) =$$
$$k(h(f(\top))) \sqcap k(h(g(\top)))$$



Accuracy of Data Flow Analysis

- Ideally, we would like to compute the meet over all paths (MOP) solution:
  - Let f_s be the transfer function for statement s
  - If p is a path $\{s_1, ..., s_n\}$, let $f_p = f_n;...;f_1$
  - Let path(s) be the set of paths from the entry to s

$$\mathrm{MOP}(s) = \sqcap_{p \in \mathrm{path}(s)} f_p(\top)$$

- If a data flow problem is distributive, then solving the data flow equations in the standard way yields the MOP solution

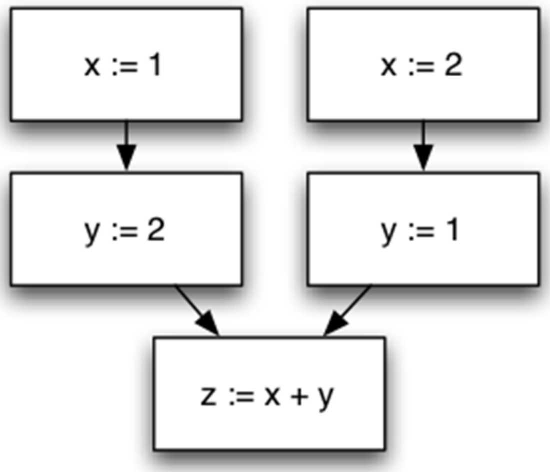
What Problems are Distributive?

- Analyses of *how* the program computes
  - Live variables
  - Available expressions
  - Reaching definitions
  - Very busy expressions

- All Gen/Kill problems are distributive

A Non-Distributive Example

- Constant propagation



- In general, analysis of *what* the program computes is not distributive

Constant Propagation

- L = (S, ≤, ∧, ⊥, T) for constant propagation
  - Set S
  - Partial order ≤ between elements of S.
  - Meet operator ∧
  - Least element ⊥
  - Greatest element T

Flow-Sensitivity

- Data flow analysis is *flow-sensitive*
  - The order of statements is taken into account
  - i.e., we keep track of facts per program point

- Alternative: *Flow-insensitive* analysis
  - Analysis the same regardless of statement order
  - Standard example: types

Terminology Review

- Must vs. May
  - (Not always followed in literature)
- Forwards vs. Backwards
- Flow-sensitive vs. Flow-insensitive
- Distributive vs. Non-distributive

Another Approach: Elimination

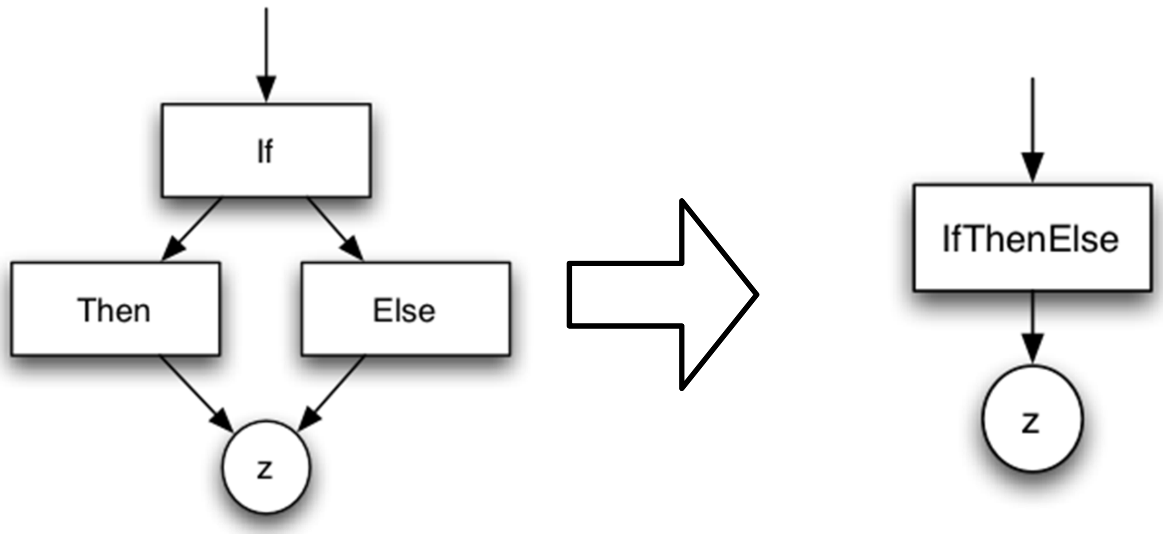
- Recall in practice, one transfer function per basic block
- Why not generalize this idea beyond a basic block?
  - "Collapse" larger constructs into smaller ones, combining data flow equations
  - Eventually program collapsed into a single node!
  - "Expand out" back to original constructs, rebuilding information

Lattices of Functions

- Let $(P, \leq)$ be a lattice
- Let M be the set of monotonic functions on P
- Define $f \leq_f g$ if for all x, $f(x) \leq g(x)$
- Define the function $f \sqcap g$ as
  - $(f \sqcap g)(x) = f(x) \sqcap g(x)$

- Claim: $(M, \leq_f)$ forms a lattice

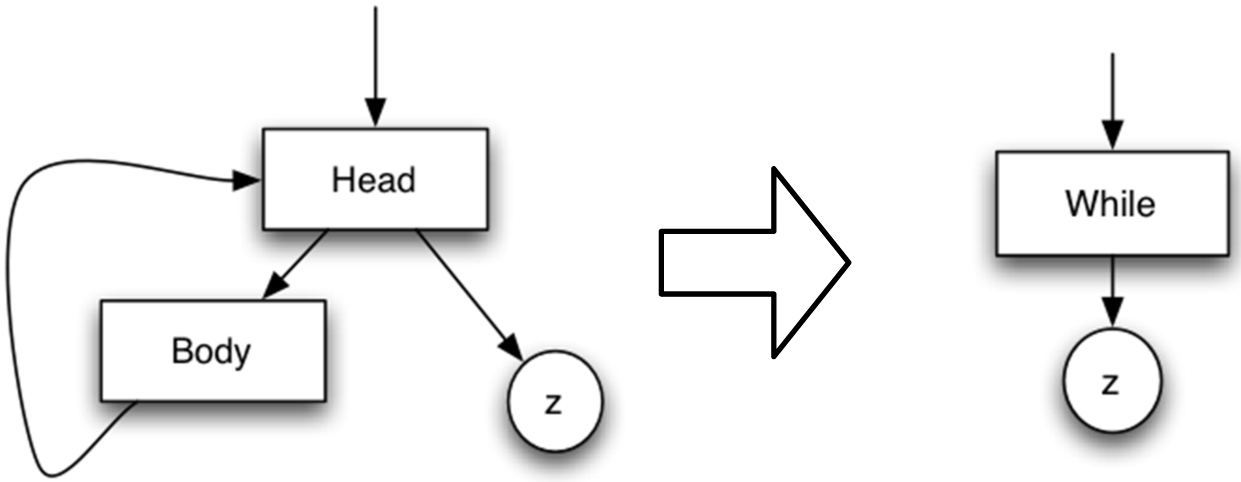
Elimination Methods: Conditionals



$$f_{\text{ite}} = (f_{\text{then}} \circ f_{\text{if}}) \sqcap (f_{\text{else}} \circ f_{\text{if}})$$

$$\begin{aligned}
\text{Out(if)} &= f_{\text{if}}(\text{In(ite)})) \\
\text{Out(then)} &= (f_{\text{then}} \circ f_{\text{if}})(\text{In(ite)})) \\
\text{Out(else)} &= (f_{\text{else}} \circ f_{\text{if}})(\text{In(ite)}))
\end{aligned}$$

Elimination Methods: Loops



$$
\begin{aligned}
f_{\text{while}} \quad = \quad & f_{\text{head}} \sqcap \\
& f_{\text{head}} \circ f_{\text{body}} \circ f_{\text{head}} \sqcap \\
& f_{\text{head}} \circ f_{\text{body}} \circ f_{\text{head}} \circ f_{\text{body}} \circ f_{\text{head}} \sqcap \cdots
\end{aligned}
$$

Elimination Methods: Loops (cont)

- Let $f^i = f \circ f \circ \ldots \circ f$ (i times)
  - $f^0 = id$
- Let

$$g(j) = \sqcap_{i \in [0..j]} (f_{\text{head}} \circ f_{\text{body}})^i \circ f_{\text{head}}$$

- Need to compute limit as j goes to infinity
  - Does such a thing exist?
- Observe: $g(j+1) \leq g(j)$

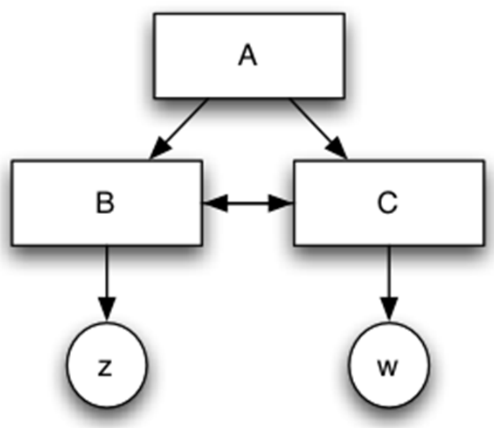
Height of Function Lattice

- Assume underlying lattice $(P, \leq)$ has finite height
  - What is height of lattice of monotonic functions?
  - Claim: At most $|P| \times \text{Height}(P)$
- Therefore, $g(j)$ converges

Non-Reducible Flow Graphs

- Elimination methods usually only applied to *reducible* flow graphs
  - Ones that can be collapsed
  - Standard constructs yield only reducible flow graphs

- Unrestricted goto can yield non-reducible graphs



Comments

- Can also do backwards elimination
  - Not quite as nice (regions are usually single *entry* but often not single *exit*)
- For bit-vector problems, elimination efficient
  - Easy to compose functions, compute meet, etc.
- Elimination originally seemed like it might be faster than iteration
  - Not really the case

Dataflow Framework

- Universe of values forms a lattices
- Meet operator used at join points in CFG
- Basic attributes (e.g., gen, kill)
- Traversal order
- Transfer function

- Will it terminate?
- Is it efficient?
- Is it accurate?

Dataflow Summary

	Union (may)	intersection (must)
Forward	Reaching definitions	Available expressions
Backward	Live variables	very busy expressions

Later in course we look at bidirectional dataflow