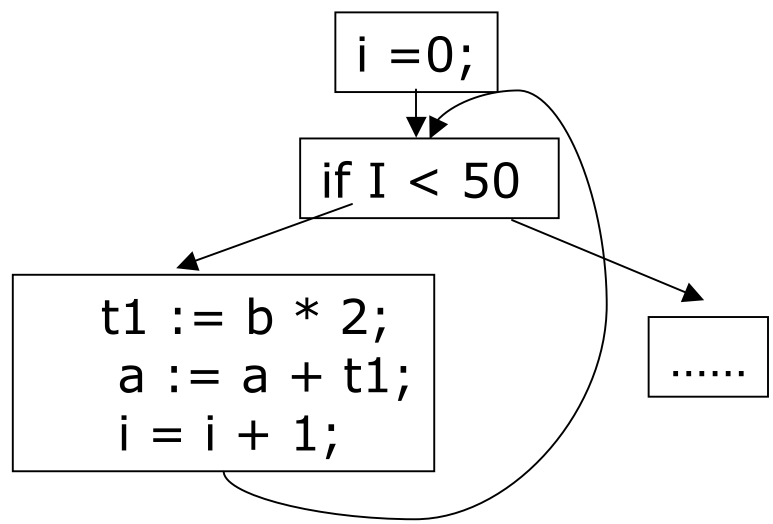
Dataflow analysis

Theory and Applications

Control-flow graph

- Graphical representation of runtime control-flow paths
  - Nodes of graph: basic blocks (straight-line computations)
  - Edges of graph: flows of control
- Useful for collecting information about computation
  - Detect loops, remove redundant computations, register allocation, instruction scheduling…
- Alternative CFG: Each node contains a single statement

```
……
    i = 0
   while (i < 50) {
     t1 = b * 2;
     a = a + t1;
     i = i + 1;
   }
….
```



Building control-flow graphs
Identifying basic blocks

- Input: a sequence of three-address statements
- Output: a list of basic blocks
- Method:
  - Determine each statement that starts a new basic block, including
    - The first statement of the input sequence
    - Any statement that is the target of a goto statement
    - Any statement that immediately follows a goto statement
  - Each basic block consists of
    - A starting statement S0 (leader of the basic block)
    - All statements following S0 up to but not including the next starting statement (or the end of input)

```
......
    i := 0
s0: if i < 50 goto s1
    goto s2
s1: t1 := b * 2
     a := a + t1
     goto s0
S2: …
```

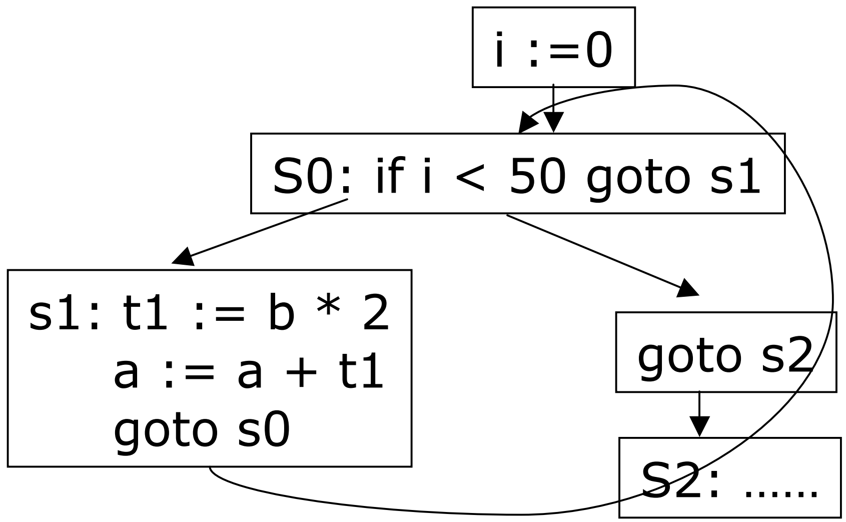
Starting statements:

i := 0
S0,
goto S2
S1,
S2

Building control-flow graphs

- Identify all the basic blocks
  - Create a flow graph node for each basic block
- For each basic block B1
  - If B1 ends with a jump to a statement that starts basic block B2, create an edge from B1 to B2
  - If B1 does not end with an unconditional jump, create an edge from B1 to the basic block that immediately follows B1 in the original evaluation order

```
......
     i := 0
s0: if i < 50 goto s1
     goto s2
s1: t1 := b * 2
      a := a + t1
      goto s0
S2: …
```

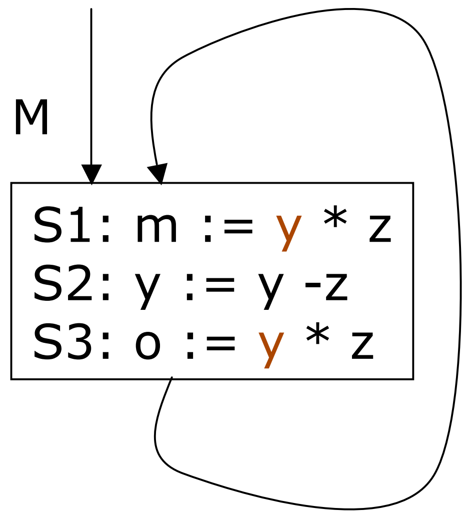


Example Dataflow
Live variable analysis

- A data-flow analysis problem
  - A variable v is live at CFG point p iff there is a path from p to a use of v along which v is not redefined
  - At any CFG point p, what variables are alive?
- Live variable analysis can be used in
  - Global register allocation
    - Dead variables no longer need to be in registers
  - Useless-store elimination
    - Dead variable don't need to be stored back to memory
  - Uninitialized variable detection
    - No variable should be alive at program entry point

Computing live variables

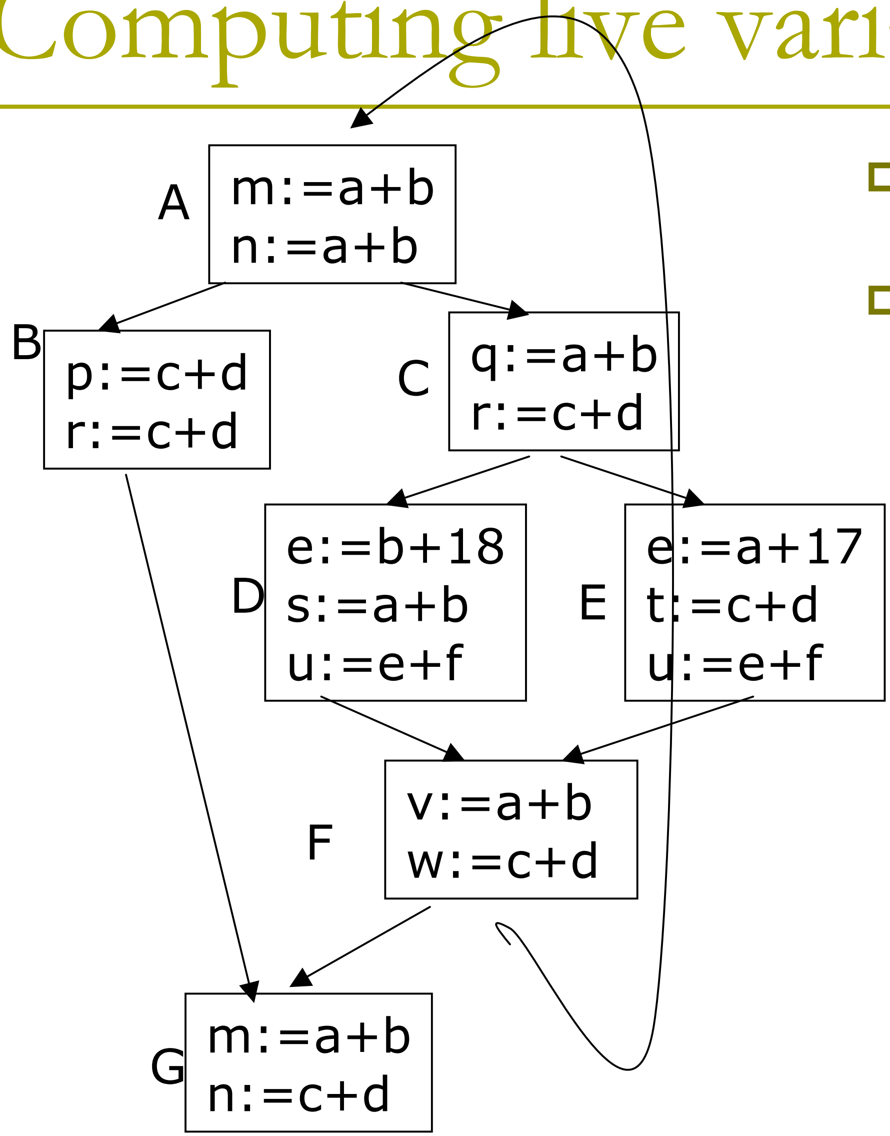
- For each basic block n, let
  - UEVar(n)=variables used before any definition in n
  - VarKill(n)=variables defined (modified) in n (killed by n)



for each basic block n:S1;S2;S3;…;Sk

```
VarKill := ∅
UEVar(n) := ∅
for i = 1 to k
   suppose Si is "x := y op z"
   if y ∉ VarKill
      UEVar(n) = UEVar(n) ∪ {y}
   if z ∉ VarKill
      UEVar(n) = UEVar(n) ∪ {z}
   VarKill = VarKill ∪ {x}
```

Computing live variables



- Domain
  - All variables inside a function
- For each basic block n, let
  - UEVar(n)
    vars used before defined
  - VarKill(n)
    vars defined (killed by n)

Goal: evaluate vars alive on entry to and exit from n

$LiveOut(n)=\bigcup_{m \in succ(n)} LiveIn(m)$

$LiveIn(m)=UEVar(m) \cup (LiveOut(m)-VarKill(m))$

==>

$LiveOut(n)= \bigcup_{m \in succ(n)} (UEVar(m) \cup (LiveOut(m)-VarKill(m))$

Algorithm: computing live variables

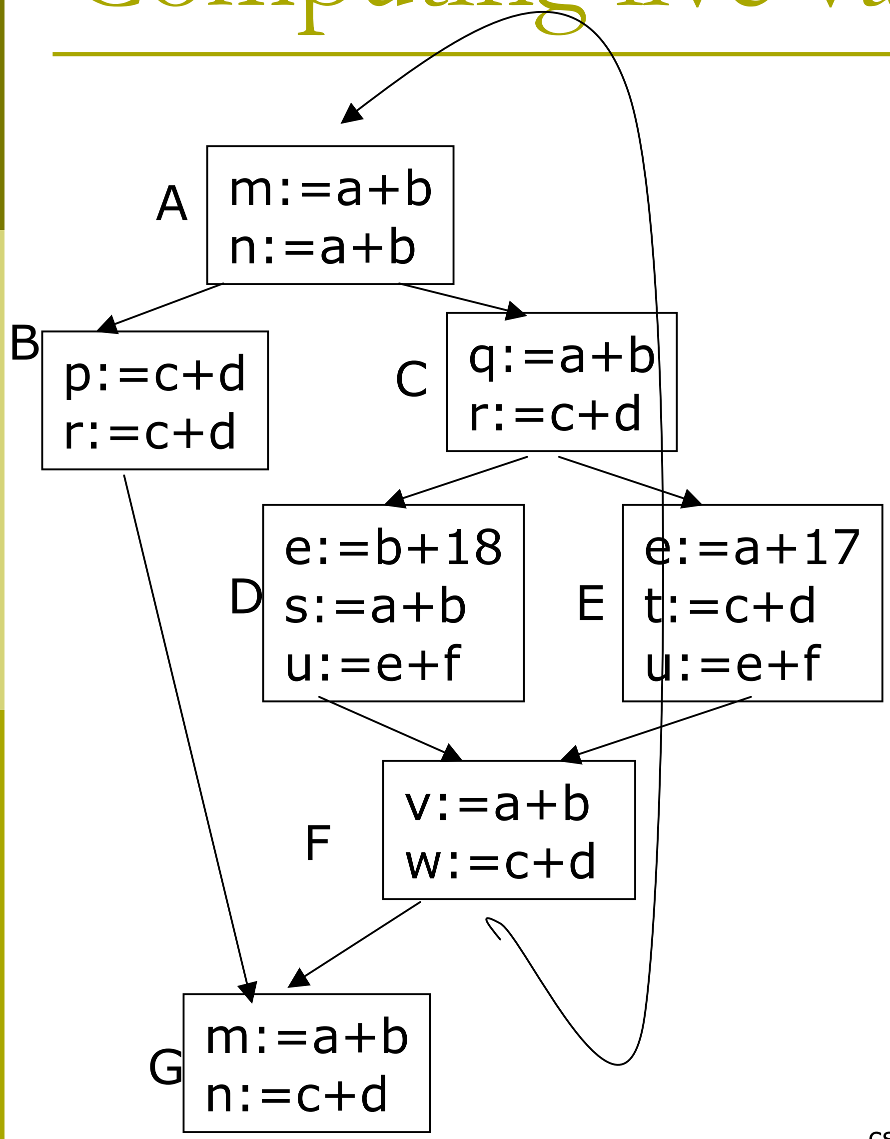
- For each basic block n, let
  - UEVar(n)=variables used before any definition in n
  - VarKill(n)=variables defined (modified) in n (killed by n)

Goal: evaluate names of variables alive on exit from n

- $LiveOut(n) = \bigcup_{m \in succ(n)} (UEVar(m) \cup (LiveOut(m) - VarKill(m))$

```
for each basic block bi
    compute UEVar(bi) and VarKill(bi)
    LiveOut(bi) := ∅
for (changed := true; changed; )
    changed = false
    for each basic block bi
        old = LiveOut(bi)
        LiveOut(bi)= ∪ (UEVar(m) ∪ (LiveOut(m) - VarKill(m))
                   m∈succ(bi)
        if (LiveOut(bi) != old) changed := true
```

Solution
Computing live variables



- Domain
  - a,b,c,d,e,f,m,n,p,q,r,s,t,u,v,w

	UE var	Varkill	LiveOut	LiveOut	LiveOut
A	a,b	m,n	∅	a,b,c,d,f	a,b,c,d,f
B	c,d	p,r	∅	a,b,c,d	a,b,c,d
C	a,b,c,d	q,r	∅	a,b,c,d,f	a,b,c,d,f
D	a,b,f	e,s,u	∅	a,b,c,d	a,b,c,d,f
E	a,c,d,f	e,t,u	∅	a,b,c,d	a,b,c,d,f
F	a,b,c,d	v,w	∅	a,b,c,d	a,b,c,d,f
G	a,b,c,d	m,n	∅	∅	∅

Another Example Available Expressions Analysis

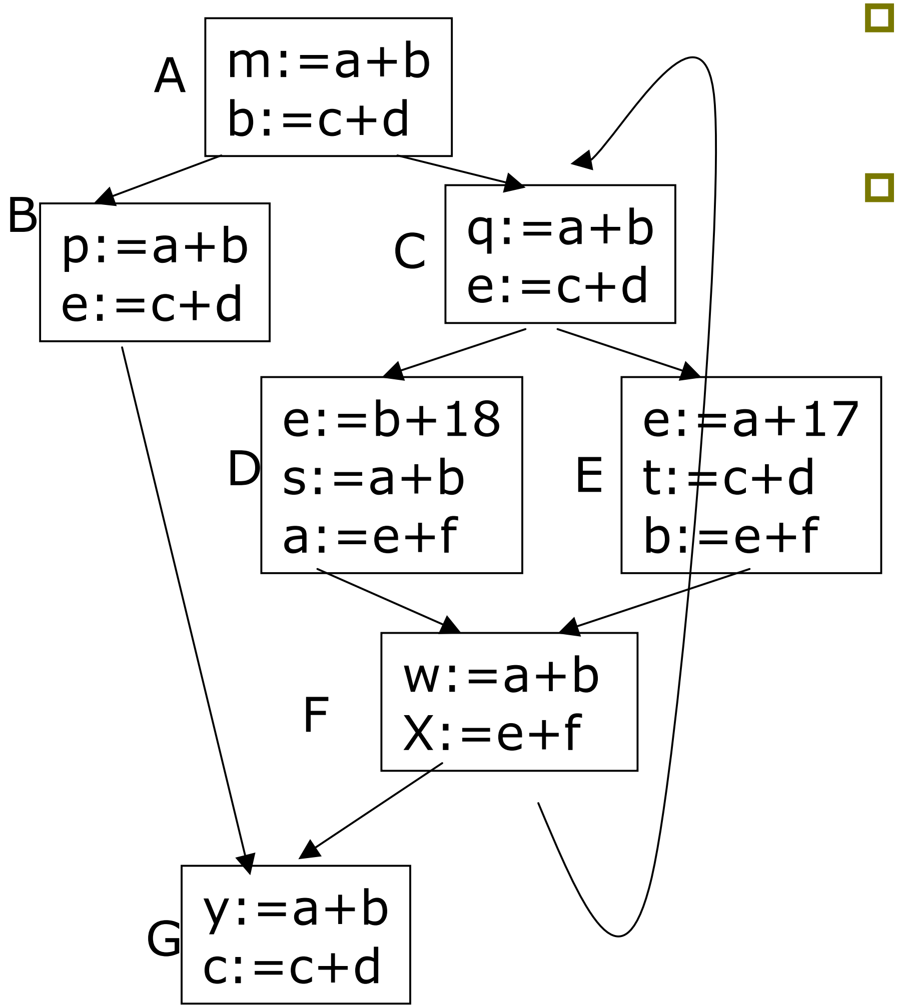
- The aim of the Available Expressions Analysis is to determine
  - For each program point, which expressions must have already been computed, and not later modified, on all paths to the program point.
  - Example

```
[x:= a+b ]1;
[y:=a*b]2;
while [y> a+b ]3 {
    [a:=a+1]4;
    [x:= a+b ]5
}
```

Optimized code:

```
[x:= a+b]1;
[y:=a*b]2;
while [y> x ]3 {
    [a:=a+1]4;
    [x:= a+b]5
}
```

Available Expression Analysis



- Domain of analysis
  - All expressions within a function
- For each basic block n, let
  - DEexp(n)
    Exps evaluated without any operand redefined
  - ExpKill(n)
    Exps whose operands are redefined (exps killed by n)

Goal: evaluate exps available on all paths entering n

AvailIn(n)=$\cap_{m \in pred(n)}$AvailOut(m)

AvailOut(m) = DEexp(m)$\cup$

(AvailIn(m)-ExpKill(m))

==>

AvailIn(n)= $\cap_{m \in pred(n)}$ (DEexp(m) $\cup$ (AvailIn(m)-ExpKill(m))

Algorithm: computing available expressions

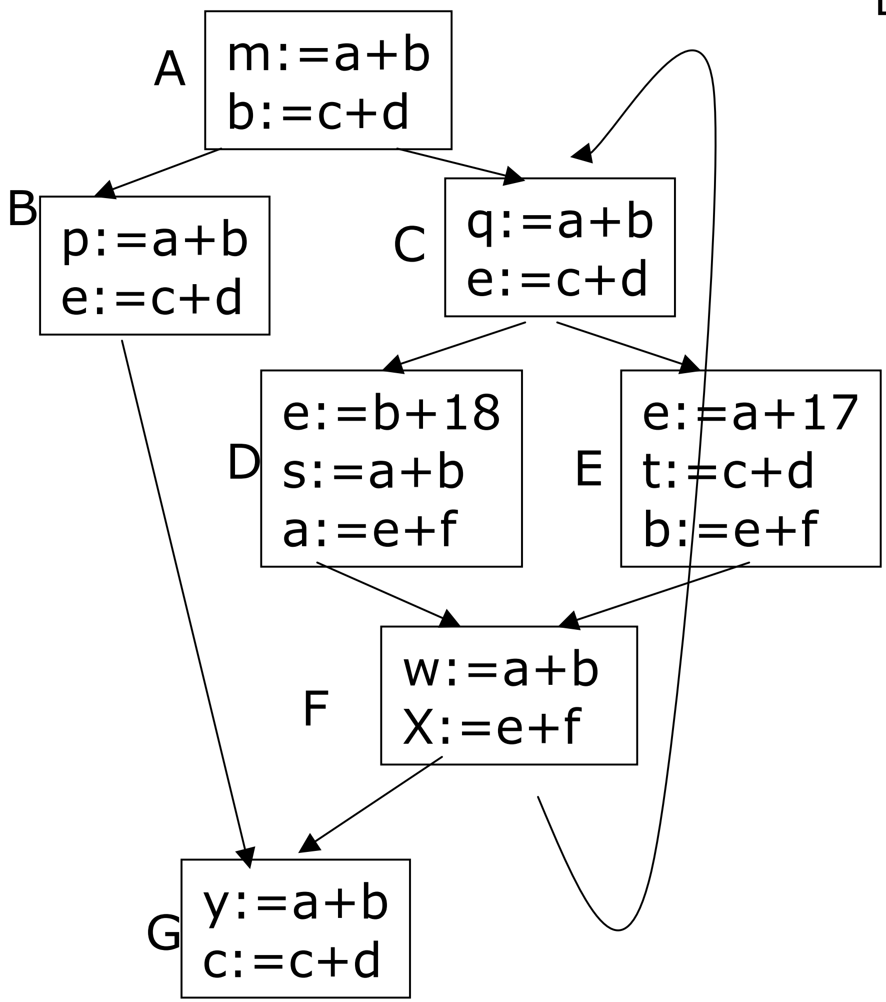
- For each basic block n, let
  - DEexp(n)=expressions evaluated without any operand redefined
  - ExpKill(n)=expressions whose operands are redefined in n

Goal: evaluate expressions available from entry to n

$$AvailIn(n) = \bigcap_{m \in pred(n)} (DEexp(m) \cup (AvailIn(m) - ExpKill(m)))$$

```
for each basic block bi
    compute DEexp(bi) and ExpKill(bi)
    AvailIn(bi) := isEntry(bi)? ∅ : Domain(Exp);
for (changed := true; changed; )
   changed = false
   for each basic block bi
      old = Avail(bi)
      AvailIn(bi)= ∩ m∈pred(bi)(DEexp(m) ∪ (AvailIn(m)-ExpKill(m))
      if (AvailIn(bi) != old) changed := true
```

Solution
Available Expression Analysis



Domain: a+b(1), c+d(2), b+18(3),e+f(4), a+17(5)

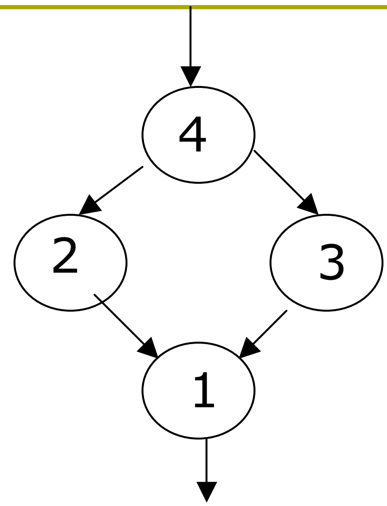
	DEexp	ExpKil	Avail	Avail
A	2	1,3	∅	∅
B	1,2	4	12345	2
C	1,2	4	12345	2
D	3,4	1,4,5	12345	1,2
E	2,4,5	1,3,4	12345	1,2
F	1,4	∅	12345	2,4
G	1,2	2	12345	1,2

Iterative dataflow algorithm

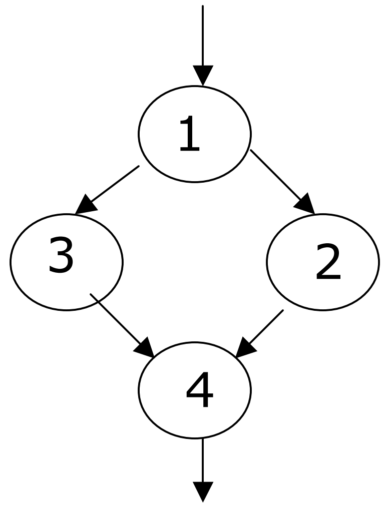
```
for each basic block bi
    compute Gen(bi) and Kill(bi)
    Result(bi) := ∅ or Domain
for (changed := true; changed; )
    changed = false
    for each basic block bi
        old = Result(bi)
        Result(bi)=
        ∩ or ∪
    [m∈pred(bi) or succ(bi)]
  (Gen(m) ∪ (Result(m)-Kill(m))
        if (Result(bi) != old)
            changed := true
```

- Iterative evaluation of result sets until a fixed point is reached
  - Does the algorithm always terminate?
    - If the result sets are bounded and grow monotonically, then yes; Otherwise, no.
    - Fixed-point solution is independent of evaluation order
  - What answer does the algorithm compute?
    - Unique fixed-point solution
    - The meet-over-all-paths solution
  - How long does it take the algorithm to terminate?
    - Depends on traversing order of basic blocks

Traversing order of basic blocks



postorder



Reverse postorder

- Facilitate fast convergence to the fixed point
- Postorder traversal
  - Visits as many of a nodes successors as possible before visiting the node
  - Used in backward data-flow analysis
- Reverse postorder traversal
  - Visits as many of a node's predecessors as possible before visiting the node
  - Used in forward data-flow analysis

The Overall Pattern

- Each data-flow analysis takes the form

  Input(n) := ∅ if n is program entry/exit

  := Λ m∈Flow(n) Result(m)   otherwise

  Result(n) = fn (Input(n))

  - where Λ is ∩ or ∪ (may vs. must analysis)
    - May analysis: detect properties satisfied by at least one path (∪)
    - Must analysis: detect properties satisfied by all paths(∩)
  - Flow(n) is either pred(n) or succ(n) (forward vs. backward flow)
    - Forward flow: data flow forward along control-flow edges.
      - Input(n) is data entering n, Result is data exiting n
      - Input(n) is ∅ if n is program entry
    - Backward flow: data flow backward along control-flow edges.
      - Input(n) is data exiting n, Result is data entering n
      - Input(n) is ∅ if n is program exit
  - Function fn is the transfer function associated with each block n

The Mathematical Foundation of Dataflow Analysis

- Mathematical formulation of dataflow analysis
  - The property space L is used to represent the data flow domain information
  - The combination operator Λ: P(L) → L is used to combine information from different paths
- A set P is an ordered set if a partial order ≤ can be defined s.t. $\forall x,y,z \in P$
  - $x \leq x$ (reflexive)
  - If $x \leq y$ and $y \leq x$, then $x = y$ (asymmetric)
  - If $x \leq y$ and $y \leq z$ implies $x \leq z$ (transitive)
- Example: Power(L) with $\subseteq$ define the partial order

Upper and lower bounds

- Given an ordered set $(P, \leq)$, for each $S \subseteq P$
- Upper bound:
  - x is an upper bound of S if $x \in P$ and $\forall y \in S: y \leq x$
  - x is the least upper bound of S if
    - x is an upper bound of S, and
    - $x \leq y$ for all upper bounds y of S
  - The join operation V
    - V S is the least upper bound of S
    - x V y is the least upper bound of {x,y}
- Lower bound:
  - x is a lower bound of S if $x \in P$ and $\forall y \in S: x \leq y$
  - x is the greatest lower bound of S if
    - x is an lower bound of S, and
    - $y \leq x$ for all lower bounds y of S
  - The meet operation Λ
    - Λ S is the least upper bound of S
    - x Λ y is the least upper bound of {x,y}

Lattices

- An ordered set (L, ≤, V, Λ) is a lattice
  - If x Λ y and x V y exist for all x,y∈L
- An lattice (L,≤, Λ) is a complete lattice if
  - Each subset Y ⊆ L has a least upper bound and a greatest lower bound
    - LeastUpperBound(Y) = $\bigvee_{m \in Y} m$
    - GreatestLowerBound(Y) = $\bigwedge_{m \in Y} m$
- All finite lattices are complete
- Example lattice that is not complete: the set of all integers I
  - For any x, y∈I, x Λ y = min(x,y), x V y = max(x,y)
  - But LeastUpperBound(I) does not exist
  - I ∪ {+∞,−∞} is a complete lattice
- Each complete lattice has
  - A top element: the least element
  - A bottom element: the greatest element

Chains

- A set S is a chain if $\forall x,y \in S.\ y \le x$ or $x \le y$
- A set S has no infinite chains if every chain in S is finite
- A set S satisfies the finite ascending chain condition if
  - For all sequences $x_1 \le x_2 \le \ldots$, there exists n such that
    - $x_n = x_{n+1} = \ldots$
  - That is, all chains in S have an finite upper bound
- A complete lattice L satisfies the finite ascending chain condition if each ascending chain of L eventually stabilizes
  - If l1≤ l2 ≤ l3 ≤ … , then there is an upper bound ln = ln+1=ln+2…
  - This means starting from an arbitrary element e ∈ L, one can only increase e by a finite number of times before reaching an upper bound

Application to Dataflow Analysis

- Dataflow information will be lattice values
  - Transfer functions operate on lattice values
  - Solution algorithm will generate increasing sequence of values at each program point
  - Ascending chain condition will ensure termination
- Can use V (join) or Λ (meet) to combine values at control-flow join points

Example Dataflow Analysis

- Reaching Definitions
  - L = Power(Assignments)
    - L is partially ordered by subset inclusion
      - $\leq$ is subset relation; $\vee$ is set union
    - The least upper bound (join) operation is set union.
    - The least (top) element is $\emptyset$
  - L satisfies the finite ascending chain condition because Assignments is finite
- What about live variable analysis and available expression analysis?

Transfer Functions

- Each basic block n in a data-flow analysis defines a transfer function fn on the property space L (fn:L->L)

  $$Out(n) = f_n(In(n))$$

- The set of transfer functions F over L must satisfy the following conditions
  - F contains the identity function;
  - F is closed under composition of functions
    - Composition of monotone functions are also monotone
- All transfer functions are monotone if
  - For each e1, e2 $\in$ L, if e1 $\leq$ e2, then fn(e1) $\leq$ fn(e2);
- Sometimes transfer functions are distributive over the join/meet op

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

  - Distributivity implies monotonicity

Reaching Definitions

- P = power set of all definitions in program (all subsets of the set of definitions in program)
  - All transfer functions have the form
    $f(x) = GEN \cup (x\text{-}KILL)$
- Does it satisfy required lattice properties?
  - Does it support the required operations?
    - Three operations: ≤, V, Λ; bottom and top
  - Does it satisfy finite ascending chain condition?
- Are transfer functions monotone (distributive)?
  - Are they valid transfer functions?
    - $Df(x) = \emptyset \cup (x\text{-} \emptyset)$ is the identity function
    - What about composition?
  - Are they monotone?
    - if $x \subseteq y$, then $GEN \cup (x\text{-}KILL) \subseteq GEN \cup (y\text{-}KILL)$?
  - Are they distributive?
    $(GEN \cup (x\text{-}KILL)) \cup (GEN \cup (y\text{-}KILL)) = GEN \cup ((x \cup y)\text{-}KILL)$?

Reaching Definitions Composition and Distributivity

- Composition: given two transfer functions (f1 and f2)
  - $f_1(x) = a_1 \cup (x-b_1)$ and $f_2(x) = a_2 \cup (x-b_2)$, $f_1(f_2(x))$ can be expressed as $a \cup (x - b)$

$$
\begin{aligned}
f_1(f_2(x)) &= a_1 \cup ((a_2 \cup (x-b_2)) - b_1) \\
&= a_1 \cup ((a_2 - b_1) \cup ((x-b_2) - b_1)) \\
&= (a_1 \cup (a_2 - b_1)) \cup ((x-b_2) - b_1)) \\
&= (a_1 \cup (a_2 - b_1)) \cup (x-(b_2 \cup b_1))
\end{aligned}
$$

  - Let $a = (a_1 \cup (a_2 - b_1))$ and $b = b_2 \cup b_1$, then $f_1(f_2(x)) = a \cup (x - b)$
- Distributivity: $f(x \cup y) = f(x) \cup f(y)$

$$
\begin{aligned}
f(x) \cup f(y) &= (a \cup (x - b)) \cup (a \cup (y - b)) \\
&= a \cup (x - b) \cup (y - b) = a \cup ((x \cup y) - b) \\
&= f(x \cup y)
\end{aligned}
$$

Monotone Frameworks

- A monotone framework consists of
  - A complete lattice (L,≤) that satisfies the Ascending Chain Condition
  - A set F of monotone functions from L to L that
    - contains the identity function and
    - is closed under function composition
- A distributive framework is a monotone framework (L,≤, Λ,F) that additionally satisfies
  - All functions f in F are required to be distributive
    - f (l1 Λ l2) = f (l1) Λ f (l2)
- A bit-vector framework is a monotone framework that
  - L = Power(D), where D is a finite set
  - Each transfer function in F has the format Gen ∪ (Res-Kill)
  - All bit-vector frameworks are distributive
- Not all monotone frameworks are distributive
  - Example non-distributive framework: constant propagation

General Result

All GEN/KILL transfer function frameworks satisfy

- Identity
- Composition
- Distributivity

Properties

Worklist Algorithm for Solving Dataflow Equations

For each basic block n do

$In_n := \emptyset$ or Domain; $Out_n := f_n(In_n)$

$In_{n0} := \emptyset$; $Out_n := f_{n0}(In_{n0})$

worklist := {all basic blocks}-{ entry/exit block n0}

while worklist $\neq \emptyset$ do

remove a node n from worklist

$In_n := \cap$ or $\cup$ [m in pred(n) or succ(n)] Out_m

$Out_n := f_n(In_n)$

if Out_n changed then

worklist := worklist $\cup$ [succ(n) or pred(n)]

Meet Over Paths Solution

- What is the ideal solution for dataflow analysis?
- Consider a path $p = n_0, n_1, \ldots, n_k n$
  - for all i $n_i \in flow(n_{i+1})$
- The solution must take this path into account:

  $fp(top) = (f_{nk}(f_{nk-1}(\ldots f_{n1}(f_{n0}(top)) \ldots)) \leq In_n$
- So the solution must have the property that

  $$\wedge\{f_p\ (top)\ .\ p \text{ is a path to } n\} \leq In_n$$

  and ideally

  $$\wedge\{f_p\ (top)\ .\ p \text{ is a path to } n\} = In_n$$

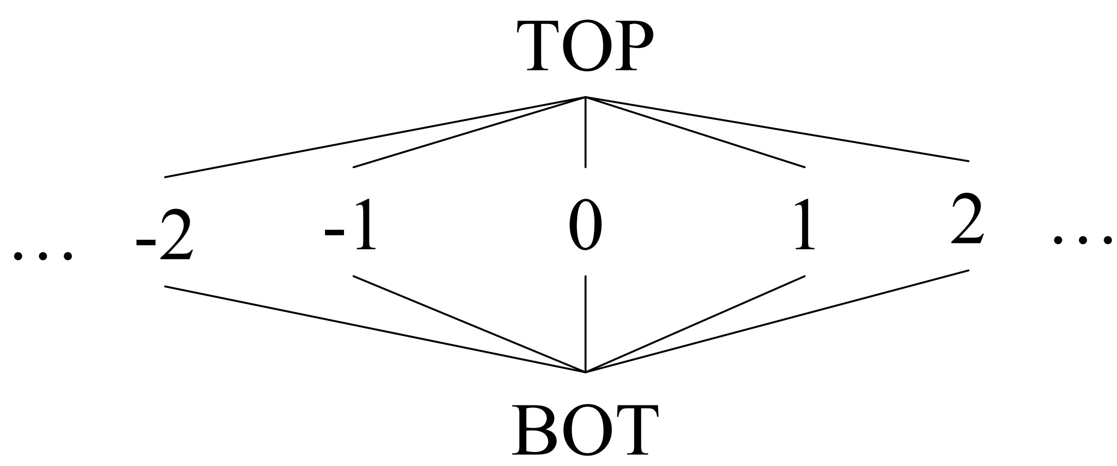
Distributivity

- Distributivity preserves control-flow precision
- If framework is distributive, then worklist algorithm produces the meet over paths solution
  - For each basic block n:

$$\wedge\{f_p\,(top)\ .\ p \text{ is a path to } n\} = In_n$$

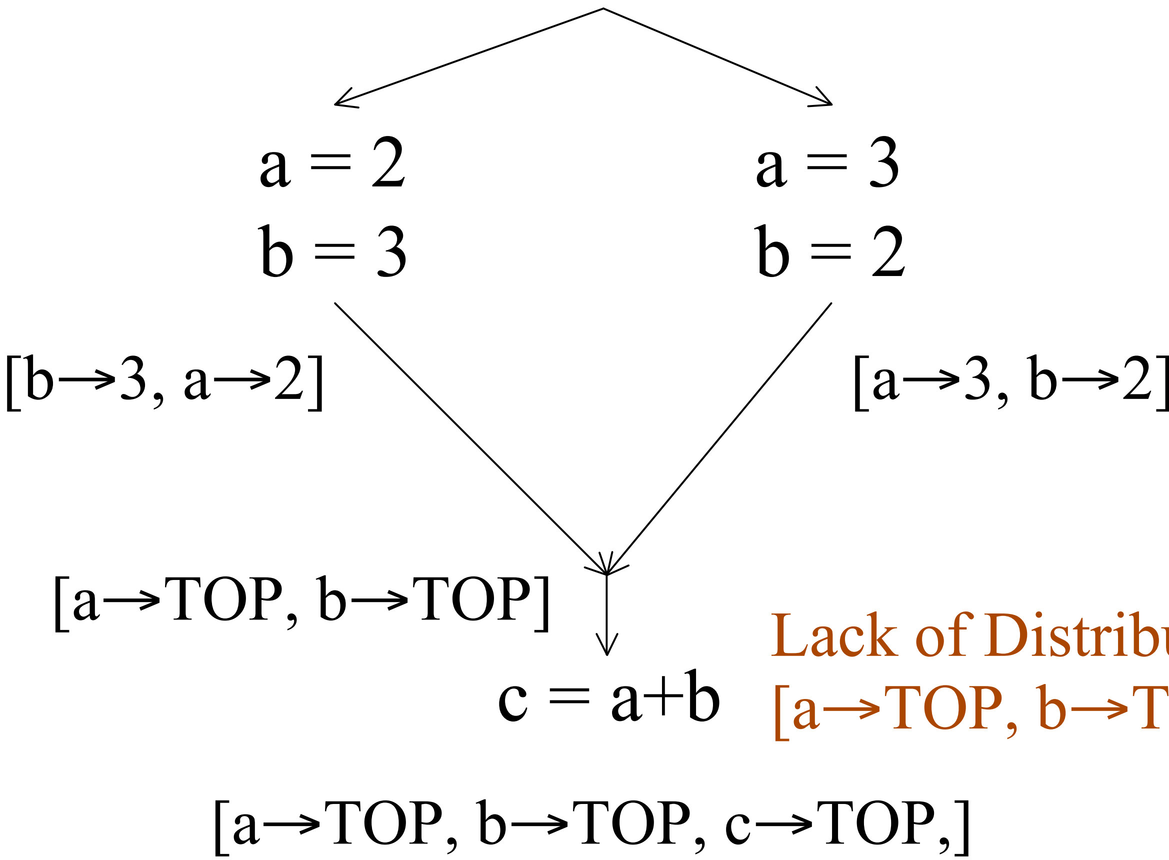
Lack of Distributivity Example

- Constant Calculator
- Flat Lattice on Integers



- Actual lattice records a single value for each variable
  - Example element: [a→3, b→2, c→5]

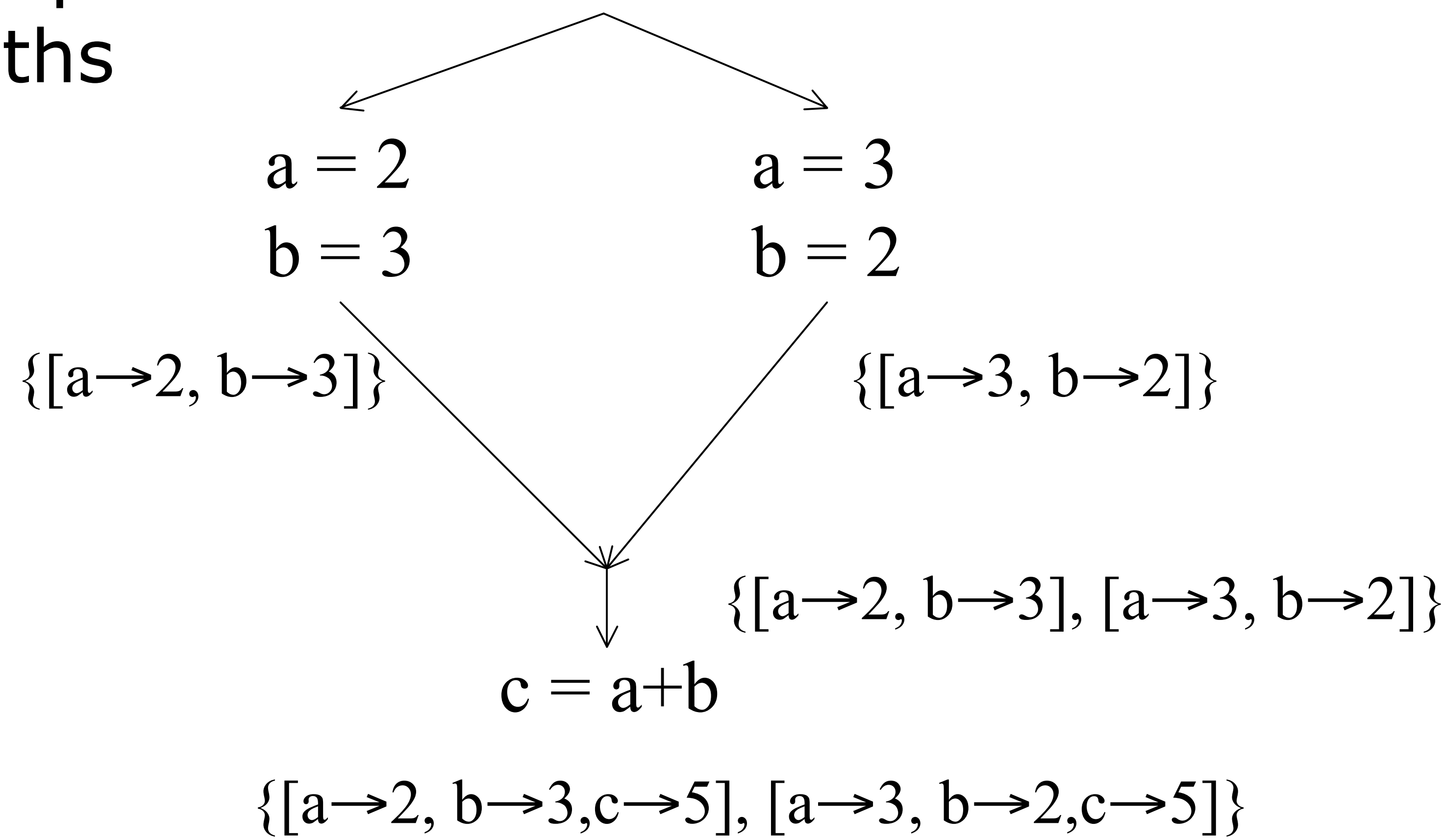
Lack of Distributivity Anomaly



Lack of Distributivity Imprecision:
[a→TOP, b→TOP, c→5] more precise

How to Make Analysis Distributive

- Keep combinations of values on different paths



Issues

- Basically simulating all combinations of values in all executions
  - Exponential blowup
  - Non-termination because of infinite ascending chains
- Non-termination solution
  - Use widening operator to eliminate blowup (can make it work at granularity of variables)
  - Lose precision in many cases

Termination Argument

- Why does algorithm terminate?
- For each basic block n,
  - Sequence of values taken on by In_n or Out_n is a chain.
  - If values stop increasing, worklist empties and algorithm terminates.
- If lattice has ascending chain property, algorithm terminates
  - Algorithm terminates for finite lattices
  - For lattices without ascending chain property, use widening operator

Widening Operators

- Detect lattice values that may be part of an infinitely ascending chain
- Artificially raise value to least upper bound of chain
- Example:
  - Lattice is set of all subsets of integers
  - Could be used to collect possible values taken on by variable during execution of program
  - Widening operator might raise all sets of size n or greater to Bottom (likely to be useful for loops)

General Sources of Imprecision

- Abstraction Imprecision
  - Concrete values (integers) abstracted as lattice values (e.g., use >0, =0, <0 to approximate values of a variable)
  - Lattice values less precise than execution values
  - Abstraction function throws away information
- Control Flow Imprecision
  - One lattice value for all possible control flow paths
  - Analysis result has a single lattice value to summarize results of multiple concrete executions
  - Join/meet operation moves up in lattice to combine values from different execution paths
  - Typically if $x \leq y$, then x is more precise than y

More about dataflow analysis

- Other data-flow problems
  - Reaching definition analysis
    - A definition point d of variable v reaches CFG point p iff there is a path from d to p along which v is not redefined
    - At any CFG point p, what definition points can reach p?
  - Very busy expression analysis
    - An expression e is very busy at a CFG point p if it is evaluated on every path leaving p, and evaluating e at p yields the same result.
    - At any CFG point p, what expressions are very busy?
  - Constant propagation analysis
    - A variable-value pair (v,c) is valid at a CFG point p if on every path from procedure entry to p, variable v has value c
    - At any CFG point p, what variables have constants?
  - Sign analysis
    - A variable-sign (>0,0,<0) pair (v,s) is valud at a CFG point p is on every path from procedure entry to p, variable v has sign s.

Theory and Application

- Dataflow analysis works (always terminates) on monotone frameworks
- Correctness
  - the iterative dataflow analysis algorithm always terminates and it computes the least (or Minimal Fixed Point) solution to the instance of monotone framework given as input
- Complexity
  - Suppose that the input control-flow graph contains
    - at most $b \geq 1$ distinct basic blocks (nodes)
    - at most $e \geq b$ edges
  - Suppose the complete lattice L has a finite height at most $h \geq 1$
  - Suppose each transfer function takes a single op (constant time)
  - Then there will be at most $O(e \cdot h)$ basic operations.
- Example: build instances of monotone frameworks for various dataflow analysis