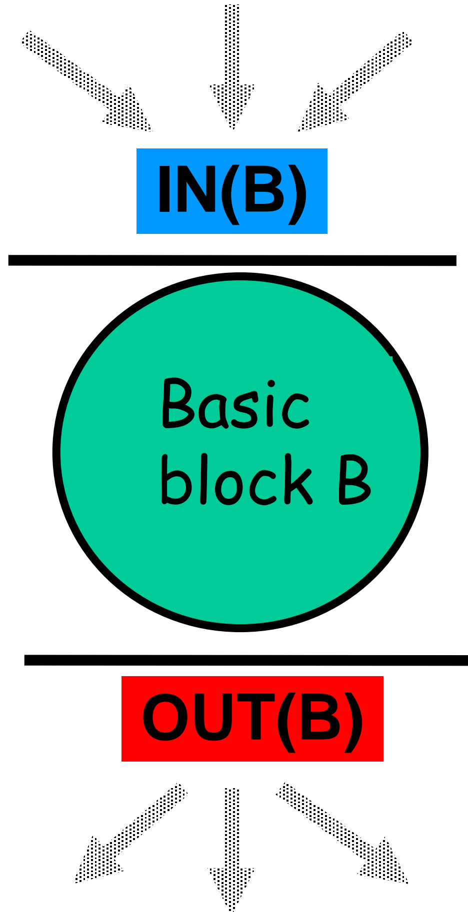
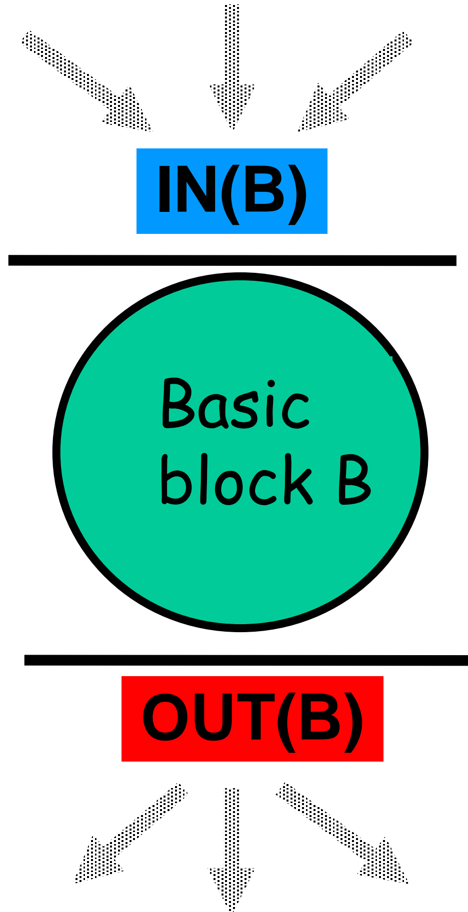
Classification of Data Flow Problems : Propagation



IN(B) : data flow information valid on entry to basic block B

OUT(B) : data flow information valid on exit from basic block B

Classification of Data Flow Problems : Propagation



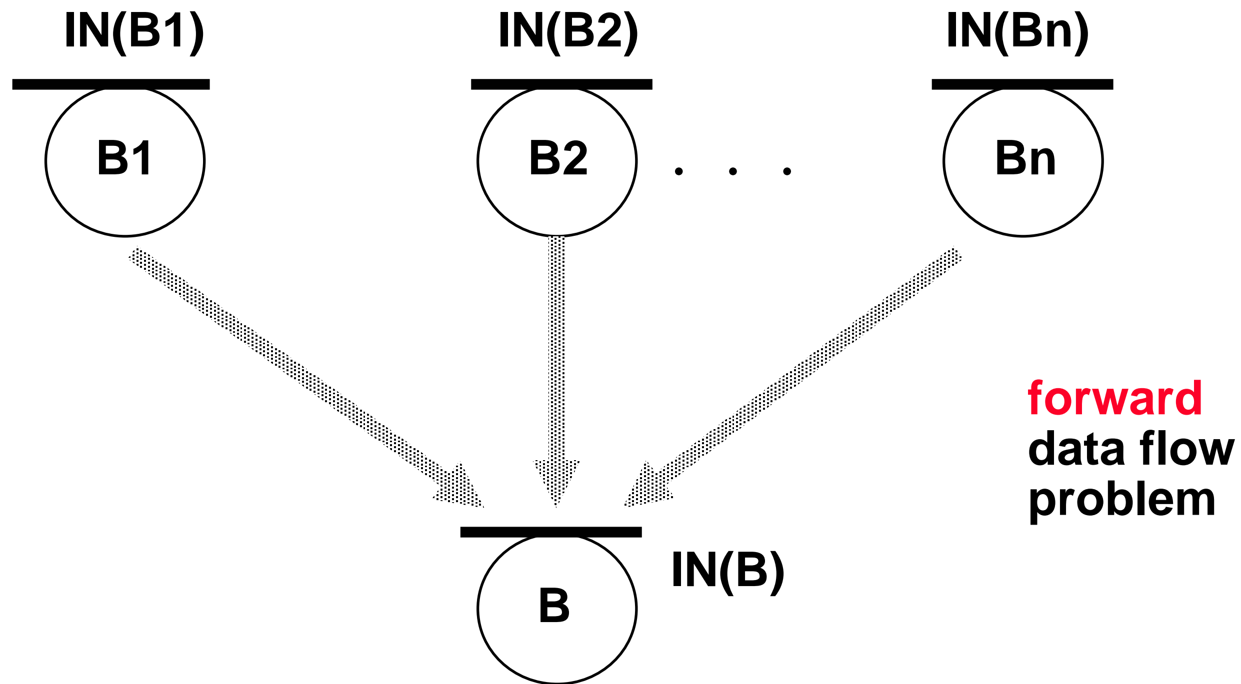
IN(B) : data flow information
valid on entry to basic block B

OUT(B) = GEN(B) + [IN(B) - KILL(B)]

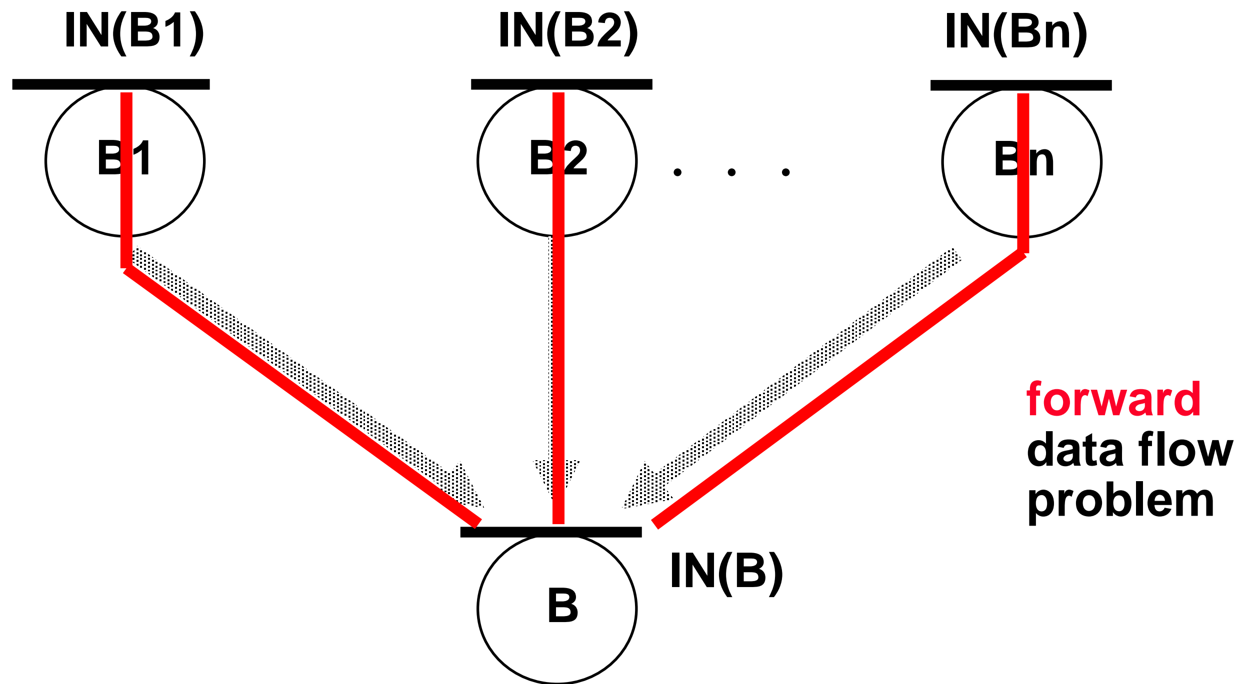
GEN and KILL describe the effect of basic block B on data flow information

OUT(B) : data flow information
valid on exit from basic block B

Classification of Data Flow Problems : Flow Direction

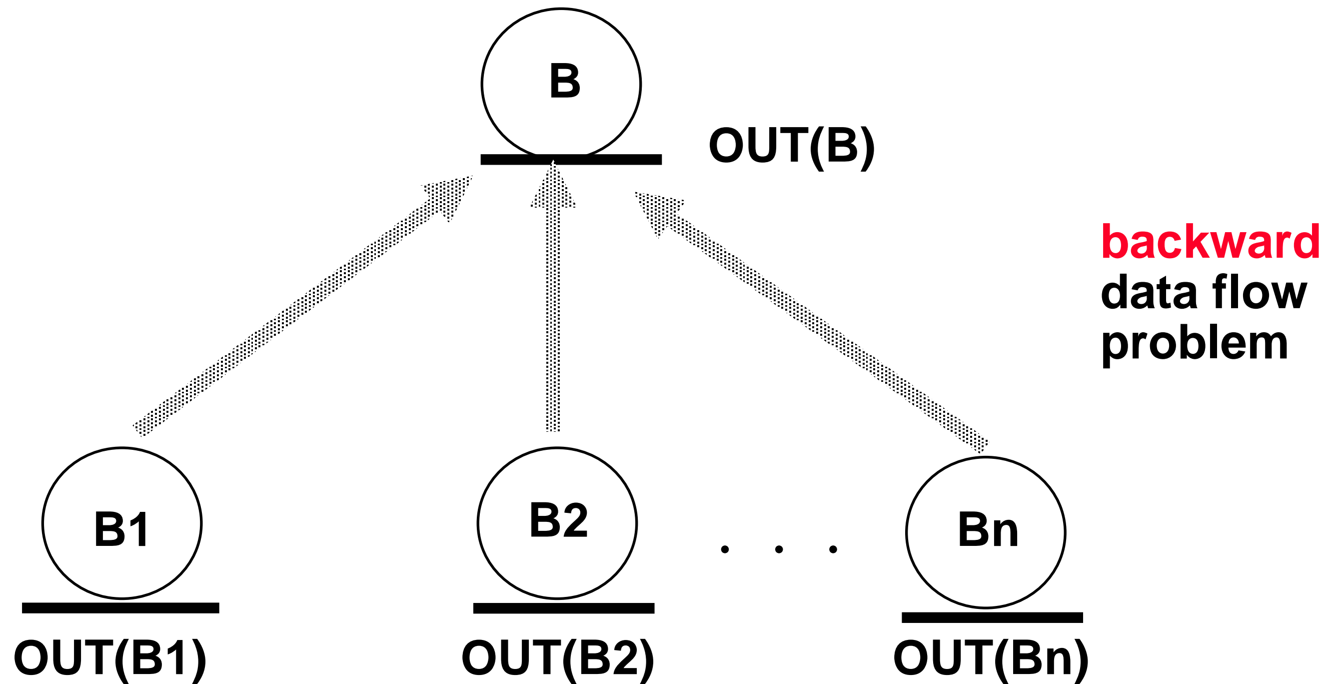


Classification of Data Flow Problems : Flow Direction

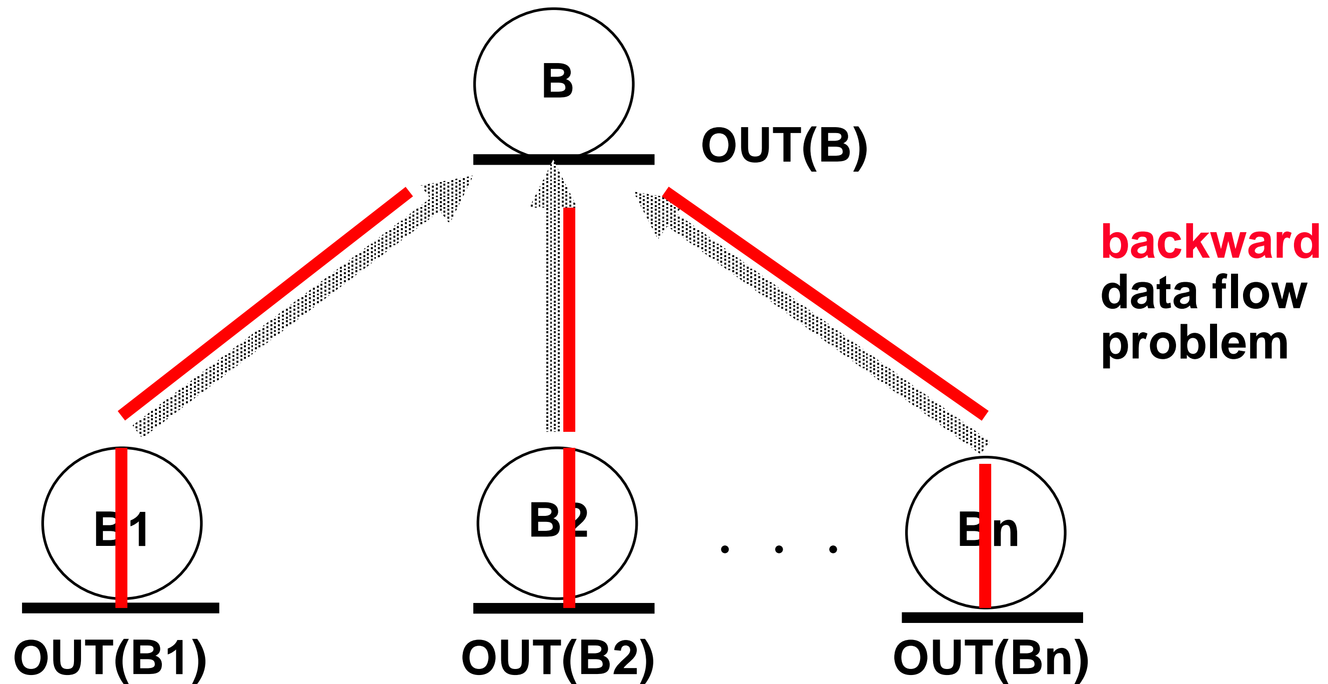


$$IN(B) = \bigotimes_{Bi \,\varepsilon\, PRED(B)} (GEN(Bi) \cup [IN(Bi) - KILL(Bi)])$$

Classification of Data Flow Problems : Flow Direction

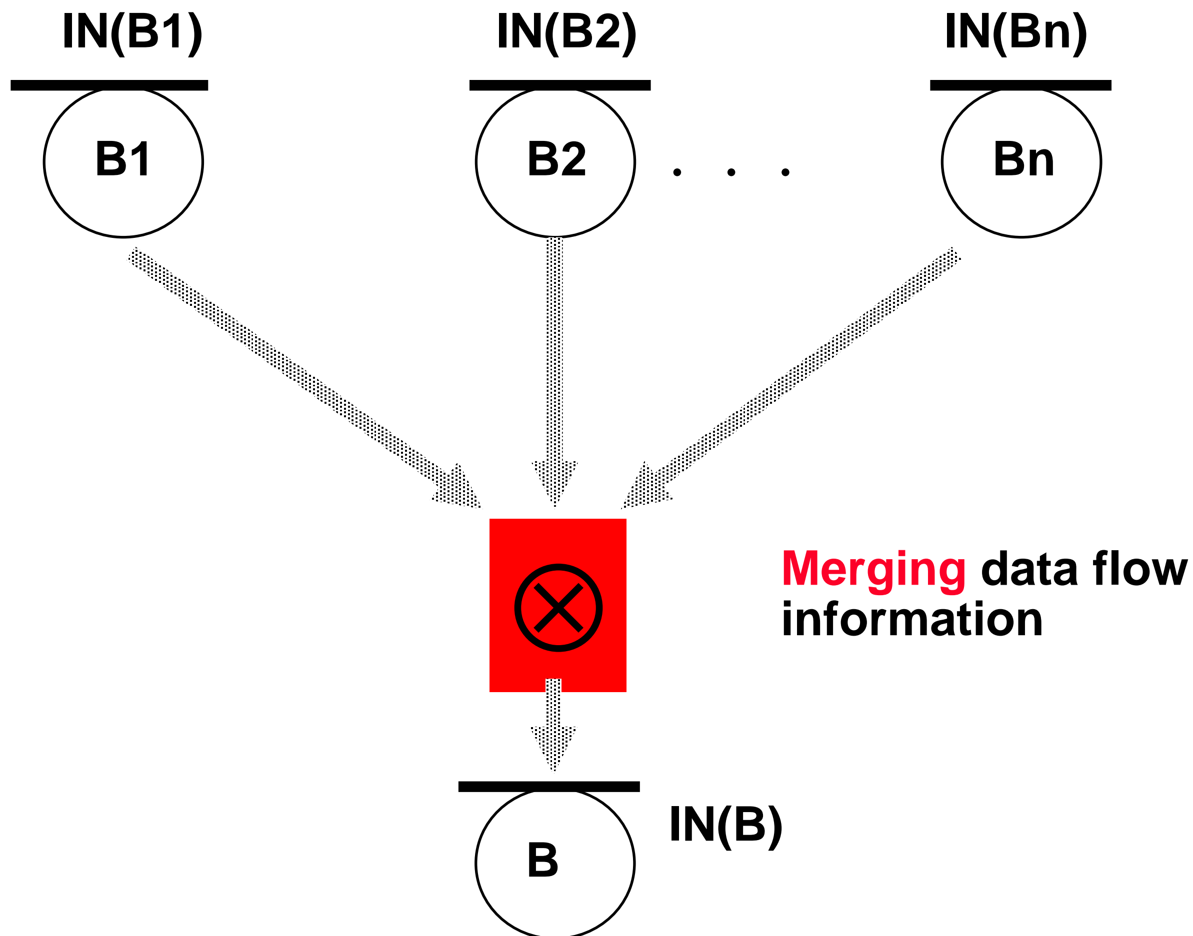


Classification of Data Flow Problems : Flow Direction

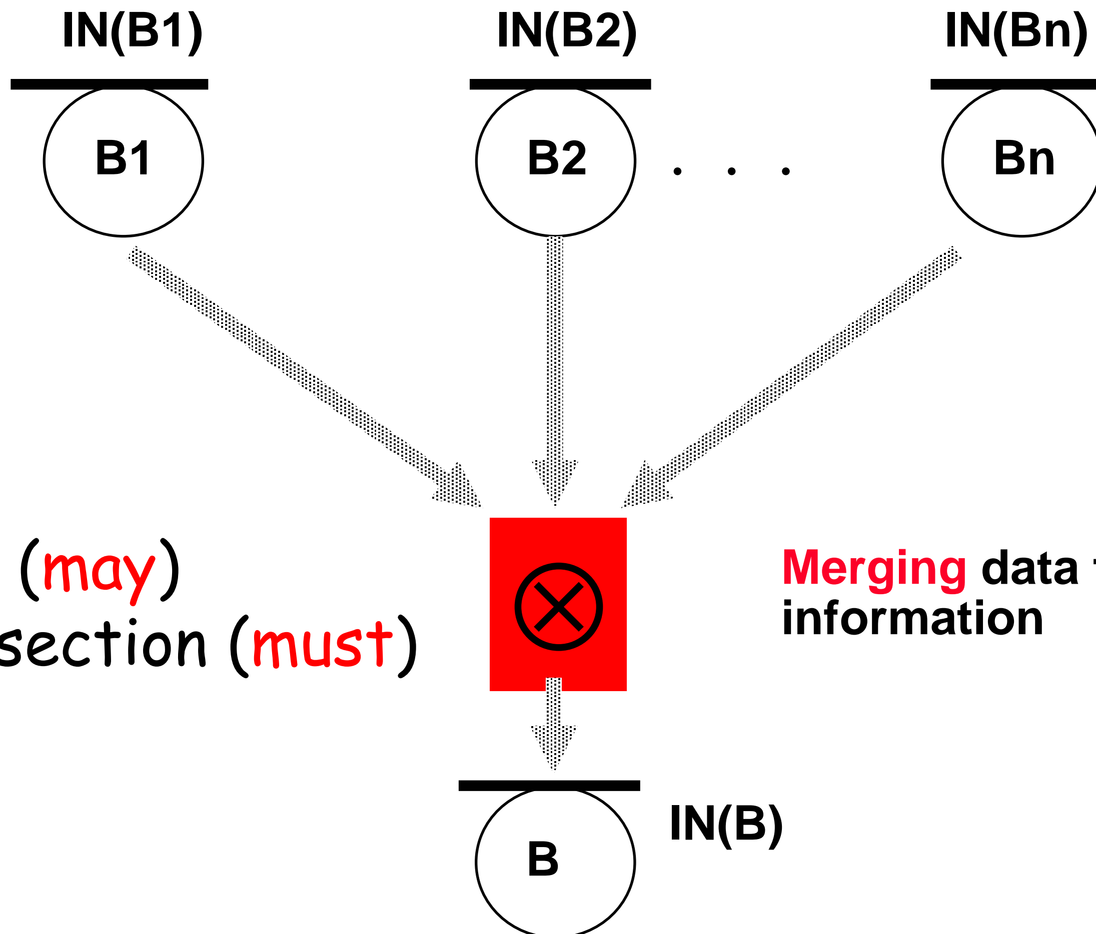


$$OUT(B) = \bigotimes_{Bi \,\varepsilon\, SUCC(B)} (GEN(Bi) \cup [OUT(Bi) - KILL(Bi)])$$

Classification of Data Flow Problems : Merging Information



Classification of Data Flow Problems : Merging Information



- ∪ union (may)
- ∩ intersection (must)

Classification of Data Flow Problems -- Examples

	forward	backward
may: there exists a path (union)	RD	LV
must: for all paths (intersection)	AVAIL	VBE

Available Expressions

An expression e is defined if its value is computed.

An expression e is killed if the values of any of its operands may have been changed.

$$AVAIL(B) = \bigcap_{Bi \, \varepsilon \, PRED(B)} (GEN(Bi) \cup [AVAIL(Bi) - KILL(Bi)])$$

GEN(Bi) : All definitions of expressions in Bi that are not subsequently killed in Bi

KILL(Bi): All expressions with operand variables defined in Bi

Available Expressions Example

Universe of facts?

GEN and KILL sets for each basic block?

Initial values of AVAIL(B) before propagation starts?

Final solution?

Very Busy Expressions

An expression e is defined if its value is computed.

An expression e is killed if the values of any of its operands may have been changed.

$$VBE(B) = \bigcap_{Bi \in SUCC(B)} (GEN(Bi) \cup [VBE(Bi) - KILL(Bi)])$$

GEN(Bi) : All definitions of expressions in Bi that are not previously killed in Bi

KILL(Bi): All expressions with operand variables defined in Bi

Very Busy Expressions Example

VBE solution supports code hoisting (code motion) to reduce code size.

Universe of facts?

GEN and KILL sets for each basic block?

Initial values of VBE(B) before propagation starts?

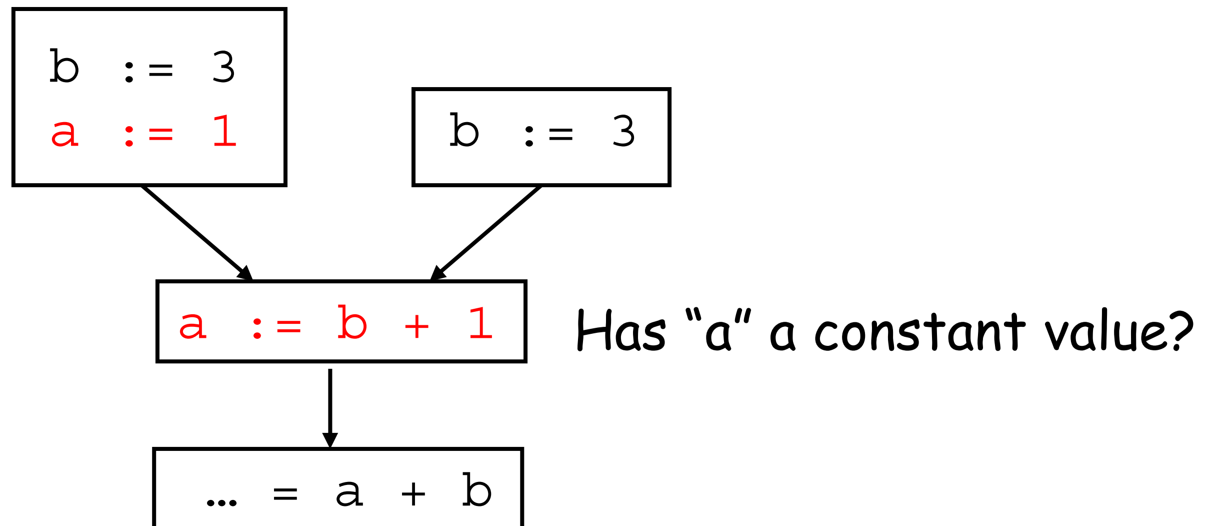
Final solution?

Aliasing Problem

Aliasing problem: multiple "names" for the same variable or data object

Example:

Is this dead code?



Aliasing Problem

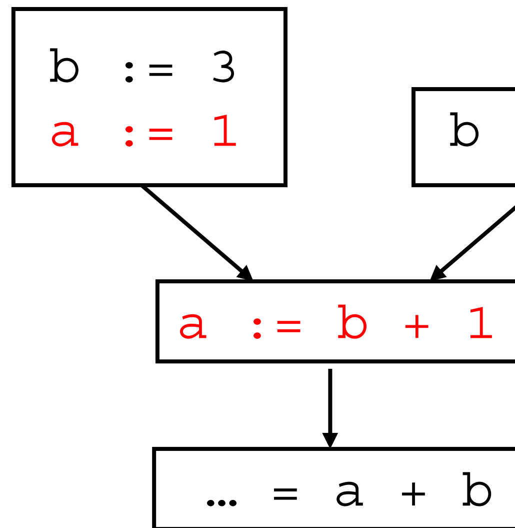
Aliasing problem: multiple "names" for the same variable or data object

May occur due to:

- globals and formal parameters
- array references
- pointers

Example:

Is this dead code?



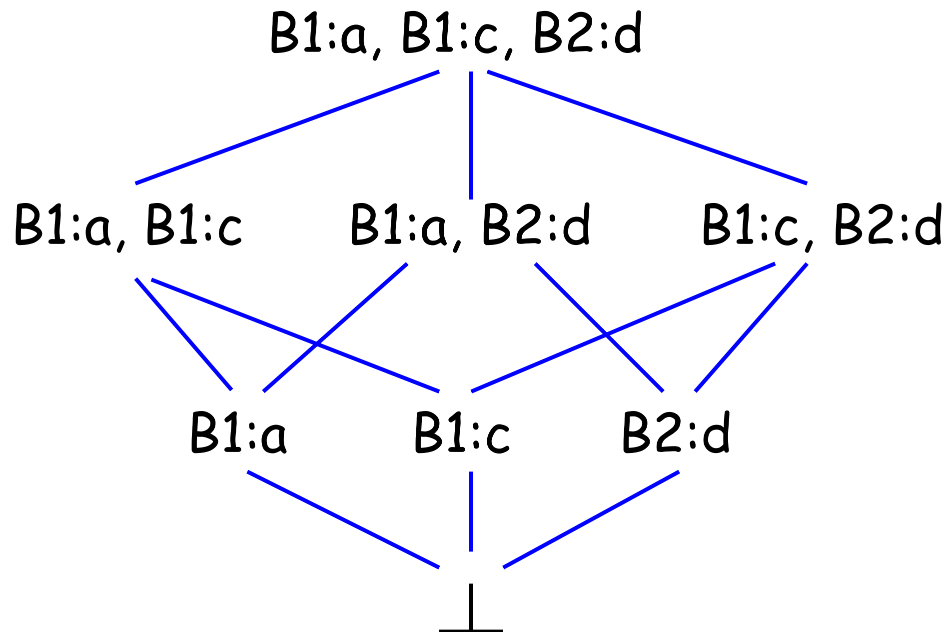
Has "a" a constant value?

MUST ALIAS vs.
MAY ALIAS

Implementation Issues: Bit-vector Problems

The set of facts (universe of facts) can often be expressed as finite subsets of a finite base set. Such sets can be represented as bit-vectors.

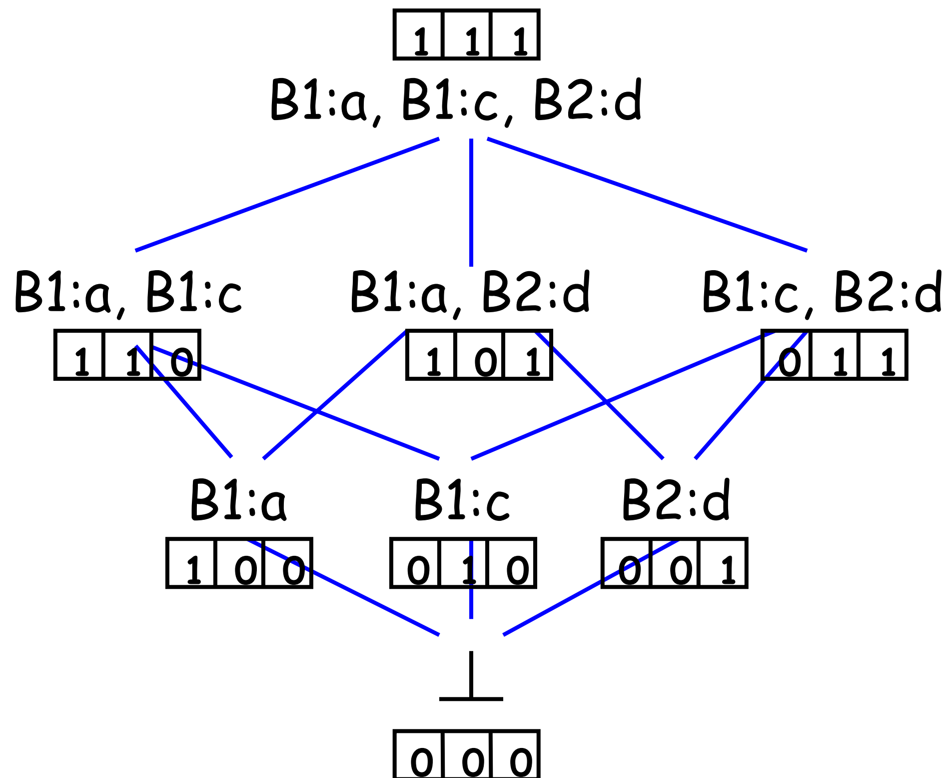
Example: RD --   { B1: a=2, B1: c = d+2, B2: d = a-5 }



Implementation Issues: Bit-vector Problems

The set of facts (universe of facts) can often be expressed as finite subsets of a finite base set. Such sets can be represented as bit-vectors.

Example: RD -- { B1: a=2, B1: c = d+2, B2: d = a-5 }



Implementation Issues: Bit-vector Problems

Meet operation is either bit-wise logical $\wedge$ or bit-wise logical $\vee$.

GEN and KILL sets can be expressed as single bit-vectors.

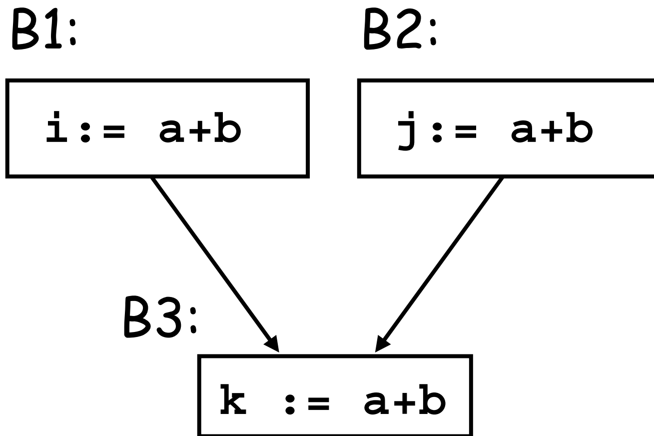
Bit-wise logical - is bit-wise negation followed by bit-wise $\wedge$.

Implementation steps:

- bit-vector construction/interpretation
- bit-vector CFG initialization (RD, GEN, and KILL vectors)
- bit-vector CFG propagation
- information post-processing (e.g.: DU/UD chains)

Bit-vector Problem?

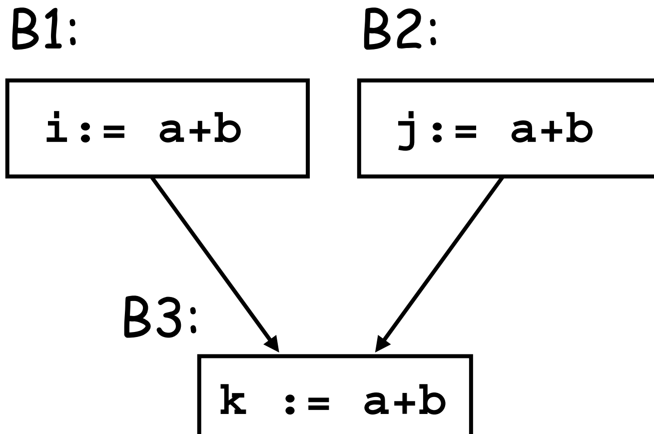
Here is a proposal:



- change lattice of AVAIL to reflect statements where expressions are generated.
- How does lattice look like?
- GEN and KILL functions?
- Confluence (meet) operator ⊗?

Bit-vector Problem?

Here is a proposal:



- change lattice of AVAIL to reflect statements where expressions are generated.

- How does lattice look like?

  `(B1:a+b),(B2:a+b),(B3:a+b)`

- GEN and KILL functions?

  `B1:(B1:a+b)in GEN, KILL is empty`
  `B2:(B2:a+b)in GEN, KILL is empty`
  `B3:(B3:a+b)in GEN, KILL is empty`

- Confluence (meet) operator ⊗?

  `NOT bit-wise logical` $\wedge$

Non Bit-vector Problem

Constant propagation can be formulated as a data flow problem without using DU/UD chains.

$$CONST(B) = \bigotimes_{B_i \in PRED(B)} \left(GEN(B_i) \cup [\, CONST(B_i) - KILL(B_i) \,] \right)$$

GEN(Bi) : Set of immediate constants (static) and dynamically encountered constants (dynamic)

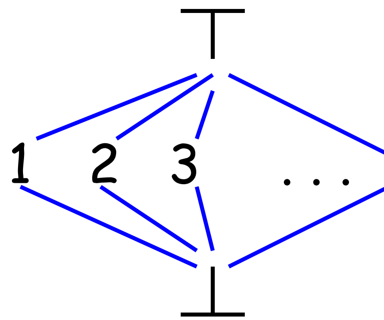
KILL(Bi): All definitions of variables x defined in Bi

Non Bit-vector Problem

The set of facts (universe of facts) are set of pairs

(variable, value)

where value is from the following lattice:



Examples: (a,T), (b,5)

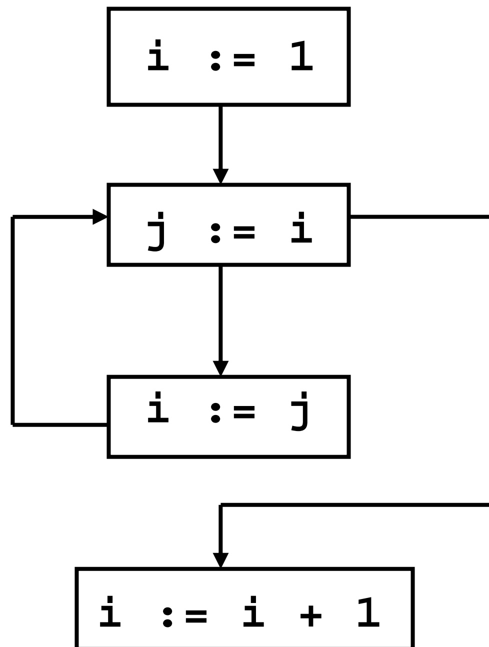
Note: Typically, constant propagation is only performed on integral values, not floating point values.

Non Bit-vector Problem

The confluence (meet) operator ⊗ for the second component of a (variable, value) pair is defined as follows:

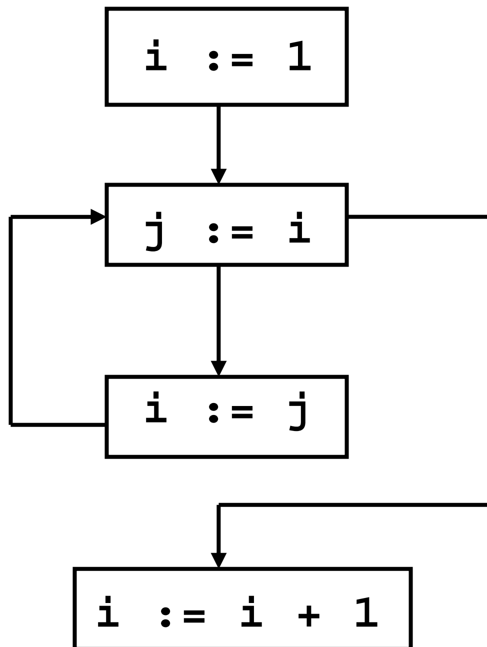
$\otimes$	$\top$	c_2	$\perp$
$\top$	$\top$	c_2	$\perp$
c_1	c_1	If $c_1 = c_2$ then c_1 else $\perp$	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

Non Bit-vector Problem



Initialization	1st prop. step	2nd prop. step
(i, $\top$), (j, $\top$)	(i, $\top$), (j, $\top$)	(i, $\top$), (j, $\top$)
(i, $\top$), (j, $\top$)	(i, 1), (j, $\top$)	(i, 1), (j, 1)
(i, $\top$), (j, $\top$)	(i, 1), (j, 1)	(i, 1), (j, 1)
(i, $\top$), (j, $\top$)	(i, 1), (j, 1)	(i, 1), (j, 1)

Non Bit-vector Problem



Remarks:

- optimistic constant propagation
- constant propagation typically done on UD/DU chains, but similar principles apply
- pessimistic constant propagation does not find any constants in our example
- intermediate results are **not safe** in optimistic approach

Basic Iterative Algorithm

```
change := true;
initialize IN(B) with T;
while (change) {
   change := false;
   while ( ∃j : IN(Bj) ≠   ⊗             ( GEN(Bi) ∪ [ IN(Bi) - KILL(Bi) ] )   ) {
                         Bi ε PRED(Bj)

               IN(Bj) :=  ⊗             ( GEN(Bi) ∪ [ IN(Bi) - KILL(Bi) ] )
                         Bi ε PRED(Bj)
               change := true;
   }
}
```

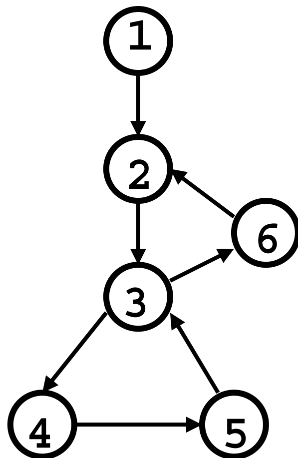
This algorithm is not very efficient: needs refinement.

Iterative Algorithms

There are three basic variants of the iterative algorithm:

- worklist version
- round robin version
- node-list version

Iterative algorithms are different from structural (elimination) algorithms.

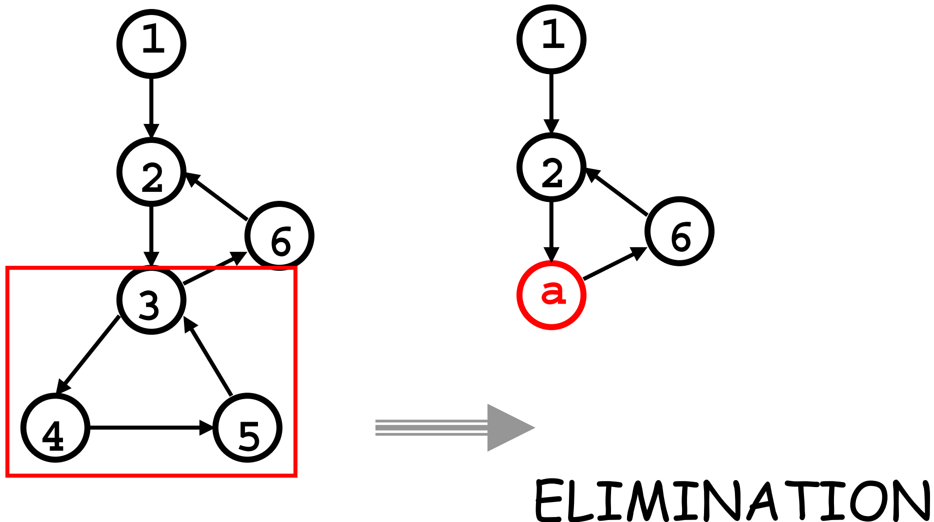


Iterative Algorithms

There are three basic variants of the iterative algorithm:

- worklist version
- round robin version
- node-list version

Iterative algorithms are different from structural (elimination) algorithms.

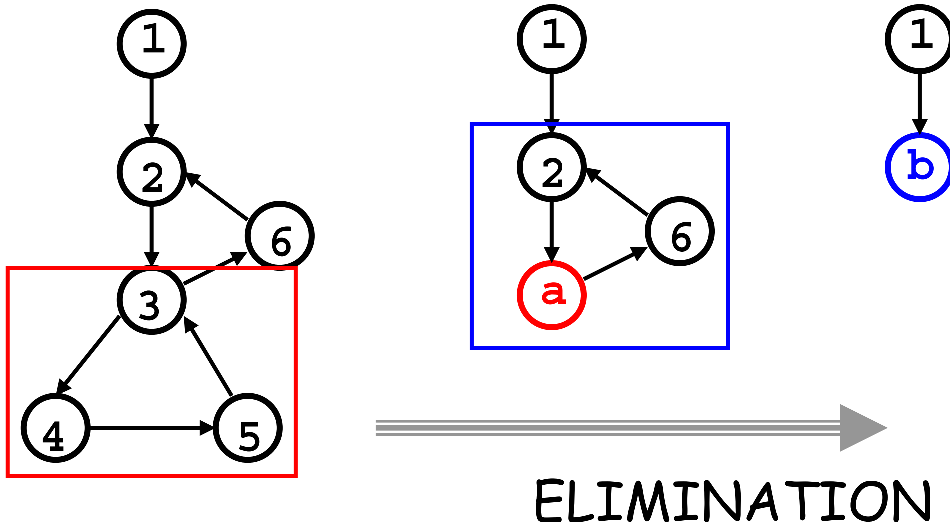


Iterative vs. Structural

There are three basic variants of the iterative algorithm:

- worklist version
- round robin version
- node-list version

Iterative algorithms are different from structural (elimination) algorithms.

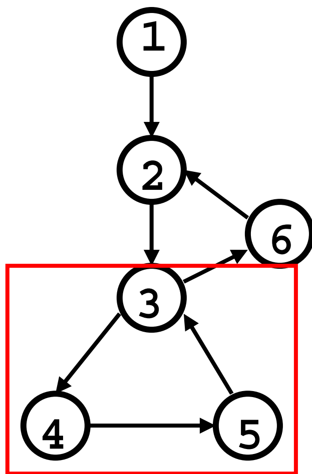


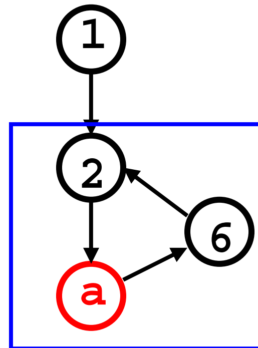
Iterative vs. Structural

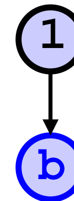
There are three basic variants of the iterative algorithm:

- worklist version
- round robin version
- node-list version

Iterative algorithms are different from structural (elimination) algorithms.







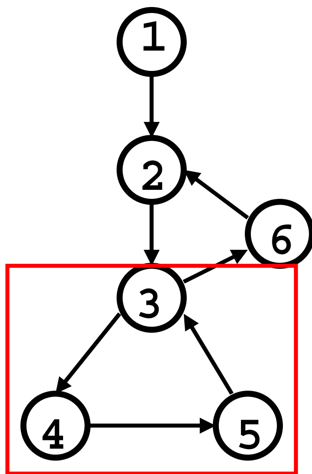
SOLUTION + PROPAGATION

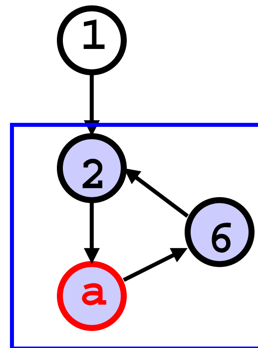
Iterative vs. Structural

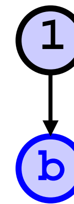
There are three basic variants of the iterative algorithm:

- worklist version
- round robin version
- node-list version

Iterative algorithms are different from structural (elimination) algorithms.







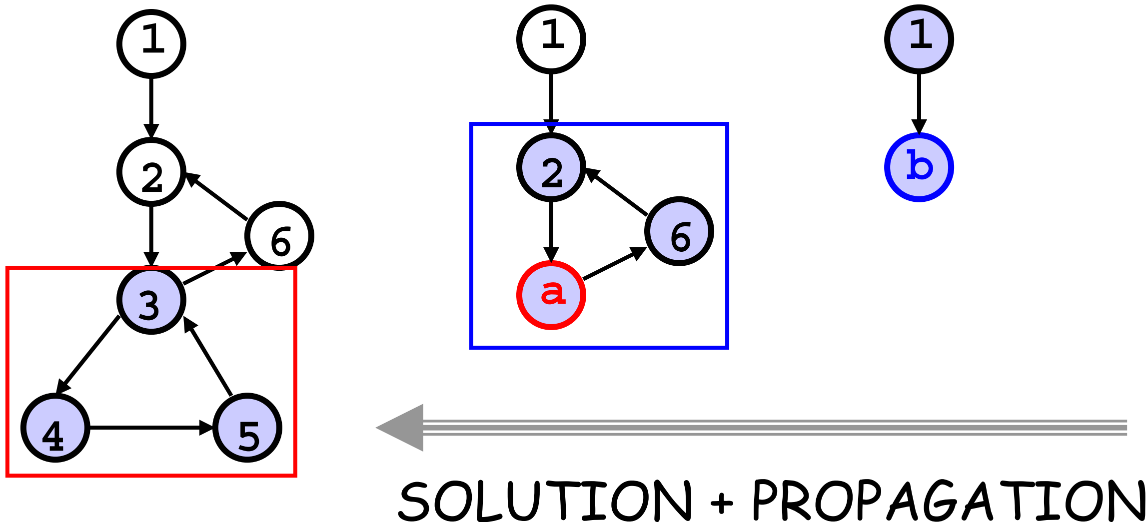
SOLUTION + PROPAGATION

Iterative vs. Structural

There are three basic variants of the iterative algorithm:

- worklist version
- round robin version
- node-list version

Iterative algorithms are different from structural (elimination) algorithms.



Iterative vs. Structural

Structural algorithms use different lattices for constructing summary functions for loop constructs.

Are more complex to program.

Up to a factor of 2x faster than iterative algorithm.

Most compilers use iterative algorithm since it is still fast as compared to other compilation phases.

Worklist Iterative Algorithm (Forward Problem)

```
for i := 1, n  do  initialize IN(Bi) with T;
W := {1, … , n} // every CFG node is initially in worklist W

while ( W ≠ Ø ) {
   remove j from W;

   new :=    ⊗      ( GEN(Bi) ∪ [ IN(Bi) - KILL(Bi) ] )
        Bi ε PRED(Bj)
  if (new ≠ IN(Bj) {
    IN(Bj) := new;
    for k ε SUCC(Bj) do
       add k to W if not already there }
}
```

Round Robin Iterative Algorithm (Forward Problem)

```
for j := 1, n  do  initialize IN(Bj) with T;
change := true;

while ( change ) {
   change := false
   for (j:= 2 to n in rPOSTORDER ) {

      new :=      ⊗       ( GEN(Bi) ∪ [ IN(Bi) - KILL(Bi) ] )
              Bi ε PRED(Bj)
      if (new ≠ IN(Bj) {
         IN(Bj) := new; change := true }
   }
}
```

POSTORDER and Reverse POSTORDER

Step 1: POSTORDER

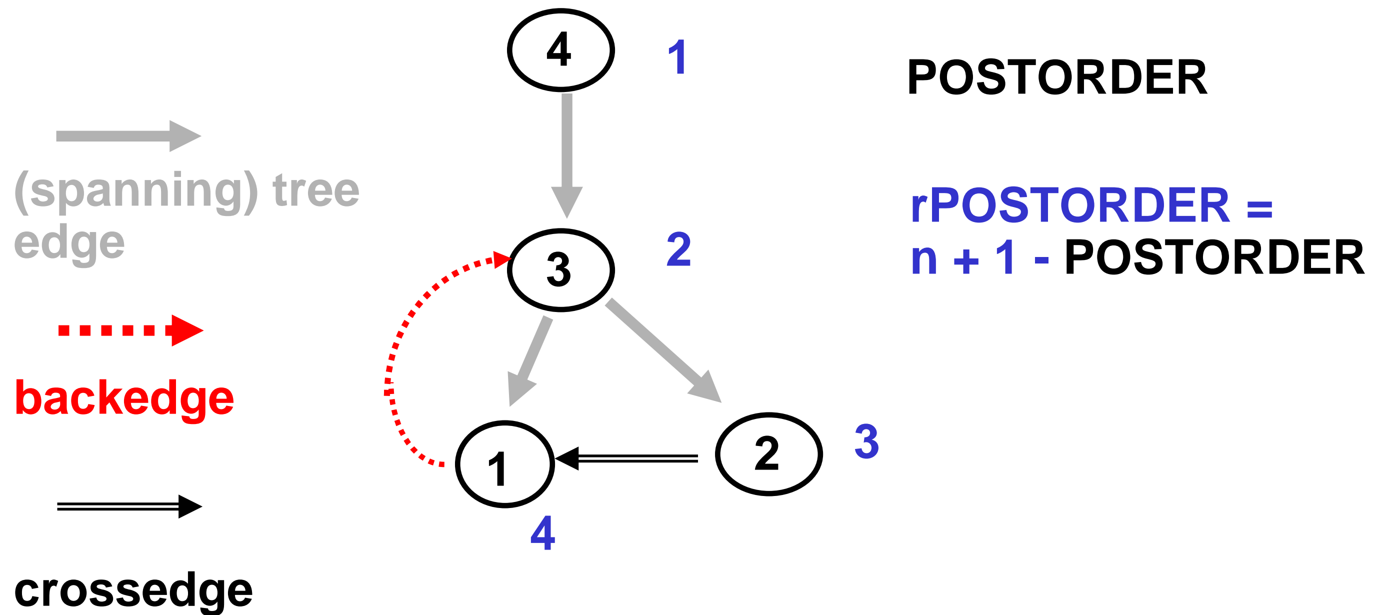
```
Main()
  count = 1;
  Visit (root)

Visit(n)
  mark  n  as visited
  for each successor s  of  n  not yet visited
       Visit(s);
  POSTORDER(n) = count;
  count = count + 1;
```

Step2: rPOSTORDER

```
For each node  n
  rPOSTORDER(n) = NumNodes + 1  - POSTORDER(n)
```

POSTORDER and Reverse POSTORDER

Example:



Node-list Iterative Algorithm (Forward Problem)

for j := 1, n do initialize IN(Bj) with T;

foreach j in NODELIST- ORDER do {

$$IN(B_j) := \bigotimes_{B_i \in PRED(B_j)} (GEN(B_i) \cup [IN(B_i) - KILL(B_i)])$$

}

Need to construct NODELIST first!

Partially Ordered Sets

- Poset: Partially ordered sets $(S, \leq)$
  - $\leq$ is relation between pair of elements in S
  - reflexive $x \leq x$
  - anti-symmetric $x \leq y, y \leq x \Rightarrow x = y$
  - transitive $x \leq y,\ y \leq z \Rightarrow x \leq z$

- 0-element ($\perp$): $\forall x \in S: \perp \leq x$

- 1-element (T): $\forall x \in S: x \leq T$

A Poset may not have a $\perp$ (bottom) or T (top) element

Lattice Theory: Poset $(S, \leq)$

- greatest lower bound (glb)

  for $x \in S$ and $y \in S$, $a \in S$ is glb(x,y) iff

  $a \leq x$ and

  $a \leq y$ and

  $\neg\exists b \in S: a < b \leq x$ and $a < b \leq y$

  if a = glb(x,y) is unique, it is called the meet of x and y, $a = x \wedge y$

- least upper bound (lub)

  for $x \in S$ and $y \in S$, $a \in S$ is lub(x,y) iff

  $x \leq a$ and

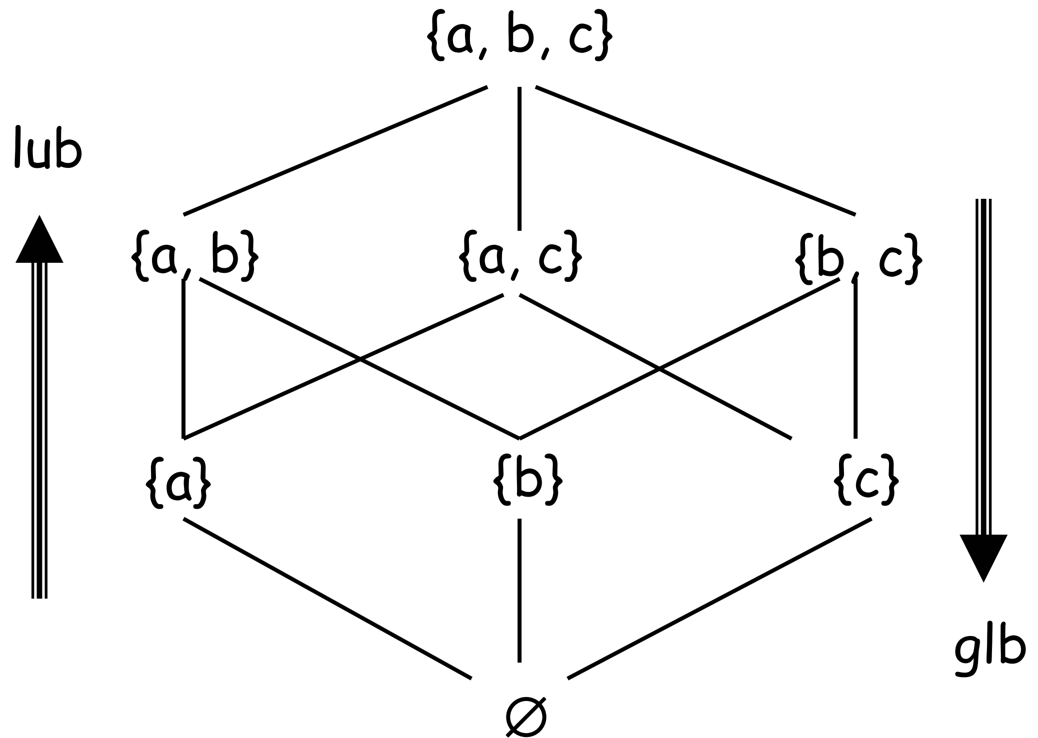
  $y \leq a$ and

  $\neg\exists b \in S: x \leq b < a$ and $y \leq b < a$

  if a = lub(x,y) is unique, it is called the join of x and y, $a = x \vee y$

Poset (2^S, ≤): Example

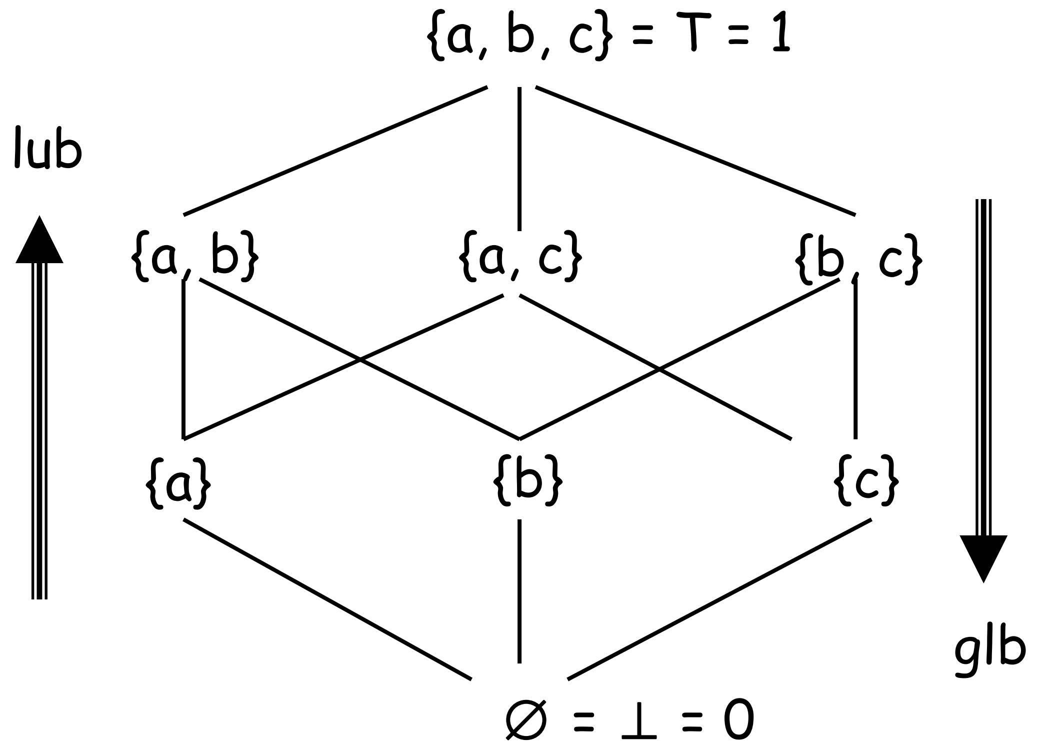
$S = \{a, b, c\}$
Poset = 2^S
≤ is set inclusion ⊆



$\{a\} \wedge \{b,c\} =$
$\{b\} \vee \{c\} =$

Poset $(2^S, \leq)$: Example

$S = \{a, b, c\}$
Poset = 2^S
$\leq$ is set inclusion $\subseteq$

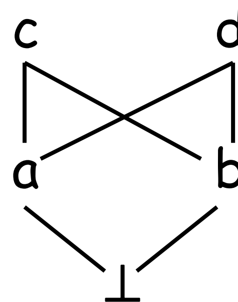


$\{a\} \wedge \{b,c\} = \varnothing$
$\{b\} \vee \{c\} = (b, c)$

Definition of a Lattice

Lattice (L, $\wedge$, $\vee$)

- L is a Poset under such that every pair of elements has a unique glb (meet) and a unique lub (join).
- A lattice need not contain a 0 (bottom $\perp$) or 1 (top T) element.
- A finite lattice must contain a 0 and 1 element.
- Not every Poset is a lattice



- $a \leq b \Leftrightarrow a \wedge b = a$
- $a \leq b \Leftrightarrow a \vee b = b$

Example Lattices

(1) $H = (2^U, \cap, \cup)$, where U is a finite set

- $\wedge$ is $\cap$
- $\vee$ is $\cup$

How to define $\leq$?

(2) $J = (N_1, gcd, lcm)$, where N_1 are natural numbers without 0

- $\wedge$ is gcd: greatest common divisor
- $\vee$ is lcm: least common multiple

How to define $\leq$?
Top element?
Bottom element?

Example Lattices

(1) $H = (2^U, \cap, \cup)$, where U is a finite set

- $\wedge$ is $\cap$
- $\vee$ is $\cup$

How to define $\leq$? Answer: $\subseteq$

(2) $J = (N_1, gcd, lcm)$, where N_1 are natural numbers without 0

- $\wedge$ is gcd: greatest common divisor
- $\vee$ is lcm: least common multiple

How to define $\leq$? Answer: divides evenly
Top element? Answer: none
Bottom element? Answer: 1

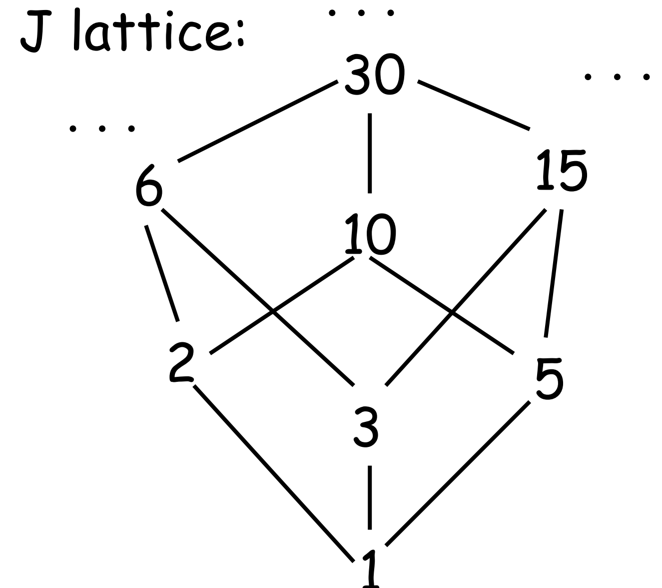
Definition of Chains

Chains: Given a Poset $(C, \leq)$.

An ascending chain is a sequence of elements $x_i \in C$, such that $x_i < x_{i+1}$: $x_1 < x_2 < x_3 < \ldots$

A descending chain is a sequence of elements $x_i \in C$, such that $x_i > x_{i+1}$: $x_1 > x_2 > x_3 > \ldots$

Examples:
1. $\emptyset < \{a\} < \{a, b\} < \{a, b, c\}$
2. J lattice: $1 < 2 < 6 < 30$
3. J lattice: $30 > 15 > 3 > 1$



More Definitions

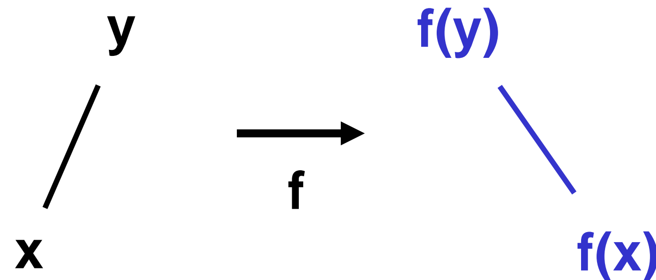
- Finite length lattice: if every chain in lattice is finite
- Bounded lattice: if lattice contains both 0 and 1 elements
- Distributive lattice: if $\forall$ x,y,z $\in$ S:

  $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ and

  $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
- (S, $\leq$) poset, function f: S --> S is monotone iff

  $\forall$ x,y $\in$ S: $x \leq y \Rightarrow f(x) \leq f(y)$

  Monotonic functions preserve domain ordering in their range values



Example Lattices

Can a finite length lattice be infinite?

Give a lattice that has infinite chains.

Give a bounded lattice that has infinite chains.

Give a bounded lattice that has finite descending chains, but infinite ascending chains.

Fixpoint Theorem (Intuition)

Given a 0 in lattice and monotone function f, $0 \leq f(0)$.

Apply f again and obtain

$$0 \leq f(0) \leq f(f(0)) = f^2(0)$$

Continuing,

$$0 \leq f(0) \leq f^2(0) \leq f^3(0) \leq \ldots f^k(0) = f^{k+1}(0)$$

for a finite chain lattice.

In other words, $\lim_{k \rightarrow infinity} f^k(0)$

exists and is called *minimal fixed point* of f,
since $f(f^k(0)) = f^k(0)$

Fixpoint Theorem (Intuition)

Given a 1 in lattice and monotone function f, $f(1) \le 1$.

Apply f again and obtain

$$f^2(1) = f(f(1)) \le f(1) \le 1$$

Continuing,

$$f^{k+1}(1) = f^k(1) \ldots \le f^3(1) \le f^2(1) \le f(1) \le 1$$

for a finite chain lattice.

In other words, $\lim_{k \to \text{infinity}} f^k(1)$

exists and is called *maximal fixed point (MFP)* of f,

since $f(f^k(1)) = f^k(1)$

Fixpoint Theorem

Theorem: f: S --> S monotone function on poset $(S, \leq)$ with a 0 element and finite length. The minimal fixed point of f is $f^k(0)$ where

i. $f^0(x) = x$,

ii. $f^{i+1}(x) = f(f^i(x))$, $i \geq 0$,

iii. $f^k(0) = f(f^k(0))$ and this is the smallest k for which this is true (minimal fixed point is unique).□

- For any p such that f(p)=p, $f^k(0) \leq p$.
- Theorem justifies the iterative algorithm for global data flow analysis
- Dual theorem exists for 1 element and maximal fixed point for k such that $f^k(1) = f^{k+1}(1)$.

How to Apply this to Data Flow Analysis?

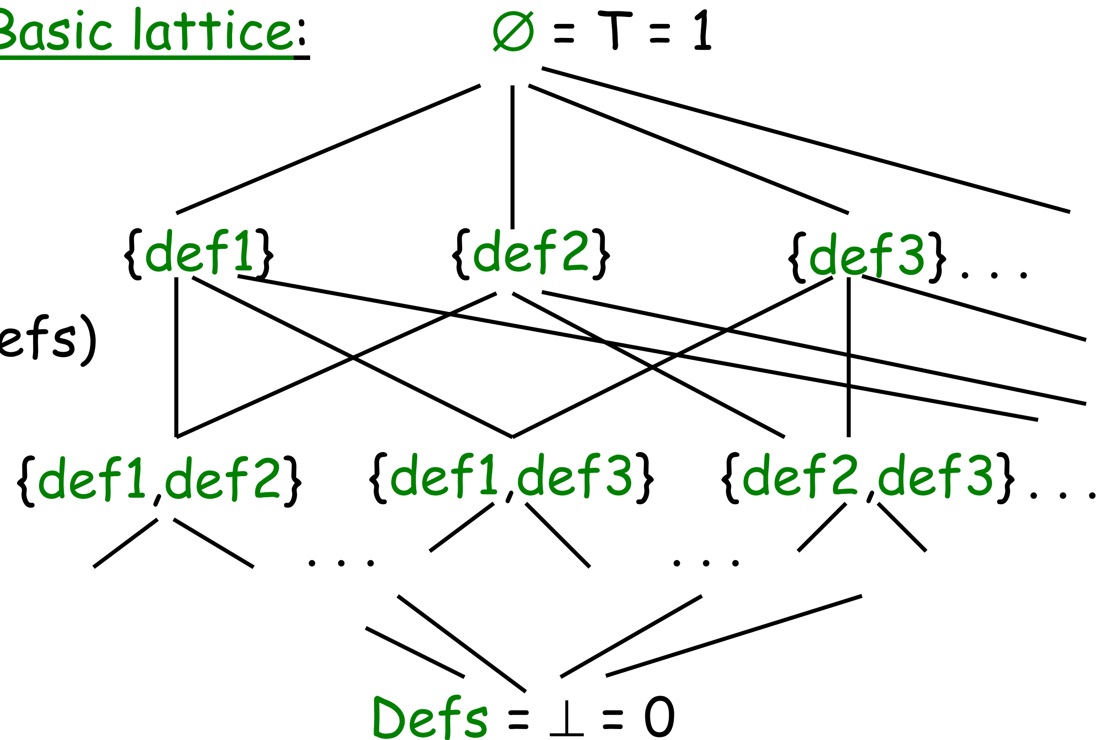
- Cartesian cross product of posets is a poset
  - $(S, \leq)$ poset $\Rightarrow$ $(S \times S \times \ldots \times S, \leq')$ is a poset whose partial ordering is component-wise $\leq$.
    $W \leq' Y$ iff $W1 \leq Y1$ and $W2 \leq Y2$, ... and $Wn \leq Yn$
- Cartesian cross product of lattices is a lattice (with induced partial ordering)

$$\bigtimes_{i=1}^{n} L_i = L$$

- Monotone function on cross product lattice
  $F(Y1, Y2, \ldots, Yn) = (g_1(Y1, Y2, \ldots, Yn), \ldots, g_n(Y1, Y2, \ldots, Yn))$
  where $g_I : (S, S, \ldots, S) \dashrightarrow S$ and $\leq'$ is component-wise $\leq$ on L

Reaching Definitions Example

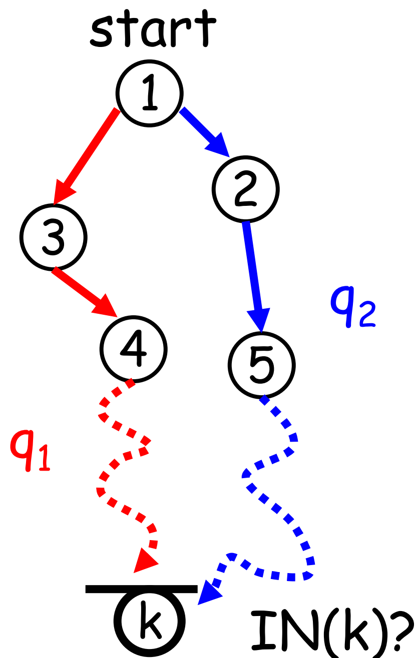
- Defs = {<node,var>}, all defs in program
- Basis lattice is 2^{Defs}, Cartesian product lattice = (2^{Defs} , 2^{Defs} ..., 2^{Defs})
- Partial order on product is $\leq' = \supseteq$ component-wise superset
- 1 element < ∅, ∅ ..., ∅ >
- 0 element (Defs, Defs, ..., Defs)
- Solution: MFP



Monotone Data Flow Frameworks

- Formalism for expressing and categorizing data flow problems (Kildall, POPL'73) <G, L, F, M>
- G: control flow graph <N, E>
- L: (semi-) lattice with meet $\wedge$
  - $a \wedge b = a \Leftrightarrow a \leq b$
  - usually assume L has a 0 and 1 element
  - finite (descending) chains
- F: function space; $\forall f \in F$, $f: L \dashrightarrow L$, f monotone
  - Contains identity function
  - Closed under composition $\forall f,g \in F$, $f \circ g \in F$
  - Closed under pointwise meet, if $h(x) = f(x) \wedge g(x)$, then $h \in F$
- M : N --> F, maps a node to corresponding transfer function that describes data flow effect of traversing that node

Meet Over All Paths Solution (MOP)



- Maximal data flow information desired is obtained by traversing ALL PATH from start to node k
- q_x is a path (start, n_1, n_2, … , n_{i-1}), and n_{i-1} is predecessor or k
- $f[q_x] = M[n_{i-1}] \circ \ldots \circ M[n_2] \circ M[n_1] \circ M[start](init)$
- Let Q be the set of all path from start to node k:

$$IN(k) = \bigwedge_{q \in Q} f[q]$$

This is MOP, and we cannot do better than this at compile time. What is the relation between MOP and MFP ?

Data Flow Function Properties

Monotonicity

- Defined as (1) $x \le y \Rightarrow f(x) \le f(y)$
  Smaller input cannot result in larger output

- Equivalent formulation of definition:
  (2) $f(x \wedge y) \le f(x) \wedge f(y)$ PROOF is coming up!

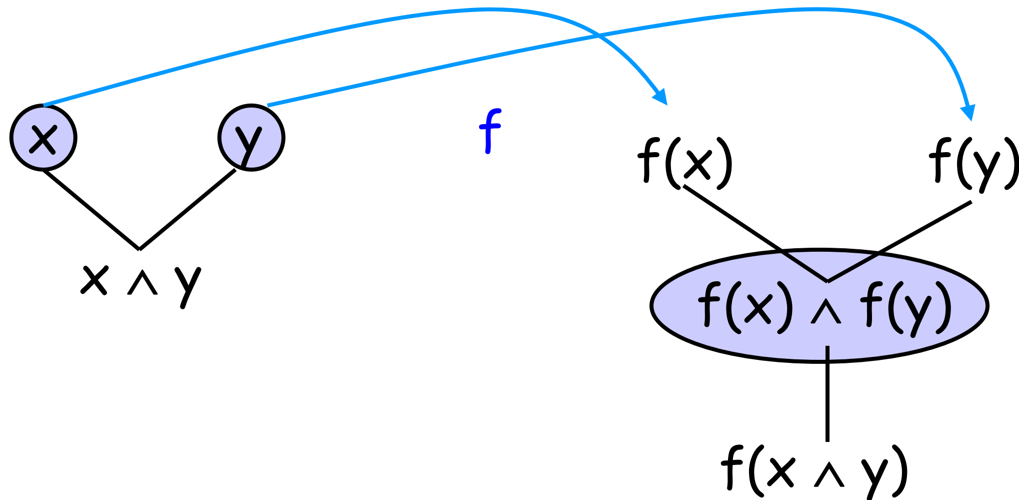
Distributivity

- f is distributive if
  $f(x \wedge y) = f(x) \wedge f(y)$

- Distributivity implies monotonicity

- Our four classical data flow problems are distributive

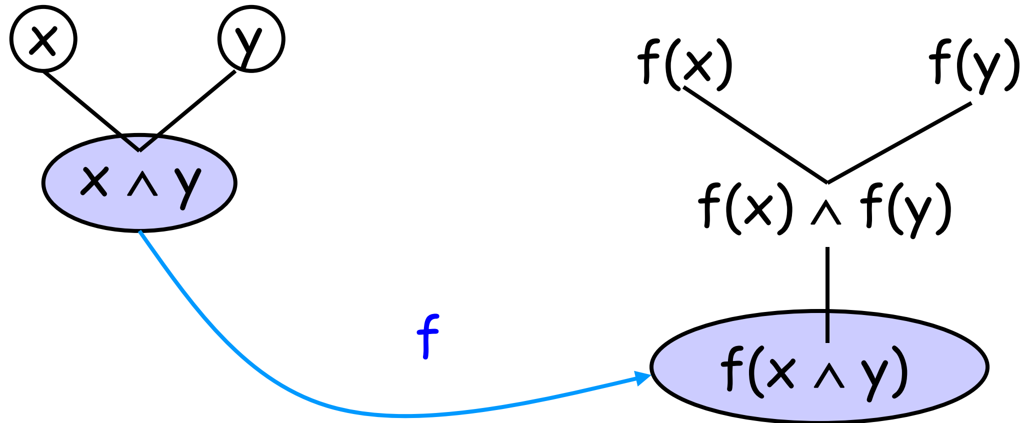
Monotone Functions



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

If merge inputs, then apply f, then result is "smaller or equal" to applying f individually, and merging result.

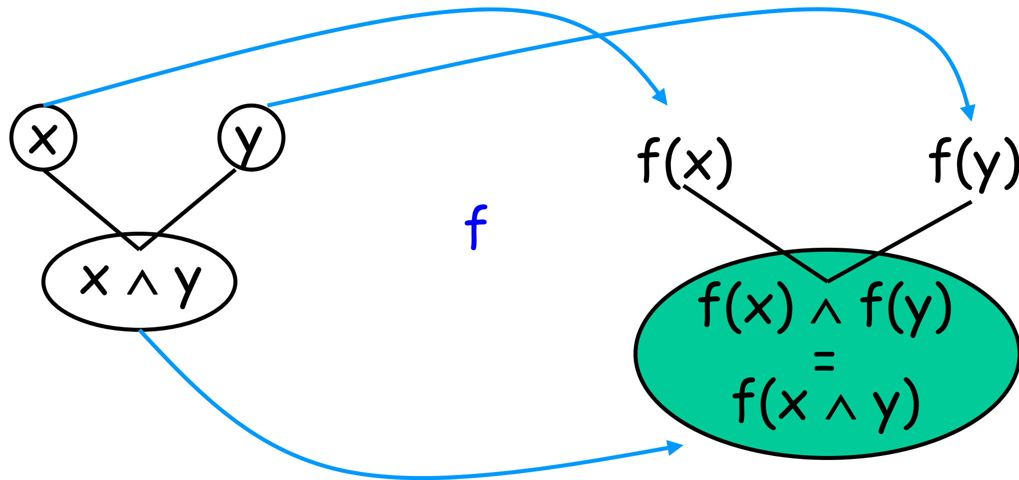
Monotone Functions



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

If merge inputs, then apply f, then result is "smaller or equal" to applying f individually, and merging result.

Monotone and Distributive Functions



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

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

If merge inputs, then apply f, then result is "equal"
to applying f individually, and merging result. "MOP-like" solution!

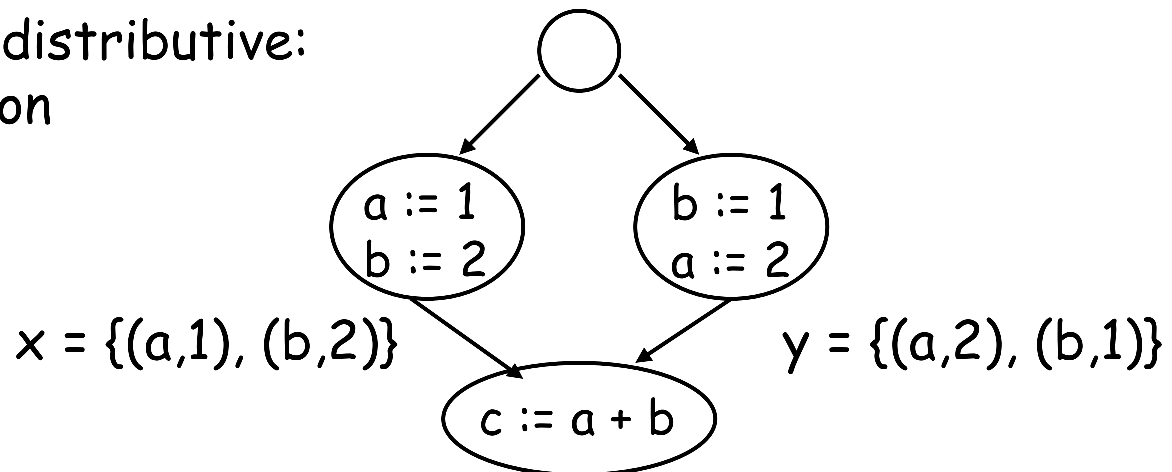
Monotone and Distributive Functions

Distributivity means that you can apply meets in the domain, and then apply f

OR

you can apply f and then take the meets in the range, and in both cases calculate the same answer.

Monotone, but not distributive: constant propagation



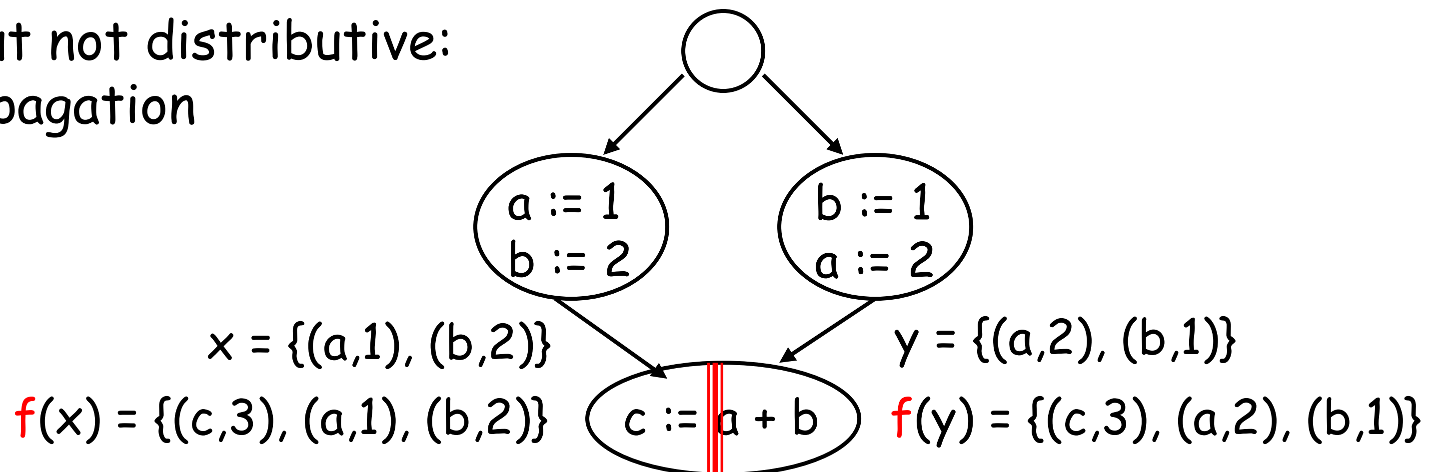
Monotone and Distributive Functions

Distributivity means that you can apply meets in the domain, and then apply f

OR

you can apply f and then take the meets in the range, and in both cases calculate the same answer.

Monotone, but not distributive: constant propagation



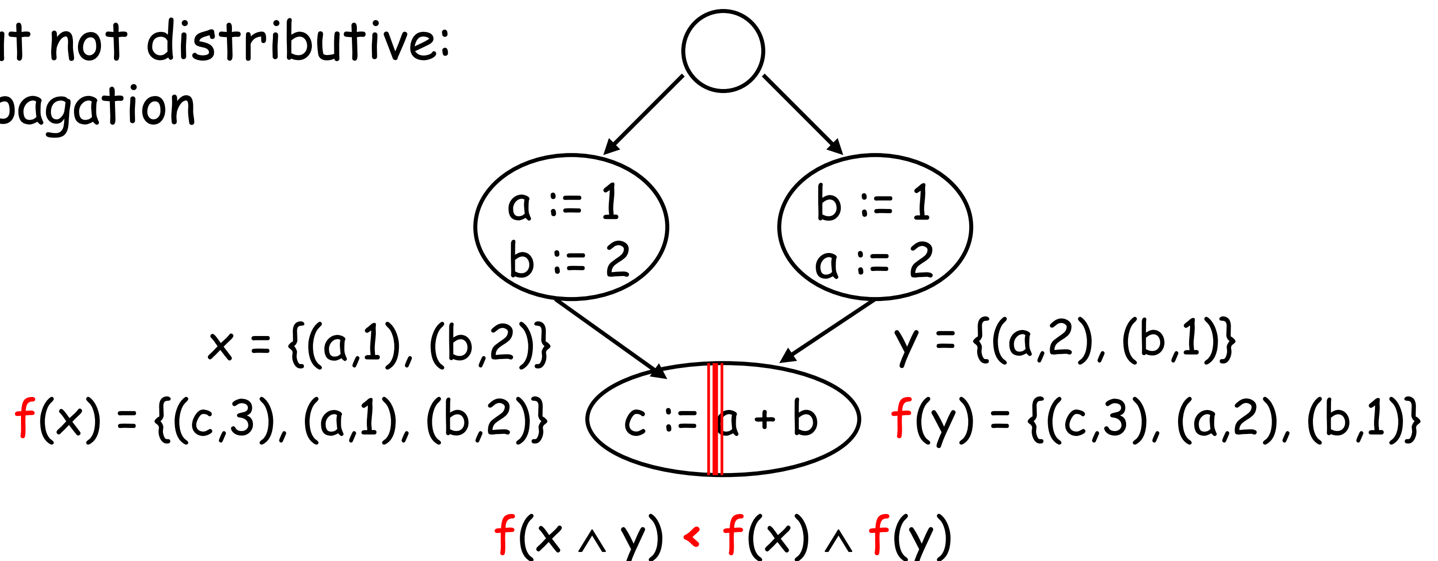
Monotone and Distributive Functions

Distributivity means that you can apply meets in the domain, and then apply f

OR

you can apply f and then take the meets in the range, and in both cases calculate the same answer.

Monotone, but not distributive: constant propagation



Data Flow Function Properties

Theorem

$$(1)\ x \le y \Rightarrow f(x) \le f(y)$$

iff

$$(2)\ f(x \wedge y) \le f(x) \wedge f(y)$$

Proof $(1) \Rightarrow (2)$:

$(x \wedge y) \wedge x = x \wedge y$	$\Rightarrow\ x \wedge y \le x$	Def. of $\wedge$ and $\le$
$(x \wedge y) \wedge y = x \wedge y$	$\Rightarrow\ x \wedge y \le y$	Def. of $\wedge$ and $\le$
$x \wedge y \le x \Rightarrow f(x \wedge y) \le f(x)$	$\Rightarrow \boxed{f(x \wedge y)} \wedge f(x) = f(x \wedge y)$	Mono. & Def. $\wedge$, $\le$
$x \wedge y \le y \Rightarrow f(x \wedge y) \le f(y)$	$\Rightarrow f(x \wedge y) \wedge f(y) = \boxed{f(x \wedge y)}$	Mono. & Def. $\wedge$, $\le$
$\boxed{(f(x \wedge y) \wedge f(y))} \wedge f(x) = f(x \wedge y)$		Substitution
$f(x \wedge y) \wedge (f(y) \wedge f(x)) = f(x \wedge y)$	$\Rightarrow f(x \wedge y) \le f(x) \wedge f(y)$	Associativity of $\wedge$

Data Flow Function Properties

Theorem

$$(1)\ x \le y \Rightarrow f(x) \le f(y)$$

iff

$$(2)\ f(x \wedge y) \le f(x) \wedge f(y)$$

Proof $(2) \Rightarrow (1)$:

Assume $x \le y$:

$x \le y \Rightarrow x \wedge y = x \ \Rightarrow f(x \wedge y) = f(x)$ — Def. of $\wedge$ and $\le$

$f(x \wedge y) = \boxed{f(x) \le f(x) \wedge f(y)}$ — Assumption (2)

$\boxed{f(x) \wedge (f(x) \wedge f(y)) = f(x)} \Rightarrow f(x) \wedge f(y) = f(x) \Rightarrow f(x) \le f(y)$ — Assoc. $\wedge$, Def. $\le$

Reaching Definitions Example

Show that the RD problem is distributive

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

$f(x) = GEN \cup (x - KILL)$

$f(x \cup y) =$

Quality Solution and Complexity of Iterative Algorithm

- On monotone data flow frameworks, iterative algorithms converge to MFP of data flow equations
- MFP $\leq$ MOP always
- There is a monotone problem that is not distributive for which MOP $\neq$ MFP (e.g.: constant propagation)
- Monotonicity and distributivity guarantee the existence of a fixed point solution, and affect its precision; does not make statements about speed of convergence of iterative algorithm

Worklist Iterative Algorithm (Forward Problem)

```
for i := 1, n  do  initialize IN(Bi) with T;
W := {1, … , n} // every CFG node is initially in worklist W

while ( W ≠ Ø ) {
   remove j from W;

   new :=    ⊗      ( GEN(Bi) ∪ [ IN(Bi) - KILL(Bi) ] )
         Bi ε PRED(Bj)
  if (new ≠ IN(Bj) {
     IN(Bj) := new;
     for k ε SUCC(Bj) do
        add k to W if not already there }
}
```

Complexity of Worklist Iterative Algorithm (Forward)

Assumptions:

- Lattice L represents data flow information for single node
- $|PRED(n)| \leq p$
- complexity in terms of bit-vector operations
- for each node, descending chain in L has length $\leq c$

- each node can be at most c times in the worklist
- processing a node once requires at most p propagation functions to be evaluated
- each single application of a propagation function requires 3 bit-vector operations for ($\vee$, -, $\otimes$)

$$O(3\ p\ c\ N) = O(c\ N)$$

Round Robin Iterative Algorithm (Forward Problem)

```
for j := 1, n  do  initialize IN(Bj) with T;
change := true;

while ( change ) {
   change := false
   for (j:= 2 to n in rPOSTORDER ) {

      new := ⊗          ( GEN(Bi) ∪ [ IN(Bi) - KILL(Bi) ] )
           Bi ε PRED(Bj)

      if (new ≠ IN(Bj) {
        IN(Bj) := new; change := true }
   }
}
```

Complexity of Round Robin Iterative Algorithm (Forward)

Assumptions:

- Lattice L represents data flow information for single node
- E = O(N) for Control Flow Graphs that occur in practice
- complexity in terms of bit-vector operations
- for each node, descending chain in L has length $\leq$ c

- in worst case, $c * |N|$ rPOSTORDER iterations
- processing all nodes during a single rPOSTORDER iterations requires evaluation of O(E) propagation functions to be evaluated
- each single application of a propagation function requires 3 bit-vector operations for ($\vee$, -, $\otimes$)
- computation of rPOSTORDER is O(E) = O(N) in CFGs!

$$O(3\ c\ N^2) = O(c\ N^2)$$

Complexity of Round Robin Iterative Algorithm (Forward)

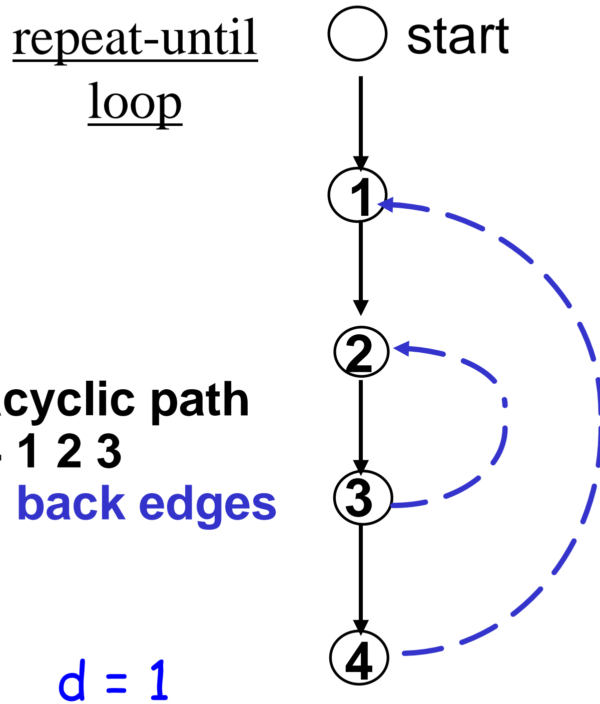
Why is the round robin algorithm interesting?

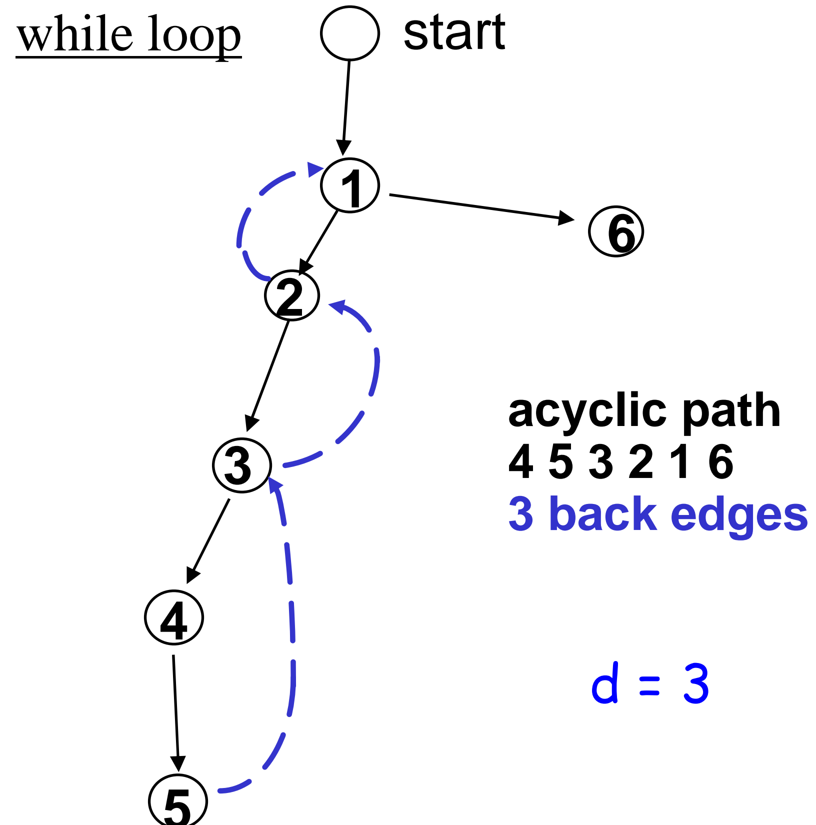
- For subclass of data flow problems, called rapid, a MFP solution can be computed in (d+2) iterations, where d is the loop connectedness.
- Assuming E = O(N), and a bit-vector problem, the complexity is O(c d N), which in practice is O(N) bit-vector operations (same as worklist version).
- Round robin is easier to code than worklist; very efficient in practice.
- Question: How to avoid explicit initialization with T ?

Loop Connectedness

Loop connectedness of a (reducible) flowgraph: largest number of back edges on any cycle-free path (d);

d ≤ maximal loop nesting depth in a program.





Rapid Data Flow Problems

Kam-Ullman rapid: "Global DFA and Iterative Algorithms", JACM, Jan 1976.

rapid: $\forall f \in F, \forall x \in L : f(x) \geq x \wedge f(T)$

Contribution of a cycle is independent of value at entry node; 1 pass through the cycle is enough

or

If you run once through the cycle, every contribution of this cycle to the overall solution has been computed

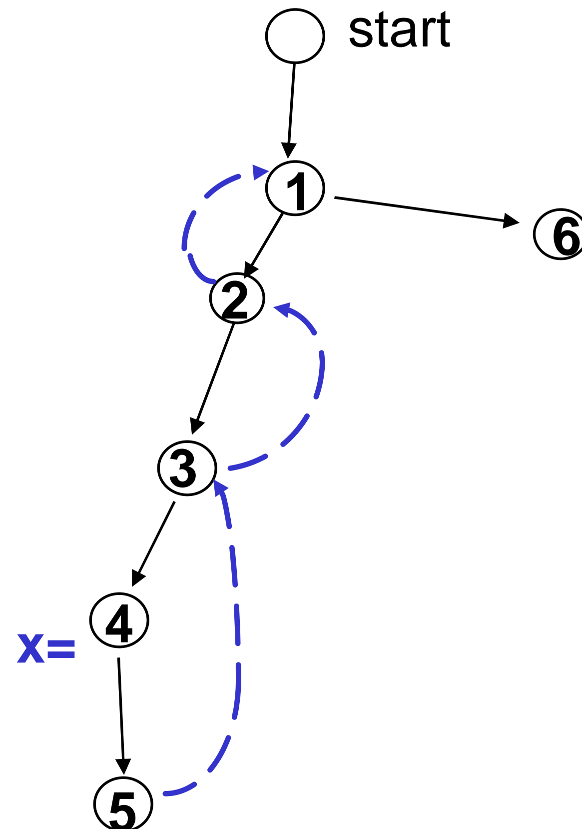
Our four bit-vector problems are rapid, but constant propagation is not.

Rapid Data Flow Problems

- Propagation of data flow information over every acyclic path in the CFG is enough to reach the fix point

- Round robin algorithm executes at most (d+2) rPOSTORDER iterations on a (reducible) flowgraph

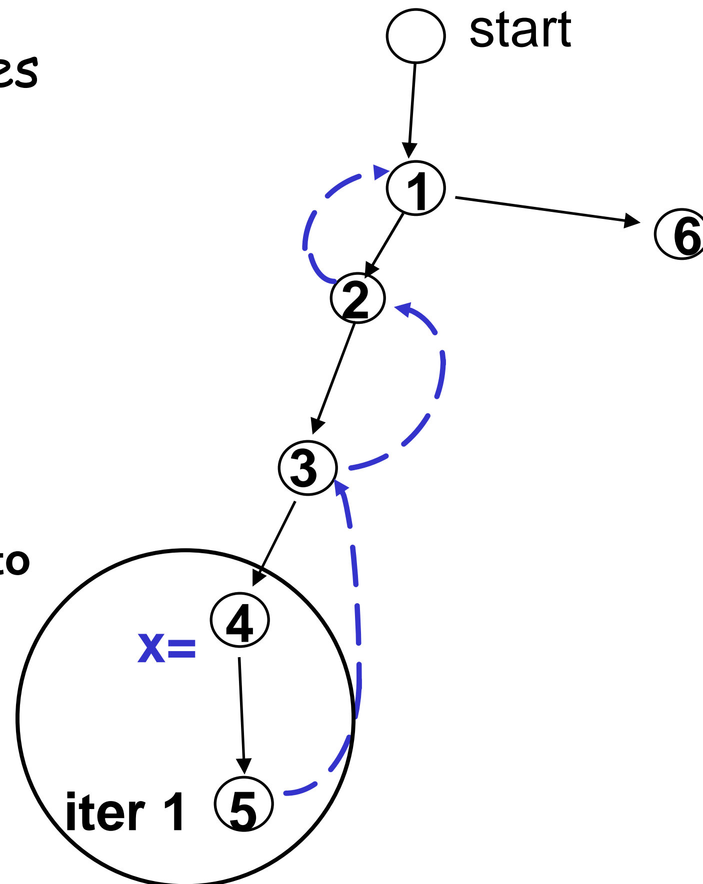
Iter 1 pushed defn as deep into CFG as possible; each subsequent iteration "lifts" the information up across a back edge level;
it takes 4 iterations for defn < 4:x> to reach nodes 1 and 6, and the last iteration is needed to show no change.



Rapid Data Flow Problems

- Propagation of data flow information over every acyclic path in the CFG is enough to reach the fix point
- Round robin algorithm executes at most (d+2) rPOSTORDER iterations on a (reducible) flowgraph

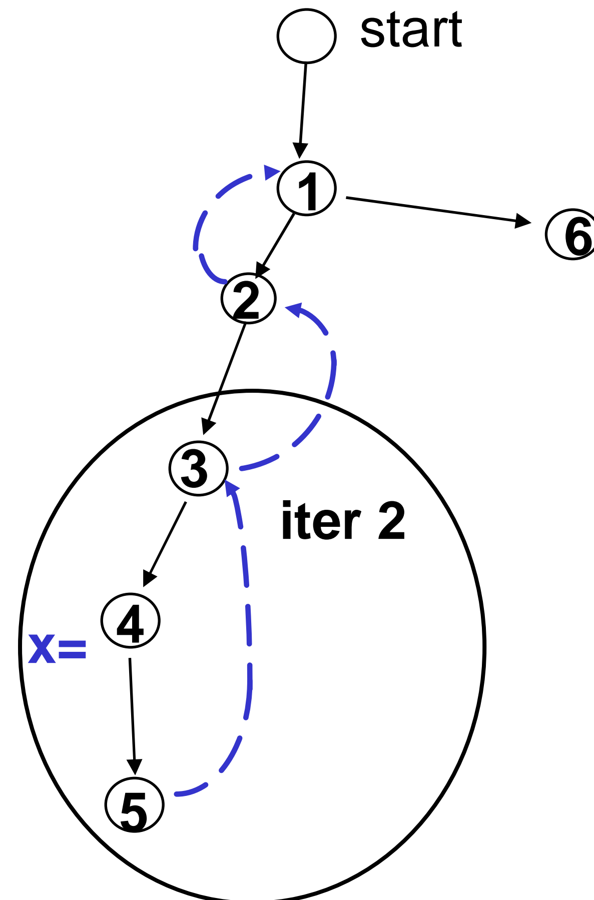
Iter 1 pushed defn as deep into CFG as possible; each subsequent iteration "lifts" the information up across a back edge level; it takes 4 iterations for defn < 4:x> to reach nodes 1 and 6, and the last iteration is needed to show no change.



Rapid Data Flow Problems

- Propagation of data flow information over every acyclic path in the CFG is enough to reach the fix point
- Round robin algorithm executes at most (d+2) rPOSTORDER iterations on a (reducible) flowgraph

Iter 1 pushed defn as deep into CFG as possible; each subsequent iteration “lifts” the information up across a back edge level; it takes 4 iterations for defn < 4:x> to reach nodes 1 and 6, and the last iteration is needed to show no change.

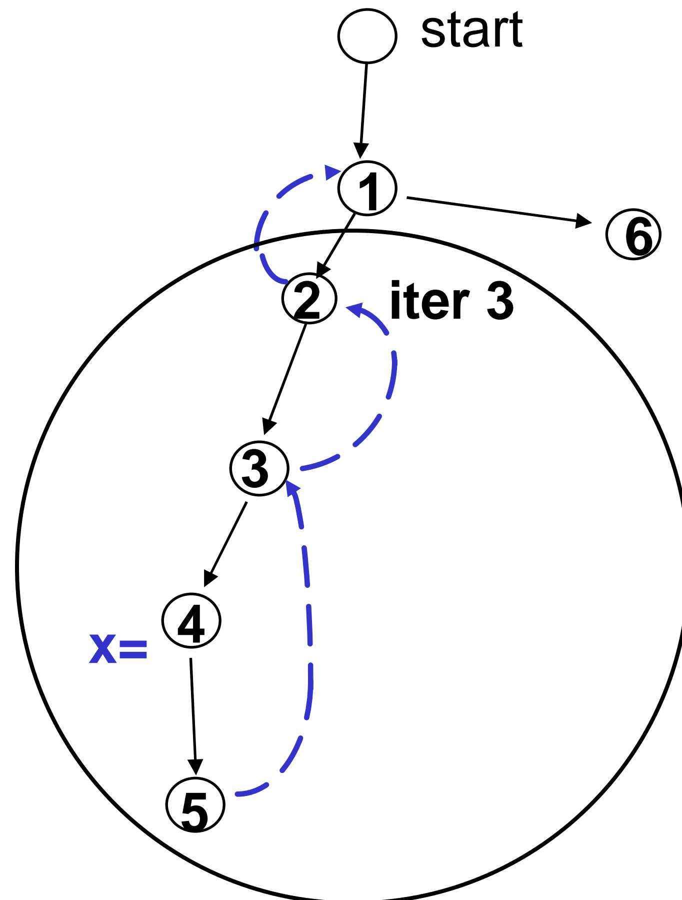


Rapid Data Flow Problems

- Propagation of data flow information over every acyclic path in the CFG is enough to reach the fix point

- Round robin algorithm executes at most (d+2) rPOSTORDER iterations on a (reducible) flowgraph

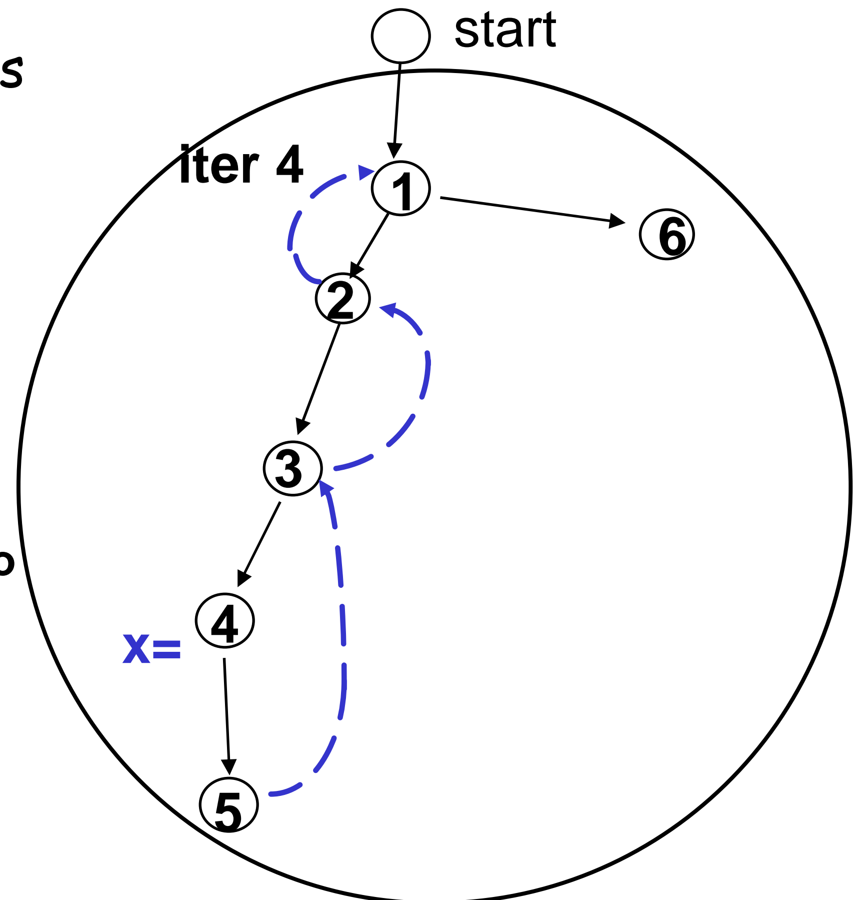
Iter 1 pushed defn as deep into CFG as possible; each subsequent iteration “lifts” the information up across a back edge level; it takes 4 iterations for defn < 4:x> to reach nodes 1 and 6, and the last iteration is needed to show no change.



Rapid Data Flow Problems

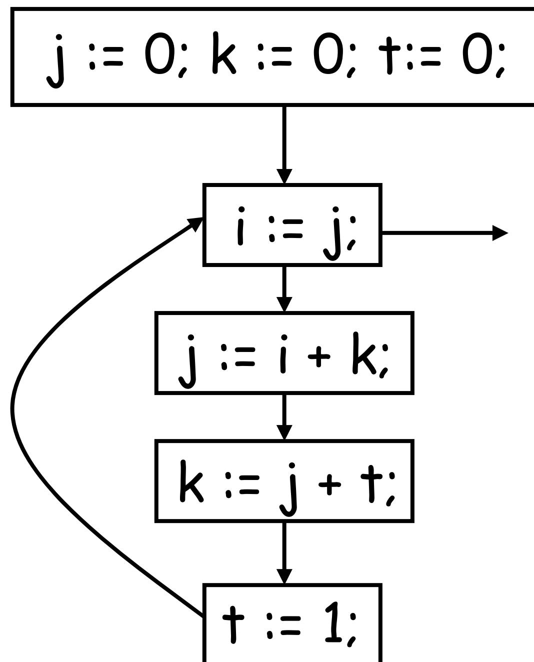
- Propagation of data flow information over every acyclic path in the CFG is enough to reach the fix point
- Round robin algorithm executes at most (d+2) rPOSTORDER iterations on a (reducible) flowgraph

Iter 1 pushed defn as deep into CFG as possible; each subsequent iteration "lifts" the information up across a back edge level; it takes 4 iterations for defn < 4:x> to reach nodes 1 and 6, and the last iteration is needed to show no change.



Fast Data Flow Problems

Not rapid: constant propagation



fast:

$\forall f \in F, \forall x \in L : f^2(x) \geq x \wedge f(x)$

1 pass around a cycle is enough to summarize its contribution
($\Rightarrow$ structural algorithms)

Generalization: k-bounded

$f^k(x) \geq x \wedge f(x) \wedge f^2(x) \ldots f^{k-1}(x)$

fast = 2-bounded

Node-list Iterative Algorithm (Forward Problem)

Observation: For rapid problems, propagating data flow information along all acyclic path is sufficient.

NODELIST: sequence of nodes in CFG such that all acyclic paths are subsequences of NODELIST (Ken Kennedy, 1975). Complexity: O(length(NODELIST))

for j := 1, n do initialize IN(Bj) with T;

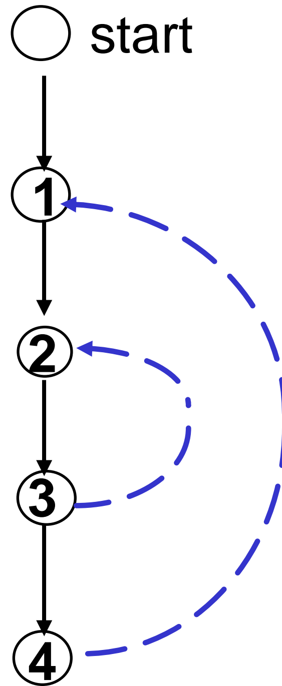
foreach j in NODELIST do {

$$IN(Bj) := \bigotimes_{Bi \, \varepsilon \, PRED(Bj)} (GEN(Bi) \cup [IN(Bi) - KILL(Bi)])$$

}

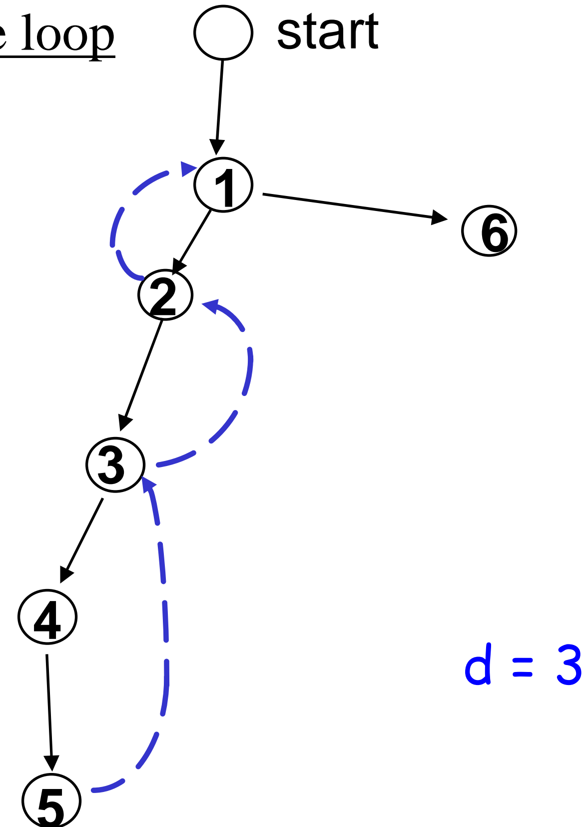
Node-list Iterative Algorithm (Forward Problem)

repeat-until loop



d = 1

while loop

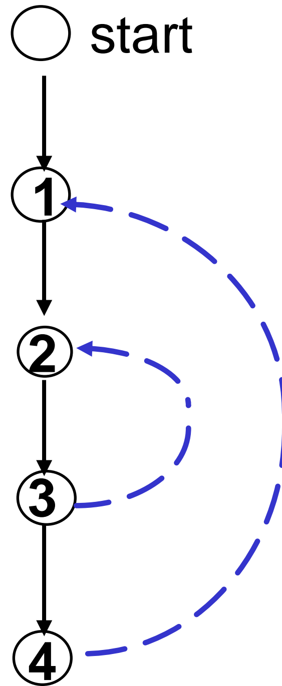


d = 3

example NODELIST:

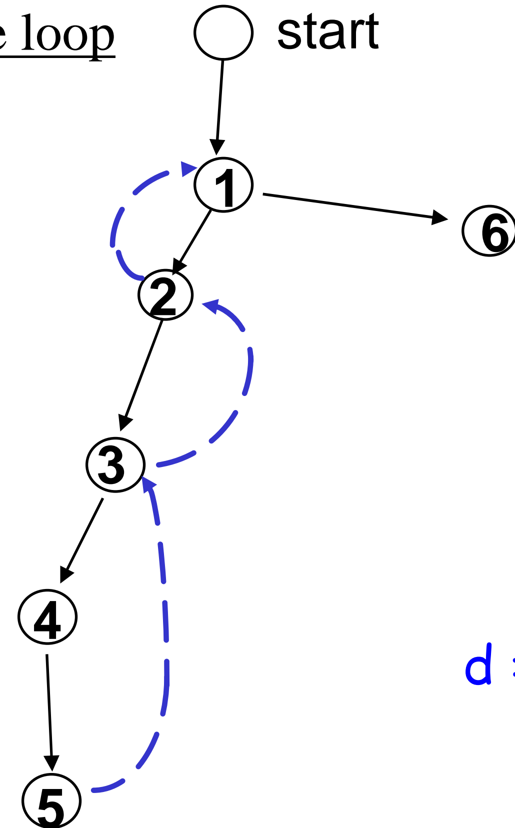
Node-list Iterative Algorithm (Forward Problem)

repeat-until loop



d = 1

while loop



d = 3

example

NODELIST: 1 2 3 4 1 2 3

1 2 3 4 5 3 2 1 6 2 3 4

Optimizations

How to use DU/UD chains for the following optimizations?

- dead code elimination
- constant propagation
- code motion (challenge problem ?)

Terminology:

UD(S, v): set of statements that contain definitions for variable v that reach the use of v in statement S

DU(S, v): set of statements that contain used of variable v that are reached by the definition of v in statement S

“Limits” of Optimizations

Correctness:

- statement reordering can change “stability” of numerical algorithm due to round-off errors (e.g.: + and * are not really associative due to finite precision of floating point numbers)

- some languages require exception occurrences and values of objects to be preserved across optimizations (e.g.: Java’s exception model)

Debugging:

- Breakpoints in the unoptimized program need to be related to a breakpoint in the optimized program

- What about values of dead variables?

Dead Code Elimination

Two flavors of dead code: A statement can be considered dead code if

(1) it will never be executed, or

(2) it may be executed, but the result will never be used
(including side effects)

The discussed algorithm uses UD chains.

Dead Code Elimination

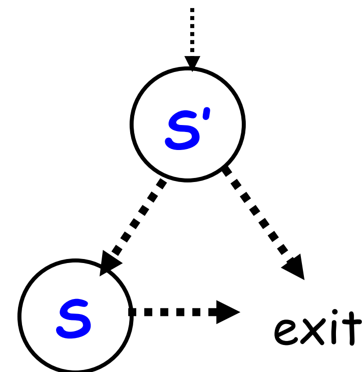
Terminology:

DEFS(S): set of variables that may be written in statement S.

USES(S): set of variables that may be read in statement S.

CONTROL(S): set of statements such that S is control dependent on these statements

S is control dependent on S': S' contains a control flow branch, and there are execution paths from S' to the program exit such that S is on one path, but not on the other.



Dead Code Elimination

```
Report all statements that are not on any path from the
entry to exit node as dead code, and remove them

Initialize all statement as not useful

Initialize worklist with critical statements (print statements)

While worklist is not empty Do

  Remove a statement from worklist, call it S; mark S useful

  For all S' in CONTROL(S): if S' is not marked useful,
  add to worklist

  Forall variables v in USES(S): if S' in UD(S,v) is not marked
  useful, add to worklist EndForall

EndWhile

Report all unmarked statements as dead code
```

Dead Code Elimination

Example

Does algorithm always terminate?

Constant Propagation

The discussed algorithm uses UD and DU chains

Constants are typically only considered for integer values, enumeration values, and boolean values.

Terminology:

immediate constant: direct assignment of a constant value, e.g.,
A := 1

(S, v, value): describes the fact that variable v has a constant value of “value” on entry to statement S.

Constant Propagation

```
Insert all immediate constants (S, v, const) into worklist;

While worklist Not empty do

    Remove entry (S, v, const) from list.

    Forall S' in DU(S,v)
        If
          ( [ Foreach w in USES(S')
              ⊗ (value of w defined at UD(S',w)) = c ] And
              u in DEFS(S') evaluates to c' )
        Then
           If (S', u, c') is a newly encountered constant,
           add (S', u, c') to worklist // generate new constant
        EndIf

    EndForall

EndWhile
```

This algorithm uses UD and DU chains

Is it pessimistic or optimistic?

Constant Propagation

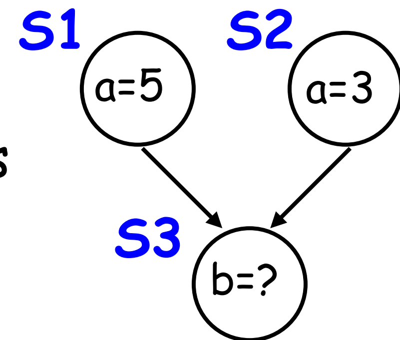
Example

Does algorithm always terminate?

Challenge Problem (homework):
- how to deal with assignments out of dead code regions?

```
    if ... then
S1    a := 5
    else
S2    a := 3
    endif

S3  b := a + 1
```

DU chains



Incremental Analysis

What to do if source code changes, for instance due to another transformation or a user edit?

GOAL: reduce compile-time cost of updating information

- low level: partitionable bitvector problems

- high level: update information as part of transformation, e.g., UD/DU chains after code motion

Cost and profitability of incremental updates will depend on type of change (e.g.: single statement edit such as variable redefinition, deletion of a statement, control flow change, ...)