

# CS 526

**A**dvanced

**C**ompiler

**C**onstruction

<http://misailo.cs.illinois.edu/courses/cs526>

# **DATAFLOW ANALYSIS**

The slides adapted from Saman Amarasinghe, Martin Rinard and Vikram Adve

# Why Dataflow Analysis?

Answers key questions about the **flow of values** and other program properties over control-flow paths **at compile-time**

# Why Dataflow Analysis?

## Compiler fundamentals

What defs. of  $x$  reach a given use of  $x$  (and vice-versa)?

What  $\{\langle \text{ptr}, \text{target} \rangle\}$  pairs are possible at each statement?

## Scalar dataflow optimizations

Are any uses reached by a particular definition of  $x$ ?

Has an expression been computed on all incoming paths?

What is the innermost loop level at which a variable is defined?

## Correctness and safety:

Is variable  $x$  defined on every path to a use of  $x$ ?

Is a pointer to a local variable live on exit from a procedure?

## Parallel program optimization

**Where is dataflow analysis used?**

**Everywhere**

# Where is dataflow analysis used?

## **Preliminary Analyses**

Pointer Analysis

Detecting uninitialized variables

Type inference

Strength Reduction for Induction

Variables

## **Static Computation Elimination**

Dead Code Elimination (DCE)

Constant Propagation

Copy Propagation

## **Redundancy Elimination**

Local Common Subexpression

Elimination (CSE)

Global Common Subexpression

Elimination (GCSE)

Loop-invariant Code Motion (LICM)

Partial Redundancy Elimination (PRE)

## **Code Generation**

Liveness analysis for register  
allocation

# Basic Term Review

**Point:** A location in a basic block just before or after some statement.

**Path:** A path from points  $p_1$  to  $p_n$  is a sequence of points  $p_1, p_2, \dots, p_n$  such that (intuitively) some execution can visit these points in order.

**Kill of a Definition:** A definition  $d$  of variable  $V$  is killed on a path if there is an unambiguous (re)definition of  $V$  on that path.

**Kill of an Expression:** An expression  $e$  is killed on a path if there is a possible definition of any of the variables of  $e$  on that path.

# Dataflow Analysis (Informally)

Symbolically simulate execution of program

- Forward (Reaching Definitions)
- Backward (Variable Liveness)

Stacked analyses and transformations that work together, e.g.

- Reaching Definitions → Constant Propagation
- Variable Liveness → Dead code elimination

Our plan:

- Examples first (analysis + theory)
- Theory follows



# Analysis: Reaching Definitions

A definition **d** reaches point **p** if there is a path from the point after **d** to **p** such that **d** is not killed along that path.

Example Statements:

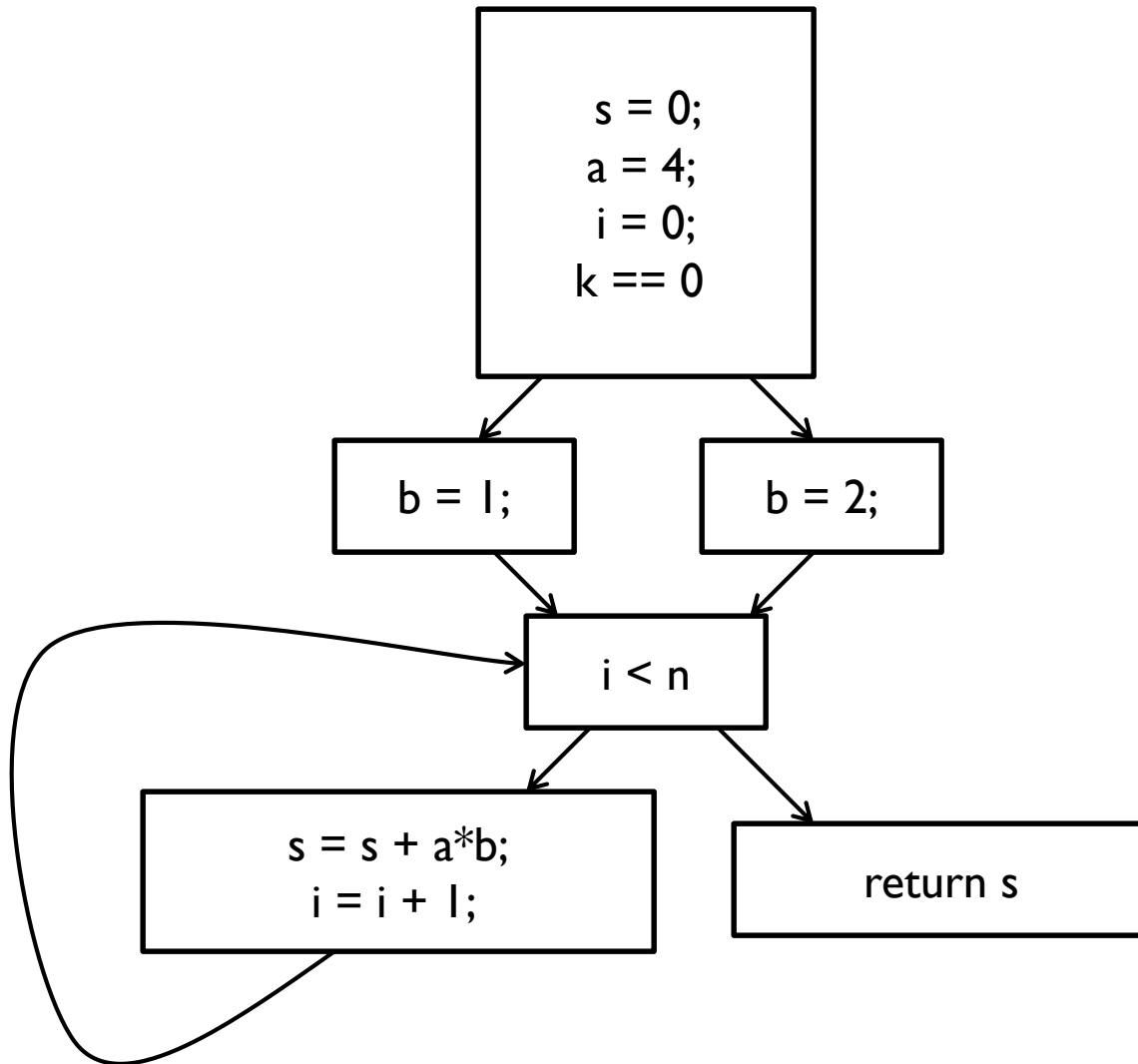
**a = x+y**

- It is a definition of a
- It is a use of x and y

**b = a+l**

- It is a definition of ? And use of ??

*A definition reaches a use if the value written by the definition may be read by the use*



# Reaching Definitions (Declarative)

## Dataflow variables (for each block B)

**In(B)**  $\equiv$  the set of defs that reach the point before first statement in B

**Out(B)**  $\equiv$  the set of defs that reach the point after last statement in B

**Gen(B)**  $\equiv$  the set of defs in B that are not killed in B.

**Kill(B)**  $\equiv$  the set of all defs that are killed in B (i.e., on the path from entry to exit of B, if def  $d \notin B$ ; or on the path from  $d$  to exit of B, if def  $d \in B$ ).

## *The difference:*

In(B), Out(B) are **global** dataflow properties (of the function).

Gen(B), Kill(B) are **local** properties of the basic block B alone.

# Computing Reaching Definitions

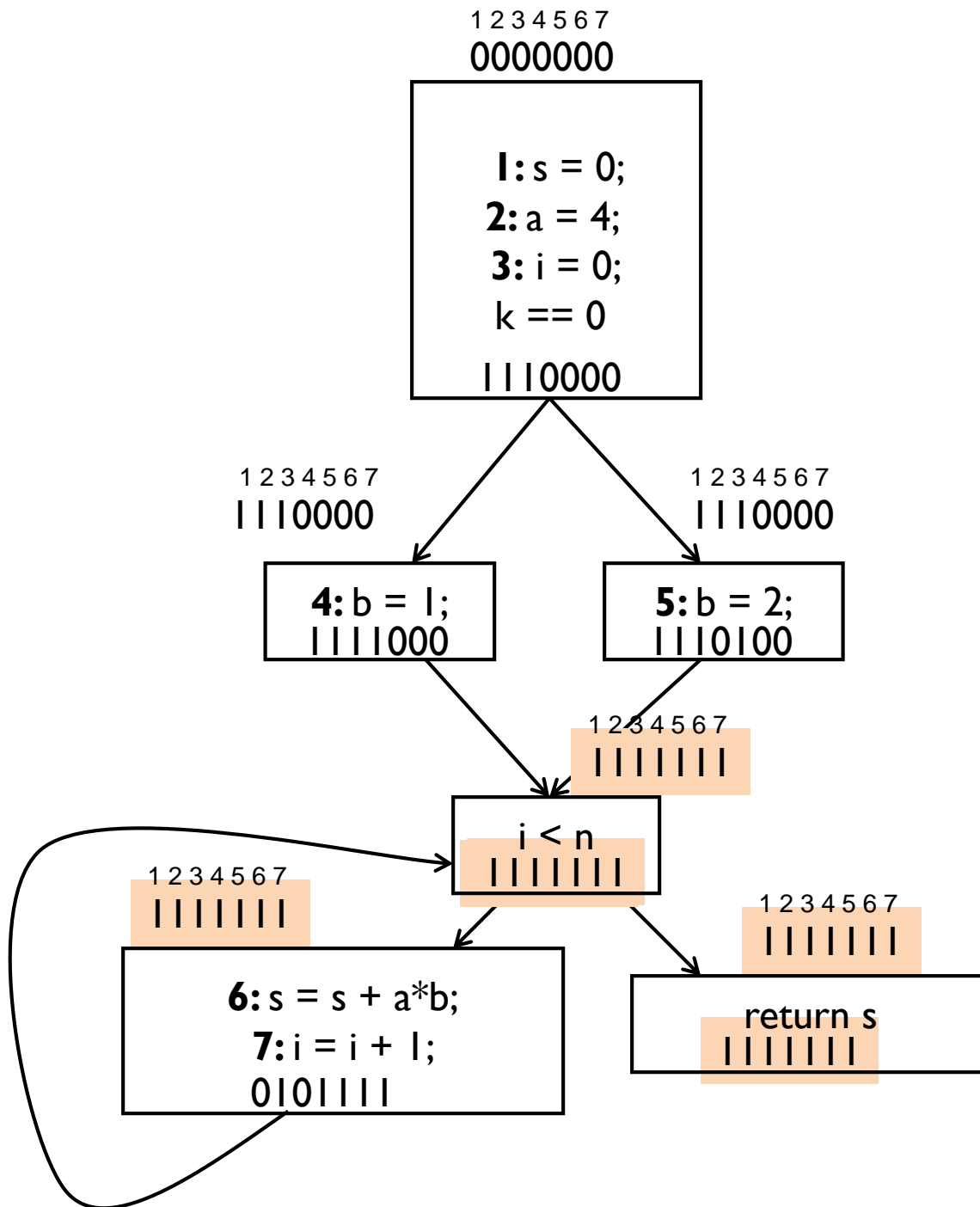
Compute with sets of definitions

- represent **sets** using **bit vectors** data structure
- each definition has a position in bit vector

At each basic block, compute

- definitions that reach the start of block
- definitions that reach the end of block

Perform computation by simulating execution of program until reach fixed point



# Formalizing the analysis: Dataflow Equations

IN and OUT combine the properties from the neighboring blocks in CFG

$$\text{IN}[b] = \text{OUT}[b_1] \cup \dots \cup \text{OUT}[b_n]$$

- where  $b_1, \dots, b_n$  are predecessors of  $b$  in CFG

$$\text{OUT}[b] = (\text{IN}[b] - \text{KILL}[b]) \cup \text{GEN}[b]$$

$$\text{IN}[\text{entry}] = 0000000$$

**Result: system of equations**

# Solving Equations

Use fixed point (worklist) algorithm

Initialize with solution of  $OUT[b] = 0000000$

- **Repeatedly apply equations**
  1.  $IN[b] = OUT[b_1] \cup \dots \cup OUT[b_n]$
  2.  $OUT[b] = (IN[b] - KILL[b]) \cup GEN[b]$
- **Until reach fixed point\***

\* Fixed point = equation application has no further effect

Use a **worklist** to *track which equation applications may have a further effect*

# Reaching Definitions Algorithm

for all nodes  $n$  in  $N$

$OUT[n] = \text{emptyset};$  //  $OUT[n] = GEN[n];$

$IN[\text{Entry}] = \text{emptyset};$

$OUT[\text{Entry}] = GEN[\text{Entry}];$

$\text{Changed} = N - \{ \text{Entry} \};$  //  $N = \text{all nodes in graph}$

while ( $\text{Changed} \neq \text{emptyset}$ )

    choose a node  $n$  in  $\text{Changed};$

$\text{Changed} = \text{Changed} - \{ n \};$  // in efficient impl. these are bitvector operations

$IN[n] = \text{emptyset};$

    for all nodes  $p$  in  $\text{predecessors}(n)$

$IN[n] = IN[n] \cup OUT[p];$

$OUT[n] = GEN[n] \cup (IN[n] - KILL[n]);$

    if ( $OUT[n]$  changed)

        for all nodes  $s$  in  $\text{successors}(n)$

$\text{Changed} = \text{Changed} \cup \{ s \};$



# Reaching Definitions: Convergence

Out[B] is finite

Out[B] never decreases for any B

⇒ must eventually stop changing

At most n iterations if n blocks

⇐ Definitions need propagate only over acyclic paths

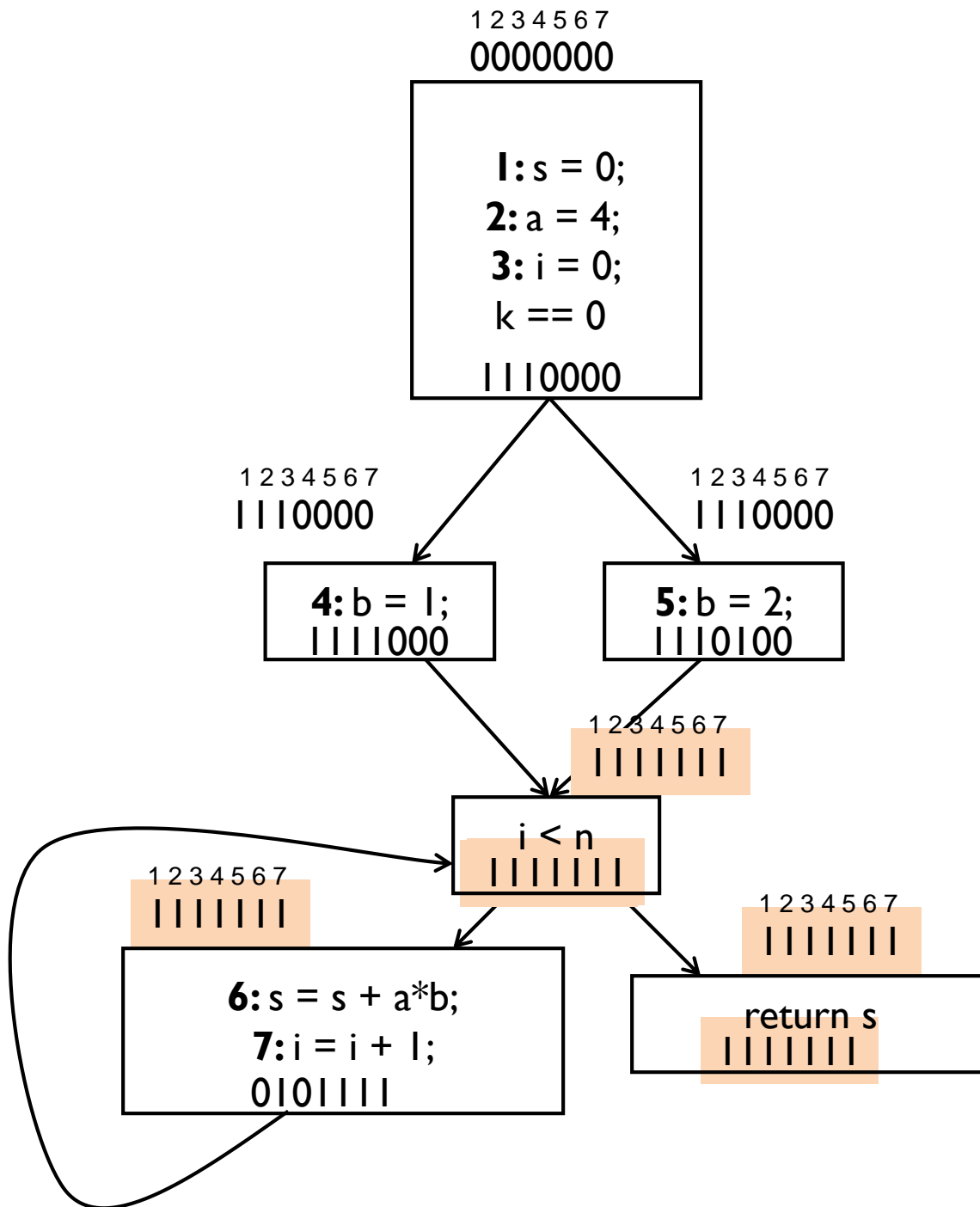
# Transform: Constant Propagation

Paired with reaching definitions (uses its results)

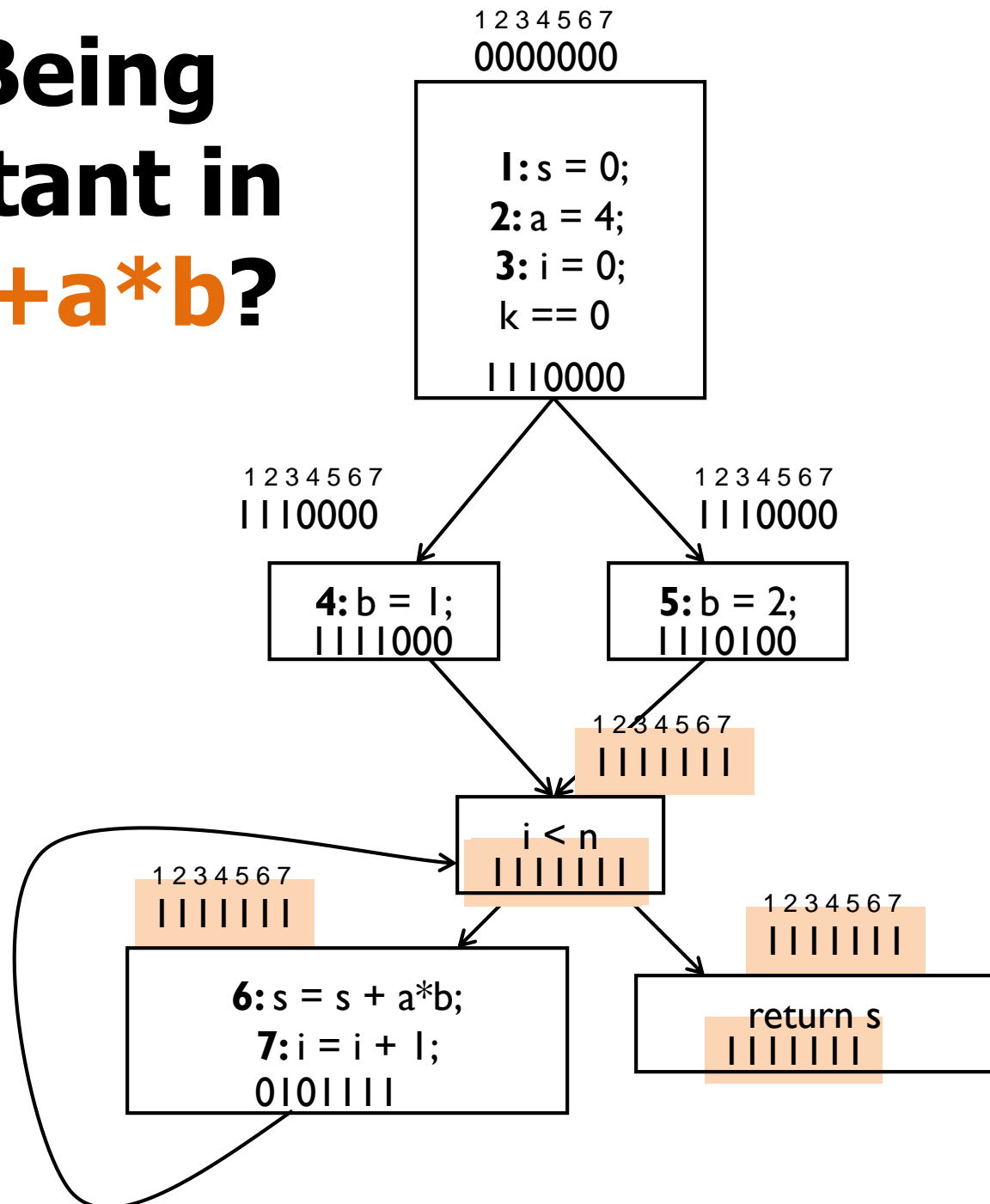
Check: Is a use of a variable a constant?

- Check all reaching definitions
- If all assign variable to same constant
- Then use is in fact a constant

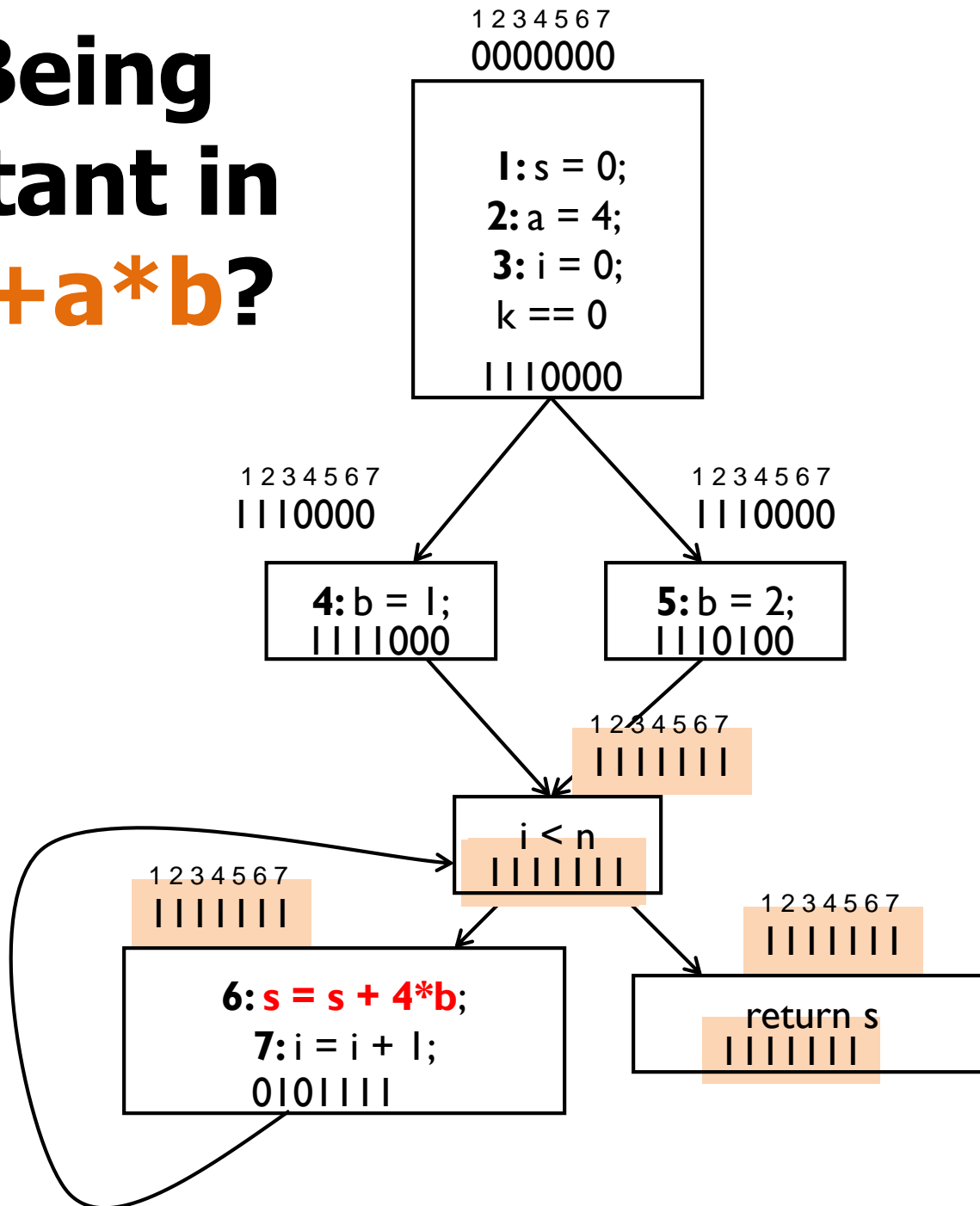
Can replace variable with constant



# Is **a** Being Constant in **s = s + a \* b**?



# Is **a** Being Constant in **s = s + a \* b**?



# Analysis: Available Expressions

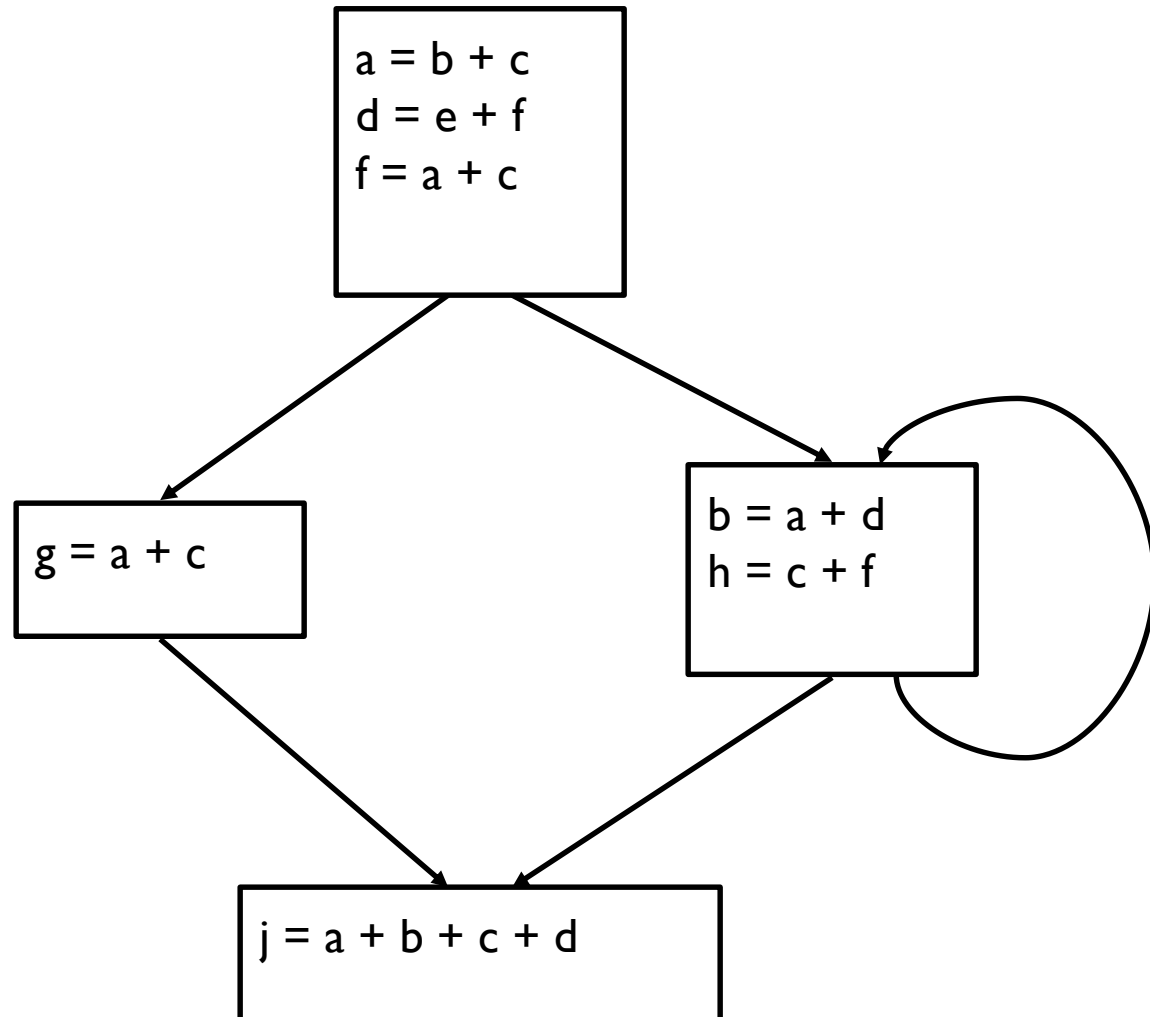
An expression  $x+y$  is available at a point  $p$  if

1. Every path from the initial node to  $p$  must evaluate  $x+y$  before reaching  $p$ ,
2. There are no assignments to  $x$  or  $y$  after the expression evaluation but before  $p$ .

Available Expression information can be used to do global (across basic blocks) CSE

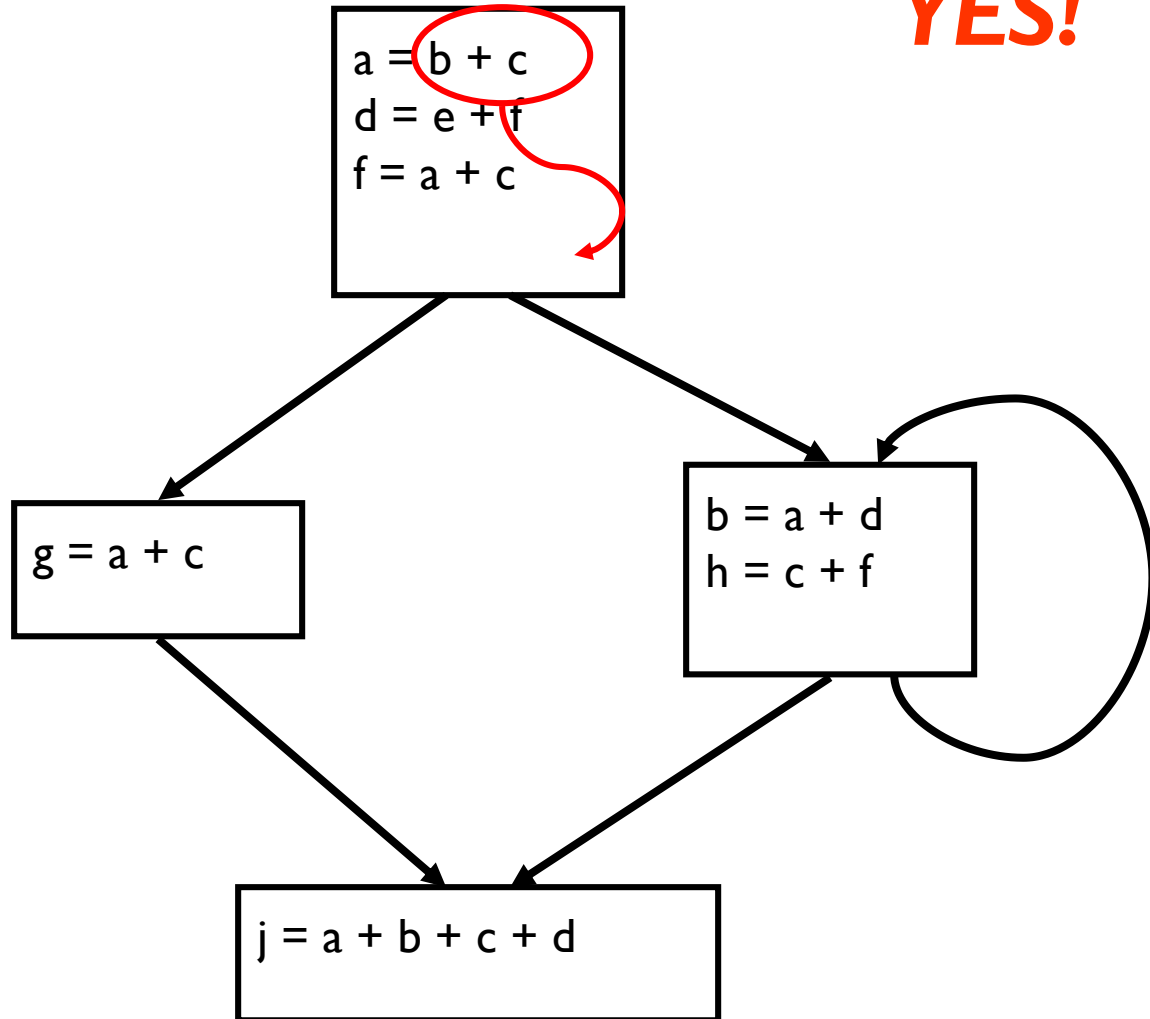
- If expression is available at use, no need to reevaluate it
- Beyond SSA-form analyses

# Example: Available Expression



# Is the Expression Available?

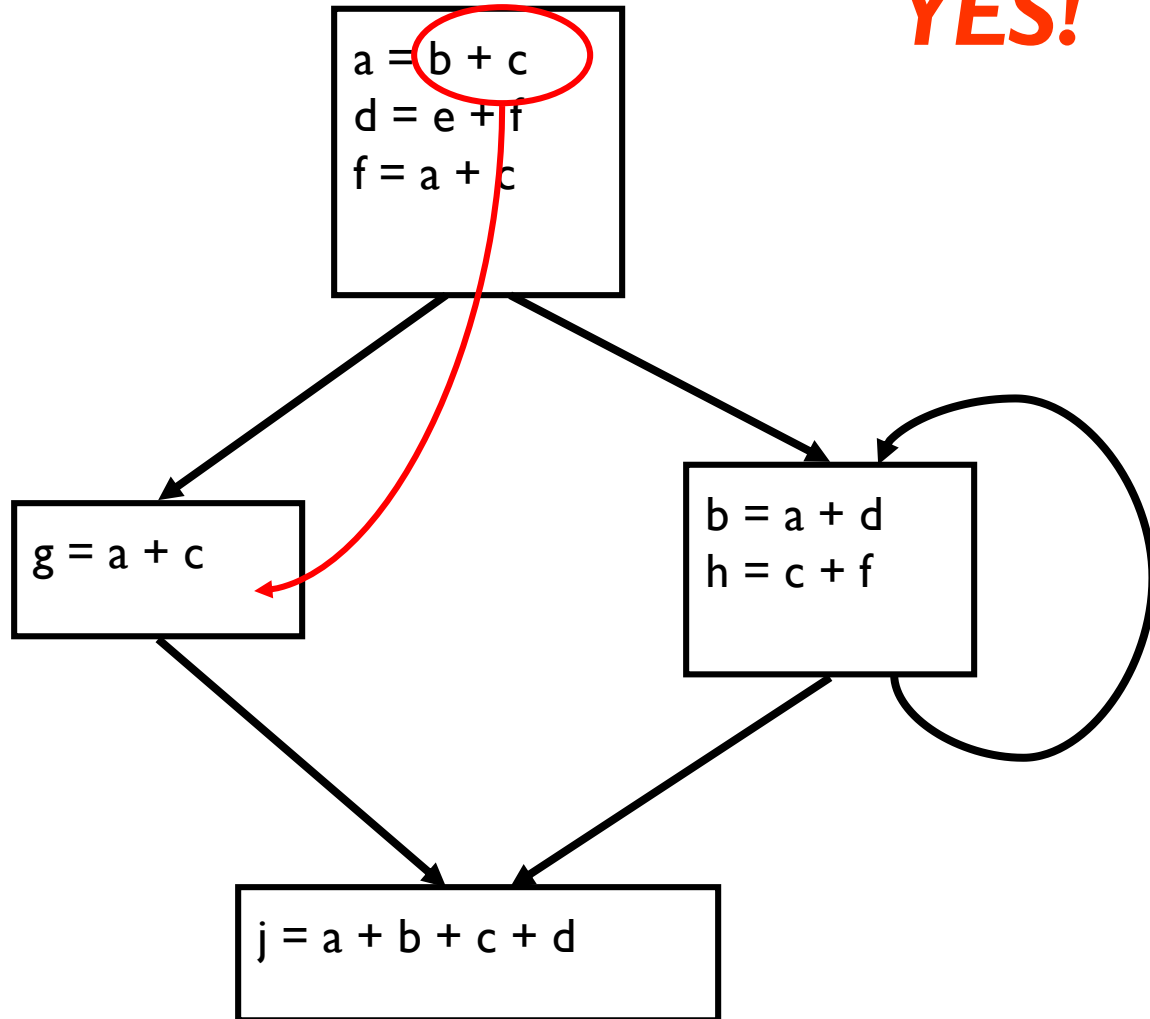
**YES!**





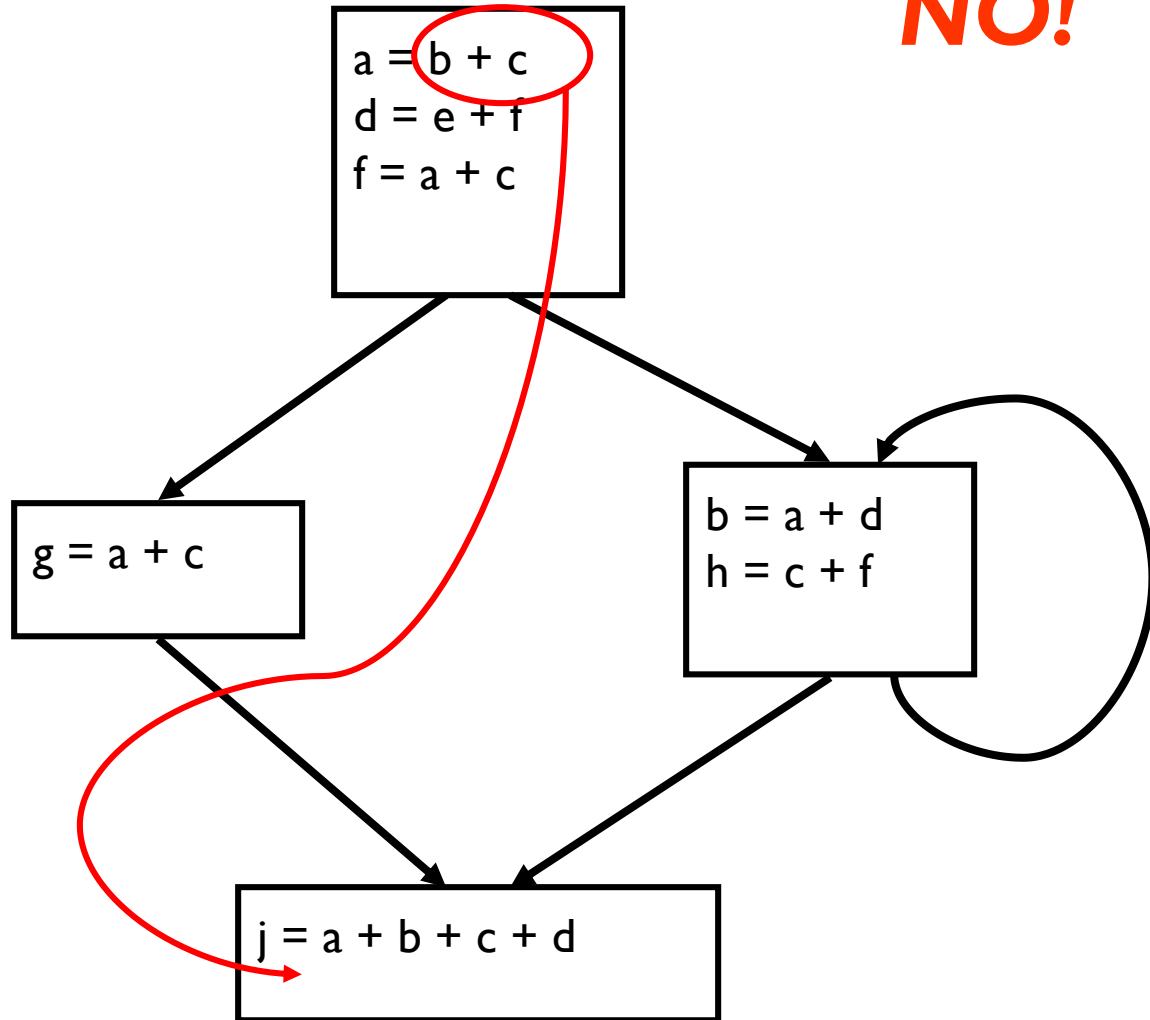
# Is the Expression Available?

**YES!**



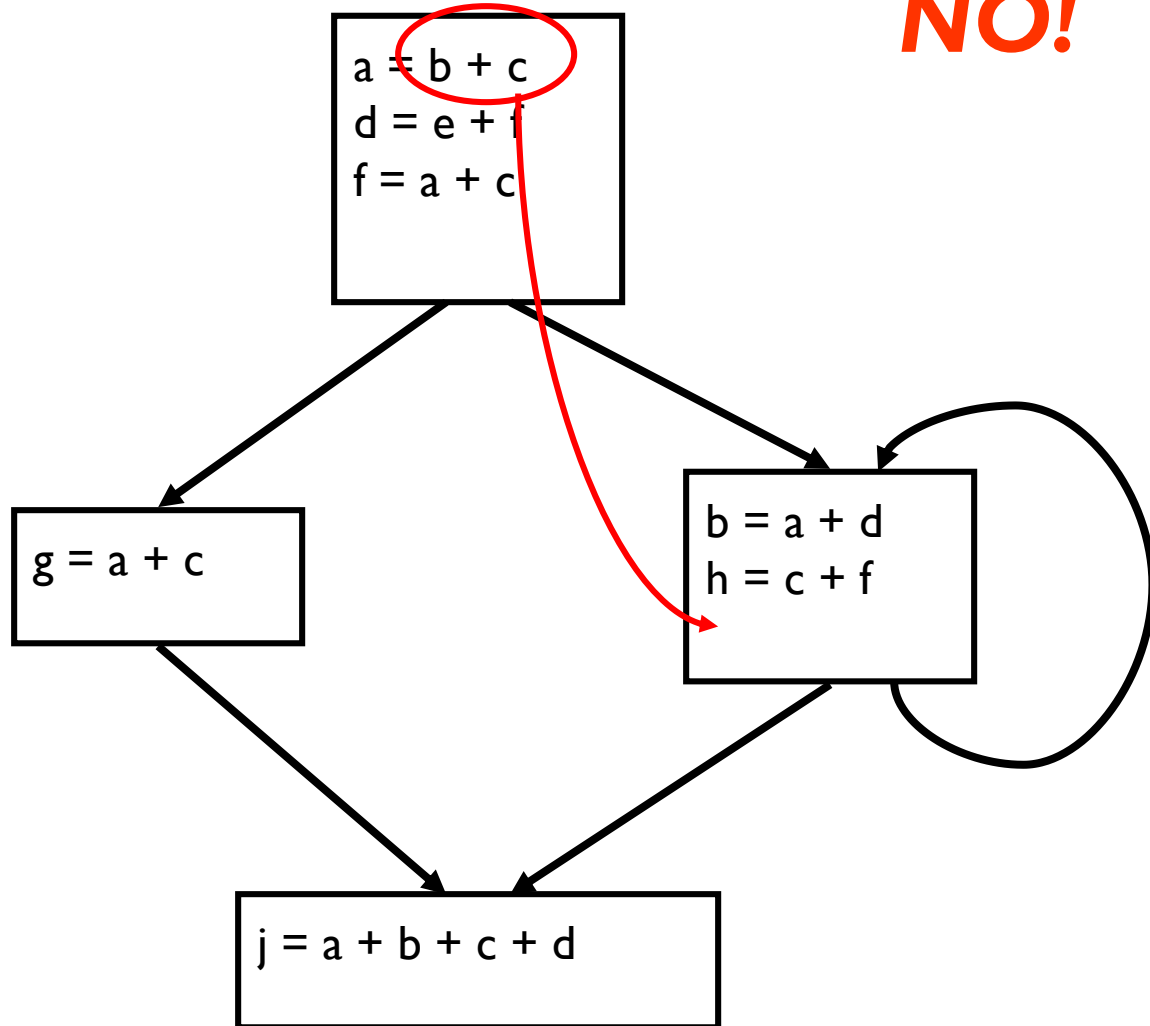
# Is the Expression Available?

**NO!**



# Is the Expression Available?

**NO!**



# **Transformation: Common Subexpression Elimination**

Uses the results of available expressions

## **Check:**

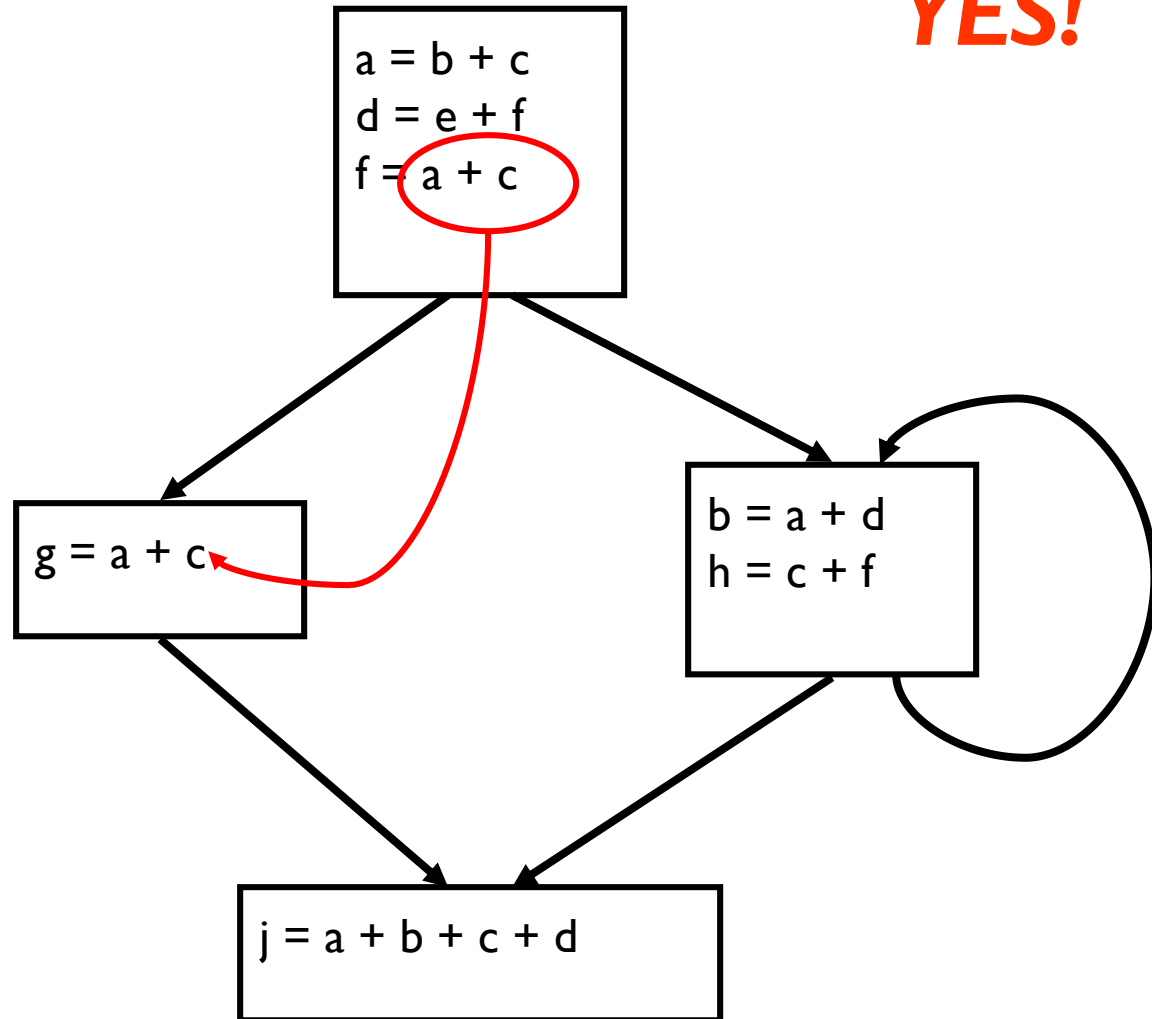
- If the expression is available and computed before,

## **Transform:**

- At the first location, create a temporary variable
- Replace the latter occurrence(s) with the temporary variable name.

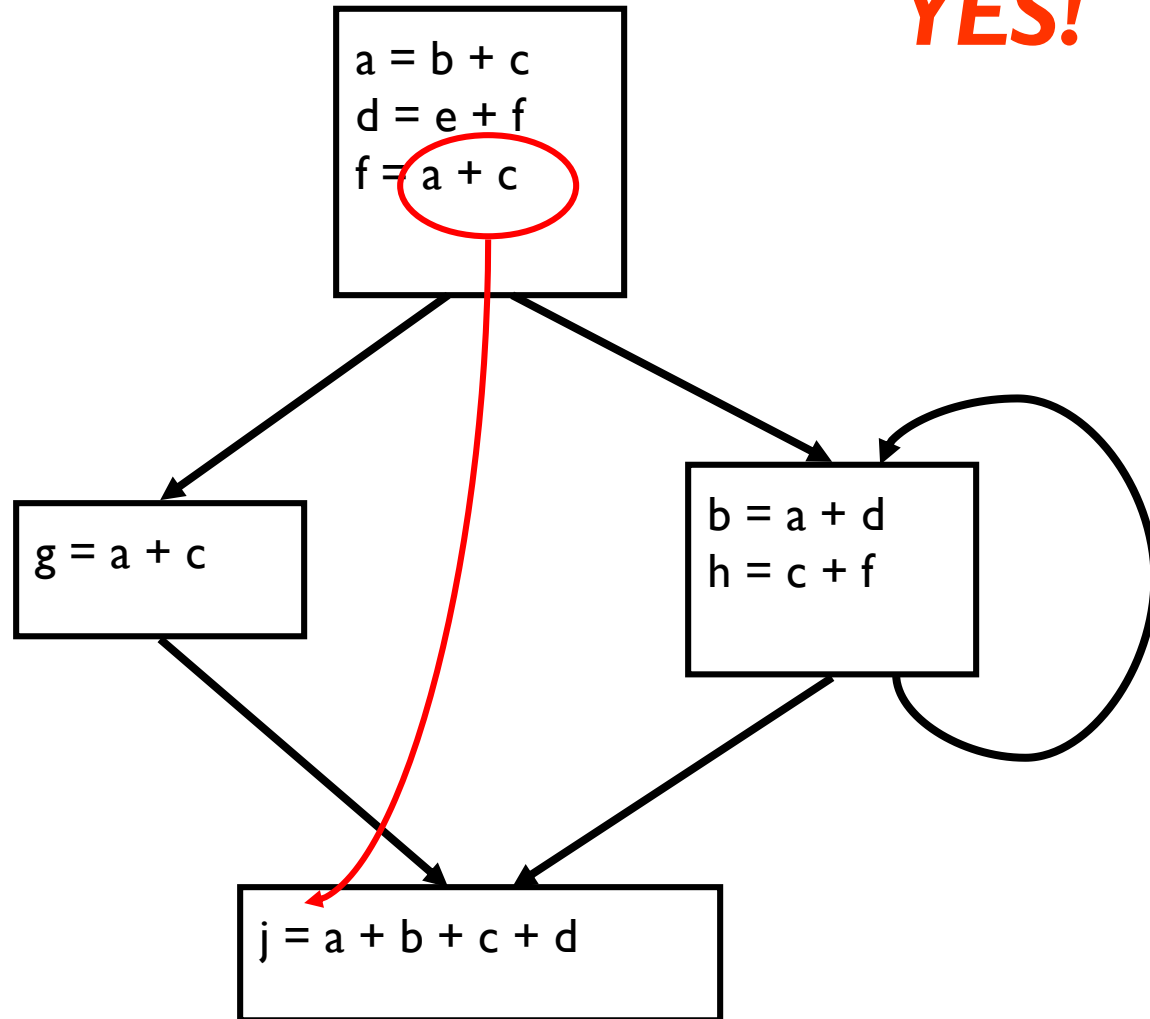
# Use of Available Expression

**YES!**

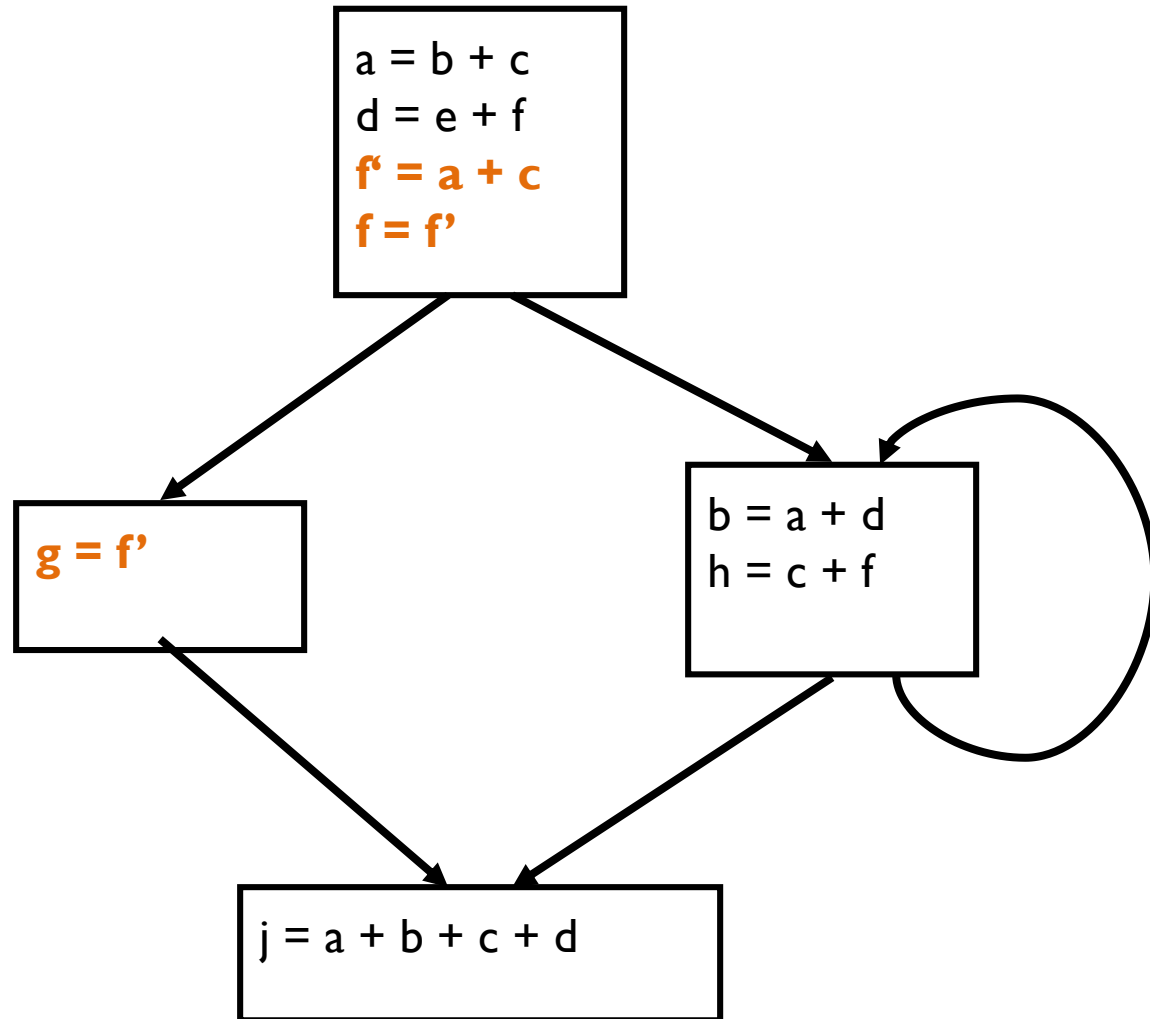


# Use of Available Expression

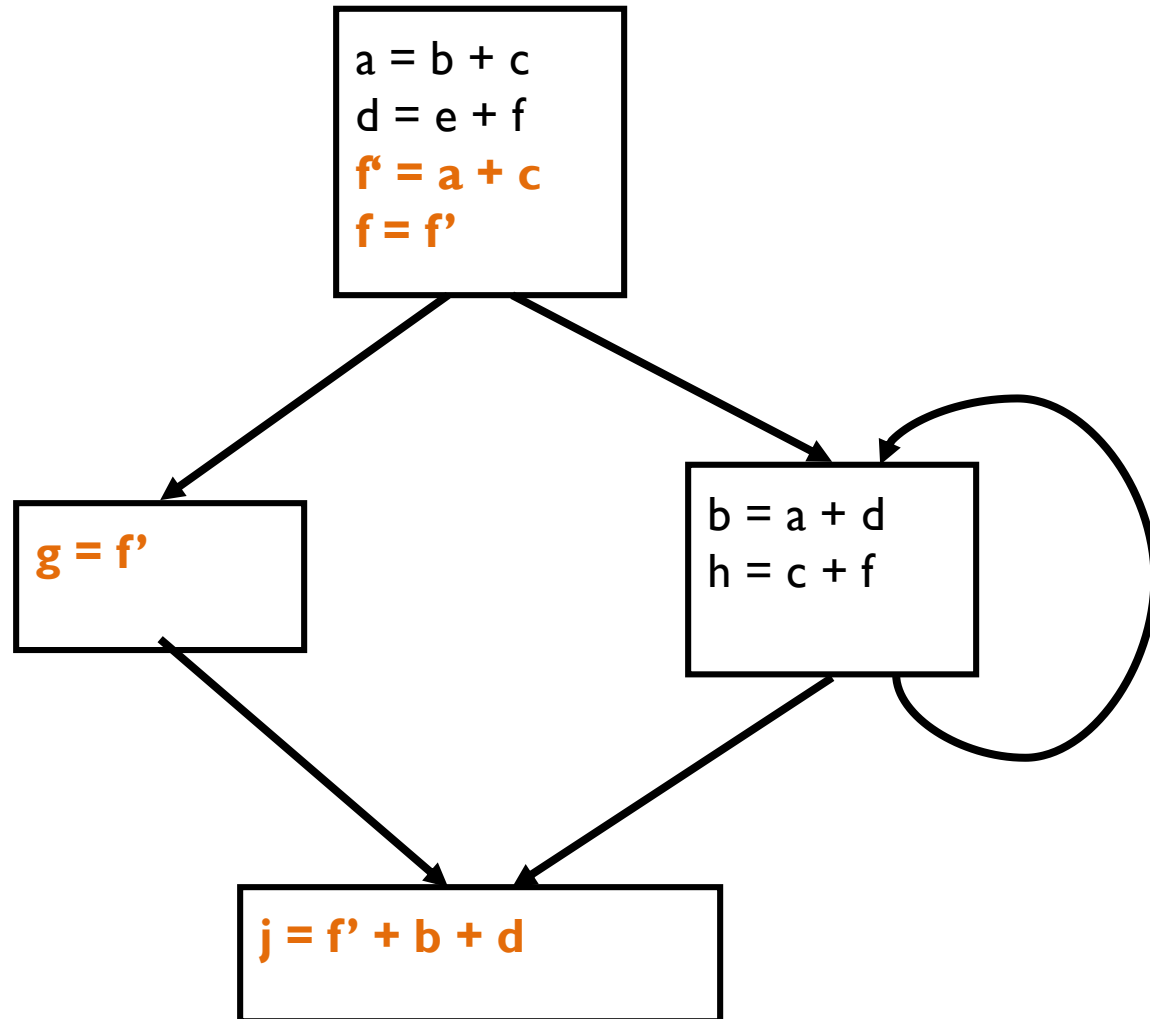
**YES!**



# Use of Available Expression



# Use of Available Expression





# Formalizing Analysis

Each basic block has

- IN = set of expressions available at start of block
- OUT = set of expressions available at end of block
- GEN = set of expressions computed in block
- KILL = set of expressions killed in in block
  
- Compiler scans each basic block to derive GEN and KILL sets
  
- Comparison with reaching definitions:
  - definition reaches a basic block if it comes from **ANY** predecessor in CFG
  - expression is available at a basic block only if it is available from **ALL** predecessors in CFG

# Dataflow Equations

- $IN[b] = OUT[b_1] \cap \dots \cap OUT[b_n]$ 
  - where  $b_1, \dots, b_n$  are predecessors of  $b$  in CFG
- $OUT[b] = (IN[b] - KILL[b]) \cup GEN[b]$
- $IN[entry] = 0000$
- Result: system of equations

# Solving Equations

- Use fixed point algorithm
- $IN[entry] = 0000$
- Initialize  $OUT[b] = 1111$
- Repeatedly apply equations
  - $IN[b] = OUT[b_1] \cap \dots \cap OUT[b_n]$
  - $OUT[b] = (IN[b] - KILL[b]) \cup GEN[b]$
- Use a worklist algorithm to reach fixed point

# Available Expressions Algorithm

for all nodes  $n$  in  $N$

$OUT[n] = E$ ; //  $OUT[n] = E - KILL[n]$ ;

$IN[Entry] = \text{emptyset}$ ;

$OUT[Entry] = GEN[Entry]$ ;

$Changed = N - \{ Entry \}$ ; //  $N = \text{all nodes in graph}$

while ( $Changed \neq \text{emptyset}$ )

  choose a node  $n$  in  $Changed$ ;

$Changed = Changed - \{ n \}$ ;

$IN[n] = E$ ; //  $E$  is set of all expressions

  for all nodes  $p$  in  $\text{predecessors}(n)$

$IN[n] = IN[n] \cap OUT[p]$ ;

$OUT[n] = GEN[n] \cup (IN[n] - KILL[n])$ ;

  if ( $OUT[n]$  changed)

    for all nodes  $s$  in  $\text{successors}(n)$

$Changed = Changed \cup \{ s \}$ ;

# Questions

Does algorithm always halt?

If expression is available in some execution, is it always marked as available in analysis?

If expression is not available in some execution, can it be marked as available in analysis?

# Analysis: Variable Liveness

A variable  $v$  is live at point  $p$  if

- $v$  is used along some path starting at  $p$ , and
- no definition of  $v$  along the path before the use.

When is a variable  $v$  dead at point  $p$ ?

- No use of  $v$  on any path from  $p$  to exit node, or
- If all paths from  $p$  redefine  $v$  before using  $v$ .

# What Use is Liveness Information?

Register allocation.

- If a variable is dead, can reassign its register

Dead code elimination.

- Eliminate assignments to variables not read later.
- But must not eliminate last assignment to variable (such as instance variable) visible outside CFG.
- Can eliminate other dead assignments.
- Handle by making all externally visible variables live on exit from CFG

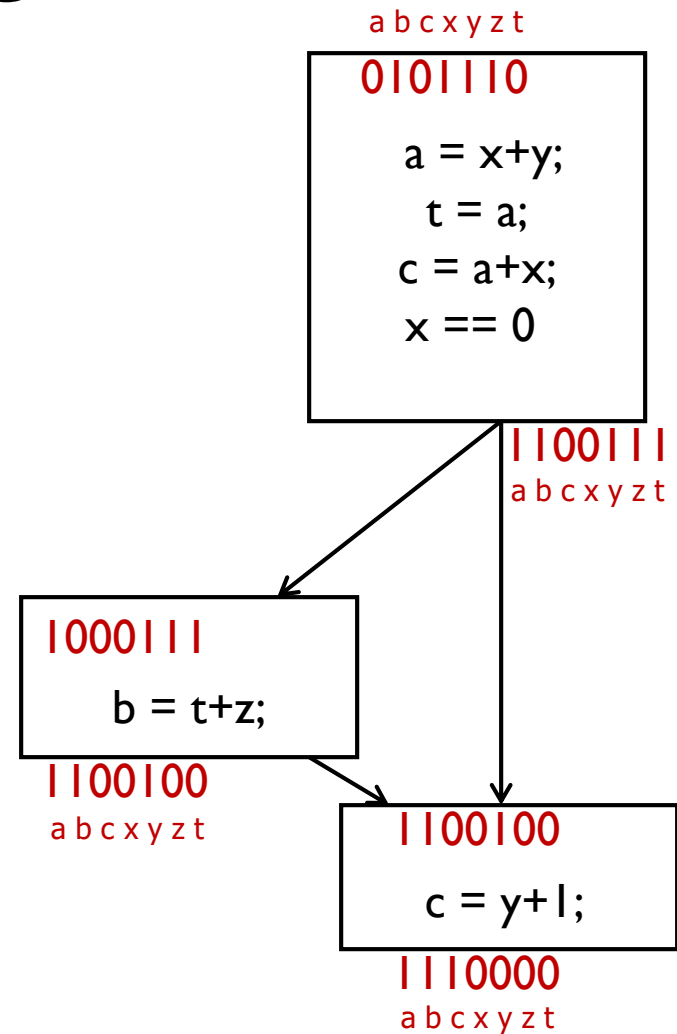
# Conceptual Idea of Analysis

- Simulate execution
- But start from exit and go backwards in CFG
- Compute liveness information from end to beginning of basic blocks



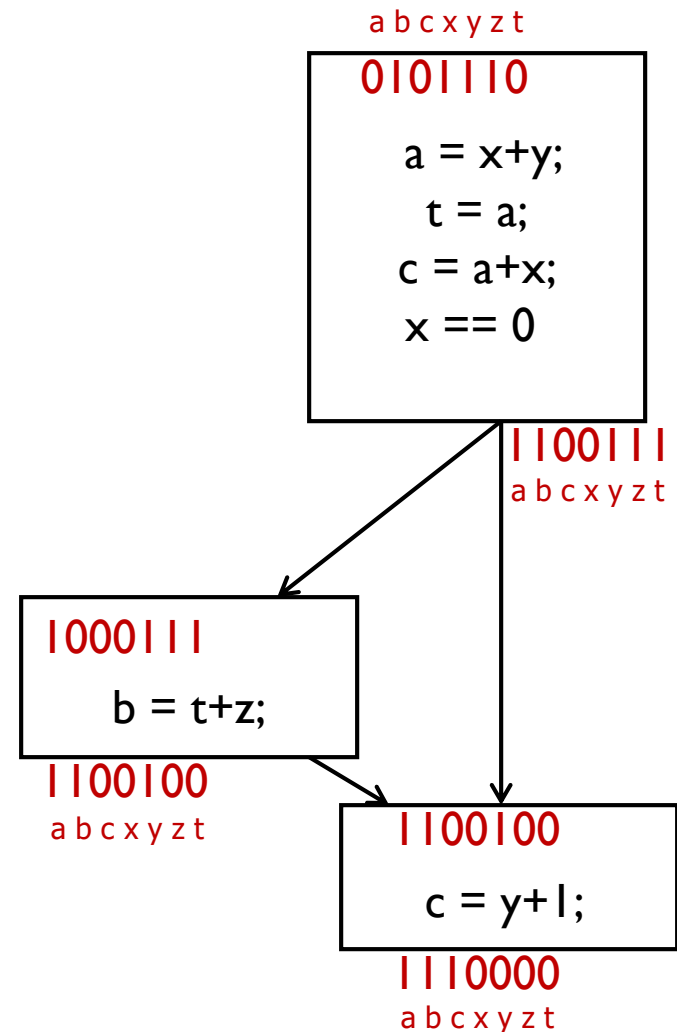
# Liveness Example

- Assume a,b,c visible outside method
  - So they are live on exit
- Assume x,y,z,t not visible outside method
- Represent Liveness Using Bit Vector
  - order is abcxyzt



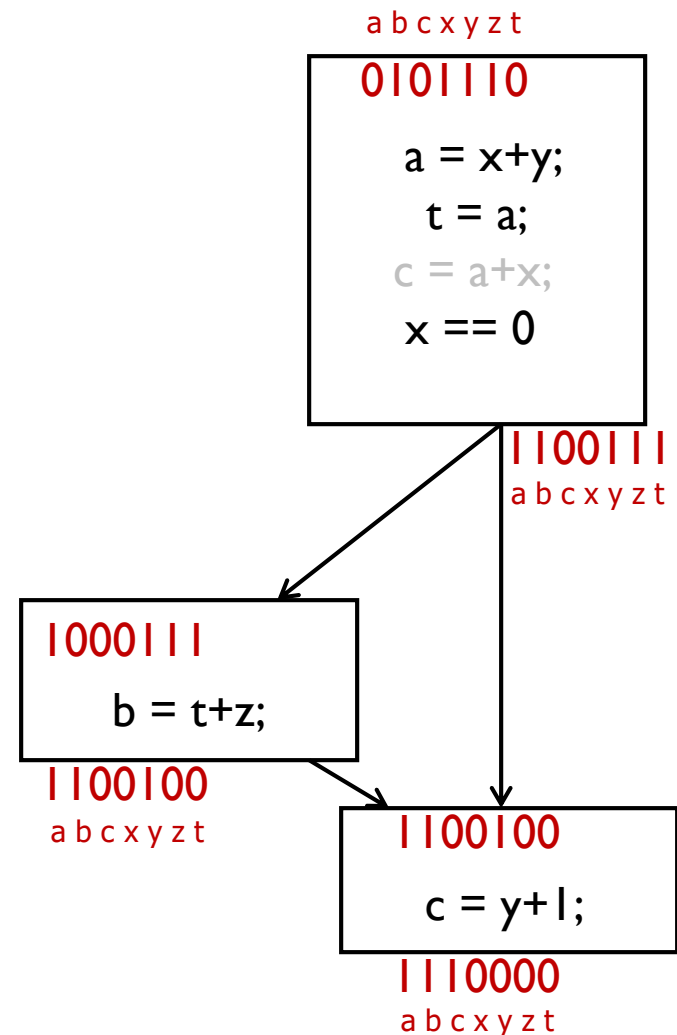
# Transformation: Dead Code Elimination

- Assume a,b,c visible outside method
  - So they are live on exit
- Assume x,y,z,t not visible outside method
- Represent Liveness Using Bit Vector
  - order is abcxyzt
- Remove dead definitions



# Transformation: Dead Code Elimination

- Assume a,b,c visible outside method
  - So they are live on exit
- Assume x,y,z,t not visible outside method
- Represent Liveness Using Bit Vector
  - order is abcxyzt
- Remove dead definitions



# Formalizing Analysis

- Each basic block has
  - IN - set of variables live at start of block
  - OUT - set of variables live at end of block
  - USE - set of variables with upwards exposed uses in block
  - DEF - set of variables defined in block
- $USE[x = z; x = x+1;] = \{ z \}$  (x not in USE)
- $DEF[x = z; x = x+1; y = 1;] = \{x, y\}$
- Compiler scans each basic block to derive USE and DEF sets

# Liveness Algorithm

for all nodes  $n$  in  $N - \{ \text{Exit} \}$

$IN[n] = \text{emptyset};$

$OUT[\text{Exit}] = \text{emptyset};$

$IN[\text{Exit}] = \text{use}[\text{Exit}];$

$\text{Changed} = N - \{ \text{Exit} \};$

while ( $\text{Changed} \neq \text{emptyset}$ )

    choose a node  $n$  in  $\text{Changed};$

$\text{Changed} = \text{Changed} - \{ n \};$

$OUT[n] = \text{emptyset};$

    for all nodes  $s$  in  $\text{successors}(n)$

$OUT[n] = OUT[n] \cup IN[s];$

$IN[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]);$

    if ( $IN[n]$  changed)

        for all nodes  $p$  in  $\text{predecessors}(n)$

$\text{Changed} = \text{Changed} \cup \{ p \};$

# Similar to Other Dataflow Algorithms

Backwards analysis, not forwards

Still have transfer functions

Still have confluence operators

Can generalize framework to work for both forwards and backwards analyses

# Comparison

## Reaching Definitions

for all nodes  $n$  in  $N$

OUT[ $n$ ] = emptyset;

IN[Entry] = emptyset;

OUT[Entry] = GEN[Entry];

Changed =  $N - \{ \text{Entry} \}$ ;

while (Changed != emptyset)

  choose a node  $n$  in Changed;

  Changed = Changed - {  $n$  };

IN[ $n$ ] = emptyset;

for all nodes  $p$  in predecessors( $n$ )

  IN[ $n$ ] = IN[ $n$ ]  $\cup$  OUT[ $p$ ];

OUT[ $n$ ] = GEN[ $n$ ]  $\cup$  (IN[ $n$ ] - KILL[ $n$ ]);

if (OUT[ $n$ ] changed)

  for all nodes  $s$  in successors( $n$ )

    Changed = Changed  $\cup$  {  $s$  };

## Available Expressions

for all nodes  $n$  in  $N$

OUT[ $n$ ] = E;

IN[Entry] = emptyset;

OUT[Entry] = GEN[Entry];

Changed =  $N - \{ \text{Entry} \}$ ;

while (Changed != emptyset)

  choose a node  $n$  in Changed;

  Changed = Changed - {  $n$  };

IN[ $n$ ] = E;

for all nodes  $p$  in predecessors( $n$ )

  IN[ $n$ ] = IN[ $n$ ]  $\cap$  OUT[ $p$ ];

OUT[ $n$ ] = GEN[ $n$ ]  $\cup$  (IN[ $n$ ] - KILL[ $n$ ]);

if (OUT[ $n$ ] changed)

  for all nodes  $s$  in successors( $n$ )

    Changed = Changed  $\cup$  {  $s$  };

## Liveness

for all nodes  $n$  in  $N - \{ \text{Exit} \}$

IN[ $n$ ] = emptyset;

OUT[Exit] = emptyset;

IN[Exit] = use[Exit];

Changed =  $N - \{ \text{Exit} \}$ ;

while (Changed != emptyset)

  choose a node  $n$  in Changed;

  Changed = Changed - {  $n$  };

OUT[ $n$ ] = emptyset;

for all nodes  $s$  in successors( $n$ )

  OUT[ $n$ ] = OUT[ $n$ ]  $\cup$  IN[ $p$ ];

IN[ $n$ ] = use[ $n$ ]  $\cup$  (out[ $n$ ] - def[ $n$ ]);

if (IN[ $n$ ] changed)

  for all nodes  $p$  in predecessors( $n$ )

    Changed = Changed  $\cup$  {  $p$  };

# Comparison

## Reaching Definitions

for all nodes  $n$  in  $N$

$OUT[n] = \text{emptyset};$

$IN[\text{Entry}] = \text{emptyset};$

$OUT[\text{Entry}] = \text{GEN}[\text{Entry}];$

$\text{Changed} = N - \{ \text{Entry} \};$

while ( $\text{Changed} \neq \text{emptyset}$ )

  choose a node  $n$  in  $\text{Changed}$ ;

$\text{Changed} = \text{Changed} - \{ n \};$

$IN[n] = \text{emptyset};$

for all nodes  $p$  in  $\text{predecessors}(n)$

$IN[n] = IN[n] \cup OUT[p];$

$OUT[n] = \text{GEN}[n] \cup (IN[n] - \text{KILL}[n]);$

if ( $OUT[n]$  changed)

  for all nodes  $s$  in  $\text{successors}(n)$

$\text{Changed} = \text{Changed} \cup \{ s \};$

## Available Expressions

for all nodes  $n$  in  $N$

$OUT[n] = E;$

$IN[\text{Entry}] = \text{emptyset};$

$OUT[\text{Entry}] = \text{GEN}[\text{Entry}];$

$\text{Changed} = N - \{ \text{Entry} \};$

while ( $\text{Changed} \neq \text{emptyset}$ )

  choose a node  $n$  in  $\text{Changed}$ ;

$\text{Changed} = \text{Changed} - \{ n \};$

$IN[n] = E;$

for all nodes  $p$  in  $\text{predecessors}(n)$

$IN[n] = IN[n] \cap OUT[p];$

$OUT[n] = \text{GEN}[n] \cup (IN[n] - \text{KILL}[n]);$

if ( $OUT[n]$  changed)

  for all nodes  $s$  in  $\text{successors}(n)$

$\text{Changed} = \text{Changed} \cup \{ s \};$



# Comparison

## Reaching Definitions

for all nodes  $n$  in  $N$

OUT[ $n$ ] = emptyset;  
IN[Entry] = emptyset;  
OUT[Entry] = GEN[Entry];  
Changed =  $N - \{ \text{Entry} \}$ ;

while (Changed  $\neq$  emptyset)  
  choose a node  $n$  in Changed;  
  Changed = Changed - {  $n$  };

IN[ $n$ ] = emptyset;  
for all nodes  $p$  in predecessors( $n$ )  
  IN[ $n$ ] = IN[ $n$ ]  $\cup$  OUT[ $p$ ];

OUT[ $n$ ] = GEN[ $n$ ]  $\cup$  (IN[ $n$ ] - KILL[ $n$ ]);

if (OUT[ $n$ ] changed)  
  for all nodes  $s$  in successors( $n$ )  
    Changed = Changed  $\cup$  {  $s$  };

## Liveness

for all nodes  $n$  in  $N$

IN[ $n$ ] = emptyset;  
OUT[Exit] = emptyset;  
IN[Exit] = use[Exit];  
Changed =  $N - \{ \text{Exit} \}$ ;

while (Changed  $\neq$  emptyset)  
  choose a node  $n$  in Changed;  
  Changed = Changed - {  $n$  };

OUT[ $n$ ] = emptyset;  
for all nodes  $s$  in successors( $n$ )  
  OUT[ $n$ ] = OUT[ $n$ ]  $\cup$  IN[ $p$ ];

IN[ $n$ ] = use[ $n$ ]  $\cup$  (out[ $n$ ] - def[ $n$ ]);

if (IN[ $n$ ] changed)  
  for all nodes  $p$  in predecessors( $n$ )  
    Changed = Changed  $\cup$  {  $p$  };

# Basic Idea

Information about program represented using values from algebraic structure called **lattice**

Analysis produces lattice value for each program point

## **Two flavors** of analysis

- Forward dataflow analysis [e.g., Reachability]
- Backward dataflow analysis [e.g. Live Variables]

# Forward Dataflow Analysis

Analysis propagates values forward through control flow graph *with flow of control*

- Each node has a ***transfer function***  $f$ 
  - Input – value at program point before node
  - Output – new value at program point after node
- Values flow from program points after predecessor nodes to program points before successor nodes
- **At join points**, values are combined using a merge function

# Backward Dataflow Analysis

Analysis propagates values backward through control flow graph *against flow of control*

- Each node has a **transfer function**  $f$ 
  - Input – value at program point after node
  - Output – new value at program point before node
- Values flow from program points before successor nodes to program points after predecessor nodes
- **At split points**, values are combined using a merge function

# Partial Orders

## Set P

Partial order relation  $\leq$  such that  $\forall x, y, z \in P$

- $x \leq x$  (reflexive)
- $x \leq y$  and  $y \leq x$  implies  $x = y$  (asymmetric)
- $x \leq y$  and  $y \leq z$  implies  $x \leq z$  (transitive)

Can use partial order to define

- Upper and lower bounds
- Least upper bound
- Greatest lower bound

# Upper Bounds

If  $S \subseteq P$  then

- $x \in P$  is an upper bound of  $S$  if  $\forall y \in S. y \leq x$
- $x \in P$  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  $S$
- $\vee$  - **join**, least upper bound, **lub**, supremum, **sup**
  - $\vee S$  is the least upper bound of  $S$
  - $x \vee y$  is the least upper bound of  $\{x, y\}$

# Lower Bounds

If  $S \subseteq P$  then

- $x \in P$  is a lower bound of  $S$  if  $\forall y \in S. x \leq y$
- $x \in P$  is the greatest lower bound of  $S$  if
  - $x$  is a lower bound of  $S$ , and
  - $y \leq x$  for all lower bounds  $y$  of  $S$
- $\wedge$  - **meet**, greatest lower bound, **glb**, infimum, **inf**
  - $\wedge S$  is the greatest lower bound of  $S$
  - $x \wedge y$  is the greatest lower bound of  $\{x, y\}$

# Covering

$x < y$  if  $x \leq y$  and  $x \neq y$

**$x$  is covered by  $y$**  ( $y$  covers  $x$ ) if

- $x < y$ , and
- $x \leq z < y$  implies  $x = z$

Conceptually,  $y$  covers  $x$  if there are no elements between  $x$  and  $y$

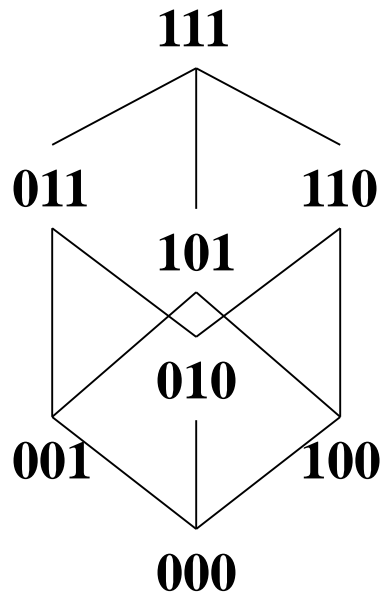


# Example

$P = \{ 000, 001, 010, 011, 100, 101, 110, 111 \}$

(standard boolean lattice, also called **hypercube**)

$x \leq y$  is equivalent to  $(x \text{ bitwise-and } y) = x$



## Hasse Diagram

- If  $y$  covers  $x$ 
  - Line from  $y$  to  $x$
  - $y$  above  $x$  in diagram

# Lattices

Consider poset  $(P, \leq)$  and the operators  $\wedge$  (meet) and  $\vee$  (join)

If for all  $x, y \in P$  there exist  $x \wedge y$  and  $x \vee y$ ,  
then  $P$  is a **lattice**.

If for all  $S \subseteq P$  there exist  $\wedge S$  and  $\vee S$   
then  $P$  is a **complete lattice**.

All **finite** lattices are **complete**

Example of a lattice that is not complete: Integers  $\mathbb{Z}$

- For any  $x, y \in \mathbb{Z}$ ,  $x \vee y = \max(x, y)$ ,  $x \wedge y = \min(x, y)$
- But  $\vee \mathbb{Z}$  and  $\wedge \mathbb{Z}$  do not exist
- $\mathbb{Z} \cup \{+\infty, -\infty\}$  is a complete lattice

# Top and Bottom

Greatest element of  $P$  (if it exists) is top ( $\top$ )

Least element of  $P$  (if it exists) is bottom ( $\perp$ )

# Connection Between $\leq$ , $\wedge$ , and $\vee$

The following 3 properties are equivalent:

- $x \leq y$
- $x \vee y = y$
- $x \wedge y = x$

Let's prove:

- $x \leq y$  implies  $x \vee y = y$  and  $x \wedge y = x$
- $x \vee y = y$  implies  $x \leq y$
- $x \wedge y = x$  implies  $x \leq y$

Then by transitivity, we can obtain

- $x \vee y = y$  implies  $x \wedge y = x$
- $x \wedge y = x$  implies  $x \vee y = y$

# Connecting Lemma Proofs

Prove:  $x \leq y$  implies  $x \vee y = y$

- $x \leq y$  implies  $y$  is an upper bound of  $\{x,y\}$ .
- Any upper bound  $z$  of  $\{x,y\}$  must satisfy  $y \leq z$ .
- So  $y$  is least upper bound of  $\{x,y\}$  and  $x \vee y = y$

Prove:  $x \leq y$  implies  $x \wedge y = x$

- $x \leq y$  implies  $x$  is a lower bound of  $\{x,y\}$ .
- Any lower bound  $z$  of  $\{x,y\}$  must satisfy  $z \leq x$ .
- So  $x$  is greatest lower bound of  $\{x,y\}$  and  $x \wedge y = x$

# Connecting Lemma Proofs

Prove:  $x \vee y = y$  implies  $x \leq y$

- $y$  is an upper bound of  $\{x, y\}$  implies  $x \leq y$

Prove:  $x \wedge y = x$  implies  $x \leq y$

- $x$  is a lower bound of  $\{x, y\}$  implies  $x \leq y$

# Lattices as Algebraic Structures

We have defined  $\vee$  and  $\wedge$  in terms of  $\leq$

We will now define  $\leq$  in terms of  $\vee$  and  $\wedge$

- Start with  $\vee$  and  $\wedge$  as arbitrary algebraic operations that satisfy ***associative, commutative, idempotence, and absorption*** laws
- Will define  $\leq$  using  $\vee$  and  $\wedge$
- Will show that  $\leq$  is a partial order

Intuitive concept of  $\vee$  and  $\wedge$  as information combination operators (or, and)

# Algebraic Properties of Lattices

Assume arbitrary operations  $\vee$  and  $\wedge$  such that

- $(x \vee y) \vee z = x \vee (y \vee z)$  (associativity of  $\vee$ )
- $(x \wedge y) \wedge z = x \wedge (y \wedge z)$  (associativity of  $\wedge$ )
- $x \vee y = y \vee x$  (commutativity of  $\vee$ )
- $x \wedge y = y \wedge x$  (commutativity of  $\wedge$ )
- $x \vee x = x$  (idempotence of  $\vee$ )
- $x \wedge x = x$  (idempotence of  $\wedge$ )
- $x \vee (x \wedge y) = x$  (absorption of  $\vee$  over  $\wedge$ )
- $x \wedge (x \vee y) = x$  (absorption of  $\wedge$  over  $\vee$ )



# Connection Between $\wedge$ and $\vee$

$x \vee y = y$  if and only if  $x \wedge y = x$

Proof of  $x \vee y = y$  implies  $x = x \wedge y$

$$x = x \wedge (x \vee y) \quad (\text{by absorption})$$

$$= x \wedge y \quad (\text{by assumption})$$

Proof of  $x \wedge y = x$  implies  $y = x \vee y$

$$y = y \vee (y \wedge x) \quad (\text{by absorption})$$

$$= y \vee (x \wedge y) \quad (\text{by commutativity})$$

$$= y \vee x \quad (\text{by assumption})$$

$$= x \vee y \quad (\text{by commutativity})$$

# Properties of $\leq$

Define  $x \leq y$  if  $x \vee y = y$

Proof of transitive property. Must show that

$x \vee y = y$  and  $y \vee z = z$  implies  $x \vee z = z$

$$x \vee z = x \vee (y \vee z) \quad (\text{by assumption})$$

$$= (x \vee y) \vee z \quad (\text{by associativity})$$

$$= y \vee z \quad (\text{by assumption})$$

$$= z \quad (\text{by assumption})$$

# Properties of $\leq$

Proof of asymmetry property. Must show that

$x \vee y = y$  and  $y \vee x = x$  implies  $x = y$

$x = y \vee x$  (by assumption)

$= x \vee y$  (by commutativity)

$= y$  (by assumption)

Proof of reflexivity property. Must show that

$x \vee x = x$

$x \vee x = x$  (by idempotence)

# Properties of $\leq$

Induced operation  $\leq$  agrees with original definitions of  $\vee$  and  $\wedge$ , i.e.,

- $x \vee y = \sup \{x, y\}$
- $x \wedge y = \inf \{x, y\}$

# Proof of $x \vee y = \sup \{x, y\}$

Consider any upper bound  $u$  for  $x$  and  $y$ .

Given  $x \vee u = u$  and  $y \vee u = u$ , must show

$x \vee y \leq u$ , i.e.,  $(x \vee y) \vee u = u$

$$u = x \vee u \quad (\text{by assumption})$$

$$= x \vee (y \vee u) \quad (\text{by assumption})$$

$$= (x \vee y) \vee u \quad (\text{by associativity})$$

# Proof of $x \wedge y = \inf \{x, y\}$

- Consider any lower bound  $L$  for  $x$  and  $y$ .
- Given  $x \wedge L = L$  and  $y \wedge L = L$ , must show

$$L \leq x \wedge y, \text{ i.e., } (x \wedge y) \wedge L = L$$

$$\begin{aligned} L &= x \wedge L && \text{(by assumption)} \\ &= x \wedge (y \wedge L) && \text{(by assumption)} \\ &= (x \wedge y) \wedge L && \text{(by associativity)} \end{aligned}$$

# Chains

A **set S is a chain** if  $\forall x, y \in S. y \leq x$  or  $x \leq y$

P has no infinite chains if every chain in P is finite

P satisfies the **ascending chain condition** if for all sequences  $x_1 \leq x_2 \leq \dots$  there exists n such that  $x_n = x_{n+1} = \dots$

# 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**

We will use  $\vee$  to combine values at control-flow join points



# Transfer Functions

Transfer function  $f: P \rightarrow P$  is defined for each node in control flow graph

The function  $f$  models effect of the node on the program information

# Transfer Functions

Each dataflow analysis problem has a **set F of transfer functions**  $f: P \rightarrow P$ . This set F contains:

- **Identity function** belongs to the set,  $i \in F$
- F must be **closed under composition**:  
 $\forall f, g \in F$ . the function  $h = \lambda x. f(g(x)) \in F$
- Each  $f \in F$  must be **monotone**:  
 $x \leq y$  implies  $f(x) \leq f(y)$
- Sometimes all  $f \in F$  are **distributive**:  
 $f(x \vee y) = f(x) \vee f(y)$
- Note that Distributivity implies monotonicity

# Distributivity Implies Monotonicity

## Proof.

Assume distributivity:  $f(x \vee y) = f(x) \vee f(y)$

Must show:  $x \vee y = y$  implies  $f(x) \vee f(y) = f(y)$

$$f(y) = f(x \vee y) \quad (\text{by assumption})$$

$$= f(x) \vee f(y) \quad (\text{by distributivity})$$

**Putting the Pieces Together...**

# Forward Dataflow Analysis

*Simulates execution of program forward with flow of control*

For each node  $n$ , we have

- $in_n$  – value at program point before  $n$
- $out_n$  – value at program point after  $n$
- $f_n$  – transfer function for  $n$  (given  $in_n$ , computes  $out_n$ )

Requires that solution satisfied

- $\forall n. out_n = f_n(in_n)$
- $\forall n \neq n_0. in_n = \vee \{ out_m . m \text{ in } \text{pred}(n) \}$
- $in_{n_0} = I$ ,  $I$  summarizes information at start of program

# Dataflow Equations

Compiler processes program to obtain a set of dataflow equations

$$\begin{aligned} \text{out}_n &:= f_n(\text{in}_n) \\ \text{in}_n &:= \vee \{ \text{out}_m \mid m \text{ in pred}(n) \} \end{aligned}$$

Conceptually separates analysis problem from program

# Worklist Algorithm for Solving Forward Dataflow Equations

for each  $n$  do  $out_n := f_n(\perp)$

$in_{n_0} := I; out_{n_0} := f_{n_0}(I)$

worklist :=  $N - \{ n_0 \}$

while worklist  $\neq \emptyset$  do

    remove a node  $n$  from worklist

$in_n := \bigvee \{ out_m . m \text{ in } \text{pred}(n) \}$

$out_n := f_n(in_n)$

    if  $out_n$  changed then

        worklist := worklist  $\cup$  succ( $n$ )

# Correctness Argument

Why does result satisfy dataflow equations?

- Whenever process a node  $n$ , algorithm sets  $out_n := f_n(in_n)$   
Therefore, the algorithm ensures that  $out_n = f_n(in_n)$
- Whenever  $out_m$  changes, put  $succ(m)$  on worklist. Consider any node  $n \in succ(m)$ . It will eventually come off worklist and algorithm will set
$$in_n := \vee \{ out_m . m \text{ in } pred(n) \}$$
to ensure that  $in_n = \vee \{ out_m . m \text{ in } pred(n) \}$
- So final solution will satisfy dataflow equations
- Need also to ensure that the dataflow equalities correspond to the states in the program execution (this comes later!)



# Termination Argument

Why does algorithm terminate?

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**

# Termination Argument (Details)

- For lattice  $(L, \leq)$
- Start: each node  $n \in CFG$  has an initial IN set, called  $IN_0[n]$
- When  $F$  is **monotone**, for each  $n$ , successive values of  $IN[n]$  form a non-decreasing sequence.
  - Any chain starting at  $x \in L$  has at most  $c_x$  elements
  - $x = IN[n]$  can increase in value at most  $c_x$  times
  - Then  $C = \max_{n \in CFG} c_{IN[n]}$  is finite
- On every iteration, at least one IN set must increase in value
  - If loop executes  $N \times C$  times, all IN sets would be  $T$
  - The algorithm terminates in  $O(N \times C)$  steps

# Speed of Convergence

**Loop Connectedness**  $d(G)$ : for a reducible CFG  $G$ , it is the maximum number of back edges in any acyclic path in  $G$ .

**Rapid:** A Data-flow framework  $(L, \leq, F)$  is called **Rapid** if

$$\forall f \in F, \forall x \in L. \quad x \leq f(x) \wedge f(T)$$

**Kam & Ullman, 1976: Data-flow Framework is Rapid**

- The depth-first version of the iterative algorithm halts in at most  $d(G) + 3$  passes over the graph
- If the lattice  $L$  has  $T$ , at most  $d(G) + 2$  passes are needed

**In practice:**

- $d(G) < 3$ , so the algorithm makes less than 6 passes over the graph
- The rapid condition implies that the information around the loop stabilizes in 2 steps

# Widening Operators

Detect lattice values that may be part of infinitely ascending chain

Artificially raise value to least upper bound of chain

## Example:

- Lattice is set of all subsets of integers
- E.g, it can collect possible values of the variables during the execution of program
- Widening operator might raise all sets of size  $n$  or greater to TOP (likely to be useful for loops)

# Reaching Definitions Algorithm

*(Reminder)*

```
for all nodes n in N
    OUT[n] = emptyset; // OUT[n] = GEN[n];
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry }; // N = all nodes in graph

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = emptyset;
    for all nodes p in predecessors(n)
        IN[n] = IN[n] U OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
        for all nodes s in successors(n)
            Changed = Changed U { s };
```

# General Worklist Algorithm

*(Reminder)*

for each  $n$  do  $out_n := f_n(\perp)$

$in_{n_0} := I$ ;  $out_{n_0} := f_{n_0}(I)$

worklist :=  $N - \{ n_0 \}$

while worklist  $\neq \emptyset$  do

    remove a node  $n$  from worklist

$in_n := \bigvee \{ out_m . m \text{ in } \text{pred}(n) \}$

$out_n := f_n(in_n)$

    if  $out_n$  changed then

        worklist := worklist  $\cup$  succ( $n$ )

# Reaching Definitions

$P$  = powerset of set of all definitions in program (all subsets of set of definitions in program)

$\vee = \cup$  (order is  $\subseteq$ )

$\perp = \emptyset$

$l = \text{in}_{n0} = \perp$

$F$  = all functions  $f$  of the form  $f(x) = a \cup (x-b)$

- $b$  is set of definitions that node kills
- $a$  is set of definitions that node generates

General pattern for many transfer functions

- $f(x) = \text{GEN} \cup (x\text{-KILL})$

# Does Reaching Definitions Framework Satisfy Properties?

## $\subseteq$ satisfies conditions for $\leq$

- **Reflexivity:**  $x \subseteq x$
- **Assymetry:**  $x \subseteq y$  and  $y \subseteq x$  implies  $y = x$
- **Transitivity:**  $x \subseteq y$  and  $y \subseteq z$  implies  $x \subseteq z$

## F satisfies transfer function conditions

- **Identity:**  $\lambda x. \emptyset \cup (x - \emptyset) = \lambda x. x \in F$
- **Distributivity:** Will show  $f(x \cup y) = f(x) \cup f(y)$   
$$\begin{aligned} f(x) \cup f(y) &= (a \cup (x - b)) \cup (a \cup (y - b)) \\ &= a \cup (x - b) \cup (y - b) = a \cup ((x \cup y) - b) \\ &= f(x \cup y) \end{aligned}$$



# Does Reaching Definitions Framework Satisfy Properties?

## What about composition of F?

Given  $f_1(x) = a_1 \cup (x - b_1)$  and  $f_2(x) = a_2 \cup (x - b_2)$   
we must show  $f_1(f_2(x))$  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  $f_1(f_2(x)) = a \cup (x - b)$

# Reaching Definitions is **Rapid**

*Convergence Is Fast*

$$\begin{array}{lcl} f(x) & \stackrel{?}{\geq} & x \wedge f(\top) \\ a_f \cup (x \cap b_f) & \stackrel{?}{\geq} & x \cap (a_f \cup (\top \cap b_f)) \\ a_f \cup (x \cap b_f) & \stackrel{?}{\geq} & x \cap (a_f \cup b_f) \\ a_f \cup (x \cap b_f) & \stackrel{?}{\geq} & (x \cap a_f) \cup (x \cap b_f) \\ \\ a_f & \geq & x \cap a_f \\ x \cap b_f & = & x \cap b_f \end{array}$$

# General Result

**All** GEN/KILL transfer function frameworks satisfy the three properties:

- Identity
- Distributivity
- Composition

And all of them converge rapidly

# Available Expressions

$P$  = powerset of set of all expressions in program  
(all subsets of set of expressions)

$\vee = \cap$  (order is  $\supseteq$ )

$\perp = P$

$I = in_{n_0} = \emptyset$

$F$  = all functions  $f$  of the form  $f(x) = a \cup (x-b)$

- $b$  is set of expressions that node kills
- $a$  is set of expressions that node generates

Another GEN/KILL analysis

# Concept of Conservatism

Reaching definitions use  $\cup$  as join

- Optimizations must take into account all definitions that reach along **ANY path**

Available expressions use  $\cap$  as join

- Optimization requires expression to be available along **ALL paths**

Optimizations must **conservatively take all possible executions into account.**

# Backward Dataflow Analysis

- Simulates execution of program backward against the flow of control
- For each node  $n$ , we have
  - $in_n$  – value at program point before  $n$
  - $out_n$  – value at program point after  $n$
  - $f_n$  – transfer function for  $n$  (given  $out_n$ , computes  $in_n$ )
- Require that solution satisfies
  - $\forall n. in_n = f_n(out_n)$
  - $\forall n \notin N_{final}. out_n = \vee \{ in_m . m \text{ in } succ(n) \}$
  - $\forall n \in N_{final} = out_n = \bigcirc$
  - Where  $\bigcirc$  summarizes information at end of program

# Worklist Algorithm for Solving Backward Dataflow Equations

```
for each n do  $in_n := f_n(\perp)$   
for each  $n \in N_{final}$  do  $out_n := \perp; in_n := f_n(\perp)$   
worklist :=  $N - N_{final}$ 
```

```
while worklist  $\neq \emptyset$  do  
  remove a node n from worklist  
   $out_n := \vee \{ in_m . m \text{ in } succ(n) \}$   
   $in_n := f_n(out_n)$   
  if  $in_n$  changed then  
    worklist := worklist  $\cup pred(n)$ 
```

# Live Variables

$P$  = powerset of set of all variables in program  
(all subsets of set of variables in program)

$\vee = \cup$  (order is  $\subseteq$ )

$\perp = \emptyset$

$\bigcirc = \emptyset$

$F$  = all functions  $f$  of the form  $f(x) = a \cup (x-b)$

- $b$  is set of variables that node kills
- $a$  is set of variables that node reads



# Meaning of Dataflow Results

Concept of **program state** **s** for control-flow graphs

- **Program point** **n** where execution is located  
(n is node that will execute next)
- Values of variables in program

Each execution generates a trajectory of states:

- $s_0; s_1; \dots; s_k$ , where each  $s_i \in S$
- $s_{i+1}$  generated from  $s_i$  by executing basic block to
  1. Update variable values
  2. Obtain new program point n

# Relating States to Analysis Result

- Meaning of analysis results is given by an abstraction function  $AF:ST \rightarrow P$
- Correctness condition: require that for all states  $s$

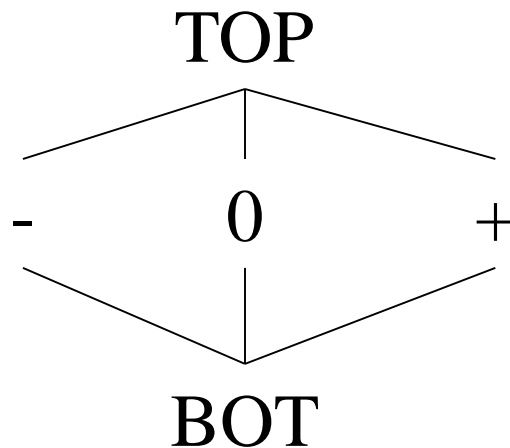
$$AF(s) \leq in_n$$

where  $n$  is the next statement to execute in state  $s$

# Sign Analysis Example

Sign analysis - compute sign of each variable  $v$

Base Lattice:  $P = \text{flat lattice on } \{-, 0, +\}$



Actual lattice records a value for each variable

- Example element:  $[a \rightarrow +, b \rightarrow 0, c \rightarrow -]$

# Interpretation of Lattice Values

If value of  $v$  in lattice is:

- BOT: no information about the sign of  $v$
- -: variable  $v$  is negative
- 0: variable  $v$  is 0
- +: variable  $v$  is positive
- TOP:  $v$  may be positive or negative or zero

What is abstraction function AF?

- $AF([v_1, \dots, v_n]) = [\text{sign}(v_1), \dots, \text{sign}(v_n)]$

$$\bullet \text{sign}(x) = \begin{cases} 0 & \text{if } v = 0 \\ + & \text{if } v > 0 \\ - & \text{if } v < 0 \end{cases}$$

# Operation $\otimes$ on Lattice

$\otimes$	BOT	-	0	+	TOP
BOT	BOT	BOT	0	BOT	BOT
-	BOT	+	0	-	TOP
0	0	0	0	0	0
+	BOT	-	0	+	TOP
TOP	BOT	TOP	0	TOP	TOP

# Transfer Functions

If  $n$  of the form  $v = c$

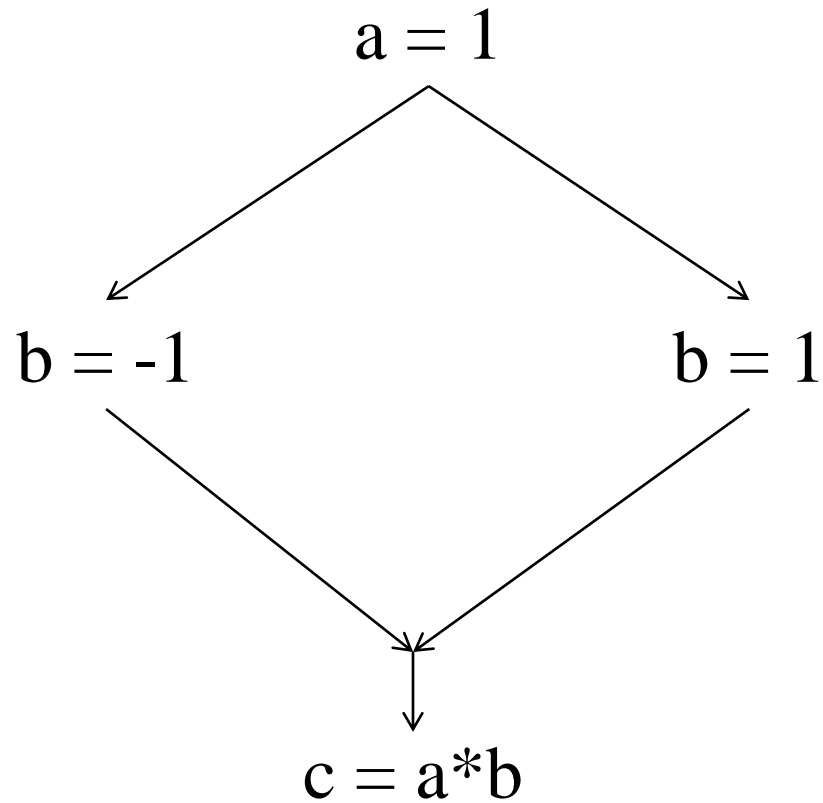
- $f_n(x) = x[v \rightarrow +]$  if  $c$  is positive
- $f_n(x) = x[v \rightarrow 0]$  if  $c$  is 0
- $f_n(x) = x[v \rightarrow -]$  if  $c$  is negative

If  $n$  of the form  $v_1 = v_2 * v_3$

- $f_n(x) = x[v_1 \rightarrow x[v_2] \otimes x[v_3]]$

I = TOP (uninitialized variables may have any sign)

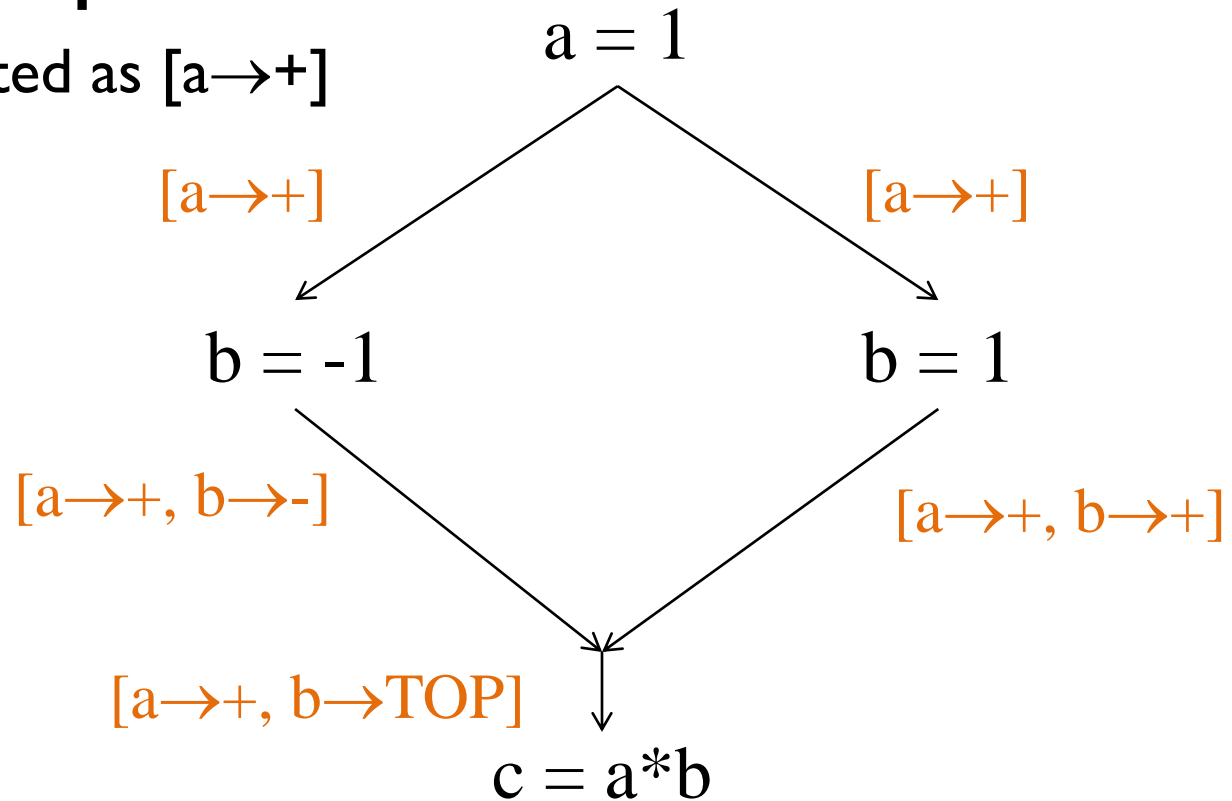
# Sign Analysis Example



# Imprecision In Example

## Abstraction Imprecision:

$[a \rightarrow 1]$  abstracted as  $[a \rightarrow +]$



## Control Flow Imprecision:

$[b \rightarrow \text{TOP}]$  summarizes results of all executions.

*(In any concrete execution state  $s$ ,  $AF(s)[b] \neq \text{TOP}$ )*



# General Sources of Imprecision

## Abstraction Imprecision

- Concrete values (integers) abstracted as lattice values (-,0, and +)
- 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 operation  $\vee$  moves up in lattice to combine values from different execution paths
- Typically if  $x \leq y$ , then  $x$  is more precise than  $y$

# Why To Allow Imprecision?

Make analysis tractable

Unbounded sets of values in execution

- Typically abstracted by finite set of lattice values

Execution may visit unbounded set of states

- Abstracted by computing joins of different paths

# Correctness of Solution

## Correctness condition:

- $\forall v . AF(s)[v] \leq in_n[v]$  (n is node, s is state)
- Reflects possibility of imprecision

## Proof:

- By the induction on the structure of the computation that produces s

# Meet Over Paths\* Solution

What solution would be ideal for a forward dataflow problem?

Consider a path  $p = n_0, n_1, \dots, n_k, n$  to a node  $n$   
(note that for all  $i, n_i \in \text{pred}(n_{i+1})$ )

The solution must take this path into account:

$$f_p(\perp) = (f_{n_k}(f_{n_{k-1}}(\dots f_{n_1}(f_{n_0}(\perp)) \dots))) \leq in_n$$

So the solution must have the property that

$$\bigvee \{f_p(\perp) \mid p \text{ is a path to } n\} \leq in_n$$

and ideally

$$\bigvee \{f_p(\perp) \mid p \text{ is a path to } n\} = in_n$$

\* Name exists for historical reasons; this is really a join

# Soundness Proof of Analysis Algorithm

**Property to prove:** For all paths  $p$  to  $n$ ,  $f_p(\perp) \leq in_n$

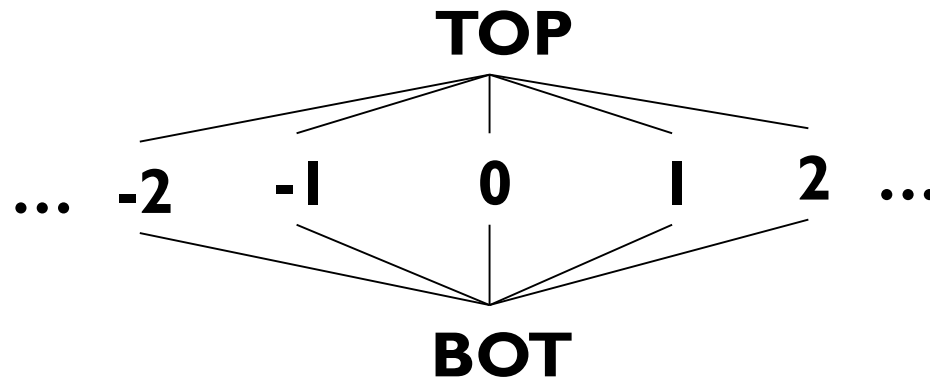
- Proof is by induction on length of  $p$
- Uses monotonicity of transfer function

## Connections between MOP and worklist solution:

- [Kildall, 1973] The iterative worklist algorithm: (1) converges and (2) computes a MFP (maximum fixed point) solution of the set of equations using the worklist algorithm
- [Kildall, 1973] **If  $F$  is distributive, MOP = MFP**  
$$\bigvee \{f_p(\perp) \mid p \text{ is a path to } n\} = in_n$$
- [Kam & Ullman, 1977] If  $F$  is monotone, MOP  $\leq$  MFP

# Lack of Distributivity Example

**Constant Calculator:** Flat Lattice on Integers



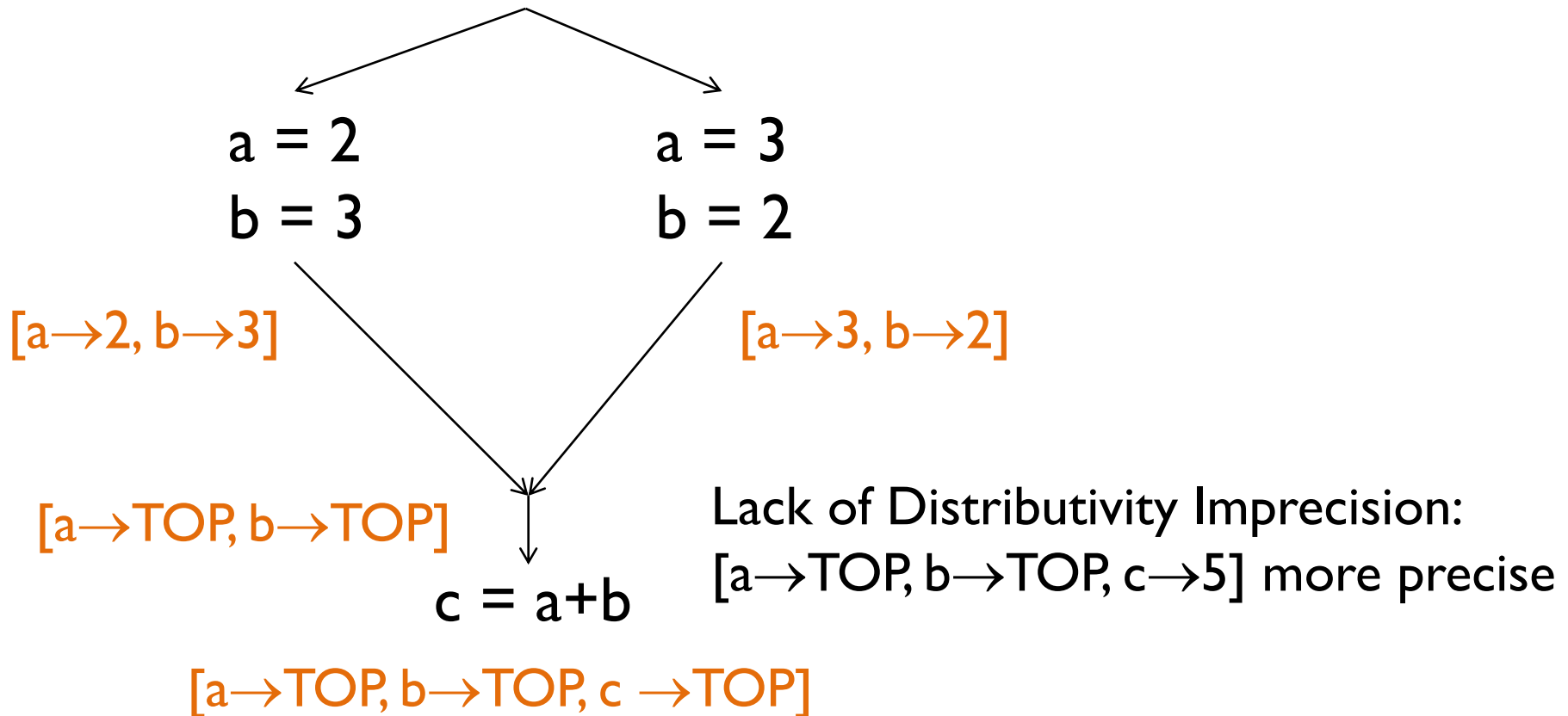
Actual lattice records a value for each variable

- Example element:  $[a \rightarrow 3, b \rightarrow 2, c \rightarrow 5]$

**Transfer function:**

- If  $n$  of the form  $v = c$ , then  $f_n(x) = x[v \rightarrow c]$
- If  $n$  of the form  $v_1 = v_2 + v_3$ ,  $f_n(x) = x[v_1 \rightarrow x[v_2] + x[v_3]]$

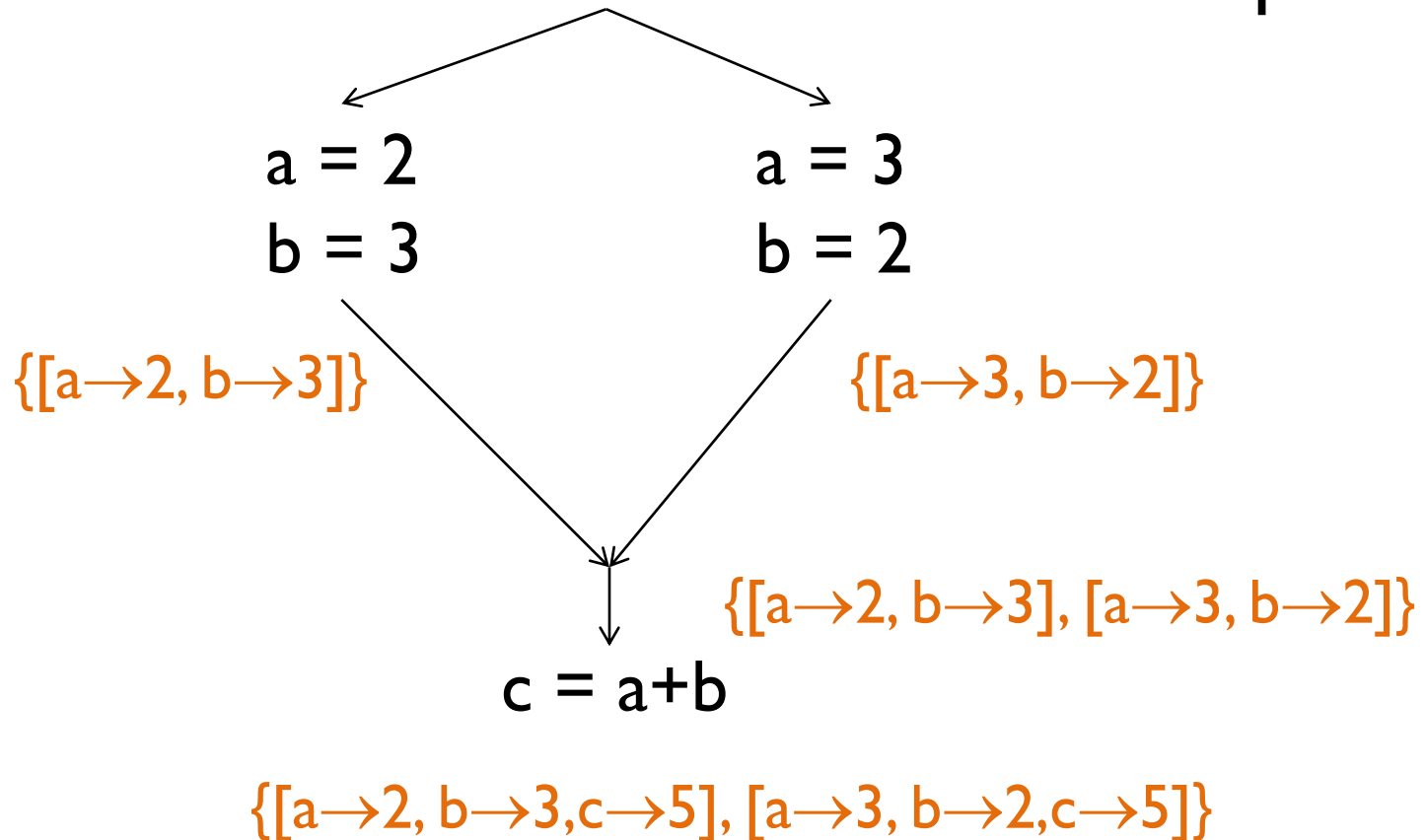
# Lack of Distributivity Anomaly



*What is the meet over all paths solution?*

# Make Analysis Distributive

Keep combinations of values on different paths





# Discussion of the Solution

It basically simulates **all combinations** of values in **all executions**

- Exponential blowup
- Nontermination because of infinite ascending chains

Terminating solution:

- Use widening operator to eliminate blowup (can make it work at granularity of variables)
- However, loses precision in many cases
- Not trivial to select optimal point to do widening

# Look Forward

We will return to these problems later in the semester

- **Interprocedural analysis:** how to handle function calls and global variables in the analysis?
- **Abstract interpretation:** how to automate analysis with infinite chains and rich abstract domains?

## Additional readings:

- Long comparison: Flemming Nielson; Hanne R. Nielson; Chris Hankin. Principles of Program Analysis (2004). Springer. (available online)
- Short comparison: Wolfgang Woegerer. A Survey of Static Program Analysis Techniques (available online)