CS 526

Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526

DATAFLOW ANALYSIS

The slides adapted from 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 {<ptr,target>} 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 p1 to pn is a sequence of points p1, p2, . . . pn 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 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 $\rightarrow$ Constant Propagation
- Variable Liveness $\rightarrow$ 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

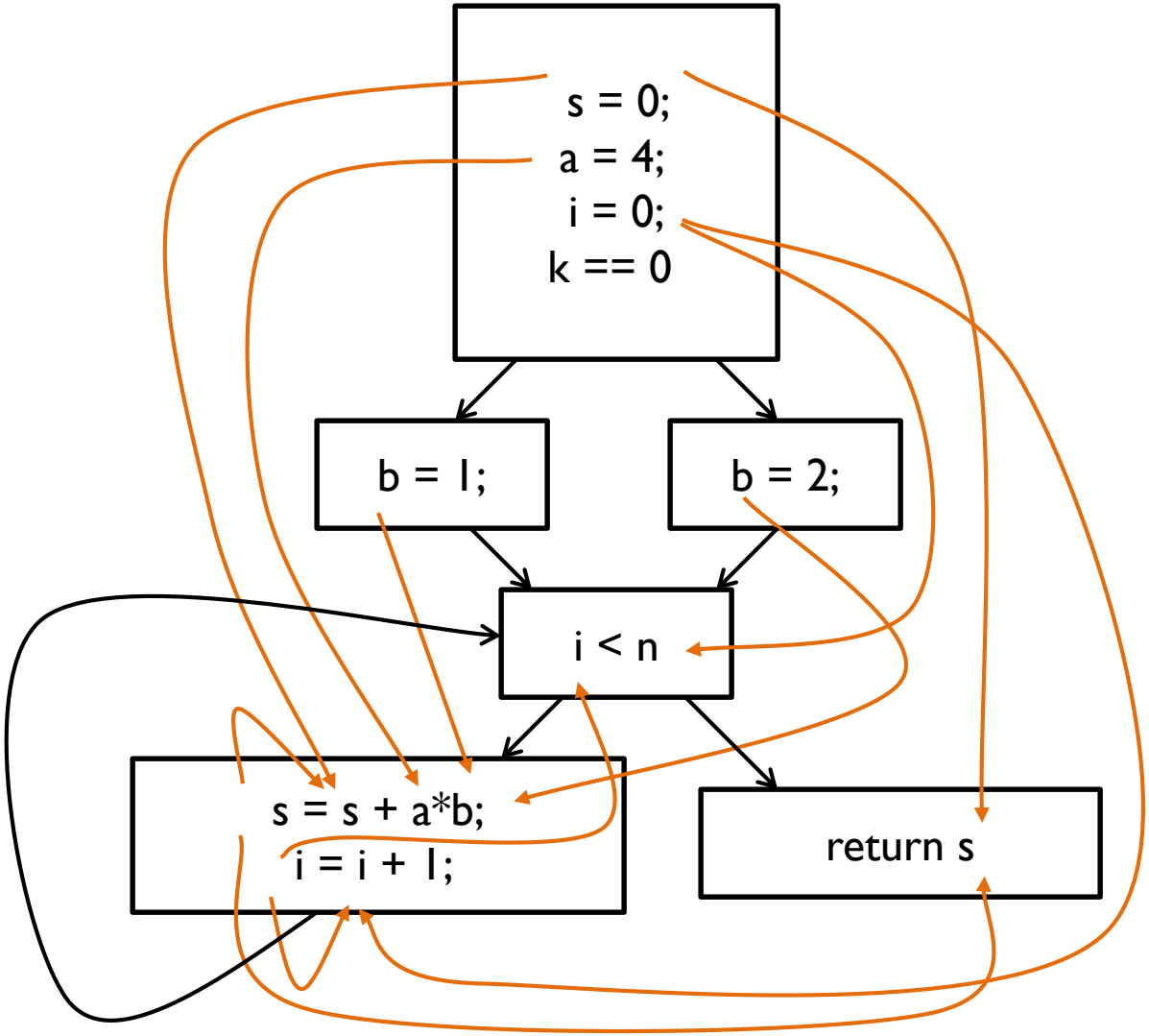
- It is a definition of a
- It is a use of x and y

b = a+1

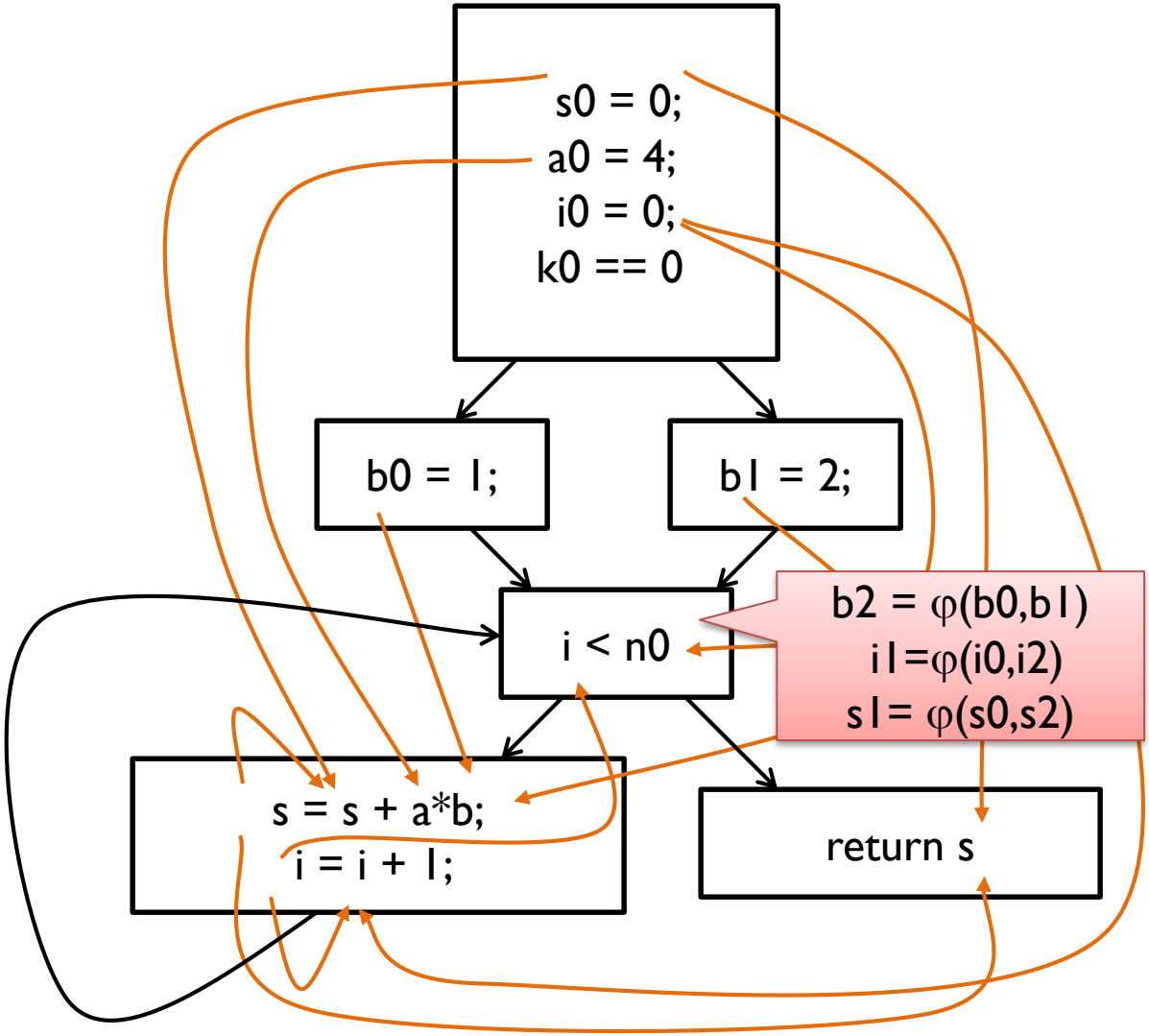
- It is a definition of ? And use of ??

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

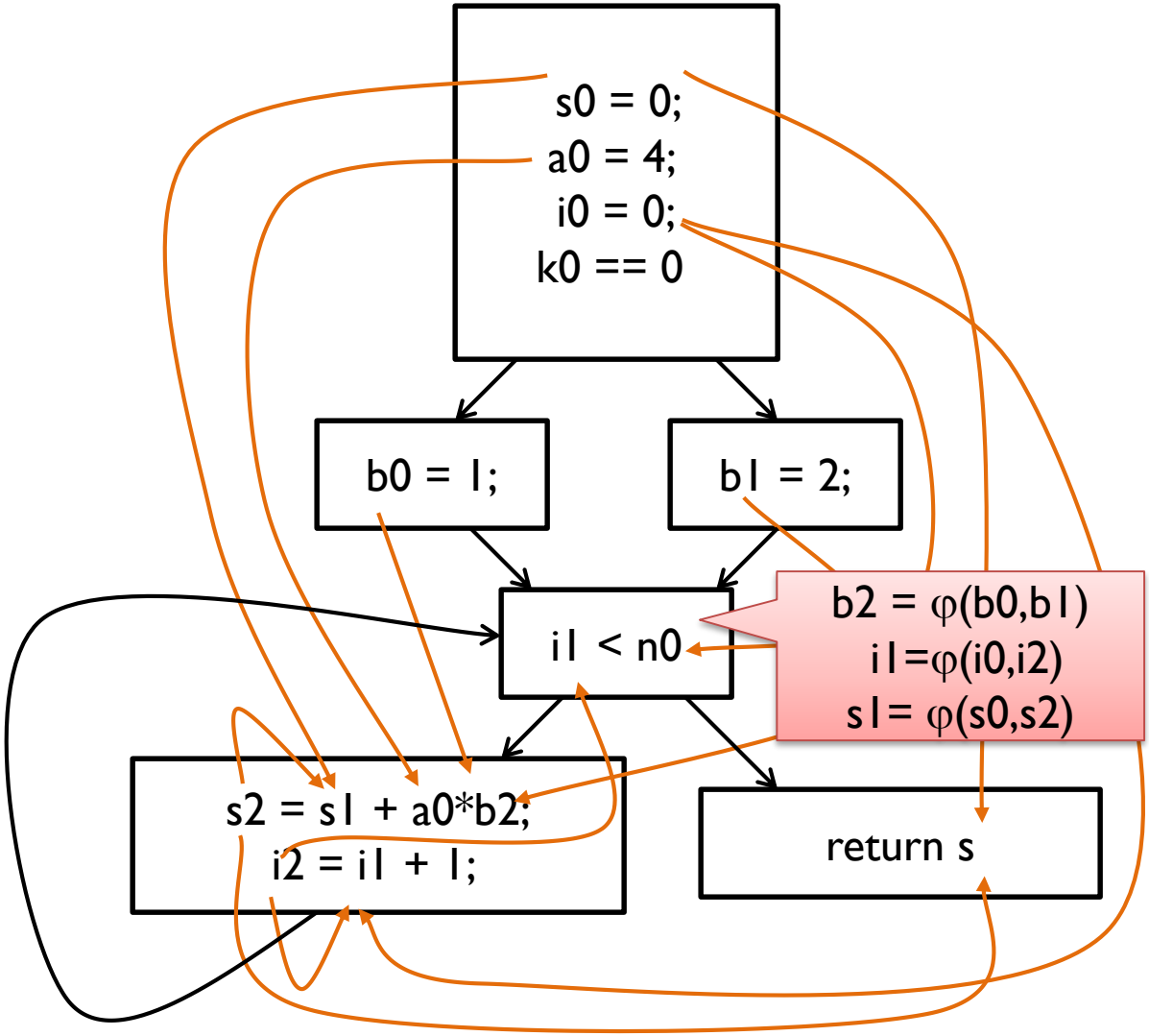
Reaching Definitions



Reaching Definitions



Reaching Definitions



Transform: Constant Propagation

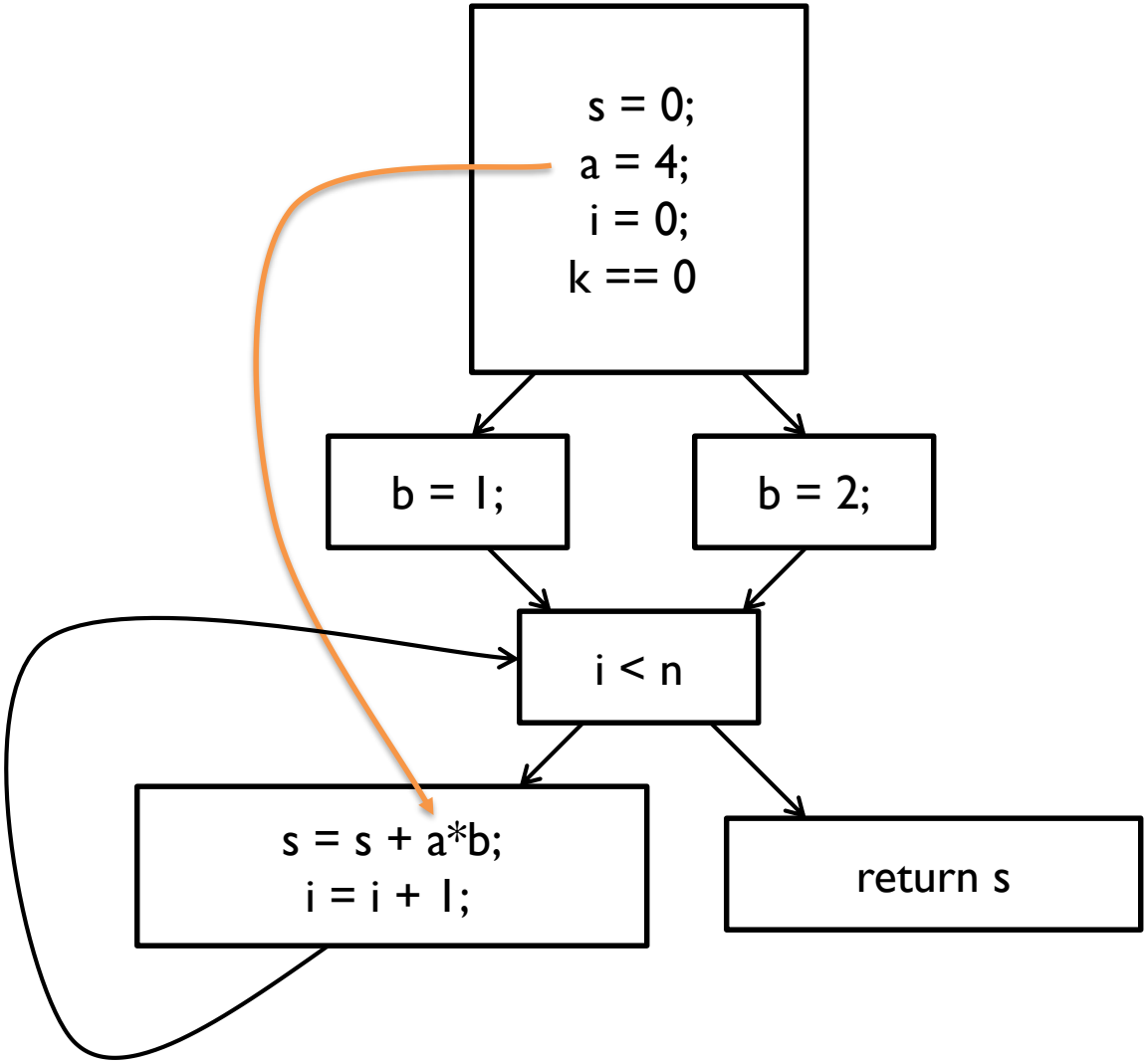
Uses reaching definitions

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

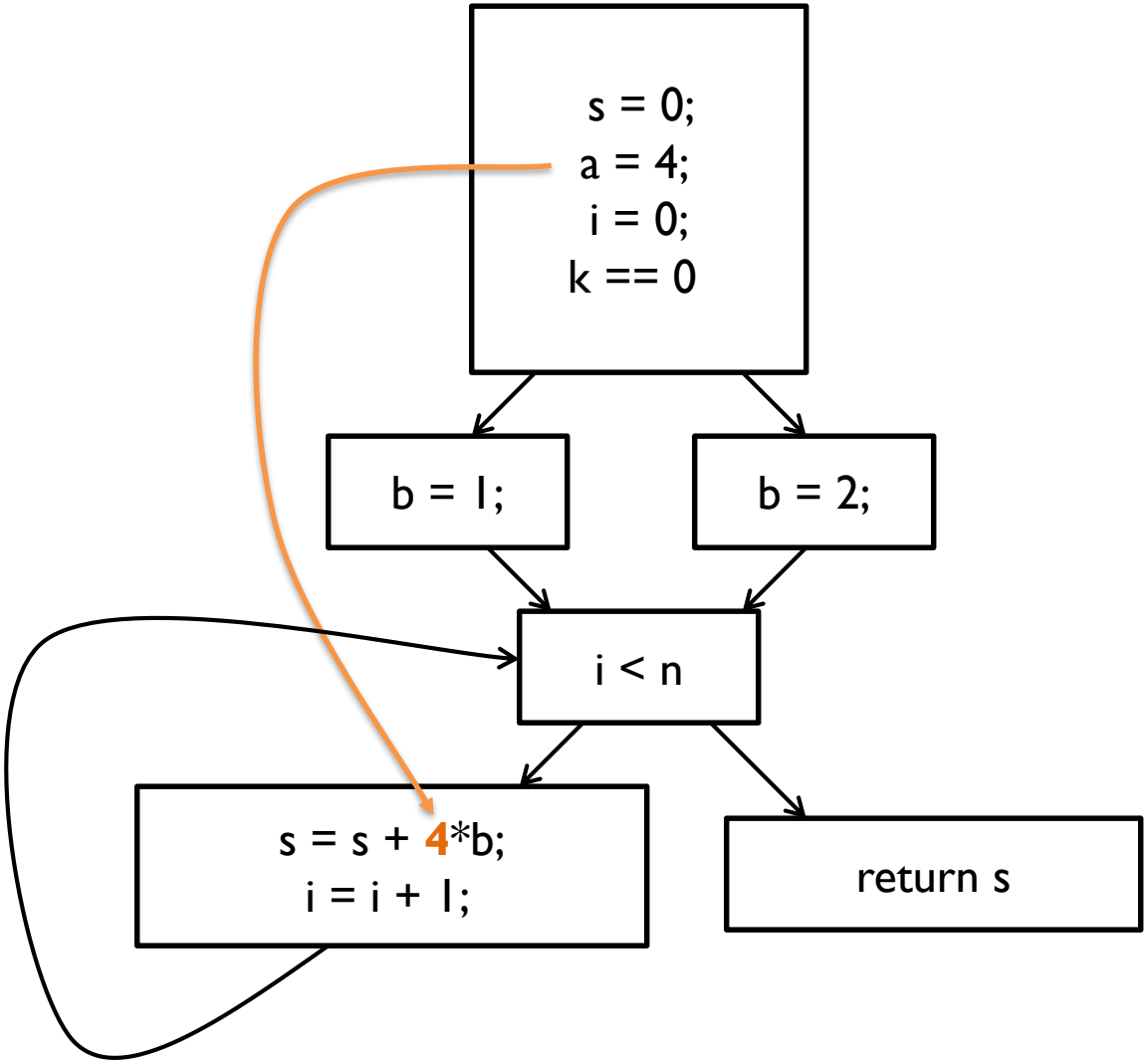
Is a Being Constant in s = s+a*b?



Yes!

a = 4

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



Yes!

a = 4

Reaching Definitions

Dataflow variables (for each block B**)**

In(B) ≡ the set of defs that reach the point before first statement in B

Out(B) ≡ the set of defs that reach the point after last statement in B

Gen(B) ≡ the set of defs in B that are not killed in B.

Kill(B) ≡ the set of all defs that are killed in B (i.e., on the path from entry to exit of B, if def d /∈ B; or on the path from d to exit of B, if def d ∈ B).

The difference:

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

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

Computing Reaching Definitions

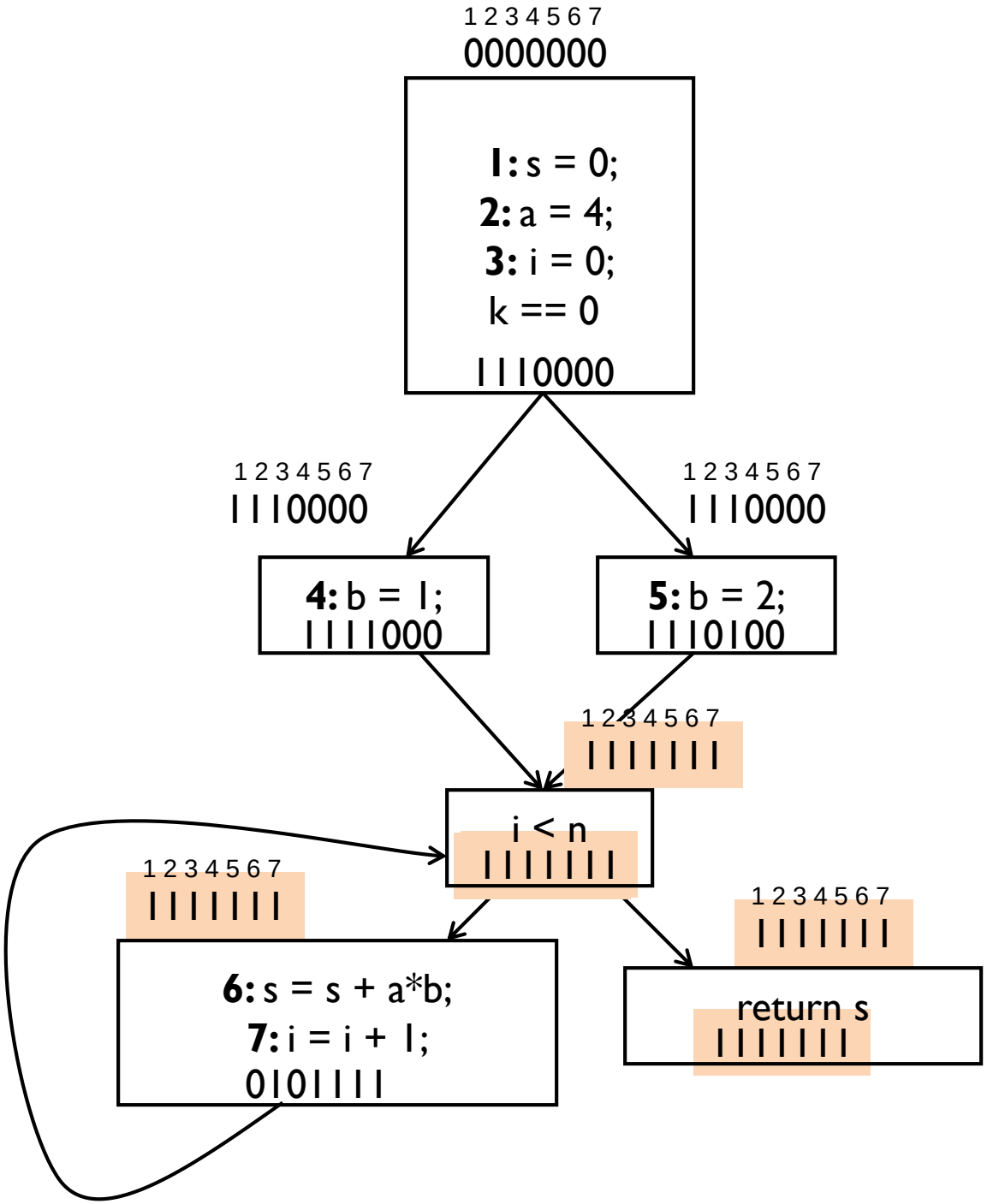
Compute with sets of definitions

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

At each basic block, compute

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

Do computation by simulating execution of program until reach fixed point


1 2 3 4 5 6 7
0000000
1: s = 0;
2: a = 4;
3: i = 0;
k == 0
1 1 1 0000
1 2 3 4 5 6 7
1 1 1 0000
4: b = 1;
1 1 1 1 000
1 2 3 4 5 6 7
1 1 1 0000
5: b = 2;
1 1 1 0 1 00
1 2 3 4 5 6 7
1 1 1 1 1 1 1
i < n
1 1 1 1 1 1 1
1 2 3 4 5 6 7
1 1 1 1 1 1 1
6: s = s + a*b;
7: i = i + 1;
0 1 0 1 1 1 1
1 2 3 4 5 6 7
1 1 1 1 1 1 1
return s
1 1 1 1 1 1 1

Formalizing Analysis

Each basic block has

- **IN** - set of definitions that reach beginning of block
- **OUT** - set of definitions that reach end of block
- **GEN** - set of definitions generated in block
- **KILL** - set of definitions killed in block

GEN[s = s + a*b; i = i + 1;] = 0000011

KILL[s = s + a*b; i = i + 1;] = 1010000

Compiler scans each basic block to derive GEN and KILL sets

Dataflow Equations

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

IN[b] = OUT[b1] U ... U OUT[bn]

- where b1, ..., bn are predecessors of b in CFG

OUT[b] = (IN[b] - KILL[b]) U GEN[b]

IN[entry] = 0000000

Result: system of equations

Solving Equations

Use fixed point algorithm

Initialize with solution of OUT[b] = 0000000

Repeatedly apply equations

- IN[b] = OUT[b1] U ... U OUT[bn]
- OUT[b] = (IN[b] - KILL[b]) U GEN[b]

Until reach fixed point

Until equation application has no further effect

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

Order of the Analysis?

Goal: Propagate information as far as possible in each iteration

Random – Select the next node randomly

Preorder – Select the next node, than explore children in depth-first fashion

Postorder – Before selecting the node, explore all its children

Reverse Postorder – Explore the node, than explore all its children

- Opposite from postorder
- Not the same as preorder!

Reaching Definitions Algorithm

```
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 };
```

Reaching Definitions: Convergence

Out[B] is finite

Out[B] never decreases for any B

$\Rightarrow$ must eventually stop changing

At most n iterations if n blocks

$\Leftarrow$ Definitions need propagate only over acyclic paths

Available Expressions

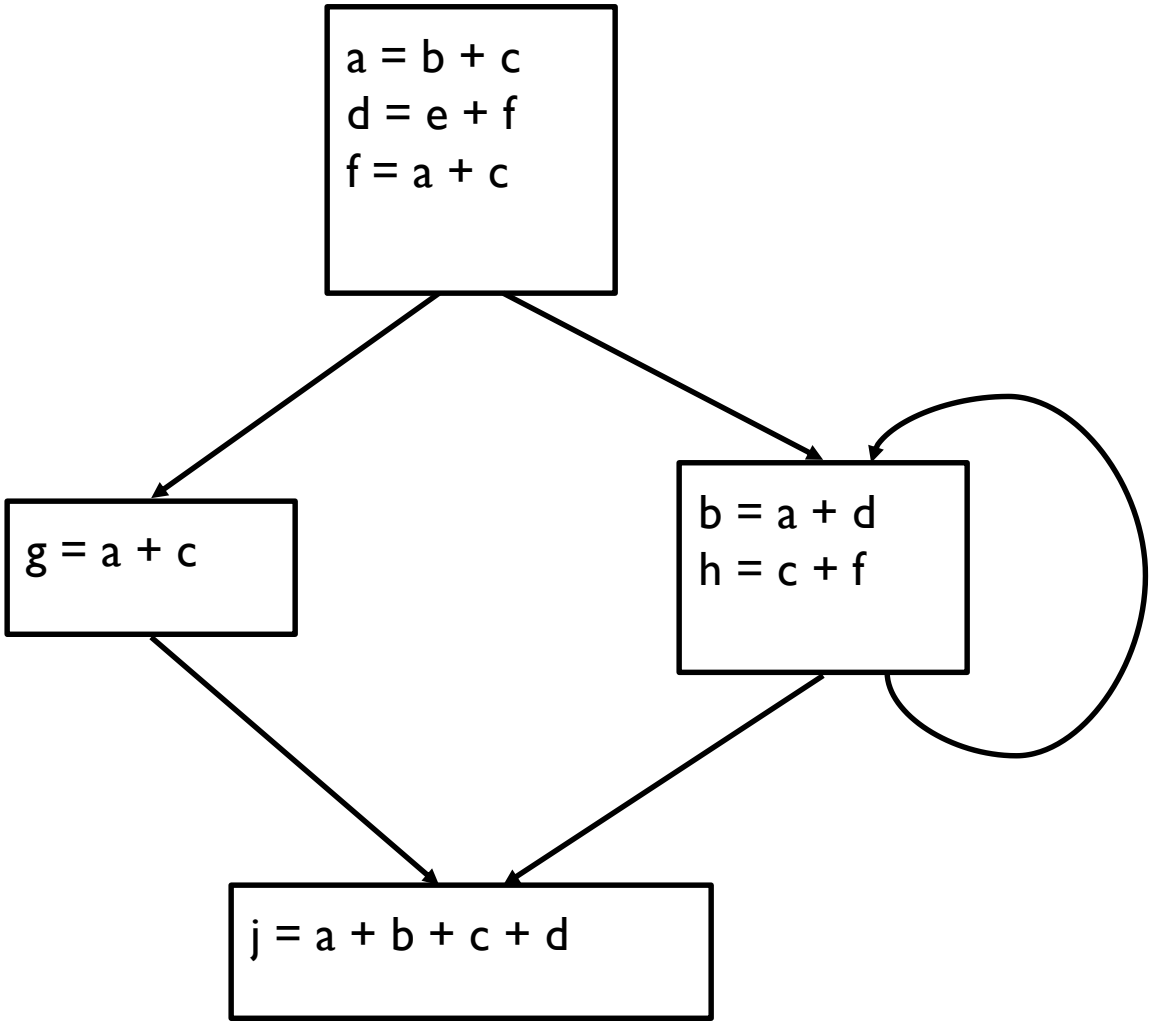
An expression x+y is available at a point p if

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

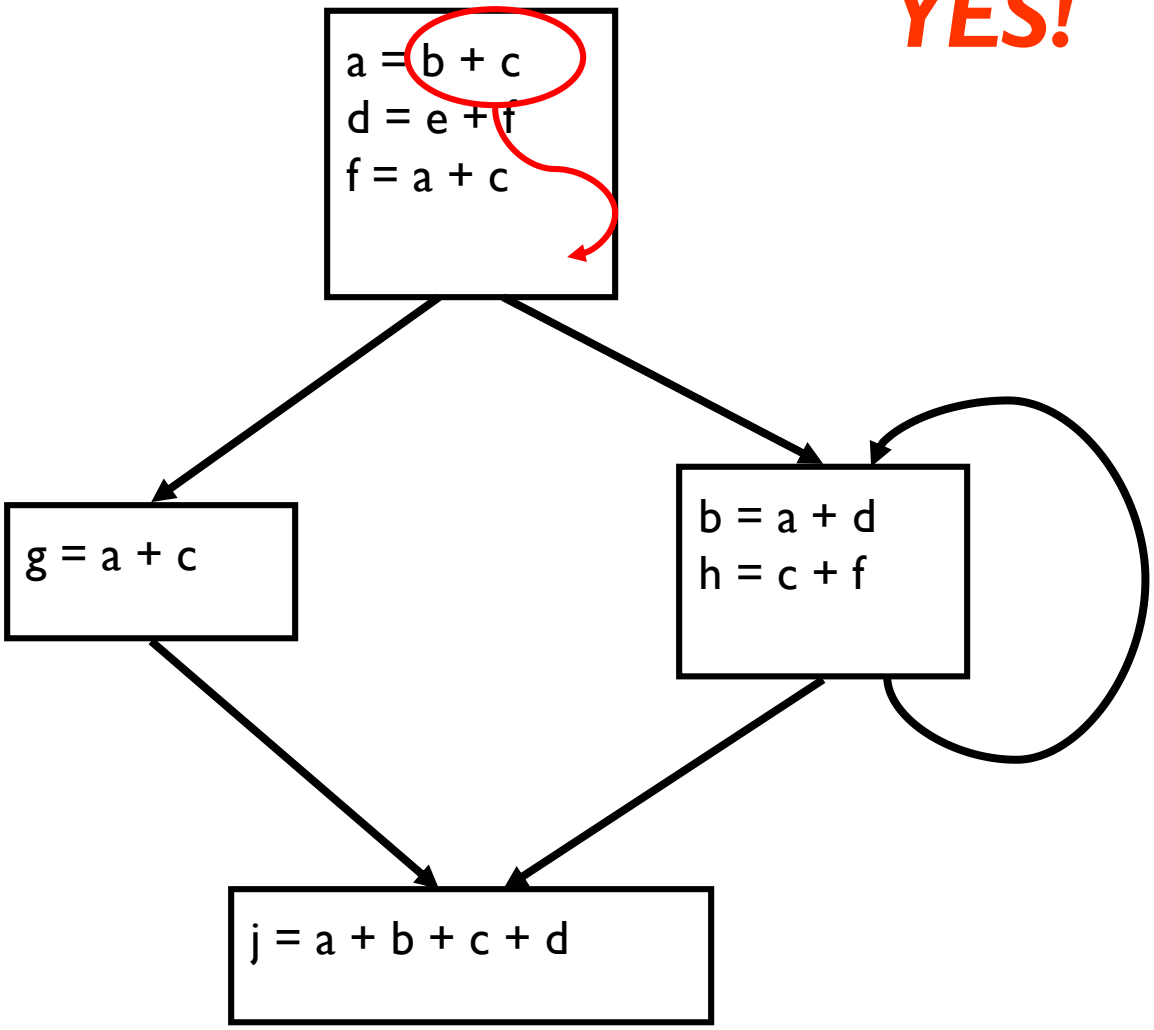
- If expression is available at use, no need to reevaluate it
- Beyond SSA-form analyses

Example: Available Expression



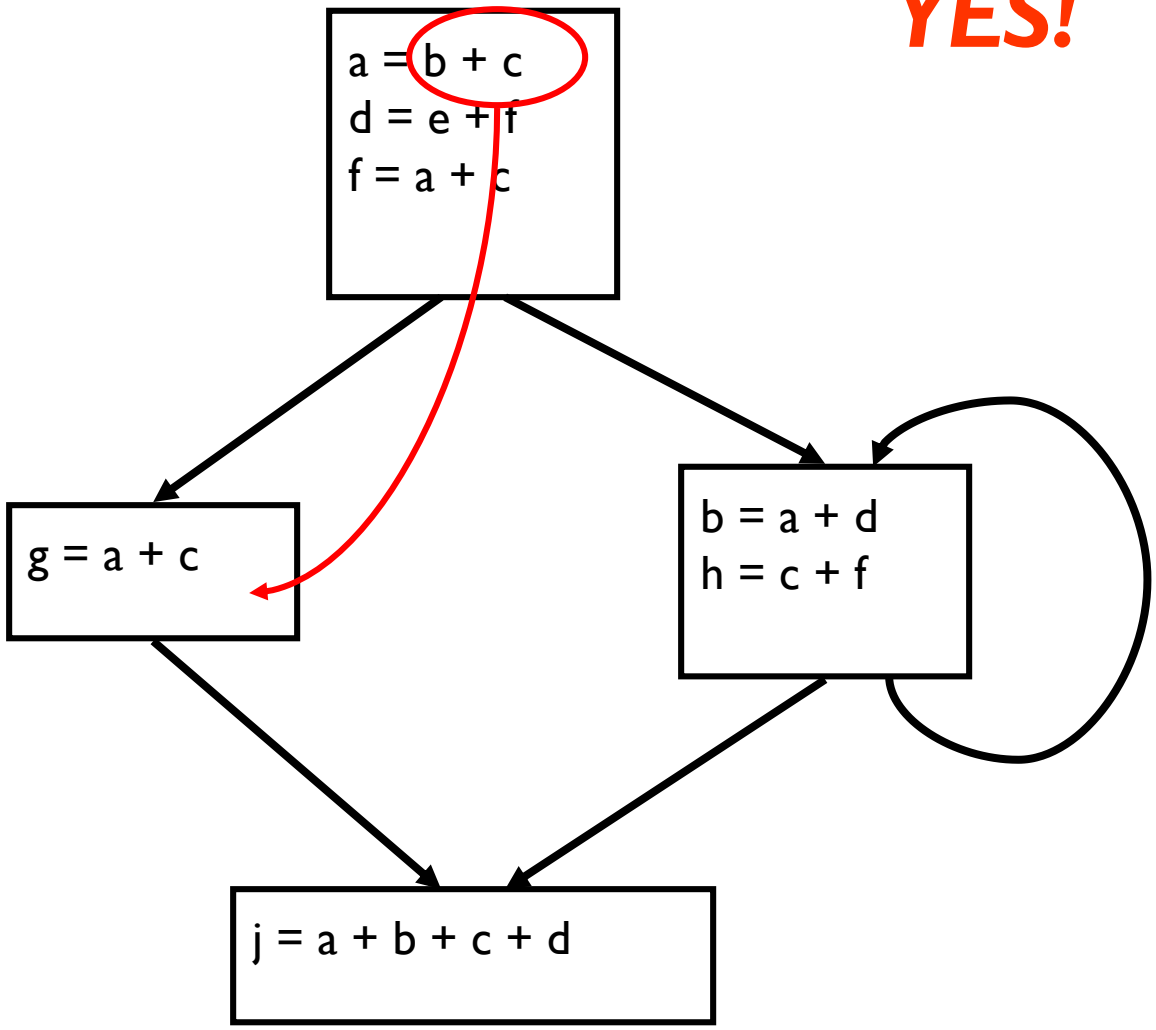
Is the Expression Available?

YES!



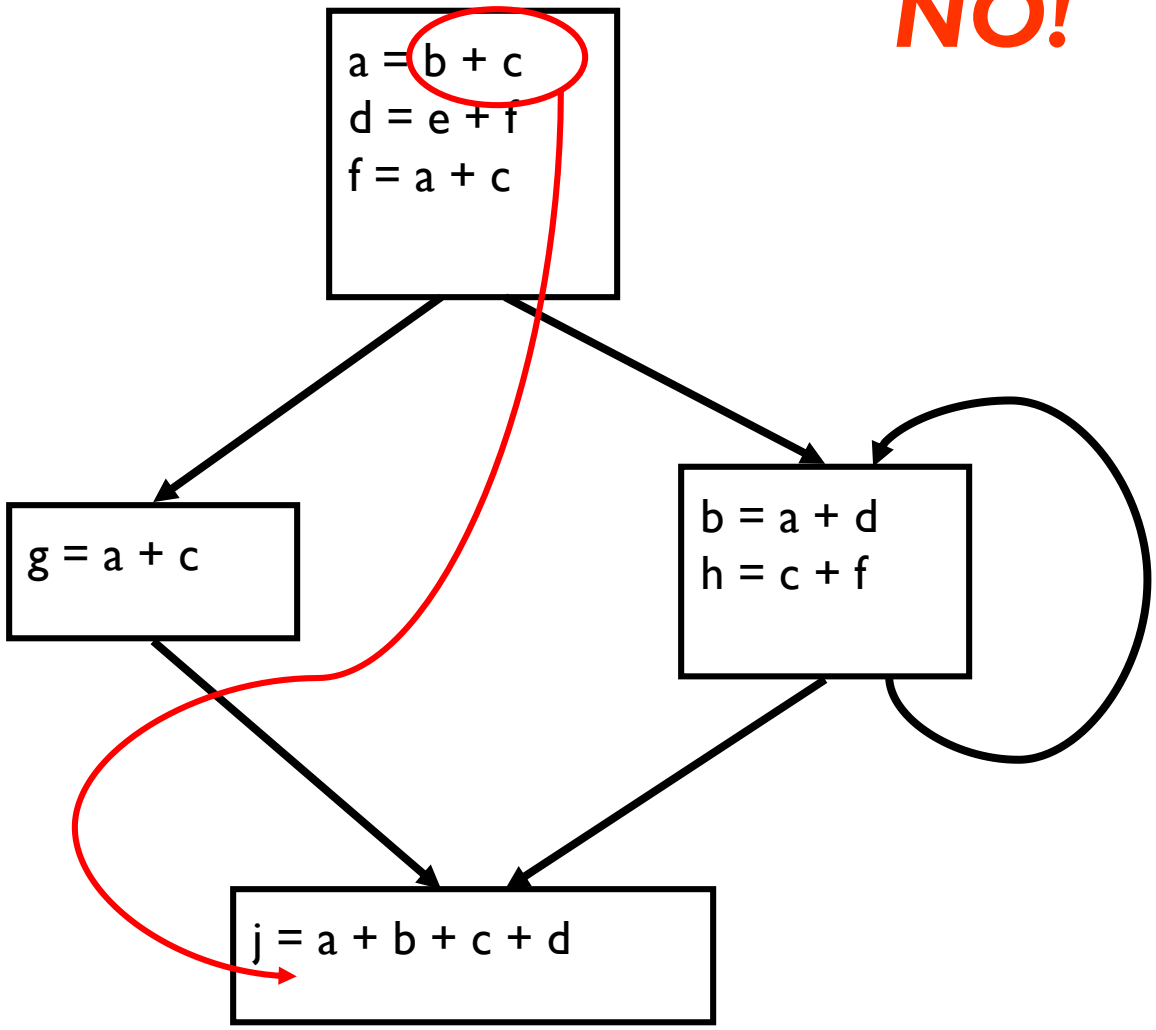
Is the Expression Available?

YES!

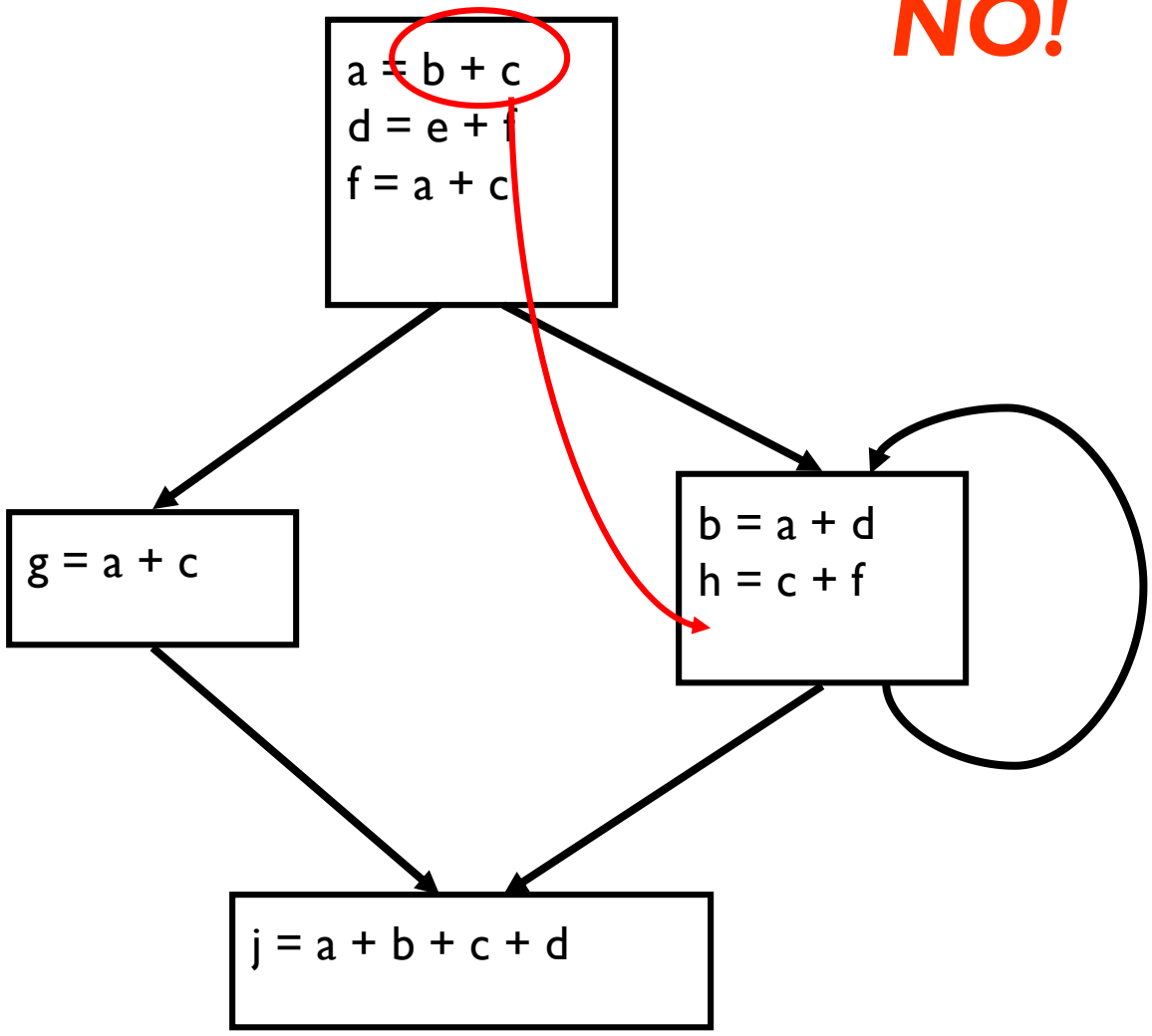


Is the Expression Available?

NO!

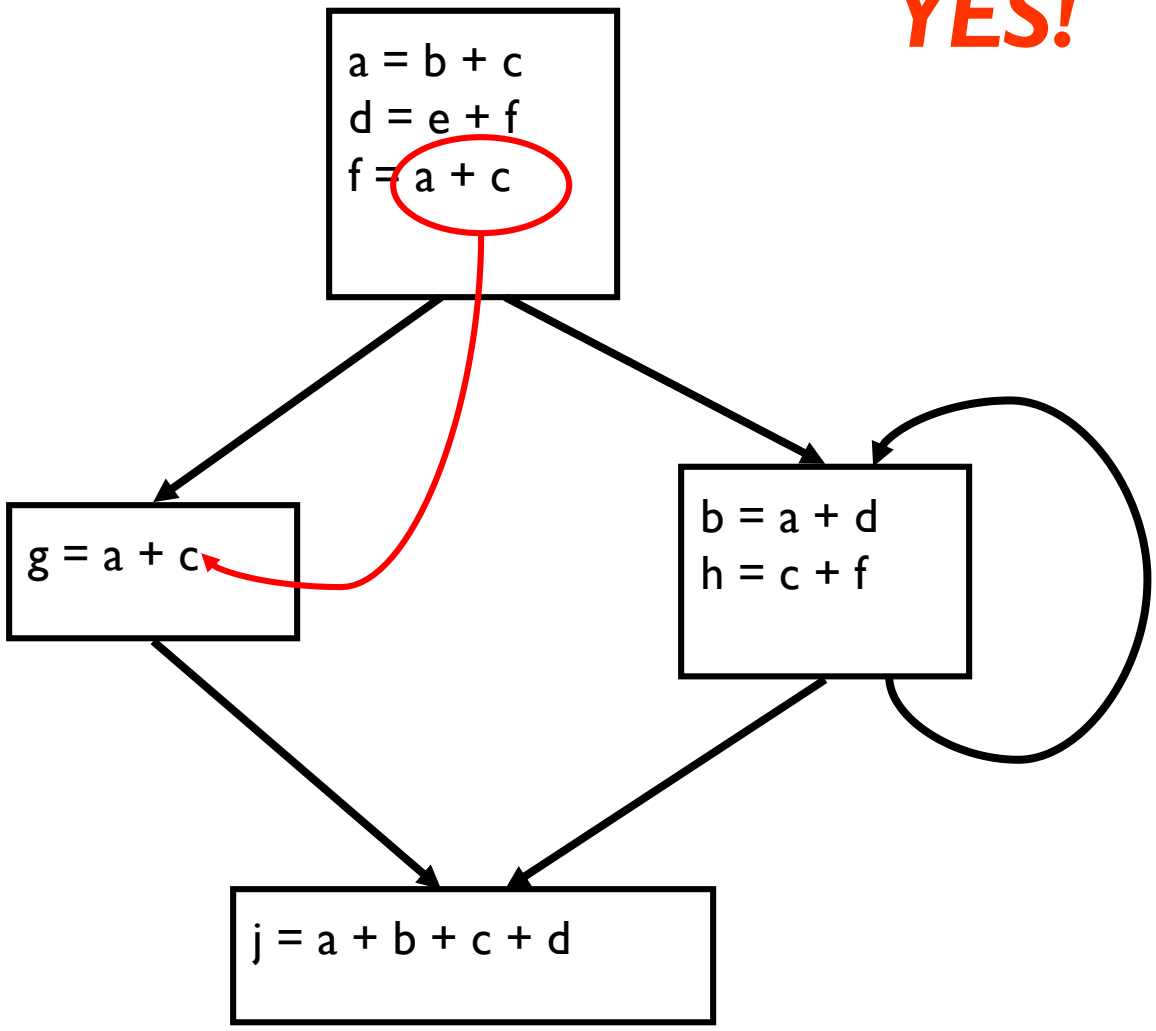


Is the Expression Available?



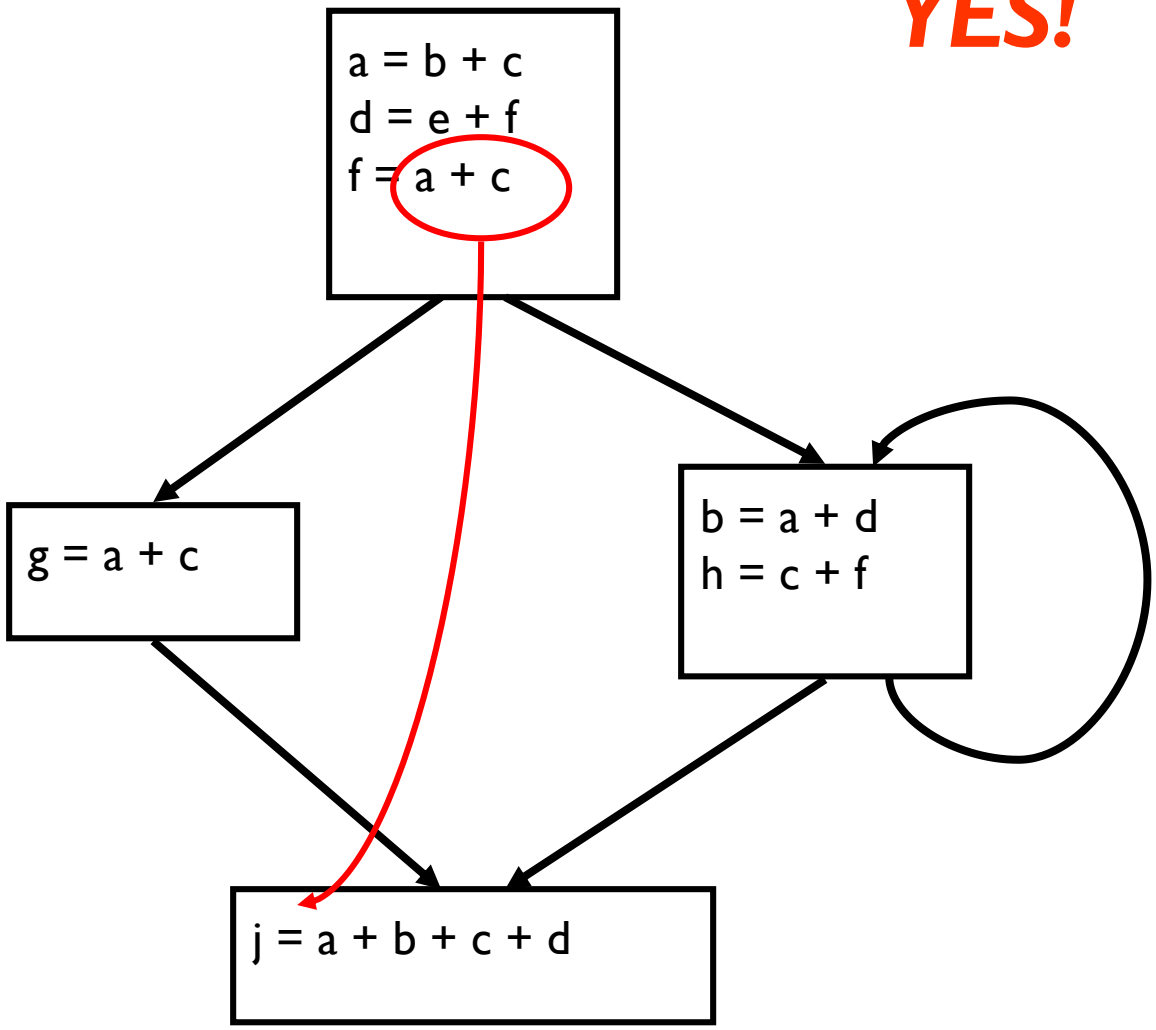
Use of Available Expression

YES!

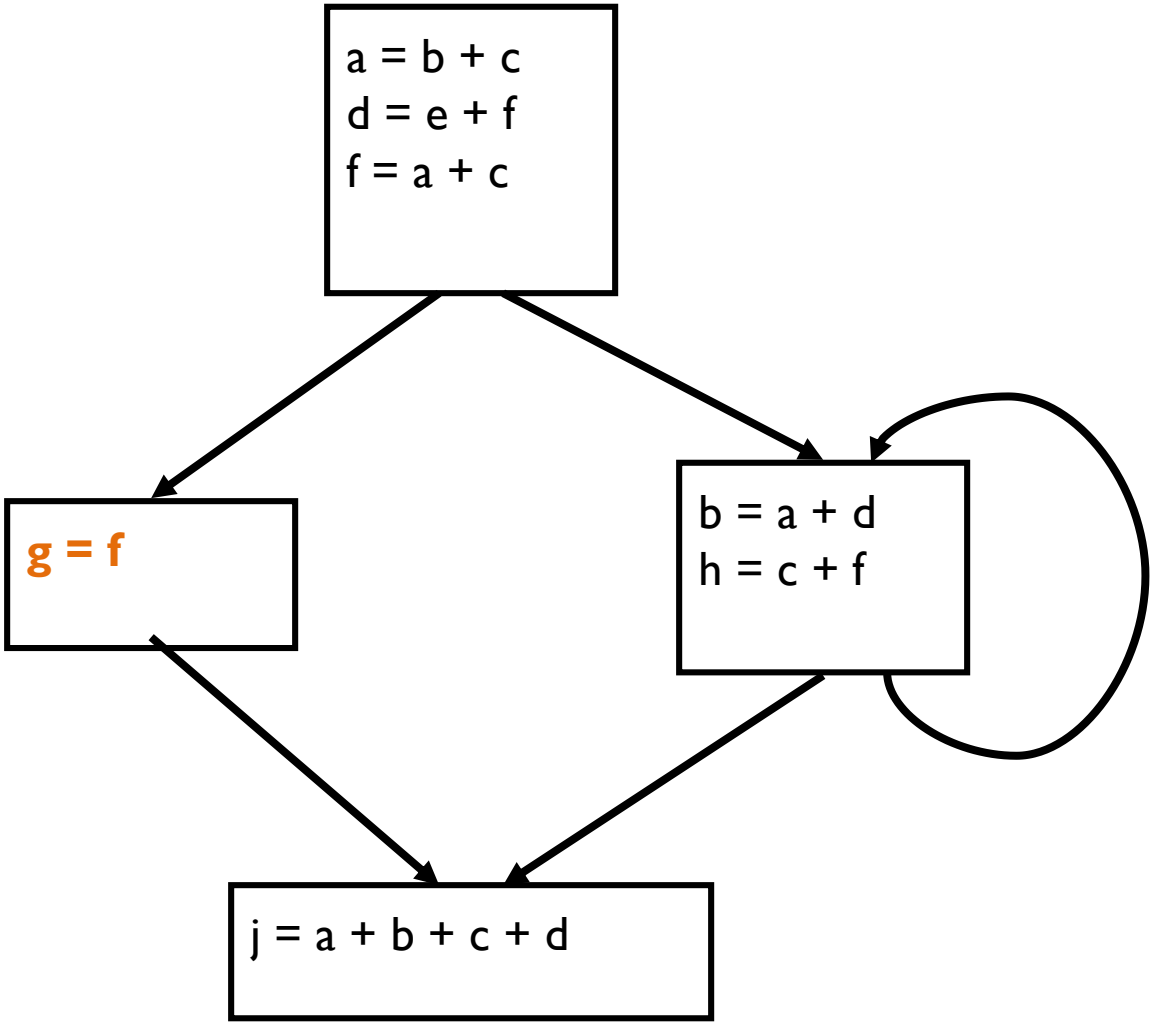


Use of Available Expression

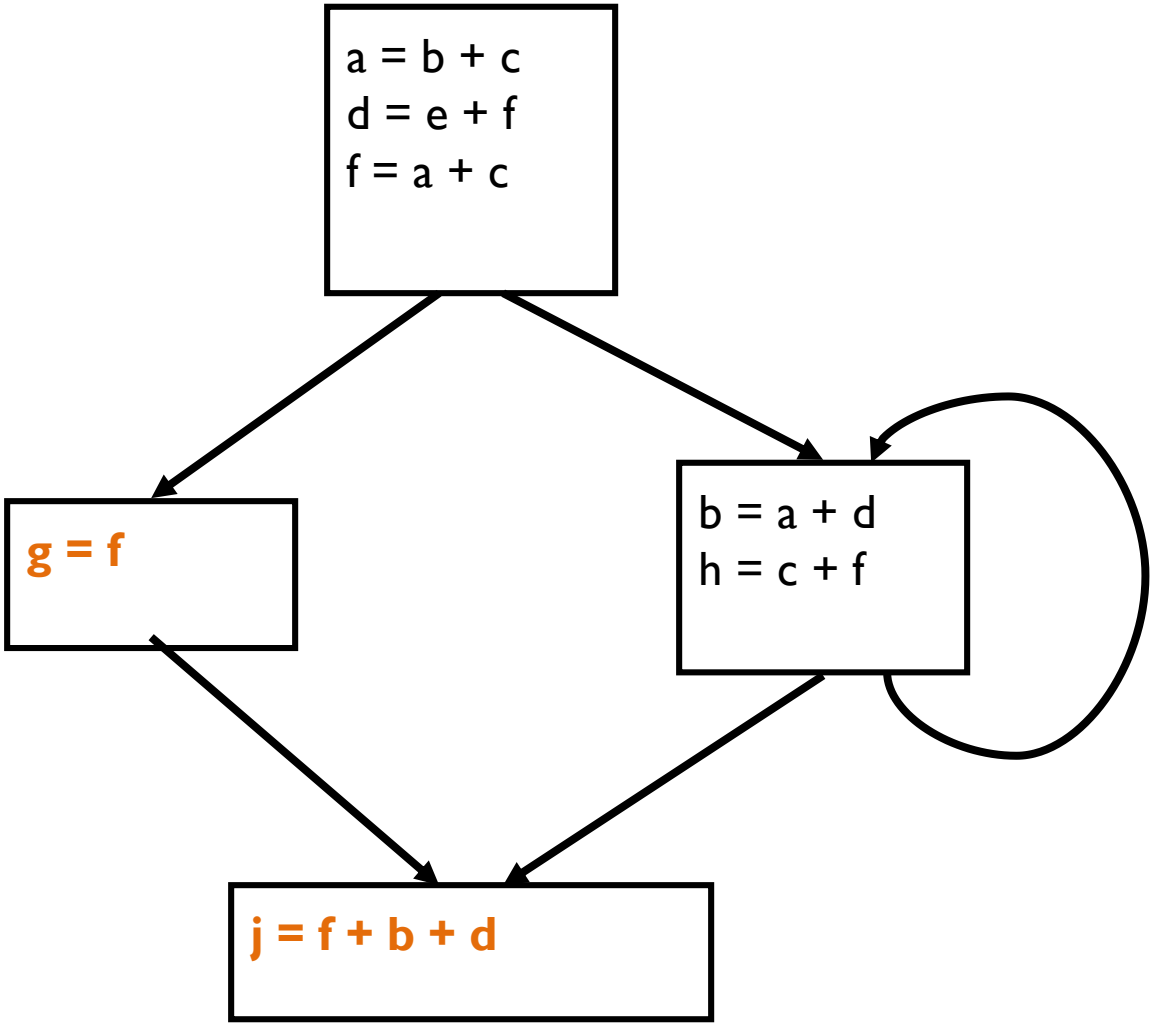
YES!



Use of Available Expression



Use of Available Expression



Compute Available Expressions

Represent sets of expressions using bit vectors

Each expression corresponds to a bit

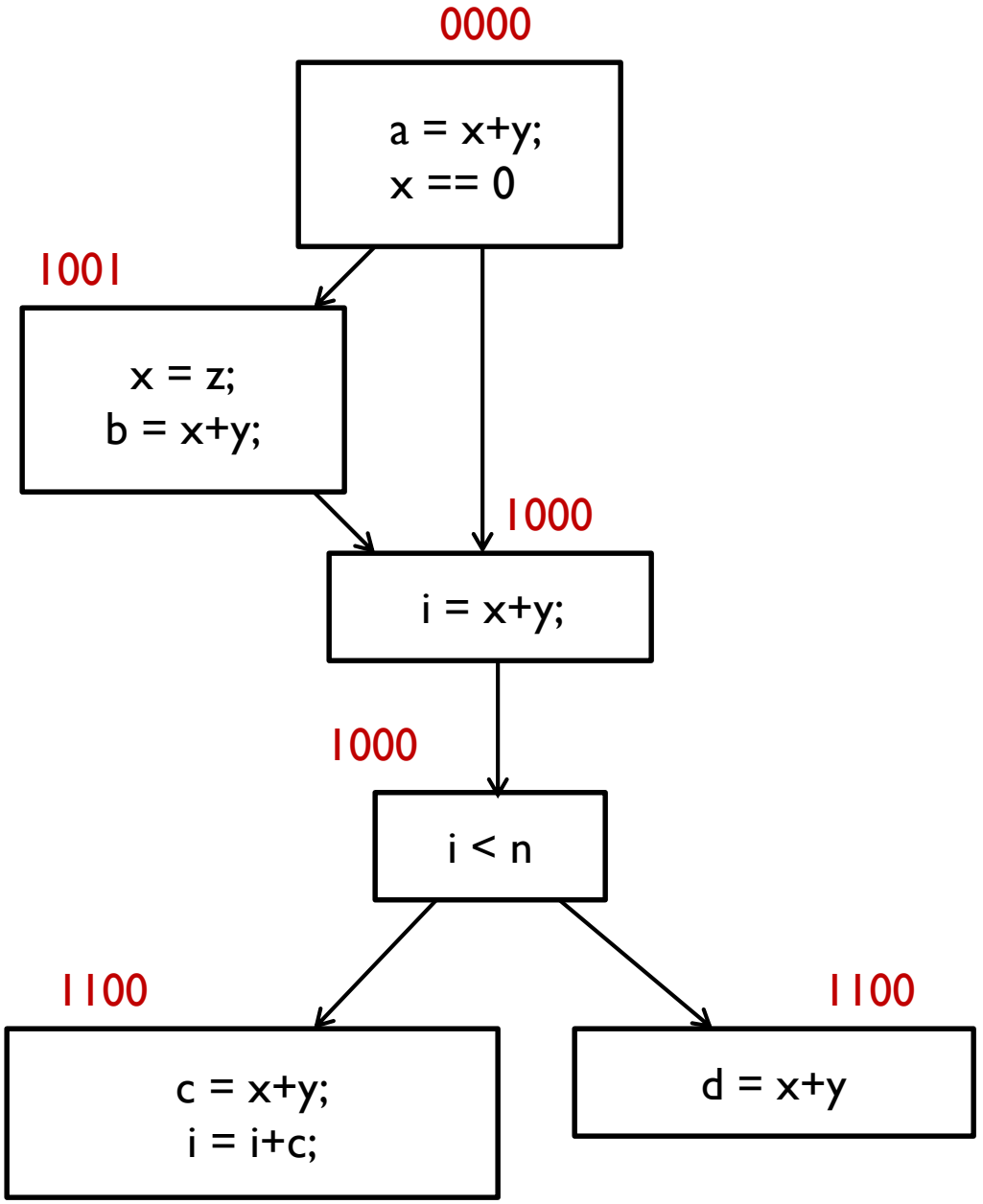
Run dataflow algorithm similar to reaching definitions

Big difference

- 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

Available Expressions

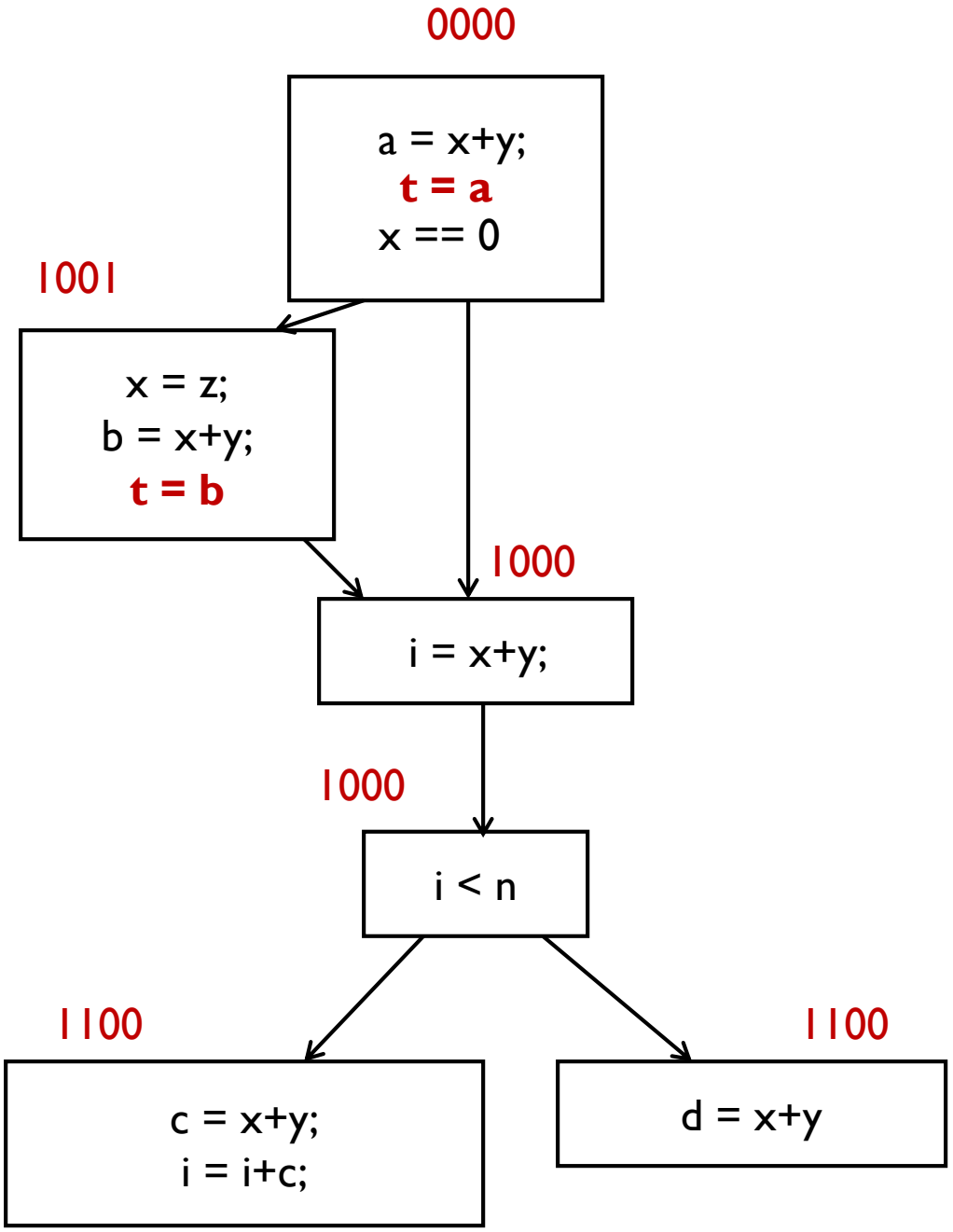
Expressions
1: x+y
2: i<n
3: i+c
4: x==0



Global CSE Transform

Expressions
1: x+y
2: i<n
3: i+c
4: x==0

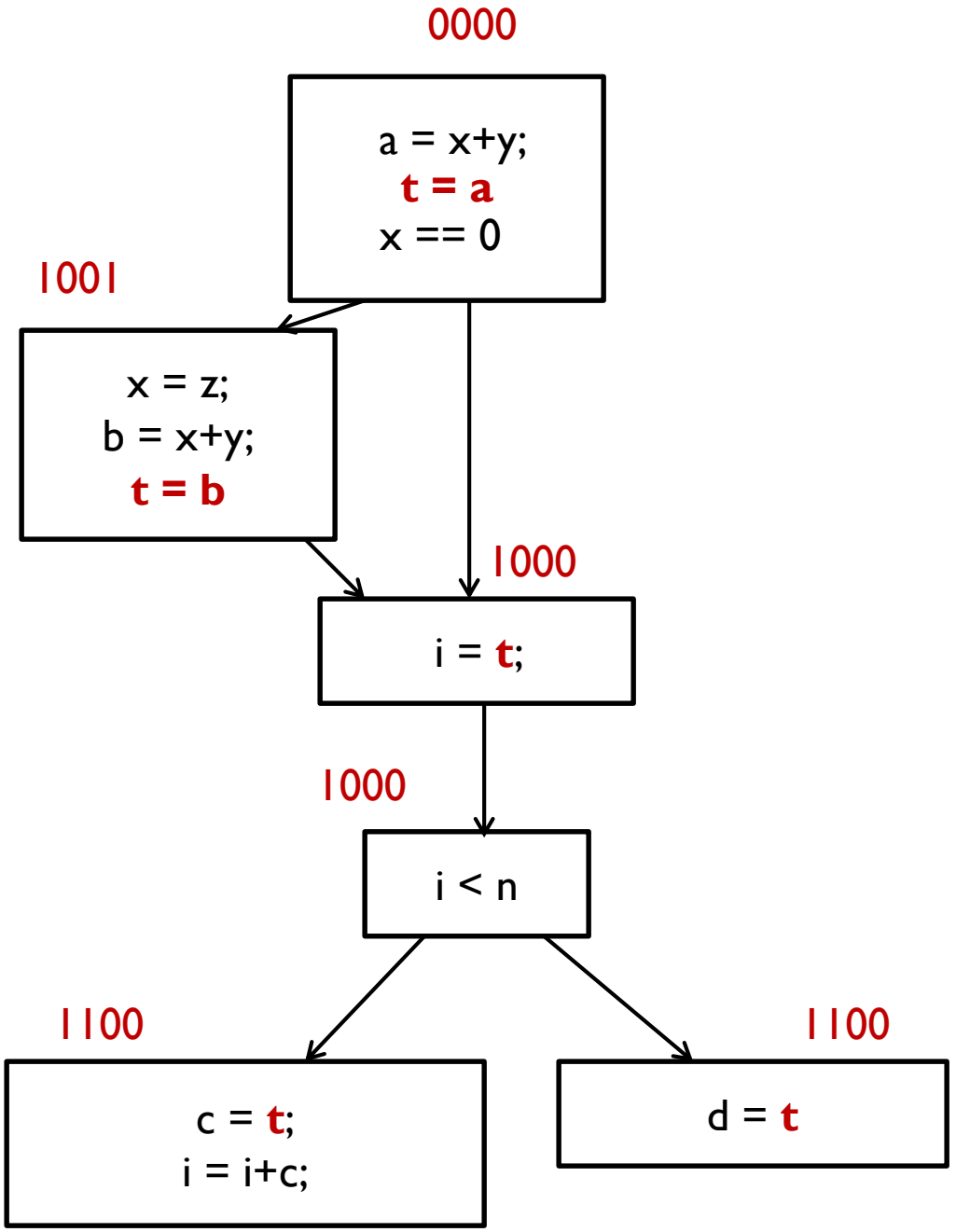
must use same temporary for CSE in all blocks



Global CSE Transform

Expressions
1: x+y
2: i<n
3: i+c
4: x==0

must use same temporary for CSE in all blocks



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

Expressions
1: x+y
2: i<n
3: i+c
4: x==0

- GEN[x = z; b = x+y] = 1000
- KILL[x = z; b = x+y] = 1001

- Compiler scans each basic block to derive GEN and KILL sets

Dataflow Equations

- IN[b] = OUT[b1] ∩ ... ∩ OUT[bn]
  - where b1, ..., bn are predecessors of b in CFG
- OUT[b] = (IN[b] - KILL[b]) U 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[b1] ∩ … ∩ OUT[bn]
  - OUT[b] = (IN[b] - KILL[b]) U 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] = 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] = E; // E is set of all expressions
        for all nodes p in predecessors(n)
                IN[n] = IN[n] ∩ 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 };
```

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?

Variable Liveness Analysis

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

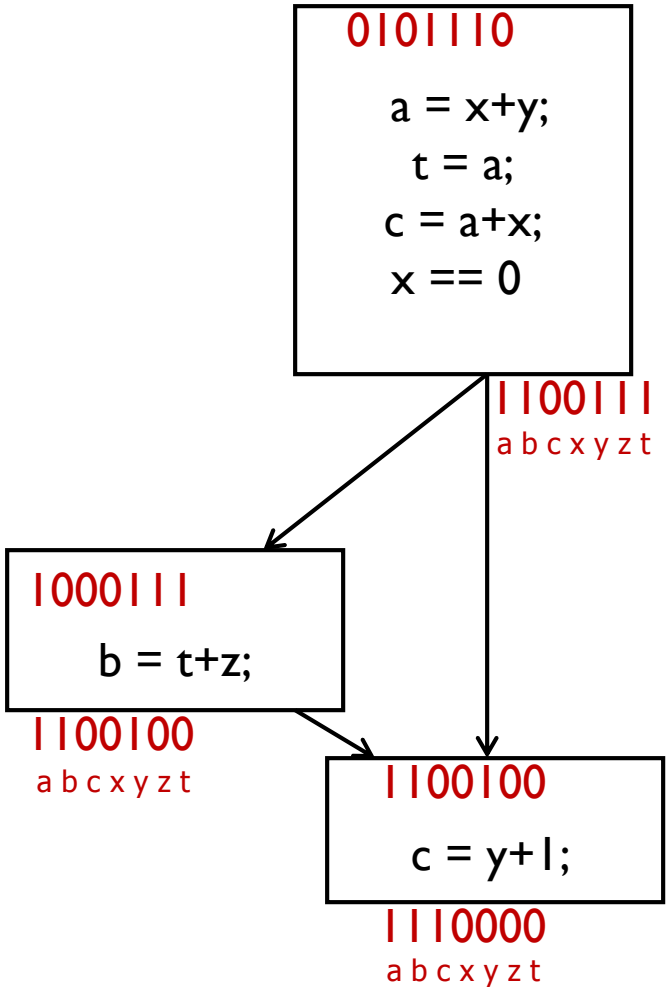
Simulate execution

But start from exit and go backwards in CFG

Compute liveness information from end to beginning of basic blocks

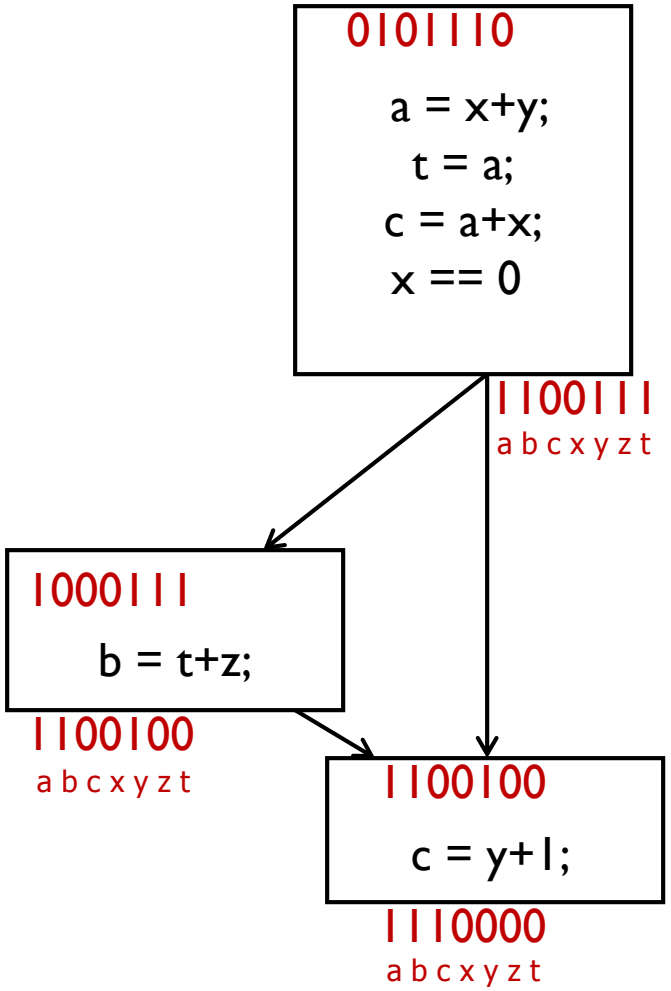
Liveness Example

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



Dead Code Elimination

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



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 - { Exit }
        IN[n] = emptyset;
OUT[Exit] = emptyset;
IN[Exit] = use[Exit];
Changed = N - { 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] U IN[p];

        IN[n] = use[n] U (out[n] - def[n]);

        if (IN[n] changed)
                for all nodes p in predecessors(n)
                        Changed = Changed U { 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 - { 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] 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 };
```

Available Expressions

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

Liveness

```
for all nodes n in N - { Exit }
    IN[n] = emptyset;
OUT[Exit] = emptyset;
IN[Exit] = use[Exit];
Changed = N - { 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] U IN[p];

    IN[n] = use[n] U (out[n] - def[n]);

    if (IN[n] changed)
        for all nodes p in predecessors(n)
            Changed = Changed U { p };
```

Comparison

Reaching Definitions

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

Available Expressions

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

Comparison

Reaching Definitions

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

Liveness

```
for all nodes n in N
    IN[n] = emptyset;
OUT[Exit] = emptyset;
IN[Exit] = use[Exit];
Changed = N - { 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] U IN[p];

    IN[n] = use[n] U (out[n] - def[n]);

    if (IN[n] changed)
        for all nodes p in predecessors(n)
            Changed = Changed U { p };
```

WHY?

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]

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 $x = z$

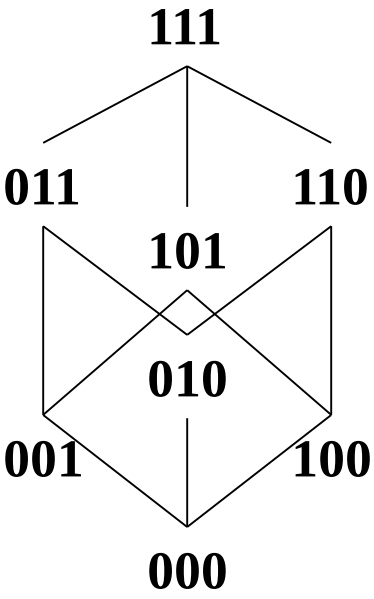
Conceptually, y covers x if there are no elements between x and y

Example

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

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

$x \leq y$ **if** (x bitwise and y) = x



Hasse Diagram

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

Lattices

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

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

All finite lattices are complete

Example of a lattice that is not complete

- Integers I
- For any $x, y \in I$, $x \vee y = \max(x,y)$, $x \wedge y = \min(x,y)$
- But $\vee$ I and $\wedge$ I do not exist
- $I \cup \{+\infty, -\infty\}$ 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 ($\bot$)

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

The following 3 properties are equivalent:

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

Will 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, 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 $x = x \wedge y$

$x = x \wedge (x \vee y)$ (by absorption)

$= x \wedge y$ (by assumption)

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

$y = y \vee (y \wedge x)$ (by absorption)

$= y \vee (x \wedge y)$ (by commutativity)

$= y \vee x$ (by assumption)

$= x \vee y$ (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 $x \vee z = z$

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

$= (x \vee y) \vee z$ (by associativity)

$= y \vee z$ (by assumption)

$= z$ (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., $(x \vee y) \vee u = u$

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

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

$= (x \vee y) \vee u$ (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 $l \leq x \wedge y$, 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 $x_1 \leq x_2 \leq \ldots$ there exists n such that $x_n = x_{n+1} = \ldots$