CS 526

Advanced Compiler Construction

https://charithm.web.illinois.edu/cs526/sp2024/
(slides adapted from Sasa and Vikram)

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 {<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 points 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 (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 $\rightarrow$ 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:

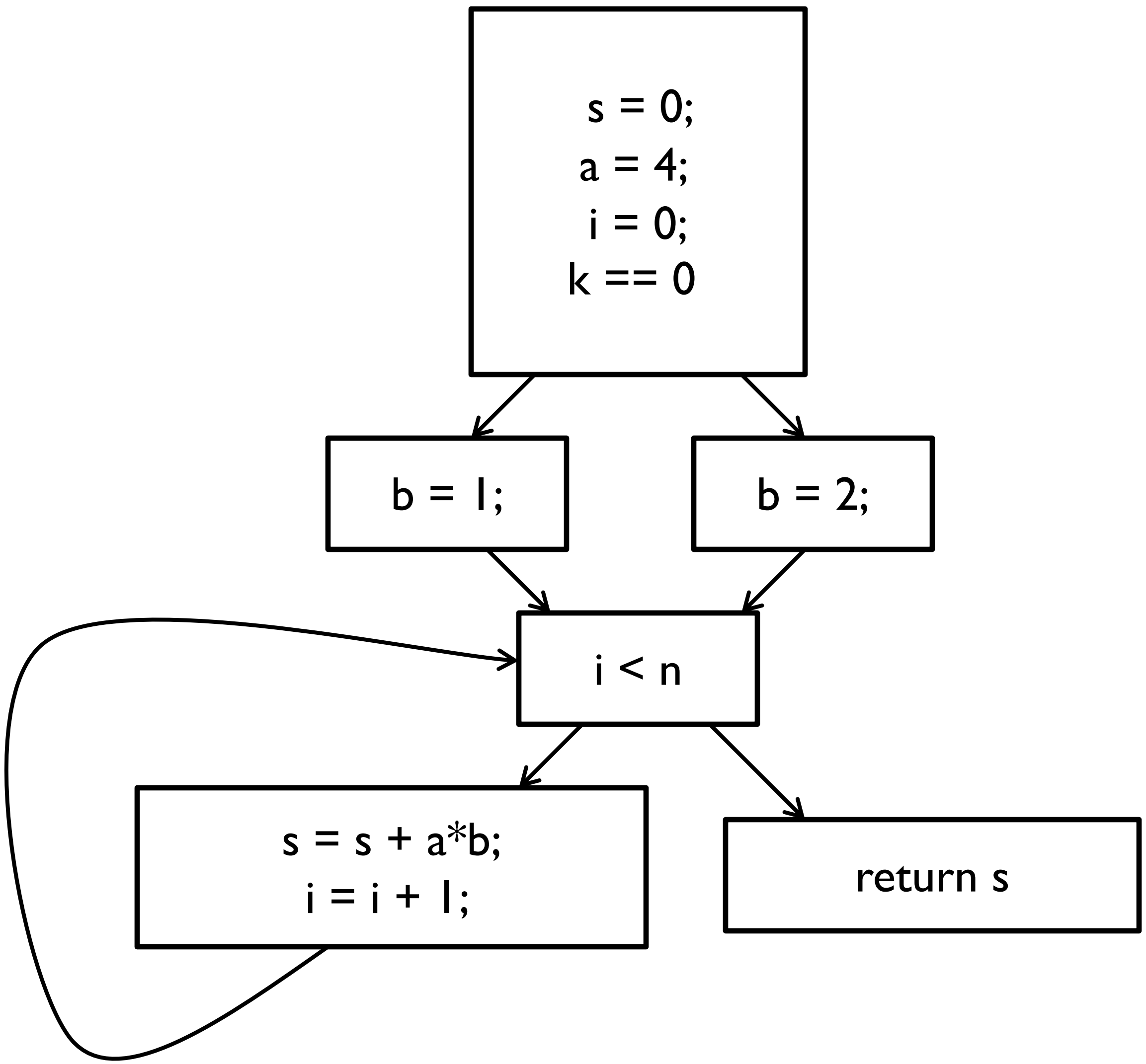
a = x+y

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

b = a+1

- It is a definition of b And use of a

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


s = 0;
a = 4;
i = 0;
k == 0
b = 1;
b = 2;
i < n
s = s + a*b;
i = i + 1;
return s

Reaching Definitions (Declarative)

Dataflow variables (for each block B)

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

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

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

Kill(B) ≡ the set of all definitions that are killed in B, i.e.,

1. on the path from entry to exit of B, if definition d $\notin$ B; or
2. on the path from d to exit of B, if definition 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

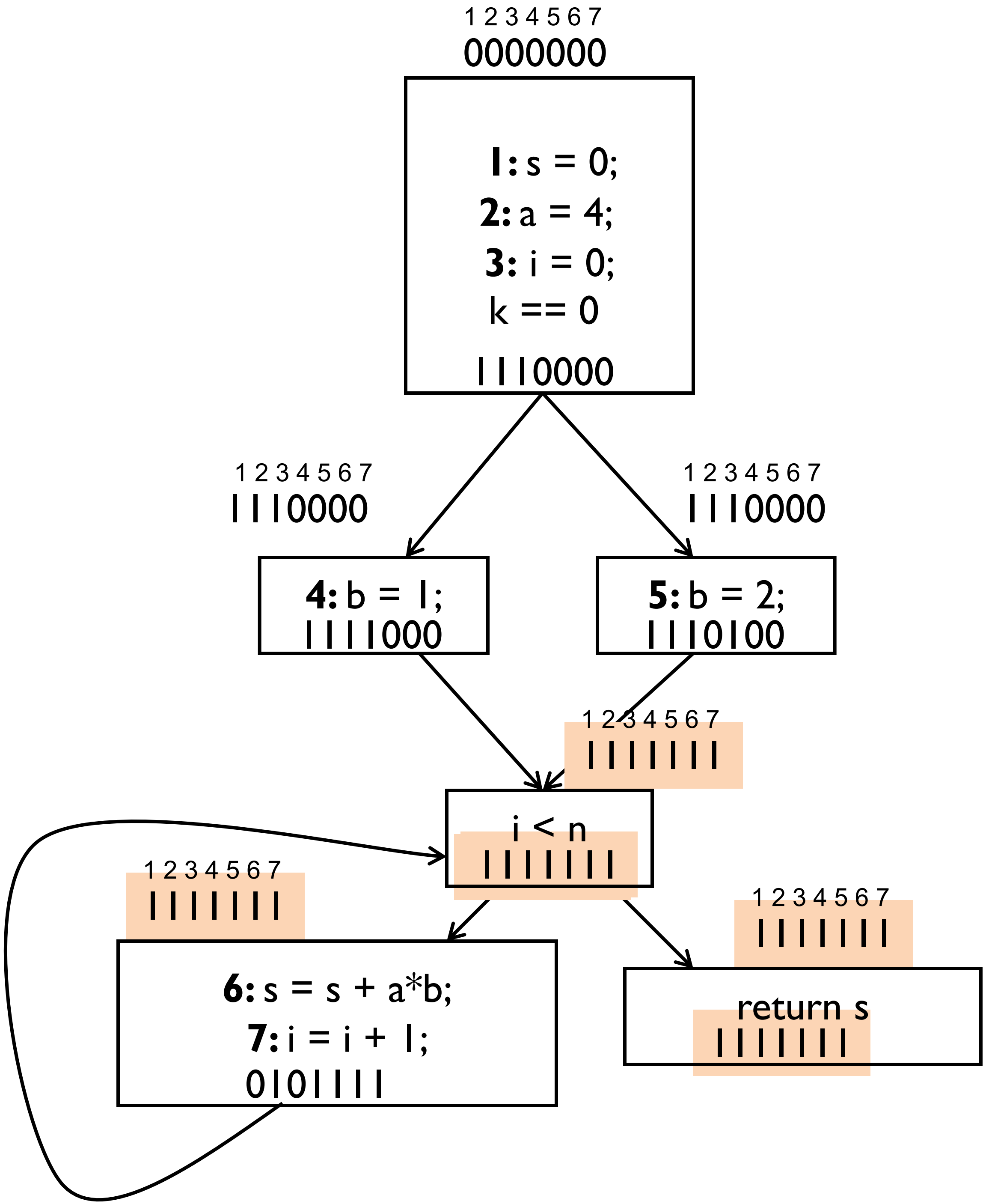
Compute with sets of definitions

- 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


1 2 3 4 5 6 7
0000000
1: s = 0;
2: a = 4;
3: i = 0;
k == 0
1110000
1 2 3 4 5 6 7
1110000
1 2 3 4 5 6 7
1110000
4: b = 1;
1111000
5: b = 2;
1110100
1 2 3 4 5 6 7
1111111
i < n
1111111
1 2 3 4 5 6 7
1111111
1 2 3 4 5 6 7
1111111
6: s = s + a*b;
7: i = i + 1;
0101111
return s
1111111

Formalizing the analysis: 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 (worklist) algorithm

Initialize with solution of OUT[b] = 0000000

- **Repeatedly apply equations**
  1. **IN[b] = OUT[b1] U ... U OUT[bn]**
  2. **OUT[b] = (IN[b] - KILL[b]) U 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] = 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 efficient impl. these are bitvector operations

    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 to propagate only over acyclic paths

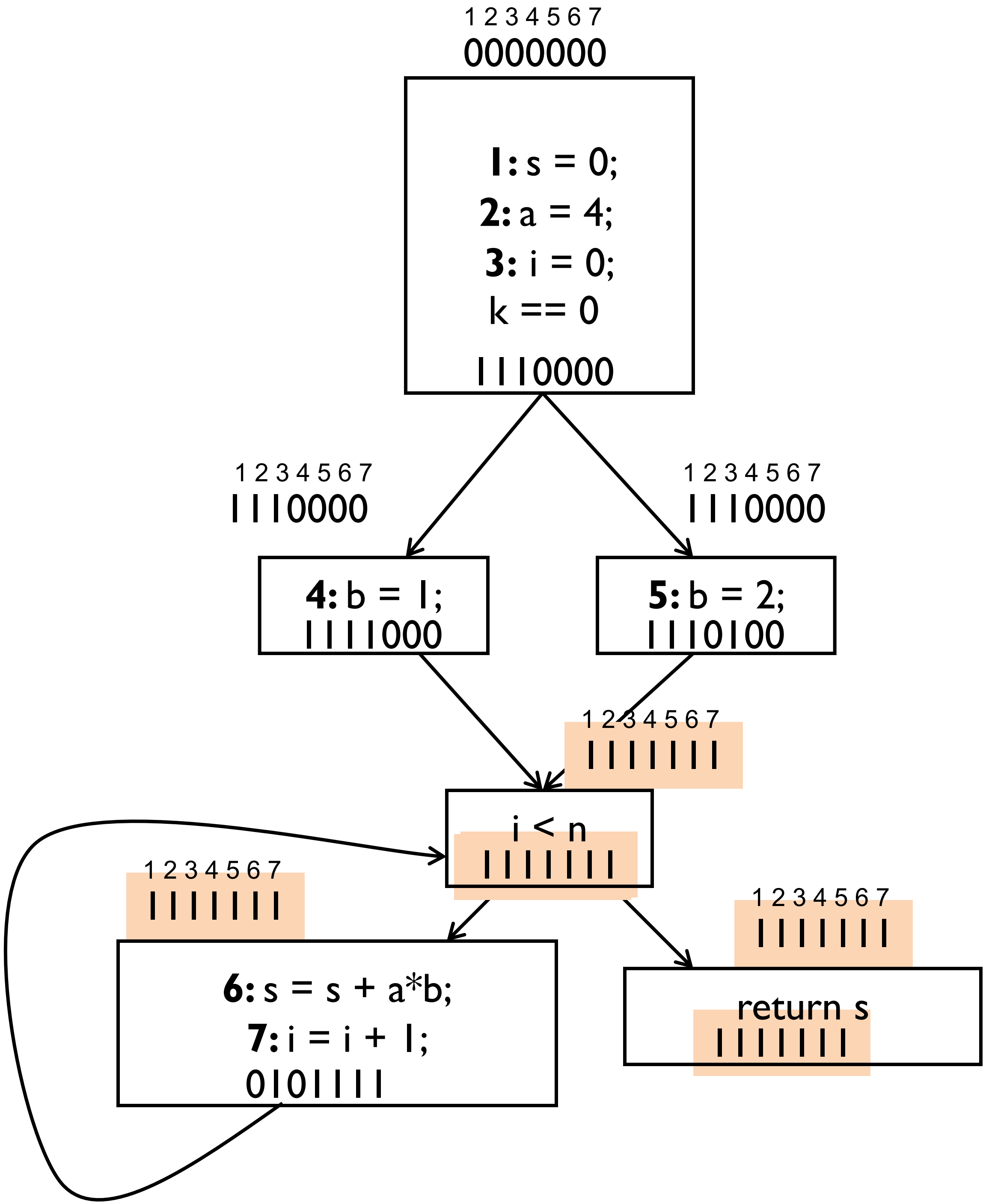
Transform: Constant Propagation

Paired with reaching definitions (uses its results)

Check: Is a use of a variable a constant?

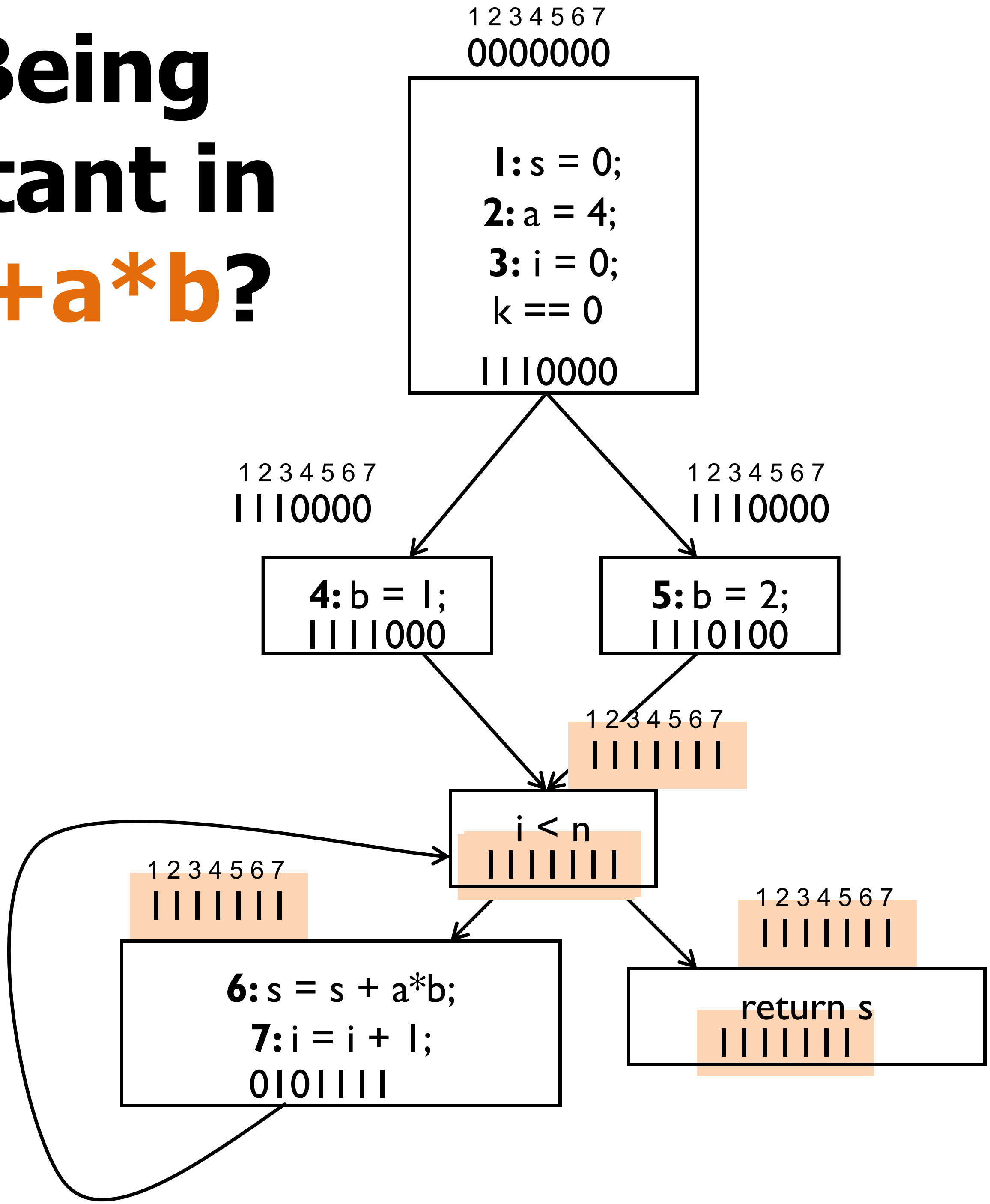
- Check all reaching definitions
- If all assign variable to same constant
- Then use is in fact a constant

Can replace variable with constant

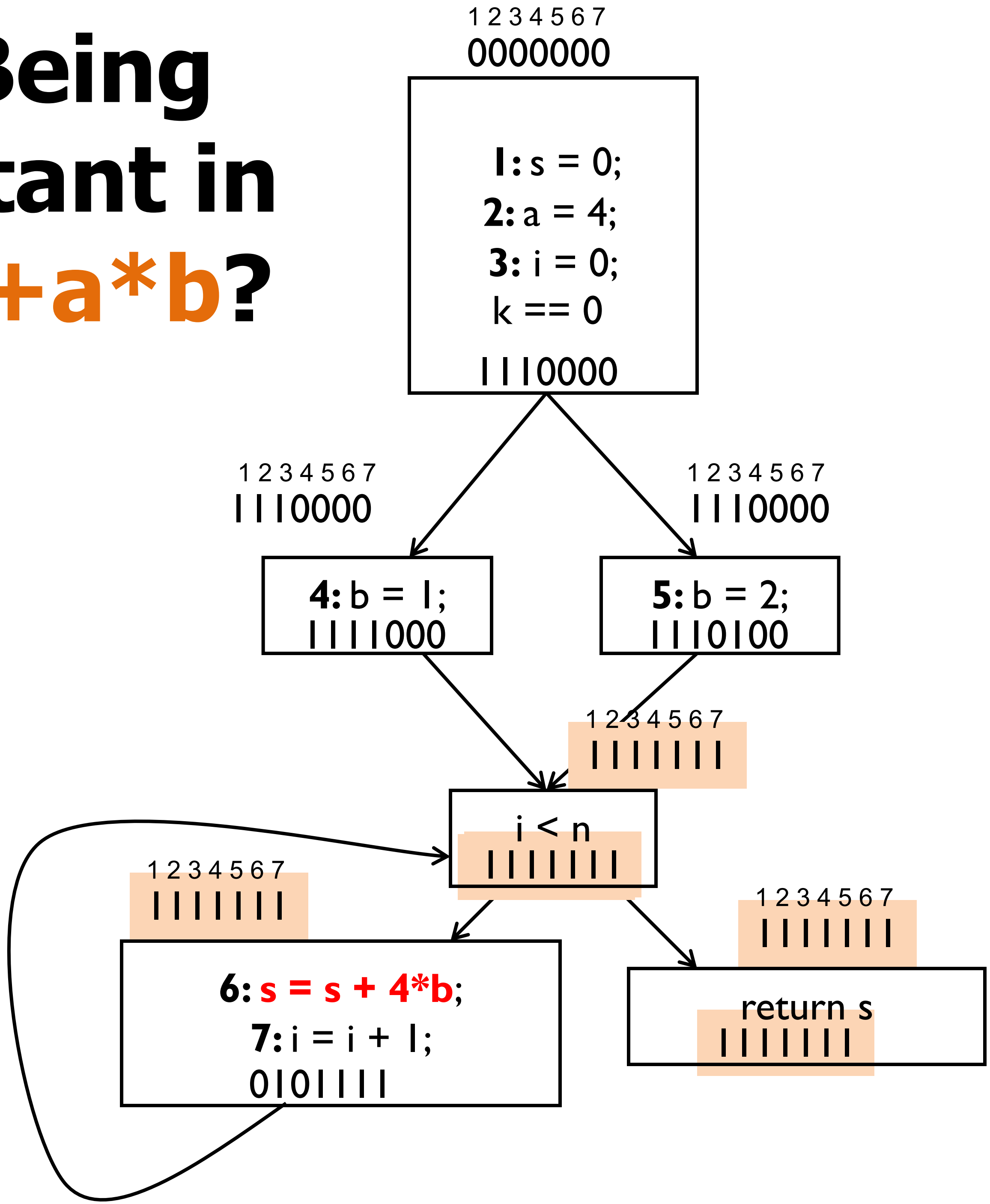


1 2 3 4 5 6 7
0000000
1: s = 0;
2: a = 4;
3: i = 0;
k == 0
1110000
1 2 3 4 5 6 7
1110000
4: b = 1;
1111000
1 2 3 4 5 6 7
1110000
5: b = 2;
1110100
1 2 3 4 5 6 7
1111111
i < n
1111111
1 2 3 4 5 6 7
1111111
6: s = s + a*b;
7: i = i + 1;
0101111
1 2 3 4 5 6 7
1111111
return s
1111111

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



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



Analysis: Available Expressions

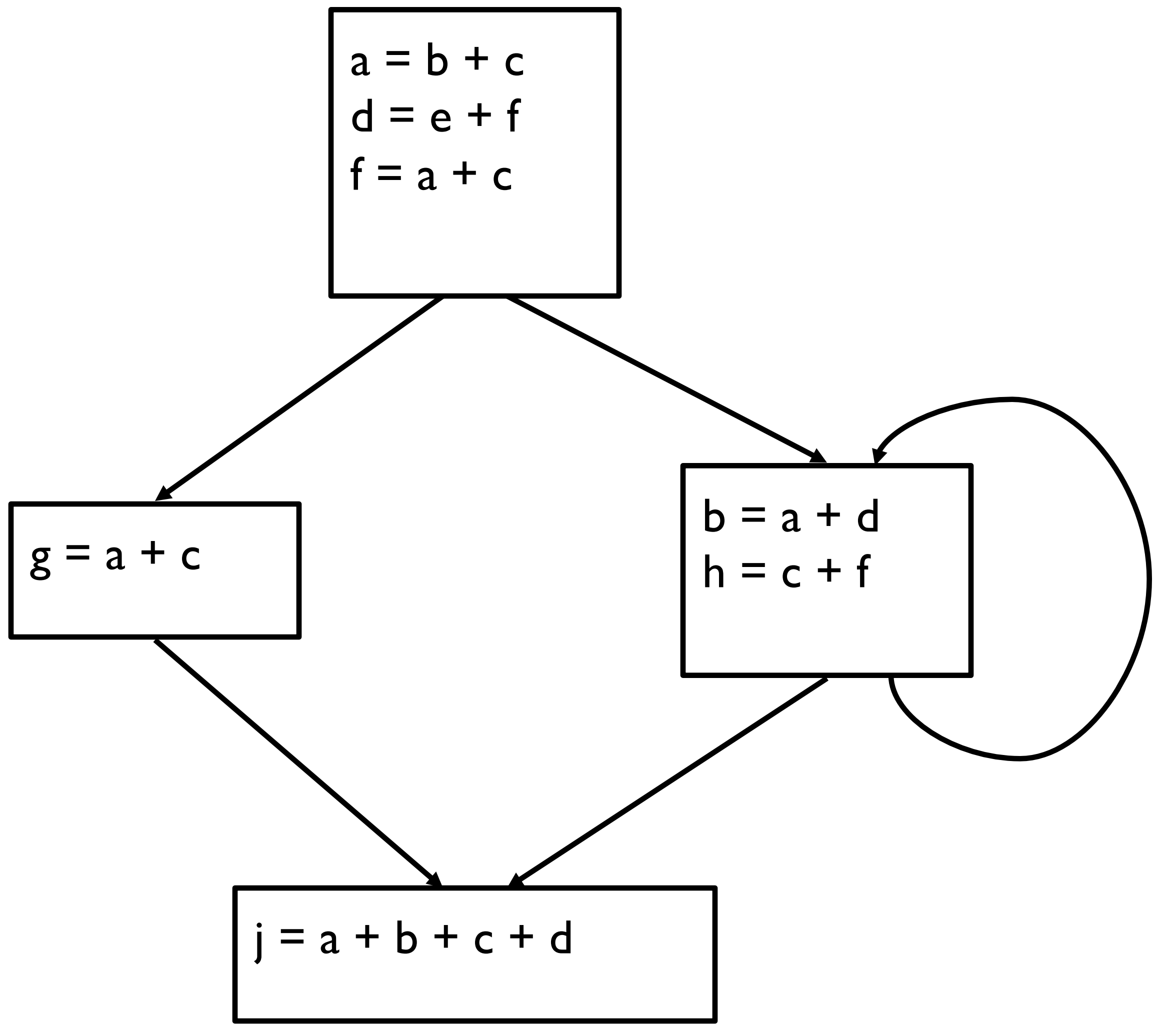
An expression x+y is available at a point p if

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

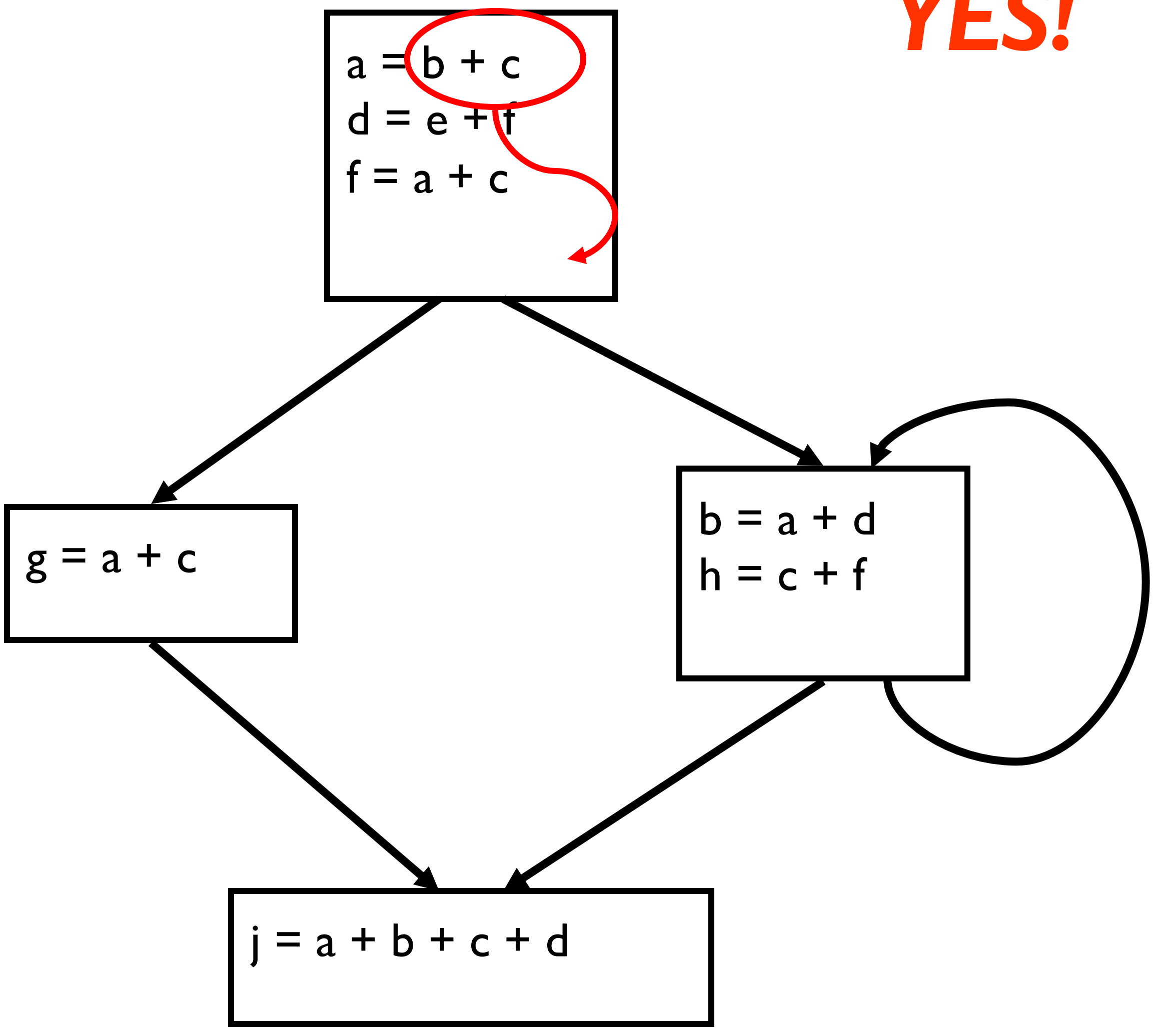
- If expression is available at use, no need to reevaluate it
- Beyond SSA-form analyses

Example: Available Expression



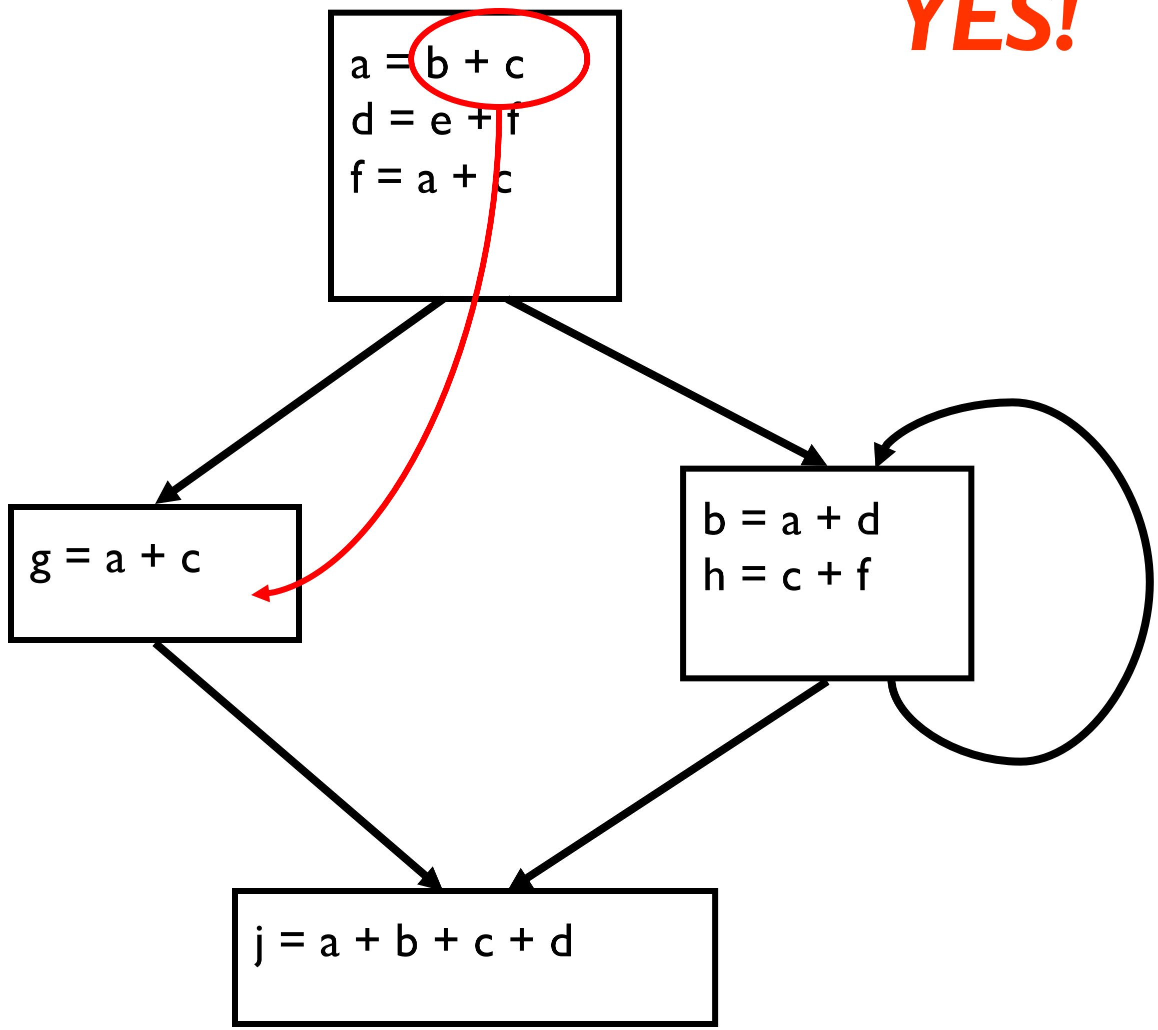
Is the Expression Available?

YES!

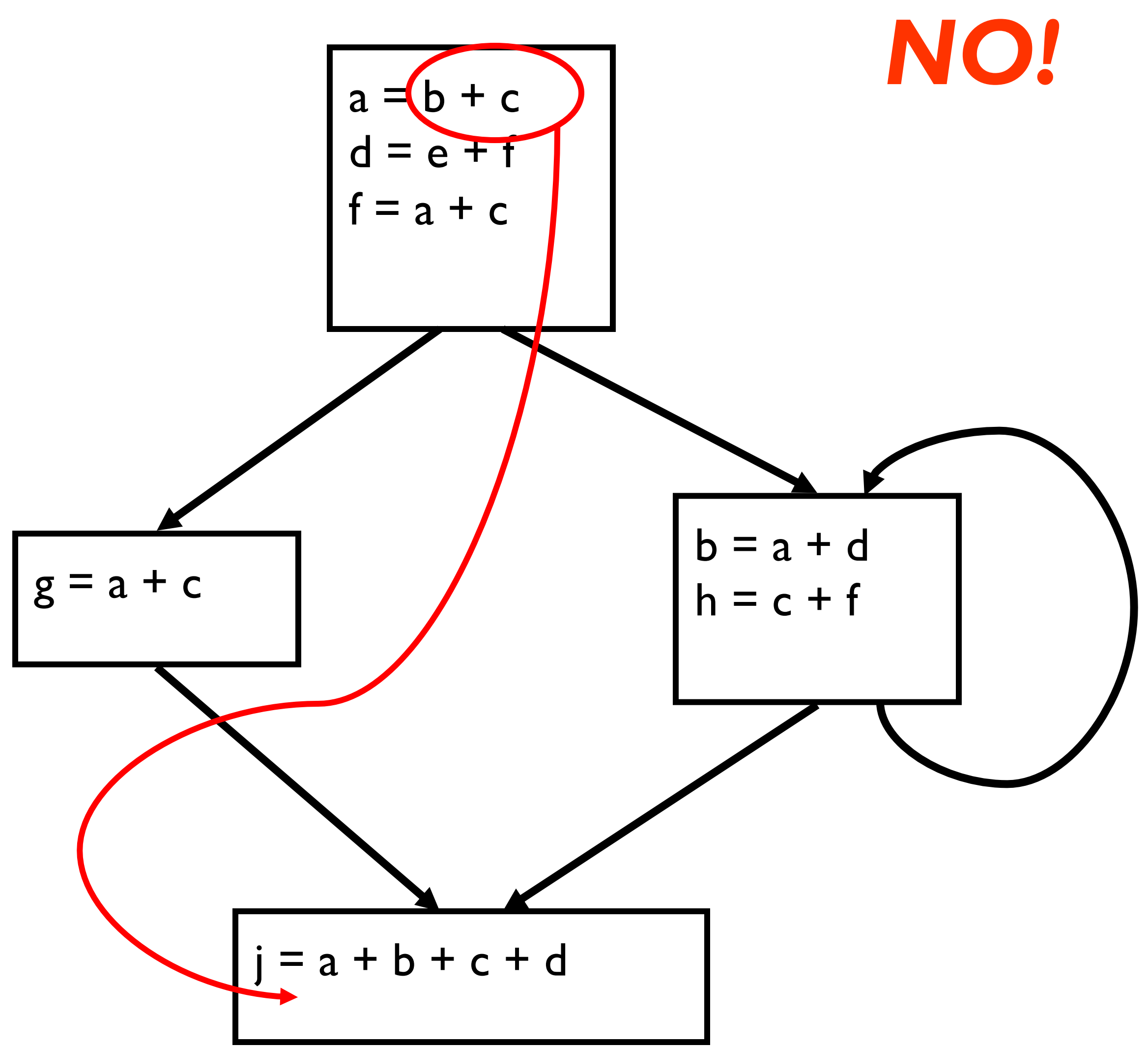


Is the Expression Available?

YES!

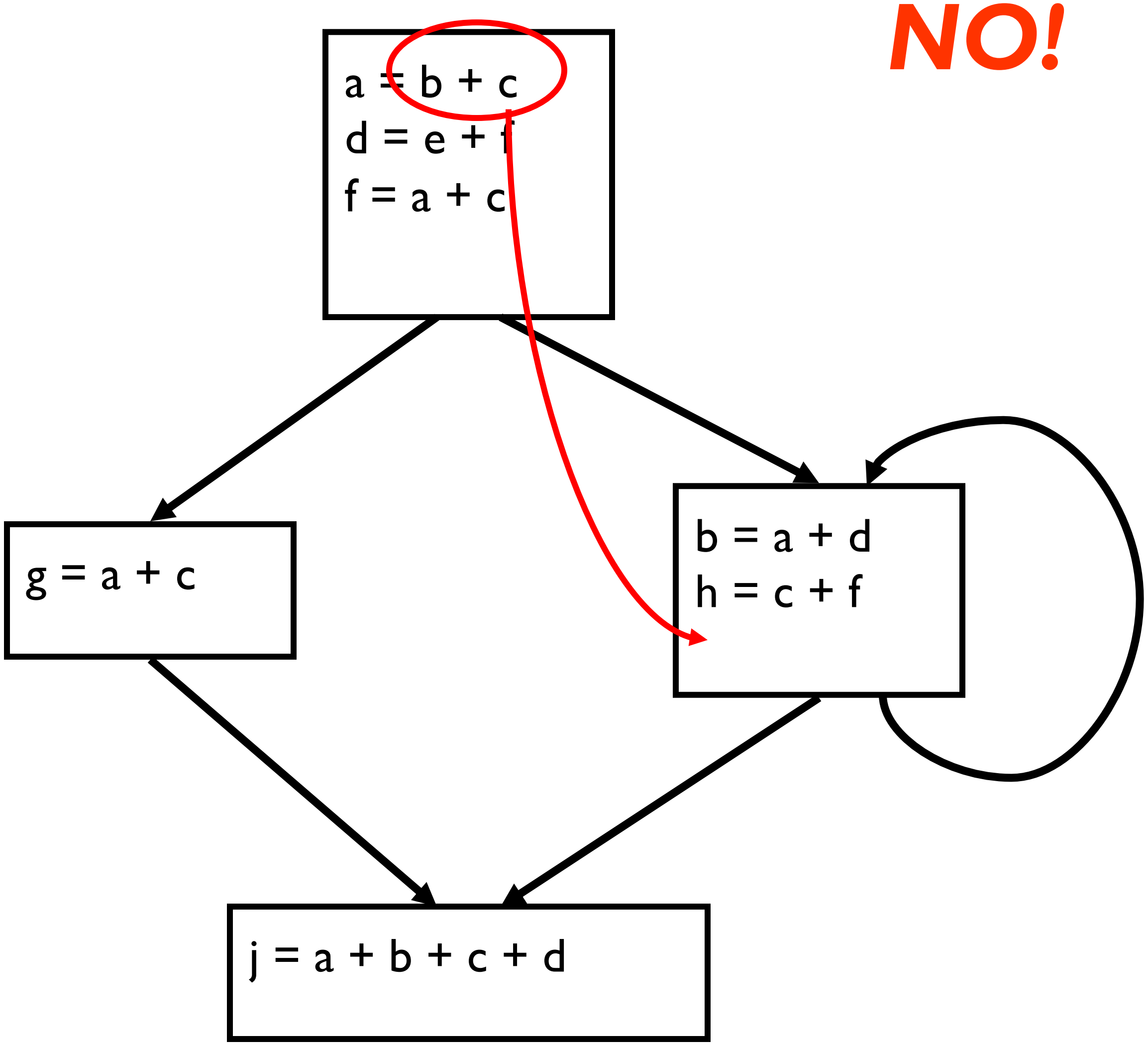


Is the Expression Available?



Is the Expression Available?

NO!



Transformation: Common Subexpression Elimination

Uses the results of available expressions

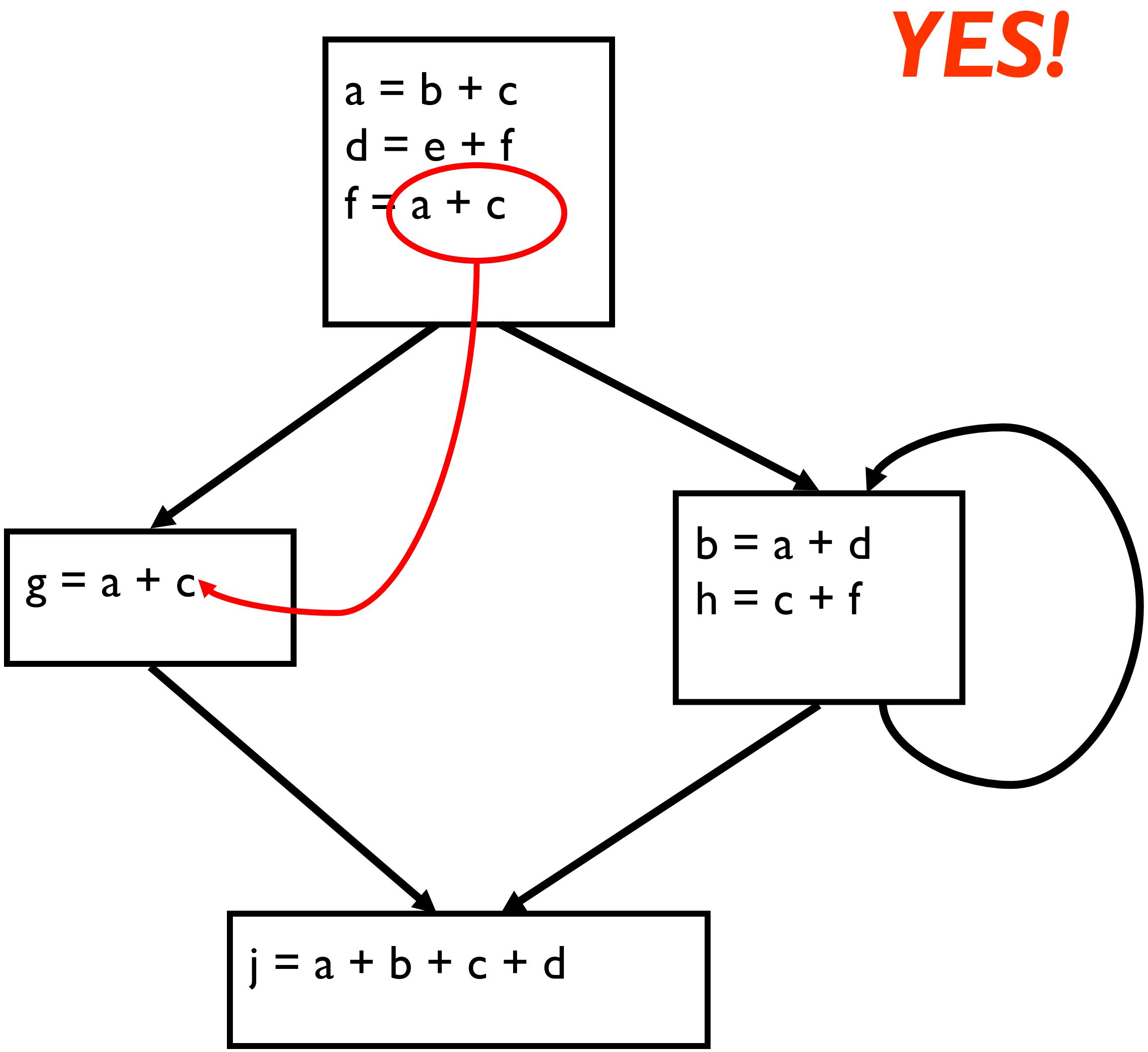
Check:

- If the expression is available and computed before,

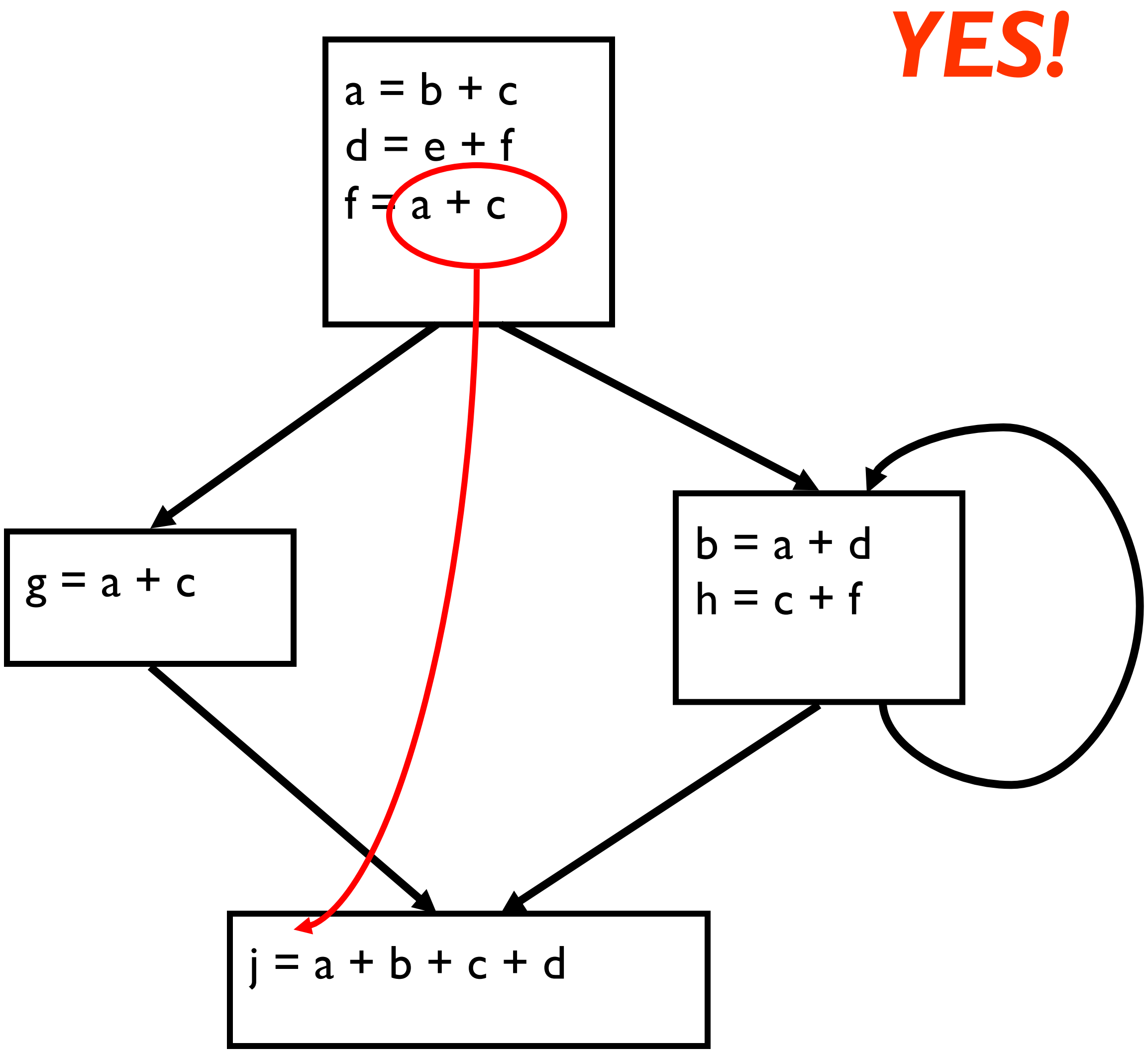
Transform:

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

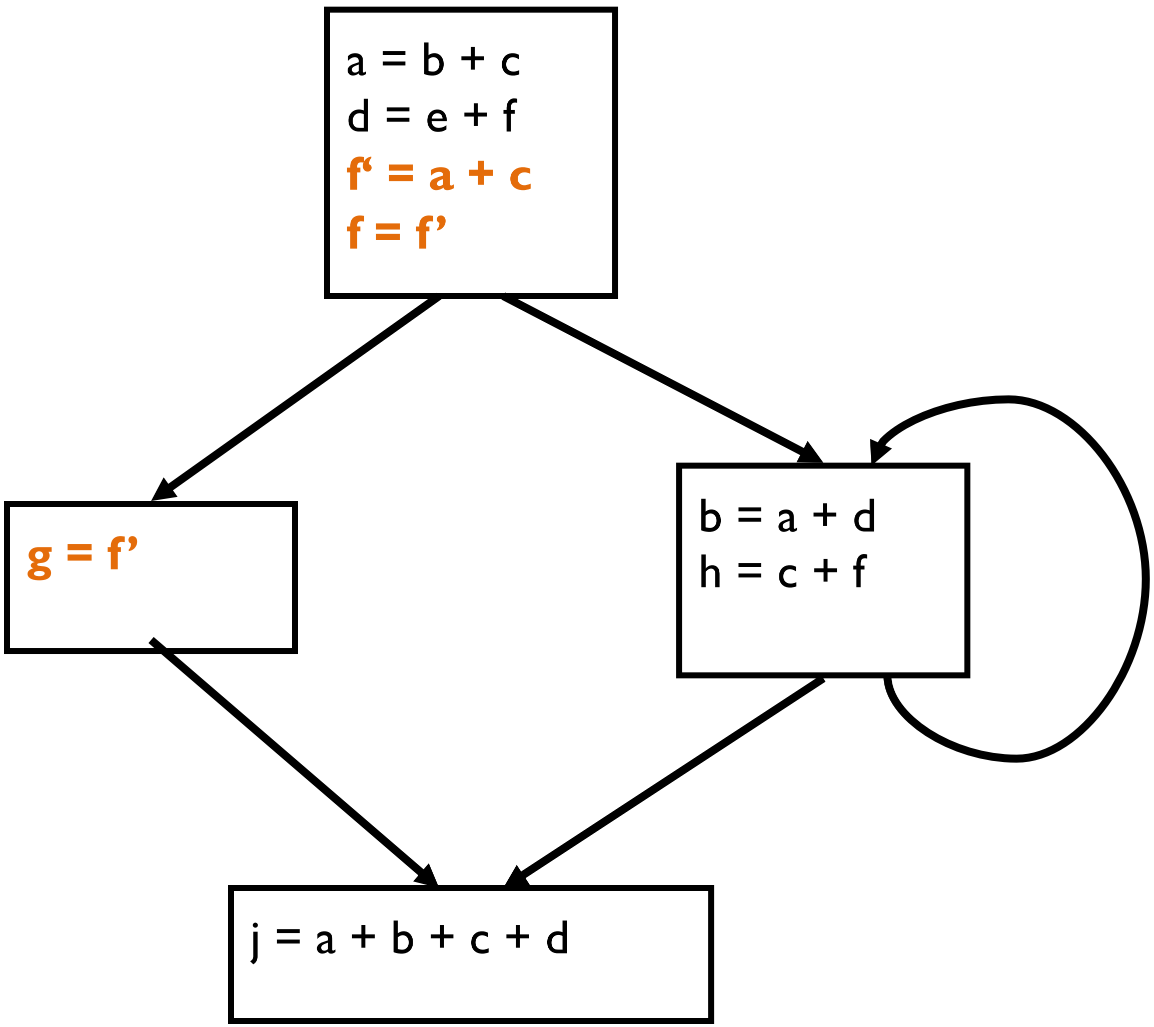
Use of Available Expression



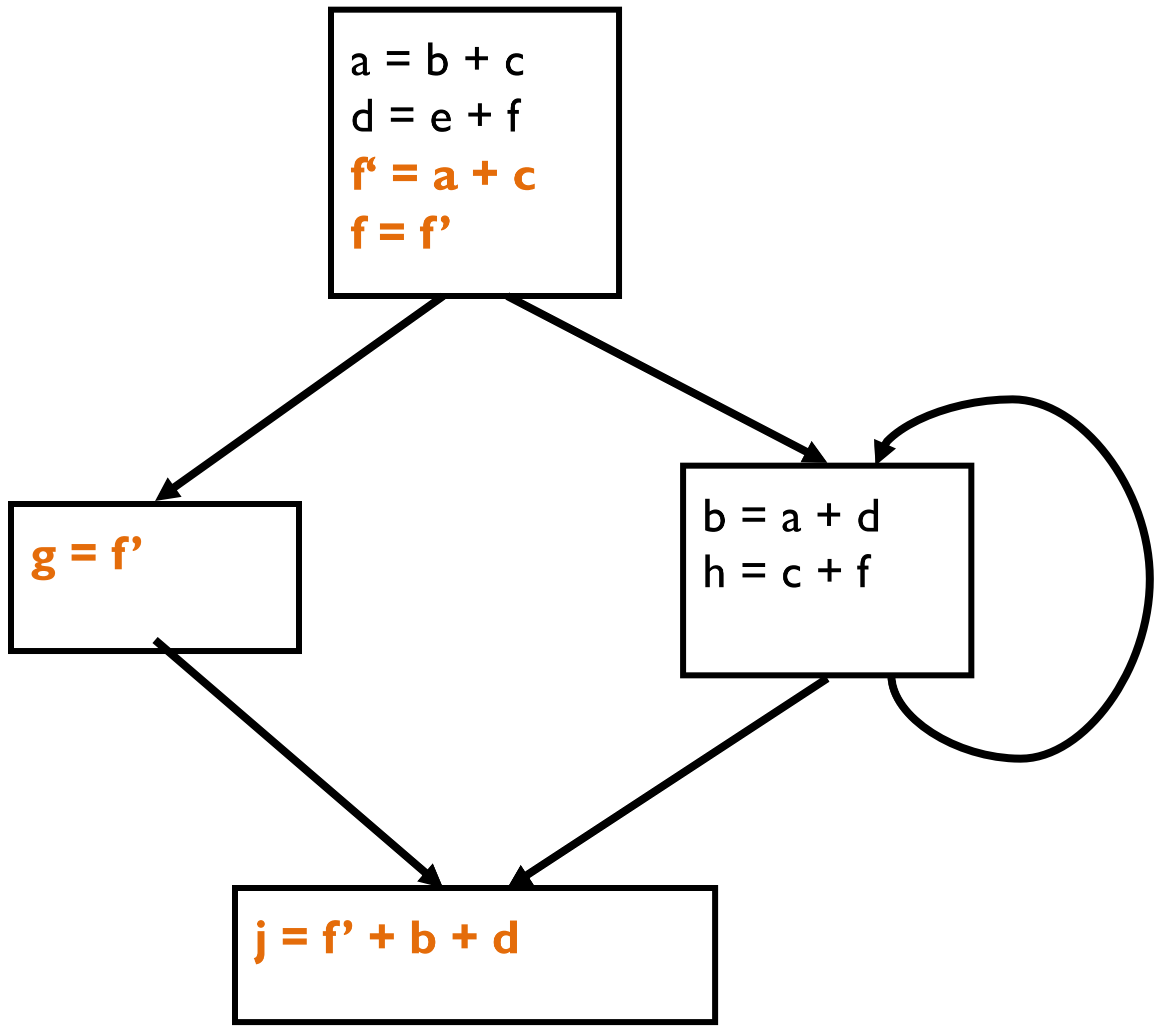
Use of Available Expression



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[b1] $\cap$ ... $\cap$ 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] $\cap$ ... $\cap$ 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]; // OUT[Entry] = GEN[Entry] U (∅ - KILL[n]);
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 };
```

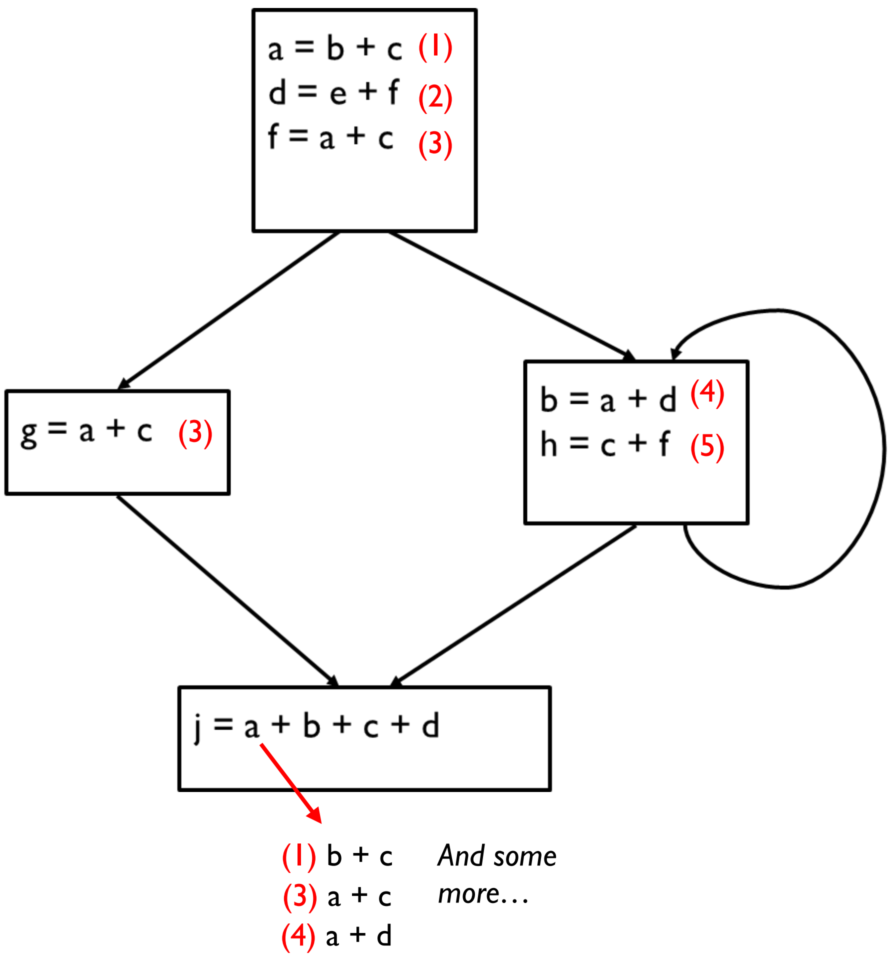
Questions

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?

Example: Available Expression



Common Subexpression Elimination

Inputs:

(1) CFG for a procedure
(2) Numbered set of expressions $E = \{e_1, \ldots e_N\}$
(3) Available expressions, $AVAIL_{in}(B)$, for each block B

Algorithm:

```
∀i, 1 ≤ i ≤ N : EverRedundant[i] = false;

for each block B  // replace all uses first
    for each statement S : X = Y op Z in B
        if (e_j = "Y op Z" ∈ AVAIL_in(B) and e_j is not killed before S in B) {
            EverRedundant[j] = true
            Replace S with X = tmp_j
        }

for each block B // assign temporaries
    for each original statement S : X = Y op Z in B
        if ("Y op Z" = e_j and EverRedundant[j]) {
             Allocate new temporary tmp_j
             replace S with the pair: "tmp_j = Y op Z; X = tmp_j"
}
```

CSE vs Value Numbering

One does not dominate the other

- CSE (through availability) considers the lexical names
- GVN (through numbering) considers the underlying values

GVN better:

```
a = b + c
d = b
e = c + d
```

CSE better:

```
if (…) {
  c = a + 1
  d = b + c
} else {
  c = a + 2
  d = b + c
}
e = b + c
```

- **In practice, run both!**

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.

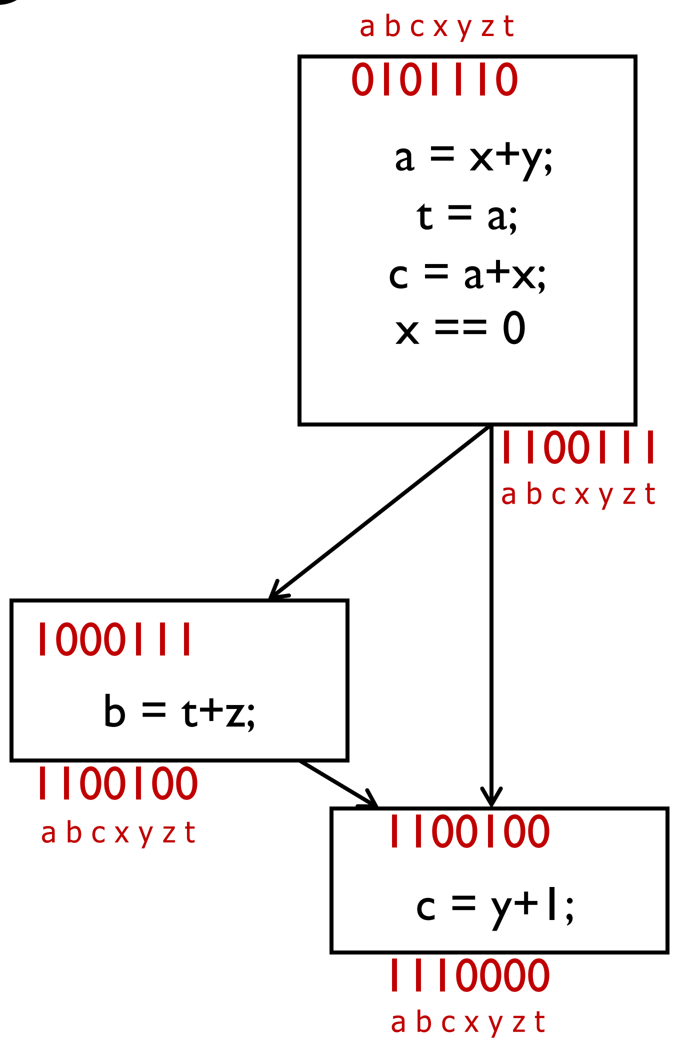
- Eliminate assignments to variables not read later.
- But must not eliminate last assignment to variable (such as instance variable) visible outside CFG.
- Can eliminate other dead assignments.
- Handle by making all externally visible variables live on exit from CFG

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

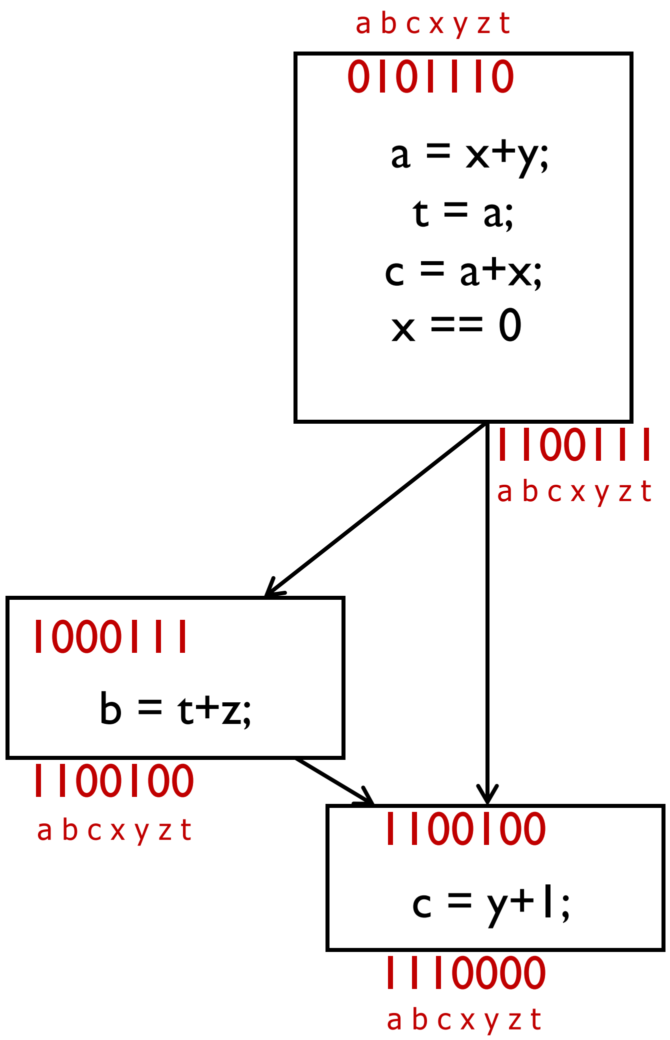
Liveness Example

- 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



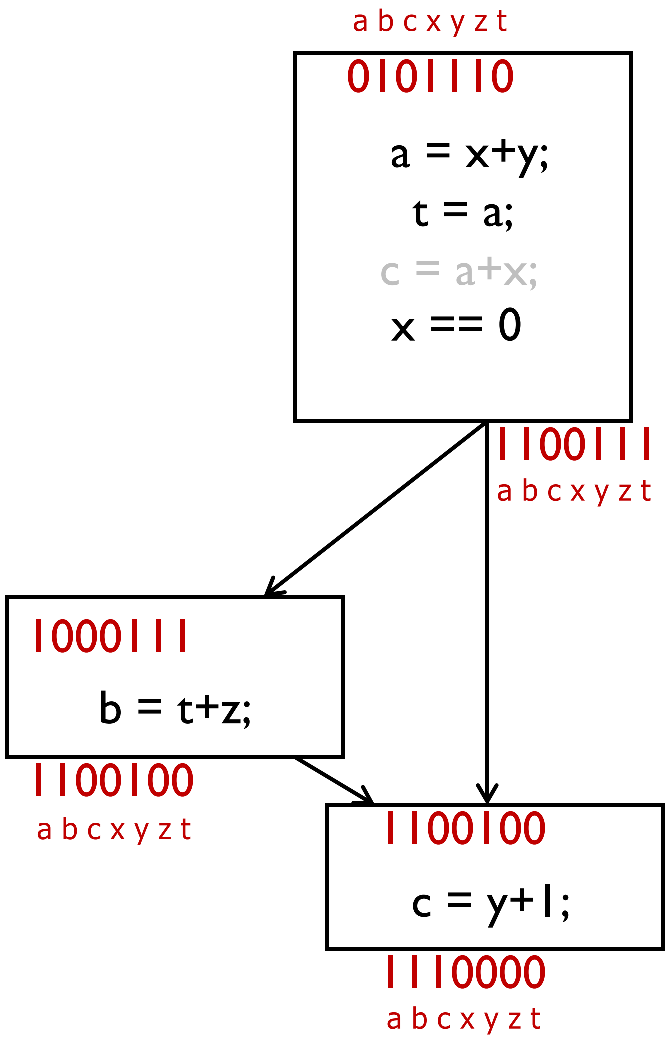
Transformation: Dead Code Elimination

- 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 - { 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[s];

    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

Can generalize framework to work for both forwards and backwards analyses

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!

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[s];

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

    if (IN[n] changed)
        for all nodes p in predecessors(n)
            Changed = Changed U { 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 $x = y$ (antisymmetric)
- $x \leq y$ 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 $x = z$

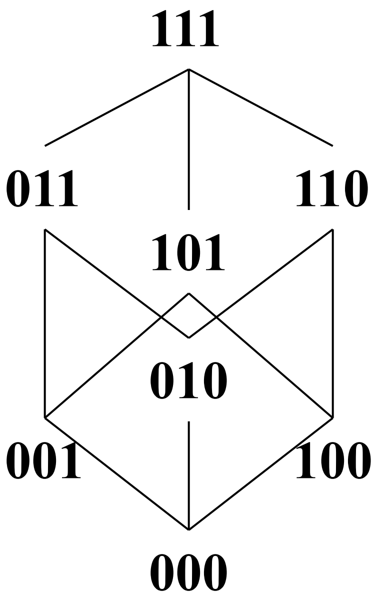
Conceptually, y covers x if there are no elements between x and y

Example

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

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

x ≤ y **is equivalent to** (x 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 Z

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

Top and Bottom

Greatest element of P (if it exists) is top (⊤)

- $\forall a \in L\,.\, a \vee \top = \top$
- Note: $\forall a \in L\,.\; a \leq \top$ and $\top \wedge a = a$

Least element of P (if it exists) is bottom (⊥)

- $\forall a \in L\,.\; a \wedge \bot = \bot$
- Note: $\forall a \in L\,.\; \bot \leq a$ and $\bot \vee a = a$

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

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

Proof:

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

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

Proof:

- $x \leq y$ 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

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

Proof:

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

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

Proof:

- 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
- We will define $\leq$ using $\vee$ and $\wedge$
- We will show that $\leq$ is a partial order

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

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 ('if'): $x \vee y = y \Rightarrow x = x \wedge y$

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

$= x \wedge y$ (by assumption)

Proof ('only if'): $x \wedge y = x \Rightarrow y = x \vee y$

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

$$x \vee y = y \text{ and } y \vee z = z \text{ implies } x \vee z = z$$

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

$$x \vee y = y \text{ and } y \vee x = x \text{ implies } x = y$$

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

$= x \vee y$ (by commutativity)

$= y$ (by assumption)

Proof of reflexivity property. Must show that

$x \vee x = x$, which follows directly

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

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

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

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

$L = x \wedge L$ (by assumption)

$= x \wedge (y \wedge L)$ (by assumption)

$= (x \wedge y) \wedge L$ (by associativity)

Semi-lattice $(P, \wedge)$

Set P and binary operation $\wedge$ such that $\forall x,y,z \in P$

- $x \wedge x = x$ (idempotent)
- $x \wedge y = y \wedge x$ implies $x = y$ (commutative)
- $(x \wedge y) \wedge z = x \wedge (y \wedge z)$ (associative)

The operation $\wedge$ imposes a partial order on P

If $((L, \leq), \wedge, \vee)$ is a lattice, then

- $(L, \wedge)$ is a **meet semi-lattice**
- $(L, \vee)$ is a **join semi-lattice**

Give us more flexibility to define the analysis.

- Since our analyses deal with complete lattices, we will represent the framework on them, but it can also be defined on semi-lattices
- Some dataflow analyses can be only represented on semi-lattices

Announcements and Plan

- Project II:
  - Will be released this week. Please start early.
- 02/24 class is cancelled; we will be having a class on 04/12
- What we covered on the previous day:
  - Theory of partial orders, meets and joins
- Today's plan:
  - A small revision
  - Relate the theory to dataflow analysis

Chains

A **poset (S, ≤) is a chain** if $\forall x,y \in S.\ y \le x$ or $x \le y$

Height of a poset/lattice: the size of the maximum chain.

(S, ≤) is finite if it has the finite height.

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

- When a particular ascending chain has the property that $x_n = x_{n+1} = \ldots$ we say that it stabilizes
- Then ascending chain condition means that all ascending chains stabilize

From one variable to more

If L is a poset then so is the Cartesian product LxL:

Let $(L_1, \leq_1)$ and $(L_2, \leq_2)$ be posets. Then $(L^*, \leq^*)$ is also a poset, where
$L^* = \{ (l_1, l_2) \mid l_1 \in L_1, l_2 \in L_2\}$ and $(l_{11}, l_{21}) \leq^* (l_{12}, l_{22}^{\top})$ iff
$l_{11} \leq_1 l_{12}$ and $l_{21} \leq_2 l_{22}$

This construction extends immediately on lattices, so that for $S \subseteq L^*$, we define $\bot^* = (\bot_1, \bot_2)$, we define
$glb(Y) = (glb\{ l_1 \mid (l_1, \_) \in Y, glb\{ l_2 \mid (\_ , l_2) \in Y)$ and same for lub and $\top^*$

See Nielsen, Nielsen and Hankin book

From one variable to more

Total function space (S -> L) :

Let $(L, \leq)$ be a poset, S a set and f total function. Then $(L^f, \leq^f)$ is also a poset, where

$L^f = \{f: S \to L\}$ and $f' \leq^f f''$ iff $\forall s \in S \,.\, f'(s) \leq \quad f''(s)$.

To extend to lattices, we define $\bot^f = \lambda s \,.\, \bot$ and $glb(Y) = \lambda s \;.\; glb_0 \{\, f(s) \mid f \in Y)$ and same for lub and $\top^f$

Monotone Function Space (L_1 -> L_2) :

Let $(L_1, \leq_1)$ and $(L_2, \leq_2)$ be posets and f monotone. Then $(L^f, \leq^f)$ is also a poset, where $\bot^f = \lambda s \,.\, \bot_2$ and

$L^f = \{f: L_1 \to L_2\}$ and $f' \leq^f f''$ iff $\forall l_1 \in L_1 \,.\, f'(l_1) \leq_2 f''(l_1)$

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

- Maps lattice elements to lattice elements

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** $f: P \to P$. 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 **monotonic**:
  $x \leq y$ 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

*One can also define distributivity in terms of $\wedge$ ("meet"): $f(x \wedge y) = f(x) \wedge f(y)$

Distributivity Implies Monotonicity

Proof.*

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

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

$$\begin{aligned} f(y) &= f(x \vee y) && \text{(by assumption)} \\ &= f(x) \vee f(y) && \text{(by distributivity)} \end{aligned}$$

*For $f(x \wedge y) = f(x) \wedge f(y)$, show $x \wedge y = x \Rightarrow f(x) \wedge f(y) = f(x)$; $f(x) = f(x \wedge y) = f(x) \wedge f(y)$

Knaster-Tarsky Fixed-point Theorem

Let:

- $(L, \leq, \wedge, \vee, \top, \bot)$ be a complete lattice
- $f : L \rightarrow L$ be a monotonic function
- ***fix (f)*** is the set of fixed points of f

The set ***fix (f)*** with relation $\leq$, and operators $\wedge$, $\vee$ is forming a complete lattice.

- There will be a least fixed-point and greatest fixed point

Consequences:

- f has at least one fixpoint
- That fixpoint is the largest element in the chain

$$\bot, f(\bot), f(f(\bot)), f(f(f(\bot))), \ldots , f^n(\bot)$$

Putting the Pieces Together…

Forward Dataflow Analysis

Simulates execution of program forward with flow of control

Tuple **(G, (L, ≤), F, I)** – (graph, (lattice), transfer fs., initial val.)

For each node $n \in$ **G**, we have

- in_n – value at program point before n
- out_n – value at program point after n
- $f_n \in$ **F** – transfer function for n (given in_n, computes out_n)
- Signature of in_n, out_n, f_n : **L** $\rightarrow$ **L**

Requires that solution satisfies

- $\forall n.$ $\quad out_n = f_n(in_n)$
- $\forall n \neq n_0.$ $\quad in_n = \vee \{ out_m . m \text{ in } pred(n) \}$
- in_{n0} = **I**, summarizes information at the start of program

Dataflow Equations

Compiler processes program to obtain a set of dataflow equations

$$out_n := f_n(in_n)$$

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

Conceptually separates analysis problem from program

Worklist Algorithm for Solving Forward Dataflow Equations

```
for each n do out_n := f_n(⊥)

in_n0 := I; out_n0 := f_n0(I)
worklist := N - { n_0 }

while worklist ≠ ∅ do
      remove a node n from worklist
      in_n  := ∨ { out_m . m in pred(n) }
      out_n := f_n(in_n)
      if out_n changed then
            worklist := worklist ∪ succ(n)
```

Correctness Argument

Why does the result satisfy dataflow equations?

- Whenever it processes 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, it puts 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 with infinite length, use **widening operator**
    - Detect lattice values that may be part of infinitely ascending chain
    - Artificially raise value to least upper bound of chain

Termination Argument (Details)

- For finite lattice **(L, ≤)**
- Start: each node n ∈ 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 ∈ 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 × C times, all IN[.] sets would be ⊤
  - The algorithm terminates in **O(N × C)** steps (but this is conservative)

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.

Kam & Ullman, 1976:

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

In practice:

- d(G) < 3, so the algorithm makes less than 6 passes over the graph

For mode details, see also Properties of data flow frameworks, Marlowe and Ryder (1990)

General Worklist Algorithm

(Reminder)

```
for each n do out_n := f_n(⊥)

in_n0 := I; out_n0 := f_n0(I)
worklist := N - { n_0 }

while worklist ≠ ∅ do
      remove a node n from worklist
      in_n  := ∨ { out_m . m in pred(n) }
      out_n := f_n(in_n)
      if out_n changed then
            worklist := worklist ∪ succ(n)
```

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 };
```

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 succ(n)
      Changed = Changed U { s };
```

General Worklist

```
for each n do out_n := f_n(⊥)

in_n0 := I; out_n0 := f_n0(I)
worklist := N - { n_0 }

while worklist ≠ ∅ do
   remove a node n from worklist

   in_n  := ∨ { out_m . m in pred(n) }

   out_n := f_n(in_n)

    if out_n changed then
      worklist := worklist ∪ 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 $\subseteq$)

$\perp = \emptyset$

$I = in_{n0} = \perp$

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

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

Does Reaching Definitions Framework Satisfy Properties?

$\subseteq$ satisfies conditions for $\leq$

- **Reflexivity:** $x \subseteq x$
- **Antisymmetry:** $x \subseteq y$ 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 $f(x \cup y) = f(x) \cup f(y)$

$$f(x) \cup f(y) = (a \cup (x - b)) \cup (a \cup (y - b))$$
$$= a \cup (x - b) \cup (y - b) = a \cup ((x \cup y) - b)$$
$$= f(x \cup y)$$

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 $f_1(f_2(x)) = a \cup (x - b)$

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 $\supseteq$)

$\perp = P$

$I = in_{n0} = \emptyset$

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

- 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
  - $\forall n.\ in_n = f_n(out_n)$
  - $\forall n \notin N_{final}.\ out_n = \vee \{ in_m .\ m \text{ in } succ(n) \}$
  - $\forall n \in N_{final} = out_n = O$
  - Where O summarizes information at end of program

Worklist Algorithm for Solving Backward Dataflow Equations

```
for each n do in_n := f_n(⊥)
for each n ∈ N_final do out_n := O; in_n := f_n(out_n)
worklist := N - N_final

while worklist ≠ ∅ do
  remove a node n from worklist
  out_n := ∨ { in_m . m in succ(n) }
  in_n := f_n(out_n)
  if in_n changed then
     worklist := worklist ∪ 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 $\subseteq$)

$\perp$ = $\emptyset$

O = $\emptyset$

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:

- $s_0;s_1;\ldots;s_k$, 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 $AF : ST \to P$

- Correctness condition: require that for all states s

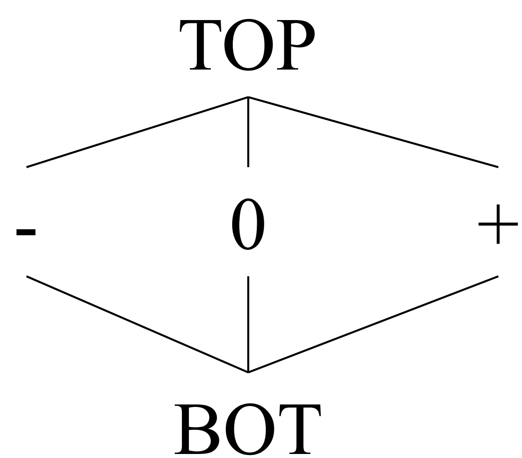
$$AF(s) \leq in_n$$

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 = flat lattice on {-,0,+}



Actual lattice records a value for each variable

- Example element: [a→+, b→0, c→-]

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?

- $AF([v_1,\ldots,v_n]) = [sign(v_1), \ldots, sign(v_n)]$
- $$\mathrm{sign}(x) = \begin{cases} 0 \text{ if } v = 0 \\ + \text{ if } v > 0 \\ - \text{ if } v < 0 \end{cases}$$

Transfer Functions

Transfer function modifies a map x : (Varname -> Sign)

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 $v_1 = v_2 * v_3$

- $f_n(x) =$ let newsign $= x[v_2] \otimes x[v_3]$ in
  $x[v_1 \rightarrow \text{newsign}]$

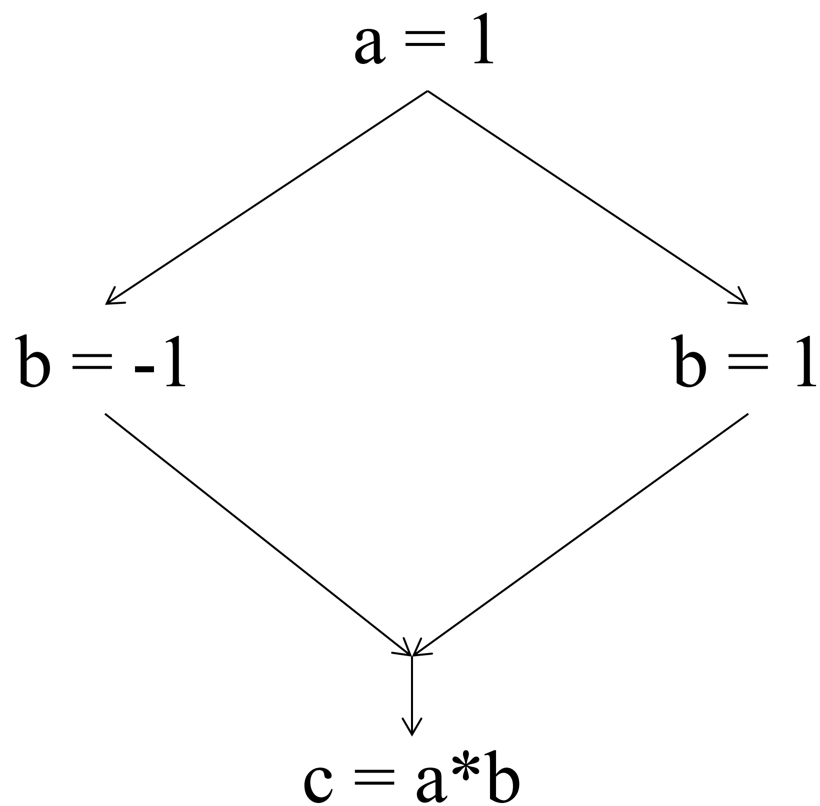
Init = for each variable assign TOP

(uninitialized variables may have any sign)

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	$\top$	$\top$

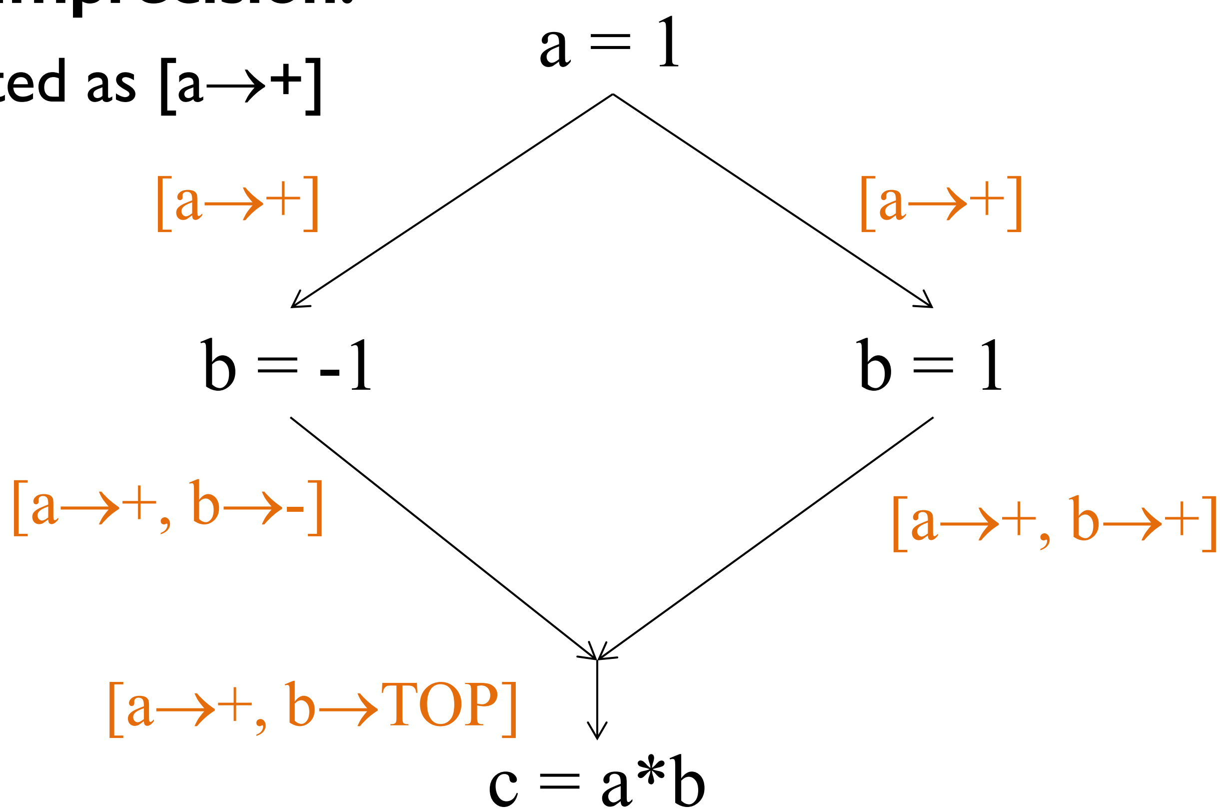
Sign Analysis Example



Imprecision In Example

Abstraction Imprecision:

[a→1] abstracted as [a→+]



Control Flow Imprecision:

[b→TOP] summarizes results of all executions.

(In any concrete execution state s, AF(s)[b]≠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

Abstraction Function Soundness (Sign Analysis)

Will show $\forall$ v. AF(s)[v] $\leq$ in(n)[v] (n is node for s)

by induction on length of computation that produced s

Base case:

- $\forall$ v. in(n_0)[v] = TOP, which implies that
- $\forall$ v. AF(s)[v] $\leq$ TOP

Abstraction Function Soundness: Induction step (Sign Analysis)

Assume $\forall$ v. AF(s)[v] $\leq$ in(n)[v] for computations of length k

Prove for computations of length k+1

Proof:

We are given s (state), n (node to execute next), and in(n). Goal: Find p (the node that just executed), s_p(the previous state), and in(p)

- By induction hypothesis $\forall$ v. AF(s_p)[v] $\leq$ in(p) [v]
- Case analysis on form of n:
  - If n of the form v = c, then

    1. ***s[v] = c and out$_p$ [v] = sign(c),*** so
       AF(s)[v] = sign(c) = out(p) [v] $\leq$ in(n)[v]
    2. ***If v'≠v, s[v'] = s_p [v'] and out(p) [v'] = in(p)[v'],*** so
       AF(s)[v'] = AF(s_p)[v'] $\leq$ in(p)[v'] = out(p)[v'] $\leq$ in(n)[v']

  - Similar reasoning if n of the form $v_1 = v_2$ op v_3

Augmented Execution States

Abstraction functions for some analyses require augmented execution states

- **Reaching definitions:** states are augmented with definition that created each value
- **Available expressions:** states are augmented with expression for each value

Meet Over Paths* Solution

What solution would be ideal for a forward dataflow problem?

Consider a path $p = \mathbf{n_0}, n_1, \ldots, n_k, \mathbf{n}$ 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:

$f_p(\perp) = (f_{nk}(f_{nk-1}(\ldots f_{n1}(f_{n0}(\perp)) \ldots)) \leq in_n$

So the solution must have the property that

$$\vee\{f_p(\perp) \,.\, p \text{ is a path to } n\} \leq in(n)$$

and ideally

$$\vee\{f_p(\perp) \,.\, p \text{ is a path to } n\} = in(n)$$

***** Name exists for historical reasons; this will be a join-over-paths in our formulation for this problem. One can reformulate this with ∧ ("meet") instead***

See Nielsen, Nielsen and Hankin book for more on "join" and Dragon book for the classical "meet" formalization

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 functions
- Uses following lemma

Lemma:

Worklist algorithm produces a solution such that

$out(n) = f_n(in(n))$

if $n \in pred(m)$ then $out(n) \leq in(m)$

Proof

Base case: p is of length 1

- Then $p = n_0$ and $f_p(\perp) = \perp = in(n_0)$

Induction step:

- Assume theorem for all paths of length k
- Show for an arbitrary path p of length k+1

Induction Step Proof

$p = n_0, \ldots, n_k, n$

Must show $f_k(f_{k-1}(\ldots f_1(f_0(\bot)) \ldots)) \leq in(n)$

- By induction $(f_{k-1}(\ldots f_1(f_0(\bot)) \ldots)) \leq in(n_k)$
- Apply f_k to both sides, by monotonicity we get
  $$f_k(f_{k-1}(\ldots f_1(f_0(\bot)) \ldots)) \leq f_k(in(n_k))$$
- By lemma, $f_k(in(n_k)) = out(n_k)$
- By lemma, $out(n_k) \leq in(n)$
- By transitivity, $f_k(f_{k-1}(\ldots f_1(f_0(\bot)) \ldots)) \leq in(n)$

Distributivity

Distributivity preserves precision

If framework is distributive, then worklist algorithm produces the meet over paths solution

- For all n:

$$\vee\{f_p(\bot) \,.\, p \text{ is a path to } n\} = in_n$$

Soundness Proof of Analysis Algorithm

Connections between MOP and worklist solution:

- [Kildall, 1973] The **iterative worklist algorithm**: (1) converges and (2) computes a MFP (in our "join" case the least fixed point; in classical paper "meet", it computes the maximum fixed point) solution of the set of equations using the worklist algorithm

- [Kildall, 1973] **If F is distributive, MOP = MFP**

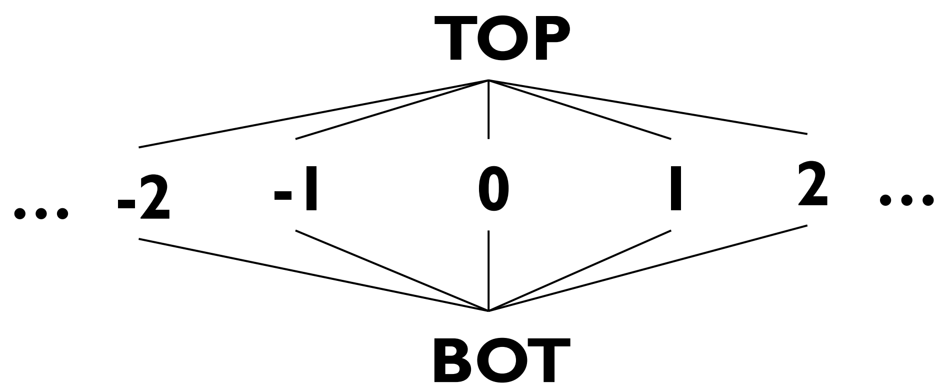
  $\vee \{f_p(\bot) \,.\, p \text{ is a path to } n\} = in_n$

- [Kam & Ullman, 1977] If **F is monotone**, **MOP ≤ MFP** (i.e. MFP is more conservative)

Note: if you reformulate the framework formulas with the "meet" operator, in that case MFP ≤ MOP

Lack of Distributivity Example

Constant Calculator: Flat Lattice on Integers



Actual lattice records a value for each variable

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

Transfer function:

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

Lack of Distributivity Anomaly

a = 2
b = 3

a = 3
b = 2

[a→2, b→3]

[a→3, b→2]

[a→TOP, b→TOP]

c = a+b

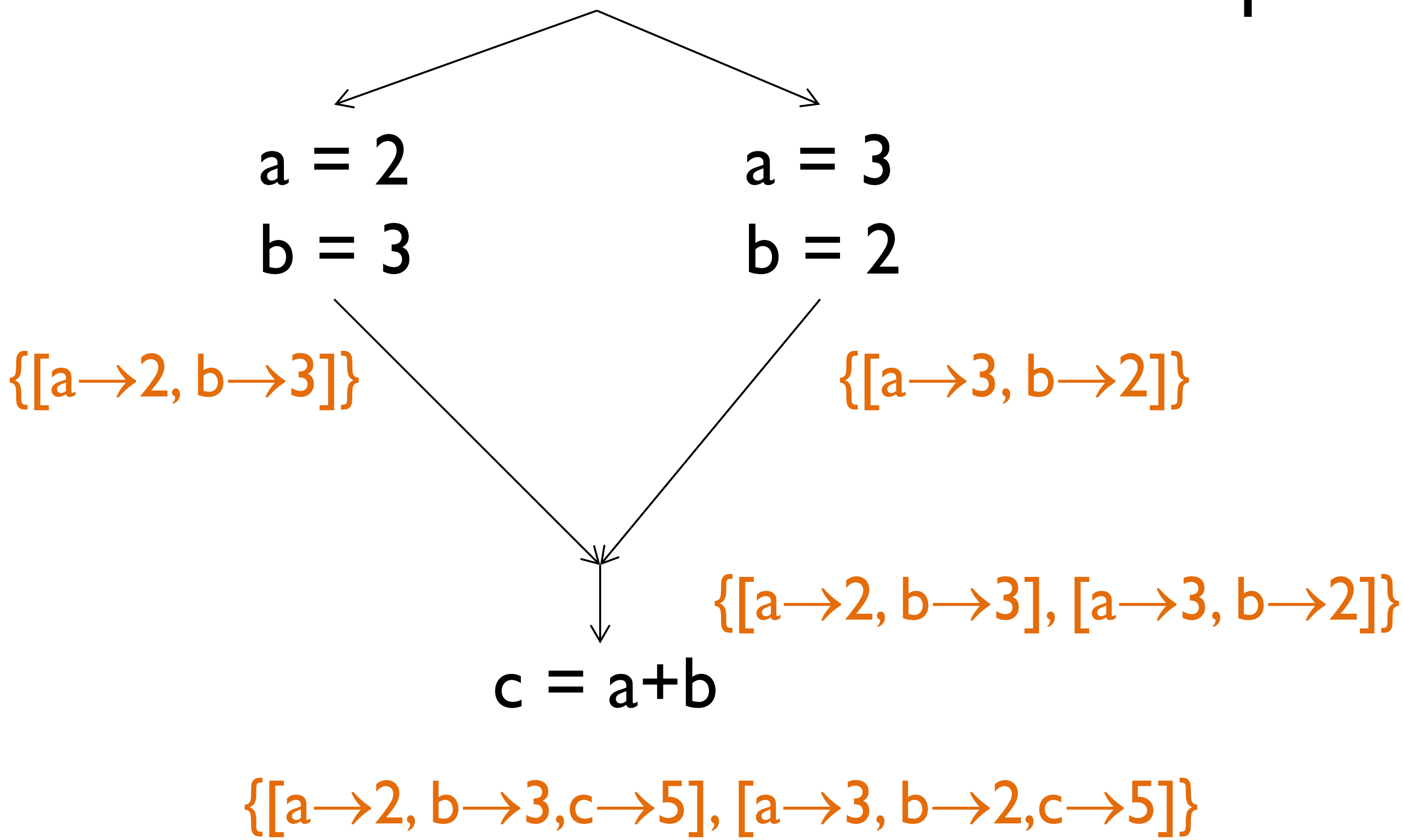
[a→TOP, b→TOP, c →TOP]

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

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

III Precise Sign Analysis

In this question we will build a more precise sign analysis. The purpose of this analysis is to enable the compiler to perform safety checks for calls to the log(x) function.

The analysis will analyze programs with one variable x. The language is defined as a sequence of the statements of this form:

$$
\begin{aligned}
S ::= &\ x = c \\
&\mid x = x + c \\
&\mid \texttt{if}\ (x == c)\ \{S_1\}\ \texttt{else}\ \{S_2\};
\end{aligned}
$$

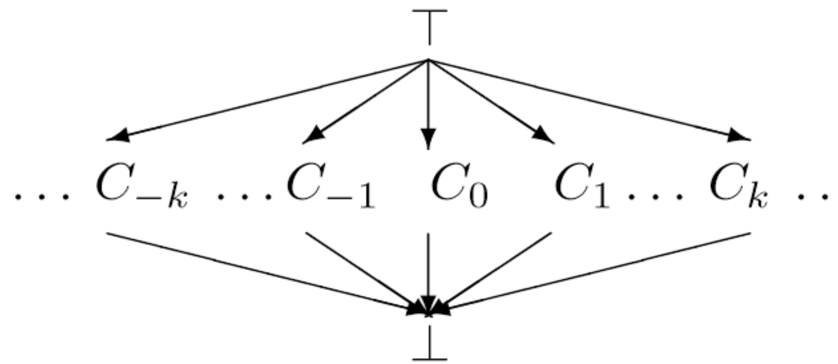
In addition, the very last statement in the program is a call to the `log(x)` function. In the previous definition, each c is a (signed) integer constant.

To keep track of the sign of variable x, we will use the lattice $(\mathcal{P}(\{-, 0, +\}), \subseteq)$. For example, if x has a non-negative value, then the analysis will represent this as a set $\{0, +\}$. If x has a positive value, the analysis will represent this as a set $\{+\}$.

Bonus #1: SCCP Revisited

Lattice L≡{⊤, Ci,⊥}.

- ⊤ intuitively means “May be constant.”
- ⊥ intuitively means “Not constant.”



A Partial Order ≤:

- ⊥ ≤ Ci for any Ci.
- Ci ≤ ⊤ for any Ci.
- Ci ≤ Cj (i.e., no ordering).

Meet of X and Y (X⊓Y) is the greatest value Z, s.t. Z ≤ X and Z ≤ Y.

SCCP Revisited

Assume:

- Only assignment or branch statements
- Every non-ϕ statement is in separate BB

Key Ideas:

1. Constant propagation lattice = {⊤, Ci,⊥}
2. 2.Initially: every def. has value⊤(“may be constant”).Initially: every CFG edge is infeasible, exceptedges froms
3. 3. Use 2 worklists: FlowWL, SSAWL

Highlights:

- Visit S only if some incoming edge is executable
- Ignore ϕ-argument if incoming CFG edge not executable
- If variable changes value, add SSA out-edges to SSAWL
- If CFG edge executable, add to FlowWL

SCCP Revisited

```
SCCP()
Initialize(ExecFlags[], LatCell[], FlowWL, SSAWL);
while ((Edge E = GetEdge(FlowWL ∪ SSAWL)) != 0)
    if (E is a flow edge && ExecFlag[E] == false)
       ExecFlag[E] = true
       VisitPhi(ϕ) ∀ ϕ ∈ E->sink
       if (first visit to E->sink via flow edges)
           VisitInst(E->sink)
       if (E->sink has only one outgoing flow edge Eout)
           add Eout to FlowWL
    else if (E is an SSA edge)
       if (E->sink is a ϕ node)
           VisitPhi(E->sink)
       else if (E->sink has 1 or more executable in-edges)
           VisitInst(E->sink)
```

SCCP Revisited

```
VisitPhi(ϕ) :
   for (all operands Uk of ϕ)
      if (ExecFlag[InEdge(k)] == true)
         LatCell(ϕ) ⊓ = LatCell(Uk)
         if (LatCell(ϕ) changed)
             add SSAOutEdges(ϕ) to SSAWL

VisitInst(S) :
   val = Evaluate(S)
   LatCell(S) = val
   if (LatCell(S) changed) // cannnot be Top
      if (S is Assignment)
      add SSAOutEdges(S) to SSAWL
   else // S must be a Branch
      Add one or both outgoing edges to FlowWL
```

Bonus #2: Partial Redundancy Elimination

Finds additional optimization opportunities, redundant only over some branches

Combines multiple dataflow analyses (e.g. commonly 5)

Lazy code motion:

- Compute available expressions
- Compute very busy expressions
    - an expression is very busy iff along *every* path from *p* there is an expression *A op B* before redefining *A* or *B.*
- Compute an earliest placement for each expression
- Move expressions down the CFG while the semantics remain the same

References:

J. Knoop, O. Rüthing, and B. Steffen, "Lazy Code Motion," In *PLDI*, 1992.

1. *Original paper:* E. Morel and C. Renvoise, "Global optimization by suppression of partial redundancies," CACM 22(2), Feb, 1979.

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)