CS 477: Dataflow Analysis and Abstract Interpretation

Sasa Misailovic

Based on previous slides by Saman Amarasinghe, Martin Rinard, and by Vikram Adve and Martin Vechev

University of Illinois at Urbana-Champaign

Recap

Representing program execution:

- Big Step Semantics
- Small Step Semantics
- Symbolic Execution
- Control-flow Graph

Today: Static analysis

Answers key questions about program properties over control-flow paths **at compile-time (without running the program)**

Static Analysis (Informally)

Symbolically “simulate” execution of program

- Forward (from program start to end)
- Backward (from program end to start)

Our plan:

- Examples first
- Theory follows
- (And the theory is rich!)

Static Analysis Uses

Analysis for program ***correctness***

- Ensures the program satisfies its specification (i.e., is correct)
- Make sure it does not crash, diverge or yield unacceptable results

Analysis for program ***optimization***

- Optimizing and just in time compilers
- Make sure the optimization preserves the semantics of the program (i.e., produces the same outputs as the original one)

Analysis for program ***development***

- Support debugging and refactoring
- Makes programmer’s life easier, with trustable hints

Static program analysis, Moller and Schwartzbacher, 2021

Example from last time

```
int x = input()
int y = 0

if x > 0
      y = x + 1
else
      y = -x
```

// Specification:
// y $\geq$ 0 after the run

We know:

- Concrete execution
- Symbolic execution
- CFG

We can infer what the specification is (mathematically).

Example from last time

```
int x = input()
int y = 0

if x > 0
      y = x + 1
else
      y = -x
```

// Specification:
// y $\geq$ 0 after the run

Do we need all the execution details to check the specification?

Sign Analysis

Sign analysis - compute sign of each variable v

Propagate information:

- No known sign
- Minus or Zero or Plus
- Multiple Possible Signs

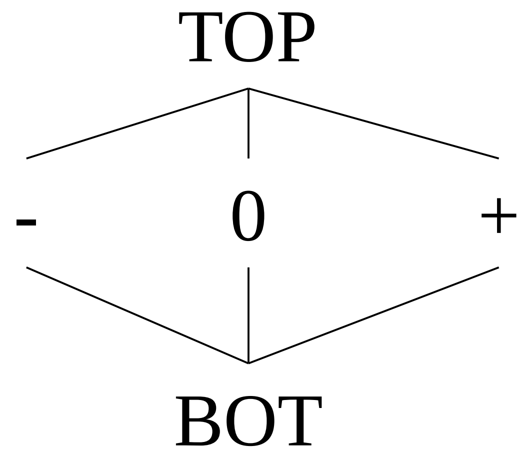
Mathematical foundation of the analysis:

- ***Lattice*** (partially ordered sets to keep track about the prevision of operations)
- ***Abstraction function*** (how we convert concrete values and states to abstract)
- ***Transfer function*** (how the abstract values propagate through the program)

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:

- $\perp$: 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?

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

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 ressign = $x[v_2] \otimes x[v_3]$ in $x[v_1 \rightarrow \text{ressign}]$

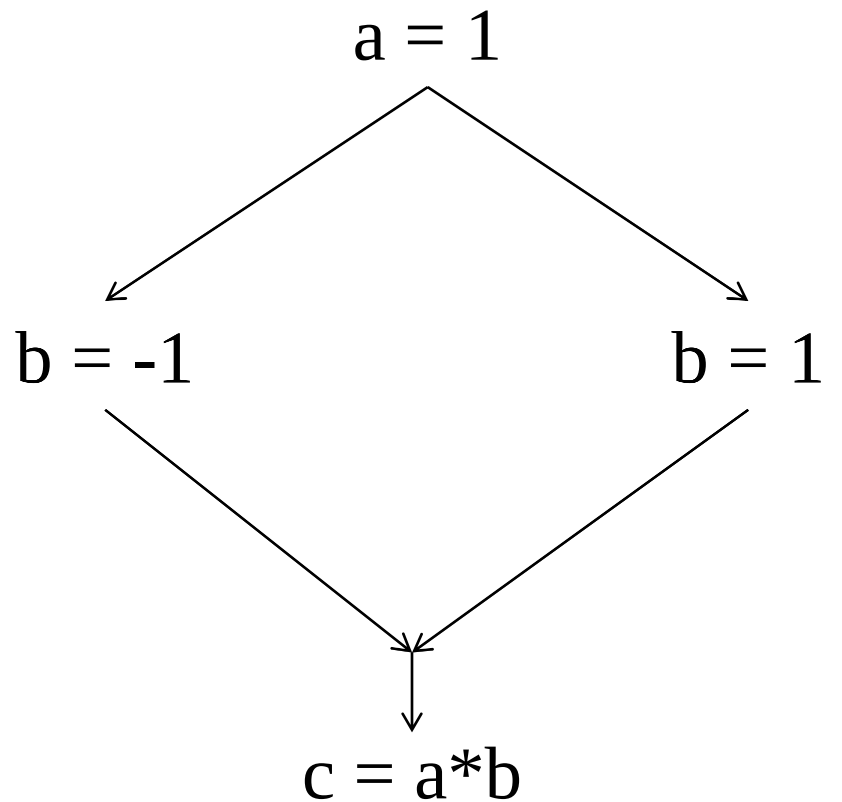
Init = for each variable assign TOP

(uninitialized variables may have any sign)

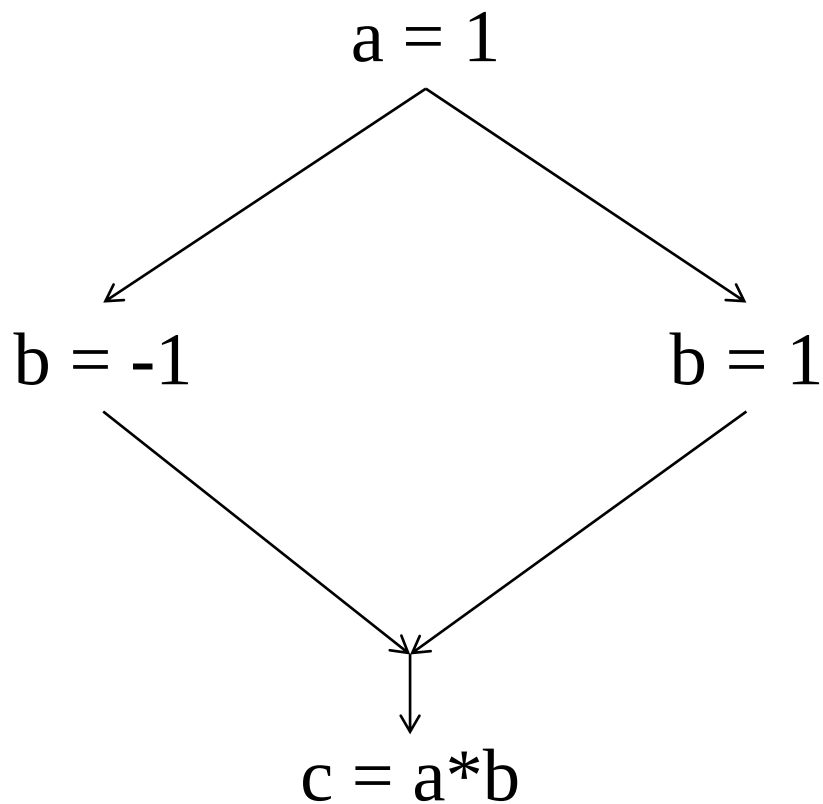
Operation ⊗ on Lattice

$\otimes$	$\bot$	-	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



Soundness in Example



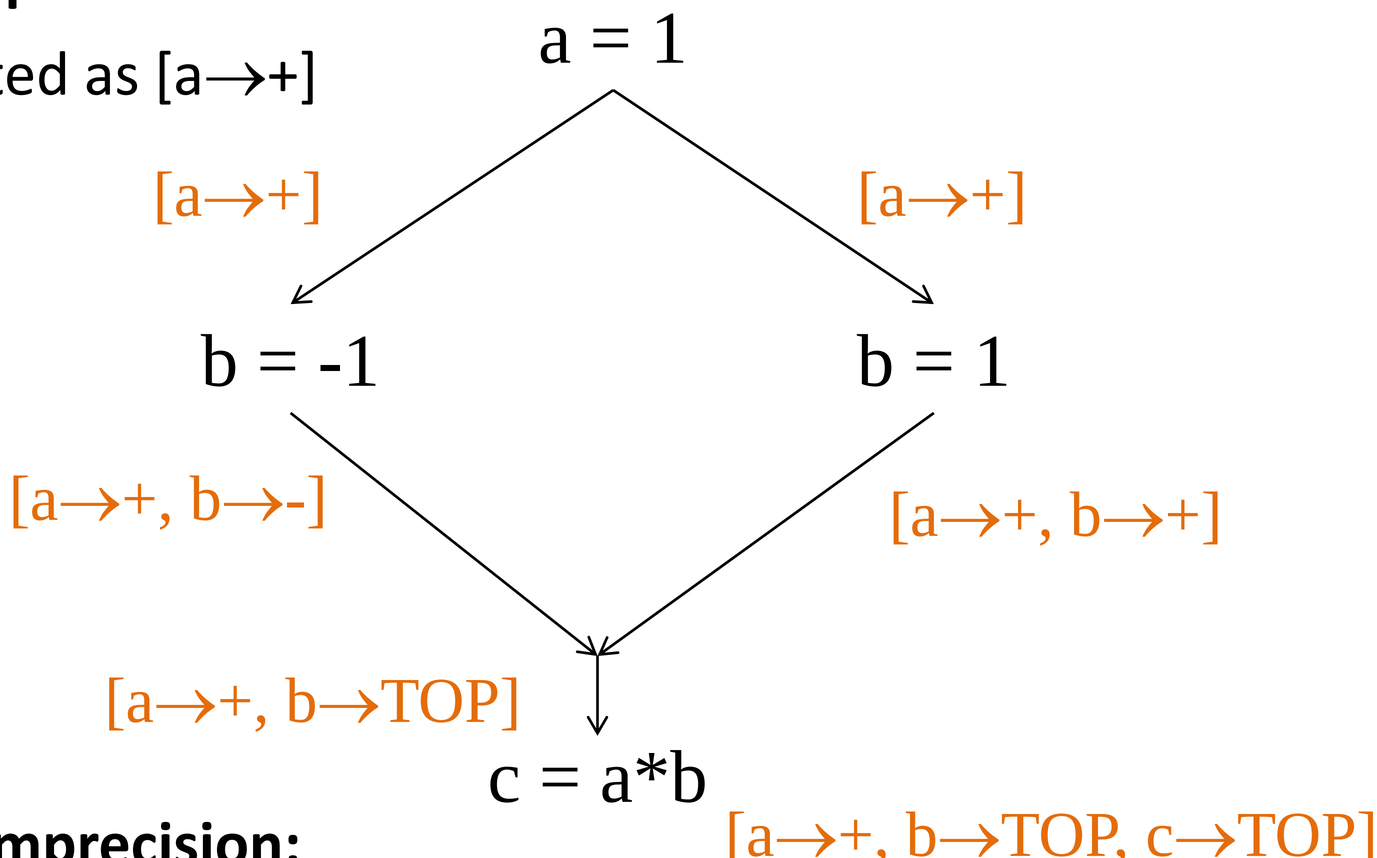
If the analysis returns that the sign of a is positive, then any and all concrete executions will have this property.

- Follows: at any program point, abstract state contains all possible concrete states

Imprecision In Example

Abstraction Imprecision:

[a→1] abstracted as [a→+]



Control Flow Imprecision:

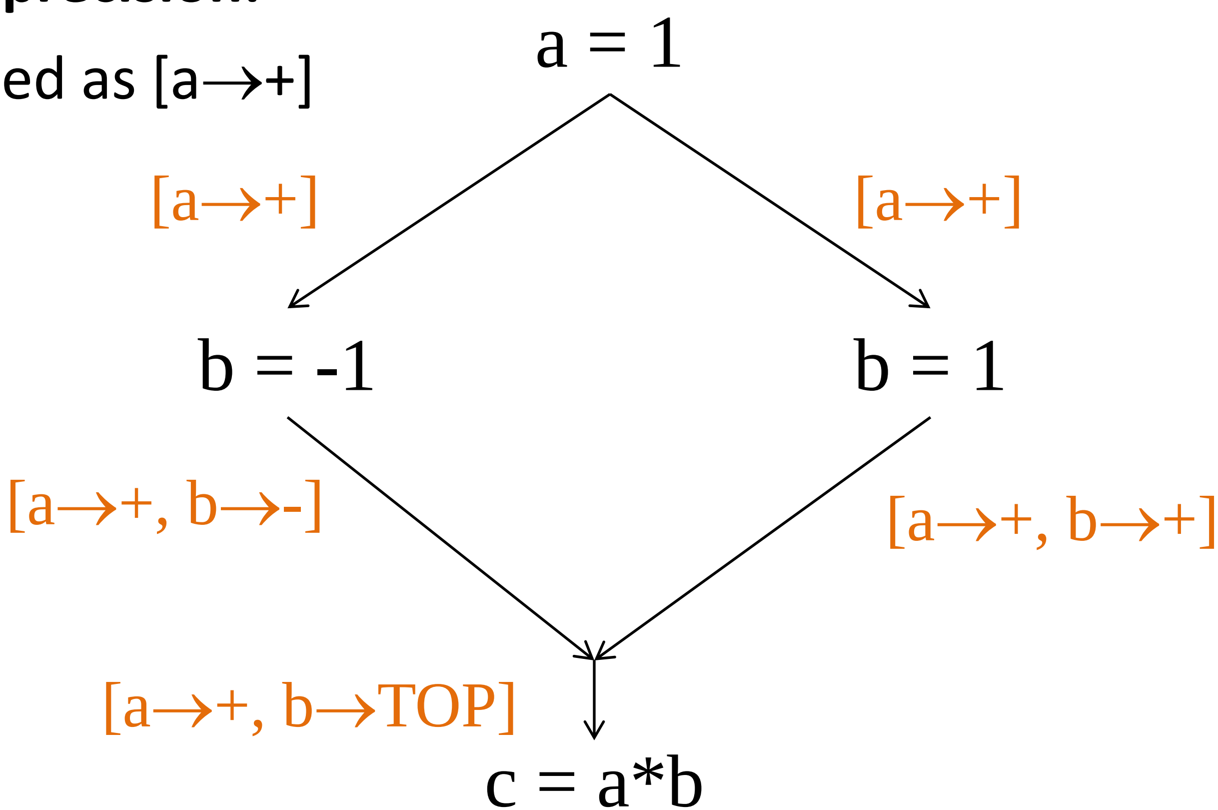
[b→TOP] summarizes results of all executions.

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

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)

Example (almost as) from the last time

```
int x = input()
int y = 0

if x > 0
      y = x + 1
else
      y = 1
```

// Specification:
// y > 0 after the run

Example from the last time

```
int x = input()
int y = 0

if x > 0
      y = x + 1
else
      y = -x
```

// Specification:
// $y \geq 0$ after the run

Interval Analysis

Interval analysis - compute the interval of each variable v

Propagate information:

- [a,b]
- a is the lower bound of the interval
- b is the upper bound of the interval

Interval Analysis (informal)

Abstraction function:

AF(v1, … vn) = { [v1, v1], … [vn, vn] }

Transfer function:

For each expression or a statement, e.g.,

for $z = x + y$ where $x \in [x_{min}, x_{max}]$ and $y \in [y_{min}, y_{max}]$

- Lower bound: $x_{min} + y_{min}$
- Upper bound: $x_{max} + y_{max}$

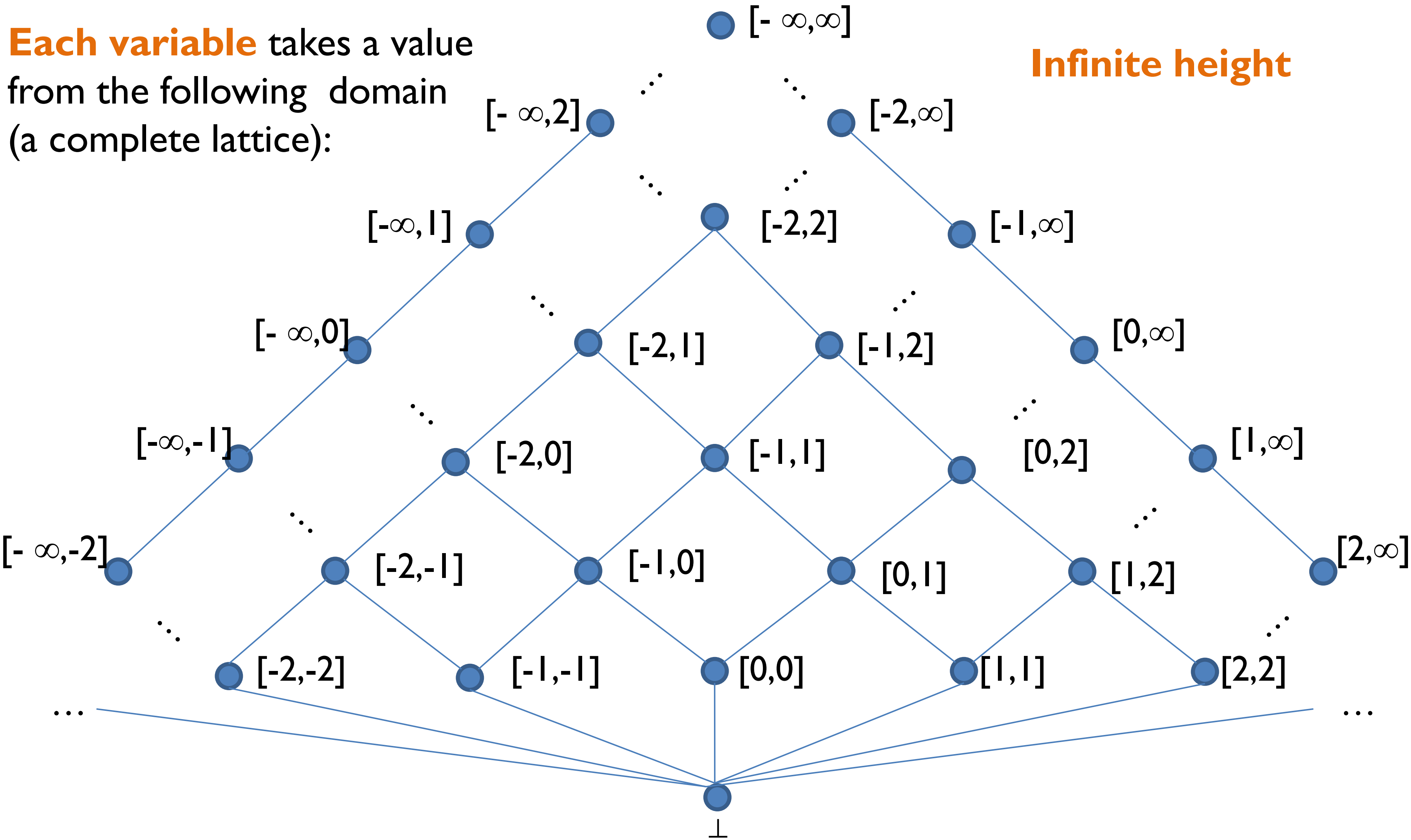
for $z = x * y$ where $x \in [x_{min}, x_{max}]$ and $y \in [y_{min}, y_{max}]$

- Lower bound: **min** $(x_{min}*y_{min}, x_{min}*y_{max}, x_{max}*y_{min}, x_{max}*y_{max})$
- Upper bound: **max** $(x_{min}*y_{min}, x_{min}*y_{max}, x_{max}*y_{min}, x_{max}*y_{max})$

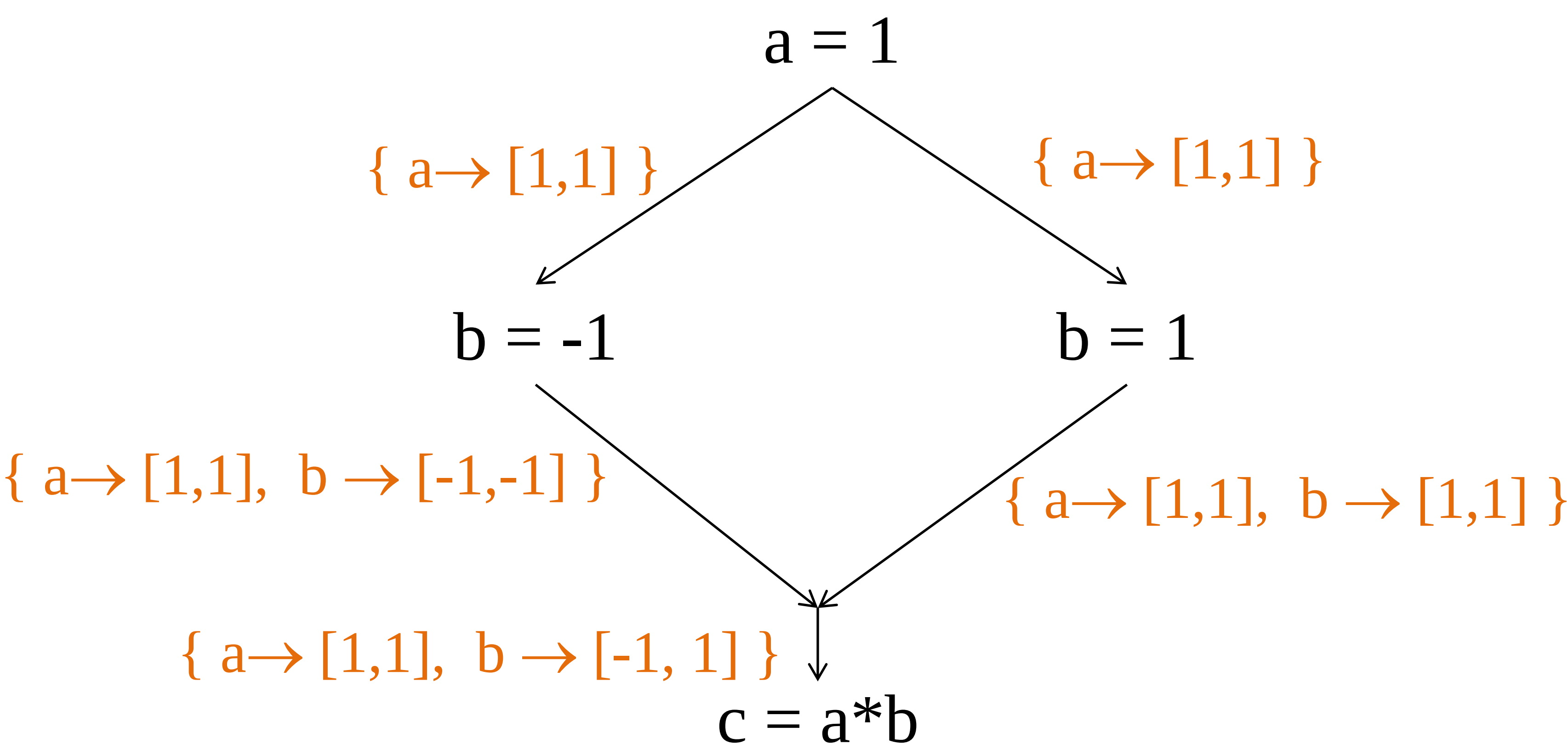
Interval Lattice (for Integers)

Each variable takes a value from the following domain (a complete lattice):

Infinite height



Interval Analysis Example



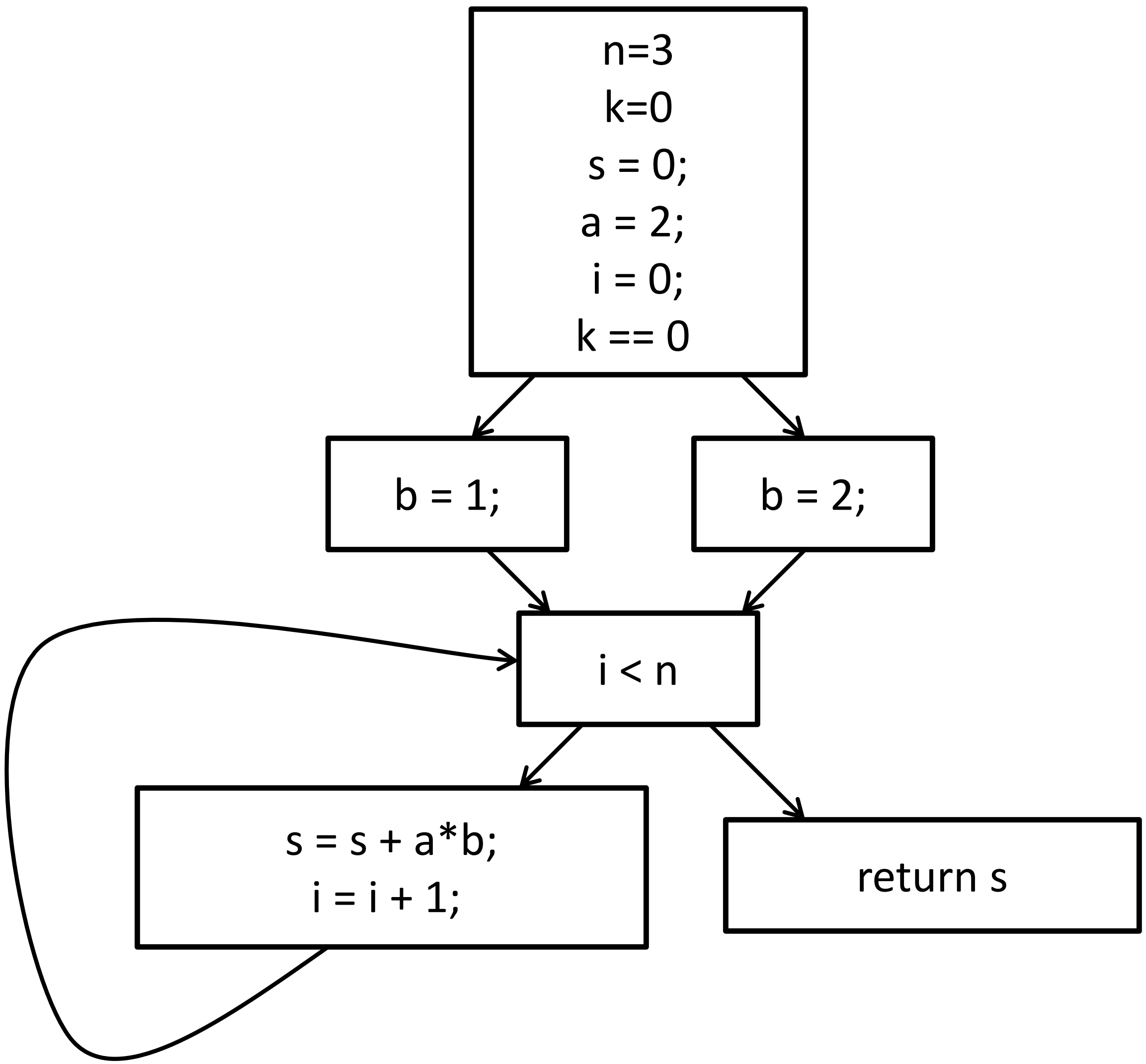
We again have imprecision:

Values of b and c cannot be zero in any concrete execution

And it is different from symbolic execution!

Interval Analysis Another Example

(try at home)



General Sources of Imprecision

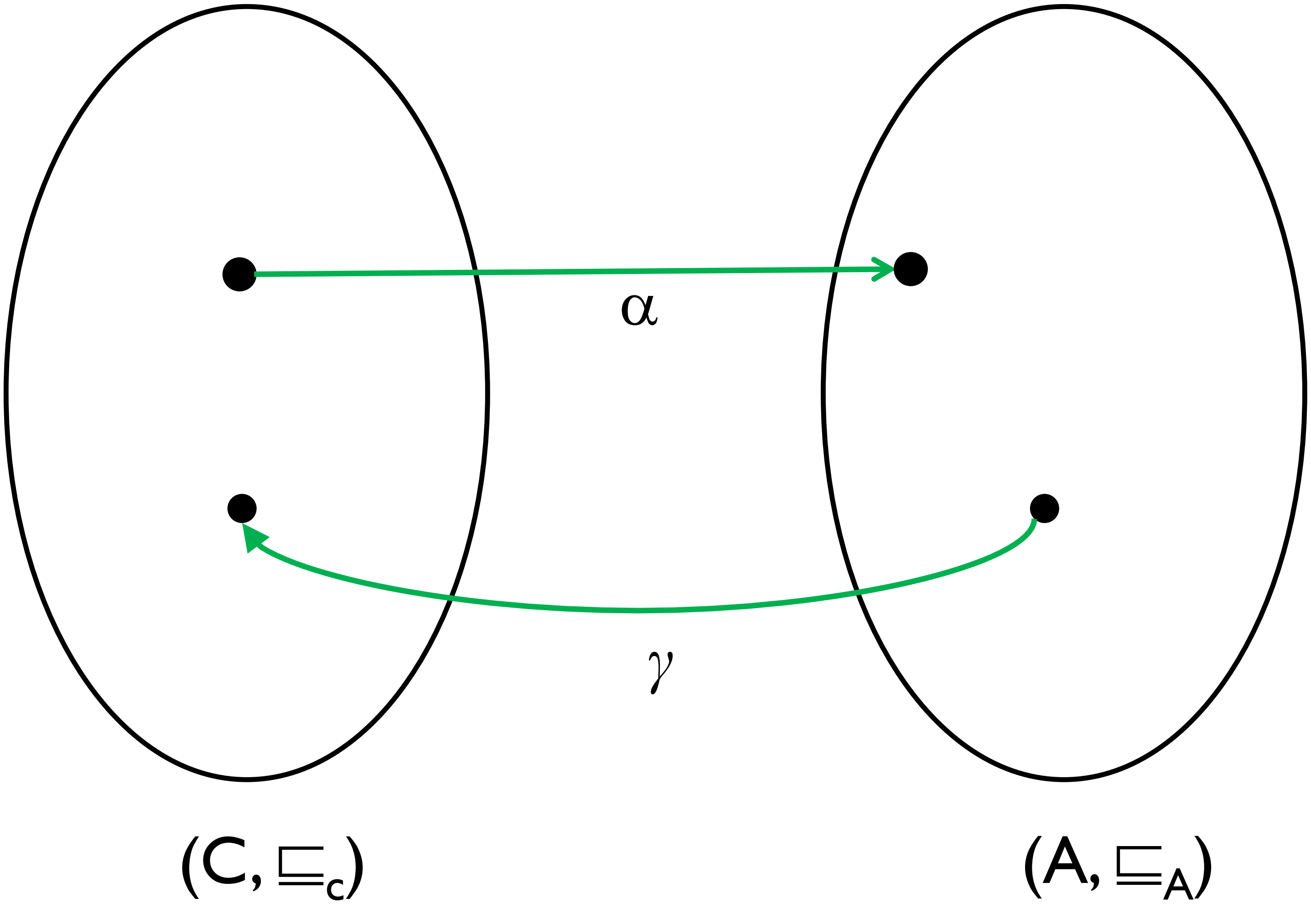
Abstraction Imprecision

- Concrete values (integers) abstracted as lattice values (-,0, and +) or [a,b]
- 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
- Values from different execution paths are combined such that they result in lattice elements not present in any particular execution

Connecting Concrete with Abstract



Why To Allow Imprecision?

Make analysis tractable

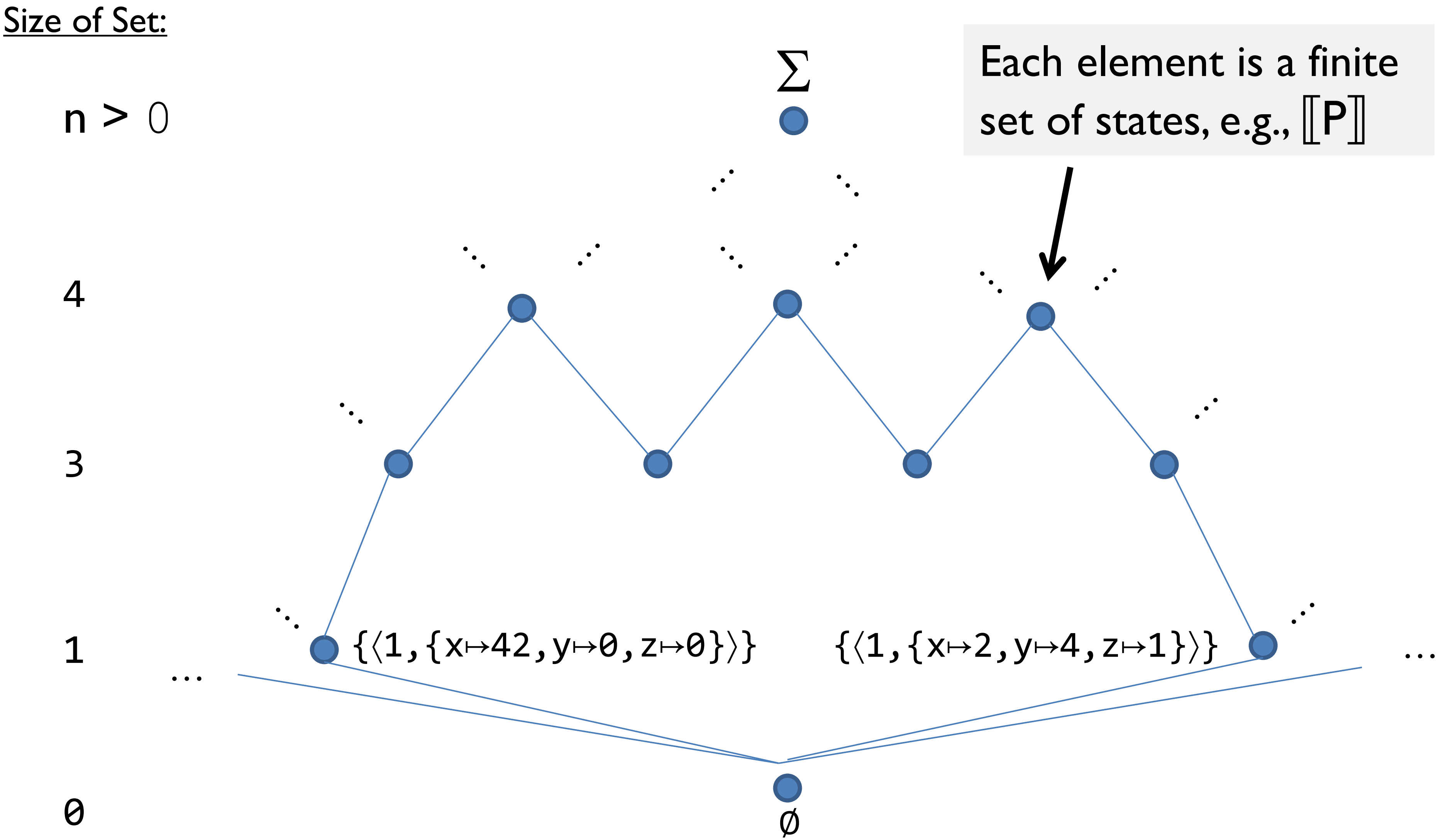
Unbounded sets of values in execution

- Typically abstracted by finite set of lattice values

Execution may visit unbounded set of states

- Abstracted by computing joins of different paths

Domain of Program States



Reaching Definitions

A variable definition reaches the use of the same variable if the value written by the definition may be read by the use

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

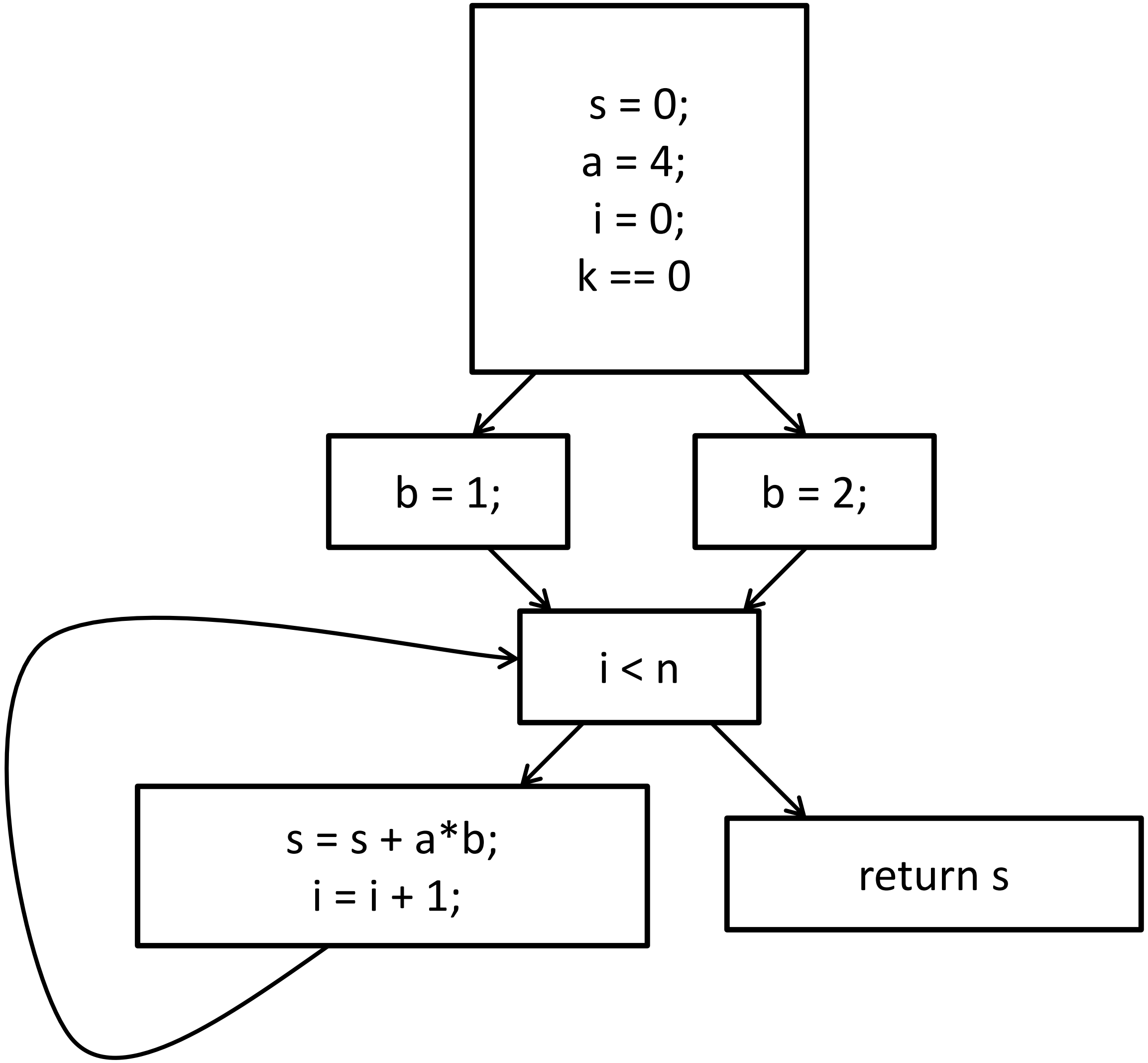
Reaching Definitions

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

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.

Some basic terms:

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


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

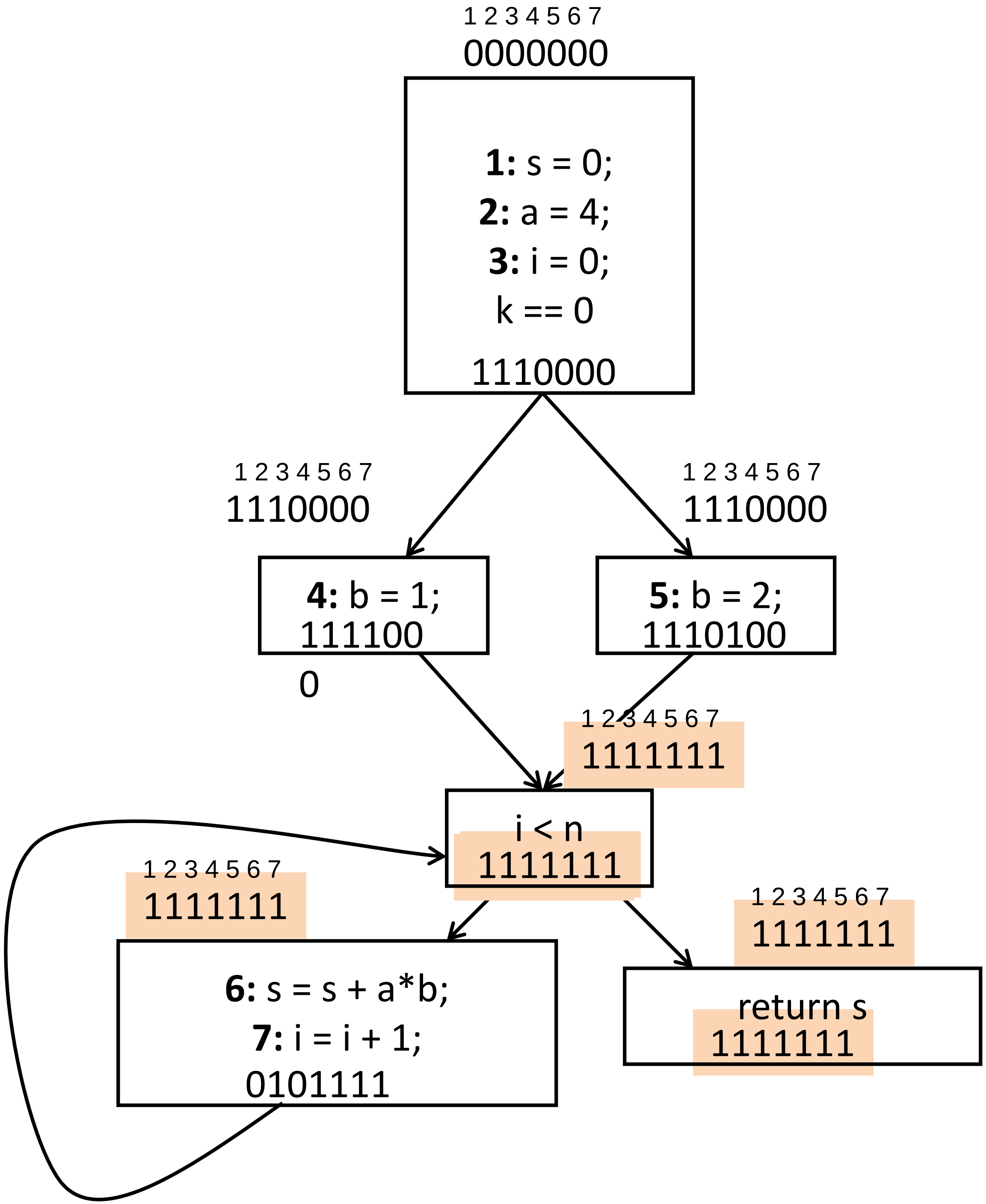
Compute with sets of definitions

- 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;
111100
0
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

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

Example:

- GEN[**6:** s = s + a*b; **7:** i = i + 1;] = 0000011
- KILL[**6:** s = s + a*b; **7:** i = i + 1;] = 1010000

Compiler scans each basic block to derive GEN and KILL sets

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

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 $x = z$

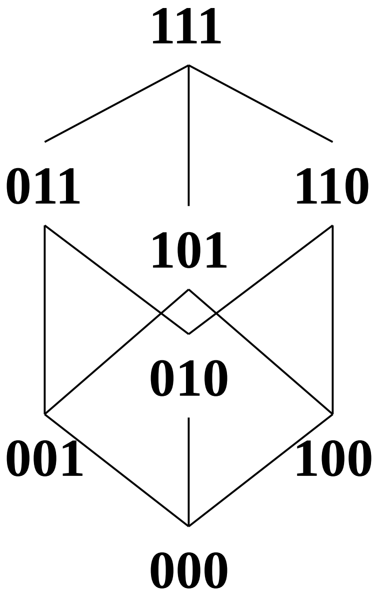
Conceptually, y covers x if there are no elements between x and y

Example

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

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

$x \leq y$ **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 $x \wedge y$ <u>and</u> $x \vee y$,
then P is a **lattice**.

If for all $S \subseteq P$ there exist $\wedge S$ and $\vee S$
then P is a **<u>complete</u> 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 ($\top$)

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

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

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

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'): $x \vee y = y \quad \Rightarrow \quad x = x \wedge y$

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

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

$$
\begin{aligned}
y &= y \vee (y \wedge x) && \text{(by absorption)} \\
&= y \vee (x \wedge y) && \text{(by commutativity)} \\
&= y \vee x && \text{(by assumption)} \\
&= x \vee y && \text{(by commutativity)}
\end{aligned}
$$

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

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

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

Proof of reflexivity property. Must show that

$$x \vee x = x, \text{ which follows directly}$$

$$x \vee x = x \qquad \text{(by idempotence)}$$

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 $L \leq x \wedge y$, 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

Chains

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

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

$(S, \leq)$ is finite if it has the finite height.

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

- 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 $\perp^* = (\perp_1, \perp_2)$, we define

$glb(Y) = (glb\ \{\ l_1\ |(l_1, \_\ ) \in \mathrm{Y}, glb\ \{\ l_2\ |(\ \_\ , l_2) \in Y)$ 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 $\perp^f = \lambda s \,.\, \perp$ and
$glb(Y) = \lambda s \;.\; glb_0 \{\, f(s) \;|\; f \in Y)$ 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 $\perp^f = \lambda s \,.\, \perp_2$ and

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