# CSE 503

Software Engineering

Winter 2021

**Intro to Abstract Interpretation**

January 13, 2021

# Recap: static vs. dynamic analysis

**Static analysis**

- Reason about the program without executing it.
- Build an abstraction of run-time states.
- Reason over abstract domain.
- Prove a property of the program.
- Sound* but conservative.

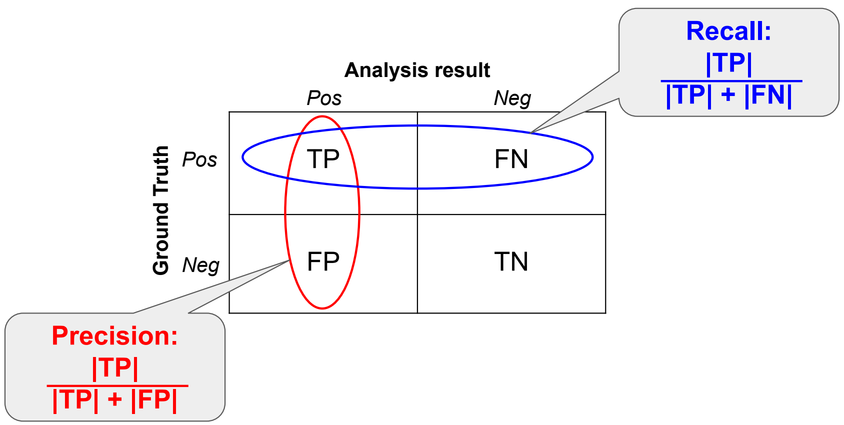
**Dynamic analysis**

- Reason about the program based on some program executions.
- Observe concrete behavior at run time.
- Improve confidence in correctness.
- Unsound* but precise.

* Some static analyses are unsound; dynamic analyses can be sound.

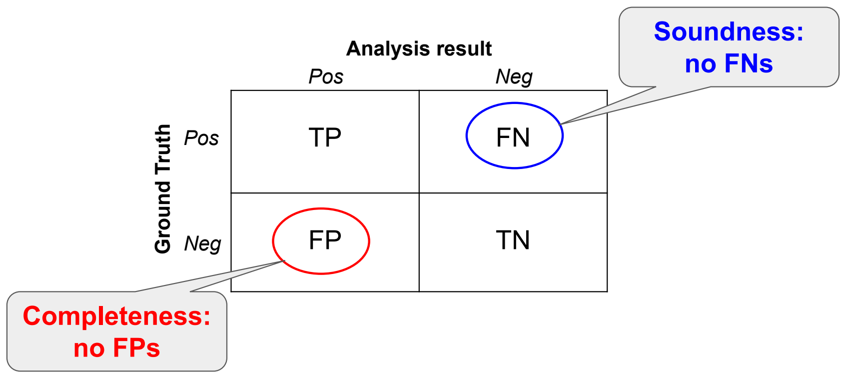
# Recap: Terminology and important concepts

1. Precision vs. Recall (and FP/FN/TP/TN)




# Recap: Terminology and important concepts

1. Precision vs. Recall (and FP/FN/TP/TN)
2. Soundness vs. Completeness



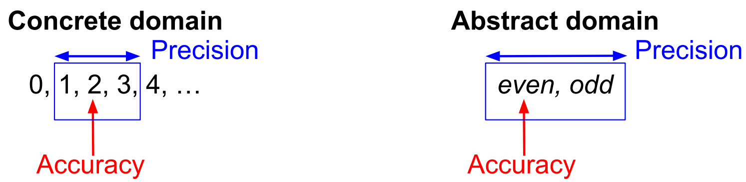
## Recap: Terminology and important concepts

1. Precision vs. Recall (and FP/FN/TP/TN)
2. Soundness vs. Completeness
3. Concrete domain vs. Abstract domain

| **Concrete domain** | **Abstract domain** |
|---|---|
| 0, 1, 2, 3, 4, … | *even, odd* |

## Recap: Terminology and important concepts

1. Precision vs. Recall (and FP/FN/TP/TN)
2. Soundness vs. Completeness
3. Concrete domain vs. Abstract domain
4. Accuracy vs. Precision

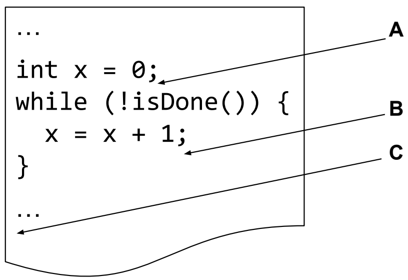



## Today

- Abstract interpretation
  - Introduction
  - Abstraction functions
  - Concretization functions
  - Transfer functions
  - Lattices

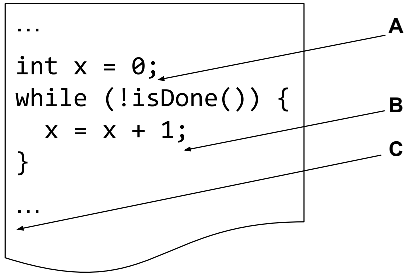
## Properties of an ideal program analysis

- Soundness
- Completeness
- Termination



# Properties of an ideal program analysis

- Soundness
- Completeness
- Termination




Abstract interpretation sacrifices completeness (precision)

# Abstract interpretation: applications

Compiler checks and optimizations

- Liveness analysis (register reallocation)
- Reachability analysis (dead code elimination)
- Code motion (`while(cond){x = comp(); ...}`)

# Abstract interpretation: code examples

**Liveness**

```
public class Liveness {
  public void liveness() {
    int a;
    if (alwaysTrue()) {
      a = 1;
    }
    System.out.println(a);
  }
}
```

**Reachability**

```
public void deadCode() {

  return;

  System.out.println("Here!");
}
```

# Abstract interpretation: example

**Program**

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

Are all statements necessary?

# Abstract interpretation: example

**Program**

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

**SSA form**

```
x1 = 0;
y1 = read_even();
x2 = y1 + 1;
y2 = 2 * x2;
x3 = y2 - 2;
y3 = x3 / 2;
```

$X_1$ is never read.

# Abstract interpretation: example

**Program**

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

**SSA form**

```
x1 = 0;
y1 = read_even();
x2 = y1 + 1;
y2 = 2 * x2;
x3 = y2 - 2;
y3 = x3 / 2;
```

$y_3 = x_3 / 2$
$y_3 = (y_2 - 2) / 2$
$y_3 = (2 * x_2 - 2) / 2$
$y_3 = (2 * (y_1 + 1) - 2) / 2$
$y_3 = (2 * y_1 + 2 - 2) / 2$
$y_3 = y_1$
$x_3 = y_1 * 2$

Symbolic reasoning shows simplification potential.

# Abstraction function

**Program**

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

**Concrete execution**

```
{x=0;  y=undef}
{x=0;  y=8}
{x=9;  y=8}
{x=9;  y=18}
{x=16; y=18}
{x=16; y=8}
```

# Abstraction function

**Program**

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

**Concrete execution** / **Mapping to abstract domain (even, odd)**

```
{x=0;  y=undef}
{x=0;  y=8}      →  {x=e; y=e}
{x=9;  y=8}      →  {x=o; y=e}
{x=9;  y=18}     →  {x=o; y=e}
{x=16; y=18}     →  {x=e; y=e}
{x=16; y=8}      →  {x=e; y=e}
```

## Abstraction function

**Program**

```
x = 0;
y = read_even();
x = y + 1;
y = 2 * x;
x = y - 2;
y = x / 2;
```

| Concrete execution | Mapping to abstract domain (even, odd) |
|---|---|
| {x=0; y=undef} | |
| {x=0; y=8} | → {x=e; y=e} |
| {x=9; y=8} | → {x=o; y=e} |
| {x=9; y=18} | → {x=o; y=e} |
| {x=16; y=18} | → {x=e; y=e} |
| {x=16; y=8} | → {x=e; y=e} |

But, what’s the purpose of the abstraction function?

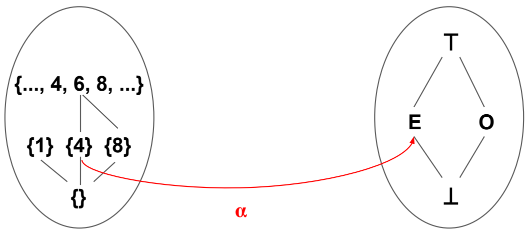
## Abstraction function

Concrete (*P*( ℕ))　　　Abstract


What is the abstraction (α) of {4}?

## Abstraction function

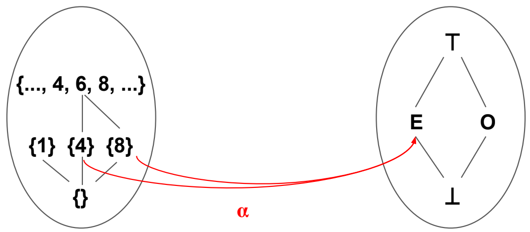
Concrete (*P*( ℕ))　　　Abstract




What is the abstraction (α) of {8}?

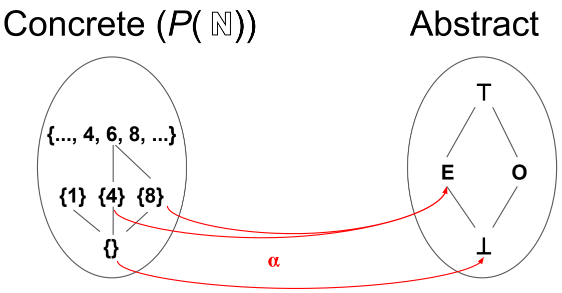
## Abstraction function

Concrete (*P*( ℕ))　　　Abstract

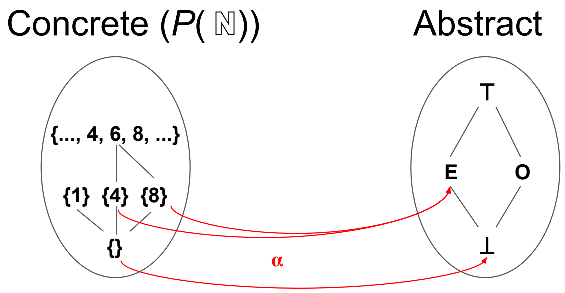



What is the abstraction (α) of {}?

## Abstraction function

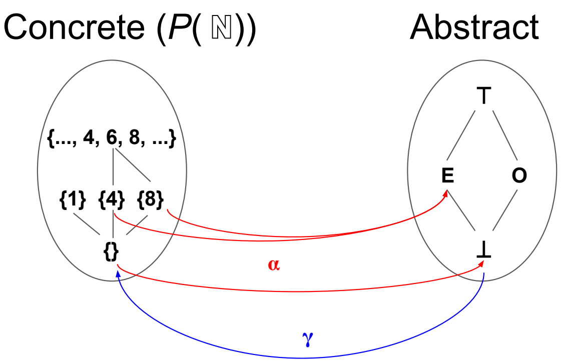



## Concretization function



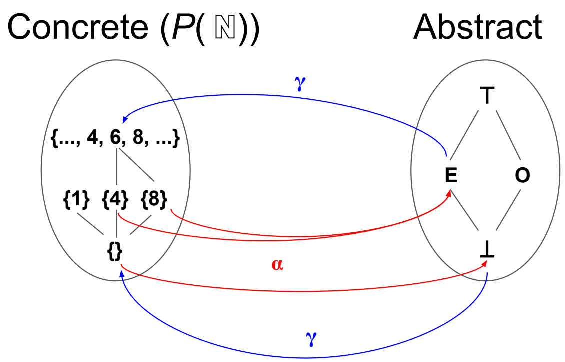

What is the concretization (γ) of ⊥?

## Concretization function

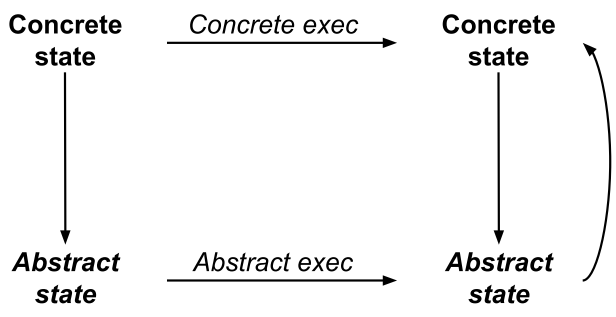



What is the concretization (γ) of **E** ?

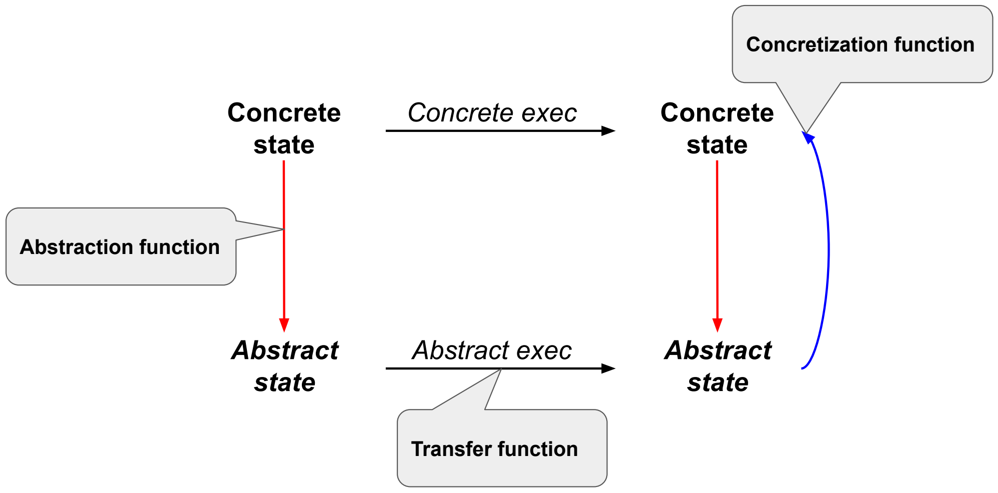
## Concretization function



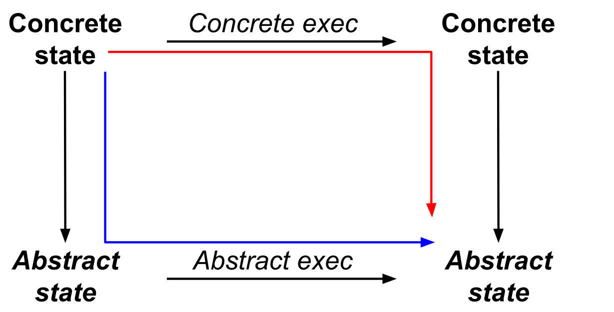
# Transfer function




# Transfer function

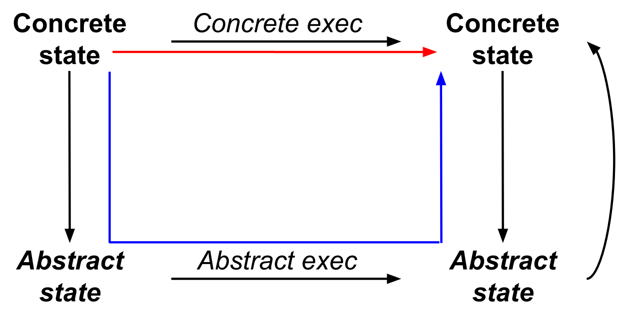



# Abstract interpretation: approximation




Do both paths lead to the same abstract state?

# Abstract interpretation: approximation




Do both paths lead to the same concrete state?

## Set, semilattice, lattice

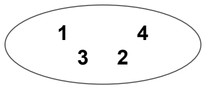
## Set, semilattice, lattice

**Set**

## Set, semilattice, lattice

**Set**

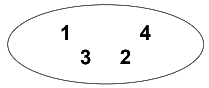
- **unordered** collection of **distinct** objects




## Set, semilattice, lattice

**Set**

- unordered collection of distinct objects

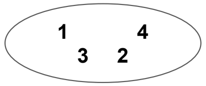



**Partially ordered set**

# Set, semilattice, lattice

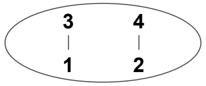
**Set**

- unordered collection of distinct objects




**Partially ordered set**

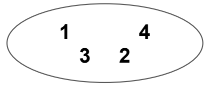
- Binary relationship $\leq$:
  - Reflexive: $x \leq x$
  - Anti-symmetric: $x \leq y \wedge y \leq x => x = y$
  - Transitive: $x \leq y \wedge y \leq z => x \leq z$




# Set, semilattice, lattice

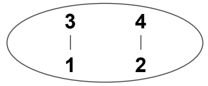
**Set**

- unordered collection of distinct objects




**Partially ordered set**

- Binary relationship $\leq$:
  - Reflexive: $x \leq x$
  - Anti-symmetric: $x \leq y \wedge y \leq x => x = y$
  - Transitive: $x \leq y \wedge y \leq z => x \leq z$



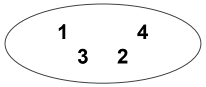

**Join semilattice**

**Meet semilattice**

# Set, semilattice, lattice

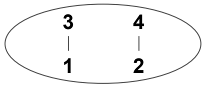
**Set**

- unordered collection of distinct objects



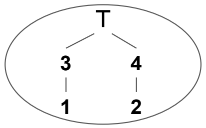

**Partially ordered set**

- Binary relationship $\leq$:
  - Reflexive: $x \leq x$
  - Anti-symmetric: $x \leq y \wedge y \leq x => x = y$
  - Transitive: $x \leq y \wedge y \leq z => x \leq z$



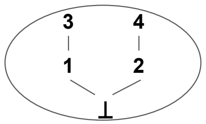

**Join semilattice**

- Partially ordered set with least upper bound (join)




**Meet semilattice**

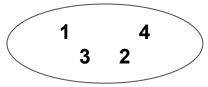
- Partially ordered set with greatest lower bound (meet)




# Set, semilattice, lattice

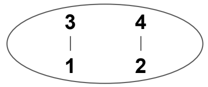
**Set**

- unordered collection of distinct objects



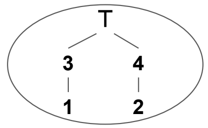

**Partially ordered set**

- Binary relationship $\leq$:
  - Reflexive: $x \leq x$
  - Anti-symmetric: $x \leq y \wedge y \leq x => x = y$
  - Transitive: $x \leq y \wedge y \leq z => x \leq z$



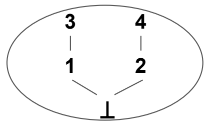

**Join semilattice**

- Partially ordered set with least upper bound (join)



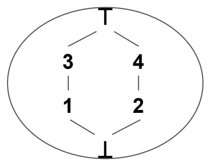

**Meet semilattice**

- Partially ordered set with greatest lower bound (meet)



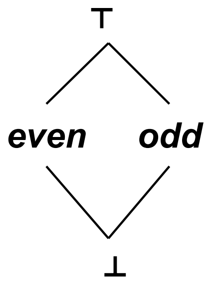

**Lattice**

- Both a join semilattice and a meet semilattice



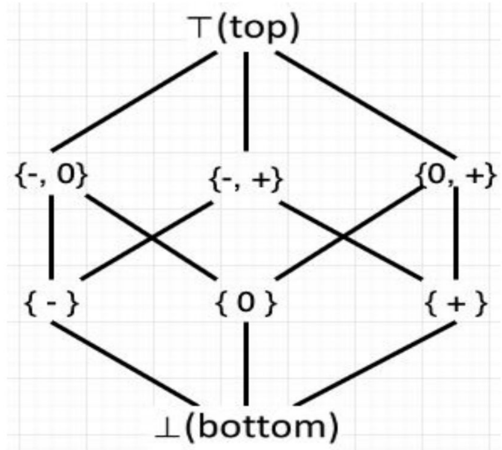
## Lattice: example

**Abstract domain: *even, odd, unknown* (⊤ ), *{}* ( ⊥)**




## Lattice: example

**Abstract domain: *-, 0, +, unknown, {}***




## Lattice: example

**Goal: approximate the values of x after the loop**

```
…
int x = 0;
while (!isDone()) {
  x = x + 1;
}
…
```

What are possible abstract domains and their trade-offs?

## Lattice: example

**Goal: approximate the values of x after the loop**

```
…
int x = 0;
while (!isDone()) {
  x = x + 1;
}
…
```

**Possible abstract domains:**

- **Powerset of set of integers**
- **Intervals**
- **...**