Static Program Analysis

Yue Li and Tian Tan

2020 Spring

Static Program Analysis

Data Flow Analysis — Foundations

Nanjing University

Yue Li

2020

Contents (1)

Contents (II)

Let us first recall the iterative algorithm for data flow analysis

This general iterative algorithm produces a solution to data flow analysis

Iterative Algorithm for May & Forward Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

```
OUT[entry] = ∅;
for (each basic block B\entry)
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = ∪_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B ∪ (IN[B] - kill_B);
    }
```

View Iterative Algorithm in Another Way

- Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.

View Iterative Algorithm in Another Way

- Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.
- Assume the domain of the values in data flow analysis is V, then we can define a k-tuple

$$(\mathrm{OUT}[n_1], \mathrm{OUT}[n_2], \ldots, \mathrm{OUT}[n_k])$$

as an element of set $(V_1 \times V_2 \ldots \times V_k)$ denoted as V^k , to hold the values of the analysis after each iteration.

View Iterative Algorithm in Another Way

- Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.
- Assume the domain of the values in data flow analysis is V, then we can define a k-tuple

$$(\mathrm{OUT}[n_1], \mathrm{OUT}[n_2], \ldots, \mathrm{OUT}[n_k])$$

as an element of set $(V_1 \times V_2 \ldots \times V_k)$ denoted as V^k , to hold the values of the analysis after each iteration.

- Each iteration can be considered as taking an action to map an element of V^k to a new element of V^k, through applying the transfer functions and control-flow handing, abstracted as a function $F: V^k \to V^k$

View Iterative Algorithm in Another Way

- Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.
- Assume the domain of the values in data flow analysis is V, then we can define a k-tuple

$$(\mathrm{OUT}[n_1], \mathrm{OUT}[n_2], \ldots, \mathrm{OUT}[n_k])$$

as an element of set $(V_1 \times V_2 \ldots \times V_k)$ denoted as V^k , to hold the values of the analysis after each iteration.

- Each iteration can be considered as taking an action to map an element of V^k to a new element of V^k, through applying the transfer functions and control-flow handing, abstracted as a function $F: V^k \rightarrow V^k$
- Then the algorithm outputs a series of k-tuples iteratively until a k-tuple is the same as the last one in two consecutive iterations

OUT[*entry*] = ∅;
for (each basic block *B**entry*)
    OUT[*B*] = ∅;
while (changes to any OUT occur)
    for (each basic block *B**entry*) {
        IN[*B*] = $\bigcup_{P \text{ a predecessor of } B}$ OUT[*P*];
        OUT[*B*] = gen_B U (IN[*B*] - $kill_B$);
    }

Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.

$$init \rightarrow (\bot, \ \bot, \ \ldots, \ \bot)$$

```
OUT[entry] = Ø;
for (each basic block B\entry)
    OUT[B] = Ø;
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = U_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B U (IN[B] - kill_B);
    }
```

Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.

$$init \rightarrow (\bot, \ \bot, \ \ldots, \ \bot)$$

$$iter\ 1 \rightarrow (v_1^1, v_2^1, \ldots, v_k^1)$$

```
OUT[entry] = ∅;
for (each basic block B\entry)
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = ∪_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B ∪ (IN[B] - kill_B);
    }
```

Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.

$$\textit{init} \rightarrow (\bot,\ \bot,\ \ldots,\ \bot)$$

$$\textit{iter 1} \rightarrow (v_1^1, v_2^1, \ldots, v_k^1)$$

$$\textit{iter 2} \rightarrow (v_1^2, v_2^2, \ldots, v_k^2)$$

```
OUT[entry] = ∅;
for (each basic block B\entry)
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = ∪_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B ∪ (IN[B] - kill_B);
    }
```

Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.

$$\textit{init} \rightarrow (\bot, \ \bot, \ \ldots, \ \bot)$$

$$\textit{iter } 1 \rightarrow (v_1^1, v_2^1, \ldots, v_k^1)$$

$$\textit{iter } 2 \rightarrow (v_1^2, v_2^2, \ldots, v_k^2)$$

$$\vdots$$

$$\textit{iter } i \rightarrow (v_1^i, v_2^i, \ldots, v_k^i)$$

OUT[*entry*] = $\emptyset$;
for (each basic block *B**entry*)
    OUT[*B*] = $\emptyset$;
while (changes to any OUT occur)
    for (each basic block *B**entry*) {
        IN[*B*] = $\bigcup_{P \text{ a predecessor of } B}$ OUT[*P*];
        OUT[*B*] = $gen_B \cup (\text{IN}[B] - kill_B)$;
    }

Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.

$$
\begin{array}{ll}
init & \rightarrow (\bot, \ \bot, \ \ldots, \ \bot) \\
iter\ 1 & \rightarrow (v_1^1, v_2^1, \ldots, v_k^1) \\
iter\ 2 & \rightarrow (v_1^2, v_2^2, \ldots, v_k^2) \\
 & \quad \vdots \\
iter\ i & \rightarrow (v_1^i, v_2^i, \ldots, v_k^i) \\
iter\ i+1 & \rightarrow (v_1^i, v_2^i, \ldots, v_k^i)
\end{array}
$$

```
OUT[entry] = ∅;
for (each basic block B\entry)
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = ∪_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B ∪ (IN[B] - kill_B);
    }
```

Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.

$$\begin{aligned}
&\textit{init} \rightarrow (\bot, \bot, \ldots, \bot) = X_0 \\
&\textit{iter 1} \rightarrow (v_1^1, v_2^1, \ldots, v_k^1) = X_1 \\
&\textit{iter 2} \rightarrow (v_1^2, v_2^2, \ldots, v_k^2) = X_2 \\
&\qquad \vdots \\
&\textit{iter i} \rightarrow (v_1^i, v_2^i, \ldots, v_k^i) = X_i \\
&\textit{iter i+1} \rightarrow (v_1^i, v_2^i, \ldots, v_k^i) = X_{i+1}
\end{aligned}$$

```
OUT[entry] = ∅;
for (each basic block B\entry)
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = ∪_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B ∪ (IN[B] - kill_B);
    }
```

Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.

Each iteration takes an action $F: V^k \rightarrow V^k$

$$\begin{array}{lll}
\textit{init} & \rightarrow (\bot, \ \bot, \ \ldots, \ \bot) & = X_0 \\
\textit{iter 1} & \rightarrow (v_1^1, v_2^1, \ldots, v_k^1) & = X_1 = F(X_0) \\
\textit{iter 2} & \rightarrow (v_1^2, v_2^2, \ldots, v_k^2) & = X_2 = F(X_1) \\
 & \vdots & \\
\textit{iter i} & \rightarrow (v_1^i, v_2^i, \ldots, v_k^i) & = X_i = F(X_{i-1}) \\
\textit{iter i+1} & \rightarrow (v_1^i, v_2^i, \ldots, v_k^i) & = X_{i+1} = F(X_i)
\end{array}$$

```
OUT[entry] = ∅;
for (each basic block B\entry)
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = ∪_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B ∪ (IN[B] - kill_B);
    }
```

Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.

Each iteration takes an action $F: V^k \rightarrow V^k$

$$\textit{init} \rightarrow (\bot, \ \bot, \ \ldots, \ \bot) = X_0$$

$$\textit{iter 1} \rightarrow (v_1^1, v_2^1, \ldots, v_k^1) = X_1 = F(X_0)$$

$$\textit{iter 2} \rightarrow (v_1^2, v_2^2, \ldots, v_k^2) = X_2 = F(X_1)$$

$$\vdots$$

$$\textit{iter i} \rightarrow (v_1^i, v_2^i, \ldots, v_k^i) = X_i = F(X_{i-1})$$

$$\textit{iter i+1} \rightarrow (v_1^i, v_2^i, \ldots, v_k^i) = X_{i+1} = F(X_i)$$

```
OUT[entry] = ∅;
for (each basic block B\entry)
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = ∪_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B ∪ (IN[B] - kill_B);
    }
```

Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.

Each iteration takes an action $F: V^k \rightarrow V^k$

$$\textit{init} \rightarrow (\bot, \bot, \ldots, \bot) = X_0$$

$$\textit{iter } 1 \rightarrow (v_1^1, v_2^1, \ldots, v_k^1) = X_1 = F(X_0)$$

$$\textit{iter } 2 \rightarrow (v_1^2, v_2^2, \ldots, v_k^2) = X_2 = F(X_1)$$

$$\vdots$$

$$\textit{iter } i \rightarrow (v_1^i, v_2^i, \ldots, v_k^i) = X_i = F(X_{i-1}) \qquad \because X_i = X_{i+1}$$

$$\textit{iter } i+1 \rightarrow (v_1^i, v_2^i, \ldots, v_k^i) = X_{i+1} = F(X_i) \qquad \therefore X_i = X_{i+1} = F(X_i)$$

```
OUT[entry] = ∅;
for (each basic block B\entry)
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = ∪_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B ∪ (IN[B] - kill_B);
    }
```

Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.

Each iteration takes an action
$F: V^k \rightarrow V^k$

init → $(\perp, \perp, \ldots, \perp) = X_0$

iter 1 → $(v_1^1, v_2^1, \ldots, v_k^1) = X_1$

iter 2 → $(v_1^2, v_2^2, \ldots, v_k^2) = X_2$

$\vdots$

iter i → $(v_1^i, v_2^i, \ldots, v_k^i) = X_i = F(X_{i-1})$ $\quad \because X_i = X_{i+1}$

iter i+1 → $(v_1^i, v_2^i, \ldots, v_k^i) = X_{i+1} = F(X_i)$ $\quad \therefore X_i = X_{i+1} = F(X_i)$

X is a fixed point of function F if
$X = F(X)$

```
OUT[entry] = ∅;
for (each basic block B\entry)
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block B\entry) {
        IN[B] = ∪_{P a predecessor of B} OUT[P];
        OUT[B] = gen_B ∪ (IN[B] - kill_B);
    }
```

Given a CFG (program) with k nodes, the iterative algorithm updates OUT[n] for every node n in each iteration.

Each iteration takes an action
$F: V^k \rightarrow V^k$

init $\rightarrow (\bot, \bot, \ldots, \bot) = X_0$

iter 1 $\rightarrow (v_1^1, v_2^1, \ldots, v_k^1) =$

iter 2 $\rightarrow (v_1^2, v_2^2, \ldots, v_k^2) =$

$\vdots$

iter i $\rightarrow (v_1^i, v_2^i, \ldots, v_k^i) =$

iter i+1 $\rightarrow (v_1^i, v_2^i, \ldots, v_k^i) = X_{i+1} = F(X_i)$ $\quad \therefore X_i = X_{i+1} = F(X_i)$

X is a fixed point of function F if
$X = F(X)$

The iterative algorithm reaches
a fixed point

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

- Is the algorithm guaranteed to terminate or reach the fixed point, or does it always have a solution?

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

- Is the algorithm guaranteed to terminate or reach the fixed point, or does it always have a solution?

- If so, is there only one solution or only one fixed point? If more than one, is our solution the best one (most precise)?

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

- Is the algorithm guaranteed to terminate or reach the fixed point, or does it always have a solution?

- If so, is there only one solution or only one fixed point? If more than one, is our solution the best one (most precise)?

- When will the algorithm reach the fixed point, or when can we get the solution?

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

- Is the algorithm guaranteed to terminate or reach the fixed point, or does it always have a solution?
- If so, is there only one solution or only one fixed point? If more than one, is our solution the best one (most precise)?
- When will the algorithm reach the fixed point, or when can we get the solution?

To answer these questions, let us learn some math first

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 1. Is $(S, \sqsubseteq)$ a poset where S is a set of integers and $\sqsubseteq$ represents $\leq$ (less than or equal to)?

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 1. Is $(S, \sqsubseteq)$ a poset where S is a set of integers and $\sqsubseteq$ represents $\leq$ (less than or equal to)?

(1) *Reflexivity*

(2) *Antisymmetry*

(3) *Transitivity*

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 1. Is $(S, \sqsubseteq)$ a poset where S is a set of integers and $\sqsubseteq$ represents $\leq$ (less than or equal to)?

(1) *Reflexivity* $1 \leq 1, 2 \leq 2$

(2) *Antisymmetry*

(3) *Transitivity*

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 1. Is $(S, \sqsubseteq)$ a poset where S is a set of integers and $\sqsubseteq$ represents $\leq$ (less than or equal to)?

✓ (1) *Reflexivity* $1 \leq 1, 2 \leq 2$

(2) *Antisymmetry*

(3) *Transitivity*

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 1. Is $(S, \sqsubseteq)$ a poset where S is a set of integers and $\sqsubseteq$ represents $\leq$ (less than or equal to)?

✓ (1) *Reflexivity* $1 \leq 1, 2 \leq 2$

(2) *Antisymmetry* $x \leq y \wedge y \leq x$ then $x = y$

(3) *Transitivity*

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 1. Is $(S, \sqsubseteq)$ a poset where S is a set of integers and $\sqsubseteq$ represents $\leq$ (less than or equal to)?

✓ (1) *Reflexivity* $1 \leq 1, 2 \leq 2$

✓ (2) *Antisymmetry* $x \leq y \wedge y \leq x$ then $x = y$

(3) *Transitivity*

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 1. Is $(S, \sqsubseteq)$ a poset where S is a set of integers and $\sqsubseteq$ represents $\leq$ (less than or equal to)?

✓ (1) *Reflexivity* $1 \leq 1, 2 \leq 2$

✓ (2) *Antisymmetry* $x \leq y \wedge y \leq x$ then $x = y$

(3) *Transitivity* $1 \leq 2 \wedge 2 \leq 3$ then $1 \leq 3$

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 1. Is $(S, \sqsubseteq)$ a poset where S is a set of integers and $\sqsubseteq$ represents $\leq$ (less than or equal to)?

✓ (1) *Reflexivity* $1 \leq 1, 2 \leq 2$

✓ (2) *Antisymmetry* $x \leq y \wedge y \leq x$ then $x = y$

✓ (3) *Transitivity* $1 \leq 2 \wedge 2 \leq 3$ then $1 \leq 3$

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \land y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \land y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 2. Is $(S, \sqsubseteq)$ a poset where S is a set of integers and $\sqsubseteq$ represents $<$ (less than)?

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 2. Is $(S, \sqsubseteq)$ a poset where S is a set of integers and $\sqsubseteq$ represents $<$ (less than)?

(1) *Reflexivity*

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 2. Is $(S, \sqsubseteq)$ a poset where S is a set of integers and $\sqsubseteq$ represents $<$ (less than)?

(1) *Reflexivity* $1 < 1, 2 < 2$

Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 2. Is $(S, \sqsubseteq)$ a poset where S is a set of integers and $\sqsubseteq$ represents $<$ (less than)?

❌ (1) *Reflexivity* $1 < 1, 2 < 2$

Partial Order

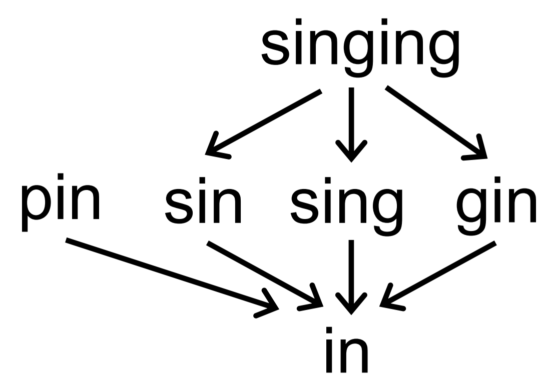
We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \land y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \land y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 3. Is $(S, \sqsubseteq)$ a poset where S is a set of English words and $\sqsubseteq$ represents the *substring* relation, i.e., s1 $\sqsubseteq$ s2 means s1 is a substring of s2?



Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

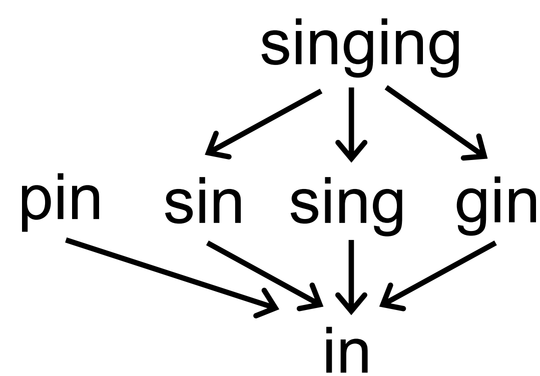
(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 3. Is $(S, \sqsubseteq)$ a poset where S is a set of English words and $\sqsubseteq$ represents the *substring* relation, i.e., s1 $\sqsubseteq$ s2 means s1 is a substring of s2?

(1) *Reflexivity*

(2) *Antisymmetry*

(3) *Transitivity*



Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \land y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

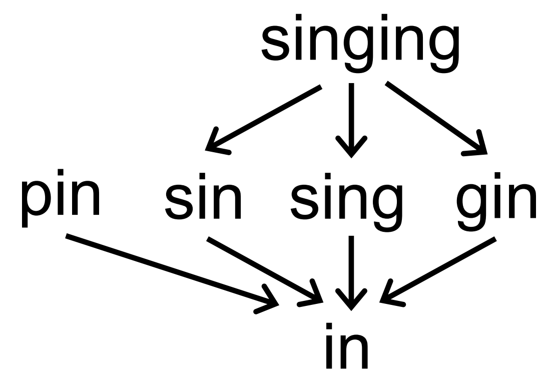
(3) $\forall x, y, z \in P, x \sqsubseteq y \land y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 3. Is $(S, \sqsubseteq)$ a poset where S is a set of English words and $\sqsubseteq$ represents the *substring* relation, i.e., s1 $\sqsubseteq$ s2 means s1 is a substring of s2?

(1) *Reflexivity*

(2) *Antisymmetry*

(3) *Transitivity*



Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

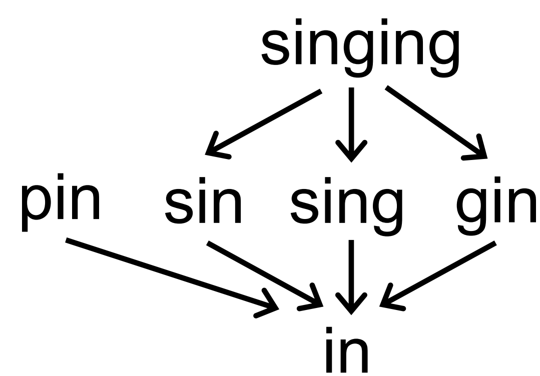
(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 3. Is $(S, \sqsubseteq)$ a poset where S is a set of English words and $\sqsubseteq$ represents the *substring* relation, i.e., s1 $\sqsubseteq$ s2 means s1 is a substring of s2?

(1) *Reflexivity*

(2) *Antisymmetry*

(3) *Transitivity*



Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

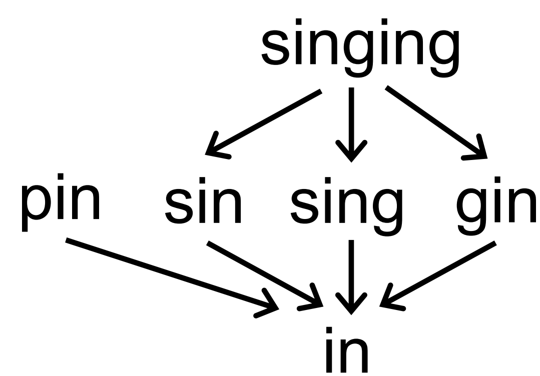
(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 3. Is $(S, \sqsubseteq)$ a poset where S is a set of English words and $\sqsubseteq$ represents the *substring* relation, i.e., s1 $\sqsubseteq$ s2 means s1 is a substring of s2?

(1) *Reflexivity*

(2) *Antisymmetry*

(3) *Transitivity*



Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

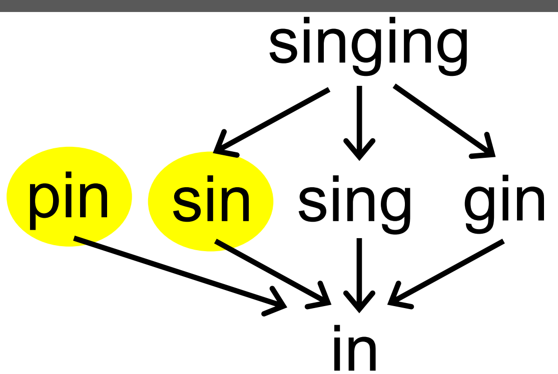
(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

partial means for a pair of set elements in P, they could be incomparable; in other words, not necessary that every pair of set elements must satisfy the ordering $\sqsubseteq$

(1) *Reflexivity*

(2) *Antisymmetry*

(3) *Transitivity*



Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 4. Is $(S, \sqsubseteq)$ a poset where S is the power set of set {a,b,c} and $\sqsubseteq$ represents $\subseteq$ (subset)?

Partial Order

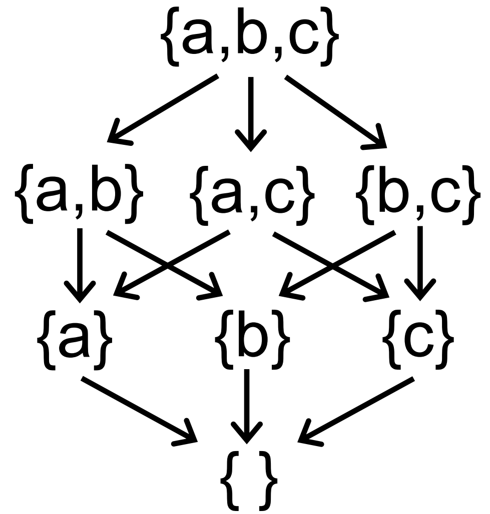
We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \land y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

(3) $\forall x, y, z \in P, x \sqsubseteq y \land y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 4. Is $(S, \sqsubseteq)$ a poset where S is the power set of set {a,b,c} and $\sqsubseteq$ represents $\subseteq$ (subset)?



Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

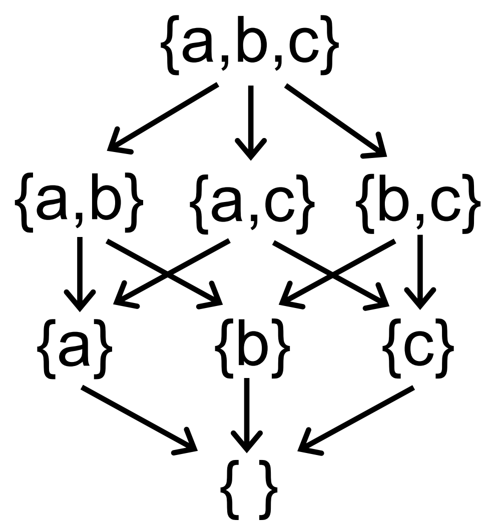
(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 4. Is $(S, \sqsubseteq)$ a poset where S is the power set of set {a,b,c} and $\sqsubseteq$ represents $\subseteq$ (subset)?

(1) *Reflexivity*

(2) *Antisymmetry*

(3) *Transitivity*



Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

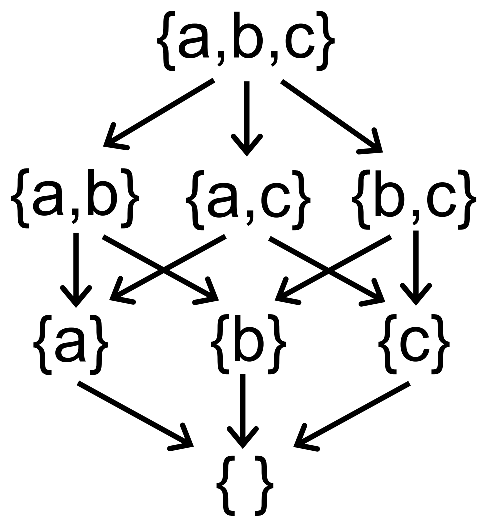
(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 4. Is $(S, \sqsubseteq)$ a poset where S is the power set of set {a,b,c} and $\sqsubseteq$ represents $\subseteq$ (subset)?

✓ (1) *Reflexivity*

(2) *Antisymmetry*

(3) *Transitivity*



Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ *(Reflexivity)*

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ *(Antisymmetry)*

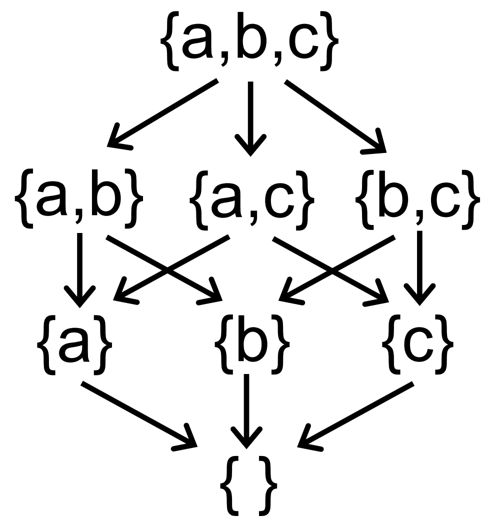
(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ *(Transitivity)*

Example 4. Is $(S, \sqsubseteq)$ a poset where S is the power set of set {a,b,c} and $\sqsubseteq$ represents $\subseteq$ (subset)?

✓ (1) *Reflexivity*

✓ (2) *Antisymmetry*

(3) *Transitivity*



Partial Order

We define **poset** as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

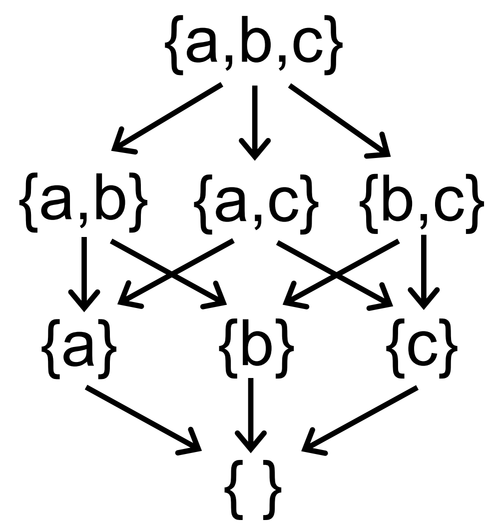
(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 4. Is $(S, \sqsubseteq)$ a poset where S is the power set of set {a,b,c} and $\sqsubseteq$ represents $\subseteq$ (subset)?

(1) *Reflexivity*

(2) *Antisymmetry*

(3) *Transitivity*



Partial Order

We define poset as a pair $(P, \sqsubseteq)$ where $\sqsubseteq$ is a binary relation that defines a partial ordering over P, and $\sqsubseteq$ has the following properties:

(1) $\forall x \in P, x \sqsubseteq x$ (*Reflexivity*)

(2) $\forall x, y \in P, x \sqsubseteq y \wedge y \sqsubseteq x \Longrightarrow x = y$ (*Antisymmetry*)

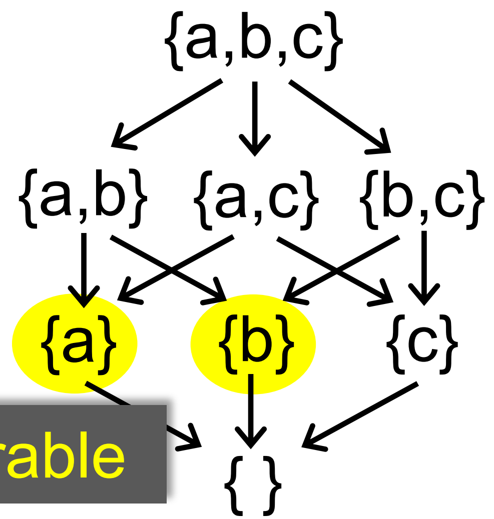
(3) $\forall x, y, z \in P, x \sqsubseteq y \wedge y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ (*Transitivity*)

Example 4. Is $(S, \sqsubseteq)$ a poset where S is the power set of set {a,b,c} and $\sqsubseteq$ represents $\subseteq$ (subset)?

✓ (1) *Reflexivity*

✓ (2) *Antisymmetry*

✓ (3) *Transitivity*



partial → incomparable

Upper and Lower Bounds

Given a poset $(P, \sqsubseteq)$ and its subset S that $S \subseteq P$, we say that

Upper and Lower Bounds

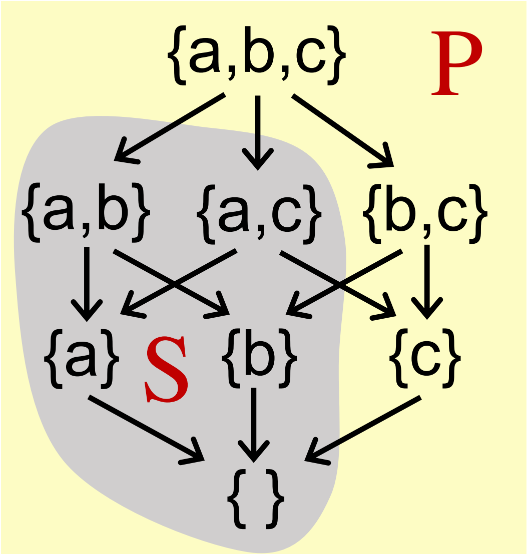
Given a poset $(P, \sqsubseteq)$ and its subset S that $S \subseteq P$, we say that $u \in P$ is an *upper bound* of S, if $\forall x \in S$, $x \sqsubseteq u$. Similarly,

Upper and Lower Bounds

Given a poset $(P, \sqsubseteq)$ and its subset S that $S \subseteq P$, we say that $u \in P$ is an *upper bound* of S, if $\forall x \in S$, $x \sqsubseteq u$. Similarly, $l \in P$ is an *lower bound* of S, if $\forall x \in S$, $l \sqsubseteq x$.

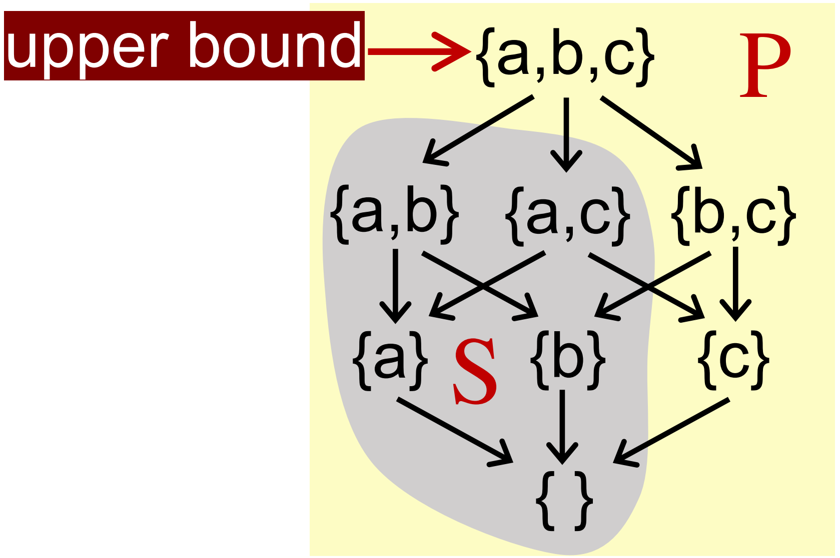
Upper and Lower Bounds

Given a poset $(P, \sqsubseteq)$ and its subset S that $S \subseteq P$, we say that $u \in P$ is an *upper bound* of S, if $\forall x \in S, x \sqsubseteq u$. Similarly, $l \in P$ is an *lower bound* of S, if $\forall x \in S, l \sqsubseteq x$.



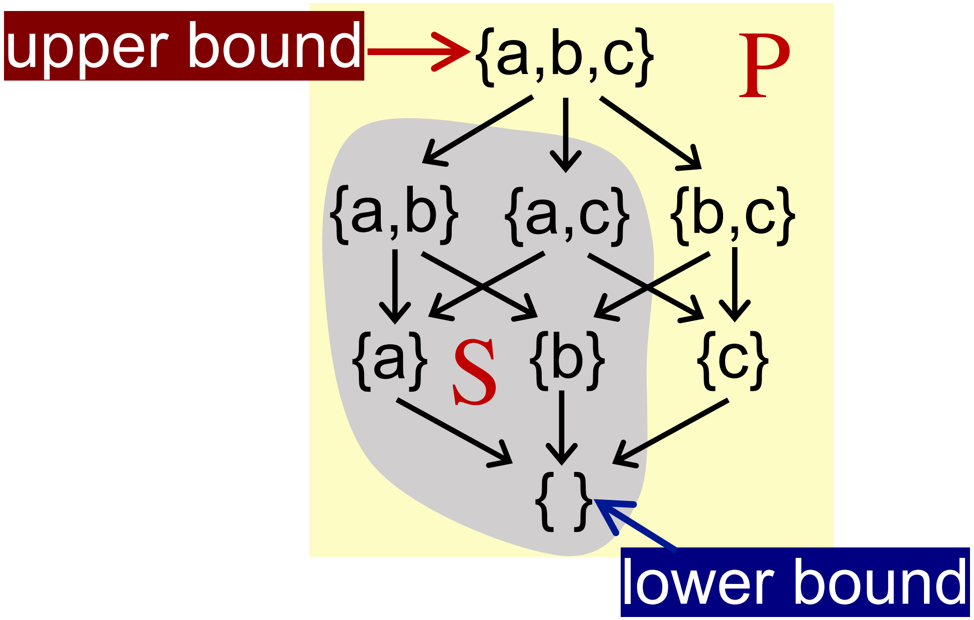
Upper and Lower Bounds

Given a poset $(P, \sqsubseteq)$ and its subset S that $S \subseteq P$, we say that $u \in P$ is an *upper bound* of S, if $\forall x \in S, x \sqsubseteq u$. Similarly, $l \in P$ is an *lower bound* of S, if $\forall x \in S, l \sqsubseteq x$.



Upper and Lower Bounds

Given a poset $(P, \sqsubseteq)$ and its subset S that $S \subseteq P$, we say that $u \in P$ is an *upper bound* of S, if $\forall x \in S, x \sqsubseteq u$. Similarly, $l \in P$ is an *lower bound* of S, if $\forall x \in S, l \sqsubseteq x$.



Upper and Lower Bounds

Given a poset $(P, \sqsubseteq)$ and its subset S that $S \subseteq P$, we say that $u \in P$ is an *upper bound* of S, if $\forall x \in S, x \sqsubseteq u$. Similarly, $l \in P$ is an *lower bound* of S, if $\forall x \in S, l \sqsubseteq x$.

We define the *least upper bound* (lub or join) of S, written $\sqcup S$, if for every upper bound of S, say u, $\sqcup S \sqsubseteq u$. Similarly,

Upper and Lower Bounds

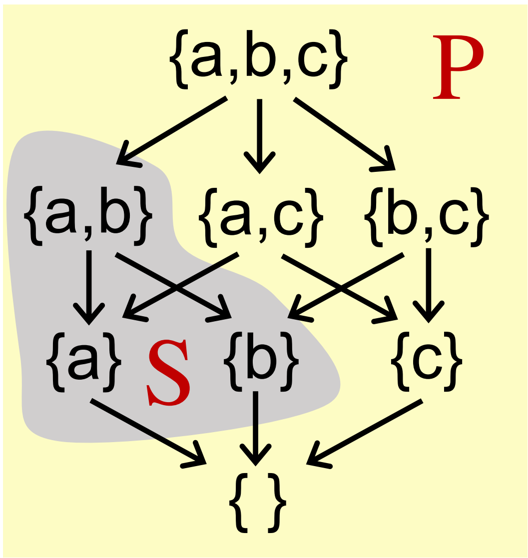
Given a poset $(P, \sqsubseteq)$ and its subset S that $S \subseteq P$, we say that $u \in P$ is an *upper bound* of S, if $\forall x \in S, x \sqsubseteq u$. Similarly, $l \in P$ is an *lower bound* of S, if $\forall x \in S, l \sqsubseteq x$.

We define the *least upper bound* (lub or join) of S, written $\sqcup S$, if for every upper bound of S, say u, $\sqcup S \sqsubseteq u$. Similarly,
We define the *greatest lower bound* (glb, or meet) of S, written $\sqcap S$, if for every lower bound of S, say l, $l \sqsubseteq \sqcap S$.

Upper and Lower Bounds

Given a poset $(P, \sqsubseteq)$ and its subset S that $S \subseteq P$, we say that $u \in P$ is an *upper bound* of S, if $\forall x \in S, x \sqsubseteq u$. Similarly, $l \in P$ is an *lower bound* of S, if $\forall x \in S, l \sqsubseteq x$.

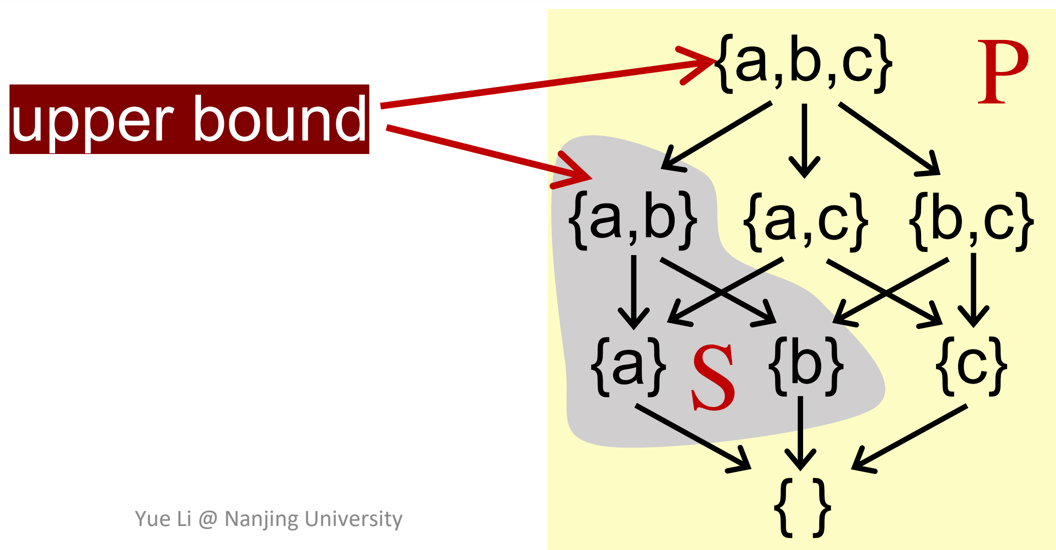
We define the *least upper bound* (lub or join) of S, written $\sqcup S$, if for every upper bound of S, say u, $\sqcup S \sqsubseteq u$. Similarly, We define the *greatest lower bound* (glb, or meet) of S, written $\sqcap S$, if for every lower bound of S, say l, $l \sqsubseteq \sqcap S$.



Upper and Lower Bounds

Given a poset (P, ⊑) and its subset S that S ⊆ P, we say that $u \in P$ is an *upper bound* of S, if $\forall x \in S, x \sqsubseteq u$. Similarly, $l \in P$ is an *lower bound* of S, if $\forall x \in S, l \sqsubseteq x$.

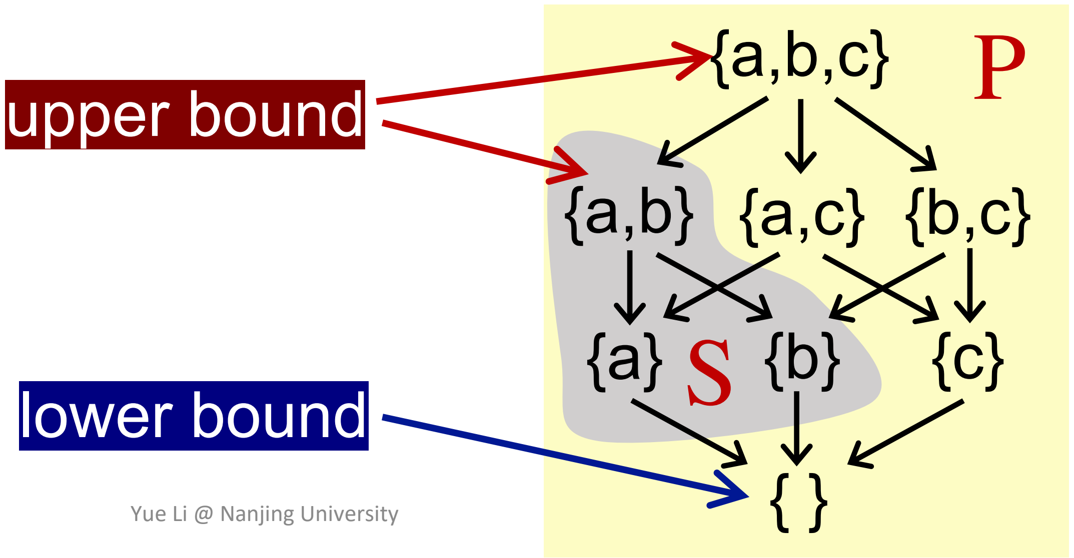
We define the *least upper bound* (lub or join) of S, written $\sqcup S$, if for every upper bound of S, say u, $\sqcup S \sqsubseteq u$. Similarly, We define the *greatest lower bound* (glb, or meet) of S, written $\sqcap S$, if for every lower bound of S, say l, $l \sqsubseteq \sqcap S$.



Upper and Lower Bounds

Given a poset $(P, \sqsubseteq)$ and its subset S that $S \subseteq P$, we say that $u \in P$ is an *upper bound* of S, if $\forall x \in S$, $x \sqsubseteq u$. Similarly, $l \in P$ is an *lower bound* of S, if $\forall x \in S$, $l \sqsubseteq x$.

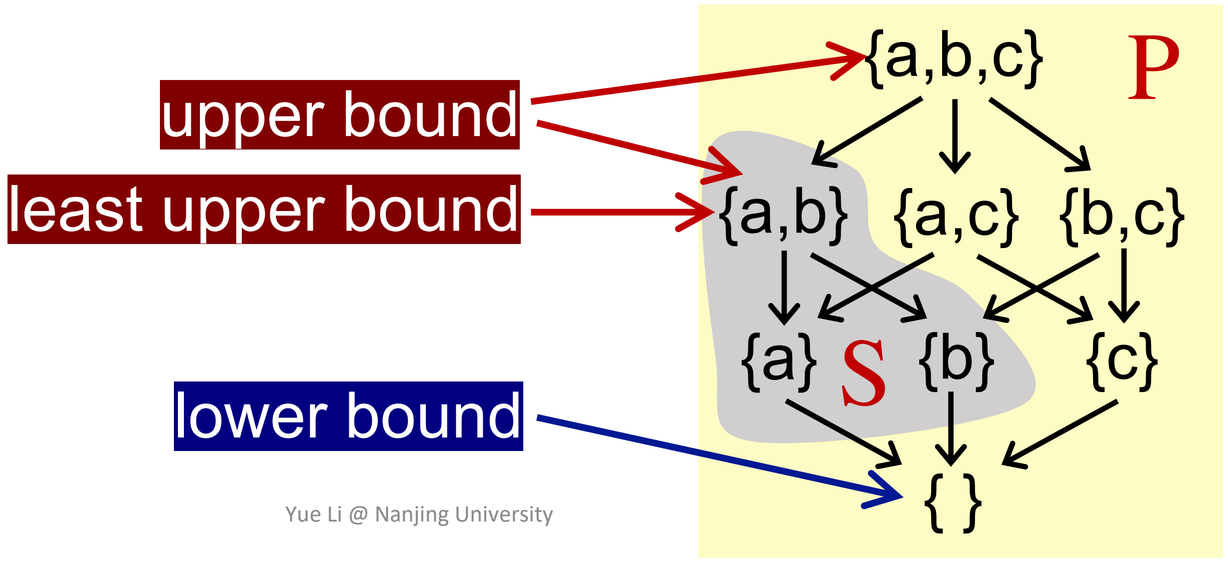
We define the *least upper bound* (lub or join) of S, written $\sqcup S$, if for every upper bound of S, say u, $\sqcup S \sqsubseteq u$. Similarly,
We define the *greatest lower bound* (glb, or meet) of S, written $\sqcap S$, if for every lower bound of S, say l, $l \sqsubseteq \sqcap S$.



Upper and Lower Bounds

Given a poset (P, ⊑) and its subset S that S ⊆ P, we say that u ∈ P is an *upper bound* of S, if ∀x ∈ S, x ⊑ u. Similarly, l ∈ P is an *lower bound* of S, if ∀x ∈ S, l ⊑ x.

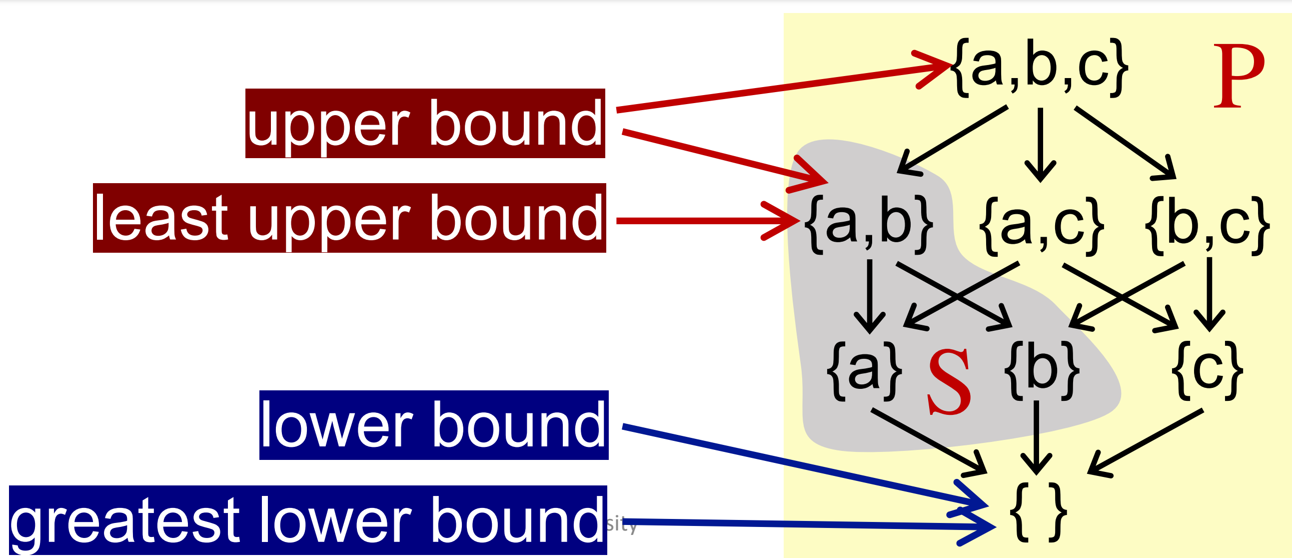
We define the *least upper bound* (lub or join) of S, written ⊔S, if for every upper bound of S, say u, ⊔S ⊑ u. Similarly, We define the *greatest lower bound* (glb, or meet) of S, written ⊓S, if for every lower bound of S, say l, l ⊑ ⊓S.



Upper and Lower Bounds

Given a poset $(P, \sqsubseteq)$ and its subset S that $S \subseteq P$, we say that $u \in P$ is an *upper bound* of S, if $\forall x \in S$, $x \sqsubseteq u$. Similarly, $l \in P$ is an *lower bound* of S, if $\forall x \in S$, $l \sqsubseteq x$.

We define the *least upper bound* (lub or join) of S, written $\sqcup S$, if for every upper bound of S, say u, $\sqcup S \sqsubseteq u$. Similarly, We define the *greatest lower bound* (glb, or meet) of S, written $\sqcap S$, if for every lower bound of S, say l, $l \sqsubseteq \sqcap S$.



Upper and Lower Bounds

Given a poset $(P, \sqsubseteq)$ and its subset S that $S \subseteq P$, we say that $u \in P$ is an *upper bound* of S, if $\forall x \in S, x \sqsubseteq u$. Similarly, $l \in P$ is an *lower bound* of S, if $\forall x \in S, l \sqsubseteq x$.

We define the *least upper bound* (lub or join) of S, written $\sqcup S$, if for every upper bound of S, say u, $\sqcup S \sqsubseteq u$. Similarly,
We define the *greatest lower bound* (glb, or meet) of S, written $\sqcap S$, if for every lower bound of S, say l, $l \sqsubseteq \sqcap S$.

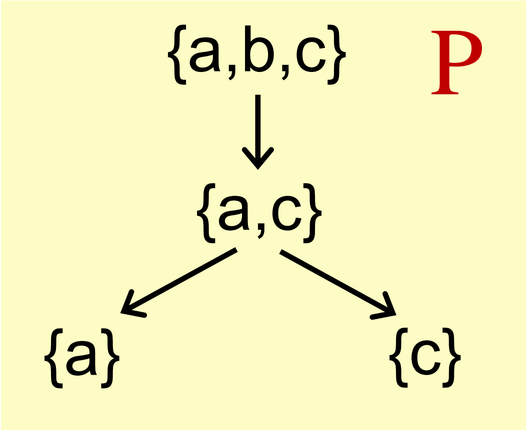
Usually, if S contains only two elements a and b ($S = \{a, b\}$), then
$\sqcup S$ can be written $a \sqcup b$ (the join of a and b)
$\sqcap S$ can be written $a \sqcap b$ (the meet of a and b)

Some Properties

- Not every poset has *lub* or *glb*

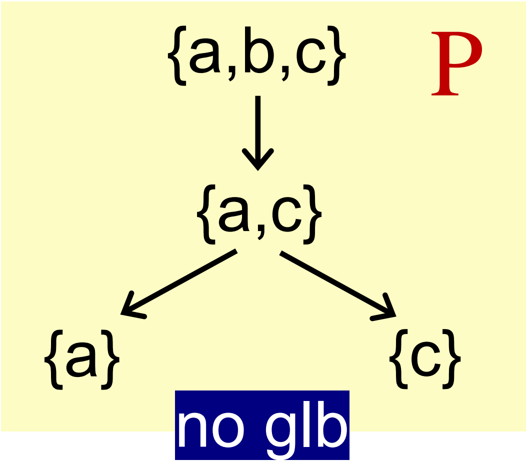
Some Properties

- Not every poset has *lub* or *glb*



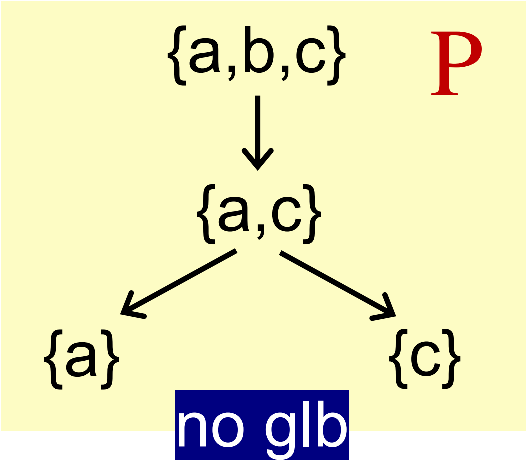
Some Properties

- Not every poset has *lub* or *glb*



Some Properties

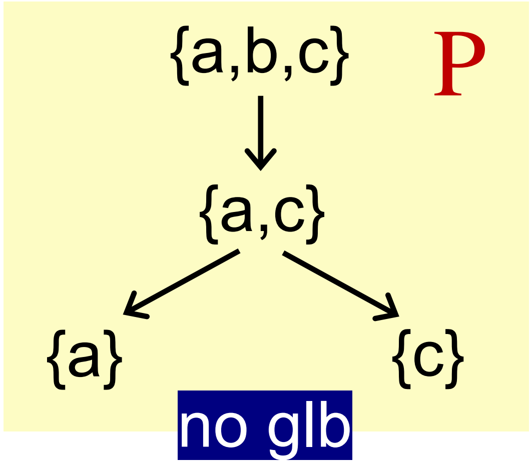
- Not every poset has *lub* or *glb*



- But if a poset has *lub* or *glb*, it will be unique

Some Properties

- Not every poset has *lub* or *glb*

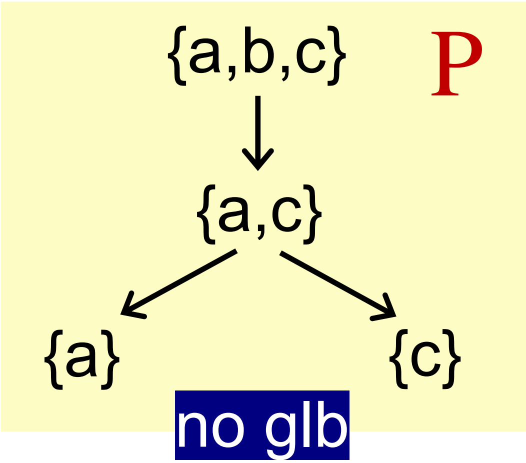


- But if a poset has *lub* or *glb*, it will be unique

Proof.

Some Properties

- Not every poset has *lub* or *glb*

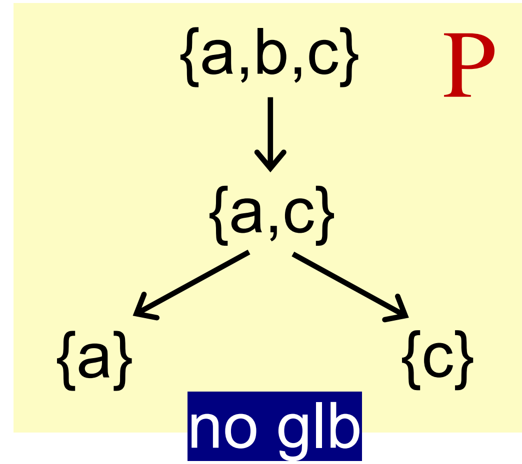


- But if a poset has *lub* or *glb*, it will be unique

> *Proof*.
>
> assume g_1 and g_2 are both glbs of poset P

Some Properties

- Not every poset has *lub* or *glb*



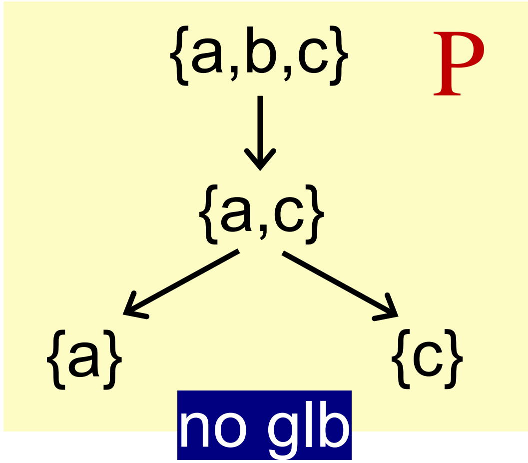
- But if a poset has *lub* or *glb*, it will be unique

Proof.

assume g_1 and g_2 are both glbs of poset P

according to the definition of glb

Some Properties



- Not every poset has *lub* or *glb*

- But if a poset has *lub* or *glb*, it will be unique

Proof.

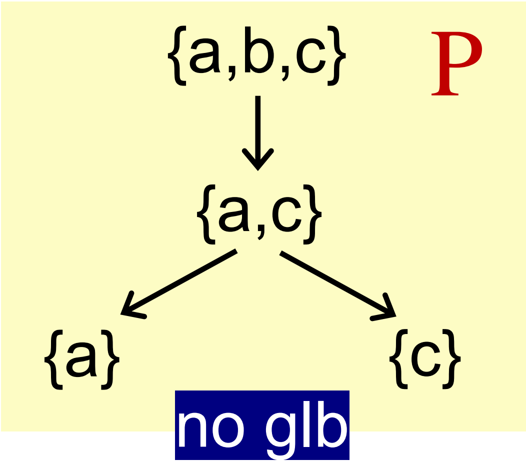
assume g_1 and g_2 are both glbs of poset P

according to the definition of glb

$g_1 \sqsubseteq (g_2 = \sqcap P)$ and $g_2 \sqsubseteq (g_1 = \sqcap P)$

Some Properties

- Not every poset has *lub* or *glb*



- But if a poset has *lub* or *glb*, it will be unique

Proof.

assume g_1 and g_2 are both glbs of poset P

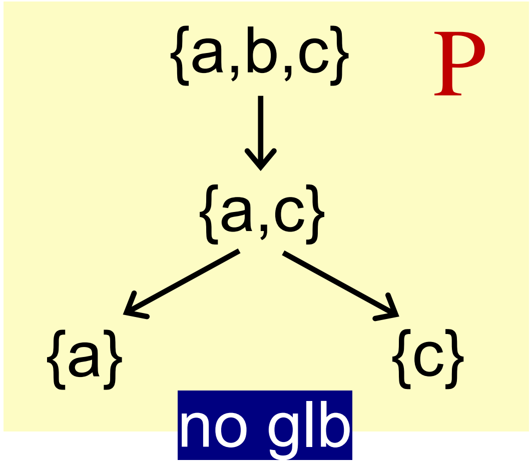
according to the definition of glb

$g_1 \sqsubseteq (g_2 = \sqcap P)$ and $g_2 \sqsubseteq (g_1 = \sqcap P)$

by the *antisymmetry* of partial order $\sqsubseteq$

Some Properties

- Not every poset has *lub* or *glb*



- But if a poset has *lub* or *glb*, it will be unique

Proof.

assume g_1 and g_2 are both glbs of poset P

according to the definition of glb

$g_1 \sqsubseteq (g_2 = \sqcap P)$ and $g_2 \sqsubseteq (g_1 = \sqcap P)$

by the *antisymmetry* of partial order $\sqsubseteq$

$g_1 = g_2$

Lattice

Given a poset $(P, \sqsubseteq)$, $\forall a, b \in P$, if $a \sqcup b$ and $a \sqcap b$ exist, then $(P, \sqsubseteq)$ is called a lattice

Lattice

Given a poset $(P, \sqsubseteq)$, $\forall a, b \in P$, if $a \sqcup b$ and $a \sqcap b$ exist, then $(P, \sqsubseteq)$ is called a lattice

A poset is a lattice if every pair of its elements has a least upper bound and a greatest lower bound

Lattice

Given a poset $(P, \sqsubseteq)$, $\forall a, b \in P$, if $a \sqcup b$ and $a \sqcap b$ exist, then $(P, \sqsubseteq)$ is called a lattice

A poset is a lattice if every pair of its elements has a least upper bound and a greatest lower bound

Example 1. Is $(S, \sqsubseteq)$ a lattice where S is a set of integers and $\sqsubseteq$ represents $\leq$ (less than or equal to)?

Lattice

Given a poset $(P, \sqsubseteq)$, $\forall a, b \in P$, if $a \sqcup b$ and $a \sqcap b$ exist, then $(P, \sqsubseteq)$ is called a lattice

A poset is a lattice if every pair of its elements has a least upper bound and a greatest lower bound

Example 1. Is $(S, \sqsubseteq)$ a lattice where S is a set of integers and $\sqsubseteq$ represents $\leq$ (less than or equal to)?

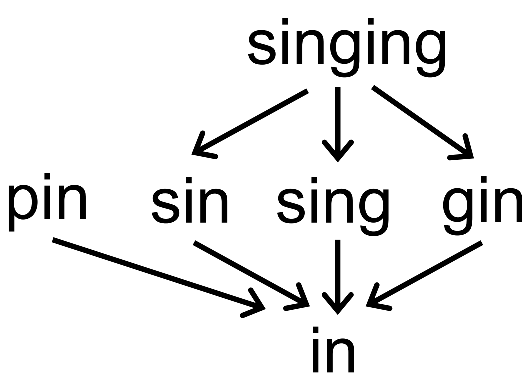
✓ The $\sqcup$ operator means "max" and $\sqcap$ operator means "min"

Lattice

Given a poset $(P, \sqsubseteq)$, $\forall a, b \in P$, if $a \sqcup b$ and $a \sqcap b$ exist, then $(P, \sqsubseteq)$ is called a lattice

A poset is a lattice if every pair of its elements has a least upper bound and a greatest lower bound

Example 2. Is $(S, \sqsubseteq)$ a lattice where S is a set of English words and $\sqsubseteq$ represents the *substring* relation, i.e., $s1 \sqsubseteq s2$ means s1 is a substring of s2?



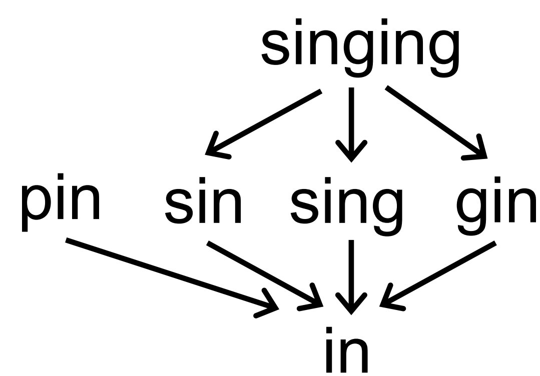
Lattice

Given a poset $(P, \sqsubseteq)$, $\forall a, b \in P$, if $a \sqcup b$ and $a \sqcap b$ exist, then $(P, \sqsubseteq)$ is called a lattice

A poset is a lattice if every pair of its elements has a least upper bound and a greatest lower bound

Example 2. Is $(S, \sqsubseteq)$ a lattice where S is a set of English words and $\sqsubseteq$ represents the *substring* relation, i.e., $s1 \sqsubseteq s2$ means s1 is a substring of s2?

pin $\sqcup$ sin = ?

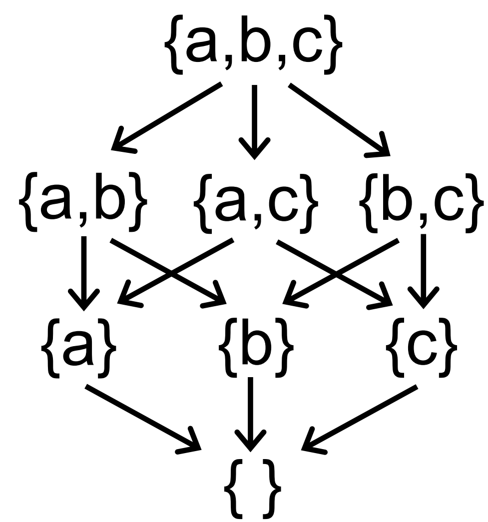


Lattice

Given a poset $(P, \sqsubseteq)$, $\forall a, b \in P$, if $a \sqcup b$ and $a \sqcap b$ exist, then $(P, \sqsubseteq)$ is called a lattice

A poset is a lattice if every pair of its elements has a least upper bound and a greatest lower bound

Example 3. Is $(S, \sqsubseteq)$ a lattice where S is the power set of set {a,b,c} and $\sqsubseteq$ represents $\subseteq$ (subset)?



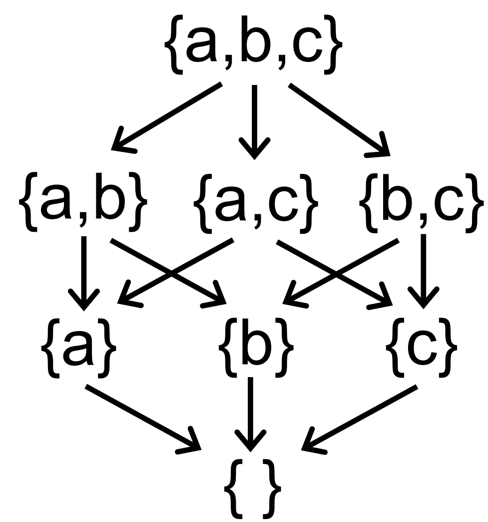
Lattice

Given a poset (P, ⊑), ∀a, b ∈ P, if a ⊔ b and a ⊓ b exist, then (P, ⊑) is called a lattice

A poset is a lattice if every pair of its elements has a least upper bound and a greatest lower bound

Example 3. Is (S, ⊑) a lattice where S is the power set of set {a,b,c} and ⊑ represents ⊆ (subset)?

✓ The ⊔ operator means ∪ and ⊓ operator means ∩



Lattice

Given a poset $(P, \sqsubseteq)$, $\forall a, b \in P$, if $a \sqcup b$ and $a \sqcap b$ exist, then $(P, \sqsubseteq)$ is called a lattice

A poset is a lattice if every pair of its elements has a least upper bound and a greatest lower bound

Semilattice

Given a poset $(P, \sqsubseteq)$, $\forall a, b \in P$,

if only $a \sqcup b$ exists, then $(P, \sqsubseteq)$ is called a join semilattice

if only $a \sqcap b$ exists, then $(P, \sqsubseteq)$ is called a meet semilattice

Complete Lattice

Given a lattice (P, $\sqsubseteq$), for arbitrary subset S of P, if $\sqcup S$ and $\sqcap S$ exist, then (P, $\sqsubseteq$) is called a complete lattice

Complete Lattice

Given a lattice (P, ⊑), for arbitrary subset S of P, if ⊔S and ⊓S exist, then (P, ⊑) is called a complete lattice

All subsets of a lattice have a least upper bound and a greatest lower bound

Complete Lattice

Given a lattice (P, ⊑), for arbitrary subset S of P, if ⊔S and ⊓S exist, then (P, ⊑) is called a complete lattice

All subsets of a lattice have a least upper bound and a greatest lower bound

Example 1. Is (S, ⊑) a complete lattice where S is a set of integers and ⊑ represents ≤ (less than or equal to)?

Complete Lattice

Given a lattice $(P, \sqsubseteq)$, for arbitrary subset S of P, if $\sqcup S$ and $\sqcap S$ exist, then $(P, \sqsubseteq)$ is called a complete lattice

All subsets of a lattice have a least upper bound and a greatest lower bound

Example 1. Is $(S, \sqsubseteq)$ a complete lattice where S is a set of integers and $\sqsubseteq$ represents $\leq$ (less than or equal to)?

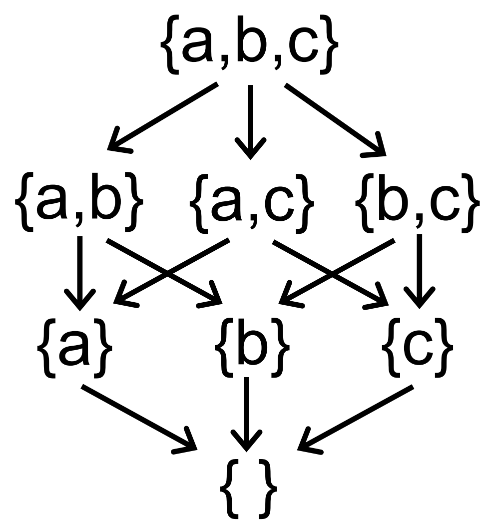
✗ For a subset S^+ including all positive integers, it has no $\sqcup S^+$ $(+\infty)$

Complete Lattice

Given a lattice (P, ⊑), for arbitrary subset S of P, if ⊔S and ⊓S exist, then (P, ⊑) is called a complete lattice

All subsets of a lattice have a least upper bound and a greatest lower bound

Example 2. Is (S, ⊑) a complete lattice where S is the power set of set {a,b,c} and ⊑ represents ⊆ (subset)?



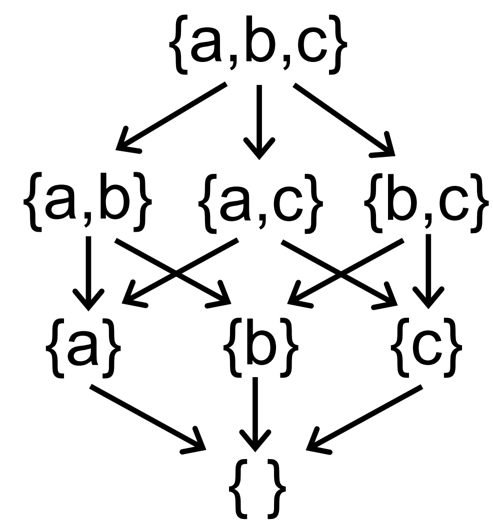
Complete Lattice

Given a lattice (P, ⊑), for arbitrary subset S of P, if ⊔S and ⊓S exist, then (P, ⊑) is called a complete lattice

All subsets of a lattice have a least upper bound and a greatest lower bound

Example 2. Is (S, ⊑) a complete lattice where S is the power set of set {a,b,c} and ⊑ represents ⊆ (subset)?

✓ Note: the definition of bounds implies that the bounds are not necessarily in the subsets (but they must be in the lattice)

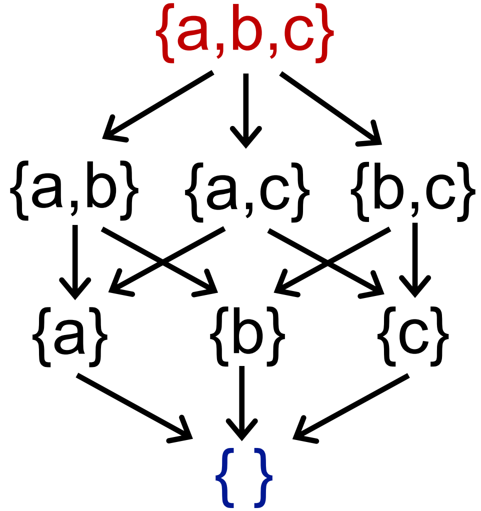


Complete Lattice

Given a lattice (P, ⊑), for arbitrary subset S of P, if ⊔S and ⊓S exist, then (P, ⊑) is called a complete lattice

All subsets of a lattice have a least upper bound and a greatest lower bound

Every complete lattice (P, ⊑) has
a greatest element ⊤ = ⊔P called top and
a least element ⊥ = ⊓P called bottom



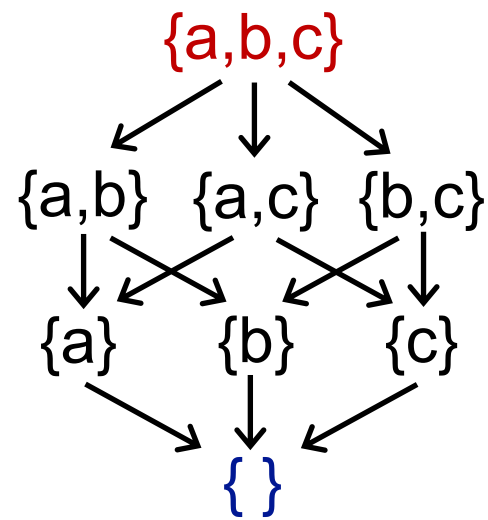
Complete Lattice

Given a lattice $(P, \sqsubseteq)$, for arbitrary subset S of P, if $\sqcup S$ and $\sqcap S$ exist, then $(P, \sqsubseteq)$ is called a complete lattice

All subsets of a lattice have a least upper bound and a greatest lower bound

Every complete lattice $(P, \sqsubseteq)$ has
a greatest element $\top = \sqcup P$ called top and
a least element $\bot = \sqcap P$ called bottom

Every finite lattice (P is finite) is a complete lattice



Complete Lattice

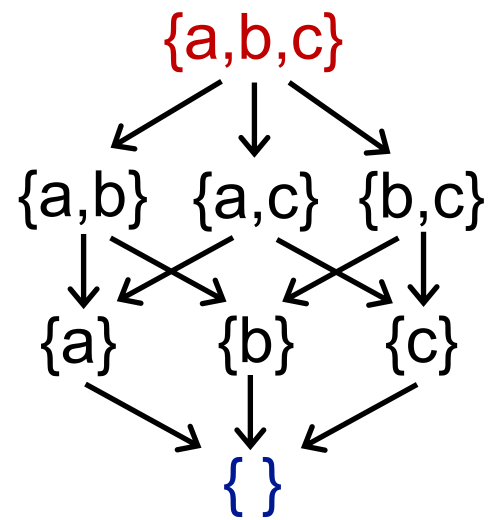
Given a lattice (P, ⊑), for arbitrary subset S of P, if ⊔S and ⊓S exist, then (P, ⊑) is called a complete lattice

All subsets of a lattice have a least upper bound and a greatest lower bound

Every complete lattice (P, ⊑) has
a greatest element ⊤ = ⊔P called top and
a least element ⊥ = ⊓P called bottom

Every finite lattice (P is finite) is a complete lattice

What about the opposite one?



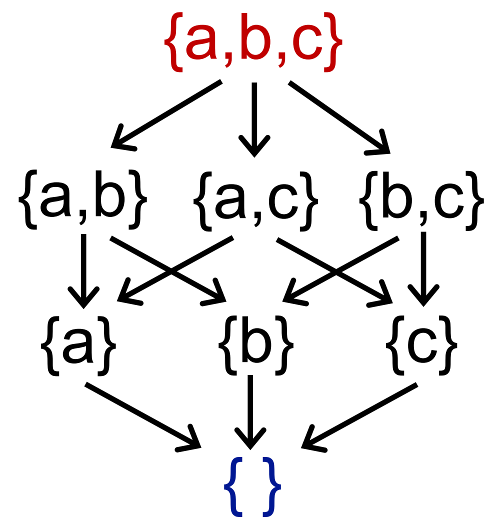
Complete Lattice **Mostly focused in data flow analysis**

Given a lattice $(P, \sqsubseteq)$, for arbitrary subset S of P, if $\sqcup S$ and $\sqcap S$ exist, then $(P, \sqsubseteq)$ is called a complete lattice

All subsets of a lattice have a least upper bound and a greatest lower bound

Every complete lattice $(P, \sqsubseteq)$ has
a **greatest** element $\top = \sqcup P$ called **top** and
a **least** element $\bot = \sqcap P$ called **bottom**

Every **finite** lattice (P is finite) is a complete lattice



Product Lattice

Given lattices $L_1 = (P_1, \sqsubseteq_1)$, $L_2 = (P_2, \sqsubseteq_2)$, …, $L_n = (P_n, \sqsubseteq_n)$, if for all i, $(P_i, \sqsubseteq_i)$ has $\sqcup_i$ (least upper bound) and $\sqcap_i$ (greatest lower bound), then we can have a product lattice $L^n = (P, \sqsubseteq)$ that is defined by:

Product Lattice

Given lattices $L_1 = (P_1, \sqsubseteq_1)$, $L_2 = (P_2, \sqsubseteq_2)$, …, $L_n = (P_n, \sqsubseteq_n)$, if for all i, $(P_i, \sqsubseteq_i)$ has $\sqcup_i$ (least upper bound) and $\sqcap_i$ (greatest lower bound), then we can have a product lattice $L^n = (P, \sqsubseteq)$ that is defined by:

- $P = P_1 \times \ldots \times P_n$

Product Lattice

Given lattices $L_1 = (P_1, \sqsubseteq_1)$, $L_2 = (P_2, \sqsubseteq_2)$, …, $L_n = (P_n, \sqsubseteq_n)$, if for all i, $(P_i, \sqsubseteq_i)$ has $\sqcup_i$ (least upper bound) and $\sqcap_i$ (greatest lower bound), then we can have a product lattice $L^n = (P, \sqsubseteq)$ that is defined by:

- $P = P_1 \times \ldots \times P_n$
- $(x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots, y_n) \Leftrightarrow (x_1 \sqsubseteq y_1) \wedge \ldots \wedge (x_n \sqsubseteq y_n)$

Product Lattice

Given lattices $L_1 = (P_1, \sqsubseteq_1)$, $L_2 = (P_2, \sqsubseteq_2)$, …, $L_n = (P_n, \sqsubseteq_n)$, if for all i, $(P_i, \sqsubseteq_i)$ has $\sqcup_i$ (least upper bound) and $\sqcap_i$ (greatest lower bound), then we can have a product lattice $L^n = (P, \sqsubseteq)$ that is defined by:

- $P = P_1 \times \ldots \times P_n$
- $(x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots, y_n) \Leftrightarrow (x_1 \sqsubseteq y_1) \land \ldots \land (x_n \sqsubseteq y_n)$
- $(x_1, \ldots, x_n) \sqcup (y_1, \ldots, y_n) = (x_1 \sqcup_1 y_1, \ldots, x_n \sqcup_n y_n)$

Product Lattice

Given lattices $L_1 = (P_1, \sqsubseteq_1), L_2 = (P_2, \sqsubseteq_2), \ldots, L_n = (P_n, \sqsubseteq_n)$, if for all i, $(P_i, \sqsubseteq_i)$ has $\sqcup_i$ (least upper bound) and $\sqcap_i$ (greatest lower bound), then we can have a product lattice $L^n = (P, \sqsubseteq)$ that is defined by:

- $P = P_1 \times \ldots \times P_n$
- $(x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots, y_n) \Leftrightarrow (x_1 \sqsubseteq y_1) \land \ldots \land (x_n \sqsubseteq y_n)$
- $(x_1, \ldots, x_n) \sqcup (y_1, \ldots, y_n) = (x_1 \sqcup_1 y_1, \ldots, x_n \sqcup_n y_n)$
- $(x_1, \ldots, x_n) \sqcap (y_1, \ldots, y_n) = (x_1 \sqcap_1 y_1, \ldots, x_n \sqcap_n y_n)$

Product Lattice

Given lattices $L_1 = (P_1, \sqsubseteq_1)$, $L_2 = (P_2, \sqsubseteq_2)$, …, $L_n = (P_n, \sqsubseteq_n)$, if for all i, $(P_i, \sqsubseteq_i)$ has $\sqcup_i$ (least upper bound) and $\sqcap_i$ (greatest lower bound), then we can have a product lattice $L^n = (P, \sqsubseteq)$ that is defined by:

- $P = P_1 \times \ldots \times P_n$
- $(x_1, \ldots, x_n) \sqsubseteq (y_1, \ldots, y_n) \Leftrightarrow (x_1 \sqsubseteq y_1) \wedge \ldots \wedge (x_n \sqsubseteq y_n)$
- $(x_1, \ldots, x_n) \sqcup (y_1, \ldots, y_n) = (x_1 \sqcup_1 y_1, \ldots, x_n \sqcup_n y_n)$
- $(x_1, \ldots, x_n) \sqcap (y_1, \ldots, y_n) = (x_1 \sqcap_1 y_1, \ldots, x_n \sqcap_n y_n)$

- A product lattice is a lattice
- If a product lattice L is a product of complete lattices, then L is also complete

Data Flow Analysis Framework via Lattice

A data flow analysis framework (D, L, F) consists of:

Data Flow Analysis Framework via Lattice

A data flow analysis framework (D, L, F) consists of:

- **D**: a direction of data flow: forwards or backwards

Data Flow Analysis Framework via Lattice

A data flow analysis framework (D, L, F) consists of:

- **D**: a direction of data flow: forwards or backwards
- **L**: a lattice including domain of the values V and a meet $\sqcap$ or join $\sqcup$ operator

Data Flow Analysis Framework via Lattice

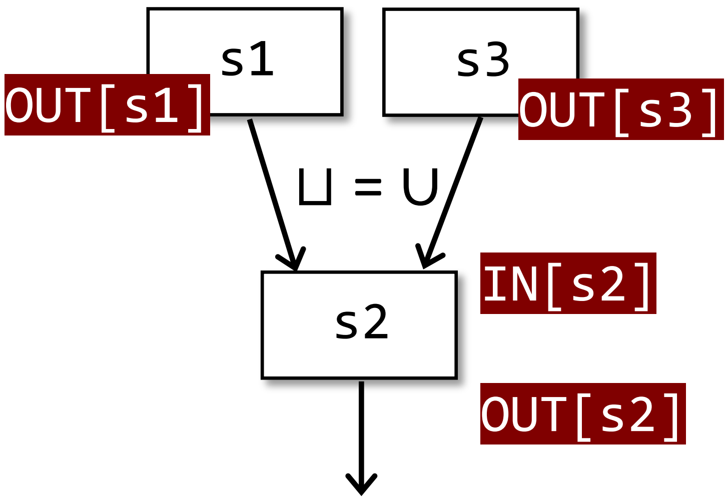
A data flow analysis framework (D, L, F) consists of:

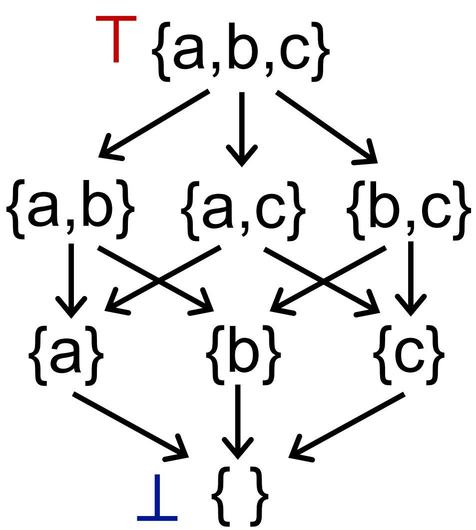
- **D**: a direction of data flow: forwards or backwards
- **L**: a lattice including domain of the values V and a meet $\sqcap$ or join $\sqcup$ operator
- **F**: a family of transfer functions from V to V

Data Flow Analysis Framework via Lattice

A data flow analysis framework (D, L, F) consists of:

- **D**: a direction of data flow: forwards or backwards
- **L**: a lattice including domain of the values V and a meet ⊓ or join ⊔ operator
- **F**: a family of transfer functions from V to V

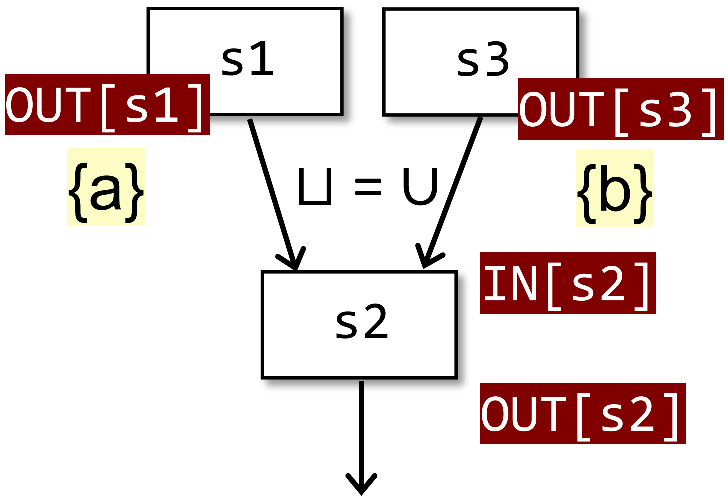


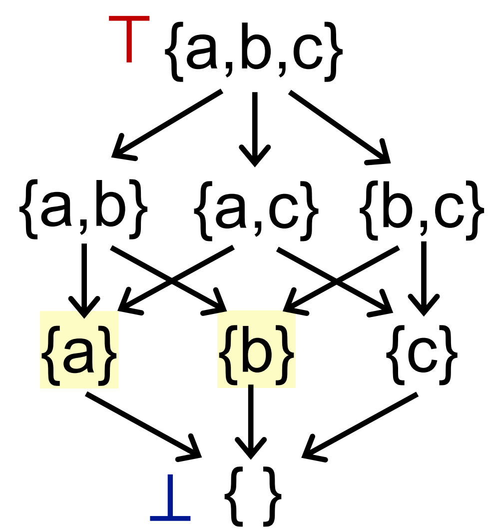


Data Flow Analysis Framework via Lattice

A data flow analysis framework (D, L, F) consists of:

- **D**: a direction of data flow: forwards or backwards
- **L**: a lattice including domain of the values V and a meet ⊓ or join ⊔ operator
- **F**: a family of transfer functions from V to V

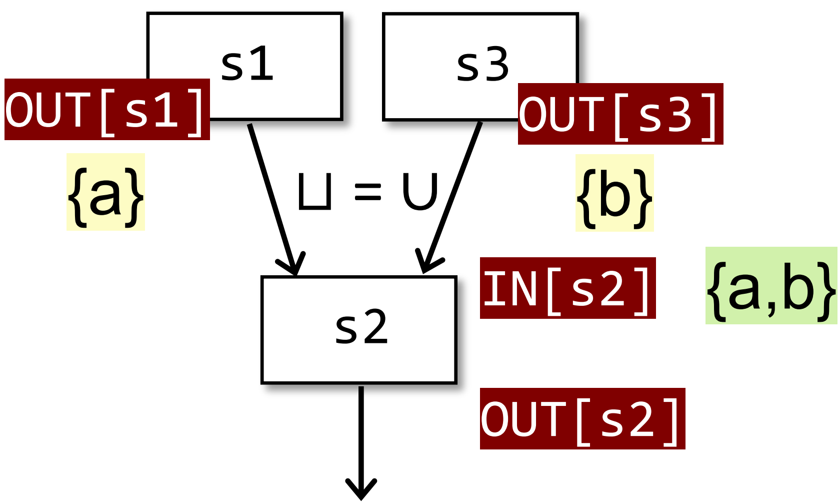


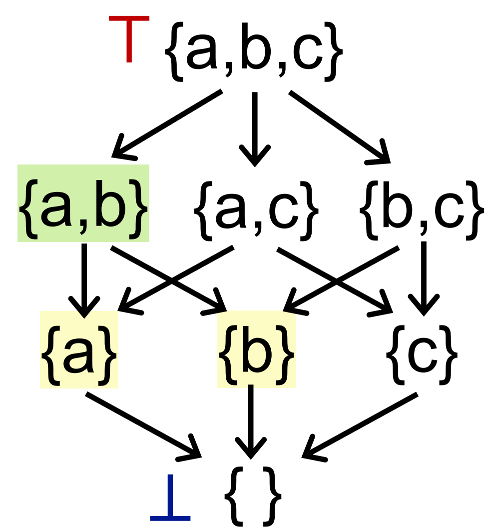


Data Flow Analysis Framework via Lattice

A data flow analysis framework (D, L, F) consists of:

- **D**: a direction of data flow: forwards or backwards
- **L**: a lattice including domain of the values V and a meet ⊓ or join ⊔ operator
- **F**: a family of transfer functions from V to V

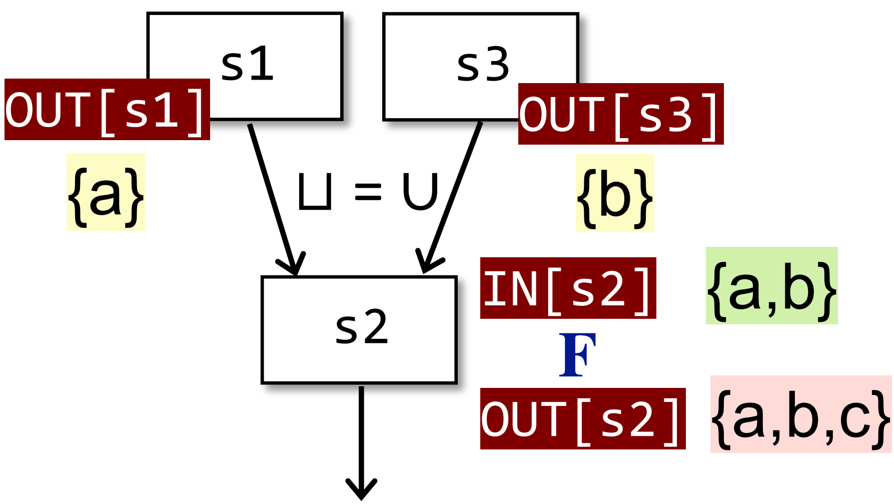


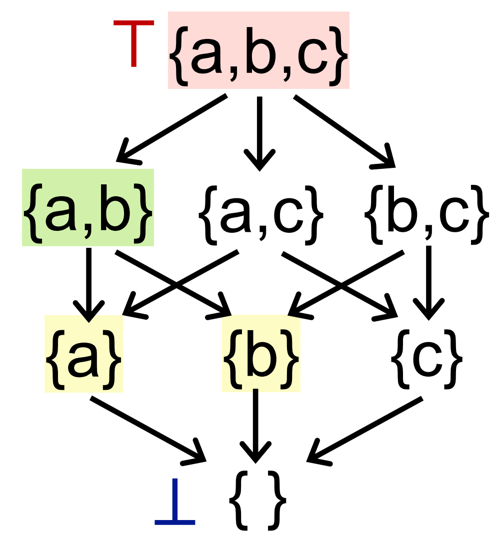


Data Flow Analysis Framework via Lattice

A data flow analysis framework (D, L, F) consists of:

- **D**: a direction of data flow: forwards or backwards
- **L**: a lattice including domain of the values V and a meet ⊓ or join ⊔ operator
- **F**: a family of transfer functions from V to V

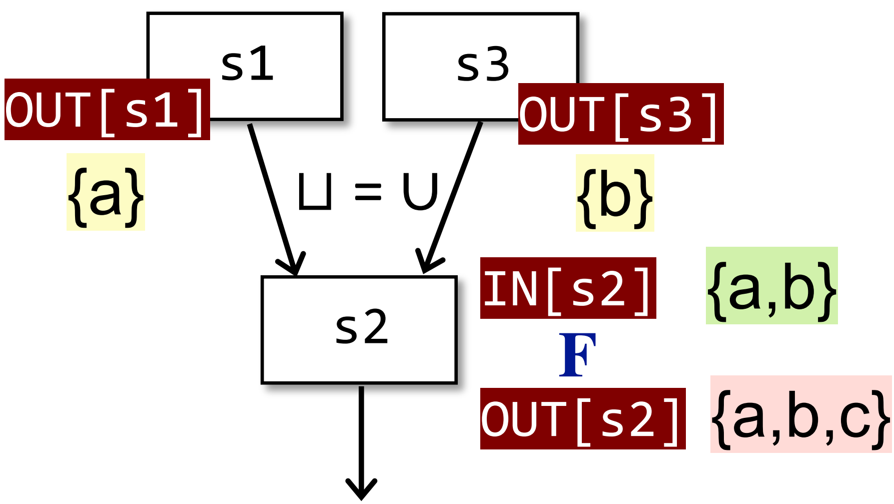


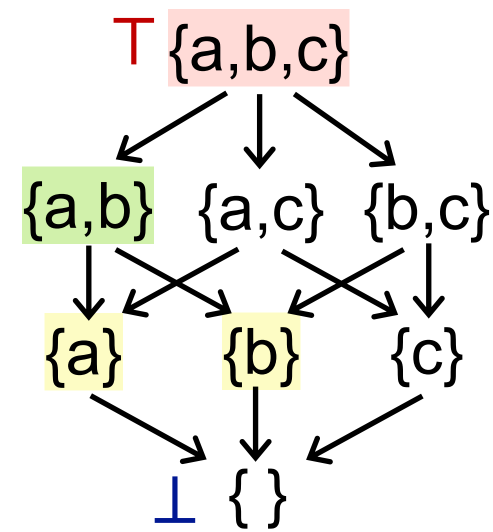


Data Flow Analysis Framework via Lattice

A data flow analysis framework (D, L, F) consists of:

- **D**: a direction of data flow: forwards or backwards
- **L**: a lattice including domain of the values V and a meet ⊓ or join ⊔ operator
- **F**: a family of transfer functions from V to V





Data flow analysis can be seen as iteratively applying transfer functions and meet/join operations on the values of a lattice

Review The Questions We Have Seen Before

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

- Is the algorithm guaranteed to terminate or reach the fixed point, or does it always have a solution?
- If so, is there only one solution or only one fixed point? If more than one, is our solution the best one (most precise)?
- When will the algorithm reach the fixed point, or when can we get the solution?

Review The Questions We Have Seen Before

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

- ? Is the algorithm guaranteed to terminate or reach the fixed point, or does it always have a solution?

- If so, is there only one solution or only one fixed point? If more than one, is our solution the best one (most precise)?

- When will the algorithm reach the fixed point, or when can we get the solution?

Review The Questions We Have Seen Before

The iterative algorithm (or the IN/OUT
produces a solution to a da

Recall "OUT never shrinks"
It is about monotonicity

- Is the algorithm guarante minate or reach the fixed point, or does it always have a solution?

- If so, is there only one solution or only one fixed point? If more than one, is our solution the best one (most precise)?

- When will the algorithm reach the fixed point, or when can we get the solution?

Review The Questions We Have Seen Before

The iterative algorithm (or the IN/OUT
produces a solution to a da

Recall "OUT never shrinks"
It is about monotonicity

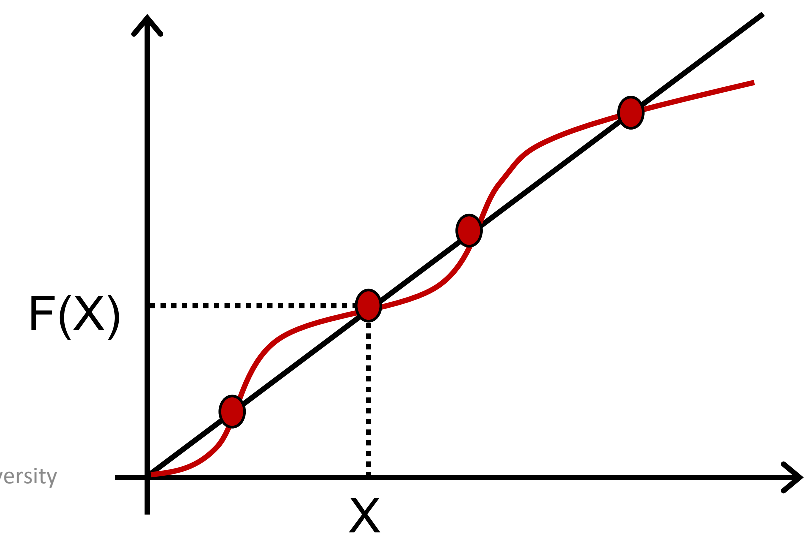
- Is the algorithm guarante ... ate or reach the fixed point, or does it always have a solution?
- If so, is there only one solution or only one fixed point? If more than one, is our solution the best one (most precise)?
- When will the algorithm reach the fixed point, or when can we get the solution?

Review The Questions We Have Seen Before

The iterative algorithm (or the IN/OUT
produces a solution to a da

Recall "OUT never shrinks"
It is about monotonicity

- Is the algorithm guarante … minate or reach the fixed point, or does it always have a solution?
- If so, is there only one solution or only one fixed point? If more than one, is our solution the best one (most precise)?
- When will the algorithm reach the fixed point, or when can we get the solution?



Monotonicity

A function f: L → L (L is a lattice) is monotonic if $\forall x, y \in L$,

$$x \sqsubseteq y \Longrightarrow f(x) \sqsubseteq f(y)$$

Monotonicity

A function f: L → L (L is a lattice) is monotonic if $\forall x, y \in L$,

$$x \sqsubseteq y \Longrightarrow f(x) \sqsubseteq f(y)$$

Fixed-Point Theorem

Given a complete lattice $(L, \sqsubseteq)$, if

Monotonicity

A function f: $L \rightarrow L$ (L is a lattice) is monotonic if $\forall x, y \in L$,

$$x \sqsubseteq y \Longrightarrow f(x) \sqsubseteq f(y)$$

Fixed-Point Theorem

Given a complete lattice $(L, \sqsubseteq)$, if

(1) f: $L \rightarrow L$ is monotonic and (2) L is finite, then

Monotonicity

A function f: $L \rightarrow L$ (L is a lattice) is monotonic if $\forall x, y \in L$,

$$x \sqsubseteq y \Longrightarrow f(x) \sqsubseteq f(y)$$

Fixed-Point Theorem

Given a complete lattice $(L, \sqsubseteq)$, if

(1) $f: L \rightarrow L$ is monotonic and (2) L is finite, then

the least fixed point of f can be found by iterating

$f(\perp), f(f(\perp)), \ldots, f^k(\perp)$ until a fixed point is reached

Monotonicity

A function $f: L \to L$ (L is a lattice) is monotonic if $\forall x, y \in L$,

$$x \sqsubseteq y \Longrightarrow f(x) \sqsubseteq f(y)$$

Fixed-Point Theorem

Given a complete lattice $(L, \sqsubseteq)$, if

(1) $f: L \to L$ is monotonic and (2) L is finite, then

the least fixed point of f can be found by iterating

$f(\bot), f(f(\bot)), \ldots, f^k(\bot)$ until a fixed point is reached

the greatest fixed point of f can be found by iterating

$f(\top), f(f(\top)), \ldots, f^k(\top)$ until a fixed point is reached

Monotonicity

> A function $f: L \rightarrow L$ (L is a lattice) is monotonic if $\forall x, y \in L$,
> $$x \sqsubseteq y \Longrightarrow f(x) \sqsubseteq f(y)$$

Fixed-Point Theorem

> Given a complete lattice $(L, \sqsubseteq)$, if
>
> (1) $f: L \rightarrow L$ is monotonic and (2) L is finite, then
>
> the least fixed point of f can be found by iterating
>
> $f(\bot), f(f(\bot)), \ldots, f^k(\bot)$ until a fixed point is reached
>
> the greatest fixed point of f can be found by iterating
>
> $f(\top$ …, $f^k(\top)$ until a fixed point is reached

Let us prove

(1) Existence of fixed point

(2) The fixed point is the least

Fixed-Point Theorem (Existence of Fixed Point)

Proof:

By the definition of $\perp$ and $f: L \rightarrow L$, we have

$$\perp \sqsubseteq f(\perp)$$

Fixed-Point Theorem (Existence of Fixed Point)

Proof:

By the definition of $\perp$ and $f: L \to L$, we have

$$\perp \sqsubseteq f(\perp)$$

As f is monotonic, we have

$$f(\perp) \sqsubseteq f(f(\perp)) = f^2(\perp)$$

Fixed-Point Theorem (Existence of Fixed Point)

Proof:

By the definition of $\perp$ and f: $L \rightarrow L$, we have

$$\perp \sqsubseteq f(\perp)$$

As f is monotonic, we have

$$f(\perp) \sqsubseteq f(f(\perp)) = f^2(\perp)$$

Similarly (by repeatedly applying f), we have

$$\perp \sqsubseteq f(\perp) \sqsubseteq f^2(\perp) \sqsubseteq \ \ldots \sqsubseteq f^i(\perp)$$

Fixed-Point Theorem (Existence of Fixed Point)

Proof:

By the definition of $\perp$ and f: $L \rightarrow L$, we have

$$\perp \sqsubseteq f(\perp)$$

As f is monotonic, we have

$$f(\perp) \sqsubseteq f(f(\perp)) = f^2(\perp)$$

Similarly (by repeatedly applying f), we have

$$\perp \sqsubseteq f(\perp) \sqsubseteq f^2(\perp) \sqsubseteq \ \dots \sqsubseteq f^i(\perp)$$

As L is finite, for some k, we have

$$f^{Fix} = f^k(\perp) = f^{k+1}(\perp)$$

Thus, the fixed point exists.

Fixed-Point Theorem (Least Fixed Point)

Proof:
Assume we have another fixed point x, i.e., $x = f(x)$

Fixed-Point Theorem (Least Fixed Point)

Proof:

Assume we have another fixed point x, i.e., x = f(x)

By the definition of $\perp$, we have $\perp \sqsubseteq x$

Fixed-Point Theorem (Least Fixed Point)

Proof:

Assume we have another fixed point x, i.e., $x = f(x)$

By the definition of $\bot$, we have $\bot \sqsubseteq x$

Induction begins:

Fixed-Point Theorem (Least Fixed Point)

Proof:

Assume we have another fixed point x, i.e., $x = f(x)$

By the definition of $\perp$, we have $\perp \sqsubseteq x$

Induction begins:

As f is monotonic, we have

$$f(\perp) \sqsubseteq f(x)$$

Fixed-Point Theorem (Least Fixed Point)

Proof:

Assume we have another fixed point x, i.e., $x = f(x)$

By the definition of $\perp$, we have $\perp \sqsubseteq x$

Induction begins:

As f is monotonic, we have

$$f(\perp) \sqsubseteq f(x)$$

Assume $f^i(\perp) \sqsubseteq f^i(x)$, as f is monotonic, we have

$$f^{i+1}(\perp) \sqsubseteq f^{i+1}(x)$$

Fixed-Point Theorem (Least Fixed Point)

Proof:

Assume we have another fixed point x, i.e., $x = f(x)$

By the definition of $\perp$, we have $\perp \sqsubseteq x$

Induction begins:

As f is monotonic, we have

$$f(\perp) \sqsubseteq f(x)$$

Assume $f^i(\perp) \sqsubseteq f^i(x)$, as f is monotonic, we have

$$f^{i+1}(\perp) \sqsubseteq f^{i+1}(x)$$

Thus by induction, we have

$$f^i(\perp) \sqsubseteq f^i(x)$$

Fixed-Point Theorem (Least Fixed Point)

Proof:

Assume we have another fixed point x, i.e., $x = f(x)$

By the definition of $\perp$, we have $\perp \sqsubseteq x$

Induction begins:

As f is monotonic, we have

$$f(\perp) \sqsubseteq f(x)$$

Assume $f^i(\perp) \sqsubseteq f^i(x)$, as f is monotonic, we have

$$f^{i+1}(\perp) \sqsubseteq f^{i+1}(x)$$

Thus by induction, we have

$$f^i(\perp) \sqsubseteq f^i(x)$$

Thus $f^i(\perp) \sqsubseteq f^i(x) = x$, then we have

$$f^{Fix} = f^k(\perp) \sqsubseteq x$$

Thus the fixed point is the least

Fixed-Point Theorem (Least Fixed Point)

Proof:

Assume we have another fixed point x, i.e., $x = f(x)$

By the definition of $\perp$, we have $\perp \sqsubseteq x$

Induction begins:

As f is monotonic, we have

$$f(\perp) \sqsubseteq f(x)$$

Assume $f^i(\perp) \sqsubseteq f^i(x)$, as f is monotonic, we have

$$f^{i+1}(\perp) \sqsubseteq f^{i+1}(x)$$

Thus by induction, we have

$$f^i(\perp) \sqsubseteq f^i(x)$$

Thus $f^i(\perp) \sqsubseteq f^i(x) = x$, then we have

$$f^{Fix} = f^k(\perp) \sqsubseteq x$$

Thus the fixed point is the least

The proof for greatest fixed point is similar

Fixed-Point Theorem

Given a complete lattice $(L, \sqsubseteq)$, if

(1) $f: L \rightarrow L$ is monotonic and (2) L is finite, then

the least fixed point of f can be found by iterating

$f(\bot), f(f(\bot)), \ldots, f^k(\bot)$ until a fixed point is reached

the greatest fixed point of f can be found by iterating

$f(\top), f(f(\top)), \ldots, f^k(\top)$ until a fixed point is reached

Review The Questions We Have Seen Before

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

- Is the algorithm guaranteed to terminate or reach the fixed point, or does it always have a solution?
- If so, is there only one solution or only one fixed point? If more than one, is our solution the best one (most precise)?
- When will the algorithm reach the fixed point, or when can we get the solution?

Review The Questions We Have Seen Before

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

- ? Is the algorithm guaranteed to terminate or reach the fixed point, or does it always have a solution?
- ? If so, is there only one solution or only one fixed point? If more than one, is our solution the best one (most ?
- When will the algorithm reach the fixed p can we get the solution?

greatest or least fixed point

Review The Questions We Have Seen Before

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

- Is the algorithm guaranteed to terminate or reach the fixed point, or does it always have a solution?
- If so, is there only one solution or only one fixed point? If more than one, is our solution the best one (most [illegible]?
- When will the algorithm reach the fixed [illegible] can we get the solution?

greatest or least fixed point

Now what we have just seen is the property (fixed point theorem) for the function on a lattice. We cannot say our iterative algorithm also has that property unless we can *relate the algorithm to the fixed point theorem*, if possible

Relate Iterative Algorithm to Fixed-Point Theorem

$\rightarrow (\perp, \perp, \ldots, \perp)$

iter 1 $\rightarrow (v_1^1, v_2^1, \ldots, v_k^1)$

iter 2 $\rightarrow (v_1^2, v_2^2, \ldots, v_k^2)$

$\vdots$

iter i $\rightarrow (v_1^i, v_2^i, \ldots, v_k^i)$

iter i+1 $\rightarrow (v_1^i, v_2^i, \ldots, v_k^i)$

$\Downarrow$

Given a complete lattice $(L, \sqsubseteq)$, if

(1) $f: L \rightarrow L$ is monotonic and (2) L is finite, then

the least fixed point of f can be found by iterating

$f(\perp), f(f(\perp)), \ldots, f^k(\perp)$ until a fixed point is reached

the greatest fixed point of f can be found by iterating

$f(\top), f(f(\top)), \ldots, f^k(\top)$ until a fixed point is reached

Relate Iterative Algorithm to Fixed-Point Theorem

(L, L, …, L)

$$\rightarrow (\bot, \ \bot, \ \ldots, \ \bot)$$

iter 1 $\rightarrow (v_1^1, v_2^1, \ldots, v_k^1)$

iter 2 $\rightarrow (v_1^2, v_2^2, \ldots, v_k^2)$

$\vdots$

iter i $\rightarrow (v_1^i, v_2^i, \ldots, v_k^i)$

iter i+1 $\rightarrow (v_1^i, v_2^i, \ldots, v_k^i)$

If a product lattice L^k is a product of complete (and finite) lattices, i.e., (L, L, …, L), then L^k is also complete (and finite)

$\Downarrow$

Given a complete lattice (L, ⊑), if

(1) f: L → L is monotonic and (2) L is finite, then

the least fixed point of f can be found by iterating

$f(\bot), f(f(\bot)), \ldots, f^k(\bot)$ until a fixed point is reached

the greatest fixed point of f can be found by iterating

$f(\top), f(f(\top)), \ldots, f^k(\top)$ until a fixed point is reached

Relate Iterative Algorithm to Fixed-Point Theorem

(L, L, …, L)

$$(\bot,\ \bot,\ \dots,\ \bot)$$

iter 1 $\rightarrow (v_1^1, v_2^1, \dots, v_k^1)$

iter 2 $\rightarrow (v_1^2, v_2^2, \dots, v_k^2)$

$\vdots$

iter i $\rightarrow (v_1^i, v_2^i, \dots, v_k^i)$

iter i+1 $\rightarrow (v_1^i, v_2^i, \dots, v_k^i)$

If a product lattice L^k is a product of complete (and finite) lattices, i.e., (L, L, …, L), then L^k is also complete (and finite)

In each iteration, it is equivalent to think that we apply function F which consists of

(1) transfer function f_i: L → L for every node
(2) join/meet function ⊔/⊓: L×L → L for control-flow confluence

Given a complete lattice (L, ⊑), if
(1) f: L → L is monotonic and (2) L is finite, then
the least fixed point of f can be found by iterating
$f(\bot), f(f(\bot)), \dots, f^k(\bot)$ until a fixed point is reached
the greatest fixed point of f can be found by iterating
$f(\top), f(f(\top)), \dots, f^k(\top)$ until a fixed point is reached

Relate Iterative Algorithm to Fixed-Point Theorem

$$(L, L, \ldots, L)$$

$$\rightarrow (\bot, \bot, \ldots, \bot)$$

iter 1 $\rightarrow (v_1^1, v_2^1, \ldots, v_k^1)$

iter 2 $\rightarrow (v_1^2, v_2^2, \ldots, v_k^2)$

$\vdots$

iter i $\rightarrow (v_1^i, v_2^i, \ldots, v_k^i)$

iter i+1 $\rightarrow (v_1^i, v_2^i, \ldots, v_k^i)$

If a product lattice L^k is a product of complete (and finite) lattices, i.e., (L, L, …, L), then L^k is also complete (and finite)

In each iteration, it is equivalent to think that we apply function F which consists of

(1) transfer function f_i: L → L for every node
(2) join/meet function ⊔/⊓: L×L → L for control-flow confluence

Given a complete lattice (L, ⊑), if (1) f: L → L is monotonic and (2) L is finite, then
the least fixed point of f can be found by iterating
$f(\bot), f(f(\bot)), \ldots, f^k(\bot)$ until
the greatest fixed point of f can b
$f(\top), f(f(\top)), \ldots, f^k(\top)$ until

Now the remaining issue is to prove that function F is monotonic

Prove Function F is Monotonic

In each iteration, it is equivalent to think that we apply function F which consists of

(1) transfer function f_i: L → L for every node

(2) join/meet function ⊔/⊓: L×L → L for control-flow confluence

Prove Function F is Monotonic

In each iteration, it is equivalent to think that we apply function F which consists of
(1) transfer function f_i: L → L for every node
(2) join/meet function ⊔/⊓: L×L → L for control-flow confluence

Gen/Kill function is monotonic

Prove Function F is Monotonic

In each iteration, it is equivalent to think that we apply function F which consists of
(1) transfer function f_i: $L \rightarrow L$ for every node
(2) join/meet function $\sqcup / \sqcap$: $L \times L \rightarrow L$ for control-flow confluence

Actually the binary operator is a basic case of $L \times L \times \ldots \times L$,

Gen/Kill function is monotonic

Prove Function F is Monotonic

In each iteration, it is equivalent to think that we apply function F which consists of

(1) transfer function f_i: $L \rightarrow L$ for every node

(2) join/meet function $\sqcup / \sqcap$: $L \times L \rightarrow L$ for control-flow confluence

Gen/Kill function is monotonic

Actually the binary operator is a basic case of $L \times L \times \ldots \times L$,

We want to show that $\sqcup$ is monotonic

Prove Function F is Monotonic

In each iteration, it is equivalent to think that we apply function F which consists of

(1) transfer function f_i: L → L for every node

(2) join/meet function ⊔/⊓: L×L → L for control-flow confluence

Gen/Kill function is monotonic

Actually the binary operator is a basic case of L × L ×… × L,

We want to show that ⊔ is monotonic

Proof.

$\forall x, y, z \in L, x \sqsubseteq y$, we want to prove $x \sqcup z \sqsubseteq y \sqcup z$

Prove Function F is Monotonic

In each iteration, it is equivalent to think that we apply function F which consists of
(1) transfer function f_i: L → L for every node
(2) join/meet function ⊔/⊓: L×L → L for control-flow confluence

Actually the binary operator is a basic case of L × L ×… × L,

Gen/Kill function is monotonic

We want to show that ⊔ is monotonic

Proof.

$\forall x, y, z \in L, x \sqsubseteq y$, we want to prove $x \sqcup z \sqsubseteq y \sqcup z$

by the definition of ⊔, $y \sqsubseteq y \sqcup z$

Prove Function F is Monotonic

In each iteration, it is equivalent to think that we apply function F which consists of
(1) transfer function f_i: L → L for every node
(2) join/meet function ⊔/⊓: L×L → L for control-flow confluence

Actually the binary operator is a basic case of L × L ×… × L,

Gen/Kill function is monotonic

We want to show that ⊔ is monotonic

Proof.

$\forall x, y, z \in L, x \sqsubseteq y$, we want to prove $x \sqcup z \sqsubseteq y \sqcup z$

by the definition of $\sqcup$, $y \sqsubseteq y \sqcup z$

by transitivity of $\sqsubseteq$, $x \sqsubseteq y \sqcup z$

Prove Function F is Monotonic

In each iteration, it is equivalent to think that we apply function F which consists of
(1) transfer function f_i: L → L for every node
(2) join/meet function ⊔/⊓: L×L → L for control-flow confluence

Actually the binary operator is a basic case of L × L ×… × L,

Gen/Kill function is monotonic

We want to show that ⊔ is monotonic

Proof.
$\forall x, y, z \in L$, $x \sqsubseteq y$, we want to prove $x \sqcup z \sqsubseteq y \sqcup z$
by the definition of ⊔, $y \sqsubseteq y \sqcup z$
by transitivity of ⊑, $x \sqsubseteq y \sqcup z$
thus $y \sqcup z$ is an upper bound for x, and also for z (by ⊔'s definition)

Prove Function F is Monotonic

In each iteration, it is equivalent to think that we apply function F which consists of
(1) transfer function f_i: L → L for every node
(2) join/meet function ⊔/⊓: L×L → L for control-flow confluence

Actually the binary operator is a basic case of L × L ×… × L,

Gen/Kill function is monotonic

We want to show that ⊔ is monotonic

Proof.

$\forall x, y, z \in L, x \sqsubseteq y$, we want to prove $x \sqcup z \sqsubseteq y \sqcup z$

by the definition of ⊔, $y \sqsubseteq y \sqcup z$

by transitivity of ⊑, $x \sqsubseteq y \sqcup z$

thus $y \sqcup z$ is an upper bound for x, and also for z (by ⊔'s definition)

as $x \sqcup z$ is the least upper bound of x and z

Prove Function F is Monotonic

In each iteration, it is equivalent to think that we apply function F which consists of
(1) transfer function f_i: L → L for every node
(2) join/meet function ⊔/⊓: L×L → L for control-flow confluence

Actually the binary operator is a basic case of L × L ×… × L,

Gen/Kill function is monotonic

We want to show that ⊔ is monotonic

Proof.
$\forall x, y, z \in L$, $x \sqsubseteq y$, we want to prove $x \sqcup z \sqsubseteq y \sqcup z$
by the definition of ⊔, $y \sqsubseteq y \sqcup z$
by transitivity of ⊑, $x \sqsubseteq y \sqcup z$
thus $y \sqcup z$ is an upper bound for x, and also for z (by ⊔'s definition)
as $x \sqcup z$ is the least upper bound of x and z
thus $x \sqcup z \sqsubseteq y \sqcup z$

Prove Function F is Monotonic

In each iteration, it is equivalent to think that we apply function F which consists of

(1) transfer function f_i: L → L for every node
(2) join/meet function ⊔/⊓: L×L → L for control-flow confluence

Actually the binary operator is a basic case of L × L ×… × L,

Gen/Kill function is monotonic

We want to show that ⊔ is monotonic

Proof.

∀x, y, z ∈ L, x ⊑ y, we want to prove x ⊔ z ⊑ y ⊔ z

by the definition of ⊔, y ⊑ y ⊔ z

by transitivity of ⊑, x ⊑ y ⊔ …

thus y ⊔ z is an upp… (by ⊔'s definition)

as x ⊔ z is the least …ound of x and z

thus x ⊔ z ⊑ y ⊔ z

Thus the fixed point theorem applies to the iterative algorithm for data flow analysis

Review The Questions We Have Seen Before

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

- Is the algorithm guaranteed to terminate or reach the fixed point, or does it always have a solution?
- If so, is there only one solution or only one fixed point? If more than one, is our solution the best one (m…

greatest or least fixed point

- When will the algorithm reach the fixed point, or when can we get the solution?

Now what we have just seen is the property (fixed point theorem) for the function on a lattice. ~~We cannot say our iterative algorithm also has that property unless~~ we can relate the algorithm to the fixed point theorem, ~~if possible~~

Review The Questions We Have Seen Before

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

YES

- Is the algorithm guaranteed to terminate or reach the fixed point, or does it always have a solution?
- If so, ~~is there only one solution or only one fixed point~~? If more than one, is our solution the best one (m

greatest or least fixed point

YES

- When will the algorithm reach the fixed point, or when can we get the solution?

Now what we have just seen is the property (fixed point theorem) for the function on a lattice. ~~We cannot say our iterative algorithm also has that property unless~~ we can relate the algorithm to the fixed point theorem, ~~if possible~~

Review The Questions We Have Seen Before

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

YES

- Is the algorithm guaranteed to terminate or reach the fixed point, or does it always have a solution?

- If so, ~~is there only one solution or only one fixed point~~? If more than one, is our solution the best one (m

greatest or least fixed point

YES

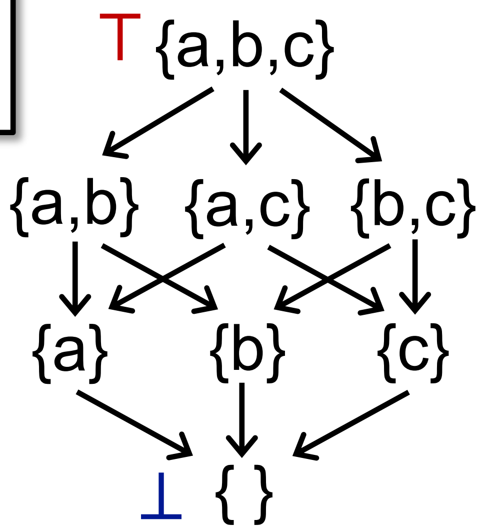
- When will the algorithm reach the fixed point, or when can we get the solution?

Now what we have just seen is the property (fixed point theorem) for the function on a lattice. ~~We cannot say our iterative algorithm also has that property unless~~ we can relate the algorithm to the fixed point theorem, ~~if possible~~

When Will the Algorithm Reach the Fixed Point?

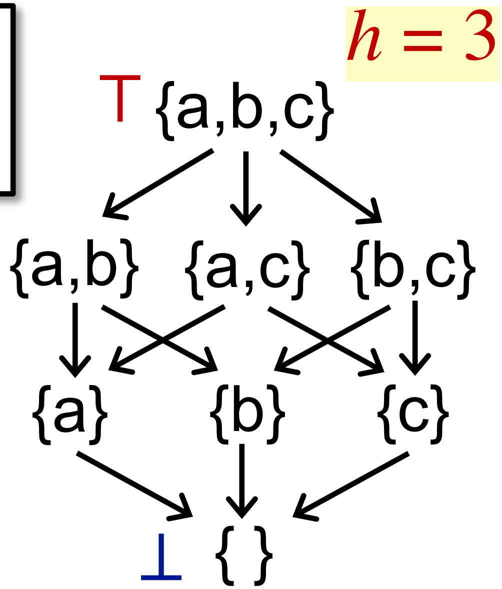
When Will the Algorithm Reach the Fixed Point?

The height of a lattice *h* is the length of the longest path from Top to Bottom in the lattice.



When Will the Algorithm Reach the Fixed Point?

The height of a lattice h is the length of the longest path from Top to Bottom in the lattice.

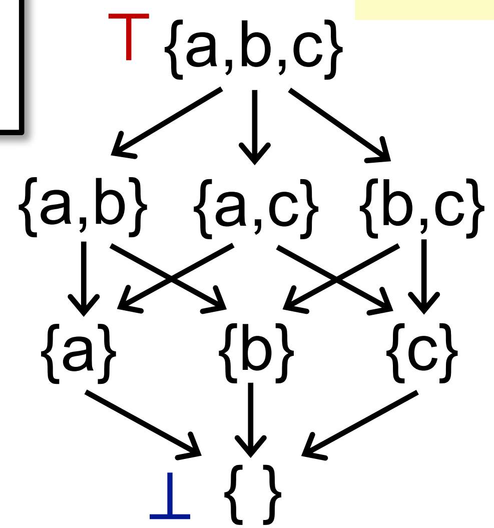


When Will the Algorithm Reach the Fixed Point?

The height of a lattice h is the length of the longest path from Top to Bottom in the lattice.

The maximum iterations i needed to reach the fixed point

$h = 3$



$$\rightarrow (\bot, \ \bot, \ \ldots, \ \bot)$$

iter 1 $\rightarrow (v_1^1, v_2^1, \ldots, v_k^1)$

iter 2 $\rightarrow (v_1^2, v_2^2, \ldots, v_k^2)$

$$\vdots$$

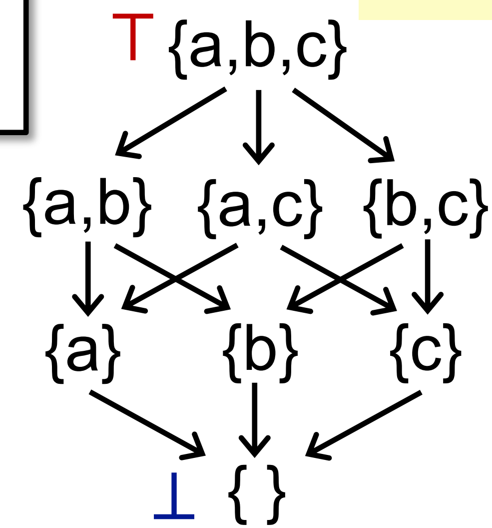
iter i $\rightarrow (v_1^i, v_2^i, \ldots, v_k^i)$

iter i+1 $\rightarrow (v_1^i, v_2^i, \ldots, v_k^i)$

When Will the Algorithm Reach the Fixed Point?

The **height of a lattice h** is the length of the longest path from Top to Bottom in the lattice.

$h = 3$



The maximum iterations i needed to reach the fixed point

$$(\perp, \ \perp, \ \ldots, \ \perp)$$

iter 1 $(v_1^1, v_2^1, \ldots, v_k^1)$

iter 2 $(v_1^2, v_2^2, \ldots, v_k^2)$

$\vdots$

iter i $(v_1^i, v_2^i, \ldots, v_k^i)$

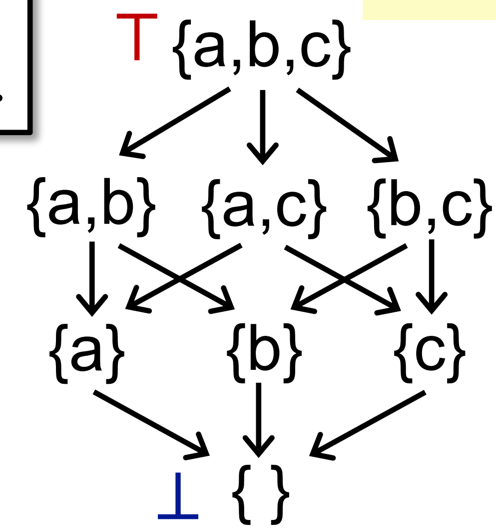
iter i+1 $(v_1^i, v_2^i, \ldots, v_k^i)$

In each iteration, assume only one step in the lattice (upwards or downwards) is made in one node (e.g., one 0->1 in RD)

When Will the Algorithm Reach the Fixed Point?

The height of a lattice h is the length of the longest path from Top to Bottom in the lattice.

$h = 3$



The maximum iterations i needed to reach the fixed point

$$\rightarrow (\perp, \ \perp, \ \ldots, \ \perp)$$

$$\textit{iter } 1 \rightarrow (v_1^1, v_2^1, \ldots, v_k^1)$$

$$\textit{iter } 2 \rightarrow (v_1^2, v_2^2, \ldots, v_k^2)$$

$$\vdots$$

$$\textit{iter } i \rightarrow (v_1^i, v_2^i, \ldots, v_k^i)$$

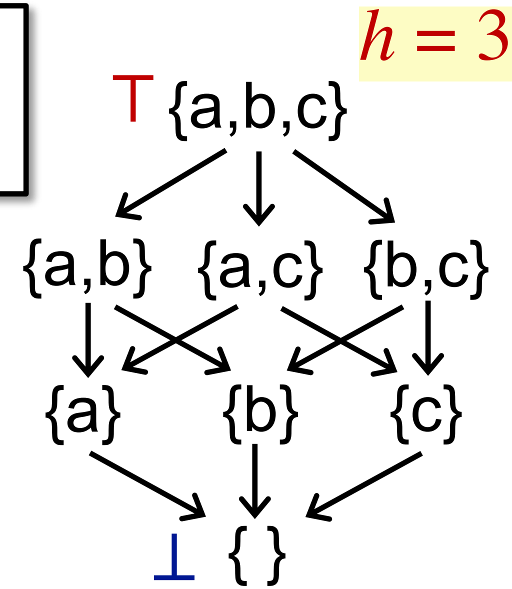
$$\textit{iter } i{+}1 \rightarrow (v_1^i, v_2^i, \ldots, v_k^i)$$

In each iteration, assume only one step in the lattice (upwards or downwards) is made in one node (e.g., one 0->1 in RD)

Assume the lattice height is h and the number of nodes in CFG is k

When Will the Algorithm Reach the Fixed Point?

The **height of a lattice h** is the length of the longest path from Top to Bottom in the lattice.



The maximum iterations i needed to reach the fixed point

$$\rightarrow (\bot, \ \bot, \ \ldots, \ \bot)$$

iter 1 $\rightarrow (v_1^1, v_2^1, \ldots, v_k^1)$

iter 2 $\rightarrow (v_1^2, v_2^2, \ldots, v_k^2)$

$\vdots$

iter i $\rightarrow (v_1^i, v_2^i, \ldots, v_k^i)$

iter i+1 $\rightarrow (v_1^i, v_2^i, \ldots, v_k^i)$

In each iteration, assume only one step in the lattice (upwards or downwards) is made in one node (e.g., one 0->1 in RD)

Assume the lattice height is h and the number of nodes in CFG is k

We need at most $i = h*k$ iterations

Review The Questions We Have Seen Before

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

YES

- Is the algorithm guaranteed to terminate or reach the fixed point, or does it always have a solution?

- If so, ~~is there only one solution or only one fixed point~~? If more than one, is our solution the best one (most precise)?

YES

- When will the algorithm reach the fixed point, or when can we get the solution?

Review The Questions We Have Seen Before

The iterative algorithm (or the IN/OUT equation system) produces a solution to a data flow analysis

YES

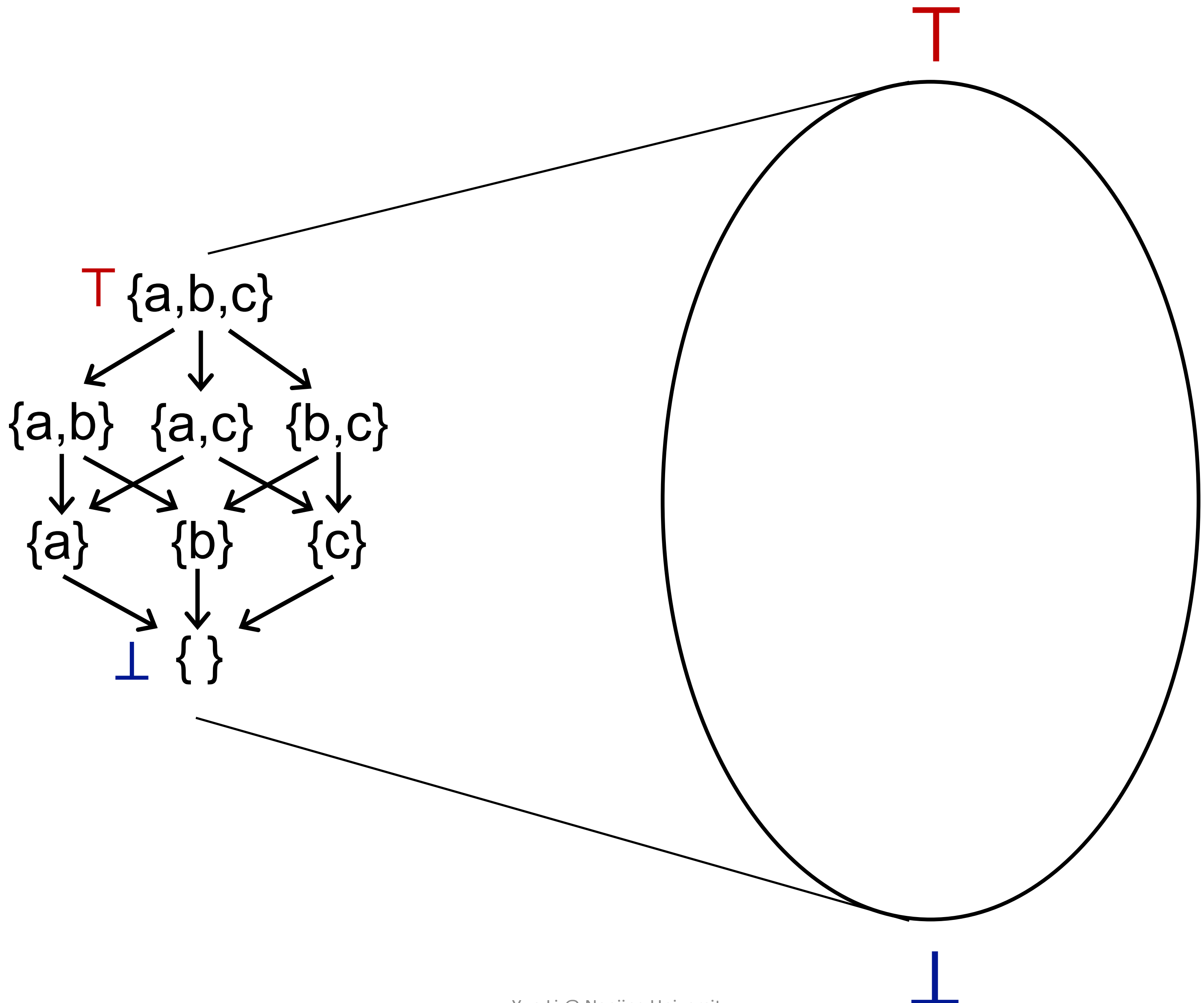
- Is the algorithm guaranteed to terminate or reach the fixed point, or does it always have a solution?
- If so, ~~is there only one solution or only one fixed point~~? If more than one, is our solution the best one (most precise)?

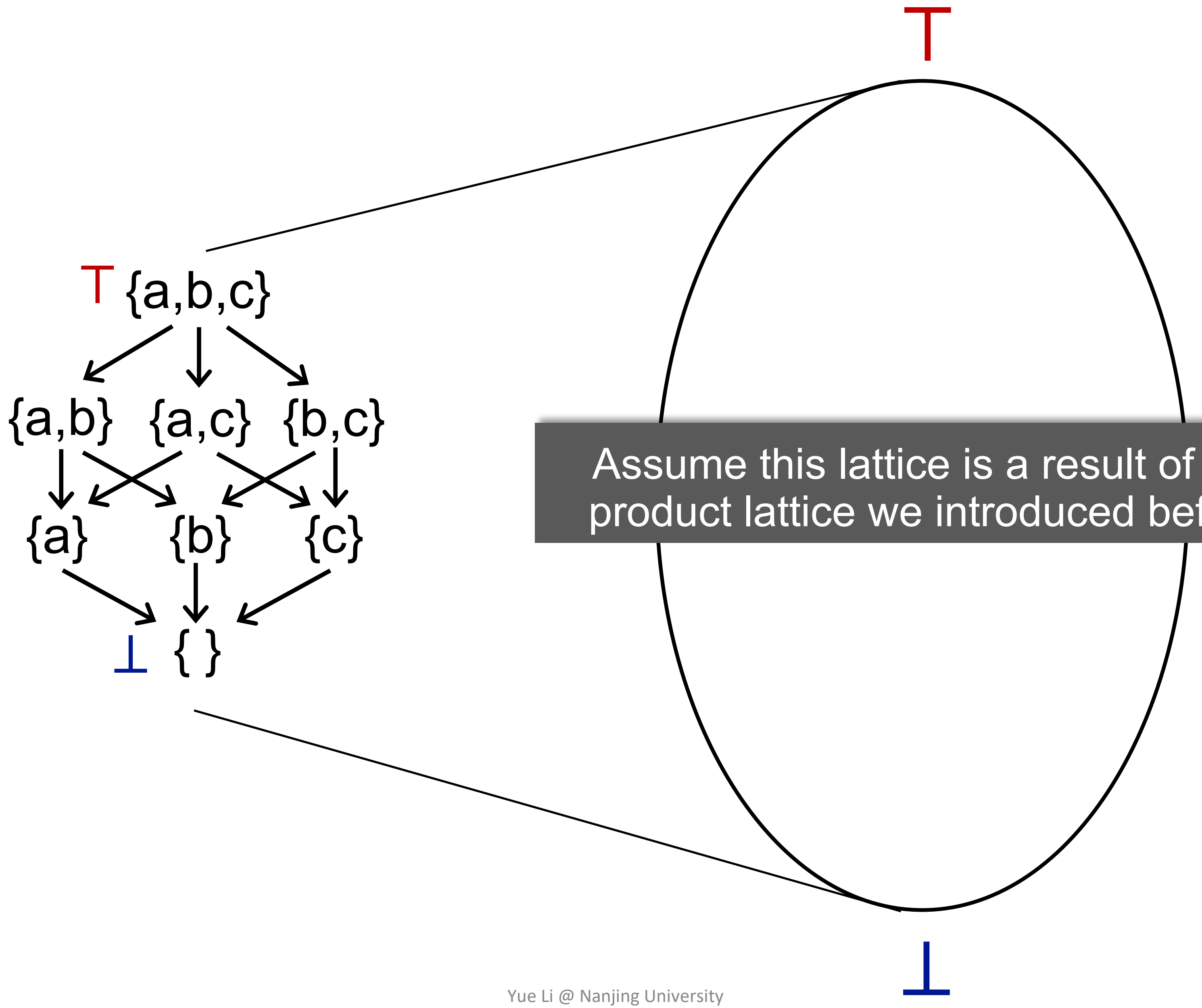
YES

- When will the algorithm reach the fixed point, or when can we get the solution?

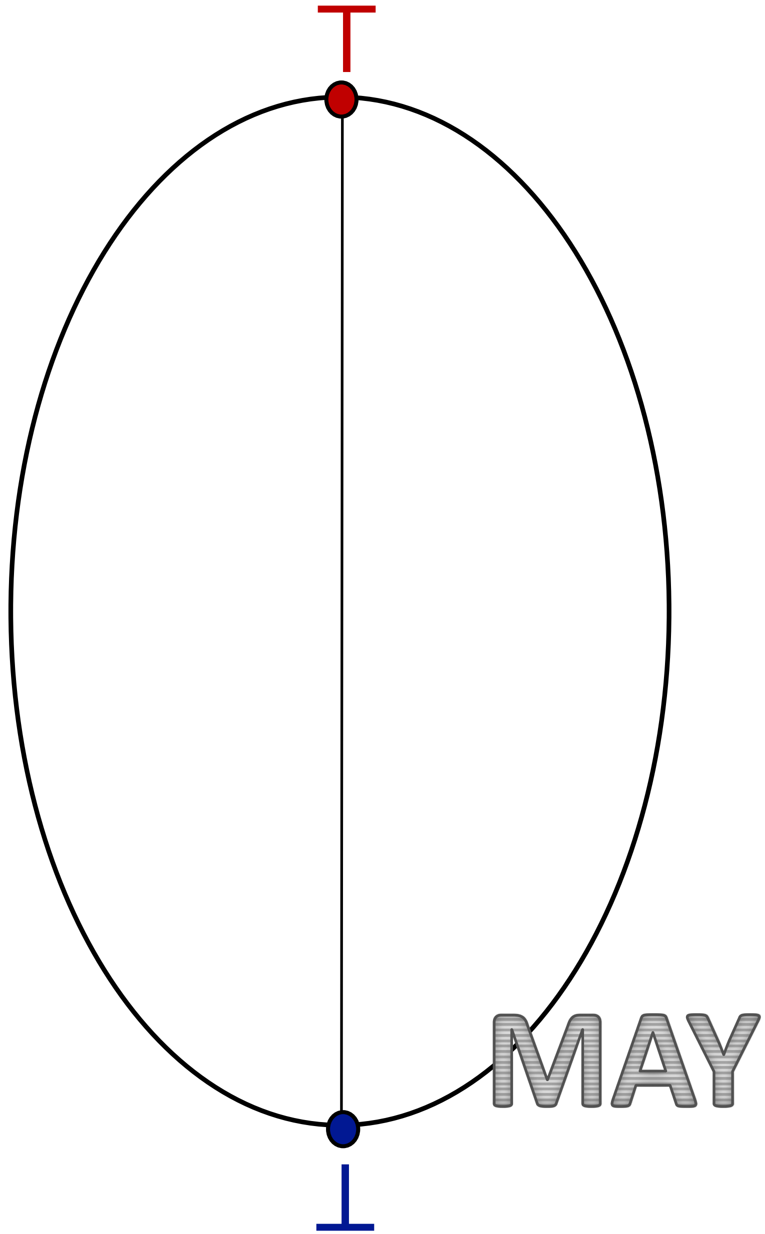
Worst case of #iterations:
the product of the lattice height and the number of nodes in CFG

May and Must Analyses, a Lattice View

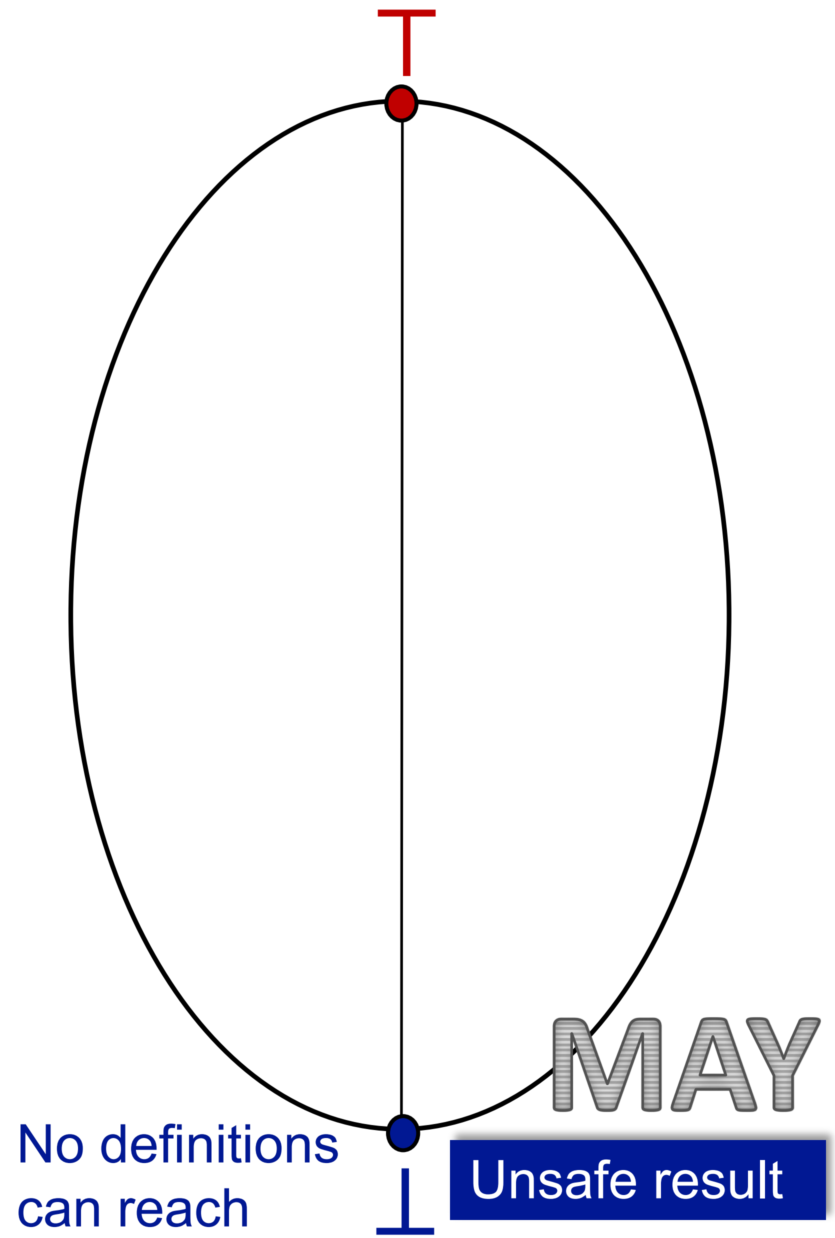

⊤
⊤ {a,b,c}
{a,b} {a,c} {b,c}
{a} {b} {c}
⊥ { }
⊥



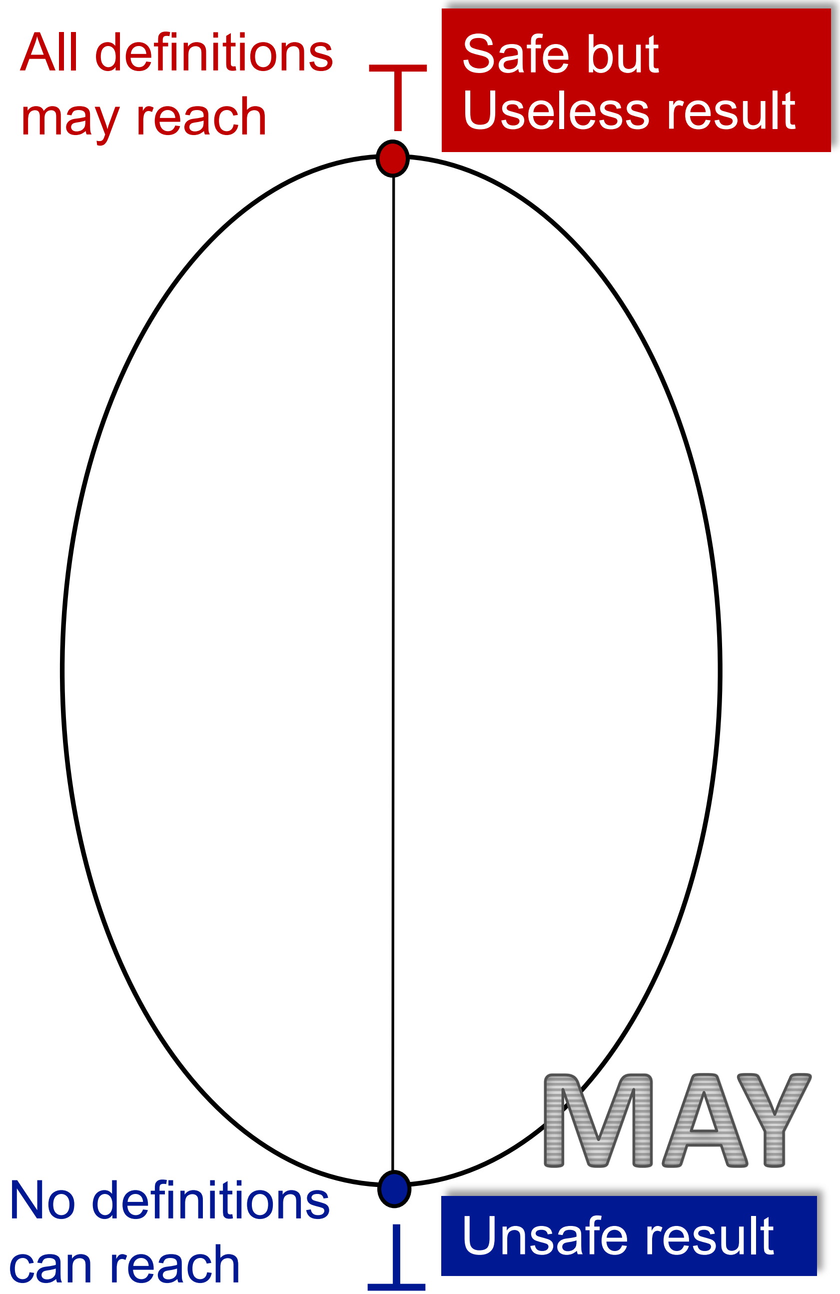
Assume this lattice is a result of the product lattice we introduced before

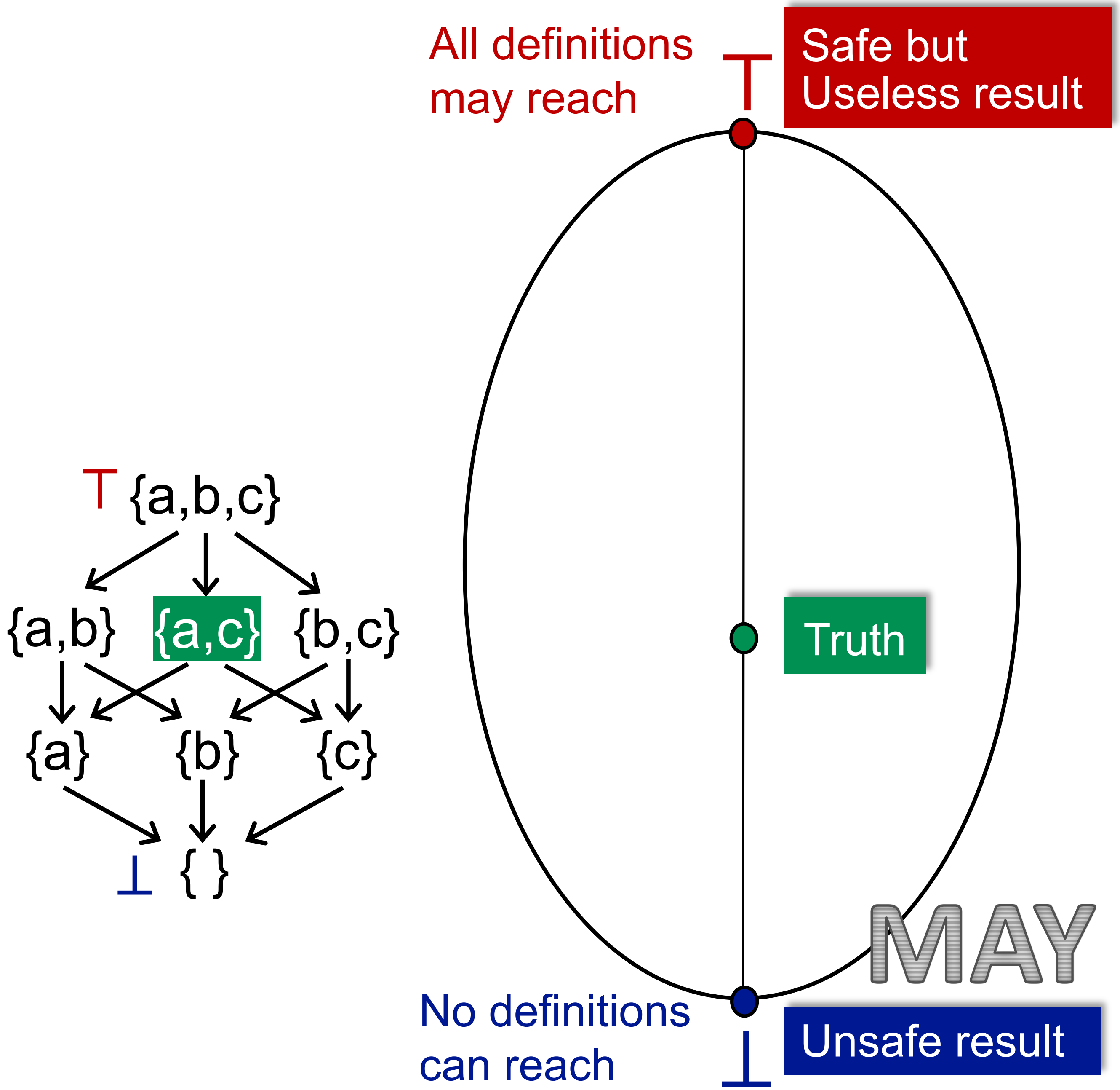


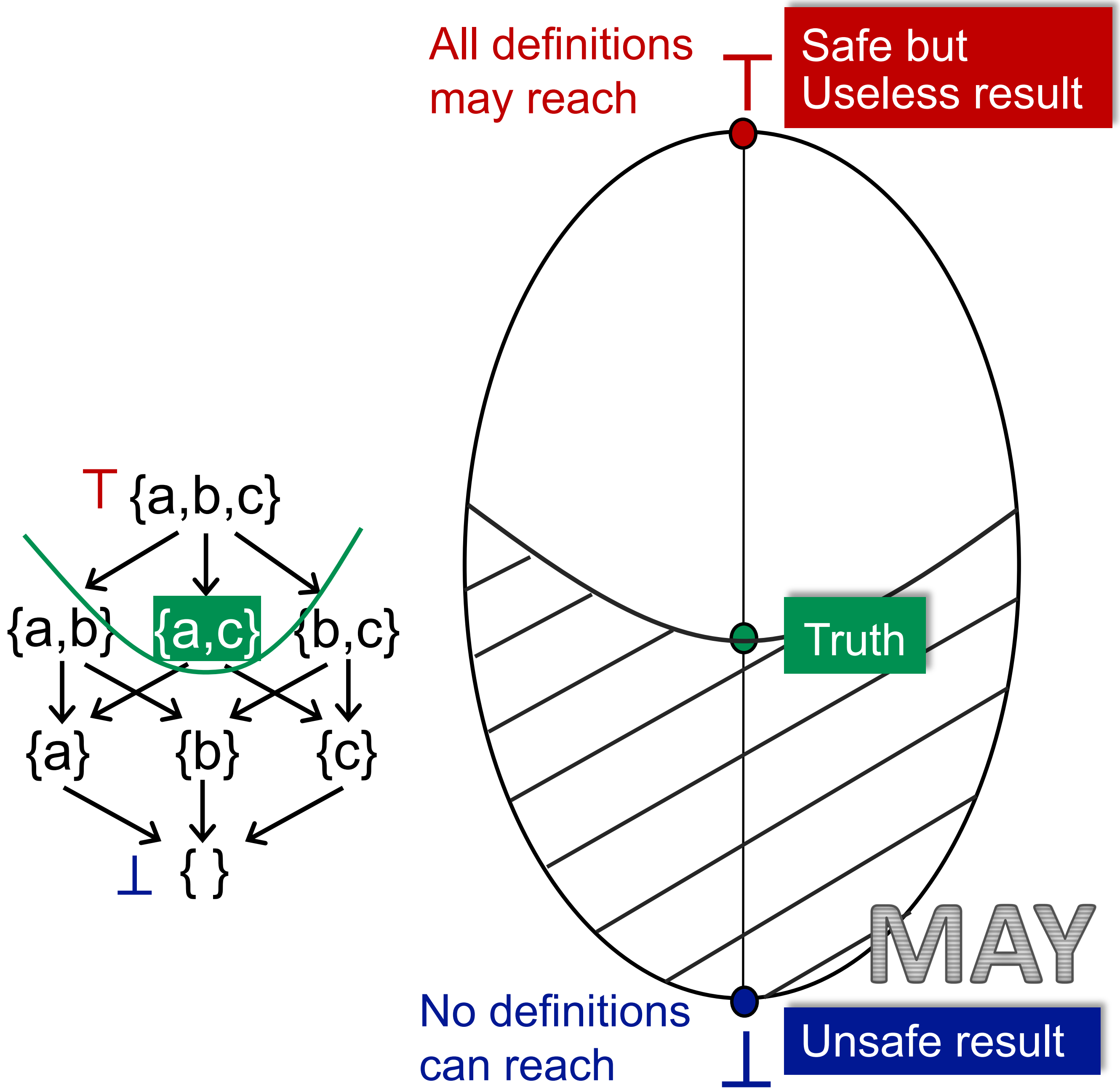
MAY



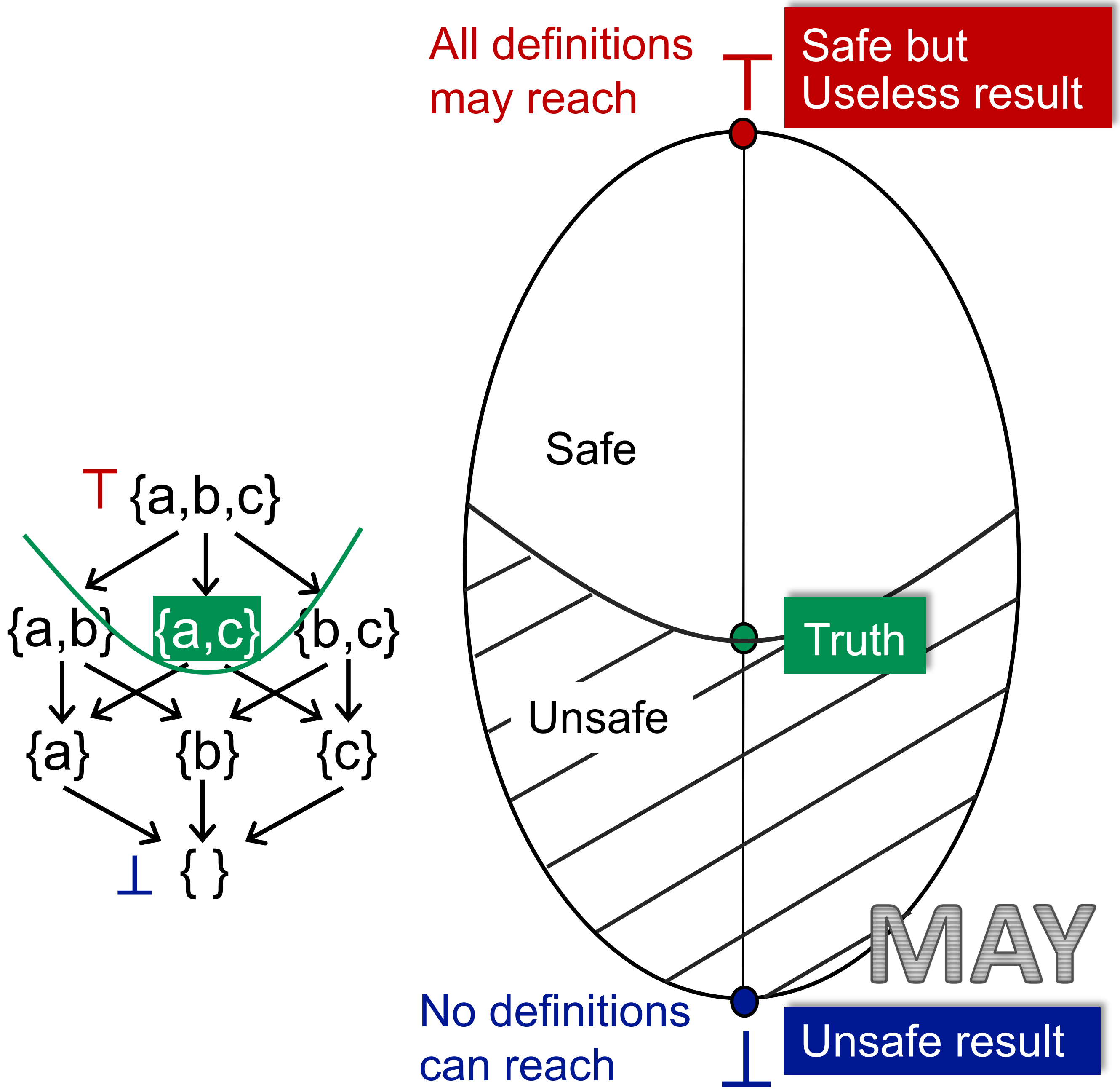
⊤
MAY
No definitions
can reach
⊥
Unsafe result

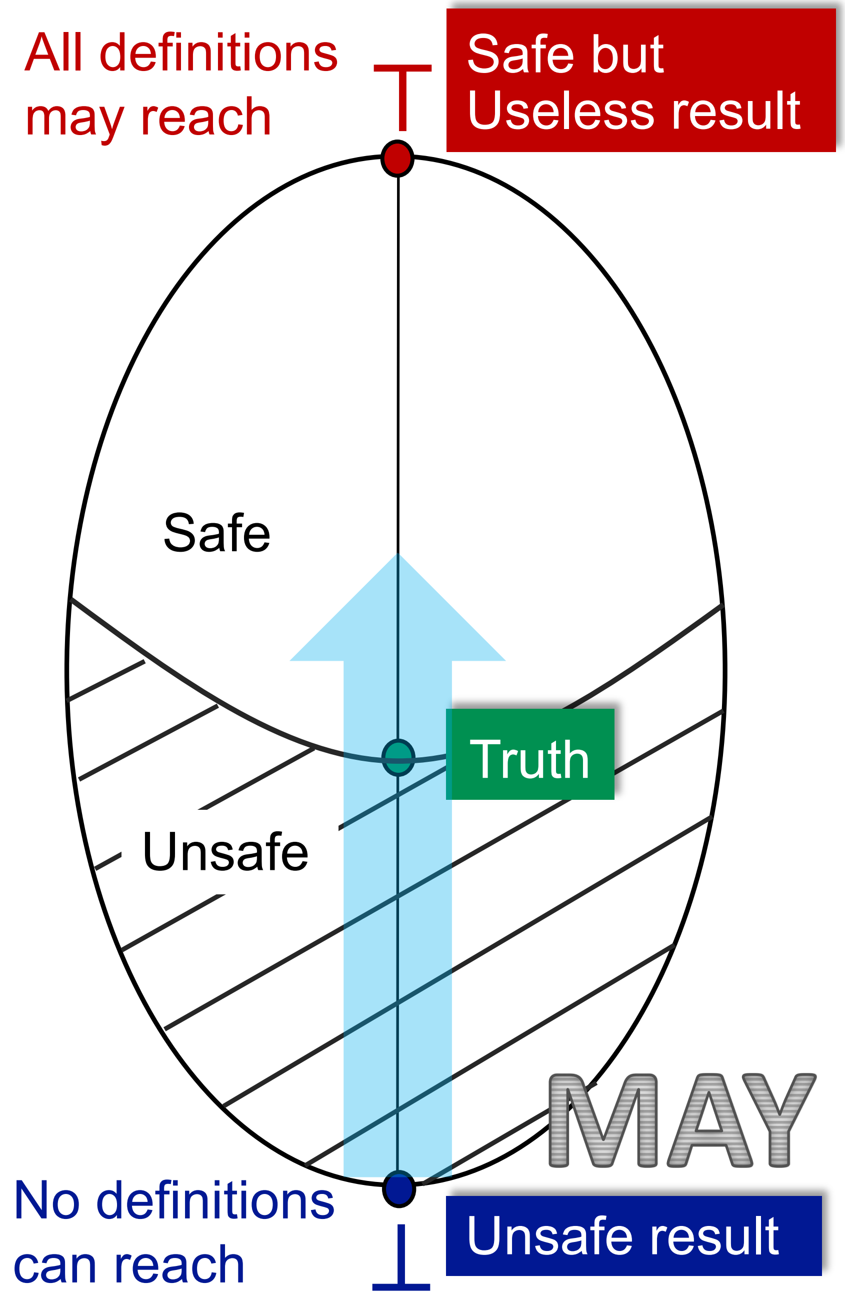

All definitions
may reach
Safe but
Useless result
MAY
No definitions
can reach
Unsafe result

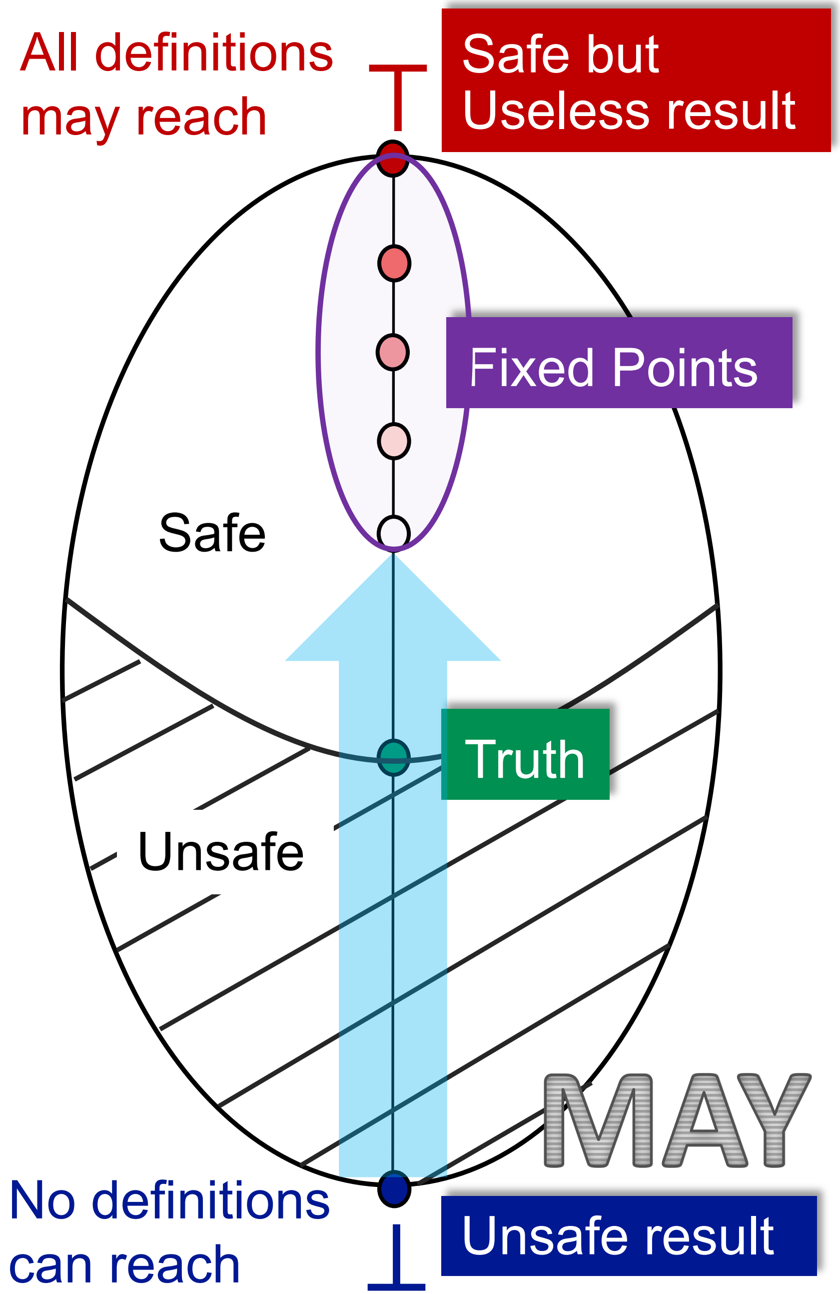

⊤ {a,b,c}
{a,b}
{a,c}
{b,c}
{a}
{b}
{c}
⊥ { }
All definitions may reach
⊤
Safe but Useless result
Truth
MAY
No definitions can reach
⊥
Unsafe result

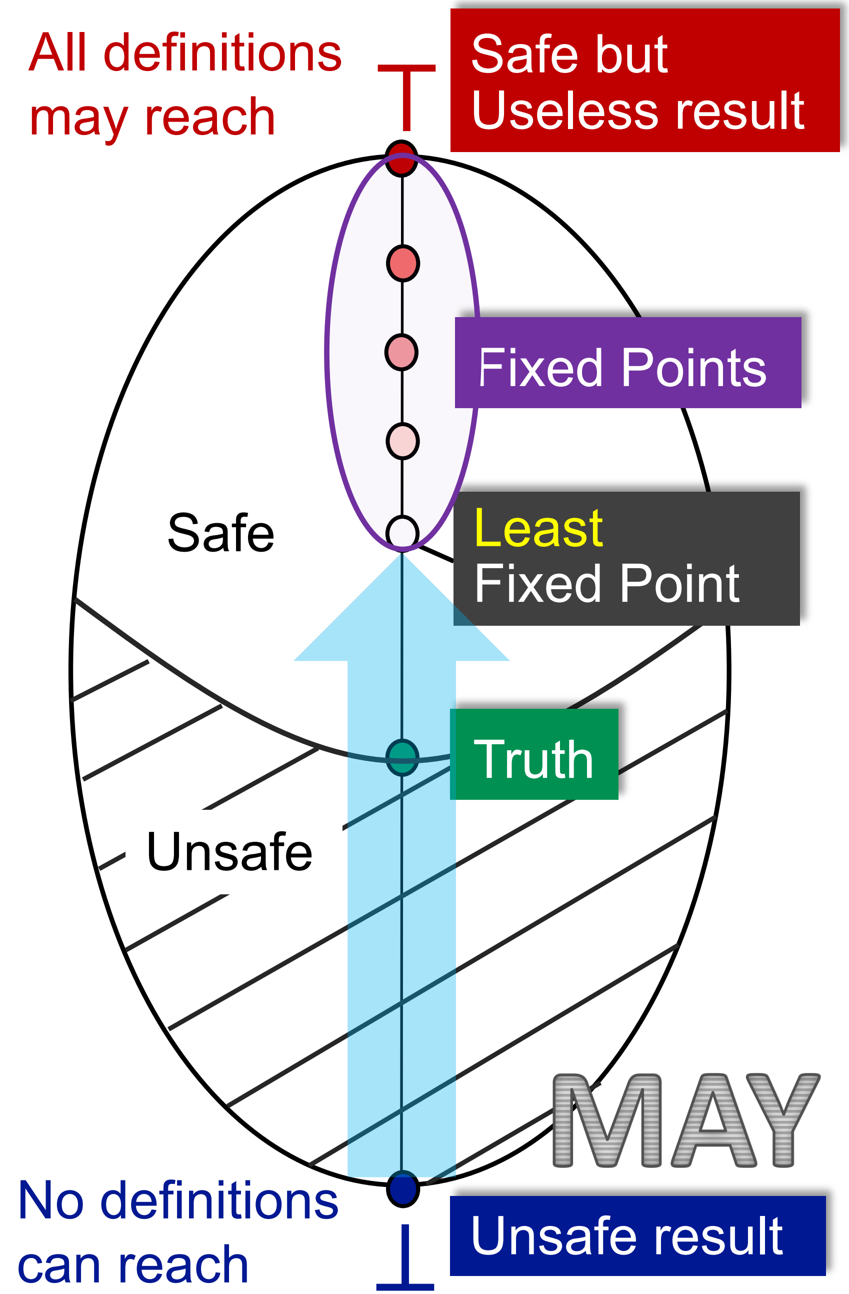


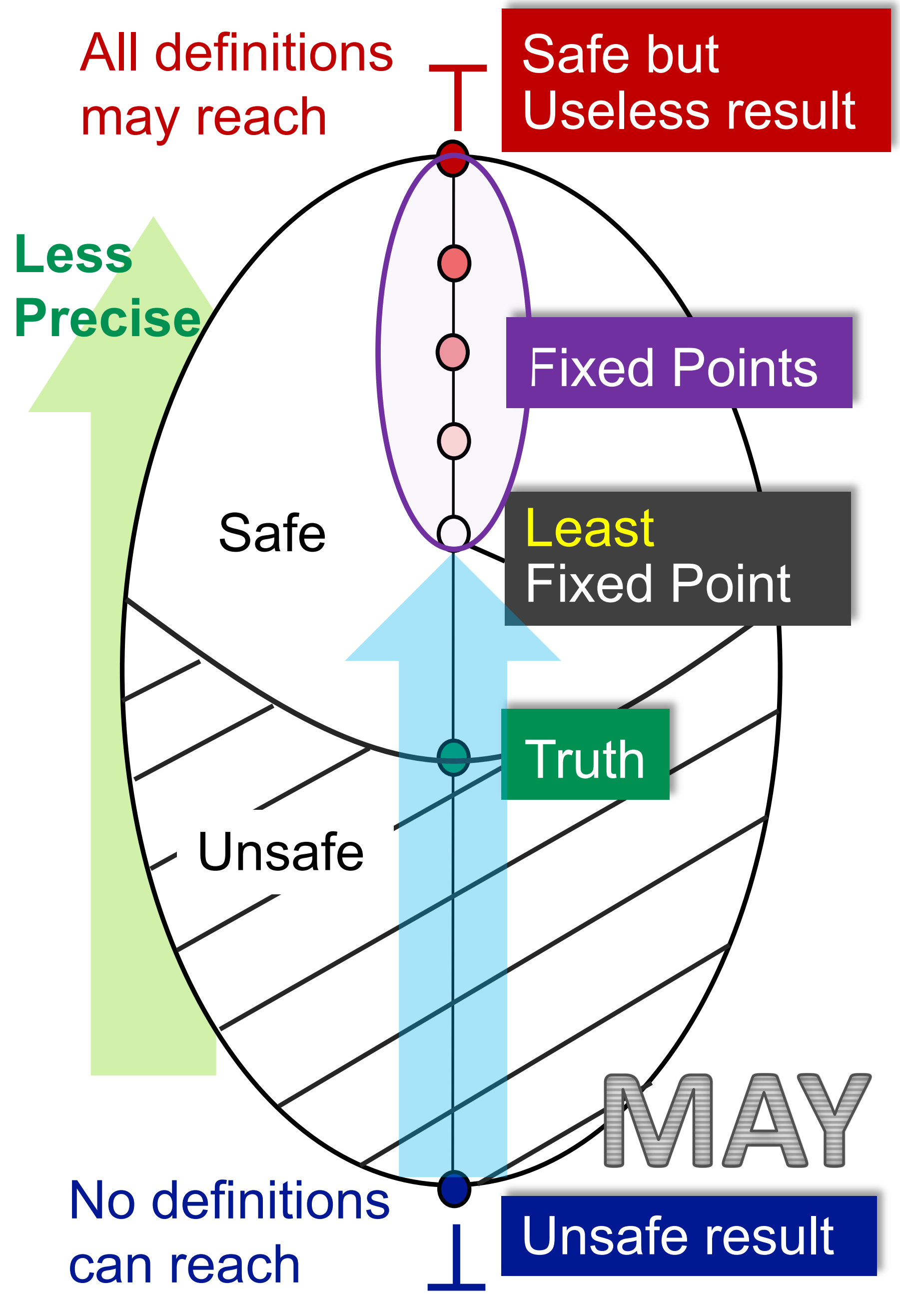
⊤ {a,b,c}
{a,b}
{a,c}
{b,c}
{a}
{b}
{c}
⊥ { }
All definitions may reach
⊤
Safe but Useless result
Truth
MAY
No definitions can reach
⊥
Unsafe result

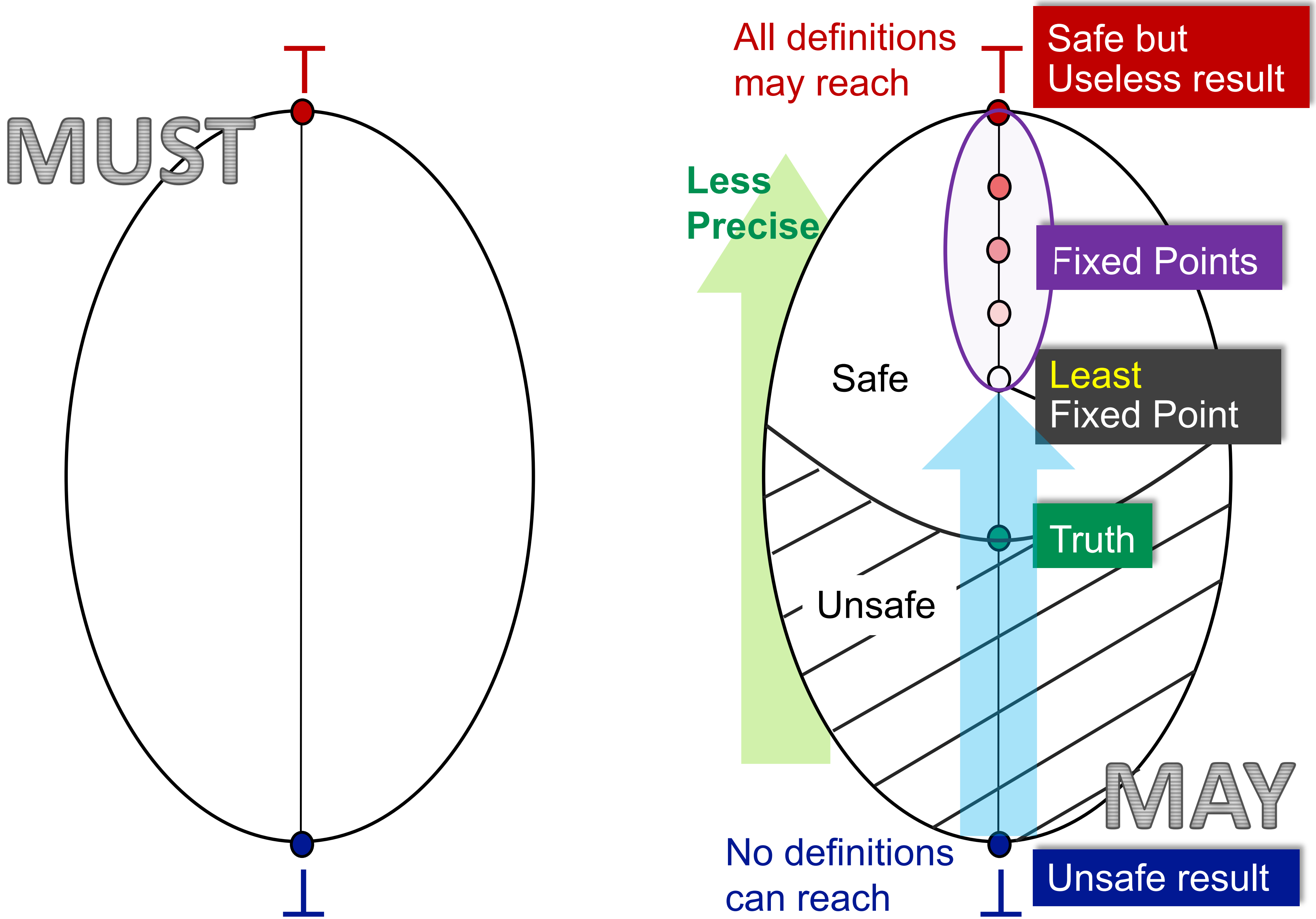

All definitions may reach
Safe but Useless result
⊤
⊤ {a,b,c}
{a,b}
{a,c}
{b,c}
{a}
{b}
{c}
⊥ { }
Safe
Truth
Unsafe
MAY
No definitions can reach
⊥
Unsafe result

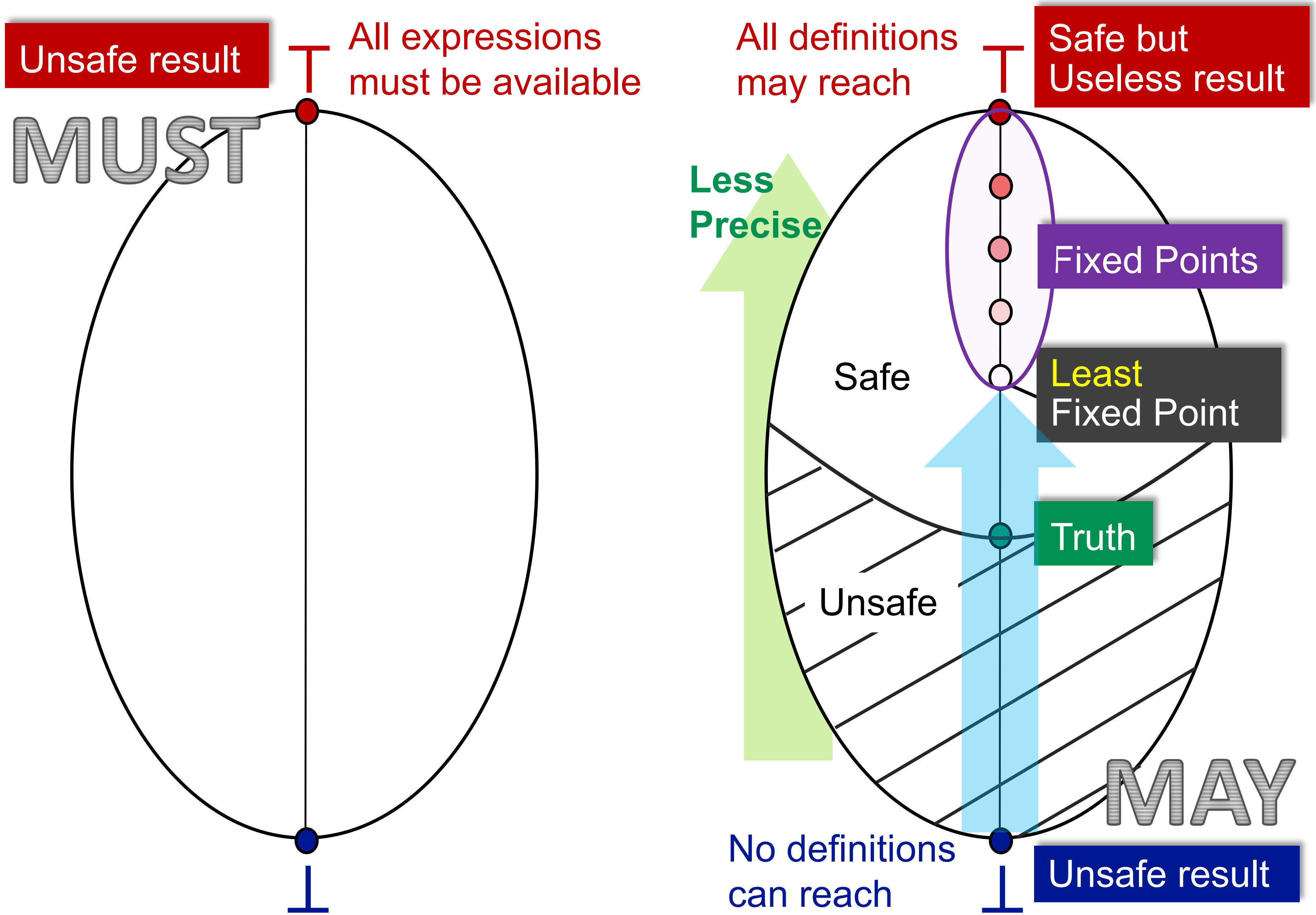

All definitions may reach
Safe but Useless result
Safe
Truth
Unsafe
MAY
No definitions can reach
Unsafe result

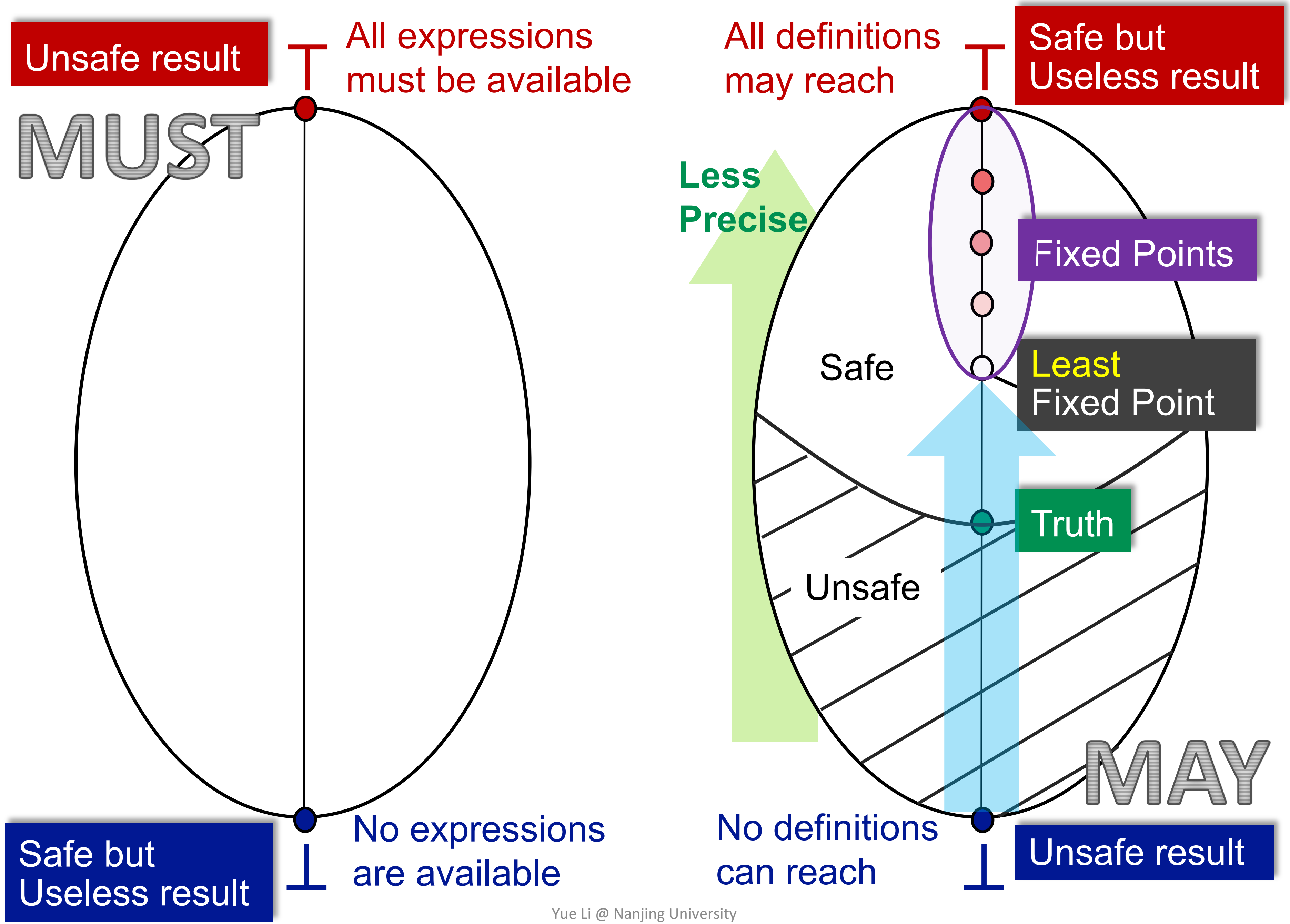

All definitions may reach
⊤
Safe but Useless result
Fixed Points
Safe
Truth
Unsafe
MAY
No definitions can reach
⊥
Unsafe result


All definitions
may reach
Safe but
Useless result
Fixed Points
Safe
Least
Fixed Point
Truth
Unsafe
MAY
No definitions
can reach
Unsafe result

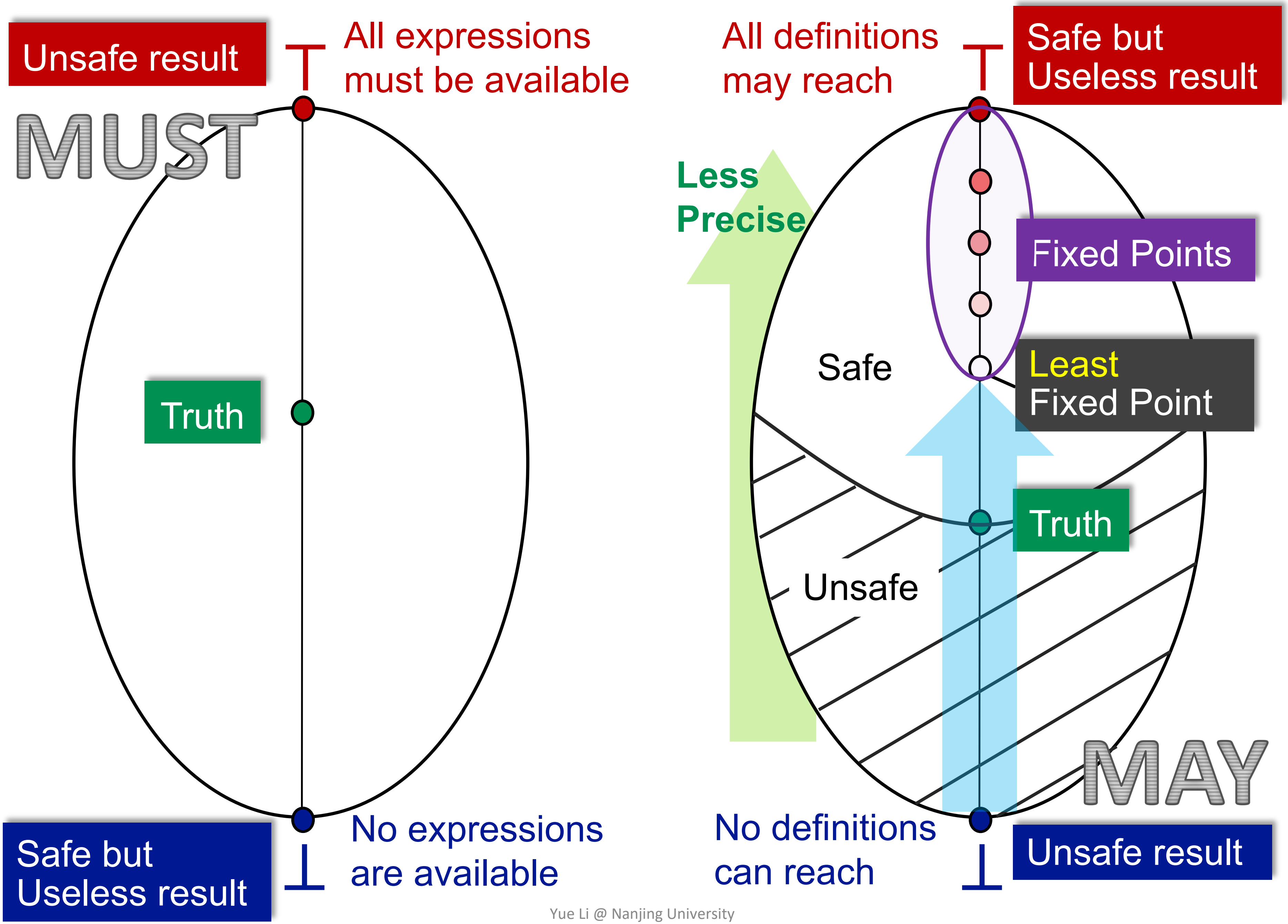

All definitions may reach
Safe but Useless result
Less Precise
Fixed Points
Safe
Least Fixed Point
Truth
Unsafe
MAY
No definitions can reach
Unsafe result


MUST
⊤
⊥
All definitions
may reach
Safe but
Useless result
Less
Precise
Fixed Points
Safe
Least
Fixed Point
Truth
Unsafe
MAY
No definitions
can reach
Unsafe result

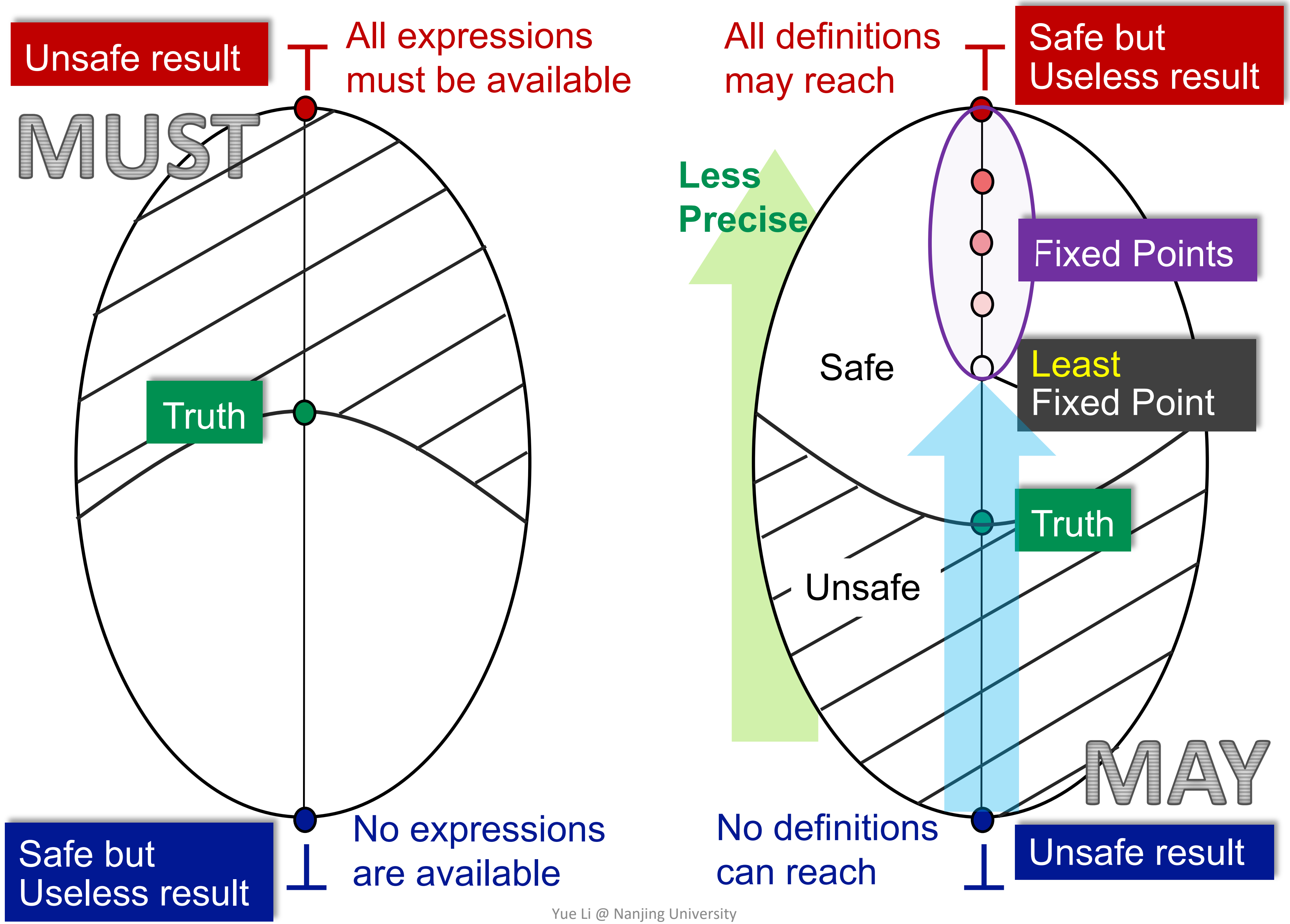

Unsafe result
All expressions
must be available
MUST
All definitions
may reach
Safe but
Useless result
Less
Precise
Fixed Points
Safe
Least
Fixed Point
Truth
Unsafe
MAY
No definitions
can reach
Unsafe result



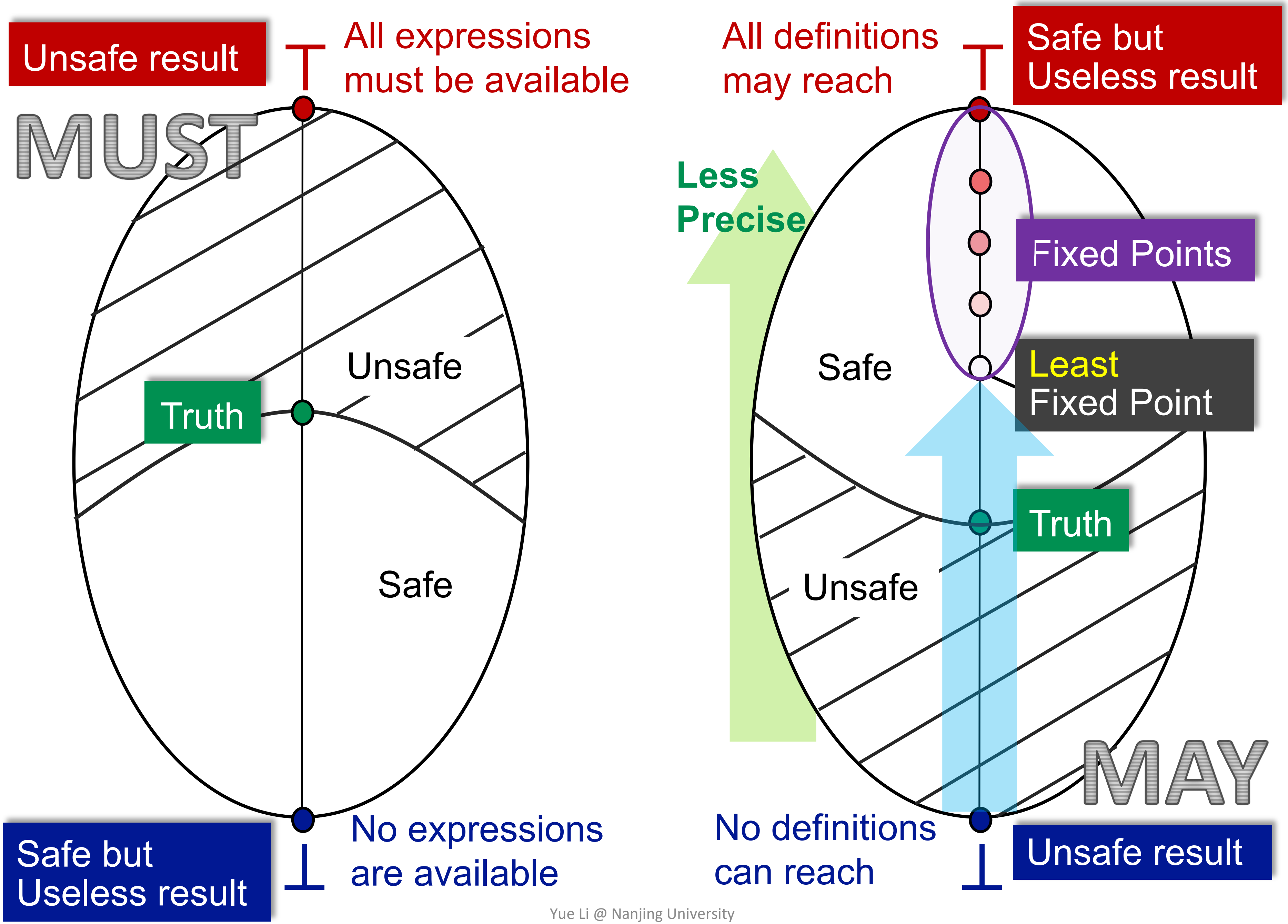
Unsafe result
All expressions
must be available
MUST
Safe but
Useless result
No expressions
are available
All definitions
may reach
Safe but
Useless result
Less
Precise
Fixed Points
Safe
Least
Fixed Point
Truth
Unsafe
MAY
No definitions
can reach
Unsafe result

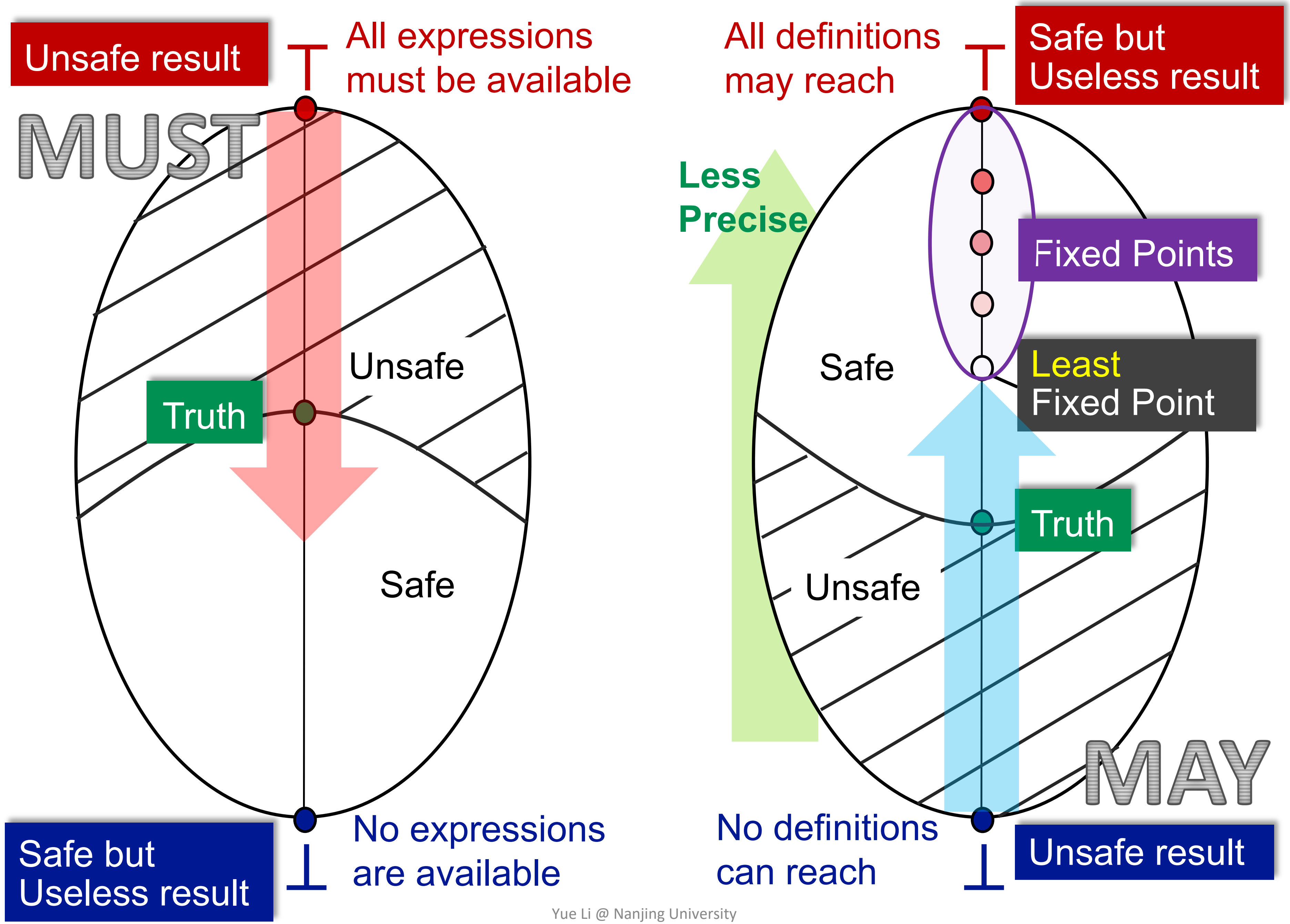


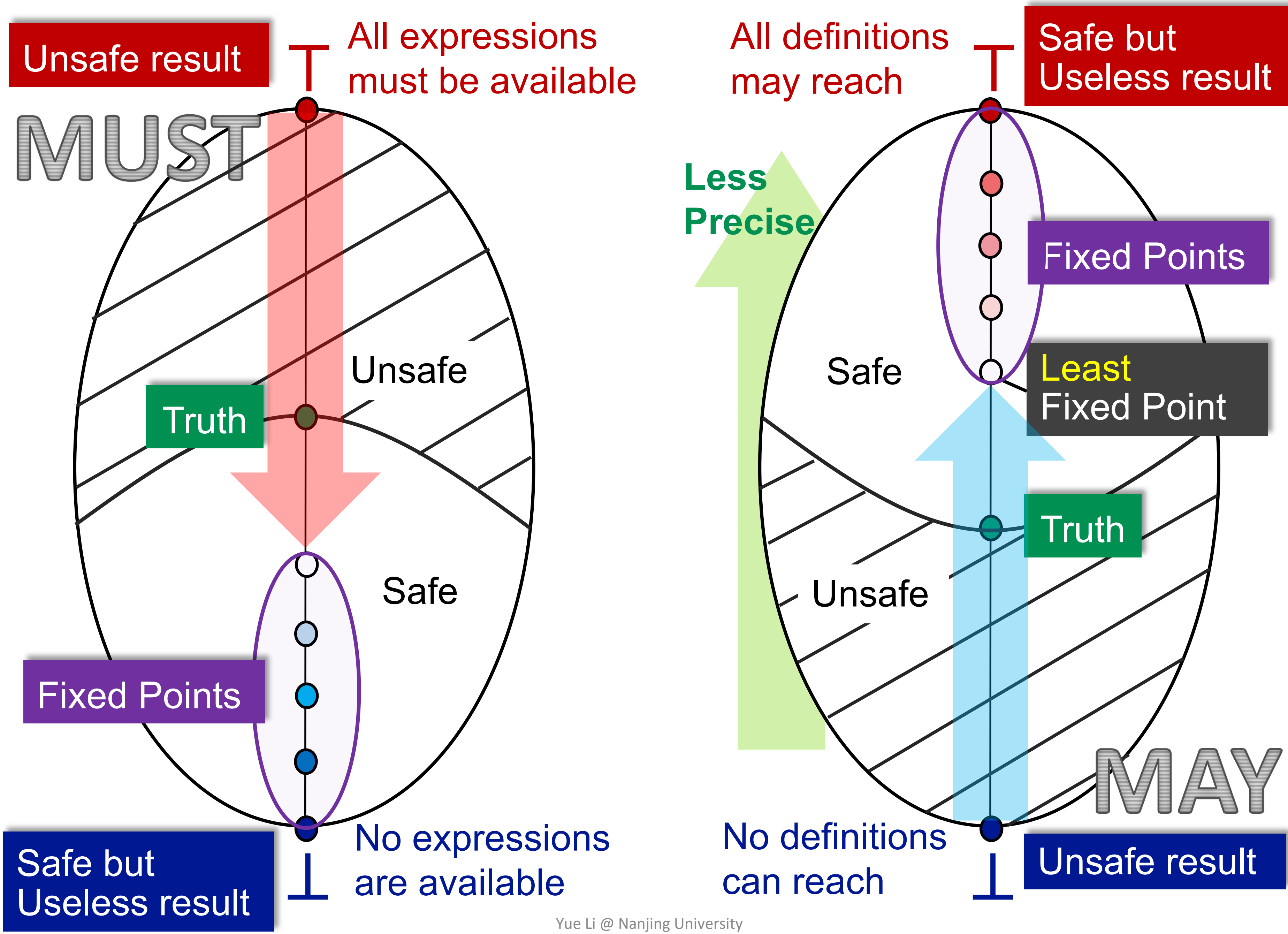
Unsafe result
All expressions
must be available
MUST
Truth
Safe but
Useless result
No expressions
are available
All definitions
may reach
Safe but
Useless result
Less
Precise
Fixed Points
Safe
Least
Fixed Point
Truth
Unsafe
MAY
No definitions
can reach
Unsafe result

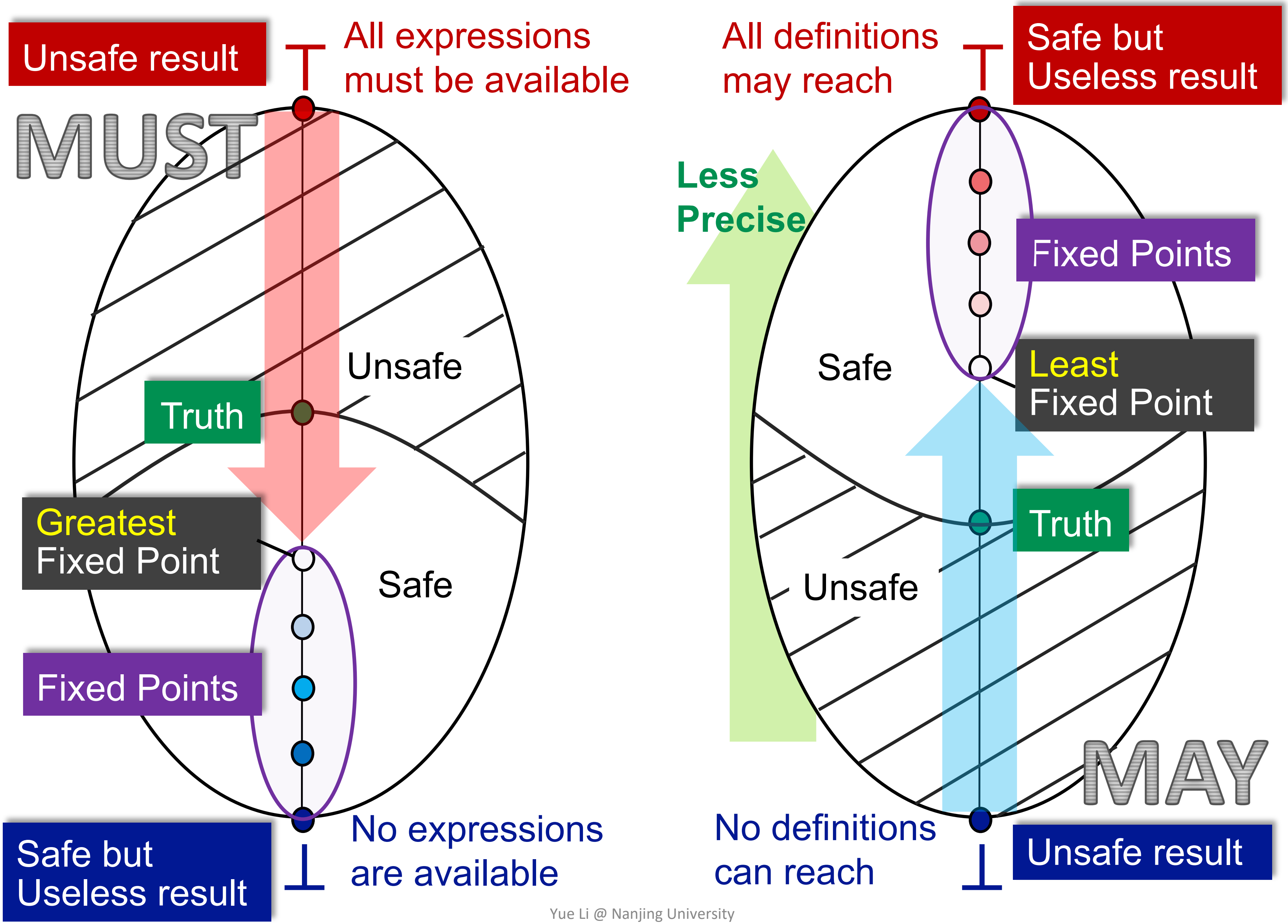


Unsafe result
All expressions
must be available
MUST
Truth
Safe but
Useless result
No expressions
are available
All definitions
may reach
Safe but
Useless result
Less
Precise
Fixed Points
Safe
Least
Fixed Point
Truth
Unsafe
MAY
No definitions
can reach
Unsafe result

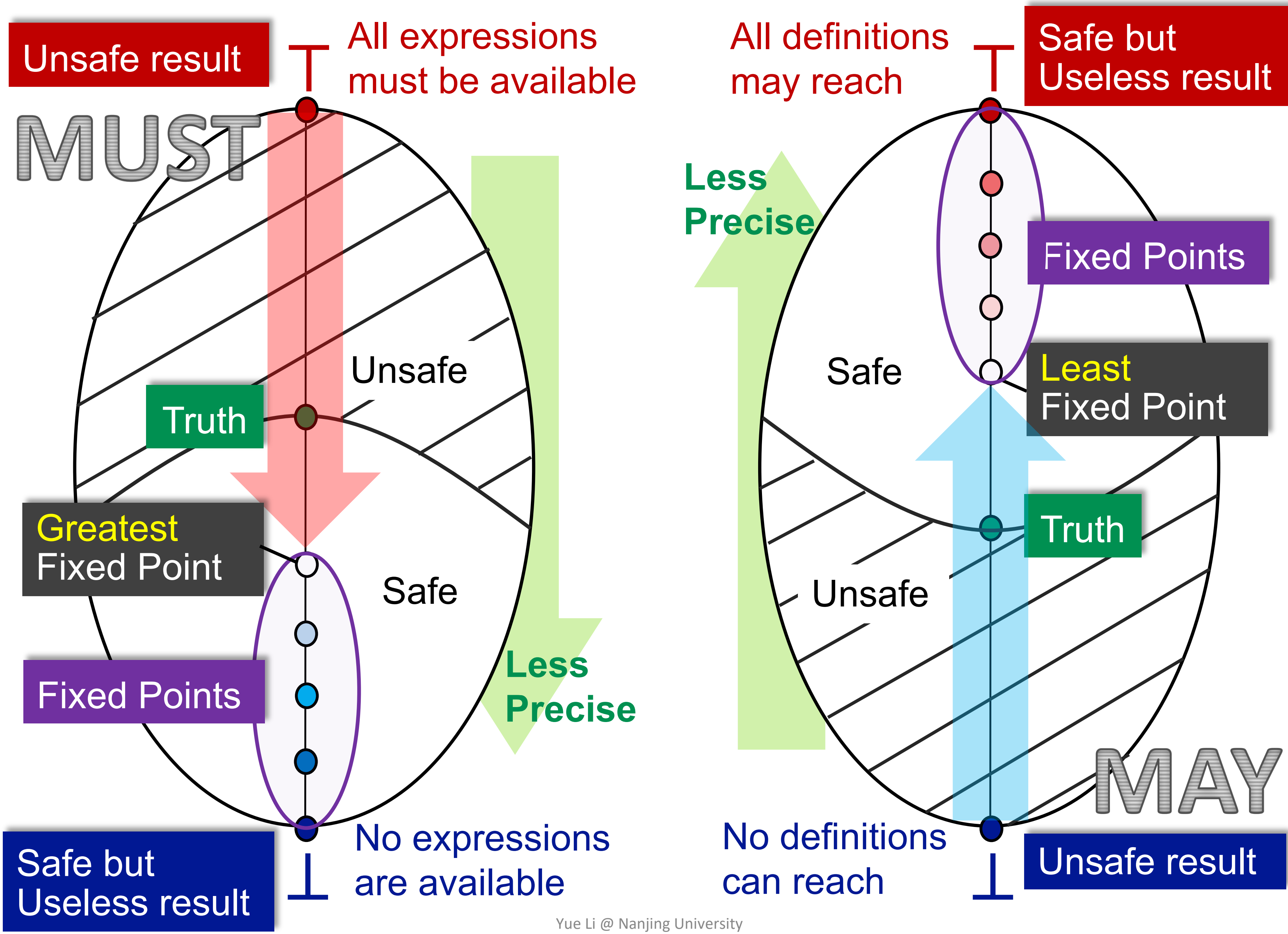

Unsafe result
All expressions
must be available
MUST
Unsafe
Truth
Safe
Safe but
Useless result
No expressions
are available
All definitions
may reach
Safe but
Useless result
Less
Precise
Fixed Points
Safe
Least
Fixed Point
Truth
Unsafe
MAY
No definitions
can reach
Unsafe result


Unsafe result
All expressions
must be available
MUST
Unsafe
Truth
Safe
Safe but
Useless result
No expressions
are available
All definitions
may reach
Safe but
Useless result
Less
Precise
Fixed Points
Safe
Least
Fixed Point
Truth
Unsafe
MAY
No definitions
can reach
Unsafe result

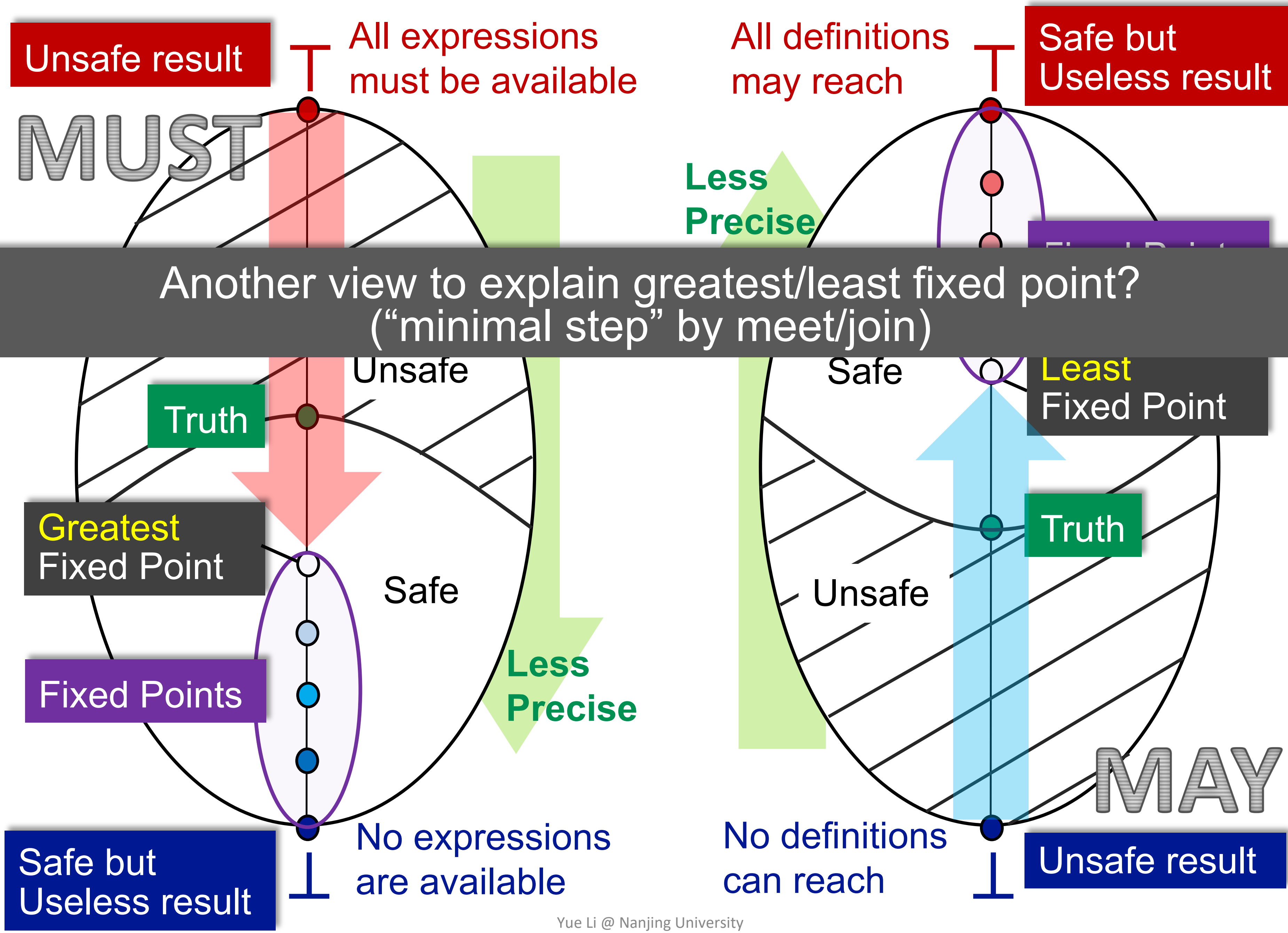

Unsafe result
⊤
All expressions
must be available
MUST
Unsafe
Truth
Safe
Fixed Points
Safe but
Useless result
⊥
No expressions
are available
All definitions
may reach
⊤
Safe but
Useless result
Less
Precise
Fixed Points
Safe
Least
Fixed Point
Truth
Unsafe
MAY
No definitions
can reach
⊥
Unsafe result



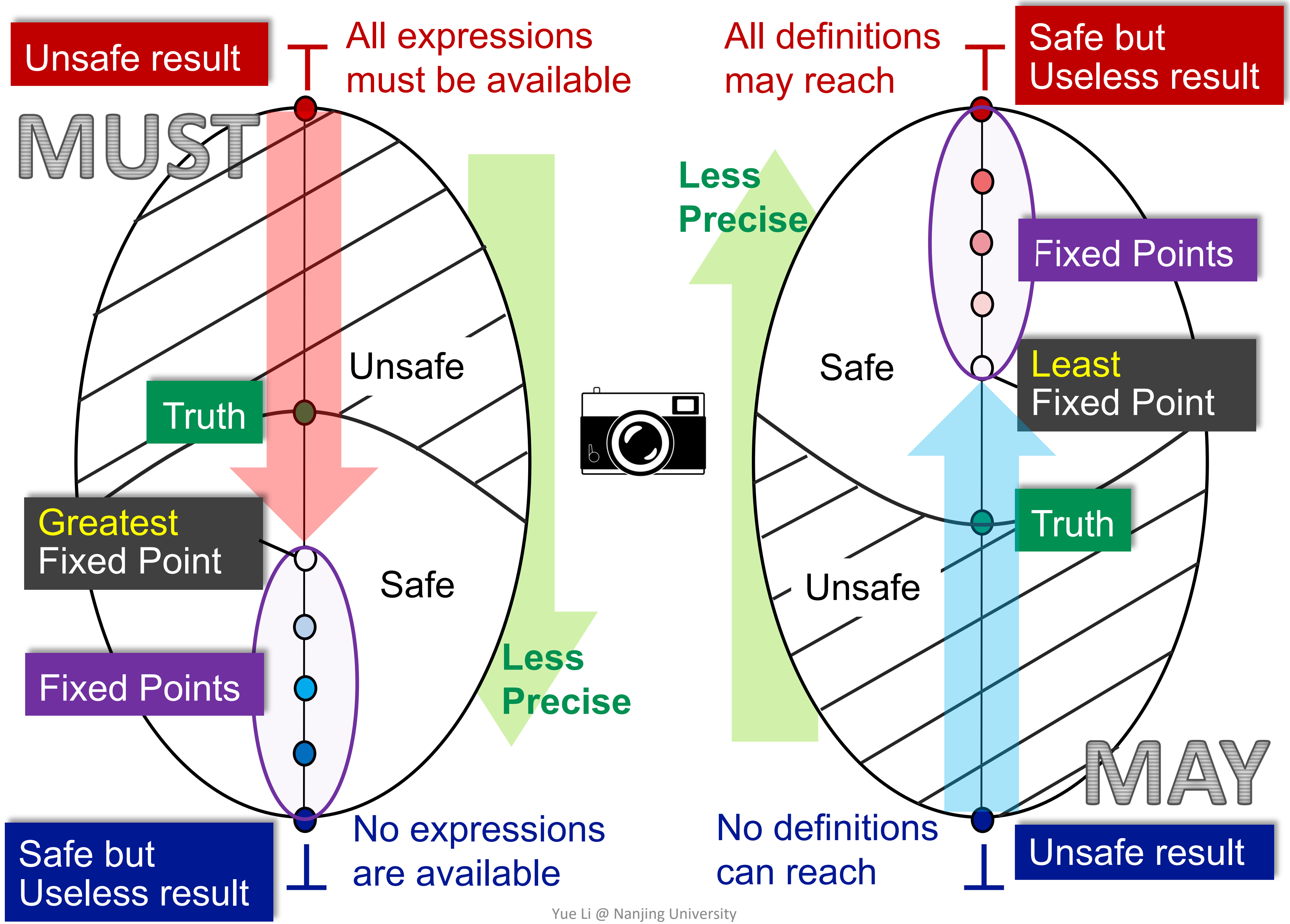
Unsafe result
All expressions
must be available
MUST
Unsafe
Truth
Greatest
Fixed Point
Safe
Fixed Points
Safe but
Useless result
No expressions
are available
All definitions
may reach
Safe but
Useless result
Less
Precise
Fixed Points
Safe
Least
Fixed Point
Truth
Unsafe
MAY
No definitions
can reach
Unsafe result



Unsafe result
All expressions
must be available
MUST
Unsafe
Truth
Greatest
Fixed Point
Safe
Fixed Points
Less
Precise
Safe but
Useless result
No expressions
are available
All definitions
may reach
Safe but
Useless result
Less
Precise
Fixed Points
Safe
Least
Fixed Point
Truth
Unsafe
MAY
No definitions
can reach
Unsafe result



Another view to explain greatest/least fixed point?
(“minimal step” by meet/join)

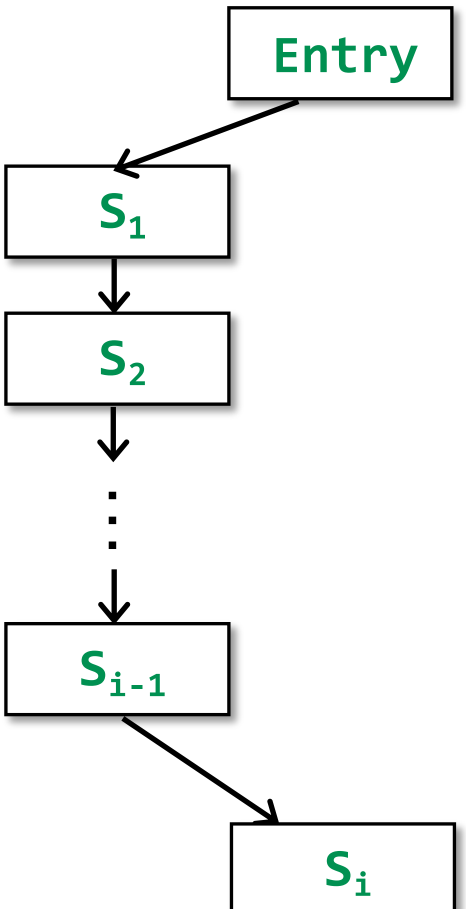

Unsafe result
All expressions
must be available
MUST
Unsafe
Truth
Greatest
Fixed Point
Safe
Fixed Points
Less
Precise
No expressions
are available
Safe but
Useless result
All definitions
may reach
Safe but
Useless result
Less
Precise
Fixed Points
Safe
Least
Fixed Point
Truth
Unsafe
MAY
No definitions
can reach
Unsafe result

How Precise Is Our Solution?

- Meet-Over-All-Paths Solution (MOP)

How Precise Is Our Solution?

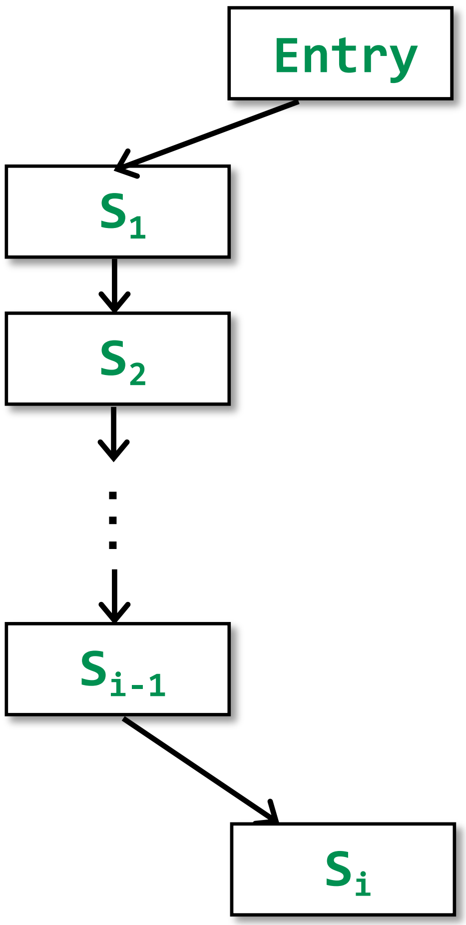
- Meet-Over-All-Paths Solution (MOP)



P = Entry → S_1 → S_2 → … → S_i

How Precise Is Our Solution?

- Meet-Over-All-Paths Solution (MOP)

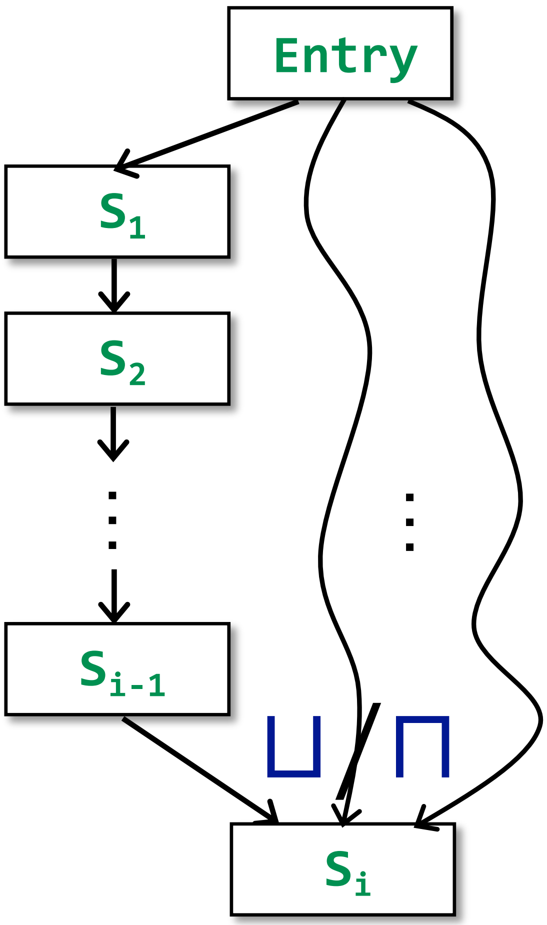


P = Entry → S_1 → S_2 → … → S_i

Transfer function F_P for a path P (from Entry to S_i) is a composition of transfer functions for all statements on that path: f_{S1}, f_{S2}, …, f_{Si-1}

How Precise Is Our Solution?

- Meet-Over-All-Paths Solution (MOP)



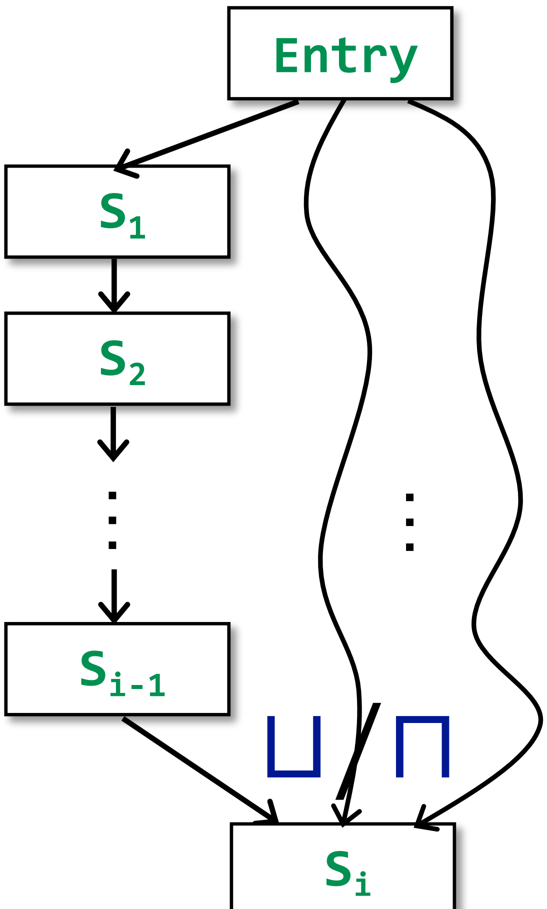
P = `Entry` → S_1 → S_2 → … → S_i

Transfer function F_P for a path P (from Entry to S_i) is a composition of transfer functions for all statements on that path: f_{S1}, f_{S2}, …, f_{Si-1}

$$MOP[S_i] = \underset{\text{A path } P \text{ from Entry to } S_i}{\sqcup / \sqcap} F_P(OUT[\text{Entry}])$$

How Precise Is Our Solution?

- Meet-Over-All-Paths Solution (MOP)



$\mathbf{P} = \mathtt{Entry} \rightarrow \mathtt{S_1} \rightarrow \mathtt{S_2} \rightarrow \ldots \rightarrow \mathtt{S_i}$

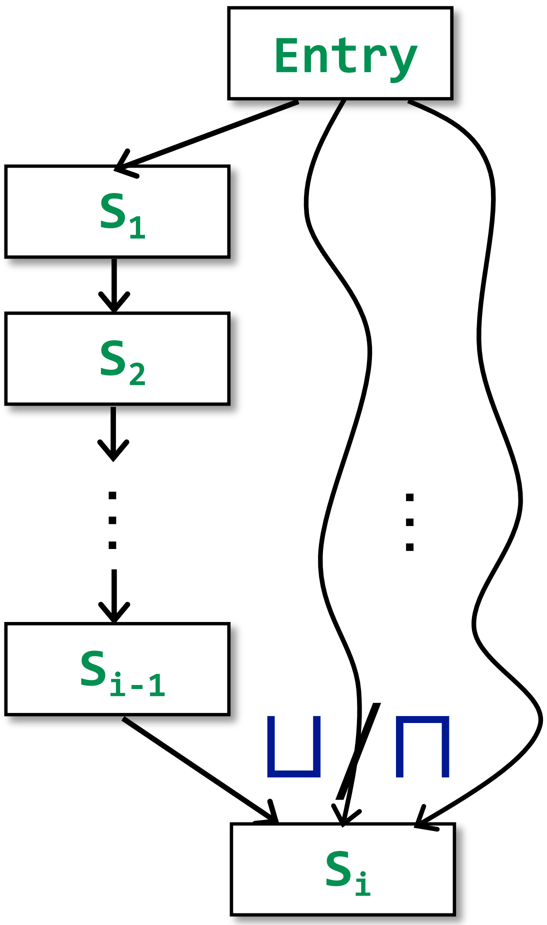
Transfer function F_P for a path P (from Entry to S_i) is a composition of transfer functions for all statements on that path: f_{S1}, f_{S2}, …, f_{Si-1}

$$MOP[S_i] = \underset{\text{A path } P \text{ from } Entry \text{ to } S_i}{\sqcup / \sqcap} F_P(OUT[Entry])$$

MOP computes the data-flow values at the end of each path and apply join / meet operator to these values to find their lub / glb

How Precise Is Our Solution?

- Meet-Over-All-Paths Solution (MOP)



$$P = \texttt{Entry} \rightarrow \texttt{S}_1 \rightarrow \texttt{S}_2 \rightarrow \ldots \rightarrow \texttt{S}_i$$

Transfer function F_P for a path P (from Entry to S_i) is a composition of transfer functions for all statements on that path: f_{S1}, f_{S2}, …, f_{Si-1}

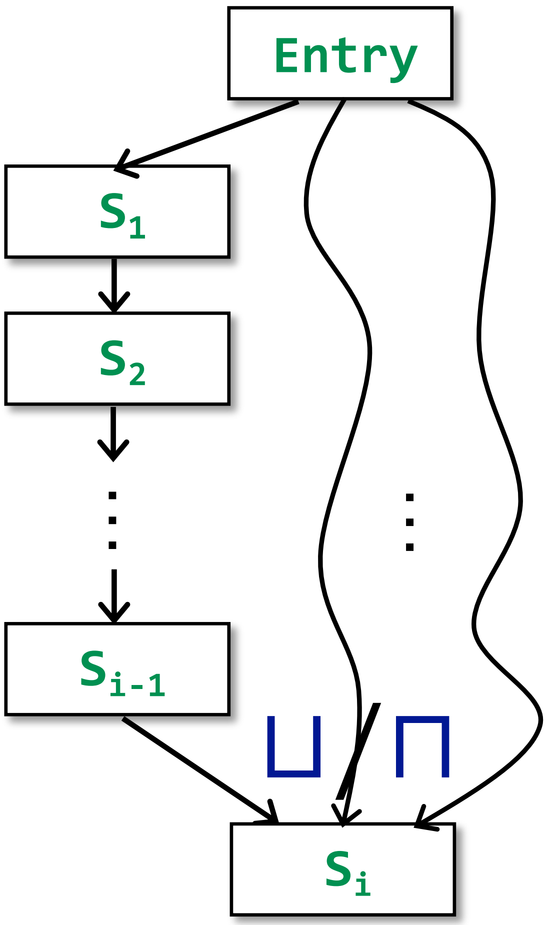
$$\mathrm{MOP}[\texttt{S}_i] = \underset{\text{A path } P \text{ from Entry to } S_i}{\bigsqcup / \bigsqcap} F_P(\mathrm{OUT}[\texttt{Entry}])$$

MOP computes the data-flow values at the end of each path and apply join / meet operator to these values to find their lub / glb

Some paths may be not executable ➔ **not fully precise**

How Precise Is Our Solution?

- Meet-Over-All-Paths Solution (MOP)



$P = \text{Entry} \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_i$

Transfer function F_P for a path P (from Entry to S_i) is a composition of transfer functions for all statements on that path: f_{S1}, f_{S2}, …, f_{Si-1}

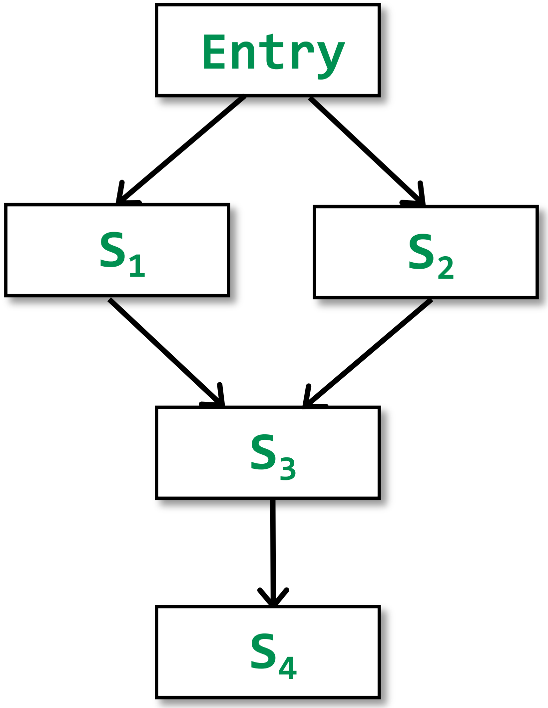
$$MOP[S_i] = \underset{\text{A path } P \text{ from Entry to } S_i}{\sqcup / \sqcap} F_P(OUT[\text{Entry}])$$

MOP computes the data-flow values at the end of each path and apply join / meet operator to these values to find their lub / glb

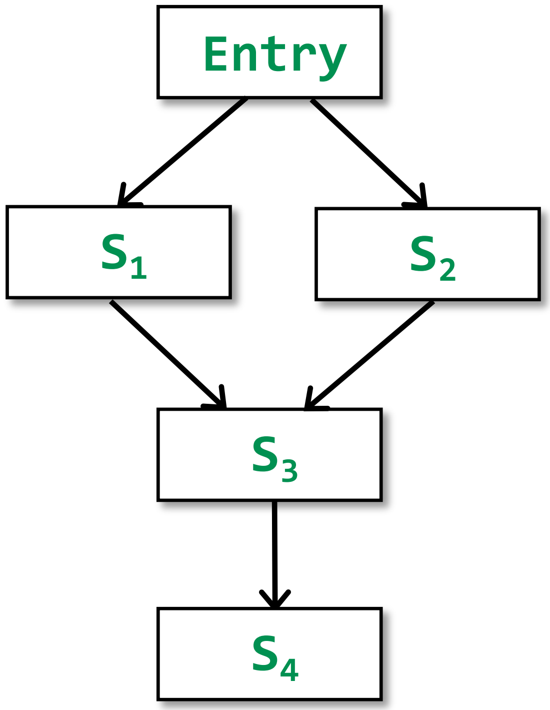
Some paths may be not executable → **not fully precise**

Unbounded, and not enumerable → **impractical**

Ours (Iterative Algorithm) vs. MOP

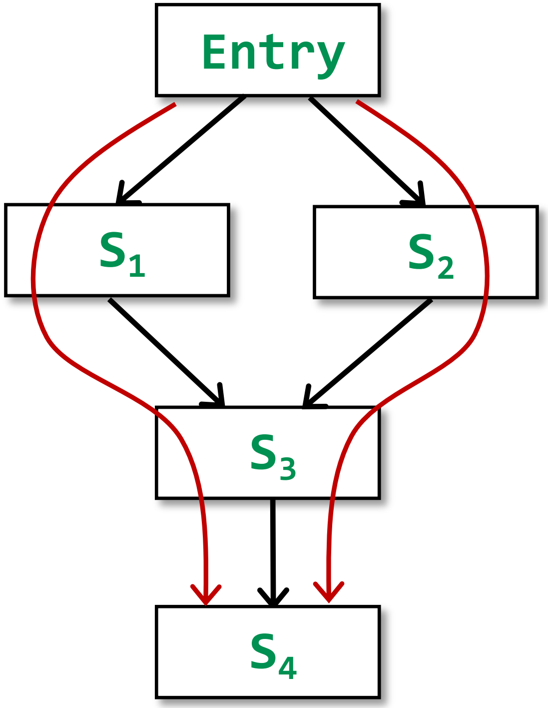


Ours (Iterative Algorithm) vs. MOP



$$\mathrm{IN}[S_4] = f_{S_3}(f_{S_1}(\mathrm{OUT}[\mathtt{Entry}]) \sqcup f_{S_2}(\mathrm{OUT}[\mathtt{Entry}]))$$

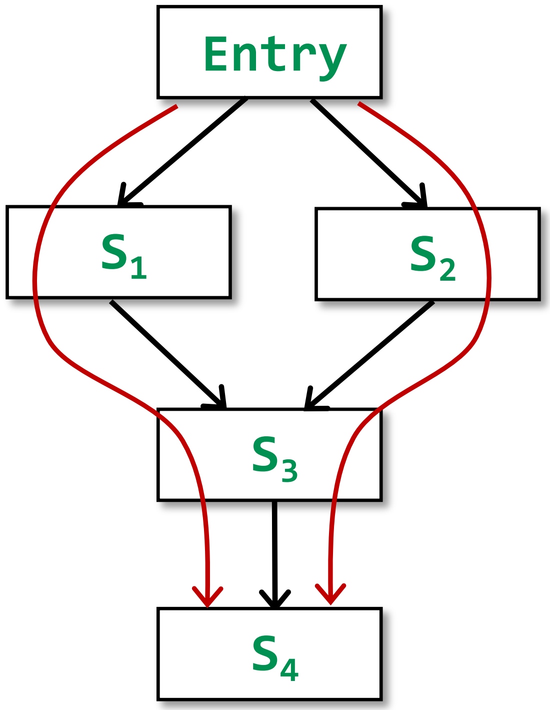
Ours (Iterative Algorithm) vs. MOP



$$\mathrm{IN}[s_4] = f_{s_3}(f_{s_1}(\mathrm{OUT}[\texttt{Entry}]) \sqcup f_{s_2}(\mathrm{OUT}[\texttt{Entry}]))$$

$$\mathrm{MOP}[s_4] = f_{s_3}(f_{s_1}(\mathrm{OUT}[\texttt{Entry}])) \sqcup f_{s_3}(f_{s_2}(\mathrm{OUT}[\texttt{Entry}]))$$

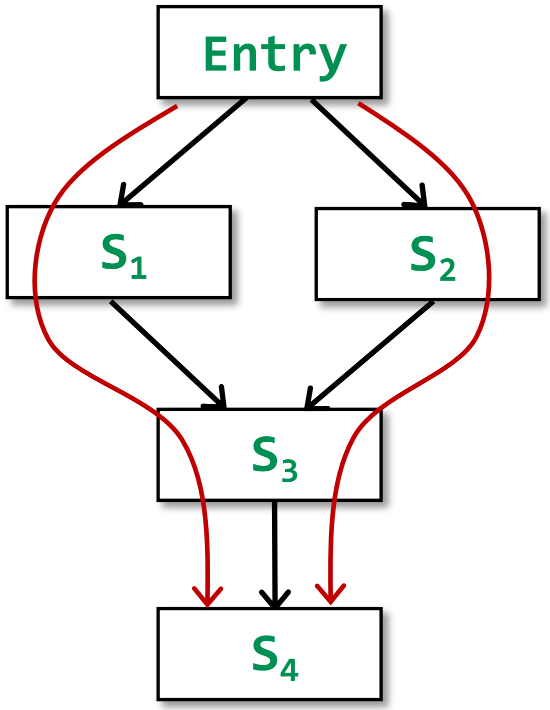
Ours (Iterative Algorithm) vs. MOP



$$\text{IN}[S_4] = f_{s_3}(f_{s_1}(\text{OUT}[\texttt{Entry}]) \sqcup f_{s_2}(\text{OUT}[\texttt{Entry}]))$$

$$\text{MOP}[S_4] = f_{s_3}(f_{s_1}(\text{OUT}[\texttt{Entry}])) \sqcup f_{s_3}(f_{s_2}(\text{OUT}[\texttt{Entry}]))$$

Ours (Iterative Algorithm) vs. MOP



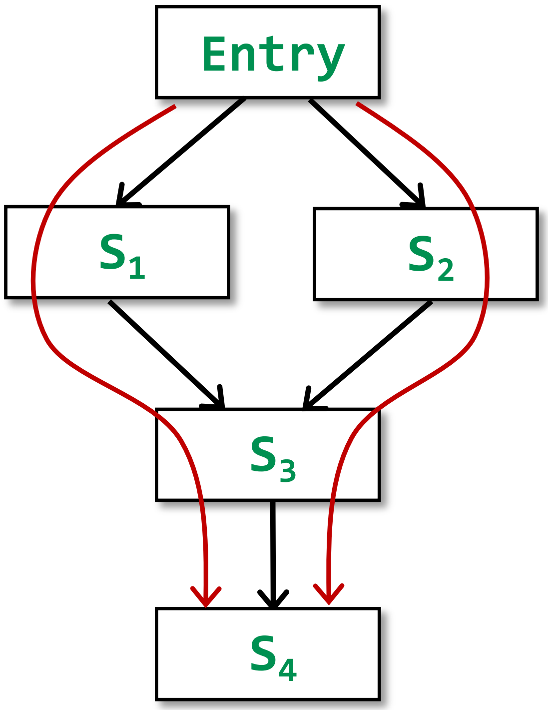
Ours = F(x ⊔ y)

MOP = F(x) ⊔ F(y)

$$\text{IN}[s_4] = f_{s_3}(f_{s_1}(\text{OUT}[\texttt{Entry}]) \sqcup f_{s_2}(\text{OUT}[\texttt{Entry}]))$$

$$\text{MOP}[s_4] = f_{s_3}(f_{s_1}(\text{OUT}[\texttt{Entry}])) \sqcup f_{s_3}(f_{s_2}(\text{OUT}[\texttt{Entry}]))$$

Ours (Iterative Algorithm) vs. MOP



$$\text{Ours} = F(x \sqcup y)$$
$$\text{MOP} = F(x) \sqcup F(y)$$

?

$$\text{IN}[s_4] = f_{s_3}(f_{s_1}(\text{OUT}[\texttt{Entry}]) \sqcup f_{s_2}(\text{OUT}[\texttt{Entry}]))$$

$$\text{MOP}[s_4] = f_{s_3}(f_{s_1}(\text{OUT}[\texttt{Entry}])) \sqcup f_{s_3}(f_{s_2}(\text{OUT}[\texttt{Entry}]))$$

Ours (Iterative Algorithm) vs. MOP

Ours = $F(x \sqcup y)$

MOP = $F(x) \sqcup F(y)$

Ours (Iterative Algorithm) vs. MOP

Ours = $F(x \sqcup y)$

MOP = $F(x) \sqcup F(y)$

By definition of lub $\sqcup$, we have

$$x \sqsubseteq x \sqcup y \text{ and } y \sqsubseteq x \sqcup y$$

Ours (Iterative Algorithm) vs. MOP

Ours = $F(x \sqcup y)$

MOP = $F(x) \sqcup F(y)$

By definition of lub $\sqcup$, we have

$$x \sqsubseteq x \sqcup y \text{ and } y \sqsubseteq x \sqcup y$$

As transfer function F is **monotonic**, we have

Ours (Iterative Algorithm) vs. MOP

Ours = $F(x \sqcup y)$

MOP = $F(x) \sqcup F(y)$

By definition of lub $\sqcup$, we have

$$x \sqsubseteq x \sqcup y \text{ and } y \sqsubseteq x \sqcup y$$

As transfer function F is **monotonic**, we have

$$F(x) \sqsubseteq F(x \sqcup y) \text{ and } F(y) \sqsubseteq F(x \sqcup y)$$

Ours (Iterative Algorithm) vs. MOP

Ours = $F(x \sqcup y)$

MOP = $F(x) \sqcup F(y)$

By definition of lub $\sqcup$, we have

$$x \sqsubseteq x \sqcup y \text{ and } y \sqsubseteq x \sqcup y$$

As transfer function F is **monotonic**, we have

$$F(x) \sqsubseteq F(x \sqcup y) \text{ and } F(y) \sqsubseteq F(x \sqcup y)$$

That means $F(x \sqcup y)$ is an upper bound of $F(x)$ and $F(y)$

Ours (Iterative Algorithm) vs. MOP

Ours = $F(x \sqcup y)$

MOP = $F(x) \sqcup F(y)$

By definition of lub $\sqcup$, we have

$$x \sqsubseteq x \sqcup y \text{ and } y \sqsubseteq x \sqcup y$$

As transfer function F is **monotonic**, we have

$$F(x) \sqsubseteq F(x \sqcup y) \text{ and } F(y) \sqsubseteq F(x \sqcup y)$$

That means $F(x \sqcup y)$ is an upper bound of $F(x)$ and $F(y)$

As $F(x) \sqcup F(y)$ is the lub of $F(x)$ and $F(y)$, we have

Ours (Iterative Algorithm) vs. MOP

Ours = $F(x \sqcup y)$

MOP = $F(x) \sqcup F(y)$

By definition of lub $\sqcup$, we have

$$x \sqsubseteq x \sqcup y \text{ and } y \sqsubseteq x \sqcup y$$

As transfer function F is **monotonic**, we have

$$F(x) \sqsubseteq F(x \sqcup y) \text{ and } F(y) \sqsubseteq F(x \sqcup y)$$

That means $F(x \sqcup y)$ is an upper bound of $F(x)$ and $F(y)$

As $F(x) \sqcup F(y)$ is the lub of $F(x)$ and $F(y)$, we have

$$F(x) \sqcup F(y) \sqsubseteq F(x \sqcup y)$$

Ours (Iterative Algorithm) vs. MOP

Ours = $F(x \sqcup y)$

MOP = $F(x) \sqcup F(y)$

By definition of lub $\sqcup$, we have

$$x \sqsubseteq x \sqcup y \text{ and } y \sqsubseteq x \sqcup y$$

As transfer function F is **monotonic**, we have

$$F(x) \sqsubseteq F(x \sqcup y) \text{ and } F(y) \sqsubseteq F(x \sqcup y)$$

That means $F(x \sqcup y)$ is an upper bound of $F(x)$ and $F(y)$

As $F(x) \sqcup F(y)$ is the lub of $F(x)$ and $F(y)$, we have

$$F(x) \sqcup F(y) \sqsubseteq F(x \sqcup y)$$

$$\text{MOP} \sqsubseteq \text{Ours}$$

Ours (Iterative Algorithm) vs. MOP

Ours = $F(x \sqcup y)$

MOP = $F(x) \sqcup F(y)$

By definition of lub $\sqcup$, we have

$$x \sqsubseteq x \sqcup y \text{ and } y \sqsubseteq x \sqcup y$$

As transfer function F is **monotonic**, we have

$$F(x) \sqsubseteq F(x \sqcup y) \text{ and } F(y) \sqsubseteq F(x \sqcup y)$$

That means $F(x \sqcup y)$ is an upper bound of $F(x)$ and $F(y)$

As $F(x) \sqcup F(y)$ is the lub of $F(x)$ and $F(y)$, we have

$$F(x) \sqcup F(y) \sqsubseteq F(x \sqcup y)$$

$$\text{MOP} \sqsubseteq \text{Ours}$$

(Ours is less precise than MOP)

Ours (Iterative Algorithm) vs. MOP

Ours = $F(x \sqcup y)$

MOP = $F(x) \sqcup F(y)$

By definition of lub $\sqcup$, we have

$$x \sqsubseteq x \sqcup y \text{ and } y \sqsubseteq x \sqcup y$$

As transfer function F is **monotonic**, we have

$$F(x) \sqsubseteq F(x \sqcup y) \text{ and } F(y) \sqsubseteq F(x \sqcup y)$$

That means $F(x \sqcup y)$ is an upper bound of $F(x)$ and $F(y)$

As $F(x) \sqcup F(y)$ is the lub of $F(x)$ and $F(y)$, we have

$$F(x) \sqcup F(y) \sqsubseteq F(x \sqcup y)$$

$$\text{MOP} \sqsubseteq \text{Ours}$$

(Ours is less precise than MOP)

When F is **distributive**, i.e.,

$$F(x \sqcup y) = F(x) \sqcup F(y)$$

Ours (Iterative Algorithm) vs. MOP

Ours = $F(x \sqcup y)$

MOP = $F(x) \sqcup F(y)$

By definition of lub $\sqcup$, we have

$$x \sqsubseteq x \sqcup y \text{ and } y \sqsubseteq x \sqcup y$$

As transfer function F is **monotonic**, we have

$$F(x) \sqsubseteq F(x \sqcup y) \text{ and } F(y) \sqsubseteq F(x \sqcup y)$$

That means $F(x \sqcup y)$ is an upper bound of $F(x)$ and $F(y)$

As $F(x) \sqcup F(y)$ is the lub of $F(x)$ and $F(y)$, we have

$$F(x) \sqcup F(y) \sqsubseteq F(x \sqcup y)$$

$$\text{MOP} \sqsubseteq \text{Ours}$$

(Ours is less precise than MOP)

When F is **distributive**, i.e.,

$$F(x \sqcup y) = F(x) \sqcup F(y)$$

$$\text{MOP} = \text{Ours}$$

Ours (Iterative Algorithm) vs. MOP

Ours = $F(x \sqcup y)$

MOP = $F(x) \sqcup F(y)$

By definition of lub $\sqcup$, we have

$$x \sqsubseteq x \sqcup y \text{ and } y \sqsubseteq x \sqcup y$$

As transfer function F is **monotonic**, we have

$$F(x) \sqsubseteq F(x \sqcup y) \text{ and } F(y) \sqsubseteq F(x \sqcup y)$$

That means $F(x \sqcup y)$ is an upper bound of $F(x)$ and $F(y)$

As $F(x) \sqcup F(y)$ is the lub of $F(x)$ and $F(y)$, we have

$$F(x) \sqcup F(y) \sqsubseteq F(x \sqcup y)$$

$$\text{MOP} \sqsubseteq \text{Ours}$$

(Ours is less precise than MOP)

When F is **distributive**, i.e.,

$$F(x \sqcup y) = F(x) \sqcup F(y)$$

$$\text{MOP} = \text{Ours}$$

(Ours is as precise as MOP)

Ours (Iterative Algorithm) vs. MOP

Ours = $F(x \sqcup y)$

MOP = $F(x) \sqcup F(y)$

By definition of lub $\sqcup$, we have

$$x \sqsubseteq x \sqcup y \text{ and } y \sqsubseteq x \sqcup y$$

As transfer function **F is monotonic**, we have

$$F(x) \sqsubseteq F(x \sqcup y) \text{ and } F(y) \sqsubseteq F(x \sqcup y)$$

That means $F(x \sqcup y)$ is an upper bound of $F(x)$ and $F(y)$

As $F(x) \sqcup F(y)$ is the lub of $F(x)$ and $F(y)$, we have

F ... $F(x \sqcup y)$

M...

(Ours is less precise tha...

Bit-vector or Gen/Kill problems (set union /intersection for join/meet) are distributive

When **F is distributive**, i.e.,

$$F(x \sqcup y) = F(x) \sqcup F(y)$$

$$\text{MOP} = \text{Ours}$$

(Ours is as precise as MOP)

Ours (Iterative Algorithm) vs. MOP

Ours = $F(x \sqcup y)$

MOP = $F(x) \sqcup F(y)$

By definition of lub $\sqcup$, we have

$$x \sqsubseteq x \sqcup y \text{ and } y \sqsubseteq x \sqcup y$$

As transfer function **F is monotonic**, we have

$$F(x) \sqsubseteq F(x \sqcup y) \text{ and } F(y) \sqsubseteq F(x \sqcup y)$$

That means $F(x \sqcup y)$ is an upper bound of $F(x)$ and $F(y)$

As $F(x) \sqcup F(y)$ is the lub of $F(x)$ and $F(y)$, we have

F ... $\sqsubseteq F(x \sqcup y)$

M...

(Ours is less precise th...

Bit-vector or Gen/Kill prob...
/intersection for join/m... ...set union
...re distributive

But some analyses are not distributive

When F is **distributiv...**

$$F(x \sqcup y) = F(x) \sqcup F(y)$$

MOP = Ours

(Ours is as precise as MOP)

Constant Propagation

Given a variable x at program point p, determine whether x is guaranteed to hold a constant value at p.

Constant Propagation

Given a variable x at program point p, determine whether x is guaranteed to hold a constant value at p.

- The OUT of each node in CFG, includes a set of pairs (x, v) where x is a variable and v is the value held by x after that node

Constant Propagation

Given a variable x at program point p, determine whether x is guaranteed to hold a constant value at p.

- The OUT of each node in CFG, includes a set of pairs (x, v) where x is a variable and v is the value held by x after that node

A data flow analysis framework (D, L, F) consists of:

- **D**: a direction of data flow: forwards or backwards
- **L**: a lattice including domain of the values V and a meet $\sqcap$ or join $\sqcup$ operator
- **F**: a family of transfer functions from V to V

Constant Propagation

Given a variable x at program point p, determine whether x is guaranteed to hold a constant value at p.

- The OUT of each node in CFG, includes a set of pairs (x, v) where x is a variable and v is the value held by x after that node

A data flow analysis framework (D, L, F) consists of:

- **D**: a direction of data flow: <u>forwards</u> or backwards
- **L**: a lattice including domain of the values V and a meet ⊓ or join ⊔ operator
- **F**: a family of transfer functions from V to V

Constant Propagation

Given a variable x at program point p, determine whether x is guaranteed to hold a constant value at p.

- The OUT of each node in CFG, includes a set of pairs (x, v) where x is a variable and v is the value held by x after that node

A data flow analysis framework (D, L, F) consists of:

- **D**: a direction of data flow: <u>forwards</u> or backwards
- **L**: a lattice including domain of the values V and a meet ⊓ or join ⊔ operator
- **F**: a family of transfer functions from V to V

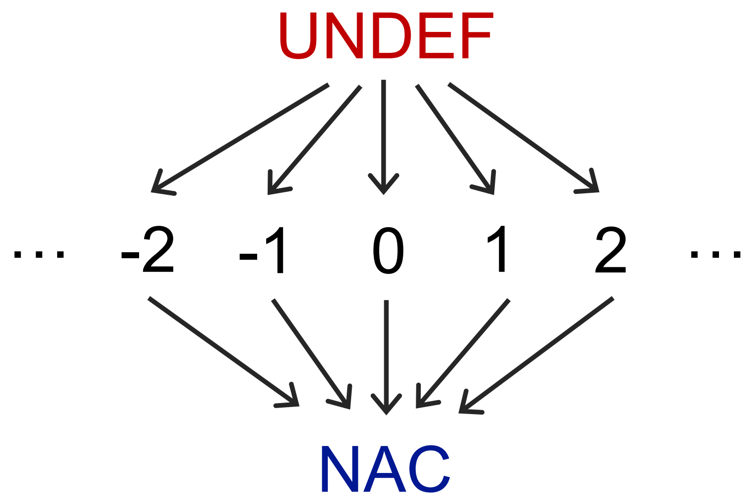
Constant Propagation – Lattice

- Domain of the values V

- Meet Operator ⊓

Constant Propagation – Lattice

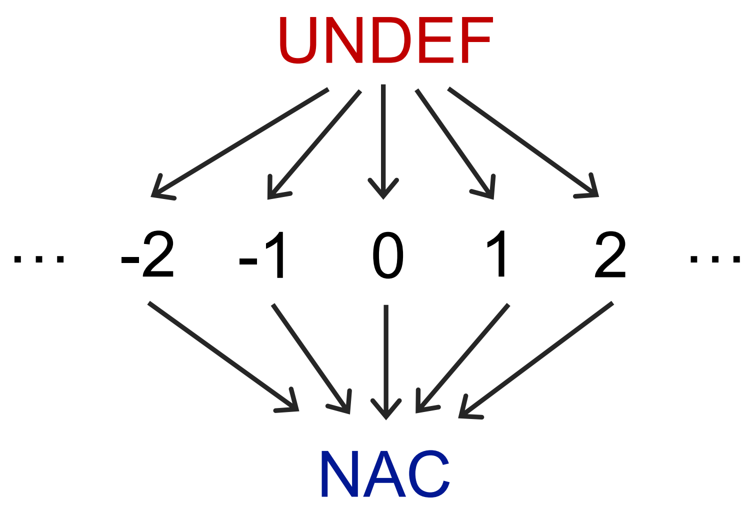
- Domain of the values V



- Meet Operator ⊓

Constant Propagation – Lattice

- Domain of the values V

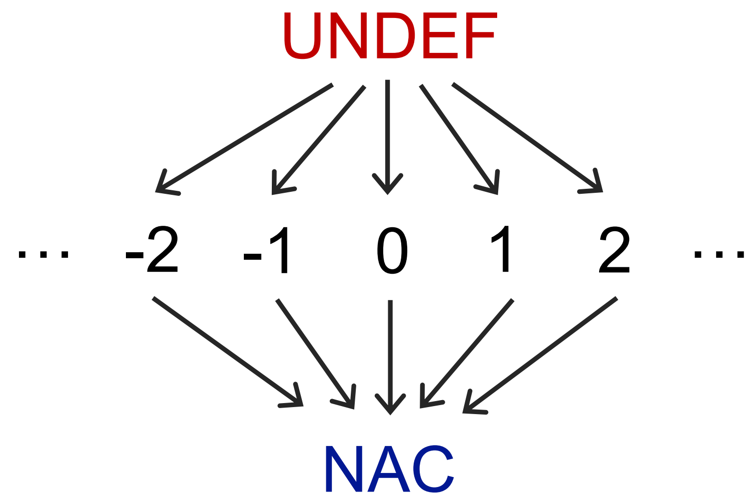


- Meet Operator ⊓

NAC ⊓ v = NAC

Constant Propagation – Lattice

- Domain of the values V



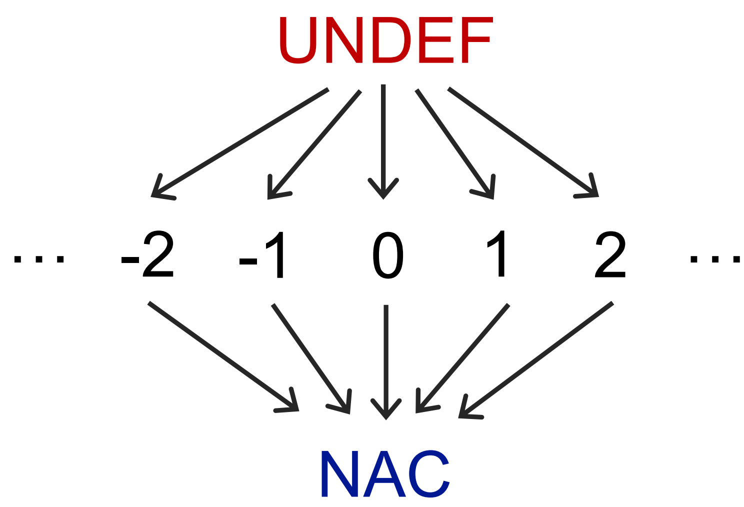
- Meet Operator ⊓

NAC ⊓ v = NAC

UNDEF ⊓ v = v

Constant Propagation – Lattice

- Domain of the values V



- Meet Operator ⊓

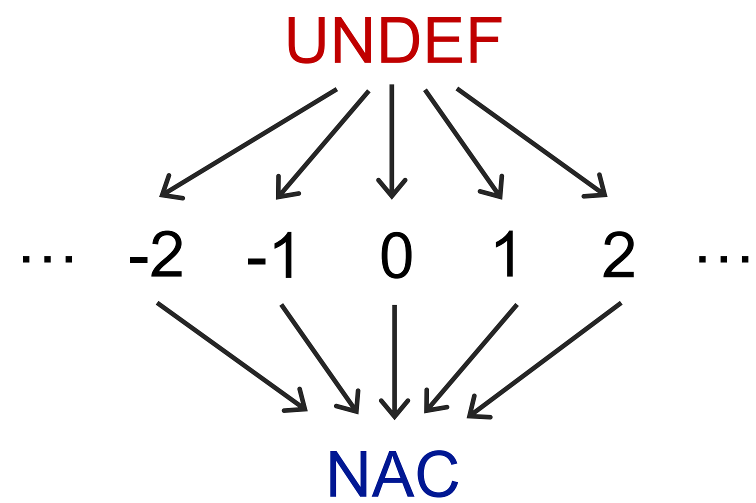
NAC ⊓ v = NAC

UNDEF ⊓ v = v

Uninitialized variables are not the focus in our constant propagation analysis

Constant Propagation – Lattice

- Domain of the values V



- Meet Operator ⊓

NAC ⊓ v = NAC

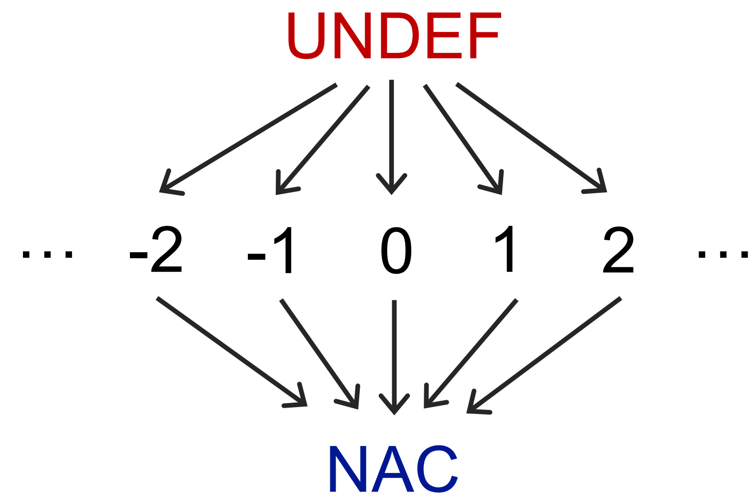
UNDEF ⊓ v = v

Uninitialized variables are not the focus in our constant propagation analysis

c ⊓ v = ?

Constant Propagation – Lattice

- Domain of the values V



- Meet Operator ⊓

NAC ⊓ v = NAC

UNDEF ⊓ v = v

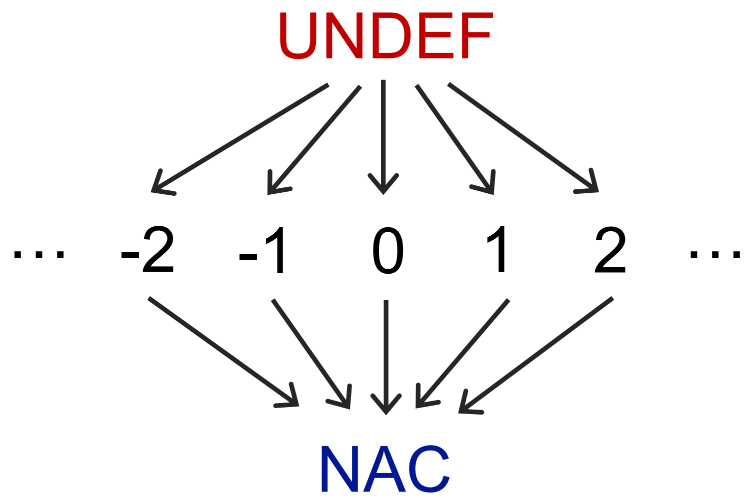
Uninitialized variables are not the focus in our constant propagation analysis

c ⊓ v = ?

- $c \sqcap c = c$
- $c_1 \sqcap c_2$ = NAC

Constant Propagation – Lattice

- Domain of the values V



- Meet Operator ⊓

NAC ⊓ v = NAC

UNDEF ⊓ v = v

Uninitialized variables are not the focus in our constant propagation analysis

c ⊓ v = ?

- $c \sqcap c = c$
- $c_1 \sqcap c_2$ = NAC

At each path confluence PC, we should apply “meet” for all variables in the incoming data-flow values at that PC

Constant Propagation – Transfer Function

Given a statement s: x = …, we define its transfer function F as

$$\boxed{F: OUT[s] = gen \cup (IN[s] - \{(x, \_)\})}$$

Constant Propagation – Transfer Function

Given a statement s: x = …, we define its transfer function F as

$$\boxed{F: OUT[s] = gen \cup (IN[s] - \{(x, \_)\})}$$

(we use val(x) to denote the lattice value that variable x holds)

Constant Propagation – Transfer Function

Given a statement s: x = …, we define its transfer function F as

$$\boxed{F: OUT[s] = gen \cup (IN[s] - \{(x, \_)\})}$$

(we use val(x) to denote the lattice value that variable x holds)

- s: x = c; // c is a constant

Constant Propagation – Transfer Function

Given a statement s: x = …, we define its transfer function F as

$$\boxed{F: OUT[s] = gen \cup (IN[s] - \{(x, \_)\})}$$

(we use val(x) to denote the lattice value that variable x holds)

- s: x = c; // c is a constant   gen = {(x, c)}

Constant Propagation – Transfer Function

Given a statement s: x = …, we define its transfer function F as

$$\boxed{F\text{: OUT[s]} = \text{gen} \cup (\text{IN[s]} - \{(x, \_)\})}$$

(we use val(x) to denote the lattice value that variable x holds)

- s: x = c; // c is a constant gen = {(x, c)}
- s: x = y;

Constant Propagation – Transfer Function

Given a statement s: x = …, we define its transfer function F as

$$\boxed{F: OUT[s] = gen \cup (IN[s] - \{(x, \_)\})}$$

(we use val(x) to denote the lattice value that variable x holds)

- s: x = c; // c is a constant gen = {(x, c)}
- s: x = y; gen = {(x, val(y))}

Constant Propagation – Transfer Function

Given a statement s: x = …, we define its transfer function F as

$$\boxed{F\text{: OUT[s]} = \text{gen} \cup (\text{IN[s]} - \{(x, \_)\})}$$

(we use val(x) to denote the lattice value that variable x holds)

- s: x = c; // c is a constant   gen = {(x, c)}
- s: x = y;   gen = {(x, val(y))}
- s: x = y *op* z;   gen = {(x, f(y,z))}

Constant Propagation – Transfer Function

Given a statement s: x = …, we define its transfer function F as

$$\boxed{F: OUT[s] = gen \cup (IN[s] - \{(x, \_)\})}$$

(we use val(x) to denote the lattice value that variable x holds)

- s: x = c; // c is a constant   gen = {(x, c)}
- s: x = y;   gen = {(x, val(y))}
- s: x = y *op* z;   gen = {(x, f(y,z))}

$$f(y,z) = \begin{cases} val(y)\ op\ val(z) & \text{// if val(y) and val(z) are constants} \\ NAC & \text{// if val(y) or val(z) is NAC} \\ UNDEF & \text{// otherwise} \end{cases}$$

Constant Propagation – Transfer Function

Given a statement s: x = …, we define its transfer function F as

$$\boxed{F: OUT[s] = gen \cup (IN[s] - \{(x, \_)\})}$$

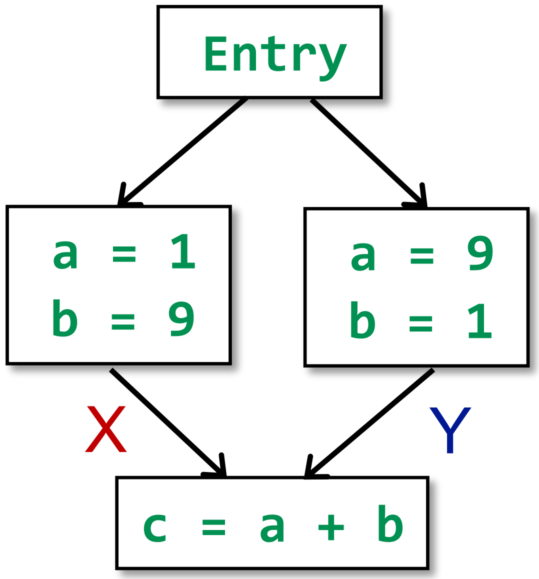
(we use val(x) to denote the lattice value that variable x holds)

- s: x = c; // c is a constant   gen = {(x, c)}
- s: x = y;   gen = {(x, val(y))}
- s: x = y *op* z;   gen = {(x, f(y,z))}

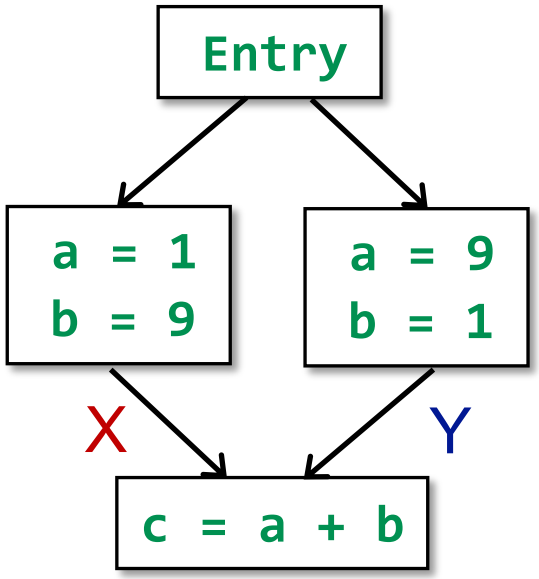
$$f(y,z) = \begin{cases} val(y)\ op\ val(z) & \text{// if val(y) and val(z) are constants} \\ NAC & \text{// if val(y) or val(z) is NAC} \\ UNDEF & \text{// otherwise} \end{cases}$$

(if s is not an assignment statement, F is the identity function)

Constant Propagation – Nondistributivity



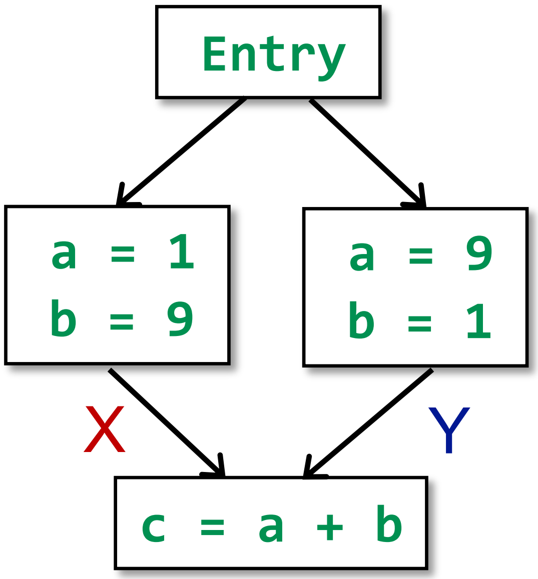
Constant Propagation – Nondistributivity



$F(X \sqcap Y) =$

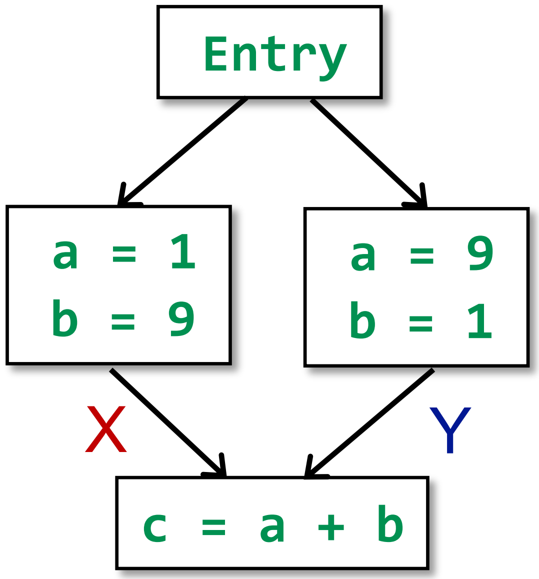
$F(X) \sqcap F(Y) =$

Constant Propagation – Nondistributivity



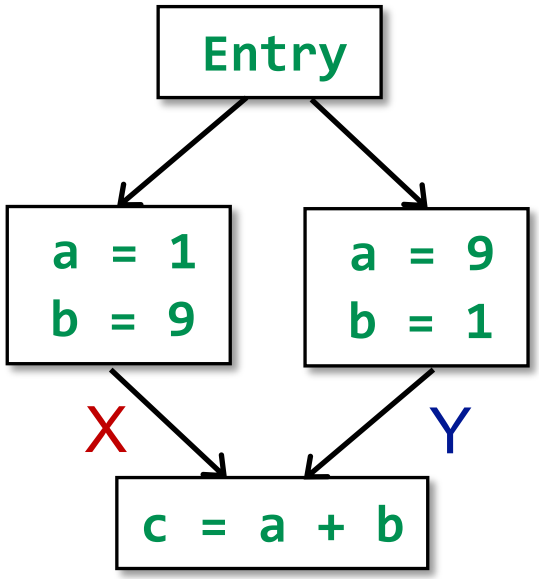
$$F(X \sqcap Y) = \{(a, NAC), (b, NAC), (c, NAC)\}$$
$$F(X) \sqcap F(Y) =$$

Constant Propagation – Nondistributivity



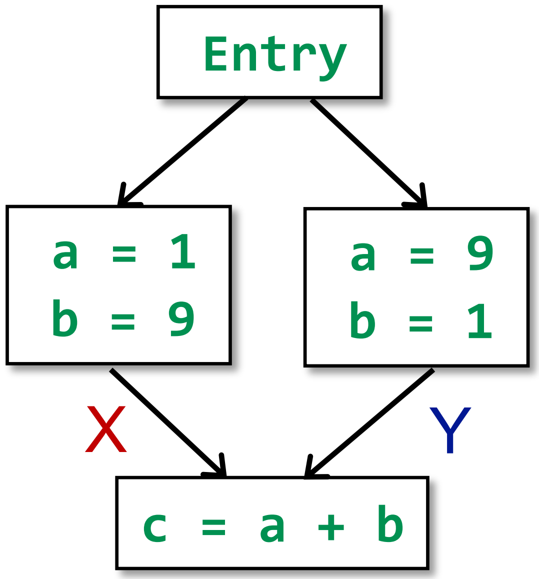
$$F(X \sqcap Y) = \{(a, NAC), (b, NAC), (c, NAC)\}$$
$$F(X) \sqcap F(Y) = \{(a, NAC), (b, NAC), (c, 10)\}$$

Constant Propagation – Nondistributivity



$$F(X \sqcap Y) = \{(a, NAC), (b, NAC), (c, NAC)\}$$
$$F(X) \sqcap F(Y) = \{(a, NAC), (b, NAC), (c, 10)\}$$
$$F(X \sqcap Y) \neq F(X) \sqcap F(Y)$$

Constant Propagation – Nondistributivity



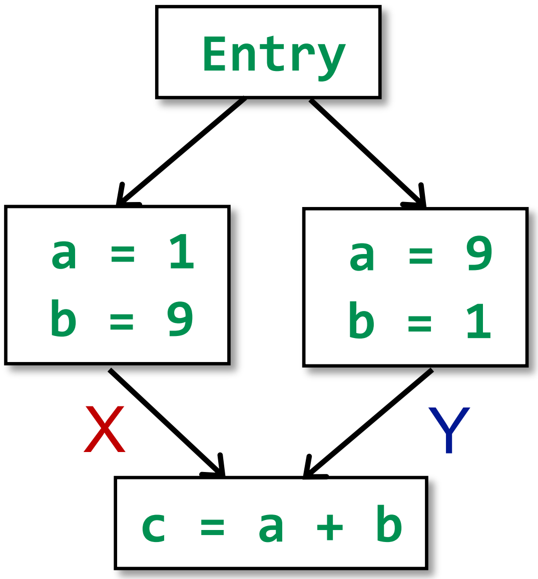
$$F(X \sqcap Y) = \{(a, NAC), (b, NAC), (c, NAC)\}$$

$$F(X) \sqcap F(Y) = \{(a, NAC), (b, NAC), (c, 10)\}$$

$$F(X \sqcap Y) \neq F(X) \sqcap F(Y)$$

$$F(X \sqcap Y) \sqsubseteq F(X) \sqcap F(Y)$$

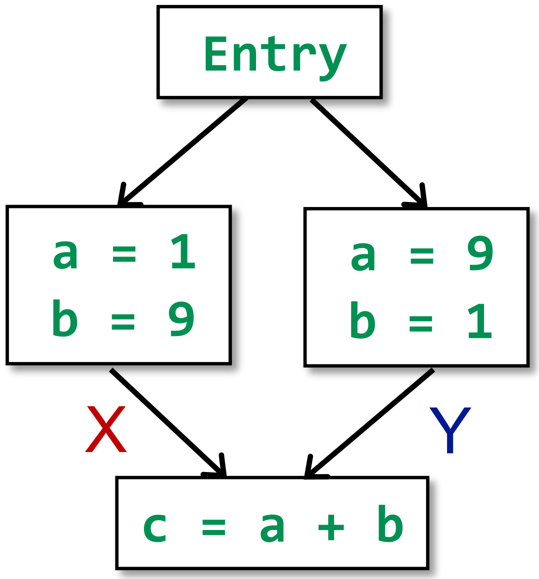
Constant Propagation – Nondistributivity



$$F(X \sqcap Y) = \{(a, NAC), (b, NAC), (c, NAC)\}$$
$$F(X) \sqcap F(Y) = \{(a, NAC), (b, NAC), (c, 10)\}$$
$$F(X \sqcap Y) \neq F(X) \sqcap F(Y)$$
$$F(X \sqcap Y) \sqsubseteq F(X) \sqcap F(Y)$$

Show our constant propagation analysis is monotonic

Constant Propagation – Nondistributivity



$$F(X \sqcap Y) = \{(a, NAC), (b, NAC), (c, NAC)\}$$

$$F(X) \sqcap F(Y) = \{(a, NAC), (b, NAC), (c, 10)\}$$

$$F(X \sqcap Y) \neq F(X) \sqcap F(Y)$$

$$F(X \sqcap Y) \sqsubseteq F(X) \sqcap F(Y)$$

Show our constant propagation analysis is monotonic

Assignment One: Constant Propagation

Worklist Algorithm,

an optimization of Iterative Algorithm

Review Iterative Algorithm for May & Forward Analysis

INPUT: CFG ($kill_B$ and gen_B computed for each basic block B)

OUTPUT: IN[B] and OUT[B] for each basic block B

METHOD:

OUT[*entry*] = ∅;
for (each basic block *B\entry*)
    OUT[B] = ∅;
while (changes to any OUT occur)
    for (each basic block *B\entry*) {
        $\text{IN}[B] = \bigsqcup_{P \text{ a predecessor of } B} \text{OUT}[P]$;
        $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$;
    }

Worklist Algorithm

Forward Analysis

$OUT[entry] = \emptyset;$
for (each basic block *B\entry*)
    $OUT[B] = \emptyset;$
Worklist ← all basic blocks
while (Worklist is not empty)
    Pick a basic block *B* from Worklist
    old_OUT = OUT[*B*]
    $IN[B] = \bigsqcup_{P \text{ a predecessor of } B} OUT[P];$
    $OUT[B] = gen_B \cup (IN[B] - kill_B);$
    if (old_OUT ≠ OUT[*B*])
        Add all successors of *B* to Worklist

Worklist Algorithm

Forward Analysis

$\text{OUT}[\textit{entry}] = \emptyset;$
for (each basic block $B \setminus \textit{entry}$)
    $\text{OUT}[B] = \emptyset;$
Worklist ← all basic blocks
while (Worklist is not empty)
    Pick a basic block B from Worklist
    old_OUT = OUT[B]
    $\text{IN}[B] = \bigsqcup_{P \textit{ a predecessor of } B} \text{OUT}[P];$
    $\text{OUT}[B] = \textit{gen}_B \cup (\text{IN}[B] - \textit{kill}_B);$
    if (old_OUT $\neq$ OUT[B])
        Add all successors of B to Worklist

OUT will not change if IN does not change

Summary (I)

1. Iterative Algorithm, Another View
2. Partial Order
3. Upper and Lower Bounds
4. Lattice, Semilattice, Complete and Product Lattice
5. Data Flow Analysis Framework via Lattice
6. Monotonicity and Fixed Point Theorem

7. Relate Iterative Algorithm to Fixed Point Theorem

8. May/Must Analysis, A Lattice View

9. MOP and Distributivity

10. Constant Propagation

11. Worklist Algorithm

Summary (II)

The X You Need To Understand in This Lecture

- Understand the functional view of iterative algorithm
- The definitions of lattice and complete lattice
- Understand the fixed-point theorem
- How to summarize may and must analyses in lattices
- The relation between MOP and the solution produced by the iterative algorithm
- Constant propagation analysis
- Worklist algorithm

软件分析

南京大学
计算机科学与技术系
程序设计语言与
静态分析研究组
李樾 谭添