CMPT 745
Software Engineering

Basic Formalisms for Software Engineering

Nick Sumner
wsumner@sfu.ca

Formalism is just a tool

- Formal systems are common

Formalism is just a tool

- Formal systems are common
  - High school algebra
  - Classic formal logic
  - Euclidean geometry

Formalism is just a tool

- Formal systems are common
  - High school algebra
  - Classic formal logic
  - Euclidean geometry
- They serve multiple useful purposes

Formalism is just a tool

- Formal systems are common
  - High school algebra
  - Classic formal logic
  - Euclidean geometry
- They serve multiple useful purposes
  - Limit the possibilities that you may consider
  - Check whether reasoning is correct
  - Enable automated techniques for finding solutions

Formalism is just a tool

- Formal systems are common
  - High school algebra
  - Classic formal logic
  - Euclidean geometry
- They serve multiple useful purposes
  - Limit the possibilities that you may consider
  - Check whether reasoning is correct
  - Enable automated techniques for finding solutions
- Choosing the *right* tool for the job can be hard

Formalism is just a tool

- Several specific systems are common
  (in CS and program analysis)

Formalism is just a tool

- Several specific systems are common
  (in CS and program analysis)
  - Order Theory

How to compare elements of a set

Formalism is just a tool

- Several specific systems are common
  (in CS and program analysis)
  - Order Theory
  - Formal Grammars & Automata

Use structure to constrain
the elements of a set

Formalism is just a tool

- Several specific systems are common
  (in CS and program analysis)
  - Order Theory
  - Formal Grammars & Automata
  - Formal Logic (Classical & otherwise)

How and when to infer facts

Formalism is just a tool

- Several specific systems are common
  (in CS and program analysis)
  - Order Theory
  - Formal Grammars & Automata
  - Formal Logic (Classical & otherwise)
- We are going to revisit these (quickly) with some insights on how they can useful in practice.

Formalism is just a tool

- Several specific systems are common
  (in CS and program analysis)
  - Order Theory
  - Formal Grammars & Automata
  - Formal Logic (Classical & otherwise)
- We are going to revisit these (quickly) with some insights on how they can useful in practice.
  - Most students don’t seem to remember them

Formalism is just a tool

- Several specific systems are common
(in CS and program analysis)
  - Order Theory
  - Formal Grammars & Automata
  - Formal Logic (Classical & otherwise)
- We are going to revisit these (quickly) with some insights on how they can useful in practice.
  - Most students don’t seem to remember them
  - Even fewer learn that formalism *can be useful*!

Formalism is just a tool

- Several specific systems are common
  (in CS and program analysis)
  - Order Theory
  - Formal Grammars & Automata
  - Formal Logic (Classical & otherwise)
- We are going to revisit these (quickly) with some insights on how they can useful in practice.
  - Most students don't seem to remember them
  - Even fewer learn that formalism *can be useful*!
  - These techniques are critical for *static program analysis*

Order Theory

Order Theory

- *Order theory* is a field examining how we compare elements of a set.

Order Theory

- *Order theory* is a field examining how we compare elements of a set.
- Simplest example is numbers on a number line:

Set: $\mathbb{Z}$ Relation: $\leq$

Order Theory

- *Order theory* is a field examining how we compare elements of a set.
- Simplest example is numbers on a number line:

Set: $\mathbb{Z}$ Relation: $\leq$

Order Theory

- *Order theory* is a field examining how we compare elements of a set.
- Simplest example is numbers on a number line:

Set: $\mathbb{Z}$ Relation: $\leq$

- $\leq$ is a *total order* on $\mathbb{Z}$.
  - Intuitively, $\forall$ a, b $\in \mathbb{Z}$, either a $\leq$ b or b $\leq$ a

Order Theory

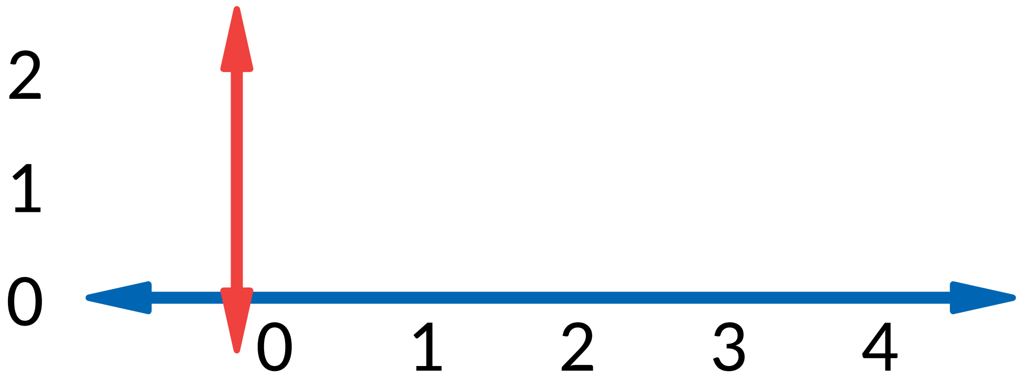
- We often want to compare complex data

Order Theory

- We often want to compare complex data
  - Ordinal, multidimensional, …

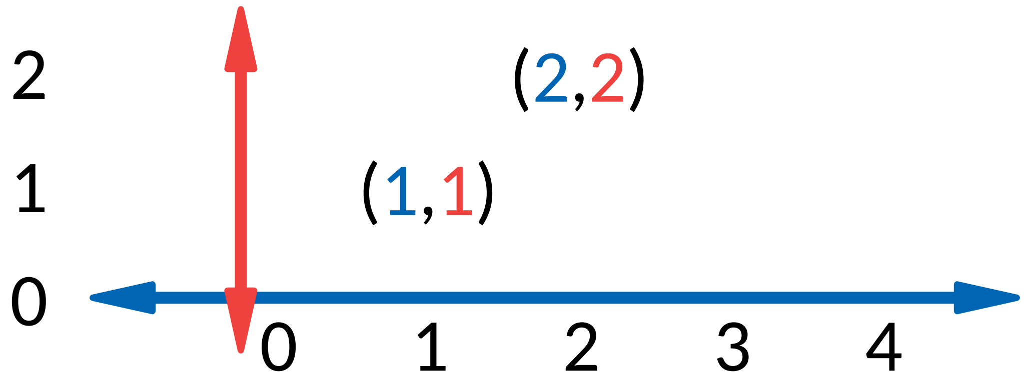
Order Theory

- We often want to compare complex data
  - Ordinal, multidimensional, …



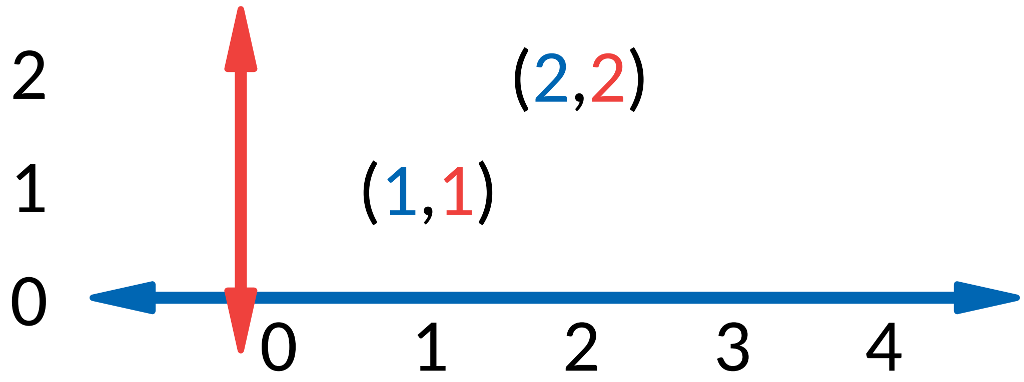
Order Theory

- We often want to compare complex data
  - Ordinal, multidimensional, …



Order Theory

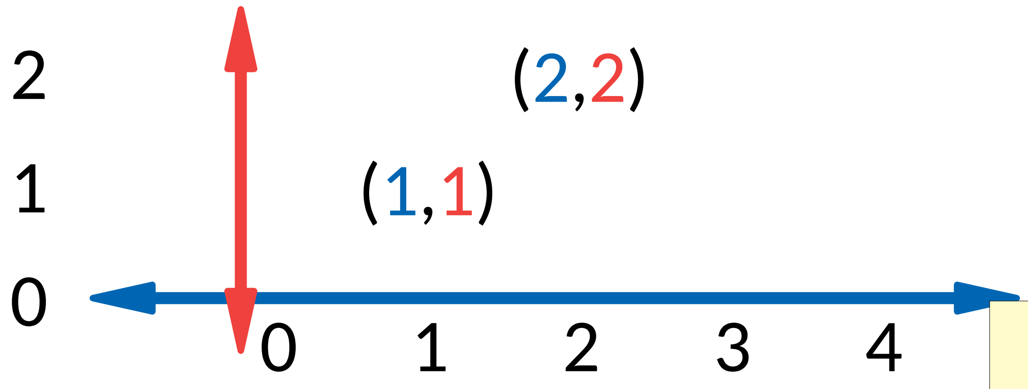
- We often want to compare complex data
  - Ordinal, multidimensional, …



What is the result of
$(1,1) \leq (2,2)$?

Order Theory

- We often want to compare complex data
  - Ordinal, multidimensional, …

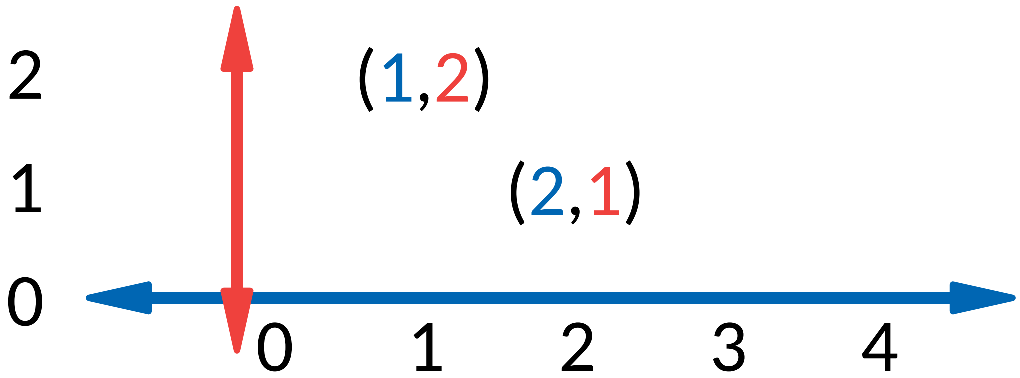


What is the result of (1,1) ≤ (2,2)?

We can take ≤ to be componentwise comparison.

Order Theory

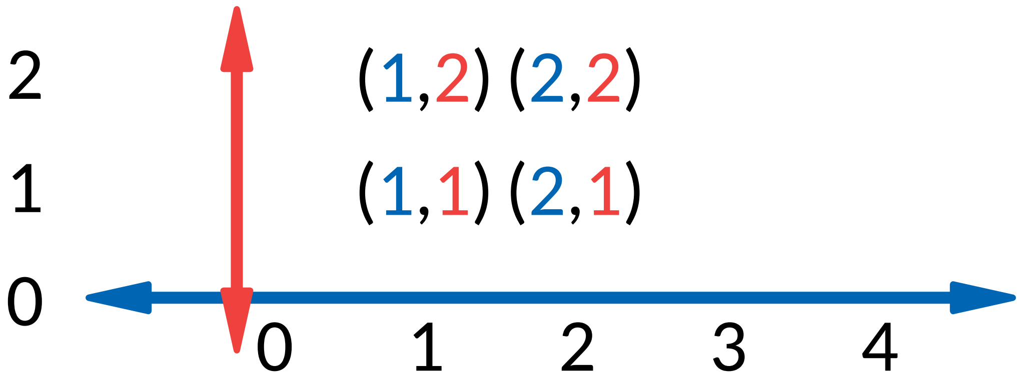
- We often want to compare complex data
  - Ordinal, multidimensional, …



What is the result of
(1,2) ≤ (2,1)?

Order Theory

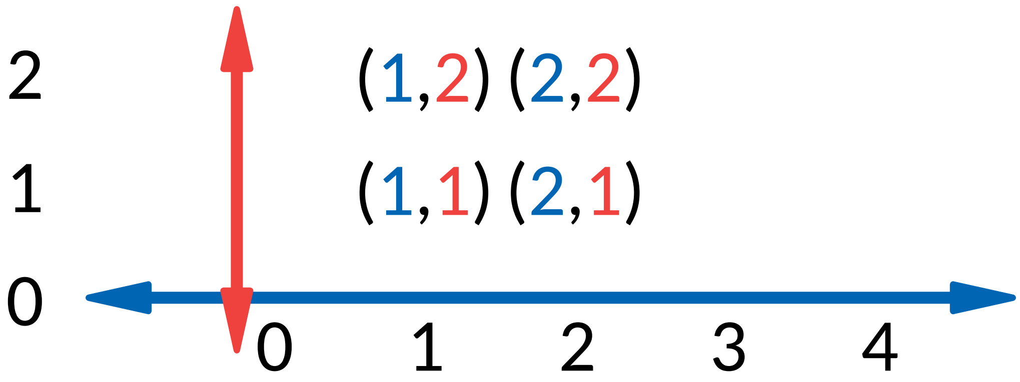
- We often want to compare complex data
  - Ordinal, multidimensional, …



- Componentwise comparison with tuples yields a *partial order*

Order Theory

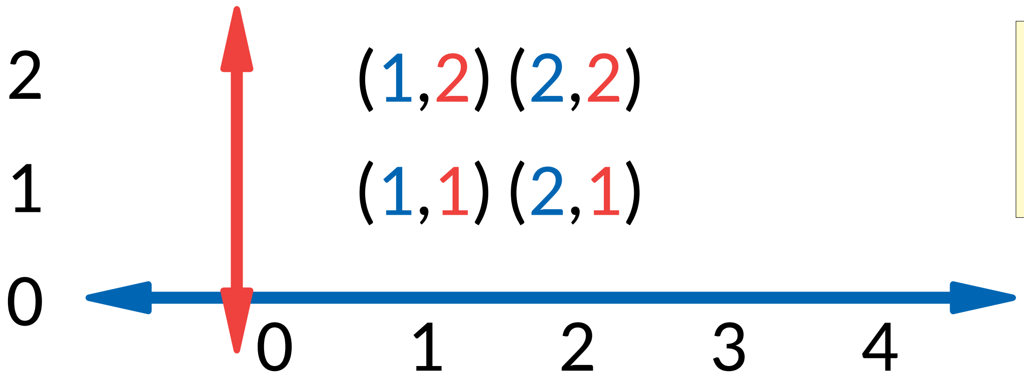
- We often want to compare complex data
  - Ordinal, multidimensional, …



- Componentwise comparison with tuples yields a *partial order*
  - Intuitively, *not all elements are comparable*

Order Theory

- We often want to compare complex data
  - Ordinal, multidimensional, …



Which of these 4 elements are comparable?

- Componentwise comparison with tuples yields a *partial order*
  - Intuitively, *not all elements are comparable*

Partial Orders

- A relation $\leq$ is a *partial order* on a set S if $\forall$ a,b,c $\in$ S
  - Reflexive: $a \leq a$
  - Antisymmetric: $a \leq b$ & $b \leq a \Rightarrow a = b$
  - Transitive: $a \leq b$ & $b \leq c \Rightarrow a \leq c$

Partial Orders

- A relation $\leq$ is a *partial order* on a set S if $\forall$ a,b,c $\in$ S
  - Reflexive: $a \leq a$
  - Antisymmetric: $a \leq b \;\&\; b \leq a \Rightarrow a = b$
  - Transitive: $a \leq b \;\&\; b \leq c \Rightarrow a \leq c$

Partial Orders

- A relation $\leq$ is a *partial order* on a set S if $\forall$ a,b,c $\in$ S
  - Reflexive: $a \leq a$
  - Antisymmetric: $a \leq b$ & $b \leq a \Rightarrow a = b$
  - Transitive: $a \leq b$ & $b \leq c \Rightarrow a \leq c$

Partial Orders

- A relation $\leq$ is a *partial order* on a set S if $\forall$ a,b,c $\in$ S
  - Reflexive: $a \leq a$
  - Antisymmetric: $a \leq b \ \& \ b \leq a \Rightarrow a = b$
  - Transitive: $a \leq b \ \& \ b \leq c \Rightarrow a \leq c$

How does a
total order compare?

Partial Orders

- A relation ≤ is a *partial order* on a set S if ∀ a,b,c ∈ S
  - Reflexive: $a \leq a$
  - Antisymmetric: $a \leq b$ & $b \leq a \Rightarrow a = b$
  - Transitive: $a \leq b$ & $b \leq c \Rightarrow a \leq c$
- When reasoning about partial orders, we prefer ⊑

Partial Orders

- A relation $\leq$ is a *partial order* on a set S if $\forall$ a,b,c $\in$ S
  - Reflexive: $a \leq a$
  - Antisymmetric: $a \leq b \;\&\; b \leq a \Rightarrow a = b$
  - Transitive: $a \leq b \;\&\; b \leq c \Rightarrow a \leq c$
- When reasoning about partial orders, we prefer $\sqsubseteq$
- Common partial orders include
  - substring, subsequence, subset relationships

Partial Orders

- A relation ≤ is a *partial order* on a set S if ∀ a,b,c ∈ S

$$ab \preceq_{str} xabyz$$

$$ab \preceq_{seq} xaybz$$

$$\{a,b\} \subseteq \{a,b,x,y,z\}$$

- Common partial orders include
  - substring, subsequence, subset relationships

Partial Orders

- A relation ≤ is a *partial order* on a set S if ∀ a,b,c ∈ S
  - Reflexive: $a \le a$
  - Antisymmetric: $a \le b$ & $b \le a \Rightarrow a = b$
  - Transitive: $a \le b$ & $b \le c \Rightarrow a \le c$
- When reasoning about partial orders, we prefer $\sqsubseteq$
- Common partial orders include
  - substring, subsequence, subset relationships
  - componentwise orderings

Partial Orders

- A relation ≤ is a *partial order* on a set S if ∀ a,b,c ∈ S

$$(1,1) \sqsubseteq (1,2)$$
$$(1,1) \sqsubseteq (2,2)$$

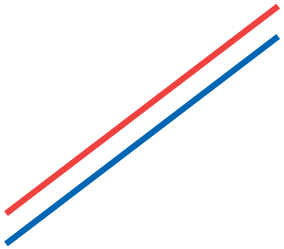
- Common partial orders include
  - substring, subsequence, subset relationships
  - componentwise orderings

Partial Orders

- A relation ≤ is a *partial order* on a set S if ∀ a,b,c ∈ S
  - Reflexive: $a \leq a$
  - Antisymmetric: $a \leq b \;\&\; b \leq a \Rightarrow a = b$
  - Transitive: $a \leq b \;\&\; b \leq c \Rightarrow a \leq c$
- When reasoning about partial orders, we prefer $\sqsubseteq$
- Common partial orders include
  - substring, subsequence, subset relationships
  - componentwise orderings
  - functions (considering all input/output mappings)

Partial Orders

- A relation ≤ is a *partial order* on a set S if ∀ a,b,c ∈ S

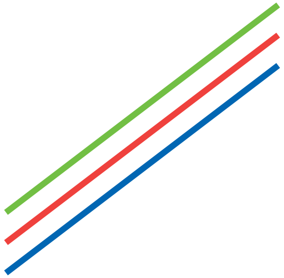


$f(x) = x + 1 \sqsubseteq g(x) = x + 2$

- Common partial orders include
  - substring, subsequence, subset relationships
  - componentwise orderings
  - functions (considering all input/output mappings)

Partial Orders

- A relation ≤ is a *partial order* on a set S if ∀ a,b,c ∈ S



$f(x) = x + 1 \sqsubseteq g(x) = x + 2$

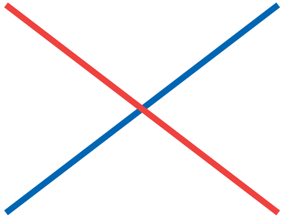
- Common partial orders include
  - substring, subsequence, subset relationships
  - componentwise orderings
  - functions (considering all input/output mappings)

Partial Orders

- A relation ≤ is a *partial order* on a set S if ∀ a,b,c ∈ S

$f(x) = x + 1 \sqsubseteq g(x) = x + 2$

$h(x) = x \not\sqsubseteq i(x) = -x$



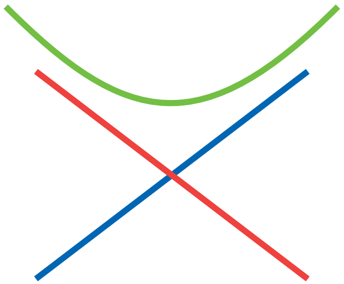
- Common partial orders include
  - substring, subsequence, subset relationships
  - componentwise orderings
  - functions (considering all input/output mappings)

Partial Orders

- A relation ≤ is a *partial order* on a set S if ∀ a,b,c ∈ S

$f(x) = x + 1 \sqsubseteq g(x) = x + 2$

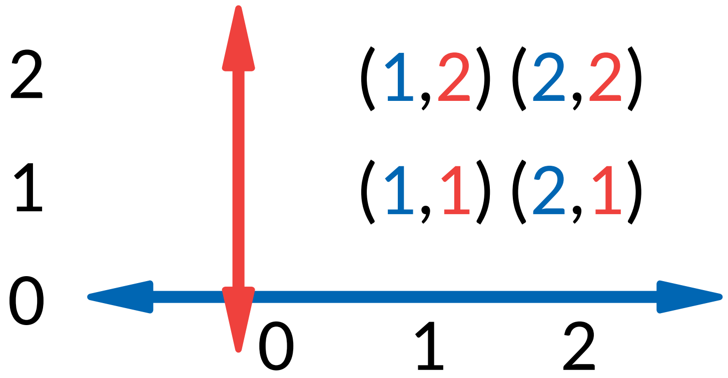
$h(x) = x \not\sqsubseteq i(x) = -x$



- Common partial orders include
  - substring, subsequence, subset relationships
  - componentwise orderings
  - functions (considering all input/output mappings)

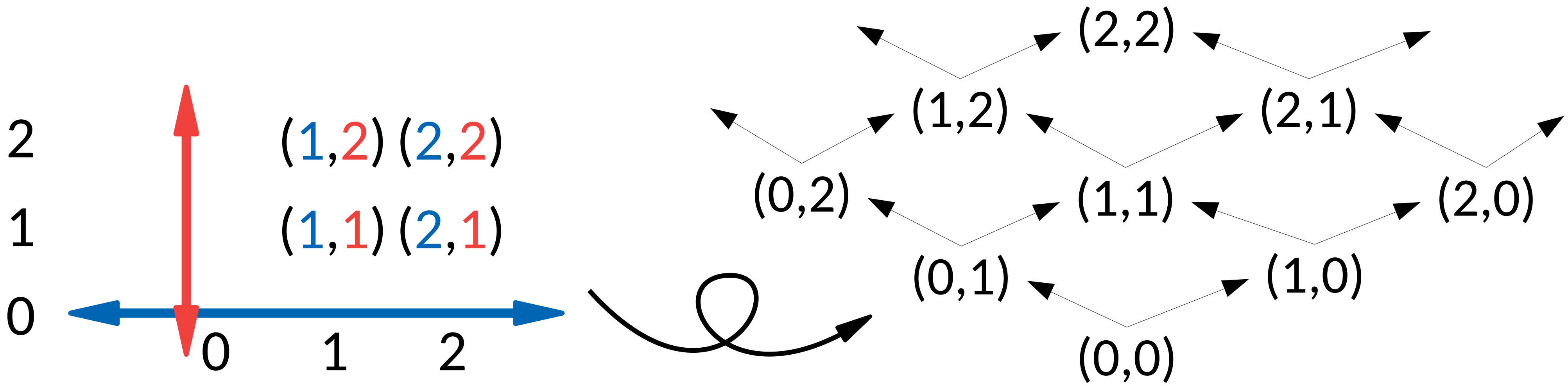
Partial Orders

- We can express the structure of partial orders using *(semi-)lattices*.



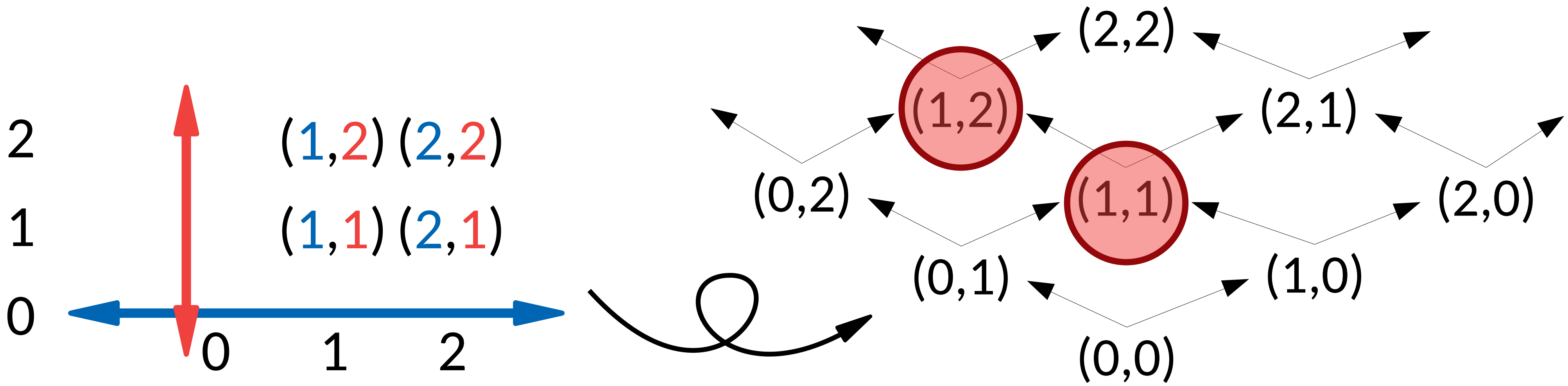
Partial Orders

- We can express the structure of partial orders using *(semi-)lattices*.



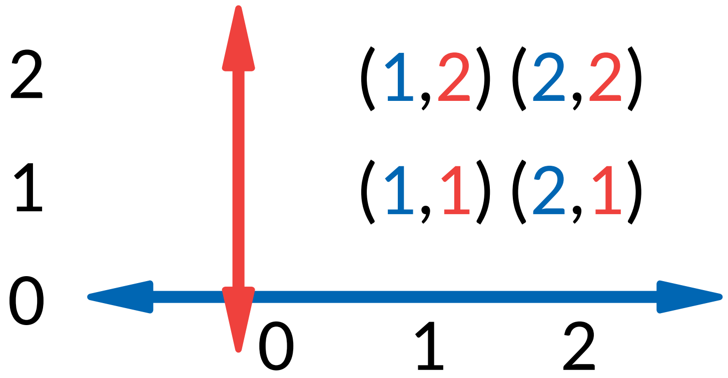
Partial Orders

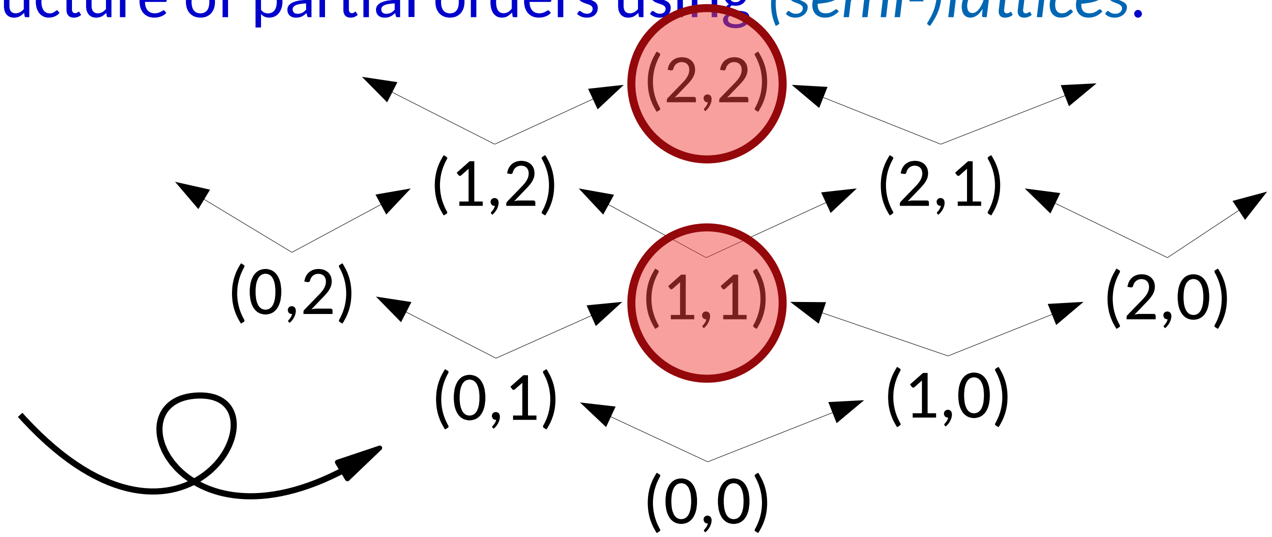
- We can express the structure of partial orders using *(semi-)lattices*.



Partial Orders

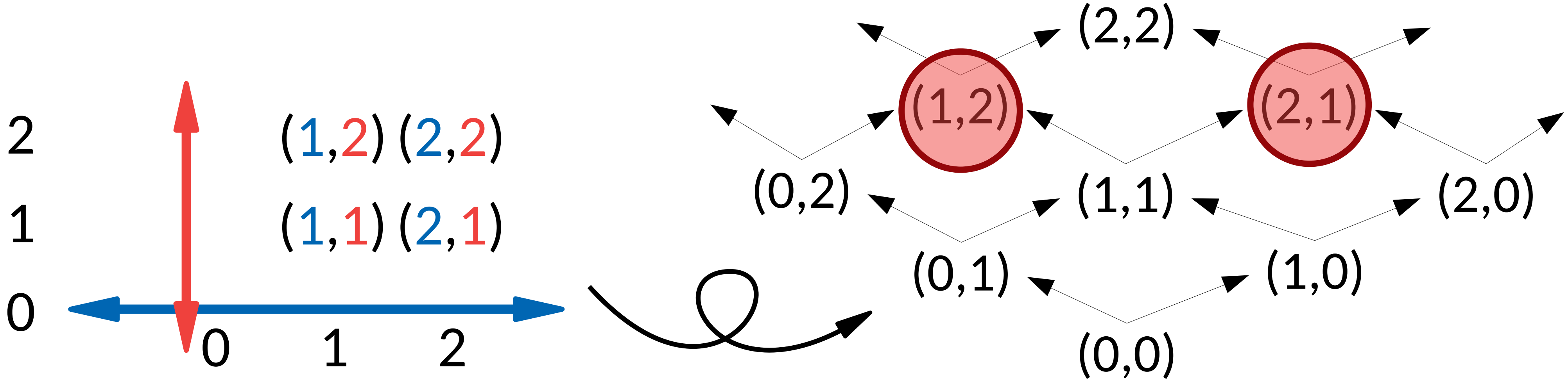
- We can express the structure of partial orders using *(semi-)lattices*.





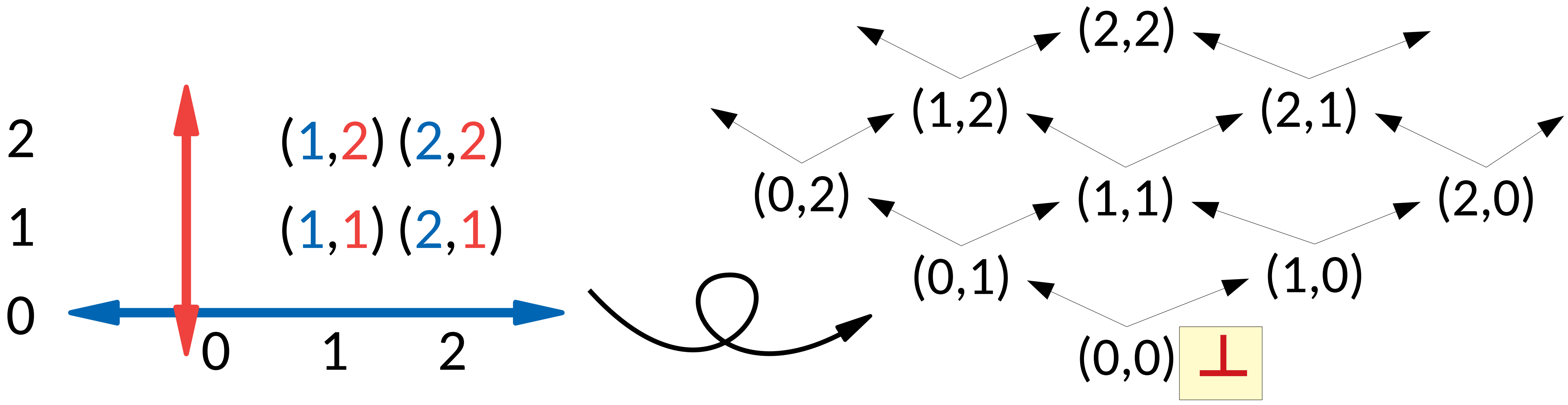
Partial Orders

- We can express the structure of partial orders using *(semi-)lattices*.



Partial Orders

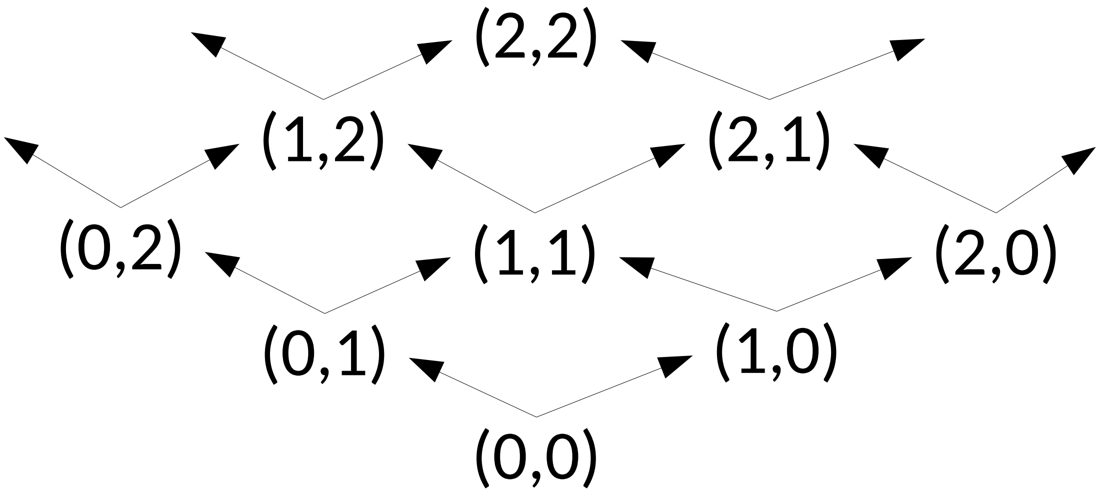
- We can express the structure of partial orders using *(semi-)lattices*.



- If unique least/greatest elements exist, we call them ⊥(bottom)/⊤(top)

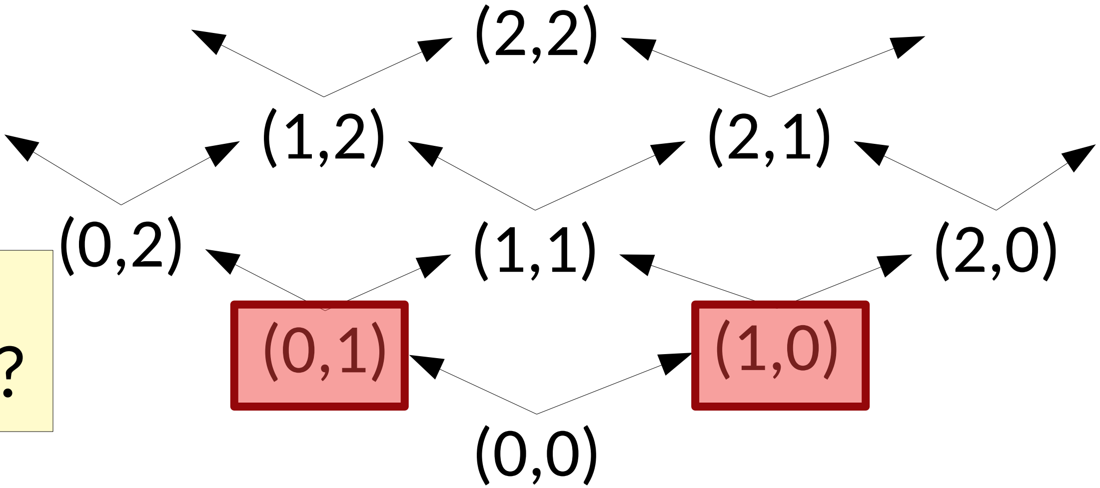
Partial Orders

- We are often interested in upper and lower bounds.



Partial Orders

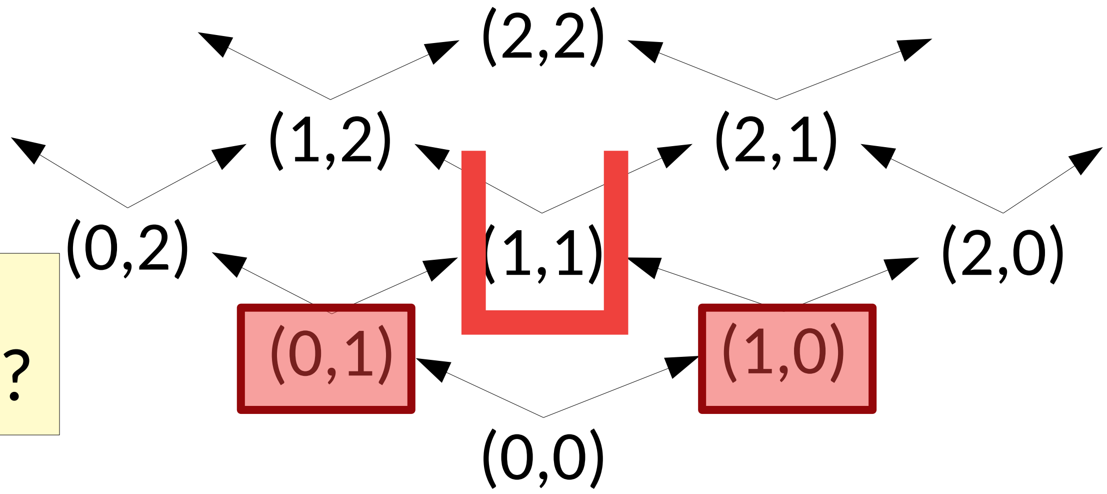
- We are often interested in upper and lower bounds.
  - A ***join*** a ⊔ b is the least upper bound of a and b



What is
(0,1)⊔(1,0)?

Partial Orders

- We are often interested in upper and lower bounds.
  - A ***join*** a ⊔ b is the least upper bound of a and b

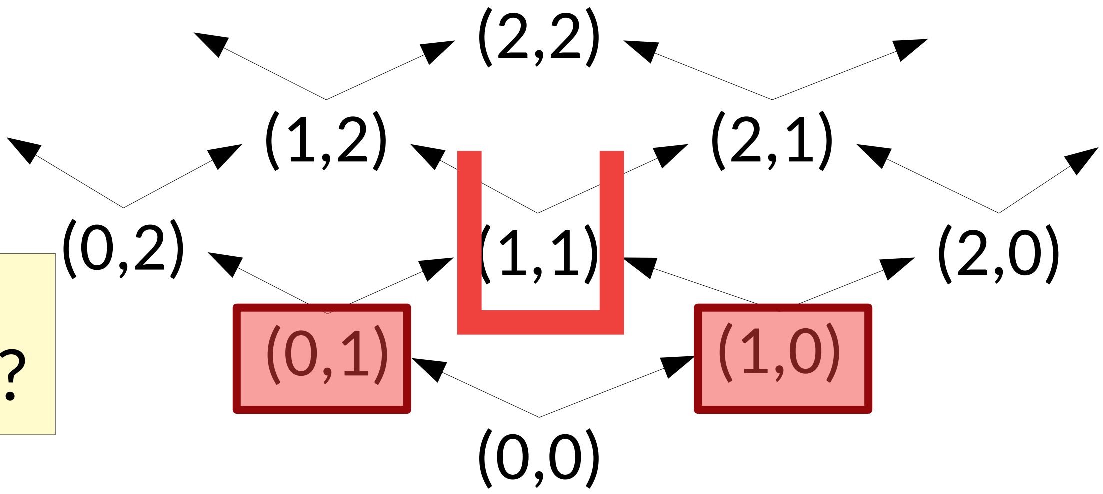


What is (0,1)⊔(1,0)?

Partial Orders

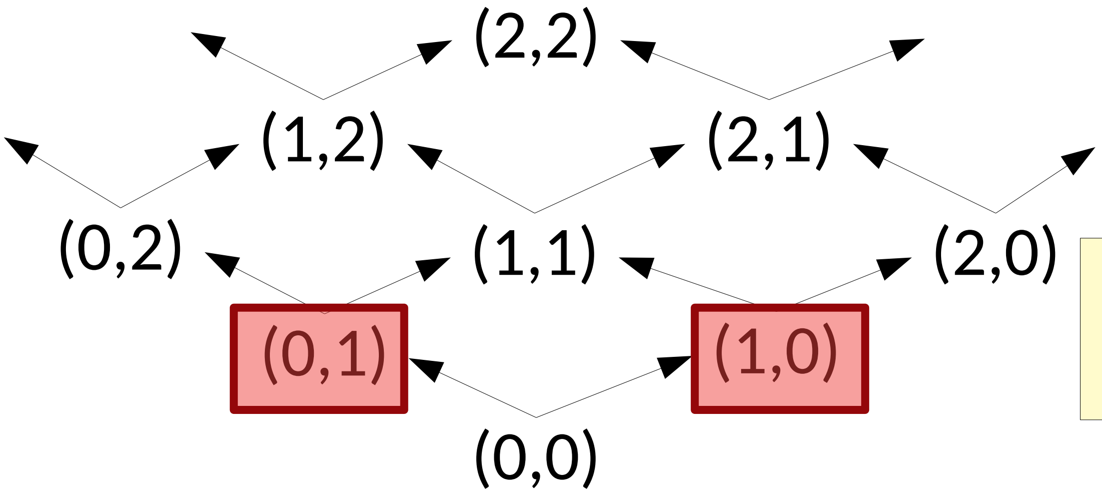
- We are often interested in upper and lower bounds.
  - A ***join*** a ⊔ b is the least upper bound of a and b
    a ⊑ (a⊔b) & b ⊑ (a⊔b) & (a ⊑ c & b⊑c → (a⊔b) ⊑ c)

What is (0,1)⊔(1,0)?



Partial Orders

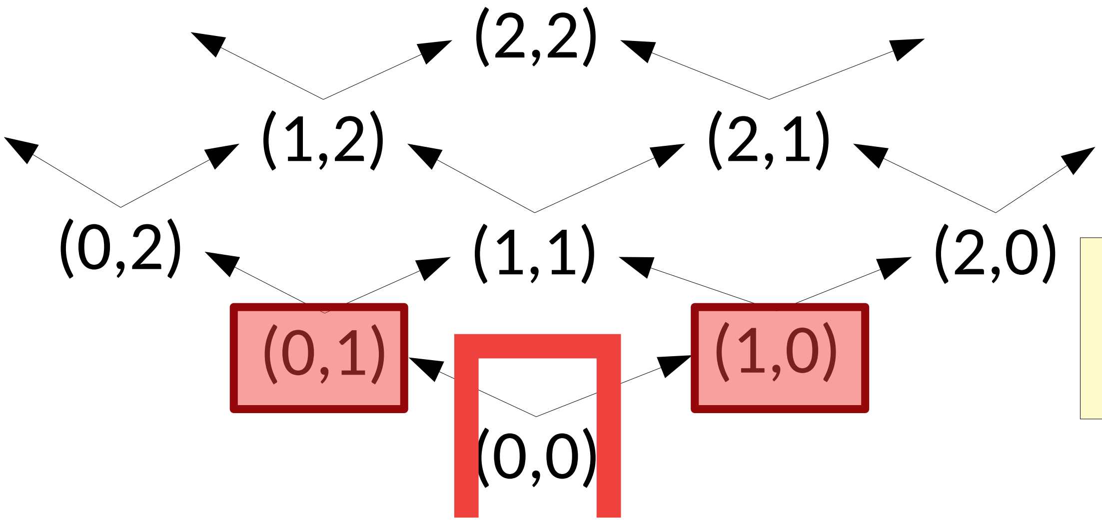
- We are often interested in upper and lower bounds.
  - A ***join*** a ⊔ b is the least upper bound of a and b
  - A ***meet*** a ⊓ b is the greatest lower bound of and and b



What is (0,1)⊓(1,0)?

Partial Orders

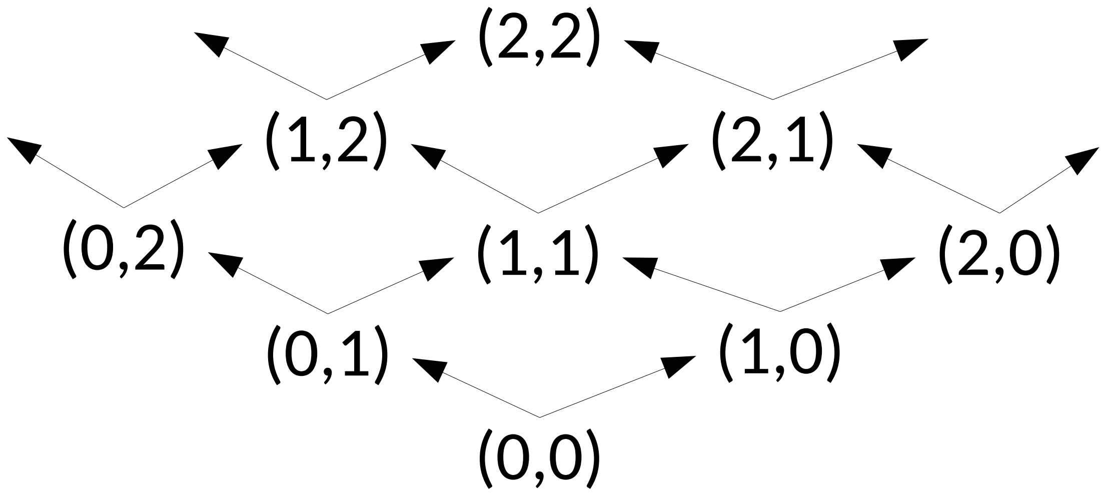
- We are often interested in upper and lower bounds.
  - A ***join*** a ⊔ b is the least upper bound of a and b
  - A ***meet*** a ⊓ b is the greatest lower bound of and and b



What is (0,1)⊓(1,0)?

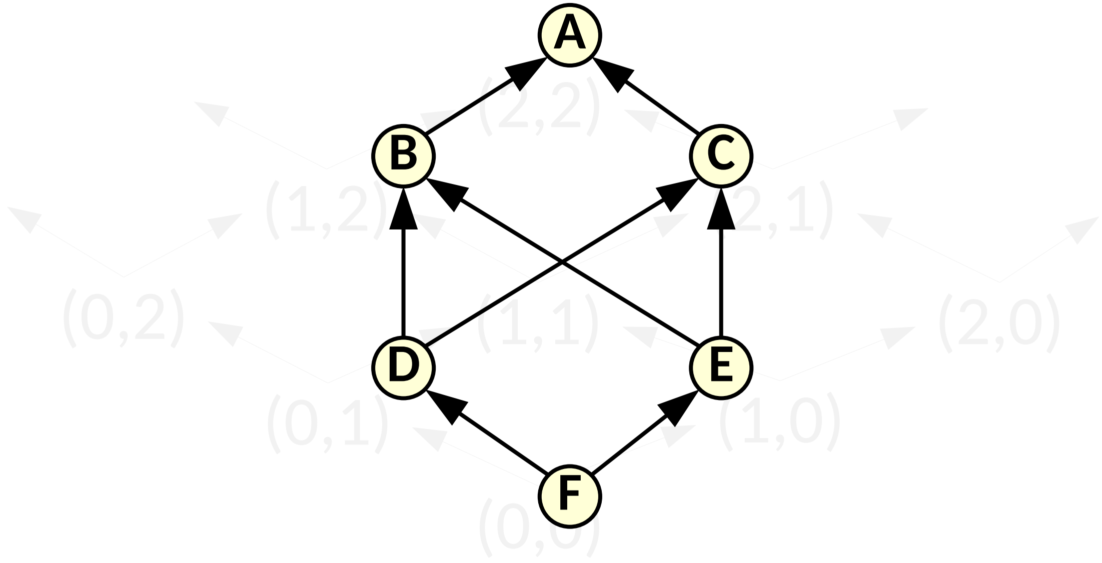
Partial Orders

- We are often interested in upper and lower bounds.
  - A ***join*** a ⊔ b is the least upper bound of a and b
  - A ***meet*** a ⊓ b is the greatest lower bound of and and b
  - Bounds must be *unique* and may not exist.



Partial Orders

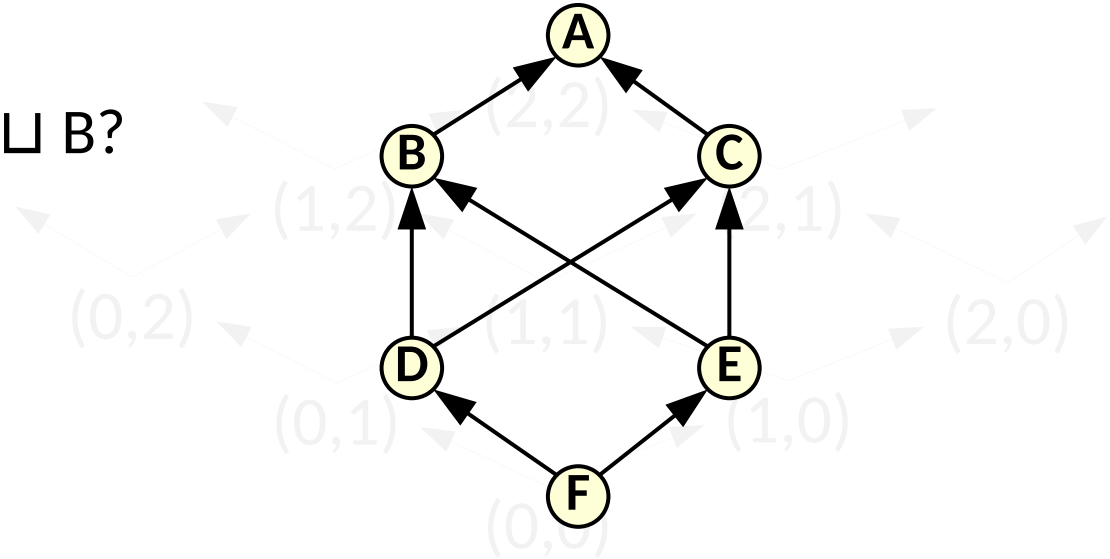
- We are often interested in upper and lower bounds.
  - A ***join*** a ⊔ b is the least upper bound of a and b
  - A ***meet*** a ⊓ b is the greatest lower bound of and and b
  - Bounds must be *unique* and may not exist.



Partial Orders

- We are often interested in upper and lower bounds.
  - A ***join*** a ⊔ b is the least upper bound of a and b
  - A ***meet*** a ⊓ b is the greatest lower bound of and and b
  - Bounds must be *unique* and may not exist.

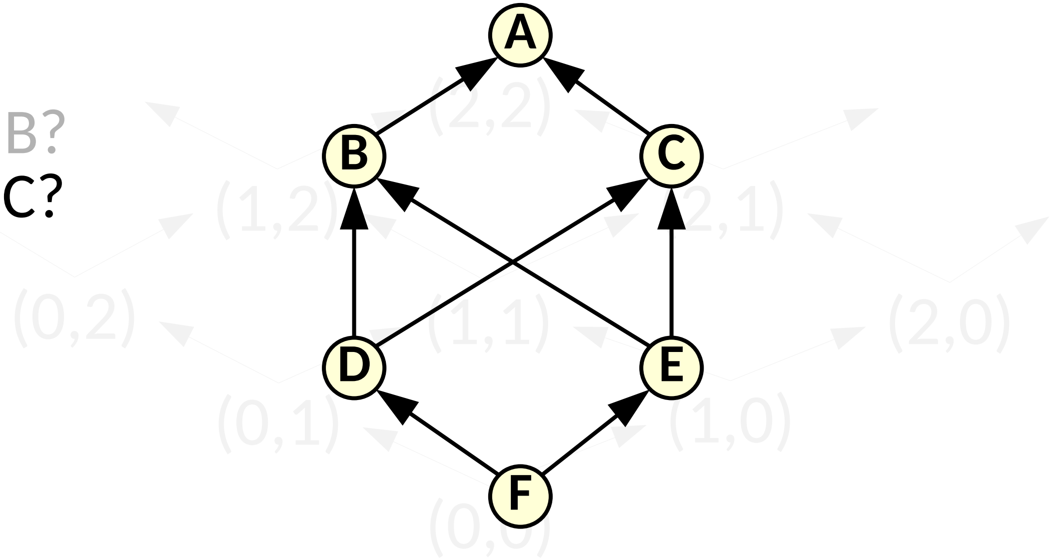
What is A ⊔ B?



Partial Orders

- We are often interested in upper and lower bounds.
  - A ***join*** a ⊔ b is the least upper bound of a and b
  - A ***meet*** a ⊓ b is the greatest lower bound of and and b
  - Bounds must be *unique* and may not exist.

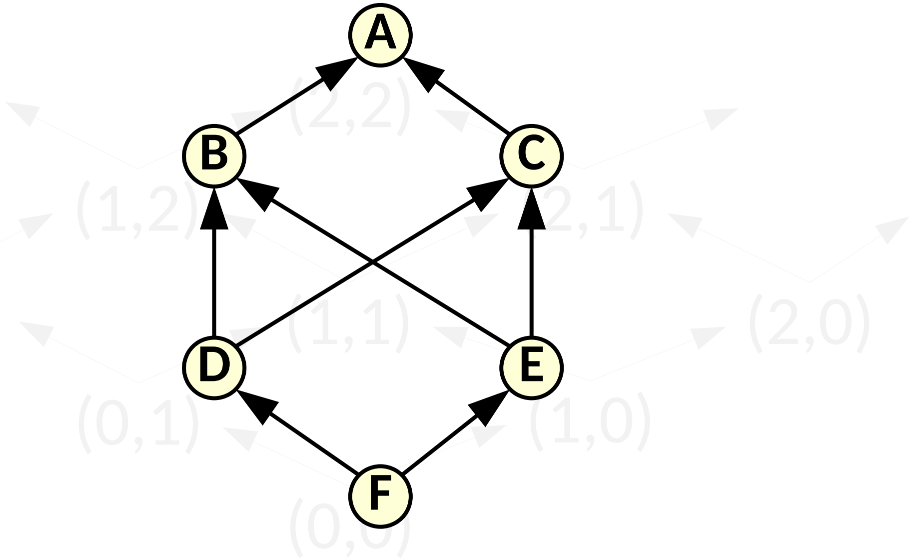
What is A ⊔ B?
What is B ⊔ C?



Partial Orders

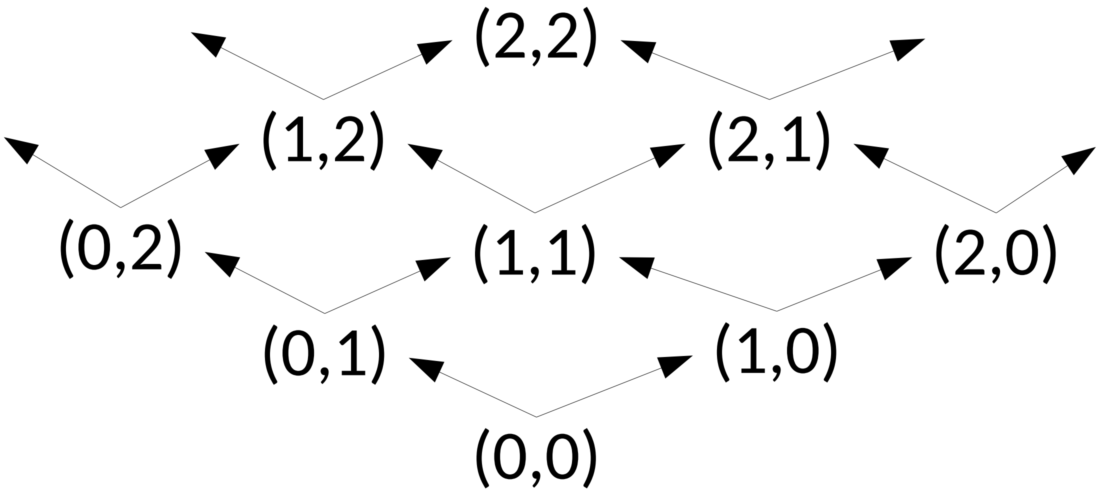
- We are often interested in upper and lower bounds.
  - A ***join*** a ⊔ b is the least upper bound of a and b
  - A ***meet*** a ⊓ b is the greatest lower bound of and and b
  - Bounds must be *unique* and may not exist.

What is A ⊔ B?
What is B ⊔ C?
What is D ⊔ E?



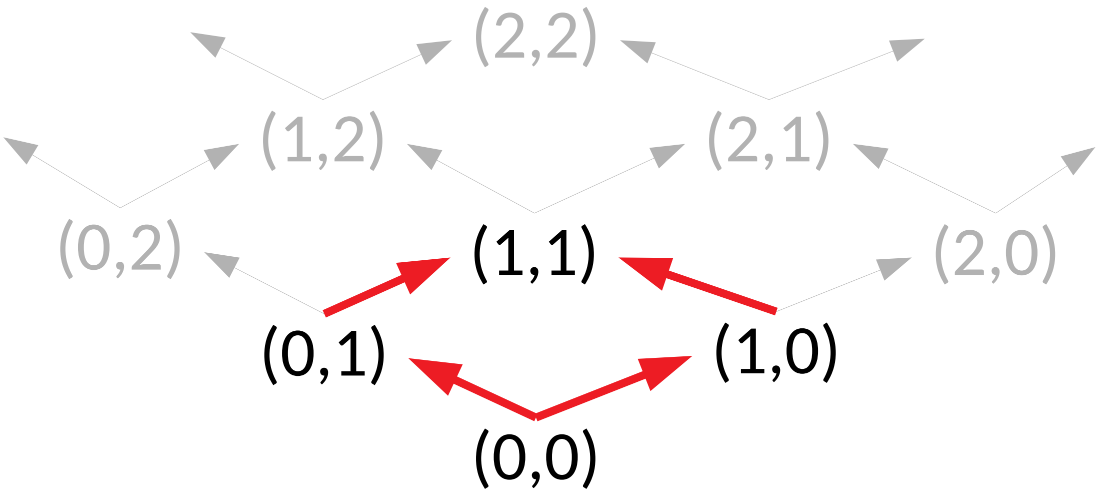
Partial Orders

- We are often interested in upper and lower bounds.
  - A ***join*** a ⊔ b is the least upper bound of a and b
  - A ***meet*** a ⊓ b is the greatest lower bound of and and b
  - Bounds must be unique and may not exist.
  - ∀S’⊆S,



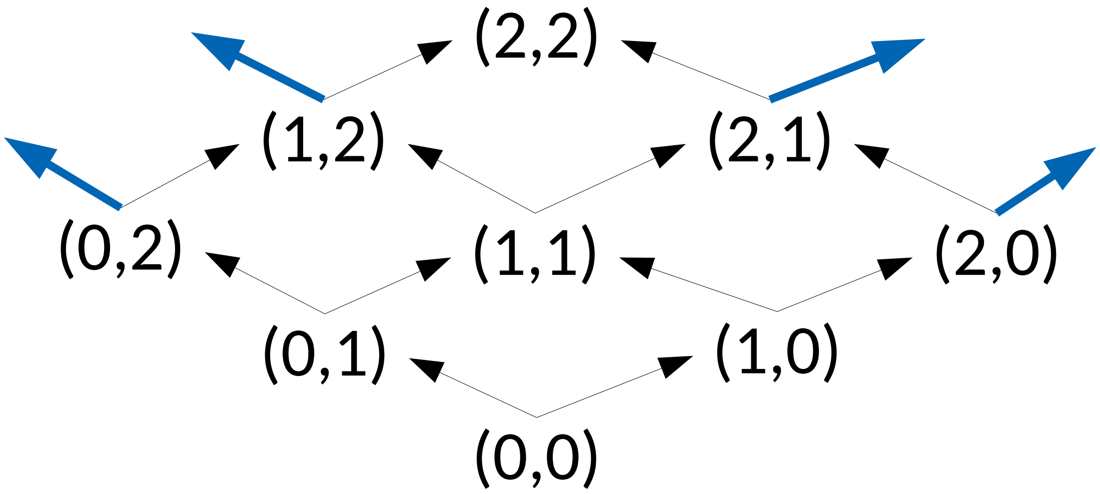
Partial Orders

- We are often interested in upper and lower bounds.
  - A ***join*** a ⊔ b is the least upper bound of a and b
  - A ***meet*** a ⊓ b is the greatest lower bound of and and b
  - Bounds must be unique and may not exist.
  - ∀S'⊆S, ∃⊔S' & ⊓S'⇒ lattice



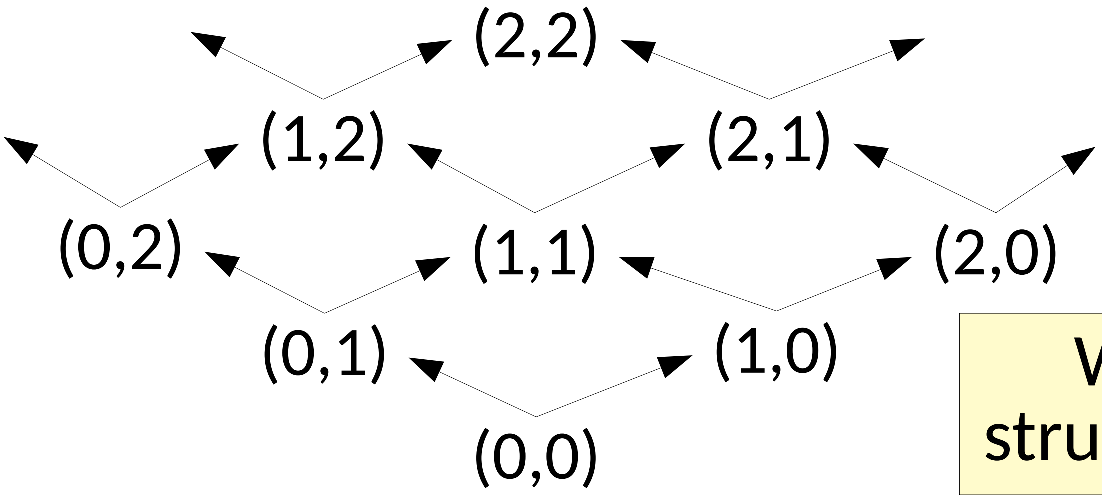
Partial Orders

- We are often interested in upper and lower bounds.
  - A ***join*** a ⊔ b is the least upper bound of a and b
  - A ***meet*** a ⊓ b is the greatest lower bound of and and b
  - Bounds must be unique and may not exist.
  - ∀S'⊆S, ∃⊔S' & ⊓S'⇒ lattice, ∃⊔S' **or** ∃⊓S'⇒ semilattice



Partial Orders

- We are often interested in upper and lower bounds.
  - A ***join*** a ⊔ b is the least upper bound of a and b
  - A ***meet*** a ⊓ b is the greatest lower bound of and and b
  - Bounds must be unique and may not exist.
  - ∀S'⊆S, ∃⊔S' **&** ⊓S'⇒ lattice, ∃⊔S' **or** ∃⊓S'⇒ semilattice



What is the structure shown?

Partial Orders

- A product of lattices (partial orders) yields a lattice (partial order)

$$L_1 \times L_2$$

Partial Orders

- A product of lattices (partial orders) yields a lattice (partial order)
  - We already saw componentwise orderings for tuples.
    This is the same.

$$L_1 \times L_2$$

$$\mathbb{Z} \times \mathbb{Z}$$

Partial Orders

- A product of lattices (partial orders) yields a lattice (partial order)
  - We already saw componentwise orderings for tuples.
    This is the same.

$$L_1 \times L_2$$

$$\mathbb{Z} \times \mathbb{Z}$$

A total order is a partial order.
Products of total orders are partial orders

Partial Orders

- A product of lattices (partial orders) yields a lattice (partial order)
  - We already saw componentwise orderings for tuples.
    This is the same.
- Several expected principles naturally apply

Partial Orders

- A product of lattices (partial orders) yields a lattice (partial order)
  - We already saw componentwise orderings for tuples.
    This is the same.
- Several expected principles naturally apply
  - Monotonicity
    $(X, \sqsubseteq_{\mathbf{x}})$, $(Y, \sqsubseteq_{\mathbf{y}})$, $f: X \to Y$

Partial Orders

- A product of lattices (partial orders) yields a lattice (partial order)
  - We already saw componentwise orderings for tuples.
    This is the same.

- Several expected principles naturally apply
  - Monotonicity
    $(X, \sqsubseteq_{\mathbf{x}})$, $(Y, \sqsubseteq_{\mathbf{y}})$, $f: X \to Y$
    $x_1 \sqsubseteq_{\mathbf{x}} x_2 \;\rightarrow\; f(x_1) \sqsubseteq_{\mathbf{y}} f(x_2)$ (f is monotonic)

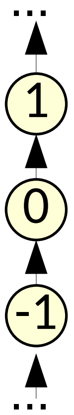
Partial Orders

- A product of lattices (partial orders) yields a lattice (partial order)
  - We already saw componentwise orderings for tuples.
    This is the same.
- Several expected principles naturally apply
  - Monotonicity
  - Continuity
  - Fixed Points
  - …

Partial Orders

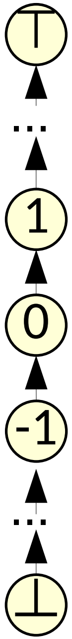
- A product of lattices (partial orders) yields a lattice (partial order)
  - We already saw componentwise orderings for tuples.
    This is the same.
- Several expected principles naturally apply
  - Monotonicity
  - Continuity
  - Fixed Points
  - …
- We can even consider different orders for the same sets!

Partial Orders



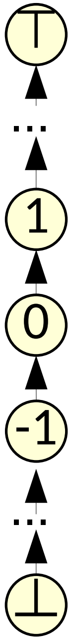
- We can even consider different orders for the same sets!

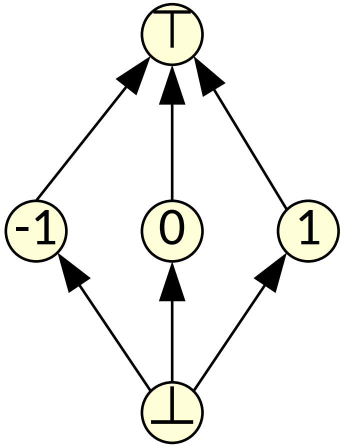
Partial Orders



- We can even consider different orders for the same sets!

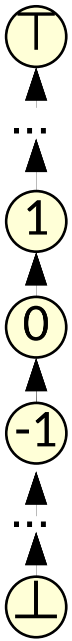
Partial Orders

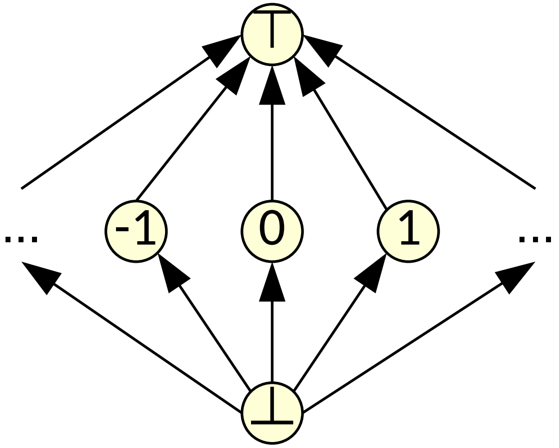




- We can even consider different orders for the same sets!

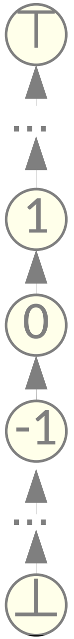
Partial Orders

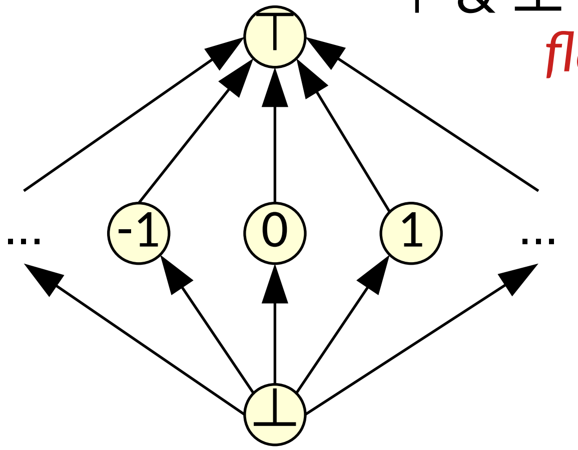




- We can even consider different orders for the same sets!

Partial Orders

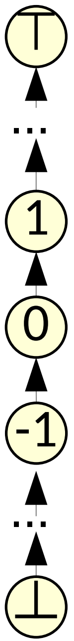


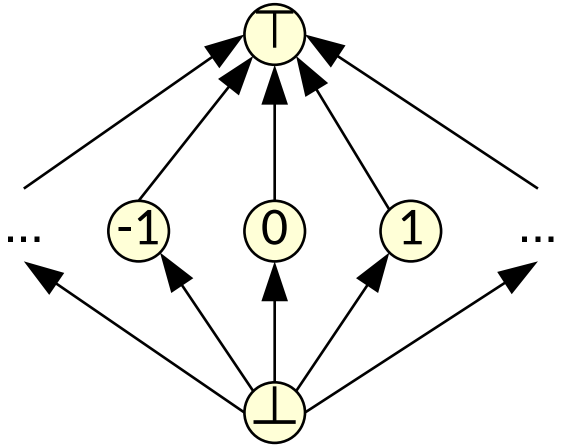


Combining a set with ⊤ & ⊥ like this yields a *flat lattice*.

- We can even consider different orders for the same sets!

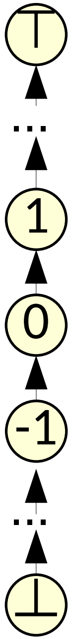
Partial Orders

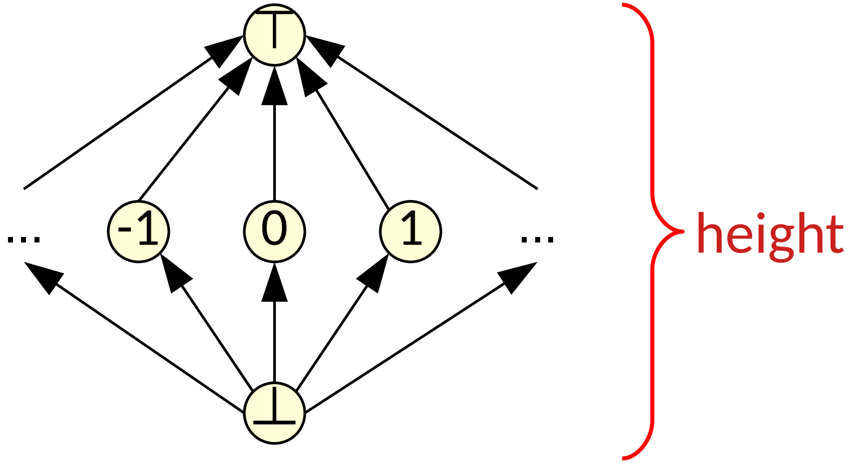




- We can even consider different orders for the same sets!
  - Careful structuring of our orderings can express different things. What do these two lattices express?

Partial Orders





- We can even consider different orders for the same sets!
  - Careful structuring of our orderings can express different things. What do these two lattices express?
  - Many use cases can also be affected by the height of a lattice.

Partial Orders

- Partial orders & lattices can be very useful

Partial Orders

- Partial orders & lattices can be very useful
  - A formal structure for reasoning about relative value

Partial Orders

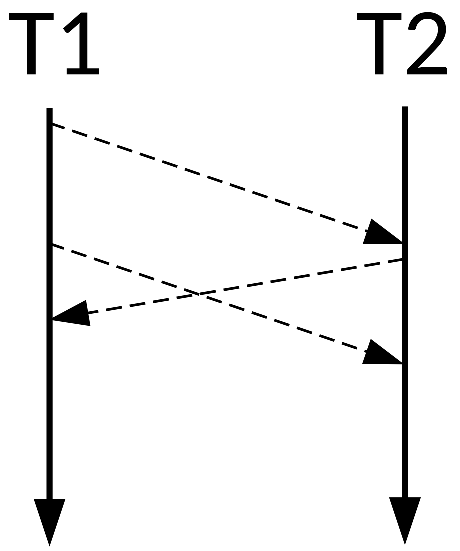
- Partial orders & lattices can be very useful
  - A formal structure for reasoning about relative value
  - modern cryptography (including post-quantum)

Partial Orders

- Partial orders & lattices can be very useful
  - A formal structure for reasoning about relative value
  - modern cryptography
  - concurrency & distributed systems

Partial Orders

- Partial orders & lattices can be very useful
  - A formal structure for reasoning about relative value
  - modern cryptography
  - concurrency & distributed systems

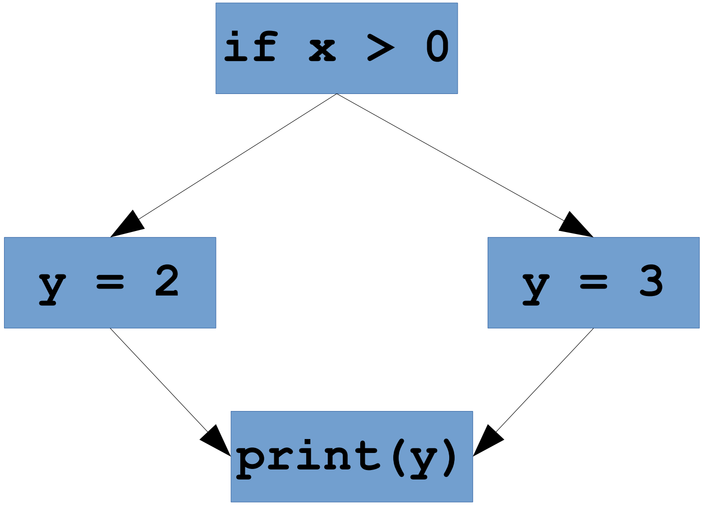


Partial Orders

- Partial orders & lattices can be very useful
  - A formal structure for reasoning about relative value
  - modern cryptography
  - concurrency & distributed systems
  - dataflow analysis & proving program properties

Partial Orders

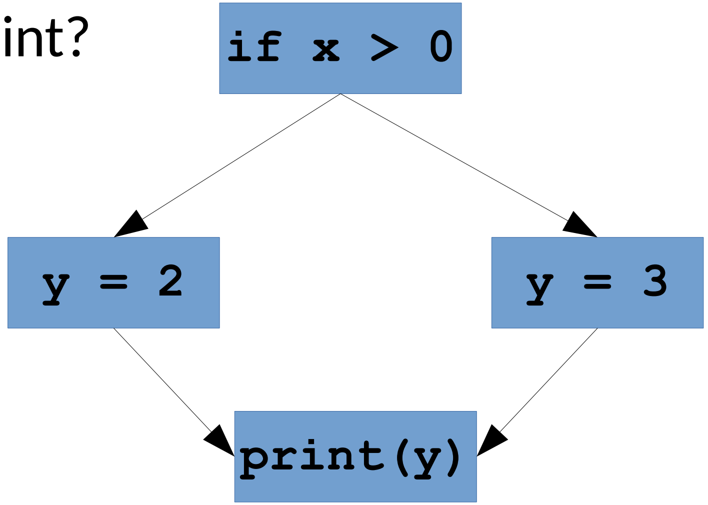
- Partial orders & lattices can be very useful
  - A formal structure for reasoning about relative value
  - modern cryptography
  - concurrency & distributed systems
  - dataflow analysis & proving program properties



Partial Orders

- Partial orders & lattices can be very useful
  - A formal structure for reasoning about relative value
  - modern cryptography
  - concurrency & distributed systems
  - dataflow analysis & proving program properties

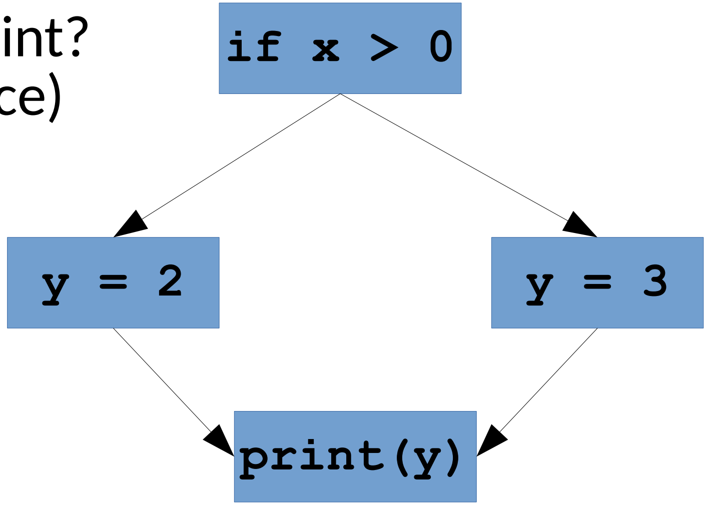
What can the last line print?



Partial Orders

- Partial orders & lattices can be very useful
  - A formal structure for reasoning about relative value
  - modern cryptography
  - concurrency & distributed systems
  - dataflow analysis & proving program properties

What can the last line print?
2 or 3? (set lattice)



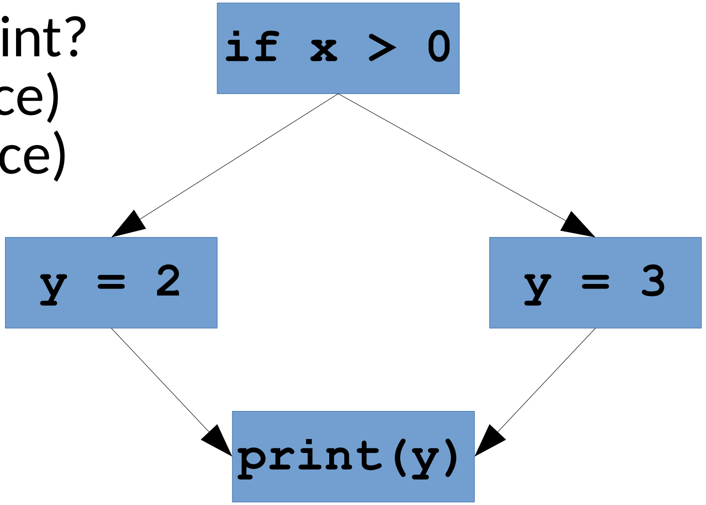
Partial Orders

- Partial orders & lattices can be very useful
  - A formal structure for reasoning about relative value
  - modern cryptography
  - concurrency & distributed systems
  - dataflow analysis & proving program properties

What can the last line print?

2 or 3? (set lattice)

unknown? (flat lattice)



Formal Grammars & Automata

Formal Grammars & Automata

- Grammars define the structure of elements in a set
  - Alternatively, they generate the set via structure

Formal Grammars & Automata

- Grammars define the structure of elements in a set
  - Alternatively, they generate the set via structure
- They commonly define *formal languages*
  - Sets of strings over a defined alphabet

Formal Grammars & Automata

- Grammars define the structure of elements in a set
  - Alternatively, they generate the set via structure
- They commonly define *formal languages*
  - Sets of strings over a defined alphabet
- They are effective at *constraining* sets & search spaces

Regular Languages & Finite Automata

- A *regular language* can be expressed via a *regular expression*

Regular Languages & Finite Automata

- A *regular language* can be expressed via a *regular expression*

```
regex  → symbol
       | `(` regex `)`
       | regex `*`
       | regex `|` regex
       | regex regex
```

Regular Languages & Finite Automata

- A *regular language* can be expressed via a *regular expression*

```
regex → symbol
      | `(` regex `)`
      | regex `*`
      | regex `|` regex
      | regex regex
```

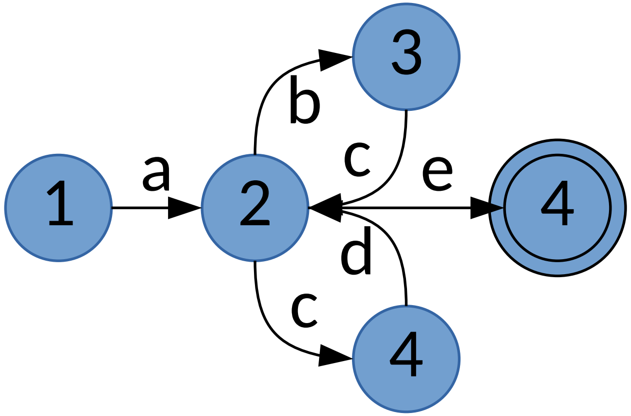
e.g. a(bc | cd)*e defines L containing abccdbce

Regular Languages & Finite Automata

- A *regular language* can be expressed via a *regular expression*
- Finite automata can be used to *recognize* or *generate* elements of a regular language

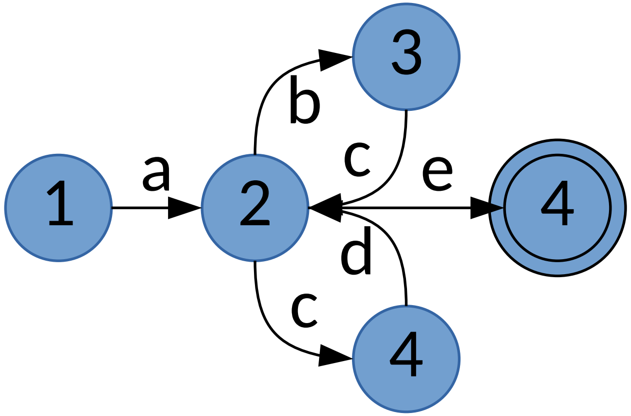
Regular Languages & Finite Automata

- A *regular language* can be expressed via a *regular expression*
- Finite automata can be used to *recognize* or *generate* elements of a regular language



Regular Languages & Finite Automata

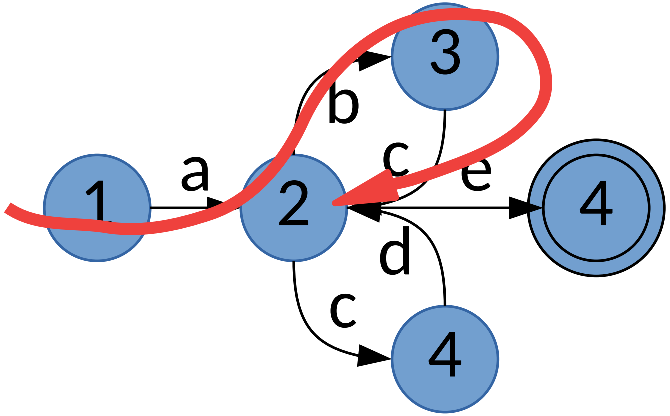
- A *regular language* can be expressed via a *regular expression*
- Finite automata can be used to *recognize* or *generate* elements of a regular language



e.g. a(bc | cd)*e recognizes L containing abccdbce

Regular Languages & Finite Automata

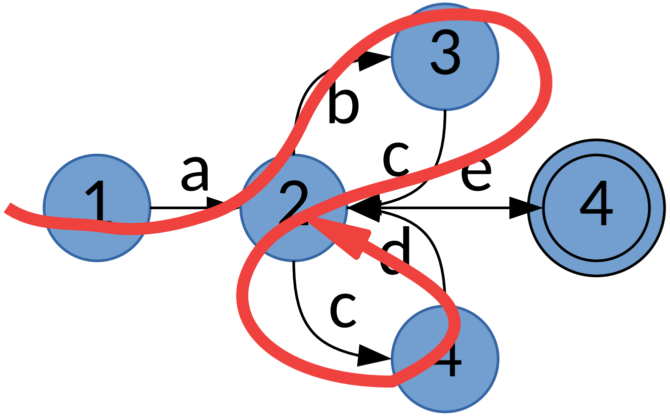
- A *regular language* can be expressed via a *regular expression*
- Finite automata can be used to *recognize* or *generate* elements of a regular language



e.g. a(bc | cd)*e recognizes L containing abccdbce

Regular Languages & Finite Automata

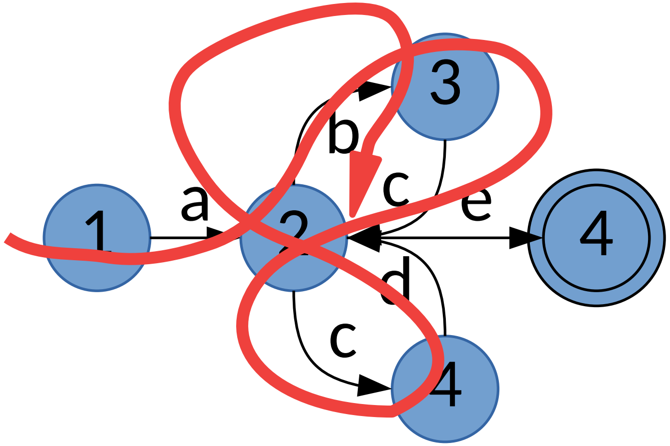
- A *regular language* can be expressed via a *regular expression*
- Finite automata can be used to *recognize* or *generate* elements of a regular language



e.g. a(bc | cd)*e recognizes L containing abccdbce

Regular Languages & Finite Automata

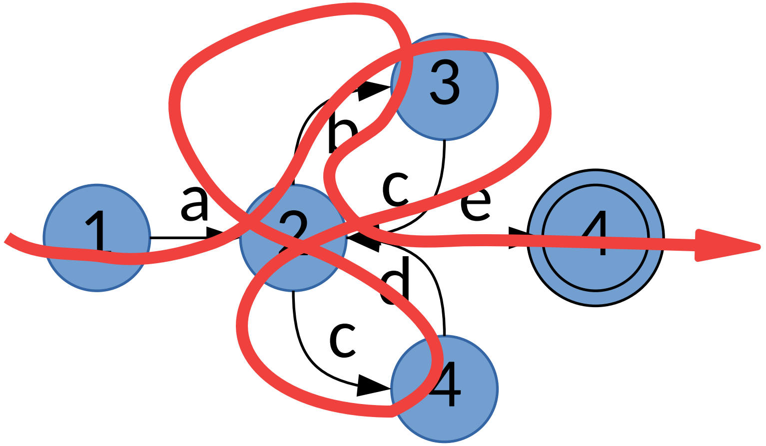
- A *regular language* can be expressed via a *regular expression*
- Finite automata can be used to *recognize* or *generate* elements of a regular language



e.g. a(bc | cd)*e recognizes L containing abccdbce

Regular Languages & Finite Automata

- A *regular language* can be expressed via a *regular expression*
- Finite automata can be used to *recognize* or *generate* elements of a regular language



e.g. a(bc | cd)*e recognizes L containing abccdbce

Regular Languages & Finite Automata

- A *regular language* can be expressed via a *regular expression*
- Finite automata can be used to *recognize* or *generate* elements of a regular language
- Recall, regular languages cannot express matched parentheses (Dyck languages)

$a^n b^n$

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition

Start = A
A → cBd
B → eBf
| g

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition

$$\begin{array}{lll} \text{Start} = A & & \\ A & \rightarrow & cBd \\ B & \rightarrow & eBf \\ & | & g \end{array}$$

$$ce^n g f^n d$$

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition

Start = A
A → cBd
B → eBf
  | g

$ce^n g f^n d$

This requires some kind of *memory*

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition

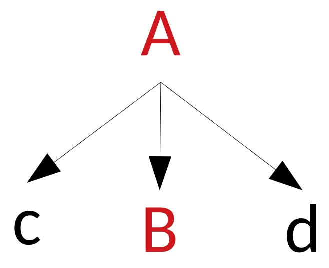
A

Start = A
$A \rightarrow cBd$
$B \rightarrow eBf$
$\quad | \quad g$

ce^ngf^nd

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition



Start = A
A → cBd
B → eBf
  | g

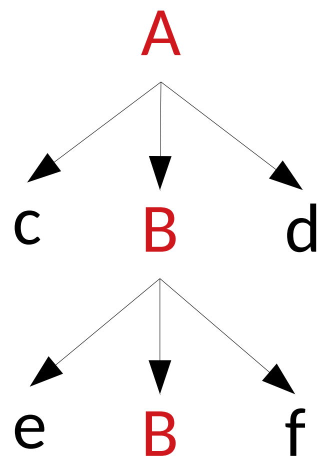
$ce^n gf^n d$

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition

Start = A
A → cBd
B → eBf
| g

ce^ngf^nd

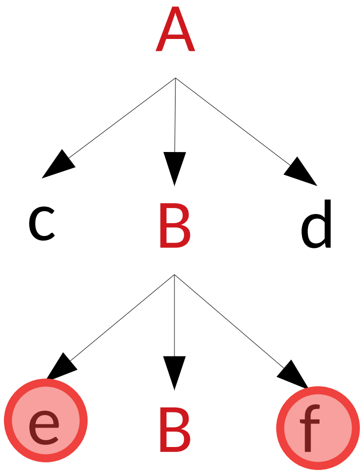


Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition

Start = A
A → cBd
B → eBf
  | g

ce^ngf^nd



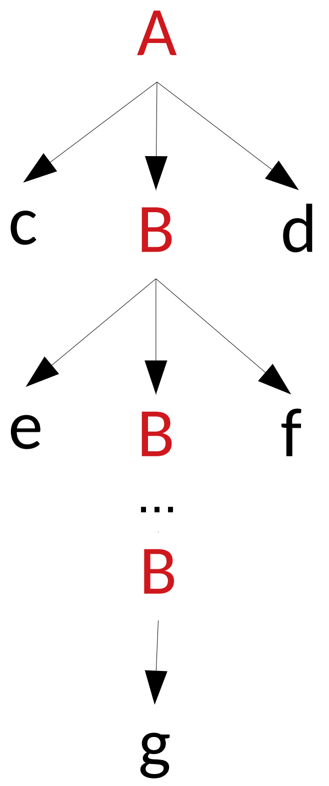
Generating symbols out of order acts as a form of memory.

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition

Start = A
A → cBd
B → eBf
  | g

ce^ngf^nd



Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
  - The grammar for regular expressions was a CFG!

```
regex  → symbol
       | `(` regex `)`
       | regex `*`
       | regex `|` regex
       | regex regex
```

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
  - The grammar for regular expressions was a CFG!

```
regex  → symbol
       | `(` regex `)`
       | regex `*`
       | regex `|` regex
       | regex regex
```

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

Context Free Grammars & Pushdown Automata

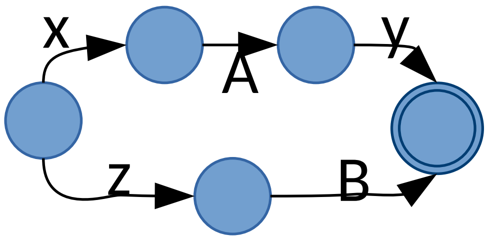
- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

$$
\begin{aligned}
S &\rightarrow xAy \mid zB \\
A &\rightarrow aA \mid t \\
B &\rightarrow bB \mid u
\end{aligned}
$$

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

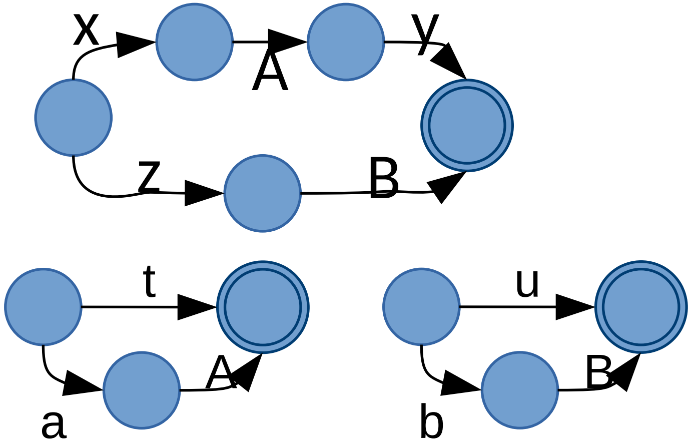
S → xAy | zB
A → aA | t
B → bB | u



Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

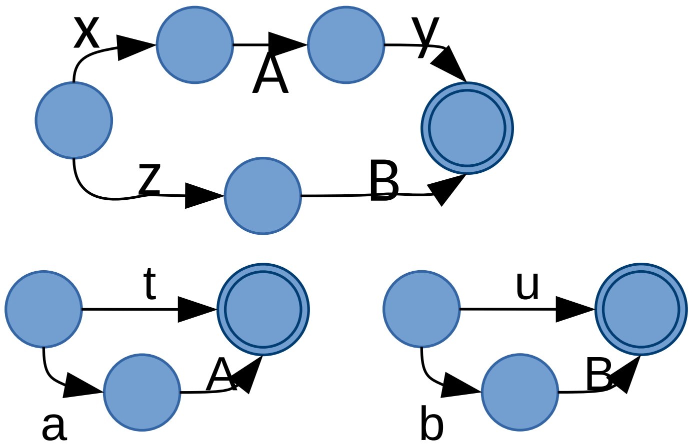
S → xAy | zB
A → aA | t
B → bB | u

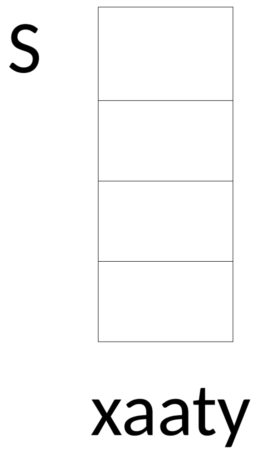


Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

S → xAy | zB
A → aA | t
B → bB | u

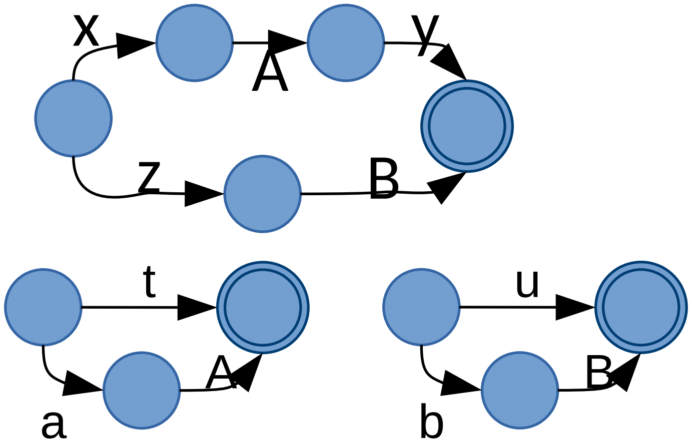


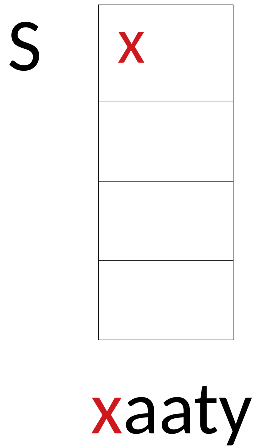


Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

S → xAy | zB
A → aA | t
B → bB | u

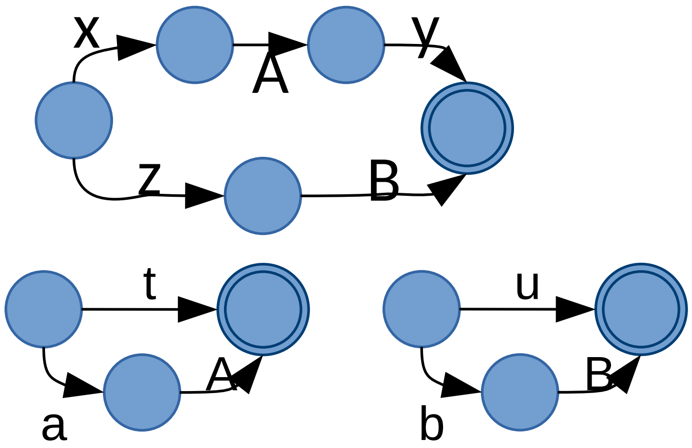


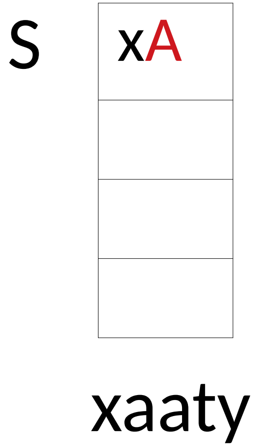


Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

S → xAy | zB
A → aA | t
B → bB | u

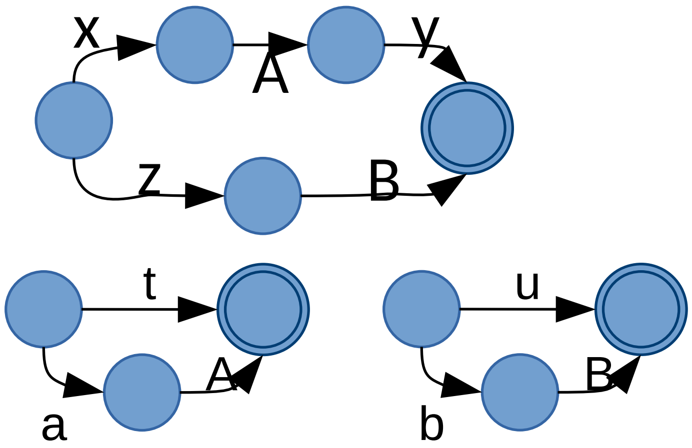


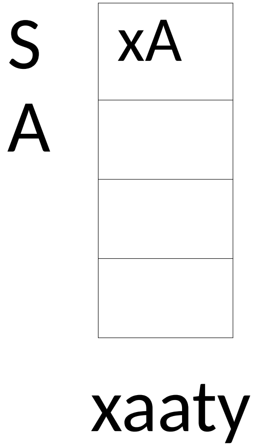


Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

S → xAy | zB
A → aA | t
B → bB | u

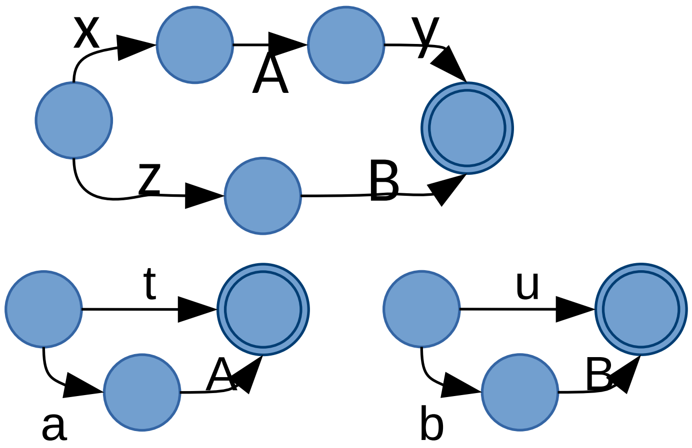




Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

S → xAy | zB
A → aA | t
B → bB | u



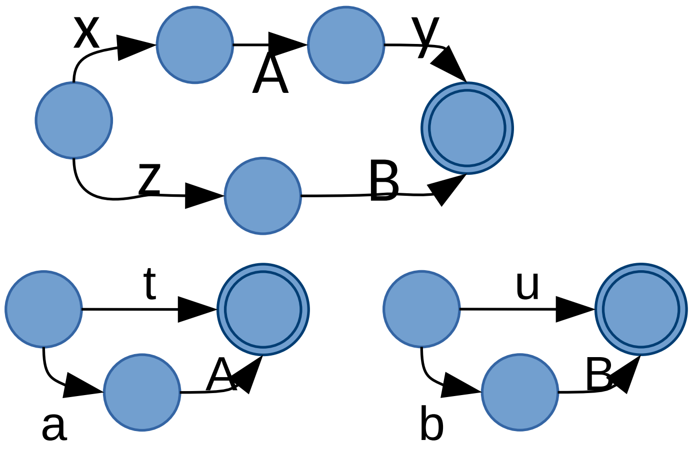
S	xA
A	a

xaaty

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

S → xAy | zB
A → aA | t
B → bB | u



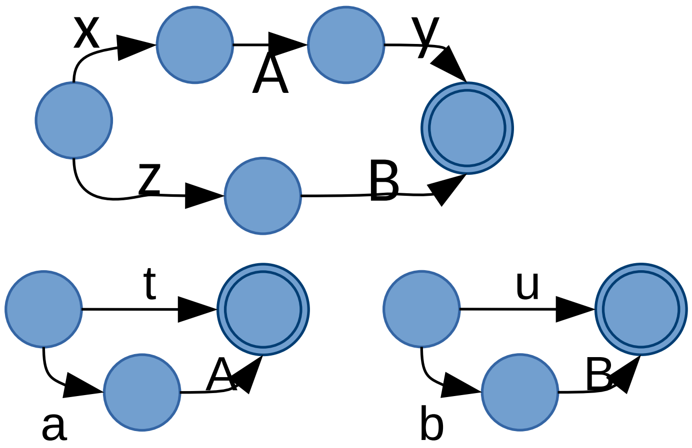
S	xA
A	aA

xaaty

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

$$
\begin{aligned}
S &\rightarrow xAy \mid zB \\
A &\rightarrow aA \mid t \\
B &\rightarrow bB \mid u
\end{aligned}
$$



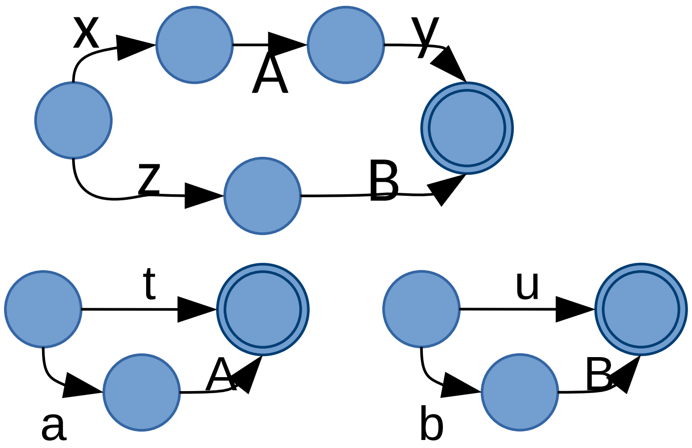
S	xA
A	aA
A	aA
A	t

xaaty

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

S $\rightarrow$ xAy | zB
A $\rightarrow$ aA | t
B $\rightarrow$ bB | u



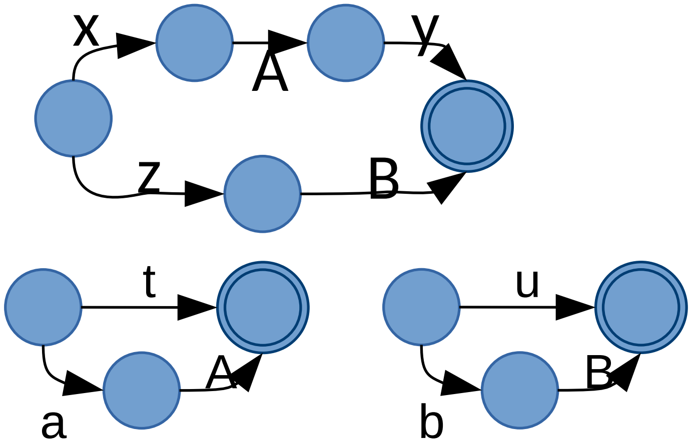
S	xA
A	aA
A	aA

xaaty

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

S → xAy | zB
A → aA | t
B → bB | u



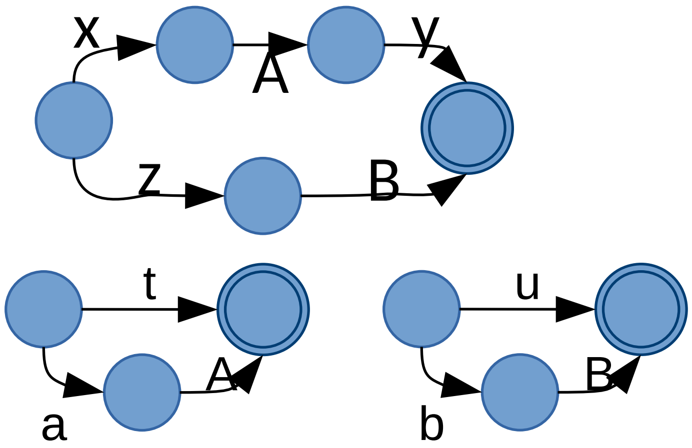
S	xA
A	aA

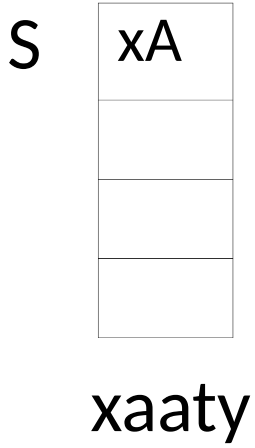
xaaty

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

S → xAy | zB
A → aA | t
B → bB | u

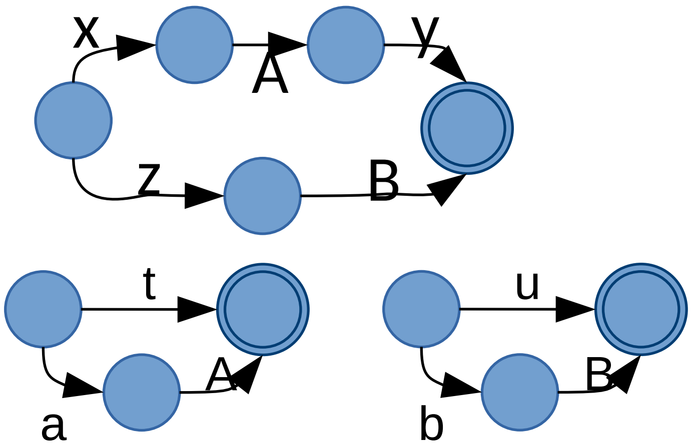


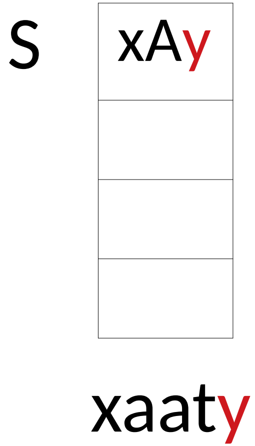


Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

$$
\begin{aligned}
S &\rightarrow xAy \mid zB \\
A &\rightarrow aA \mid t \\
B &\rightarrow bB \mid u
\end{aligned}
$$

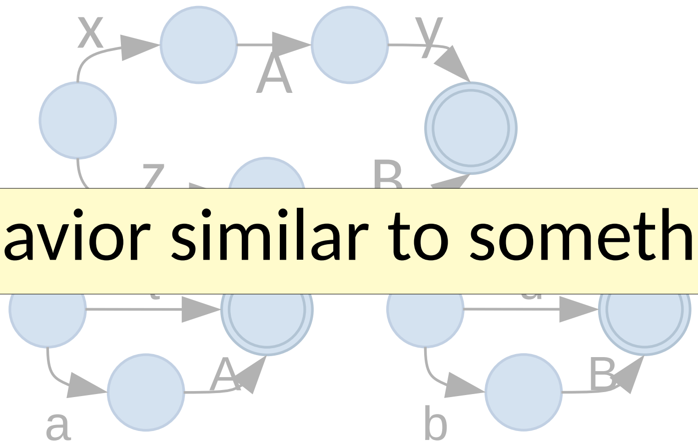


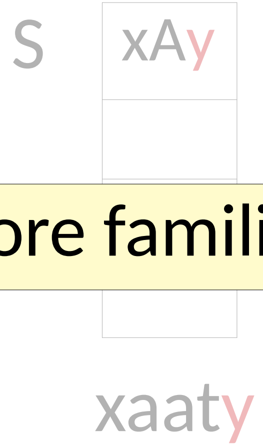


Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)

S → xAy | zB
A →
B →





Is this behavior similar to something more familiar?

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)
- Adding additional rules can extend the expressiveness
  - context sensitive languages
  - tree adjoining grammars
  - …

Context Free Grammars & Pushdown Automata

- *Context free grammars* add recursion and enable Dyck language recognition
- Augmenting a finite automaton with a stack enables recognition and generation (via *pushdown automata*)
- Adding additional rules can extend the expressiveness
- Grammars can constrain far more than strings.
  - graphs
  - semantic objects (furniture layout? sequences of actions? …)

Context Free Grammars & Pushdown Automata

- Context free grammars play a key role in

Context Free Grammars & Pushdown Automata

- Context free grammars play a key role in
  - Precise static program analysis

Context Free Grammars & Pushdown Automata

- Context free grammars play a key role in
  - Precise static program analysis
  - Program synthesis

Context Free Grammars & Pushdown Automata

- Context free grammars play a key role in
  - Precise static program analysis
  - Program synthesis

```
if (e) {
    ...
}
```

Context Free Grammars & Pushdown Automata

- Context free grammars play a key role in
  - Precise static program analysis
  - Program synthesis

Automated Repair

```
if (e) {
    ...
}
```

true
false

Context Free Grammars & Pushdown Automata

- Context free grammars play a key role in
  - Precise static program analysis
  - Program synthesis

Automated Repair

```
if (e) {
    ...
}
```

true
false
{?} == {?}
{?} < {?}
{?} <= {?}

Context Free Grammars & Pushdown Automata

- Context free grammars play a key role in
  - Precise static program analysis
  - Program synthesis

Automated Repair

```
if (e) {
     ...
}
```

true
false
{?} == {?}
{?} < {?}
{?} <= {?}
{?} || {?}
{?} && {?}

Context Free Grammars & Pushdown Automata

- Context free grammars play a key role in
  - Precise static program analysis
  - Program synthesis

Automated Repair

```
if (e) {
    ...
}
```

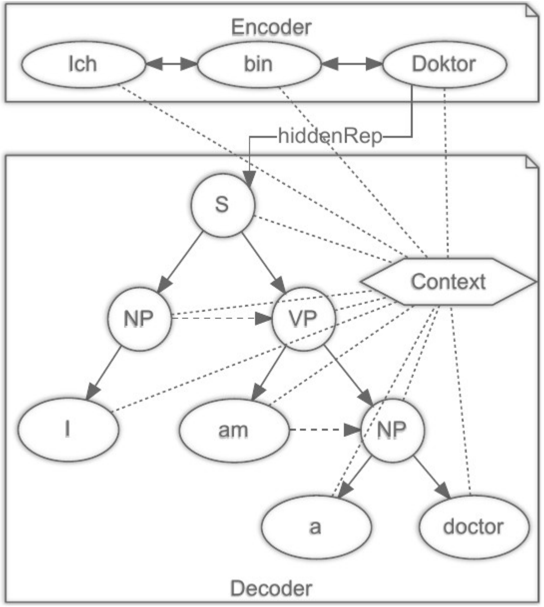
true
false
{?} == {?}
{?} < {?}
{?} <= {?}
{?} || {?}
{?} && {?}
{?} == {?} || {?}
...

Context Free Grammars & Pushdown Automata

- Context free grammars play a key role in
  - Precise static program analysis
  - Program synthesis
  - Prediction and machine learning on programs

Context Free Grammars & Pushdown Automata

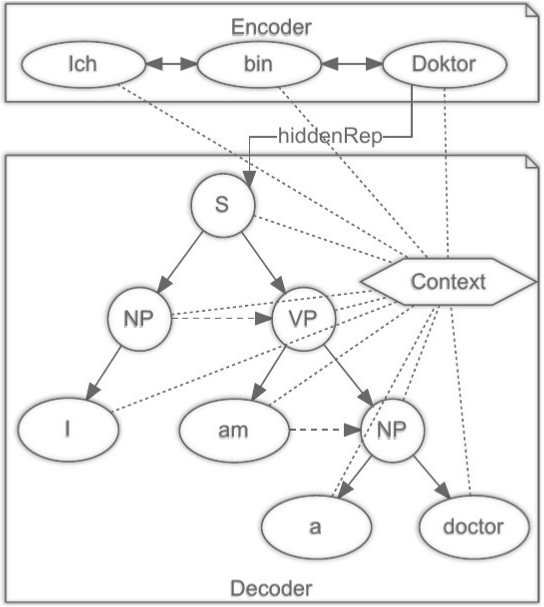
- Context free grammars play a key role in
  - Precise static program analysis
  - Program synthesis
  - Prediction and machine learning on programs



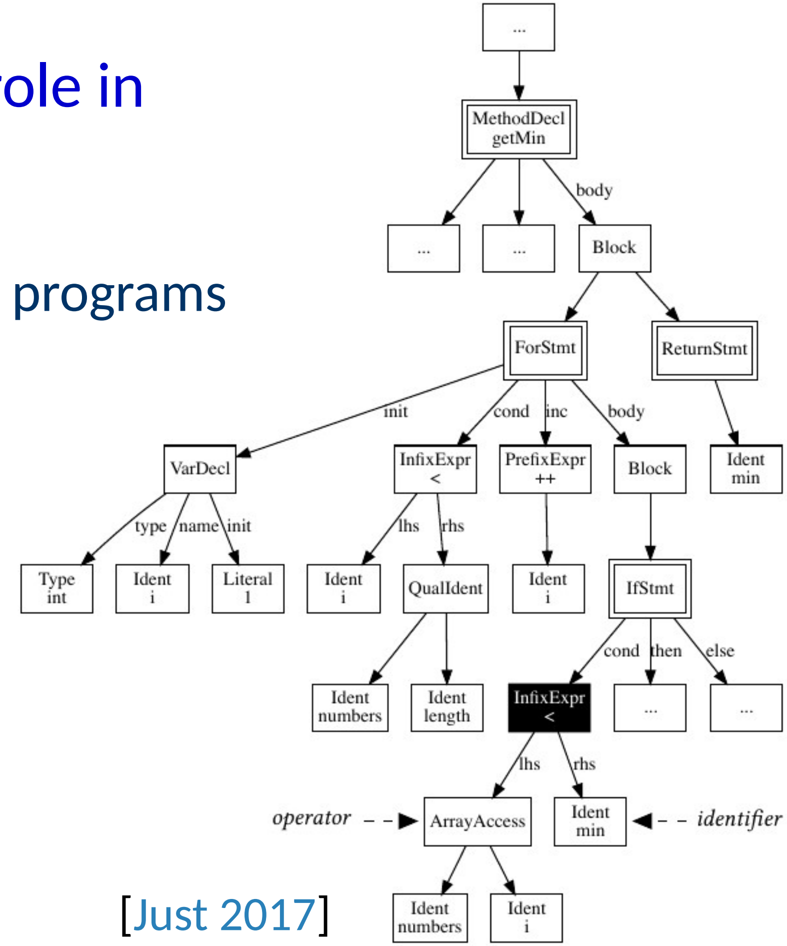
[Gu 2019]

Context Free Grammars & Pushdown Automata

- Context free grammars play a key role in
  - Precise static program analysis
  - Program synthesis
  - Prediction and machine learning on programs



[Gu 2019]



[Just 2017]

Context Free Grammars & Pushdown Automata

- Context free grammars play a key role in
  - Precise static program analysis
  - Program synthesis
  - Prediction and machine learning on programs
  - Compact encodings of complex sets

Formal Logic

Formal Logic

- Formal logic is a systematic approach to reasoning
  - Separate the messy content of an argument from its structure

Formal Logic

- Formal logic is a systematic approach to reasoning
  - Separate the messy content of an argument from its structure
- Sometimes the process can be automated
  - e.g. satisfiability problems, type inference, …

Formal Logic

- Formal logic is a systematic approach to reasoning
  - Separate the messy content of an argument from its structure
- Sometimes the process can be automated
  - e.g. satisfiability problems, type inference, …
- Program analysis has actually been one of the driving forces behind satisfiability in recent years.

Classical Logic

- You likely already know either *propositional* or *first order logic*
  - Systems for reasoning about the truth of sentences

Classical Logic

- You likely already know either *propositional* or *first order logic*
  - Systems for reasoning about the truth of sentences
- Atoms abstract away the actors of the sentences
  - Constants: $\#t$, $\#f$
  - Variables: x, y, z, …

Classical Logic

- You likely already know either *propositional* or *first order logic*
  - Systems for reasoning about the truth of sentences
- Atoms abstract away the actors of the sentences
  - Constants: $\#t$, $\#f$
  - Variables: x, y, z, …
- Connectives relate the atoms & other propositions to each other
  - $\neg$ (Not), $\wedge$ (And), $\vee$ (or)
  - $\rightarrow$ (Implies), $\leftrightarrow$(Iff)

Classical Logic

- You likely already know either *propositional* or *first order logic*
  - Systems for reasoning about the truth of sentences
- Atoms abstract away the actors of the sentences
  - Constants: #t, #f
  - Variables: x, y, z, …
- Connectives relate the atoms & other propositions to each other
  - $\neg$ (Not), $\wedge$ (And), $\vee$ (or)
  - $\rightarrow$ (Implies), $\leftrightarrow$(Iff)

$$\mathbf{x} \wedge \neg\mathbf{y} \wedge \mathbf{z}$$

Classical Logic

- First order logic augments with

Classical Logic

- First order logic augments with
  - Quantifiers- $\exists$ (there exists), $\forall$ (for all)
  - Functions & Relations- e.g. $\mathrm{father(x)},\ \mathrm{Elephant(y)}$

Classical Logic

- First order logic augments with
  - Quantifiers- $\exists$ (there exists), $\forall$ (for all)
  - Functions & Relations- e.g. $\mathrm{father(x)}$, $\mathrm{Elephant(y)}$
- Sentences can be true or false

Classical Logic

- First order logic augments with
  - Quantifiers- $\exists$ (there exists), $\forall$ (for all)
  - Functions & Relations- e.g. $\mathrm{father(x)}$, $\mathrm{Elephant(y)}$
- Sentences can be true or false

$$\forall x(\mathrm{Elephant}(x) \rightarrow \mathrm{Grey}(x))$$

Classical Logic

- First order logic augments with
  - Quantifiers- $\exists$ (there exists), $\forall$ (for all)
  - Functions & Relations- e.g. $\mathrm{father}(x)$, $\mathrm{Elephant}(y)$
- Sentences can be true or false

$$\forall x(\mathrm{Elephant}(x) \rightarrow \mathrm{Grey}(x))$$
$$\forall x(\mathrm{Elephant}(x) \rightarrow \mathrm{Elephant}(\mathrm{father}(x)))$$

Classical Logic

- First order logic augments with
  - Quantifiers- $\exists$ (there exists), $\forall$ (for all)
  - Functions & Relations- e.g. $\mathrm{father(x)}$, $\mathrm{Elephant(y)}$
- Sentences can be true or false
- An interpretation I of the world along with the rules of logic determine truth via judgment ($\vdash$)

Classical Logic

- First order logic augments with
  - Quantifiers- $\exists$ (there exists), $\forall$ (for all)
  - Functions & Relations- e.g. $\mathrm{father}(x)$, $\mathrm{Elephant}(y)$
- Sentences can be true or false
- An interpretation I of the world along with the rules of logic determine truth via judgment ($\vdash$)

$$I \vdash x \text{ and } I \vdash y \text{ iff } I \vdash x \wedge y$$

Classical Logic

- *Satisfiability*
  - A sentence s is satisfiable $\leftrightarrow$ $\exists \mathrm{I}\ (\mathrm{I} \vdash \mathrm{s})$

Classical Logic

- *Satisfiability*
  - A sentence s is satisfiable $\leftrightarrow$ $\exists I\ (I \vdash s)$
- *Validity*
  - A sentence s is valid $\leftrightarrow$ $\forall I\ (I \vdash s)$

Classical Logic

- *Satisfiability*
  - A sentence s is satisfiable $\leftrightarrow$ $\exists \mathrm{I}\ (\mathrm{I} \vdash \mathrm{s})$
- *Validity*
  - A sentence s is valid $\leftrightarrow$ $\forall \mathrm{I}\ (\mathrm{I} \vdash \mathrm{s})$
- We will see later how these can be used for a wide variety of tasks

Classical Logic

- *Satisfiability*
  - A sentence s is satisfiable $\leftrightarrow$ $\exists I\ (I \vdash s)$
- *Validity*
  - A sentence s is valid $\leftrightarrow$ $\forall I\ (I \vdash s)$
- We will see later how these can be used for a wide variety of tasks
  - Bug finding
  - Model checking (proving correctness)
  - Explaining defects
  - …

Inference using classical logic

- Rules express how some judgments enable others

$$\frac{\Gamma \vdash x \qquad \Delta \vdash y}{\Gamma, \Delta \vdash x \wedge y}$$

Inference using classical logic

- Rules express how some judgments enable others

$$\frac{\boxed{\Gamma \vdash x \qquad \Delta \vdash y}}{\Gamma, \Delta \vdash x \wedge y}$$

Inference using classical logic

- Rules express how some judgments enable others

$$\frac{\Gamma \vdash x \qquad \Delta \vdash y}{\boxed{\Gamma, \Delta \vdash x \wedge y}}$$

Inference using classical logic

- Rules express how some judgments enable others

$$\frac{\Gamma \vdash x \qquad \Delta \vdash y}{\Gamma, \Delta \vdash x \wedge y}$$

- Proofs can be written by stacking rules

Inference using classical logic

- Rules express how some judgments enable others

$$\frac{\Gamma \vdash \mathrm{x} \qquad \Delta \vdash \mathrm{y}}{\Gamma, \Delta \vdash \mathrm{x} \wedge \mathrm{y}}$$

- Proofs can be written by stacking rules

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\mathrm{Id} \qquad \dfrac{\dfrac{}{A \multimap B \vdash A \multimap B}\,\mathrm{Id} \qquad \dfrac{}{A \vdash A}\,\mathrm{Id}}{A \multimap B,\, A \vdash B}\,{\multimap}\text{-E}}{A,\, A \multimap B,\, A \vdash A \times B}\,{\times}\text{-I}}{A \multimap B,\, A,\, A \vdash A \times B}\,\text{Exchange}}{A \multimap B,\, A \vdash A \times B}\,\text{Contraction}$$

Wadler, “A Taste of Linear Logic”. 2014.

Intuitionistic & Constructive Logic

- It can be useful to modify or limit rules of inference

Intuitionistic & Constructive Logic

- It can be useful to modify or limit rules of inference
  - Suppose a compiler cannot prove variable x is an int.
    Is it reasonable for the compile to assume x is a string?

Intuitionistic & Constructive Logic

- It can be useful to modify or limit rules of inference
  - Suppose a compiler cannot prove variable x is an int.
    Is it reasonable for the compile to assume x is a string?
- *Constructivism* argues that truth comes from direct *evidence*.
  - We cannot merely assume p or not p, we must have evidence

Intuitionistic & Constructive Logic

- It can be useful to modify or limit rules of inference
  - Suppose a compiler cannot prove variable x is an int.
    Is it reasonable for the compile to assume x is a string?
- *Constructivism* argues that truth comes from direct *evidence*.
  - We cannot merely assume p or not p, we must have evidence
- *Intuitionistic logic* restricts the rules of inference to require direct evidence

Intuitionistic & Constructive Logic

- Classic logic includes several rules including

Intuitionistic & Constructive Logic

- Classic logic includes several rules including

$$\frac{}{\vdash p \lor \neg p}$$

Law of excluded middle

Intuitionistic & Constructive Logic

- Classic logic includes several rules including

$$\frac{}{\vdash p \lor \neg p}$$

$$\frac{\Gamma \vdash \neg\neg p}{\Gamma \vdash p}$$

Double negation elimination

Intuitionistic & Constructive Logic

- Classic logic includes several rules including

$$\frac{}{\vdash p \vee \neg p} \qquad \frac{\Gamma \vdash \neg\neg p}{\Gamma \vdash p}$$

- ***Intuitionistic logic*** excludes these to require direct evidence

Intuitionistic & Constructive Logic

- Classic logic includes several rules including

$$\frac{}{\vdash p \vee \neg p} \qquad \frac{\Gamma \vdash \neg\neg p}{\Gamma \vdash p}$$

- ***Intuitionistic logic*** excludes these to require direct evidence
- Note, this is commonly used in *type systems*

Linear & Substructural Logic

$$\begin{array}{l}\mathrm{sellsBurritos(store)}\\ \mathrm{has10Dollars(me)}\end{array} \vdash \mathrm{buyBurrito(me,store)}$$

Linear & Substructural Logic

$$\begin{array}{l}\text{sellsBurritos}(\text{store}) \\ \text{has10Dollars}(\text{me})\end{array} \vdash \begin{array}{l}\text{buyBurrito}(\text{me},\text{store}) \\ \wedge\ \text{buyBurrito}(\text{me},\text{store})\end{array}$$

Linear & Substructural Logic

$$\begin{array}{l} \text{sellsBurritos(store)} \\ \text{has10Dollars(me)} \end{array} \vdash \begin{array}{l} \text{buyBurrito(me,store)} \\ \land\ \text{buyBurrito(me,store)} \\ \land\ \text{buyBurrito(me,store)} \end{array}$$

Linear & Substructural Logic

$$\begin{array}{l} \text{sellsBurritos(store)} \\ \text{has10Dollars(me)} \end{array} \vdash \begin{array}{l} \text{buyBurrito(me,store)} \\ \wedge\ \text{buyBurrito(me,store)} \\ \wedge\ \text{buyBurrito(me,store)} \\ \wedge\ \text{buyBurrito(me,store)} \end{array}$$

Linear & Substructural Logic

$$\begin{array}{l} \text{sellsBurritos(store)} \\ \text{has10Dollars(me)} \end{array} \vdash \begin{array}{l} \text{buyBurrito(me,store)} \\ \wedge\ \text{buyBurrito(me,store)} \\ \wedge\ \text{buyBurrito(me,store)} \\ \wedge\ \text{buyBurrito(me,store)} \end{array}$$

Classical & intuitionistic logic
have trouble expressing consumable facts

Linear & Substructural Logic

$$\begin{array}{l}\text{sellsBurritos(store)}\\ \text{has10Dollars(me)}\end{array} \vdash \text{buyBurrito(me,store)}$$

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - <Linear> cannot be used with contraction or weakening

Linear & Substructural Logic

$$\begin{array}{l}\text{sellsBurritos(store)}\\ \text{has10Dollars(me)}\end{array} \vdash \text{buyBurrito(me,store)}$$

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - <Linear> cannot be used with contraction or weakening

$$\frac{\Gamma, A, A, \Delta \vdash p}{\Gamma, A, \Delta \vdash p} \qquad \frac{\Gamma, \Delta \vdash p}{\Gamma, A, \Delta \vdash p}$$

Linear & Substructural Logic

$$\begin{array}{l}\mathrm{sellsBurritos(store)} \\ \mathrm{has10Dollars(me)}\end{array} \vdash \mathrm{buyBurrito(me,store)}$$

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - <Linear> cannot be used with contraction or weakening
  - In essence, linear facts must be consumed *exactly once* in a proof.

$$\frac{\Gamma, A, A, \Delta \vdash p}{\Gamma, A, \Delta \vdash p} \qquad \frac{\Gamma, \Delta \vdash p}{\Gamma, A, \Delta \vdash p}$$

Linear & Substructural Logic

$$\begin{array}{l}\mathrm{sellsBurritos(store)} \\ \mathrm{has10Dollars(me)}\end{array} \vdash \mathrm{buyBurrito(me,store)}$$

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - <Linear> cannot be used with contraction or weakening
  - In essence, linear facts must be consumed *exactly once* in a proof.

$$\frac{\Gamma, A, A, \Delta \vdash p}{\Gamma, A, \Delta \vdash p} \qquad \frac{\Gamma, \Delta \vdash p}{\Gamma, A, \Delta \vdash p}$$

Idea: Some facts (resources) require careful accounting

Linear & Substructural Logic

$$\begin{array}{l}\text{sellsBurritos(store)} \\ \text{has10Dollars(me)}\end{array} \vdash \text{buyBurrito(me,store)}$$

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - <Linear> cannot be used with contraction or weakening
  - In essence, linear facts must be consumed exactly once in a proof.

Logics that remove additional rules
from intuitionistic logic are *substructural*

Linear & Substructural Logic

$$\begin{matrix}\text{sellsBurritos(store)} \\ \text{has10Dollars(me)}\end{matrix} \vdash \text{buyBurrito(me,store)}$$

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - <Linear> cannot be used with contraction or weakening
  - In essence, linear facts must be consumed exactly once in a proof.
- This forms the backbone of *ownership types* in languages like Rust!

Linear & Substructural Logic

$$\begin{array}{l}\mathrm{sellsBurritos(store)} \\ \mathrm{has10Dollars(me)}\end{array} \vdash \mathrm{buyBurrito(me,store)}$$

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - <Linear> cannot be used with contraction or weakening
  - In essence, linear facts must be consumed exactly once in a proof.

- This forms the backbone of *ownership types* in languages like Rust!

```
struct Thing(u32);
let a = Thing(5);
let b = a;
let c = a;
```

Linear & Substructural Logic

$$\begin{array}{l}\text{sellsBurritos(store)}\\ \text{has10Dollars(me)}\end{array} \vdash \text{buyBurrito(me,store)}$$

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - <Linear> cannot be used with contraction or weakening
  - In essence, linear facts must be consumed exactly once in a proof.
- This forms the backbone of *ownership types* in languages like Rust!

```
struct Thing(u32);
let a = Thing(5);
let b = a;
let c = a;
```

⊢ a:Thing

Linear & Substructural Logic

$$\begin{array}{l}\text{sellsBurritos(store)}\\ \text{has10Dollars(me)}\end{array} \vdash \text{buyBurrito(me,store)}$$

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - <Linear> cannot be used with contraction or weakening
  - In essence, linear facts must be consumed exactly once in a proof.
- This forms the backbone of *ownership types* in languages like Rust!

```
struct Thing(u32);
let a = Thing(5);
let b = a;
let c = a;
```

$\vdash$ a:Thing

a:Thing $\vdash$ b:Thing

Linear & Substructural Logic

$$\begin{array}{l}\text{sellsBurritos(store)} \\ \text{has10Dollars(me)}\end{array} \vdash \text{buyBurrito(me,store)}$$

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - <Linear> cannot be used with contraction or weakening
  - In essence, linear facts must be consumed exactly once in a proof.

- This forms the backbone of *ownership types* in languages like Rust!

```
struct Thing(u32);
let a = Thing(5);
let b = a;
let c = a;
```

$\vdash$ a:Thing

~~a:Thing~~ $\vdash$ b:Thing

Linear & Substructural Logic

$$\begin{array}{l}\mathrm{sellsBurritos}(\mathrm{store})\\ \mathrm{has10Dollars}(\mathrm{me})\end{array} \vdash \mathrm{buyBurrito}(\mathrm{me},\mathrm{store})$$

- Linear logic denotes separates facts into two kinds
  - [Intuitionistic] as before
  - <Linear> cannot be used with contraction or weakening
  - In essence, linear facts must be consumed exactly once in a proof.
- This forms the backbone of *ownership types* in languages like Rust!

```
struct Thing(u32);
let a = Thing(5);
let b = a;
let c = a;
```

$\vdash$ a:Thing

$\cancel{\text{a:Thing}} \vdash$ b:Thing

Error ($\vdash$ c:?)

Hoare Logic

- Given facts, the logics we have seen consider what is true/false

Hoare Logic

- Given facts, the logics we have seen consider what is true/false

$$x \wedge \neg y \wedge z$$

Hoare Logic

- Given facts, the logics we have seen consider what is true/false

$$x \wedge \neg y \wedge z$$

- Programs reason about *facts that change* over time

Hoare Logic

- Given facts, the logics we have seen consider what is true/false

$$x \land \neg y \land z$$

- Programs reason about *facts that change* over time
  - How do facts at one state affect facts at another?

Hoare Logic

- Given facts, the logics we have seen consider what is true/false

$$x \land \neg y \land z$$

- Programs reason about *facts that change* over time
  - How do facts at one state affect facts at another?

```
double sqrt(double n,
            double threshold) {
  double x = 1;
  while (true) {
    double newX = (x + n/x) / 2;
    if (abs(x - nx) < threshold)
      break;
    x = nx
  }
  return x;
}
```

Hoare Logic

- Given facts, the logics we have seen consider what is true/false

$$x \land \neg y \land z$$

- Programs reason about *facts that change* over time
  - How do facts at one state affect facts at another?
  - Does this do what is expected?

```
double sqrt(double n,
            double threshold) {
  double x = 1;
  while (true) {
    double newX = (x + n/x) / 2;
    if (abs(x - nx) < threshold)
      break;
    x = nx
  }
  return x;
}
```

Hoare Logic

- Given facts, the logics we have seen consider what is true/false

$$x \wedge \neg y \wedge z$$

- Programs reason about *facts that change* over time
  - How do facts at one state affect facts at another?
  - Does this do what is expected?
  - Will I dereference a null pointer?

```
double sqrt(double n,
            double threshold) {
  double x = 1;
  while (true) {
    double newX = (x + n/x) / 2;
    if (abs(x - nx) < threshold)
      break;
    x = nx
  }
  return x;
}
```

```
y = w[20]
x = *y + 5
```

Hoare Logic

- Given facts, the logics we have seen consider what is true/false

$$x \land \neg y \land z$$

- Programs reason about *facts that change* over time
  - How do facts at one state affect facts at another?
  - Does this do what is expected?
  - Will I dereference a null pointer?

```
double sqrt(double n,
            double threshold) {
  double x = 1;
  while (true) {
    double newX = (x + n/x) / 2;
    if (abs(x - nx) < threshold)
      break;
    x = nx
  }
  return x;
```

```
y = w[20]
x = *y + 5
```

We want a logic that reasons about changes in state.

Hoare Logic

- *Hoare logic* reasons about the behavior of programs and program fragments

Hoare Logic

- *Hoare logic* reasons about the behavior of programs and program fragments

$$\boxed{\{\varphi\}}\, C \{\psi\}$$

Precondition

Hoare Logic

- *Hoare logic* reasons about the behavior of programs and program fragments

$$\{\varphi\}\boxed{C}\{\psi\}$$

Precondition

Command

Hoare Logic

- *Hoare logic* reasons about the behavior of programs and program fragments

$$\{\varphi\}\,C\,\boxed{\{\psi\}}$$

Precondition

Command

Postcondition

Hoare Logic

- *Hoare logic* reasons about the behavior of programs and program fragments

$$\{\varphi\}C\{\psi\}$$

- If φ holds before C, ψ will hold after

$$\{x=3 \land y=2\}x \leftarrow 5\{x=5\}$$

Hoare Logic

- *Hoare logic* reasons about the behavior of programs and program fragments

$$\{\varphi\}C\{\psi\}$$

- If φ holds before C, ψ will hold after

$$\{x=3 \wedge y=2\}x \leftarrow 5\{x=5\}$$

- A ***weakest precondition*** $\mathrm{wp}(C,\psi)$ captures all states leading to ψ after C.

Hoare Logic

- *Hoare logic* reasons about the behavior of programs and program fragments

$$\{\varphi\}C\{\psi\}$$

- If φ holds before C, ψ will hold after

$$\{x=3 \land y=2\}x \leftarrow 5\{x=5\}$$

- A ***weakest precondition*** $wp(C,\psi)$ captures all states leading to ψ after C.

$$\{\#t\}x\leftarrow5\{x=5\}$$

Hoare Logic

- *Hoare logic* reasons about the behavior of programs and program fragments

$$\{\varphi\}C\{\psi\}$$

- If φ holds before C, ψ will hold after

$$\{x=3 \land y=2\}\texttt{x} \leftarrow \texttt{5}\{x=5\}$$

- A ***weakest precondition*** $\mathrm{wp}(C,\psi)$ captures all states leading to ψ after C.

$$\{\#t\}\texttt{x}\leftarrow\texttt{5}\{x=5\}$$

$$\{???\}\texttt{if c then x}\leftarrow\texttt{5}\{x=5\}$$

Hoare Logic

- *Hoare logic* reasons about the behavior of programs and program fragments

$$\{\varphi\}C\{\psi\}$$

- If φ holds before C, ψ will hold after

$\{x=3 \land$

You already have an *intuition* for weakest preconditions

- A ***weakest precondition*** $\mathrm{wp}(C,\psi)$ captures all states leading to ψ after C.

$$\{\#t\}x\leftarrow 5\{x=5\}$$

$$\{???\}\texttt{if c then x}\leftarrow 5\{x=5\}$$

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- A store/state σ is a partial function mapping variables to values

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- A store/state σ is a partial function mapping variables to values
  - Commands in a program can modify the store

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- A store/state σ is a partial function mapping variables to values
  - Commands in a program can modify the store

Command

$x \leftarrow 5$

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- A store/state σ is a partial function mapping variables to values
  - Commands in a program can modify the store

Store	Command
$\sigma = \{x \mapsto 3,\ y \mapsto 1\}$	
	x ← 5

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- A store/state σ is a partial function mapping variables to values
  - Commands in a program can modify the store

Store	Command
$\sigma = \{x \mapsto 3,\ y \mapsto 1\}$	
	x ← 5
$\sigma = \{\mathbf{x \mapsto 5},\ y \mapsto 1\}$	

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- A store/state σ is a partial function mapping variables to values
  - Commands in a program can modify the store

Store	Command	Conditions
$\sigma=\{x\mapsto 3,\ y\mapsto 1\}$		
	x ← 5	
$\sigma=\{x\mapsto 5,\ y\mapsto 1\}$		$\{x=5\}$

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- A store/state σ is a partial function mapping variables to values
  - Commands in a program can modify the store

Store	Command	Conditions
$\sigma=\{x\mapsto 3,\ y\mapsto 1\}$		$\{x=3 \wedge y=\overset{?}{2}\}$
	x ← 5	
$\sigma=\{x\mapsto 5,\ y\mapsto 1\}$		$\{x=5\}$

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

- A store/state σ is a partial function mappi
  - Commands in a program can modify the sto

This was *technically* true,
but not so useful
(…or even compatible with our states)

Store	Command	Conditions
$\sigma=\{x\mapsto 3,\ y\mapsto 1\}$		$\{x=3 \wedge y=2\}$
	x ← 5	
$\sigma=\{x\mapsto 5,\ y\mapsto 1\}$		$\{x=5\}$

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- A store/state σ is a partial function mapping variables to values
  - Commands in a program can modify the store

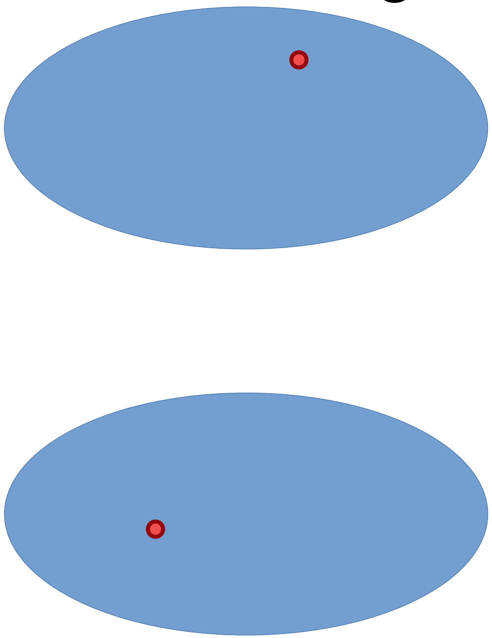
Store	Command	Conditions
$\sigma=\{x\mapsto 3,\ y\mapsto 1\}$		$\{x=3 \wedge y=2\}$
	$x \leftarrow 5$	
$\sigma=\{x\mapsto 5,\ y\mapsto 1\}$		$\{x=5\}$

  - **$\sigma \in \Sigma$**(all possible states), and we can reason about *subsets* of Σ

Hoare Logic – weakest preconditions

- What do we really mean by *captures all states*?

$\sigma = \{x \mapsto 3,\ y \mapsto 1\}$



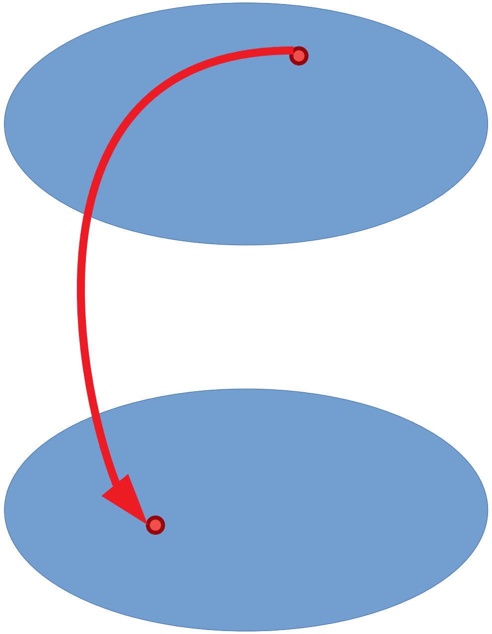
x ← 5

$\sigma = \{x \mapsto 5,\ y \mapsto 1\}$

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

$\sigma = \{x \mapsto 3,\ y \mapsto 1\}$

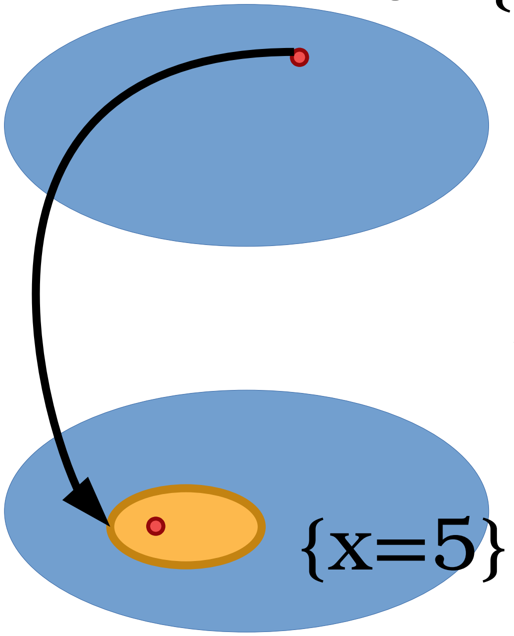


$x \leftarrow 5$

$\sigma = \{\mathbf{x \mapsto 5},\ y \mapsto 1\}$

Hoare Logic – weakest preconditions

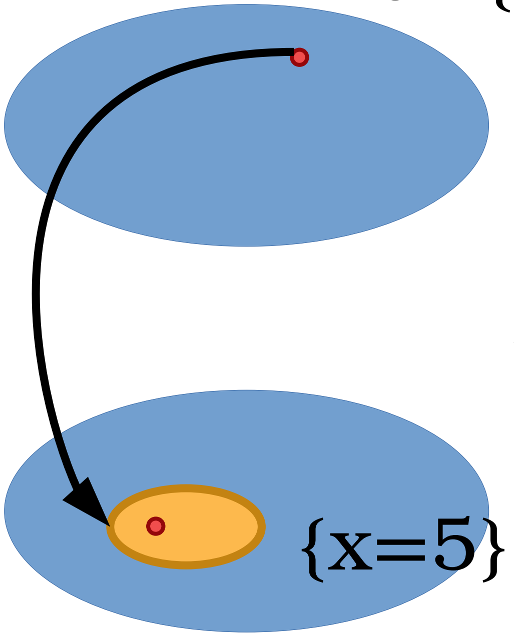
- What do we really mean by captures all states?



Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

$\sigma = \{x \mapsto 3,\ y \mapsto 1\}$



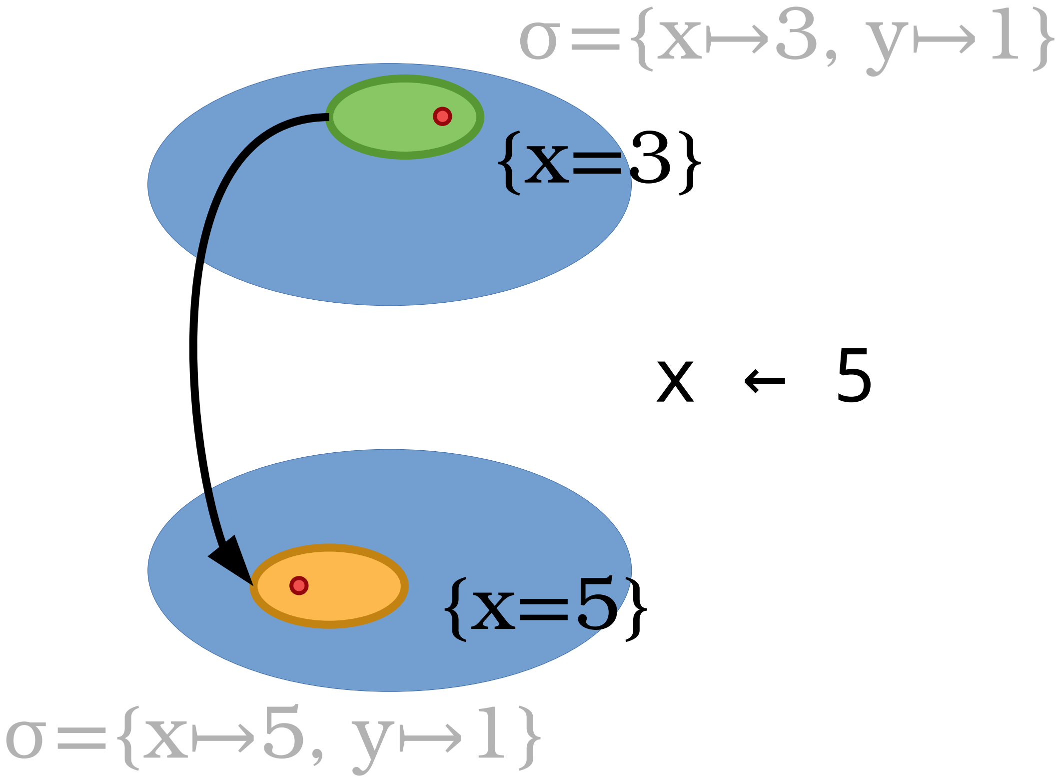
x ← 5

Each set of states corresponds to a condition defining the set

$\sigma = \{x \mapsto 5,\ y \mapsto 1\}$

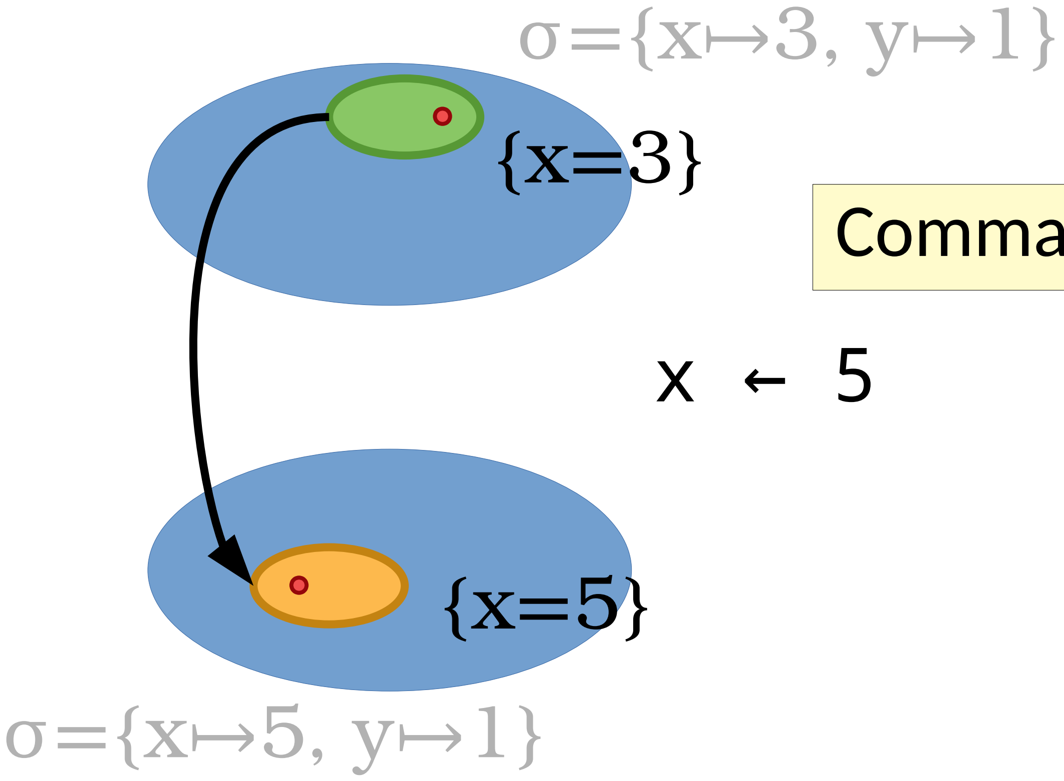
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?



Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

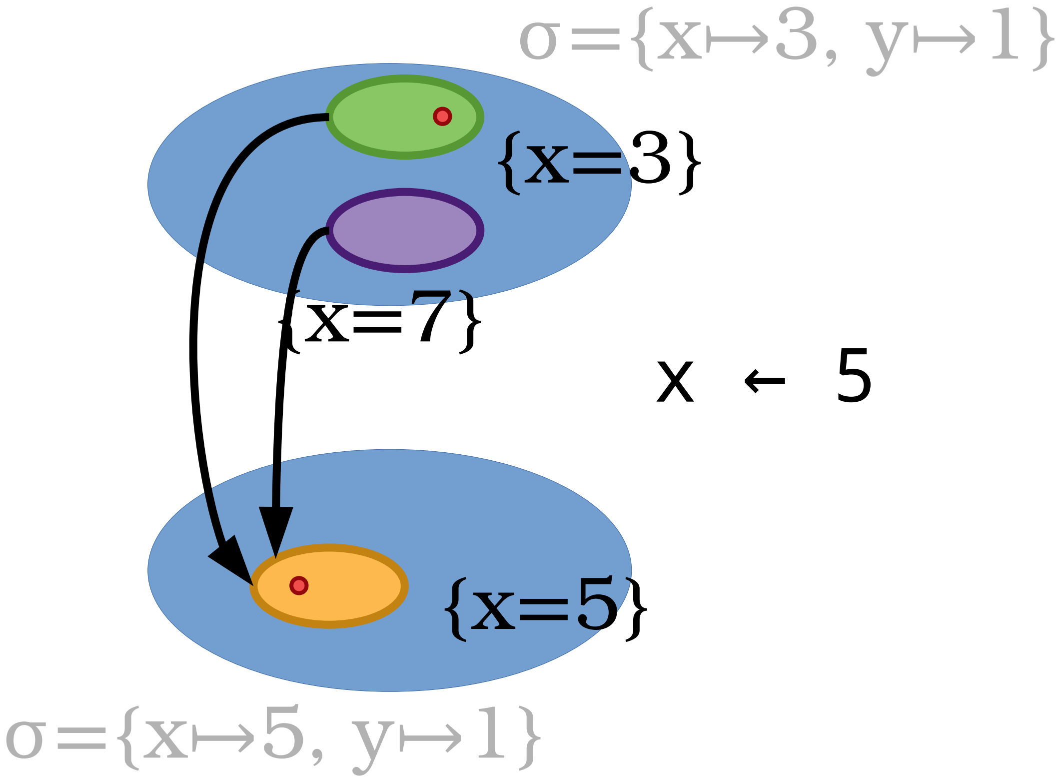


Commands map sets to sets

x ← 5

Hoare Logic – weakest preconditions

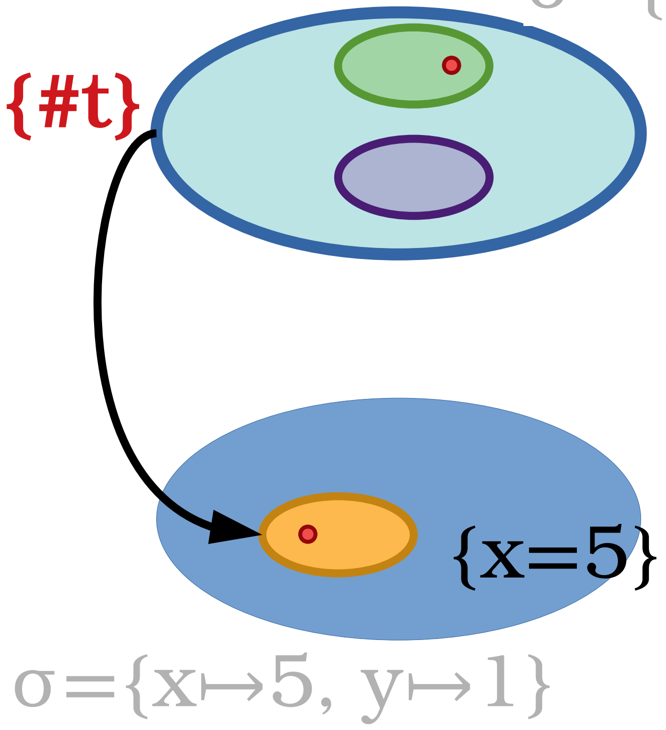
- What do we really mean by captures all states?



Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

$\sigma=\{x\mapsto 3,\ y\mapsto 1\}$

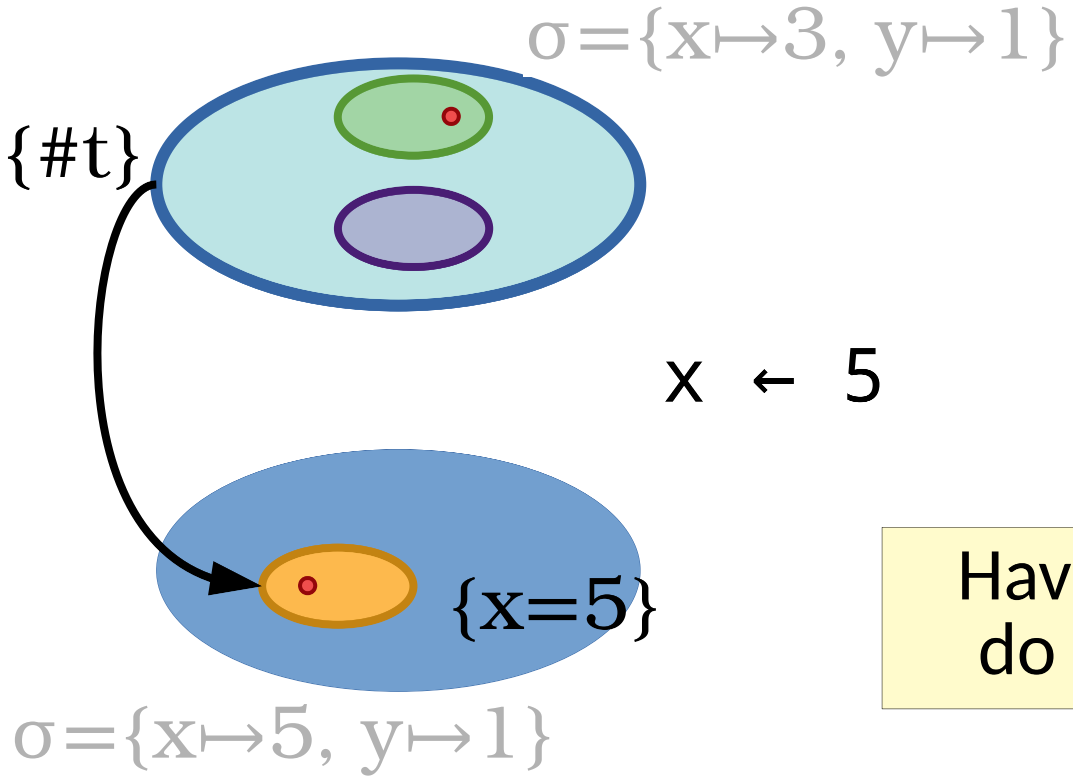


All states lead
to the postcondition!

x ← 5

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?



Have we already seen a way do describe this structure?

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

Hoare Logic – weakest preconditions

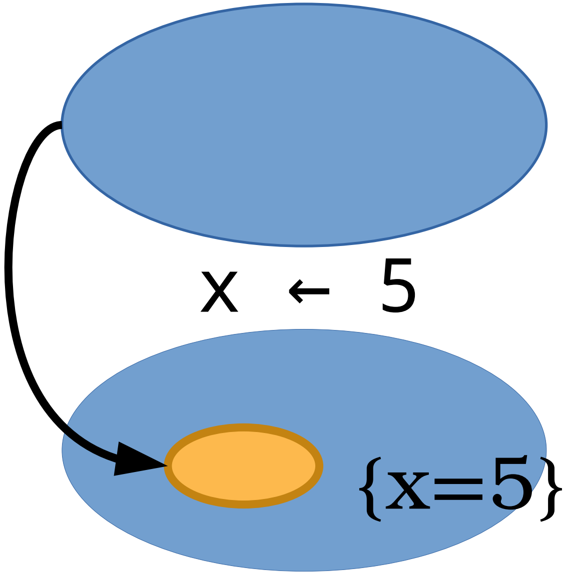
- What do we really mean by captures all states?
- $wp(C, \psi) = \bigsqcup \{x \mid \{x\}\ C\ \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A < B)$

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- $\boxed{wp(C,\psi)} = \boxed{\bigsqcup \{x \mid \boxed{\{x\}\ C\ \{\psi\}}\}}$
  - Where $(A \rightarrow B) \vdash (A < B)$

Hoare Logic – weakest preconditions

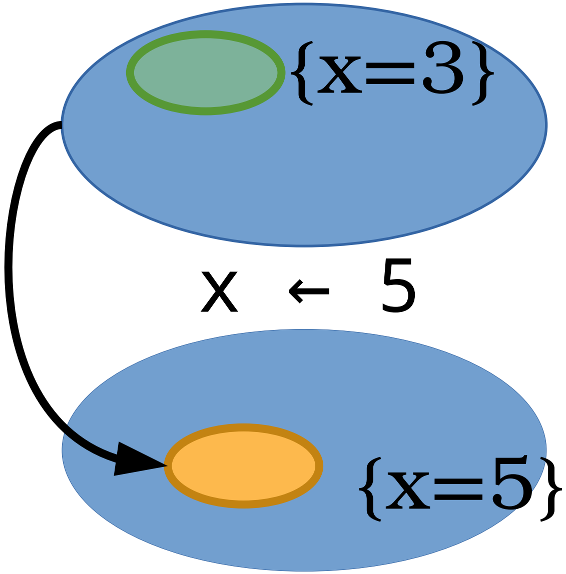
- What do we really mean by captures all states?
- $\boxed{wp(C, \psi)} = \boxed{\bigsqcup \{x \mid \boxed{\{x\}\ C\ \{\psi\}}\}}$
  - Where $(A \rightarrow B) \vdash (A < B)$



$\psi = \{x=5\}$

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- $\boxed{wp(C,\psi)} = \boxed{\bigsqcup \{x \mid \boxed{\{x\}\ C\ \{\psi\}}\}}$
  - Where $(A \rightarrow B) \vdash (A < B)$

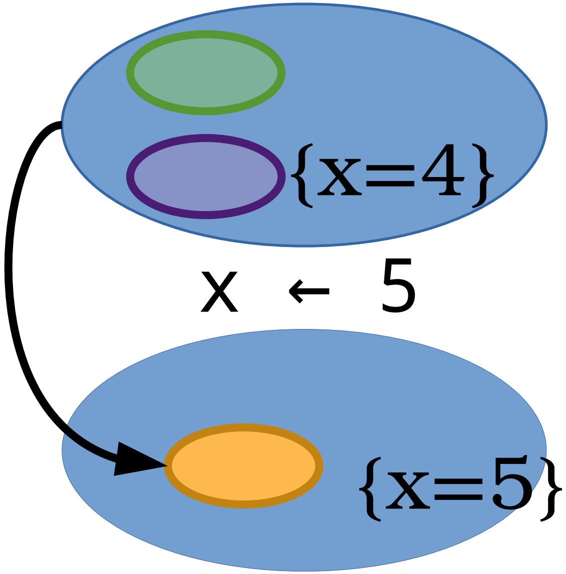


$\{x=3\}$

$\psi = \{x=5\}$

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- $\boxed{wp(C,\psi)} = \boxed{\bigsqcup \{x \mid \boxed{\{x\}\ C\ \{\psi\}}\}}$
  - Where $(A \rightarrow B) \vdash (A<B)$

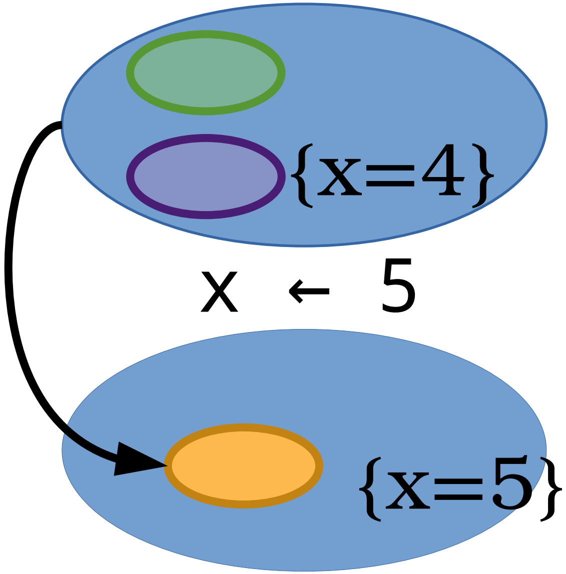


$\{x=3\} \bigsqcup \{x=4\}$
$= ?$

$\psi = \{x=5\}$

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- $\boxed{wp(C,\psi)} = \boxed{\bigsqcup \{x \mid \boxed{\{x\}\ C\ \{\psi\}}\}}$
  - Where $(A \rightarrow B) \vdash (A < B)$



$\{x=3\} \bigsqcup \{x=4\}$

$= ?$

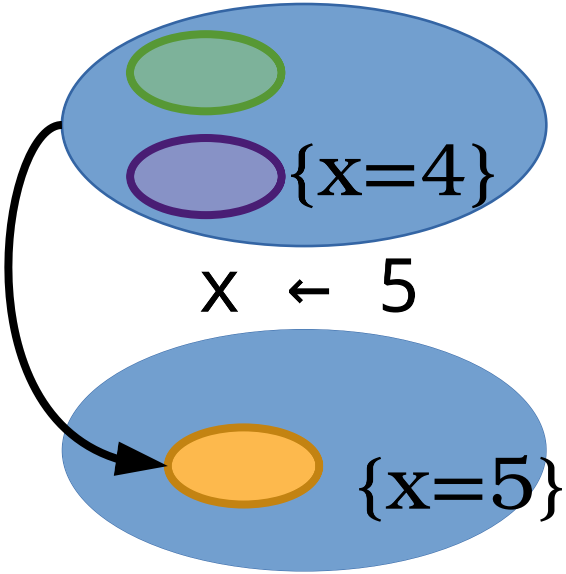
$\{x=3\} \rightarrow \{x=3 \lor x=4\}$

$\{x=4\} \rightarrow \{x=3 \lor x=4\}$

$\psi = \{x=5\}$

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- $wp(C,\psi) = \bigsqcup \{x \mid \{x\}\ C\ \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A < B)$



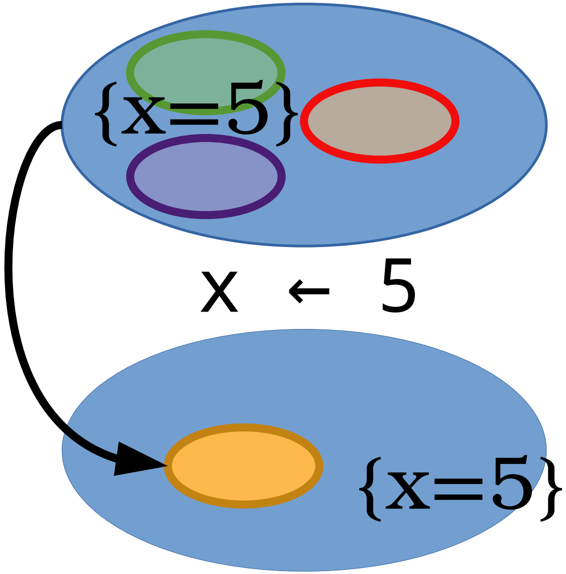
$$\{x=3\} \sqcup \{x=4\}$$
$$= \{x=3 \lor x=4\}$$

$$\{x=3\} \rightarrow \{x=3 \lor x=4\}$$
$$\{x=4\} \rightarrow \{x=3 \lor x=4\}$$

$$\psi = \{x=5\}$$

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- $\boxed{wp(C,\psi)} = \boxed{\bigsqcup \{x \mid \boxed{\{x\}\ C\ \{\psi\}}\}}$
  - Where $(A \rightarrow B) \vdash (A < B)$



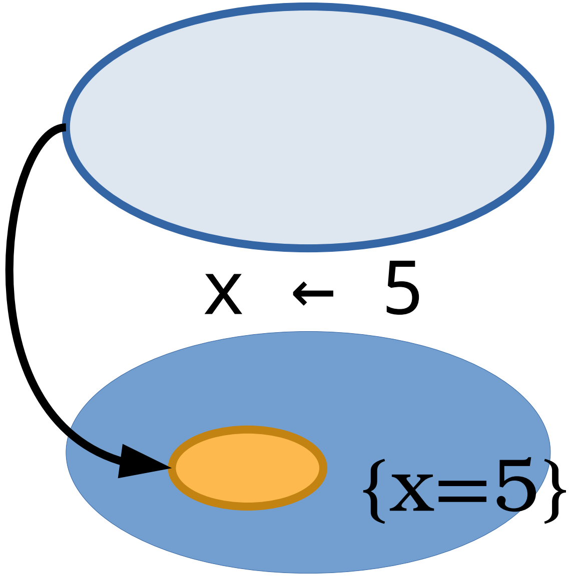
$$\{x=3\} \bigsqcup \{x=4\} \bigsqcup \{x=5\}$$
$$= \{3 \leq x \wedge x \leq 5\}$$

$$\{x=3\} \rightarrow \{3 \leq x \wedge x \leq 5\}$$
$$\{x=4\} \rightarrow \{3 \leq x \wedge x \leq 5\}$$
$$\{x=5\} \rightarrow \{3 \leq x \wedge x \leq 5\}$$

$\psi = \{x=5\}$

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- $\boxed{wp(C,\psi)} = \boxed{\bigsqcup \{x \mid \boxed{\{x\}\ C\ \{\psi\}}\}}$
  - Where $(A \rightarrow B) \vdash (A < B)$



$$\{x=3\} \bigsqcup \{x=4\} \bigsqcup \{x=5\} \bigsqcup \ldots$$
$$= \{\#T\}$$

$$\{x=3\} \rightarrow \{\#T\}$$
$$\{x=4\} \rightarrow \{\#T\}$$
$$\ldots$$

$$\psi = \{x=5\}$$

Hoare Logic – weakest preconditions

- What do we really mean by captures all states?
- $\boxed{wp(C,\psi)} = \boxed{\bigsqcup \{x \mid \boxed{\{x\}\ C\ \{\psi\}}\}}$
  - Where $(A \rightarrow B) \vdash (A < B)$

Intuitively, B is at least as general as A
(it holds in at least as many states)

Hoare Logic – weakest preconditions

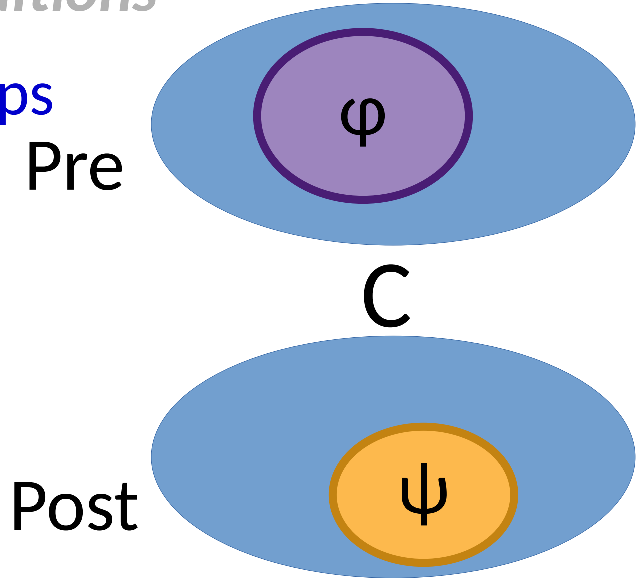
- What do we really mean by captures all states?
- $wp(C, \psi) = \bigsqcup \{x \mid \{x\}\ C\ \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A < B)$
- Technically, these are ***Weakest Sufficient Preconditions***

Hoare Logic

- What do we really mean by captures all states?
- $wp(C,\psi) = \bigsqcup \{x \mid \{x\}\ C\ \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A<B)$
- Technically, these are ***Weakest Sufficient Preconditions***
- We may also consider/compute other relationships

Hoare Logic

- What do we really mean by captures all states?
- $wp(C, \psi) = \bigsqcup \{x \mid \{x\}\ C\ \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A < B)$
- Technically, these are ***Weakest Sufficient Preconditions***
- We may also consider/compute other relationships

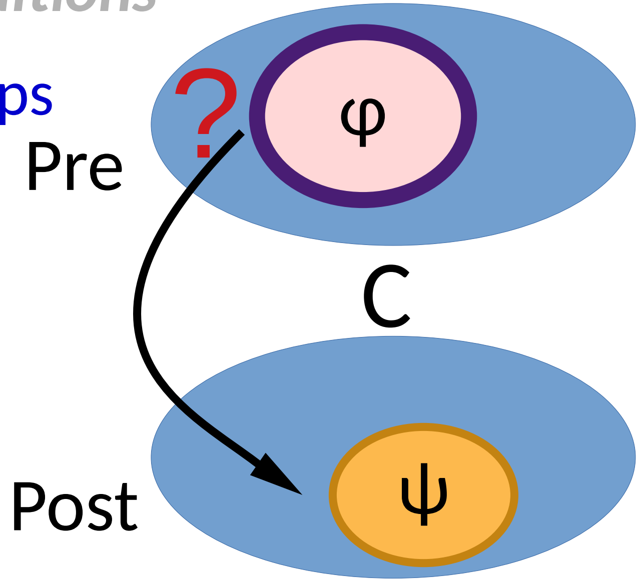


Hoare Logic

- What do we really mean by captures all states?
- $wp(C, \psi) = \bigsqcup \{x \mid \{x\}\ C\ \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A < B)$
- Technically, these are ***Weakest Sufficient Preconditions***
- We may also consider/compute other relationships
  - Weakest Sufficient Preconditions (wsp)

What states φ lead to ψ?

"Given ψ, what must be true for it to hold?"

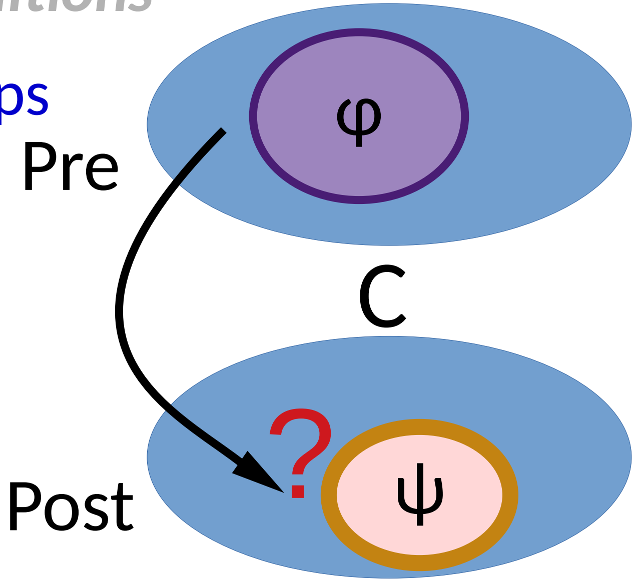


Hoare Logic

- What do we really mean by captures all states?
- $wp(C, \psi) = \bigsqcup \{x \mid \{x\}\ C\ \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A < B)$
- Technically, these are ***Weakest Sufficient Preconditions***
- We may also consider/compute other relationships
  - Weakest Sufficient Preconditions
  - Strongest Necessary Postconditions (snp)

What states ψ must φ lead to?

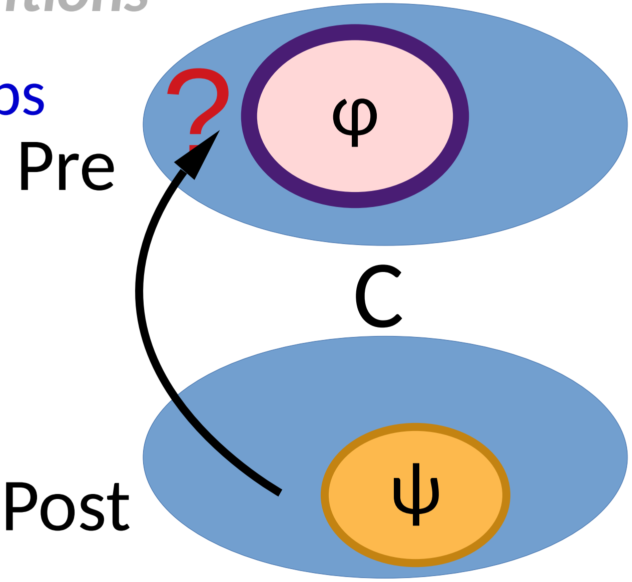
"Given φ, what is guaranteed when it holds?"



Hoare Logic

- What do we really mean by captures all states?
- $wp(C, \psi) = \bigsqcup \{x \mid \{x\}\ C\ \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A < B)$
- Technically, these are ***Weakest Sufficient Preconditions***
- We may also consider/compute other relationships
  - Weakest Sufficient Preconditions
  - Strongest Necessary Postconditions
  - Strongest Necessary Preconditions (snpre)

What states φ lead to ψ?



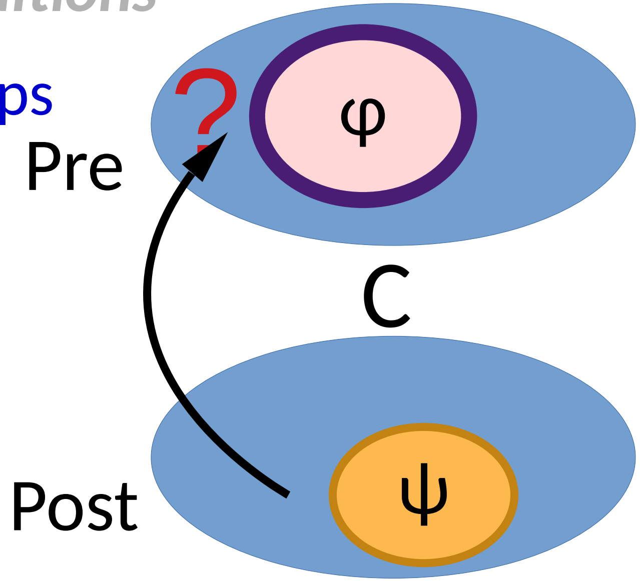
"Given ψ, what if false at φ would exclude it?"

Hoare Logic

- What do we really mean by captures all states?
- $wp(C, \psi) = \bigsqcup \{x \mid \{x\}\ C\ \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A < B)$
- Technically, these are ***Weakest Sufficient Preconditions***
- We may also consider/compute other relationships
  - Weakest Sufficient Preconditions
  - Strongest Necessary Postconditions
  - Strongest Necessary Preconditions (snpre)

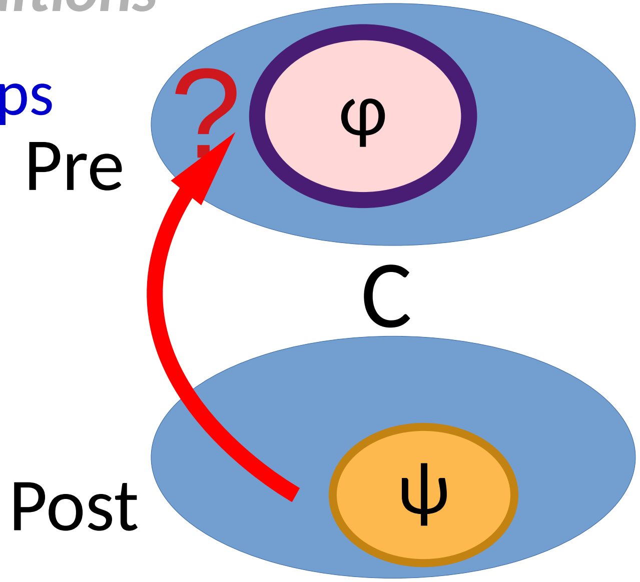
What states φ lead to ψ?

Then how does this differ from *wsp*?



Hoare Logic

- What do we really mean by captures all states?
- $wp(C, \psi) = \bigsqcup \{x \mid \{x\}\ C\ \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A < B)$
- Technically, these are ***Weakest Sufficient Preconditions***
- We may also consider/compute other relationships
  - Weakest Sufficient Preconditions
  - Strongest Necessary Postconditions
  - Strongest Necessary Preconditions



WSP

φ@pre $\rightarrow$ ψ@post

SNPre

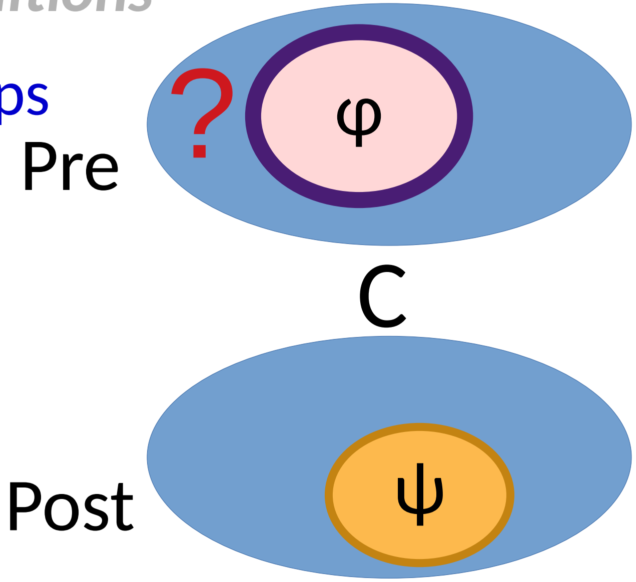
φ@pre $\leftarrow$ ψ@post

Hoare Logic

- What do we really mean by captures all states?
- $wp(C, \psi) = \bigsqcup \{x \mid \{x\}\ C\ \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A < B)$
- Technically, these are ***Weakest Sufficient Preconditions***
- We may also consider/compute other relationships
  - Weakest Sufficient Preconditions
  - Strongest Necessary Postconditions
  - Strongest Necessary Preconditions

WSP SNPre

Since solving them is technically impossible, these differ in practice!

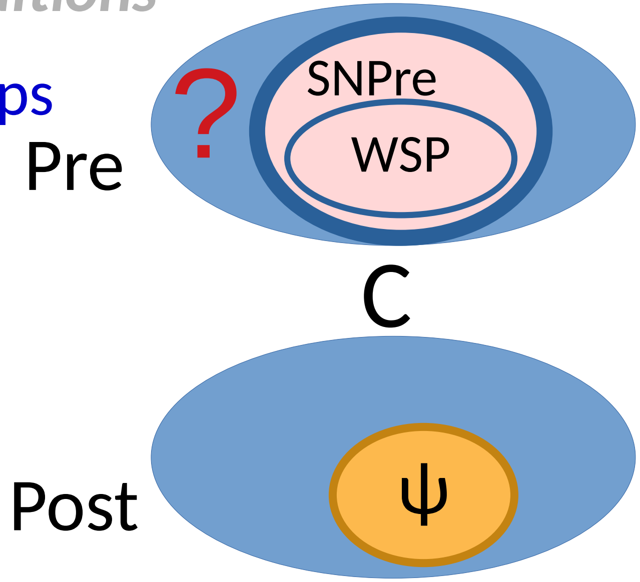


Hoare Logic

- What do we really mean by captures all states?
- $wp(C, \psi) = \bigsqcup \{x \mid \{x\}\ C\ \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A < B)$
- Technically, these are ***Weakest Sufficient Preconditions***
- We may also consider/compute other relationships
  - Weakest Sufficient Preconditions
  - Strongest Necessary Postconditions
  - Strongest Necessary Preconditions

WSP SNPre

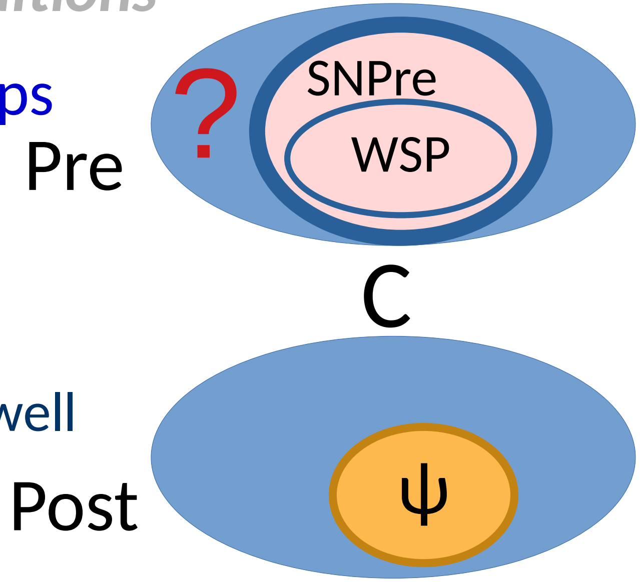
Since solving them is technically impossible,
these differ in practice!



Hoare Logic

- What do we really mean by captures all states?
- $wp(C, \psi) = \bigsqcup \{x \mid \{x\}\ C\ \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A < B)$
- Technically, these are ***Weakest Sufficient Preconditions***
- We may also consider/compute other relationships
  - Weakest Sufficient Preconditions
  - Strongest Necessary Postconditions
  - Strongest Necessary Preconditions

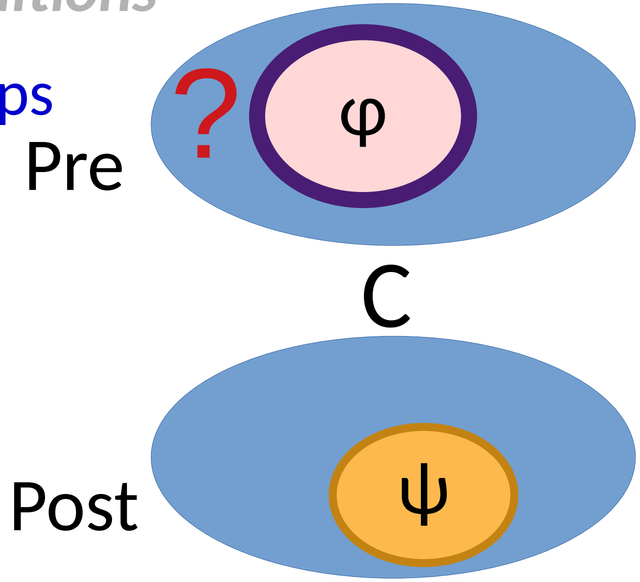
    In practice, SNPre captures *precondition assertions* well
    [Cousot 2013]



Hoare Logic

- What do we really mean by captures all states?
- $wp(C, \psi) = \bigsqcup \{x \mid \{x\}\ C\ \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A < B)$
- Technically, these are ***Weakest Sufficient Preconditions***
- We may also consider/compute other relationships
  - Weakest Sufficient Preconditions
  - Strongest Necessary Postconditions
  - Strongest Necessary Preconditions
  - Weakest *Liberal* Preconditions

What states φ lead to ψ
or *do not terminate*?



Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

$$\mathrm{wp}(x \leftarrow E, \psi) = [E/x]\psi$$

Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

$$\mathrm{wp}(x \leftarrow E, \psi) = [E/x]\psi$$

$$\{ \quad ??? \quad \}$$

$$x \leftarrow a + b$$

$$\{x<5\}$$

Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

$$wp(x \leftarrow E, \psi) = [E/x]\psi$$

$$\{a + b < 5\}$$

x ← a + b

$$\{x<5\}$$

Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

$$wp(x \leftarrow E, \psi) = [E/x]\psi$$

$$wp(S;\ T, \psi) = wp(S,\ wp(T, \psi))$$

Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

$$\mathrm{wp}(x \leftarrow E, \psi) = [E/x]\psi$$

$$\mathrm{wp}(S;\, T, \psi) = \mathrm{wp}(S,\, \mathrm{wp}(T, \psi))$$

$\{\quad ???\quad \}$

```
b ← 7;

x ← a + b
```

$\{x<5\}$

Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

$$wp(x \leftarrow E, \psi) = [E/x]\psi$$

$$wp(S;\, T, \psi) = wp(S,\, wp(T, \psi))$$

$\{ \quad ??? \quad \}$

```
b ← 7;
```

$\{a + b < 5\}$

```
x ← a + b
```

$\{x<5\}$

Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

$$wp(x \leftarrow E, \psi) = [E/x]\psi$$

$$wp(S;\, T, \psi) = wp(S,\, wp(T, \psi))$$

$\{a + 7 < 5\}$

```
b ← 7;
```

$\{a + b < 5\}$

```
x ← a + b
```

$\{x<5\}$

Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

$$wp(x \leftarrow E, \psi) = [E/x]\psi$$

$$wp(S;\, T, \psi) = wp(S,\, wp(T, \psi))$$

$$wp(\texttt{if}\ B\ \texttt{then}\ S\ \texttt{else}\ T, \psi)$$
$$= B \rightarrow wp(S, \psi)\ \wedge\ \neg B \rightarrow wp(T, \psi)$$

Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

$$\mathrm{wp}(x \leftarrow E, \psi) = [E/x]\psi$$

$$\mathrm{wp}(S;\, T, \psi) = \mathrm{wp}(S, \mathrm{wp}(T, \psi))$$

$$\mathrm{wp}(\texttt{if } B \texttt{ then } S \texttt{ else } T, \psi) = \boxed{B \to \mathrm{wp}(S, \psi)} \wedge \boxed{\neg B \to \mathrm{wp}(T, \psi)}$$

Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

$$wp(x \leftarrow E, \psi) = [E/x]\psi$$

$$wp(S;\ T, \psi) = wp(S,\ wp(T, \psi))$$

$$wp(\texttt{if}\ B\ \texttt{then}\ S\ \texttt{else}\ T, \psi)$$
$$= B \rightarrow wp(S, \psi)\ \wedge\ \neg B \rightarrow wp(T, \psi)$$

```
if c then
  d = y + 2
else
  d = y + 5
x/d
```

Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

$$\mathrm{wp}(x \leftarrow E, \psi) = [E/x]\psi$$

$$\mathrm{wp}(S;\, T, \psi) = \mathrm{wp}(S, \mathrm{wp}(T, \psi))$$

$$\mathrm{wp}(\texttt{if}\ B\ \texttt{then}\ S\ \texttt{else}\ T, \psi)$$
$$= B \rightarrow \mathrm{wp}(S, \psi) \land \neg B \rightarrow \mathrm{wp}(T, \psi)$$

{ ??? }

```
if c then
   d = y + 2
else
   d = y + 5
x/d
```

$\{d \neq 0\}$

Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

$$wp(x \leftarrow E, \psi) = [E/x]\psi$$

$$wp(S;\ T, \psi) = wp(S,\ wp(T, \psi))$$

$$wp(\texttt{if}\ B\ \texttt{then}\ S\ \texttt{else}\ T, \psi)$$
$$= B \rightarrow wp(S, \psi)\ \wedge\ \neg B \rightarrow wp(T, \psi)$$

	{ ??? }
`if c then`	
`  d = y + 2`	$\{y+2 \neq 0\}$
`else`	
`  d = y + 5`	$\{y+5 \neq 0\}$
`x/d`	$\{d \neq 0\}$

Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

$$\mathrm{wp}(x \leftarrow E, \psi) = [E/x]\psi$$

$$\mathrm{wp}(S;\ T, \psi) = \mathrm{wp}(S, \mathrm{wp}(T, \psi))$$

$$\mathrm{wp}(\texttt{if}\ B\ \texttt{then}\ S\ \texttt{else}\ T, \psi)$$
$$= B \rightarrow \mathrm{wp}(S, \psi)\ \land\ \neg B \rightarrow \mathrm{wp}(T, \psi)$$

```
if c then
  d = y + 2
else
  d = y + 5
x/d
```

$\{c \rightarrow y+2 \neq 0\ \land\ \neg c \rightarrow y+5 \neq 0\}$

$\{y+2 \neq 0\}$

$\{y+5 \neq 0\}$

$\{d \neq 0\}$

Hoare Logic

- Careful points
  - Redefinition of variables

Pre: {a < 5, c < 2}

```
b = a + 2
a = 3*c
```

Post: {??}

Hoare Logic

- Careful points
  - Redefinition of variables

Pre: {a < 5, c < 2}

```
b = a + 2
a = 3*c
```

Post: {??}

It can be necessary to rename variables that are redefined.

Hoare Logic

- Careful points
  - Redefinition of variables

Pre: {a < 5, c < 2}

```
b = a + 2
a = 3*c
```

Post: {??}

It can be necessary to rename variables that are redefined.

Hoare Logic

- Careful points
  - Redefinition of variables
  - Pointers

Pre: {??}

```
*a = *a + 5
```

Post: {*a + *b < 10}

Hoare Logic

- Careful points
  - Redefinition of variables
  - Pointers

Pre: {??}

```
*a = *a + 5
```

Post: {*a + *b < 10}

Efficiently modeling memory is challenging!
Newer logics target this directly.
(points-to analysis allows for ***weak*** and ***strong*** updates)

Hoare Logic

- Careful points
  - Redefinition of variables
  - Pointers
  - Loops

Hoare Logic

- Careful points
  - Redefinition of variables
  - Pointers
  - Loops

Loops run head first into undecidability!
They require deriving an *inductive invariant*.

Hoare Logic

- Careful points
  - Redefinition of variables
  - Pointers
  - Loops

$\{\varphi\}C\{\psi\}$

```
while B do S done
```

Hoare Logic

- Careful points
  - Redefinition of variables
  - Pointers
  - Loops

$\{\varphi\}C\{\psi\}$

```
while B do S done
```

$Inv \land \neg B \rightarrow \psi$ exit

Hoare Logic

- Careful points
  - Redefinition of variables
  - Pointers
  - Loops

$\{\varphi\}C\{\psi\}$

```
while B do S done
```

$\{Inv \land \neg B \rightarrow \psi\}$ exit

$\{Inv \land B\}\ S\ \{Inv\}$ continue

Hoare Logic

- Careful points
  - Redefinition of variables
  - Pointers
  - Loops

$\{\varphi\}C\{\psi\}$ `while B do S done`

$\{Inv \wedge \neg B \rightarrow \psi\}$ exit

$\{Inv \wedge B\}\ S\ \{Inv\}$ continue

$\{\varphi \rightarrow Inv\}$ enter

Hoare Logic

- Careful points
  - Redefinition of va
  - Pointers
  - Loops

Automatically inferring such invariants
is used for verifying safe:
avionics
machine learning
…

$\{\varphi\}C\{\psi\}$

```
while B do S done
```

$\{Inv \land \neg B \rightarrow \psi\}$ exit

$\{Inv \land B\}\ S\ \{Inv\}$ continue

$\{\varphi \rightarrow Inv\}$ enter

Separation Logic

- Linear logic allows facts to be used exactly once <> or arbitrarily many times [].

Separation Logic

- Linear logic allows facts to be used exactly once <>
  or arbitrarily many times [].
- *Separation logic* (informally) distinguishes separate facts (counting), allowing them to be used separately

Separation Logic

- Linear logic allows facts to be used exactly once <>
  or arbitrarily many times [].
- *Separation logic* (informally) distinguishes separate facts (counting), allowing them to be used separately
  - This helps to solve reasoning about pointers as we saw earlier

Separation Logic

- Linear logic allows facts to be used exactly once <>
  or arbitrarily many times [].
- *Separation logic* (informally) distinguishes separate facts (counting),
  allowing them to be used separately
- Hoare logic is extended with a separating conjunction *

Separation Logic

- Linear logic allows facts to be used exactly once <> or arbitrarily many times [].
- *Separation logic* (informally) distinguishes separate facts (counting), allowing them to be used separately
- Hoare logic is extended with a separating conjunction *

$$\{x \mapsto y * y \mapsto x\}\texttt{x = z}\{x \mapsto z * y \mapsto x\}$$

Facts separated by * do not “mix” (overlap)

Separation Logic

- Linear logic allows facts to be used exactly once <>
  or arbitrarily many times [].
- *Separation logic* (informally) distinguishes separate facts (counting), allowing them to be used separately
- Hoare logic is extended with a separating conjunction $*$

$$\{x \mapsto y * y \mapsto x\}\ x = z\ \{x \mapsto z * y \mapsto x\}$$

Suppose we used $\wedge$ instead, what problem exists?

Separation Logic

- Linear logic allows facts to be used exactly once <>
  or arbitrarily many times [].
- *Separation logic* (informally) distinguishes separate facts (counting), allowing them to be used separately
- Hoare logic is extended with a separating conjunction $*$

$$\{x \mapsto y * y \mapsto x\}\, x \;=\; z \{x \mapsto z * y \mapsto x\}$$

- Separation logic enables efficient compositional reasoning
  - It is the backbone of Facebook's Infer engine!
  - It combines Hoare logic with a substructural logic

Separation Logic

- The ***frame rule*** enables reasoning about the logical footprint of a command

$$\frac{\{\varphi\}C\{\psi\}}{\{\varphi * r\}C\{\psi * r\}}$$

Separation Logic

- The ***frame rule*** enables reasoning about the logical footprint of a command

$$\frac{\{\varphi\}C\{\psi\}}{\{\varphi * r\}C\{\psi * r\}}$$

- Part of the power is that frames can be inferred via ***bi-abduction***

Solving Problems Using Logic

Solving problems using logic

- We will look at a few ways logic can attack real problems

Solving problems using logic

- We will look at a few ways logic can attack real problems
- The exact techniques may have flaws,
  but how they attack problems with logic is interesting

Discovering & Disproving Bugs

```
foo(a,b,c) {
  if (a != null) {
    b = c;
    t = new...;
    c.f = t;
  }
  d = a;
  if (d != null) {
    b.f.g = 10;
  }
}
```

[Margoor & Komondoor, 2015]

Discovering & Disproving Bugs

```
foo(a,b,c) {
  if (a != null) {
    b = c;
    t = new...;
    c.f = t;
  }
  d = a;
  if (d != null) {
    b.f.g = 10;
  }
}
```

[Margoor & Komondoor, 2015]

Can accessing the field g cause a null pointer exception?

Discovering & Disproving Bugs

```
foo(a,b,c) {
  if (a != null) {
    b = c;
    t = new...;
    c.f = t;
  }
  d = a;
  if (d != null) {
    b.f.g = 10;
  }
}
```

{b.f=null}

[Margoor & Komondoor, 2015]

Discovering & Disproving Bugs

```
foo(a,b,c) {
  if (a != null) {
    b = c;
    t = new...;
    c.f = t;
  }
  d = a;
  if (d != null) {
    b.f.g = 10;
  }
}
```

$\{b.f = null \wedge d \neq null\}$

$\{b.f = null\}$

[Margoor & Komondoor, 2015]

Discovering & Disproving Bugs

```
foo(a,b,c) {
  if (a != null) {
    b = c;
    t = new...;
    c.f = t;
  }
  d = a;
  if (d != null) {
    b.f.g = 10;
  }
}
```

{b.f=null ∧ a≠null}

{b.f=null ∧ d≠null}

{b.f=null}

[Margoor & Komondoor, 2015]

Discovering & Disproving Bugs

```
foo(a,b,c) {
  if (a != null) {
    b = c;
    t = new...;
    c.f = t;
  }
  d = a;
  if (d != null) {
    b.f.g = 10;
  }
}
```

{$(\boxed{b \neq c} \wedge b.f = null \wedge a \neq null) \vee (\boxed{b = c} \wedge t = null \wedge a \neq null)$}

{$b.f = null \wedge a \neq null$}

{$b.f = null \wedge d \neq null$}

{$b.f = null$}

[Margoor & Komondoor, 2015]

Discovering & Disproving Bugs

```
foo(a,b,c) {
  if (a != null) {
    b = c;
    t = new...;
    c.f = t;
  }
  d = a;
  if (d != null) {
    b.f.g = 10;
  }
}
```

{b≠c ∧ b.f=null ∧ a≠null}

{(b≠c ∧ b.f=null ∧ a≠null) v (b=c∧t=null ∧ a≠null)}

{b.f=null ∧ a≠null}

{b.f=null ∧ d≠null}

{b.f=null}

[Margoor & Komondoor, 2015]

Discovering & Disproving Bugs

```
foo(a,b,c) {
  if (a != null) {
    b = c;
    t = new...;
    c.f = t;
  }
  d = a;
  if (d != null) {
    b.f.g = 10;
  }
}
```

{#f}

$\{b \neq c \land b.f = null \land a \neq null\}$

$\{(b \neq c \land b.f = null \land a \neq null) \lor (b = c \land t = null \land a \neq null)\}$

$\{b.f = null \land a \neq null\}$

$\{b.f = null \land d \neq null\}$

$\{b.f = null\}$

[Margoor & Komondoor, 2015]

Discovering & Disproving Bugs

```
foo(a,b,c) {
  if (a != null) {
    b = c;
    t = new...;
    c.f = t;
  }
  d = a;
  if (d != null) {
    b.f.g = 10;
  }
}
```

$\{(a \neq null \rightarrow \#f) \lor (a = null \rightarrow b.f = null \land a \neq null)\}$

$\{\#f\}$

$\{b \neq c \land b.f = null \land a \neq null\}$

$\{(b \neq c \land b.f = null \land a \neq null) \lor (b = c \land t = null \land a \neq null)\}$

$\{b.f = null \land a \neq null\}$

$\{b.f = null \land d \neq null\}$

$\{b.f = null\}$

[Margoor & Komondoor, 2015]

Discovering & Disproving Bugs

Safe!

```
foo(a,b,c) {
  if (a != null) {
    b = c;
    t = new...;
    c.f = t;
  }
  d = a;
  if (d != null) {
    b.f.g = 10;
  }
}
```

$\{(\cancel{a \neq null} \rightarrow \#f) \vee (a = null \rightarrow \cancel{b.f = null \wedge a \neq null})\}$ **= #f**

$\{\#f\}$

$\{b \neq c \wedge b.f = null \wedge a \neq null\}$

$\{(b \neq c \wedge b.f = null \wedge a \neq null) \vee (b = c \wedge t = null \wedge a \neq null)\}$

$\{b.f = null \wedge a \neq null\}$

$\{b.f = null \wedge d \neq null\}$

$\{b.f = null\}$

[Margoor & Komondoor, 2015]

Discovering & Disproving Bugs

Safe!

```
foo(a,b,c) {
  if (a != null) {
    b = c;
    t = new...;
    c.f = t;
  }
  d = a;
  if (d != null) {
    b.f.g = 10;
  }
}
```

[Margoor & Komondoor, 2015]

{(~~a≠null~~ → #f) v (~~a=null → b.f=null ∧~~ a≠null)} **= #f**

{#f}

{b≠c ∧ b.f=null ∧ a≠null}

{(b≠c ∧ b.f=null ∧ a≠null) v (b=c∧t=null ∧ a≠null)}

{b.f=null ∧ a≠null}

{b.f=null ∧ d≠null}

{b.f=null}

Note: this can be automated within a tool!

Localizing Bugs

[Jose & Majumdar, 2011]

```
int arr[3];
...
if (index != 1) {
  index = 2;
} else {
  index = index + 2;
}
i = index;
print(arr[i]);
```

Localizing Bugs

[Jose & Majumdar, 2011]

```
int arr[3];
...
if (index != 1) {
  index = 2;
} else {
  index = index + 2;
}
i = index;
print(arr[i]);
```

assert(0 ≤ `i` < 3) *should* hold

Localizing Bugs

[Jose & Majumdar, 2011]

```
int arr[3];
...
if (index != 1) {
  index = 2;
} else {
  index = index + 2;
}
i = index;
print(arr[i]);
```

assert(0 ≤ `i` < 3) *should* hold

When the starting `index` is 1, `i` is out of bounds

Localizing Bugs

[Jose & Majumdar, 2011]

```
int arr[3];
...
if (index != 1) {
  index = 2;
} else {
  index = index + 2;
}
i = index;
print(arr[i]);
```

assert($0 \leq$ `i` < 3) *should* hold

When the starting `index` is 1, `i` is out of bounds

We will generate constraints in the forward direction

Localizing Bugs

[Jose & Majumdar, 2011]

$\texttt{index}_1 = 1$

$\wedge\ (0 \leq \texttt{i} < 3)$

We will generate constraints in the forward direction

```
int arr[3];
...
if (index != 1) {
  index = 2;
} else {
  index = index + 2;
}
i = index;
print(arr[i]);
```

assert(0 ≤ `i` < 3) *should* hold

When the starting `index` is 1, `i` is out of bounds

Localizing Bugs

[Jose & Majumdar, 2011]

$\texttt{index}_1 = 1$
$\wedge\ \texttt{guard}_1 = (\texttt{index}_1 \neq 1)$

$\wedge\ (0 \leq \texttt{i} < 3)$

We will generate constraints in the forward direction

```
int arr[3];
...
if (index != 1) {
  index = 2;
} else {
  index = index + 2;
}
i = index;
print(arr[i]);
```

assert(0 ≤ `i` < 3) *should* hold

When the starting `index` is 1, `i` is out of bounds

Localizing Bugs

[Jose & Majumdar, 2011]

$\texttt{index}_1 = 1$
$\land\ \texttt{guard}_1 = (\texttt{index}_1 \neq 1)$
$\land\ \texttt{index}_2 = 2$

$\land\ (0 \leq \texttt{i} < 3)$

```
int arr[3];
...
if (index != 1) {
   index = 2;
} else {
   index = index + 2;
}
i = index;
print(arr[i]);
```

assert(0 ≤ `i` < 3) *should* hold

When the starting `index` is 1,
`i` is out of bounds

Localizing Bugs

[Jose & Majumdar, 2011]

$\texttt{index}_1 = 1$
$\wedge\ \texttt{guard}_1 = (\texttt{index}_1 \neq 1)$
$\wedge\ \texttt{index}_2 = 2$
$\wedge\ \texttt{index}_3 = (\texttt{index}_1 + 2)$

$\wedge\ (0 \leq \texttt{i} < 3)$

```
int arr[3];
...
if (index != 1) {
   index = 2;
} else {
   index = index + 2;
}
i = index;
print(arr[i]);
```

assert($0 \leq$ `i` < 3) *should* hold

When the starting `index` is 1,
`i` is out of bounds

Localizing Bugs

$$
\begin{aligned}
& index_1 = 1 \\
\wedge\ & guard_1 = (index_1 \neq 1) \\
\wedge\ & index_2 = 2 \\
\wedge\ & index_3 = (index_1 + 2) \\
\wedge\ & (guard_1 \rightarrow i = index_2) \\
\wedge\ & (\neg guard_1 \rightarrow i = index_3) \\
\wedge\ & (0 \leq i < 3)
\end{aligned}
$$

[Jose & Majumdar, 2011]

```
int arr[3];
...
if (index != 1) {
  index = 2;
} else {
  index = index + 2;
}
i = index;
print(arr[i]);
```

assert($0 \leq$ `i` < 3) *should* hold

When the starting `index` is 1, `i` is out of bounds

Localizing Bugs

[Jose & Majumdar, 2011]

$$
\begin{aligned}
&\texttt{index}_1 = 1 \\
&\wedge\ \texttt{guard}_1 = (\texttt{index}_1 \neq 1) \\
&\wedge\ \texttt{index}_2 = 2 \\
&\wedge\ \texttt{index}_3 = (\texttt{index}_1 + 2) \\
&\wedge\ (\texttt{guard}_1 \rightarrow \texttt{i=index}_2) \\
&\wedge\ (\neg\texttt{guard}_1 \rightarrow \texttt{i=index}_3) \\
&\wedge\ (0 \leq \texttt{i} < 3)
\end{aligned}
$$

P

```
int arr[3];
...
if (index != 1) {
  index = 2;
} else {
  index = index + 2;
}
i = index;
print(arr[i]);
```

assert($0 \leq$ i < 3) *should* hold

When the starting index is 1,
i is out of bounds

Localizing Bugs

[Jose & Majumdar, 2011]

$$
\left.\begin{array}{l}
\texttt{index}_1 = 1 \\
\wedge\ \texttt{guard}_1 = (\texttt{index}_1 \neq 1) \\
\wedge\ \texttt{index}_2 = 2 \\
\wedge\ \texttt{index}_3 = (\texttt{index}_1 + 2) \\
\wedge\ (\texttt{guard}_1 \rightarrow \texttt{i=index}_2) \\
\wedge\ (\neg\texttt{guard}_1 \rightarrow \texttt{i=index}_3)
\end{array}\right\} \mathrm{snp}(P,\#t)
$$

$\wedge\ (0 \leq \texttt{i} < 3)$

P

```
int arr[3];
...
if (index != 1) {
  index = 2;
} else {
  index = index + 2;
}
i = index;
print(arr[i]);
```

assert(0 ≤ `i` < 3) *should* hold

When the starting `index` is 1, `i` is out of bounds

Localizing Bugs

[Jose & Majumdar, 2011]

$index_1 = 1$
$\wedge\ guard_1 = (index_1 \neq 1)$
$\wedge\ index_2 = 2$
$\wedge\ index_3 = (index_1 + 2)$
$\wedge\ (guard_1 \rightarrow i = index_2)$
$\wedge\ (\neg guard_1 \rightarrow i = index_3)$
$\wedge\ (0 \leq i < 3)$

This is *always false*,
but we can *use* that!

```
int arr[3];
...
if (index != 1) {
  index = 2;
} else {
  index = index + 2;
}
i = index;
print(arr[i]);
```

assert($0 \leq i < 3$) *should* hold

When the starting index is 1,
i is out of bounds

Localizing Bugs

[Jose & Majumdar, 2011]

$\boxed{index_1 = 1}$
$\wedge\ guard_1 = (index_1 \neq 1)$
$\wedge\ index_2 = 2$
$\wedge\ index_3 = (index_1 + 2)$
$\wedge\ (guard_1 \rightarrow i = index_2)$
$\wedge\ (\neg guard_1 \rightarrow i = index_3)$
$\boxed{\wedge\ (0 \leq i < 3)}$

These constraints define our goal, so they are essential

```
if (index != 1) {
  index = 2;
} else {
  index = index + 2;
}
i = index;
print(arr[i]);
```

assert($0 \leq i < 3$) *should* hold

When the starting index is 1, i is out of bounds

Localizing Bugs

[Jose & Majumdar, 2011]

$index_1 = 1$
$\wedge\ guard_1 = (index_1 \neq 1)$
$\wedge\ index_2 = 2$
$\wedge\ index_3 = (index_1 + 2)$
$\wedge\ (guard_1 \rightarrow i = index_2)$
$\wedge\ (\neg guard_1 \rightarrow i = index_3)$
$\wedge\ (0 \leq i < 3)$

These constraints define our goal, so they are essential

Some of these constraints *conflict* with our goal

```
if (index != 1) {
  index = 2;
[illegible]
  index = index + 2;
}
i = index;
print(arr[i]);
```

assert(0 ≤ i < 3) *should* hold

When the starting index is 1,
i is out of bounds

Localizing Bugs

[Jose & Majumdar, 2011]

$$
\begin{aligned}
&\boxed{\texttt{index}_1 = 1} \\
&\land\ \texttt{guard}_1 = (\texttt{index}_1 \neq 1) \\
&\land\ \texttt{index}_2 = 2 \\
&\land\ \texttt{index}_3 = (\texttt{index}_1 + 2) \\
&\land\ (\texttt{guard}_1 \rightarrow \texttt{i}=\texttt{index}_2) \\
&\land\ (\neg\texttt{guard}_1 \rightarrow \texttt{i}=\texttt{index}_3) \\
&\boxed{\land\ (0 \leq \texttt{i} < 3)}
\end{aligned}
$$

These constraints define our goal, so they are essential

Some of these constraints ***conflict*** with our goal

```
if (index != 1) {
  index = 2;
  index = index + 2;
print(arr[i]);
```

Minimum unsat cores & ***partial MAX-SAT*** can discover the conflicts

assert(0 ≤ i < 3) *should* hold

When the starting index is 1, i is out of bounds

Localizing Bugs

[Jose & Majumdar, 2011]

$$
\begin{aligned}
& \boxed{index_1 = 1} \\
& \land\ guard_1 = (index_1 \neq 1) \\
& \land\ index_2 = 2 \\
& \land\ index_3 = (index_1 + 2) \\
& \land\ (guard_1 \rightarrow i{=}index_2) \\
& \land\ (\neg guard_1 \rightarrow i{=}index_3) \\
& \boxed{\land\ (0 \leq i < 3)}
\end{aligned}
$$

These constraints define our goal, so they are essential

Some of these constraints ***conflict*** with our goal

Minimum unsat cores & ***partial MAX-SAT*** can discover the conflicts

```
if (index != 1) {
  index = 2;
  index = index + 2;
print(arr[i]);
```

assert(0 ≤ i < 3) *should* hold

When the starting index is 1, i is out of bounds

Localizing Bugs

[Jose & Majumdar, 2011]

$\text{index}_1 = 1$
$\land\ \text{guard}_1 = (\text{index}_1 \neq 1)$
$\land\ \text{index}_2 = 2$
$\land\ \text{index}_3 = (\text{index}_1 + 2)$
$\land\ (\text{guard}_1 \rightarrow i = \text{index}_2)$
$\land\ (\neg\text{guard}_1 \rightarrow i = \text{index}_3)$
$\land\ (0 \leq i < 3)$

These constraints define our goal, so they are essential

Some of these constraints ***conflict*** with our goal

```
if (index != 1) {
    index = 2;
    index = index + 2;
print(arr[i]);
```

Minimum unsat cores & ***partial MAX-SAT*** can discover the conflicts

Must SAT

assert(0 ≤ i < 3) *should* hold

When the starting index is 1, i is out of bounds

Localizing Bugs

[Jose & Majumdar, 2011]

$$
\begin{aligned}
&\boxed{index_1 = 1}\\
&\wedge\ guard_1 = (index_1 \neq 1)\\
&\wedge\ index_2 = 2\\
&\wedge\ index_3 = (index_1 + 2)\\
&\wedge\ (guard_1 \rightarrow i = index_2)\\
&\wedge\ (\neg guard_1 \rightarrow i = index_3)\\
&\boxed{\wedge\ (0 \leq i < 3)}
\end{aligned}
$$

These constraints define our goal, so they are essential

Some of these constraints ***conflict*** with our goal

```
if (index != 1) {
    index = 2;
    index = index + 2;
print(arr[i]);
```

Minimum unsat cores & ***partial MAX-SAT*** can discover the conflicts

assert(0 ≤ i < 3) *should* hold

When the starting index is 1, i is out of bounds

Localizing Bugs

[Jose & Majumdar, 2011]

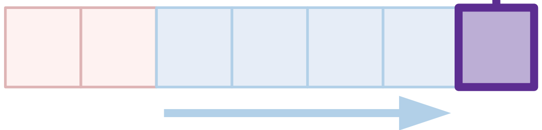
$index_1 = 1$
$\wedge\ guard_1 = (index_1 \neq 1)$
$\wedge\ index_2 = 2$
$\wedge\ index_3 = (index_1 + 2)$
$\wedge\ (guard_1 \rightarrow i=index_2)$
$\wedge\ (\neg guard_1 \rightarrow i=index_3)$
$\wedge\ (0 \leq i < 3)$

These constraints define our goal, so they are essential

```
if (index != 1) {
  index = 2;
  index = index + 2;
print(arr[i]);
```

Some of these constraints *conflict* with our goal

Minimum unsat cores & ***partial MAX-SAT*** can discover the conflicts



Could not SAT;
Blame for inconsistency

assert(0 ≤ i < 3) *should* hold

Must SAT   Max # satisfiable

When the starting index is 1, i is out of bounds

Further notes

- We will explore this further within Symbolic Execution

Further notes

- We will explore this further within Symbolic Execution
- Recognizing invariants & likely invariants can tackle many problems

Further notes

- We will explore this further within Symbolic Execution
- Recognizing invariants & likely invariants can tackle many problems
- Interpolants can help synthesize information as if “out of thin air”

Recap

- Formalism is a tool that can simplify reasoning about tasks

Recap

- Formalism is a tool that can simplify reasoning about tasks
- Many solutions involve a careful combination of
  - order theory (for comparison)
  - formal grammars (for structure)
  - formal logic (for inference)