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 be 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 be 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:

```
-4  -3  -2  -1  0  1  2  3  4
Set: ℤ  Relation: ≤
```
Order Theory

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

```
-4 -3 -2 -1 0 1 2 3 4
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:

\[
\begin{array}{cccccccc}
-4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 4 \\
\end{array}
\]

Set: \( \mathbb{Z} \) \quad 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, ...
We often want to compare complex data
  - Ordinal, multidimensional, ...
Order Theory

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

![Diagram with points (1,1), (2,2), and axes labeled 0 to 4 on the x-axis and 0 to 2 on the y-axis]
Order Theory

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

What is the result of 
\((1,1) \leq (2,2)\)?
We often want to compare complex data

- Ordinal, multidimensional, ...

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

We can take \(\leq\) to be componentwise comparison.
Order Theory

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

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

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

Which of these 4 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: $a \leq b \& b \leq c \Rightarrow a \leq c$
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 \( \leq \) is a **partial order** on a set \( S \) if \( \forall \ a, b, c \in S \)
  - Reflexive: \( a \leq a \)
  - Antisymmetric: \( a \leq b \land b \leq a \Rightarrow a = b \)
  - Transitive: \( a \leq b \land b \leq c \Rightarrow a \leq c \)

- When reasoning about partial orders, we prefer \( \sqsubseteq \)
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 $\leq$ is a *partial order* on a set $S$ if $\forall a,b,c \in S$

  $ab \leq_{\text{str}} xabyz$

  $ab \leq_{\text{seq}} xaybz$

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

• Common partial orders include
  – substring, subsequence, subset relationships
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
  - componentwise orderings
Partial Orders

- A relation \( \leq \) is a \textit{partial order} on a set \( S \) if \( \forall \ a, b, c \in S \)

\[
\begin{align*}
(1,1) & \sqsubseteq (1,2) \\
(1,1) & \sqsubseteq (2,2)
\end{align*}
\]

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

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

\[ f(x) = x + 1 \subseteq g(x) = x + 2 \]
A relation $\leq$ is a **partial order** on a set $S$ if $\forall \ a, b, c \in S$

- Reflexive: $a \leq a$
- Antisymmetric: $a \leq b \land b \leq a \Rightarrow a = b$
- Transitive: $a \leq b \land b \leq c \Rightarrow a \leq c$

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 $\leq$ 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
  - componentwise orderings
  - functions (considering all input/output mappings)

\[
\begin{align*}
  f(x) &= x + 1 \\ 
  g(x) &= x + 2 \\ 
  h(x) &= x \\ 
  i(x) &= -x
\end{align*}
\]
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
- 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 \( \bot \) (bottom)/\( \top \) (top).
Partial Orders

- We are often interested in upper and lower bounds.
We are often interested in upper and lower bounds.

- A join \( a \sqcup b \) is the least upper bound of \( a \) and \( b \)

What is \((0,1) \sqcup (1,0)\)?
Partial Orders

- We are often interested in upper and lower bounds.
  - A \textit{join} $a \sqcup b$ is the least upper bound of $a$ and $b$

What is $(0,1) \sqcup (1,0)$?
Partial Orders

- We are often interested in upper and lower bounds.
  - A join $a \sqcup b$ is the least upper bound of $a$ and $b$
    $a \sqsubseteq (a \sqcup b) \land b \sqsubseteq (a \sqcup b) \land (a \sqsubseteq c \land b \sqsubseteq c \rightarrow (a \sqcup b) \sqsubseteq c)$

What is $(0,1) \sqcup (1,0)$?
Partial Orders

- We are often interested in upper and lower bounds.
  - A join $a \sqcup b$ is the least upper bound of $a$ and $b$
  - A meet $a \sqcap b$ is the greatest lower bound of $a$ and $b$

What is $(0,1) \sqcap (1,0)$?
We are often interested in upper and lower bounds.

- A **join** $a \sqcup b$ is the least upper bound of $a$ and $b$
- A **meet** $a \sqcap b$ is the greatest lower bound of $a$ and $b$

What is $(0,1) \sqcap (1,0)$?
Partial Orders

- We are often interested in upper and lower bounds.
  - A \textit{join} $a \sqcup b$ is the least upper bound of $a$ and $b$
  - A \textit{meet} $a \sqcap b$ is the greatest lower bound of $a$ and $b$
  - Bounds must be \textit{unique} and may not exist.

\begin{tikzpicture}[scale=0.8]
  \node (0,0) at (0,0) {$(0,0)$};
  \node (0,1) at (0,1) {$(0,1)$};
  \node (0,2) at (0,2) {$(0,2)$};
  \node (1,0) at (1,0) {$(1,0)$};
  \node (1,1) at (1,1) {$(1,1)$};
  \node (1,2) at (1,2) {$(1,2)$};
  \node (2,0) at (2,0) {$(2,0)$};
  \node (2,1) at (2,1) {$(2,1)$};
  \node (2,2) at (2,2) {$(2,2)$};

  \draw (0,0) -- (0,1) -- (1,1) -- (2,1);
  \draw (0,0) -- (0,2) -- (1,2) -- (2,2);
  \draw (0,1) -- (1,1);
  \draw (0,2) -- (1,2);
\end{tikzpicture}
Partial Orders

- We are often interested in upper and lower bounds.
  - A join $a \sqcup b$ is the least upper bound of $a$ and $b$
  - A meet $a \sqcap b$ is the greatest lower bound of $a$ and $b$
  - Bounds must be *unique* and may not exist.
Partial Orders

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

What is $A \sqcup B$?
Partial Orders

- We are often interested in upper and lower bounds.
  - A \textit{join} \(a \sqcup b\) is the least upper bound of \(a\) and \(b\)
  - A \textit{meet} \(a \sqcap b\) is the greatest lower bound of \(a\) and \(b\)
  - Bounds must be \textit{unique} and may not exist.

What is \(A \sqcup B\)?
What is \(B \sqcup C\)?
We are often interested in upper and lower bounds.

- A join $a \sqcup b$ is the least upper bound of $a$ and $b$
- A meet $a \sqcap b$ is the greatest lower bound of $a$ and $b$
- Bounds must be unique and may not exist.
Partial Orders

- We are often interested in upper and lower bounds.
  - A **join** $a \sqcup b$ is the least upper bound of $a$ and $b$
  - A **meet** $a \sqcap b$ is the greatest lower bound of $a$ and $b$
  - Bounds must be unique and may not exist.
  - $\forall S' \subseteq S,$
We are often interested in upper and lower bounds.
- A join \( a \sqcup b \) is the least upper bound of \( a \) and \( b \)
- A meet \( a \sqcap b \) is the greatest lower bound of \( a \) and \( b \)
- Bounds must be unique and may not exist.
- \( \forall S' \subseteq S, \exists \sqcup S' \land \sqcap S' \Rightarrow \text{lattice} \)
We are often interested in upper and lower bounds.
- A **join** \( a \sqcup b \) is the least upper bound of \( a \) and \( b \)
- A **meet** \( a \sqcap b \) is the greatest lower bound of \( a \) and \( b \)
- Bounds must be unique and may not exist.
- \( \forall S \subseteq S', \exists \sqcup S' \) & \( \sqcap S' \Rightarrow \) lattice, \( \exists \sqcup S' \) or \( \exists \sqcap S' \Rightarrow \) semilattice
We are often interested in upper and lower bounds.

- A **join** $a \sqcup b$ is the least upper bound of $a$ and $b$
- A **meet** $a \sqcap b$ is the greatest lower bound of $a$ and $b$
- Bounds must be unique and may not exist.
- $\forall S' \subseteq S, \exists \sqcup S' \land \exists \sqcap S' \Rightarrow$ lattice, $\exists \sqcup S'$ or $\exists \sqcap S' \Rightarrow$ 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 
    \((X, \sqsubseteq_X), (Y, \sqsubseteq_Y), f: X \rightarrow Y\)
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**
    $(X, \subseteq_X), (Y, \subseteq_Y), f: X \rightarrow Y$
    $x_1 \subseteq_X x_2 \rightarrow f(x_1) \subseteq_Y f(x_2)$ (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

- 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

- 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

- 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

- 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!

Combining a set with $\top$ & $\bot$ like this yields a *flat lattice*. 
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! 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

```python
if x > 0
    y = 2
print(y)
    y = 3
    print(y)
```
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?

```
if x > 0
    y = 2
    y = 3
print(y)
```
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)

```
if x > 0
y = 2
y = 3
print(y)
```
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
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*
A regular language can be expressed via a regular expression.
A regular language can be expressed via a regular expression
A regular language can be expressed via a regular expression

\[ \text{regex} \rightarrow \text{symbol} \]
\[ \quad | (\text{`regex`}) \]
\[ \quad | \text{regex}^{*} \]
\[ \quad | \text{regex `|` regex} \]
\[ \quad | \text{regex regex} \]

E.g. \( a(bc \mid cd)^*e \) defines \( L \) containing \( abccdbce \)
A regular language can be expressed via a regular expression.

Finite automata can be used to recognize or generate elements of a regular language.
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.

Example:

\[ a(bc | cd)^*e \] 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.

\[ a(bc | cd)^*e \] 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

\[ a(bc \mid cd)^*e \] 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.

\[ a(bc \mid cd)^*e \] 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.

\[ a(bc \mid cd)^*e \] 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

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

  \[
  \text{Start} = A \\
  A \rightarrow cBd \\
  B \rightarrow eBf \\
  \mid g
  \]

  This requires some kind of memory.
Context Free Grammars & Pushdown Automata

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

\[
\begin{align*}
\text{Start} &= A \\
A &\rightarrow cBd \\
B &\rightarrow eBf \\
\mid &\rightarrow g \\
\end{align*}
\]

\[ce^n gf^n d\]
Context Free Grammars & Pushdown Automata

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

Start = A
A → cBd
B → eBf
| g

\[ce^n g f^n d\]
Context Free Grammars & Pushdown Automata

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

```
Start = A
A → cBd
B → eBf | g
```

-c$^e$e$^f$g$^n$d
Context Free Grammars & Pushdown Automata

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

Start = A

A \rightarrow cBd

B \rightarrow eBf

\mid g

\ce^n \cdot \gf^n \cdot d

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

\[ce^n gf^n d\]
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*)

\[
\begin{align*}
S & \rightarrow xAy \mid zB \\
A & \rightarrow aA \mid t \\
B & \rightarrow bB \mid u
\end{align*}
\]
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*)

\[
\begin{align*}
S & \rightarrow xAy \mid zB \\
A & \rightarrow aA \mid t \\
B & \rightarrow bB \mid u
\end{align*}
\]
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)

\[
\begin{align*}
S & \rightarrow xAy \mid zB \\
A & \rightarrow aA \mid t \\
B & \rightarrow bB \mid u
\end{align*}
\]
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*)

- \[
  S \rightarrow xAy \mid zB \\
  A \rightarrow aA \mid t \\
  B \rightarrow bB \mid u \\
  xaaty
  \]
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

![Diagram of a pushdown automaton with rules]

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

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

<table>
<thead>
<tr>
<th>A</th>
<th>xA</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>a</td>
</tr>
</tbody>
</table>

S

xaatya
Context FreeGrammars & 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)

\[
\begin{align*}
S & \rightarrow xAy \mid zB \\
A & \rightarrow aA \mid t \\
B & \rightarrow bB \mid u
\end{align*}
\]

\[
S \quad A \\
xA \\
aA \\
xaaty
\]
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)

\[
\begin{align*}
S & \rightarrow xAy \mid zB \\
A & \rightarrow aA \mid t \\
B & \rightarrow bB \mid u
\end{align*}
\]
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
```

---

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>xA</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>aA</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>aA</td>
<td></td>
</tr>
</tbody>
</table>

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**)

\[
\begin{align*}
S & \rightarrow xAy \mid zB \\
A & \rightarrow aA \mid t \\
B & \rightarrow bB \mid u
\end{align*}
\]

Diagram:

- **S** transitions to **xAy** or **zB**
- **A** transitions to **aA** or **t**
- **B** transitions to **bB** or **u**

Example path: **xaaty**

Table:

<table>
<thead>
<tr>
<th>S</th>
<th>xA</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>aA</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>
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*).

\[
\begin{align*}
S & \rightarrow xAy \mid zB \\
A & \rightarrow aA \mid t \\
B & \rightarrow bB \mid u
\end{align*}
\]

\[\text{x a a t y} \]

\[\text{xaaty}\]
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)

\[
S \rightarrow xAy \mid zB \\
A \rightarrow aA \mid t \\
B \rightarrow bB \mid u
\]
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*).

The behavior shown in the diagram is 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

```java
if (e) {
    ...
}
```
Context Free Grammars & Pushdown Automata

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

Automated Repair

```java
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

```java
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

```java
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

```java
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
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
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), $\lor$ (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
  - ¬ (Not), ∧ (And), ∨ (or)
  - → (Implies), ↔ (Iff)

\[ x \land \neg y \land z \]
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. father(x), Elephant(y)
Classical Logic

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

- Sentences can be true or false
Classical Logic

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

- Sentences can be true or false

\[ \forall x (\text{Elephant}(x) \rightarrow \text{Grey}(x)) \]
Classical Logic

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

- Sentences can be true or false

\[
\forall x (\text{Elephant}(x) \rightarrow \text{Grey}(x)) \\
\forall x (\text{Elephant}(x) \rightarrow \text{Elephant}(\text{father}(x)))
\]
Classical Logic

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

- Sentences can be true or false

- An interpretation $I$ of the world along with the rules of logic
determine truth via judgment ($I \vdash$)
Classical Logic

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

- Sentences can be true or false

- An interpretation $I$ of the world along with the rules of logic determine truth via judgment ($\vdash$)

$$I \vdash x \text{ and } I \vdash y \iff I \vdash x \land y$$
Classical Logic

- **Satisfiability**
  - A sentence $s$ is satisfiable $\iff \exists I (I \vdash s)$
Classical Logic

- **Satisfiability**
  - A sentence $s$ is satisfiable $\iff \exists I \ (I \vdash s)$

- **Validity**
  - A sentence $s$ is valid $\iff \forall I \ (I \vdash s)$
Classical Logic

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

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

- We will see later how these can be used for a wide variety of tasks
Classical Logic

- **Satisfiability**
  - A sentence $s$ is satisfiable $\iff \exists I (I \vdash s)$

- **Validity**
  - A sentence $s$ is valid $\iff \forall I (I \vdash s)$

- 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

\[
\Gamma \vdash x \quad \Delta \vdash y \\
\hline
\Gamma, \Delta \vdash x \land y
\]
Inference using classical logic

- Rules express how some judgments enable others

\[ \Gamma \vdash x \quad \Delta \vdash y \]

\[ \Gamma, \Delta \vdash x \land y \]
Inference using classical logic

- Rules express how some judgments enable others

\[
\begin{align*}
\Gamma & \vdash x \\
\Delta & \vdash y \\
\hline
\Gamma, \Delta & \vdash x \land y
\end{align*}
\]
Inference using classical logic

- Rules express how some judgments enable others
  \[ \Gamma \vdash x \quad \Delta \vdash y \]
  \[ \Gamma, \Delta \vdash x \land y \]
  - Proofs can be written by stacking rules
Inference using classical logic

- Rules express how some judgments enable others
  \[
  \Gamma \vdash x \quad \Delta \vdash y
  \]
  \[
  \therefore \Gamma, \Delta \vdash x \land y
  \]
- Proofs can be written by stacking rules

Intuitionistic & Constructive Logic

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

- Suppose a compiler cannot prove variable \( x \) is an \textit{int}. Is it reasonable for the compile to assume \( x \) is a \textit{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 compiler 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 compiler 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

\[ \vdash p \lor \neg p \]

Law of excluded middle
Intuitionistic & Constructive Logic

- Classic logic includes several rules including:

\[ \Gamma \vdash p \lor \neg p \]

\[ \Gamma \vdash \neg \neg p \]

\[ \Gamma \vdash p \]

Double negation elimination
Intuitionistic & Constructive Logic

- Classic logic includes several rules including

\[
\Gamma \vdash p \lor \neg p \quad \Gamma \vdash \neg \neg p
\]

- *Intuitionistic logic* excludes these to require direct evidence
Intuitionistic & Constructive Logic

- Classic logic includes several rules including:
  \[ \Gamma \vdash p \lor \neg p \quad \text{and} \quad \Gamma \vdash \neg \neg p \quad \text{and} \quad \Gamma \vdash p \]

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

- Note, this is commonly used in *type systems*.
sellsBurritos(store) \[\vdash\] buyBurrito(me,store)
sellsBurritos(store) ⊢ buyBurrito(me, store) 
has10Dollars(me) ∧ buyBurrito(me, store)
Linear & Substructural Logic

\[ \text{sellsBurritos}(\text{store}) \quad \text{has10Dollars}(\text{me}) \quad \vdash \quad \text{buyBurrito}(\text{me},\text{store}) \quad \land \quad \text{buyBurrito}(\text{me},\text{store}) \quad \land \quad \text{buyBurrito}(\text{me},\text{store}) \]
\[
\text{sellsBurritos}(\text{store}) \quad \text{has10Dollars}(\text{me}) \quad \vdash \quad \text{buy} \text{Burrito}(\text{me,store}) \quad \land \quad \text{buy} \text{Burrito}(\text{me,store}) \\
\quad \land \quad \text{buy} \text{Burrito}(\text{me,store}) \quad \land \quad \text{buy} \text{Burrito}(\text{me,store})
\]
Linear & Substructural Logic

\[
\text{sellsBurritos}(\text{store}) \vdash \text{buyBurrito}(\text{me}, \text{store}) \\
\wedge \text{buyBurrito}(\text{me}, \text{store}) \\
\wedge \text{buyBurrito}(\text{me}, \text{store}) \\
\wedge \text{buyBurrito}(\text{me}, \text{store})
\]

Classical & intuitionistic logic have trouble expressing consumable facts
Linear logic denotes separates facts into two kinds
- [Intuitionistic] as before
- <Linear> cannot be used with contraction or weakening
Linear & Substructural Logic

sellsBurritos(store) ⊸ buyBurrito(me,store)

has10Dollars(me)

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

\[
\Gamma, A, A, \Delta \vdash p \\
\Gamma, \Delta \vdash p \\
\Gamma, A, \Delta \vdash p \\
\Gamma, A, \Delta \vdash p
\]
Linear & Substructural Logic

sellsBurritos(store) \vdash buyBurrito(me, store)

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

\[
\begin{align*}
\Gamma, \Delta \vdash p & \quad \Gamma, \Delta \vdash p \\
\Gamma, A, \Delta \vdash p & \quad \Gamma, A, \Delta \vdash p
\end{align*}
\]
Linear & Substructural Logic

sellsBurritos(store) \quad \text{has10Dollars(me)} \quad \vdash \quad \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.

\[
\begin{align*}
\Gamma, A, A, \Delta & \vdash p \\
\Gamma, A, \Delta & \vdash p \\
\Gamma, \Delta & \vdash p \\
\Gamma, A, \Delta & \vdash p
\end{align*}
\]

**Idea:** Some facts (resources) require careful accounting.
Linear & Substructural Logic

sellsBurritos(store) → has10Dollars(me) ⊢ 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 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

sellsBurritos(store)  has10Dollars(me)  ⊢  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!

```rust
struct Thing(u32);
let a = Thing(5);
let b = a;
let c = a;
```
Linear & Substructural Logic

sellsBurritos(store) →→ 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!

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

⊢ a:Thing
Linear & Substructural Logic

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

- This forms the backbone of \textit{ownership types} in languages like Rust!

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

\[ \vdash \text{a:Thing} \quad \vdash \text{b:Thing} \]
Linear & Substructural Logic

sellsBurritos(store) has10Dollars(me) ⊢ 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!

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

⊢ a:Thing

├ a:Thing
└ b:Thing
Linear & Substructural Logic

```rust
sellsBurritos(store) ⊢ 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!

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

\[ a: \text{Thing} \]

\[ b: \text{Thing} \]

\[ c: \text{?} \]

Error \( (\not ⊢ c:?) \)
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
  \[ x \land \neg y \land z \]
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
Hoare Logic

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

\[ x \land \lnot y \land z \]

- 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?

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

```c
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?
  - Will I dereference a null pointer?

```c
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;
}
```

```c
y = \text{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?

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
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

![Diagram showing Hoare logic with precondition, command, and postcondition]
Hoare Logic

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

\[\{\varphi\} C \{\psi\}\]

- If \(\varphi\) holds before \(C\), \(\psi\) 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

\[
\{ \varphi \}\mathbf{C}\{ \psi \}
\]

- If $\varphi$ holds before C, $\psi$ will hold after

\[
\{ x=3 \land y=2 \} x \leftarrow 5 \{ x=5 \}
\]

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

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

\[
\{ \varphi \} C \{ \psi \}
\]

- If \( \varphi \) holds before \( C \), \( \psi \) will hold after

\[
\{ x=3 \land y=2 \} x \leftarrow 5 \{ x=5 \}
\]

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

\[
\{ \# t \} x \leftarrow 5 \{ x=5 \}
\]
Hoare Logic

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

\[
\{\phi\}C\{\psi\}
\]

- If \(\phi\) holds before \(C\), \(\psi\) will hold after

\[
\{x=3 \land y=2\}x \leftarrow 5\{x=5\}
\]

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

\[
\{#t\}x\leftarrow 5\{x=5\}
\]

\[
\{? \? \? \? \} \text{if } c \text{ then } x\leftarrow 5\{x=5\}\]
Hoare Logic

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

\{ \phi \} C \{ \psi \}

- If $\phi$ holds before $C$, $\psi$ will hold after

\{ x = 3 \land \}

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

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

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

You already have an *intuition* for weakest preconditions
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 $\sigma$ is a partial function mapping variables to values
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

- A store/state $\sigma$ 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 $\sigma$ 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 $\sigma$ is a partial function mapping variables to values
  - Commands in a program can modify the store

\[
\begin{align*}
\text{Store} & \quad \text{Command} \\
\sigma = \{x \mapsto 3, \ y \mapsto 1\} & \quad x \leftarrow 5
\end{align*}
\]
Hoare Logic – weakest preconditions

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

\[
\begin{align*}
\text{Store} & : \sigma = \{ x \mapsto 3, \ y \mapsto 1 \} \\
\text{Command} & : x \leftarrow 5 \\
\text{New Store} & : \sigma = \{ x \mapsto 5, \ y \mapsto 1 \}
\end{align*}
\]
Hoare Logic – weakest preconditions

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

<table>
<thead>
<tr>
<th>Store</th>
<th>Command</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma = {x \mapsto 3, y \mapsto 1}$</td>
<td>$x \leftarrow 5$</td>
<td>${x=5}$</td>
</tr>
<tr>
<td>$\sigma = {x \mapsto 5, y \mapsto 1}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hoare Logic – weakest preconditions

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

<table>
<thead>
<tr>
<th>Store</th>
<th>Command</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma = {x \mapsto 3, \ y \mapsto 1}$</td>
<td>$x \leftarrow 5$</td>
<td>${x=3 \land y=2}$</td>
</tr>
<tr>
<td>$\sigma = {x \mapsto 5, \ y \mapsto 1}$</td>
<td></td>
<td>${x=5}$</td>
</tr>
</tbody>
</table>
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

- A store/state $\sigma$ is a partial function mapping variables to values
  - Commands in a program can modify the store.

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

This was technically true, but not so useful
(...or even compatible with our states)
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

- A store/state $\sigma$ is a partial function mapping variables to values
  - Commands in a program can modify the store

<table>
<thead>
<tr>
<th>Store</th>
<th>Command</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma = {x \mapsto 3, y \mapsto 1}$</td>
<td>$x \leftarrow 5$</td>
<td>${x=3 \land y=2}$</td>
</tr>
<tr>
<td>$\sigma = {x \mapsto 5, y \mapsto 1}$</td>
<td></td>
<td>${x=5}$</td>
</tr>
</tbody>
</table>

- $\sigma \in \Sigma$ (all possible states), and we can reason about subsets of $\Sigma$
Hoare Logic – weakest preconditions

• What do we really mean by captures all states?

\[ \sigma = \{ x \mapsto 3, \ y \mapsto 1 \} \]

*Example Diagram:*

- Initial state: \( x \leftarrow 3, \ y \leftarrow 1 \)
- Transition: \( x \leftarrow 5 \)
- Final state: \( x \leftarrow 5, \ y \leftarrow 1 \)
What do we really mean by captures all states?

\[ \sigma = \{x \mapsto 3, \ y \mapsto 1\} \]

\[ \sigma = \{x \mapsto 5, \ y \mapsto 1\} \]
What do we really mean by captures all states?

\[ \sigma = \{ x \mapsto 3, \ y \mapsto 1 \} \]

\[ \sigma = \{ x \mapsto 5, \ y \mapsto 1 \} \]
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

\[ \sigma = \{ x \mapsto 3, \ y \mapsto 1 \} \]

Each set of states corresponds to a condition defining the set

\[ \{ x = 5 \} \]

\[ \sigma = \{ x \mapsto 5, \ y \mapsto 1 \} \]
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

\[ \sigma = \{x \mapsto 3, \; y \mapsto 1\} \]

\[ \{x=3\} \]

\[ x \leftarrow 5 \]

\[ \sigma = \{x \mapsto 5, \; y \mapsto 1\} \]

\[ \{x=5\} \]
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

\[ \sigma = \{ x \mapsto 3, \ y \mapsto 1 \} \]

\[ \{ x = 3 \} \]

\[ x \leftarrow 5 \]

\[ \sigma = \{ x \mapsto 5, \ y \mapsto 1 \} \]

\[ \{ x = 5 \} \]

Commands map sets to sets
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

\[ \sigma = \{ x \mapsto 3, \ y \mapsto 1 \} \]

\[ x = 3 \]

\[ x = 7 \]

\[ x \leftarrow 5 \]

\[ \sigma = \{ x \mapsto 5, \ y \mapsto 1 \} \]
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

\[ \sigma = \{ x \mapsto 3, \ y \mapsto 1 \} \]

\[ \{ \#t \} \]

\[ x \leftarrow 5 \]

\[ \sigma = \{ x \mapsto 5, \ y \mapsto 1 \} \]

\[ \{ x = 5 \} \]

All states lead to the postcondition!
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

σ={x→3, y→1}

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?
- \[ \text{wp}(C, \psi) = \bigcup \{x \mid \{x\} C \{\psi\}\} \]
  - Where \((A \rightarrow B) \models (A < B)\)
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

\[ \text{wp}(C, \psi) = \bigcup \{x \mid \{x\} C \{\psi\}\} \]
- Where \((A \rightarrow B) \vdash (A \nLeftarrow B)\)

\[ x \leftarrow 5 \]
\[ \{x=5\} \]
\[ \psi = \{x=5\} \]
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

\[
\text{wp}(C, \psi) = \biguplus \{x \mid \{x\} C \{\psi\}\}
\]

- Where \((A \rightarrow B) \vdash (A < B)\)

\[
 x \leftarrow 5
\]

\[
\{x=3\}
\]

\[
\{x=5\}
\]

\[
\psi = \{x=5\}
\]
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

\[ \text{wp}(C, \psi) = \bigcup \{ x \mid \{ x \} C \{ \psi \} \} \]
- Where \((A \rightarrow B) \vdash (A \nless B)\)

\[
\begin{align*}
\{x=3\} \sqcup \{x=4\} &= \? \\
\psi &= \{x=5\}
\end{align*}
\]

\[x \leftarrow 5\]
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

\[ \text{wp}(C, \psi) = \bigcup \{x \mid \{x\} C \{\psi\}\} \]

- Where \((A \rightarrow B) \models (A < B)\)

\[
\begin{align*}
\{x=3\} & \sqcup \{x=4\} \\
= & \text{?} \\
\{x=3\} & \rightarrow \{x=3 \lor x=4\} \\
\{x=4\} & \rightarrow \{x=3 \lor x=4\}
\end{align*}
\]

\(\psi = \{x=5\}\)
What do we really mean by captures all states?

\[ \text{wp}(C, \psi) = \bigcup \{ x \mid \{ x \} C \{ \psi \} \} \]

- Where \((A \rightarrow B) \vdash (A \land B)\)

\[
\begin{align*}
\{x=3\} \sqcup \{x=4\} & = \{x=3 \lor x=4\} \\
\{x=3\} \rightarrow \{x=3 \lor x=4\} \\
\{x=4\} \rightarrow \{x=3 \lor x=4\}
\end{align*}
\]

\(\psi = \{x=5\}\)
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

\[ \text{wp}(C, \psi) = \bigcup \{ x \mid \{ x \} C \{ \psi \} \} \]
- Where \((A \rightarrow B) \vdash (A < B)\)

\[
\{ x=5 \} \bigcup \{ x=4 \} \bigcup \{ x=5 \} = \{ 3 \leq x \land x \leq 5 \}
\]

\[
\{ x=3 \} \rightarrow \{ 3 \leq x \land x \leq 5 \}
\]
\[
\{ x=4 \} \rightarrow \{ 3 \leq x \land x \leq 5 \}
\]
\[
\{ x=5 \} \rightarrow \{ 3 \leq x \land x \leq 5 \}
\]

\[\psi = \{ x=5 \}\]
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

\[ \text{wp}(C, \psi) = \bigcup \{x \mid \{x\} C \{\psi\}\} \]
- Where \((A \rightarrow B) \models (A < B)\)

\[
\{x=3\} \bigcup \{x=4\} \bigcup \{x=5\} \bigcup \ldots = \{\#T\}
\]

\[
\{x=3\} \rightarrow \{\#T\} \\
\{x=4\} \rightarrow \{\#T\} \\
\ldots \\
\psi = \{x=5\}
\]
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

\[ \text{wp}(C, \psi) = \bigcup \{ x \mid \{ x \} C \{ \psi \} \} \]
- 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?

- \( \text{wp}(C, \psi) = \bigcup \{x \mid \{x\} C \{\psi\}\} \)
  - Where \((A \rightarrow B) \models \neg (A \land B)\)

- Technically, these are **Weakest Sufficient Preconditions**
Hoare Logic

- What do we really mean by captures all states?

- $\text{wp}(C, \psi) = \bigcup \{x \mid \{x\} C \{\psi\}\}$
  - Where $(A \rightarrow B) \vdash (A \prec B)$

- Technically, these are *Weakest Sufficient Preconditions*

- We may also consider/compute other relationships
Hoare Logic

- What do we really mean by captures all states?

\[ \text{wp}(C, \psi) = \bigcup \{x \mid \{x\} C \{\psi\}\} \]
- 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?

- \( \text{wp}(C, \psi) = \bigcup \{x \mid \{x\} C \{\psi\}\} \)
  - 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 \(\phi\) lead to \(\psi\)?

“Given \(\psi\), what must be true for it to hold?”
Hoare Logic

- What do we really mean by captures all states?
  \[ \text{wp}(C, \psi) = \bigcup \{x \mid \{x\} C \{\psi\}\} \]
  - Where \((A \rightarrow B) \vdash (A \Rightarrow B)\)

- Technically, these are Weakest Sufficient Preconditions

- We may also consider/compute other relationships
  - Weakest Sufficient Preconditions
  - Strongest Necessary Postconditions (snp)

What states \(\psi\) must \(\phi\) lead to?

“Given \(\phi\), what is guaranteed when it holds?”
Hoare Logic

- What do we really mean by captures all states?

\[ \text{wp}(C, \psi) = \bigcup \{x | \{x\} C \{\psi\}\} \]
  - \text{Where } (A \rightarrow B) \vdash (A \land 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 \(\phi\) lead to \(\psi\)?**

“Given \(\psi\), what if false at \(\phi\) would exclude it?”
Hoare Logic

- What do we really mean by captures all states?
- \[ \text{wp}(C, \psi) = \bigcup \{x | \{x\} C \{\psi}\} \]
  - Where \((A \rightarrow B) \vdash (A < B)\)
- Technically, these are \textit{Weakest Sufficient Preconditions}
- We may also consider/compute other relationships
  - Weakest Sufficient Preconditions
  - Strongest Necessary Postconditions
  - Strongest Necessary Preconditions (snpre)
- What states \(\varphi\) lead to \(\psi\)?
- Then how does this differ from \(wsp\)?
Hoare Logic

- What do we really mean by captures all states?

\[ \text{wp}(C, \psi) = \bigcup \{x \mid \{x\} C \{\psi\}\} \]

- 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

\[
\begin{align*}
\text{WSP} & : \varphi \text{@pre} \rightarrow \psi \text{@post} \\
\text{SNPre} & : \varphi \text{@pre} \leftarrow \psi \text{@post}
\end{align*}
\]
Hoare Logic

- What do we really mean by captures all states?
  \[ \text{wp}(C, \psi) = \bigcup \{x \mid \{x\} \vdash C \{\psi\}\} \]
  - Where \((A \rightarrow B) \not\vdash (A \Rightarrow B)\)

- Technically, these are **Weakest Sufficient Preconditions**

- We may also consider/compute other relationships
  - Weakest Sufficient Preconditions
  - Strongest Necessary Postconditions
  - Strongest Necessary Preconditions

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

- What do we really mean by captures all states?
- \( \text{wp}(C, \psi) = \bigcup \{ x \mid \{ x \} C \{ \psi \} \} \)
  - 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

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

- What do we really mean by captures all states?
  \[ \text{wp}(C, \psi) = \bigcup \{ x \mid \{ x \} C \{ \psi \} \} \]
  - Where \((A \rightarrow B) \vdash (A \land \neg B)\)

- 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?
  - \( \text{wp}(C, \psi) = \bigcup \{x | \{x\} C \{\psi}\} \)
    - Where \((A \rightarrow B) \vdash (A < B)\)
- Technically, these are \textbf{Weakest Sufficient Preconditions}
- We may also consider/compute other relationships
  - Weakest Sufficient Preconditions
  - Strongest Necessary Postconditions
  - Strongest Necessary Preconditions
  - Weakest \textit{Liberal} Preconditions

What states \(\phi\) lead to \(\psi\) or \textit{do not terminate}?
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

\[ \text{wp}(x \leftarrow E, \psi) = [E/x] \psi \]
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

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

\[
\begin{array}{c}
x \leftarrow a + b \\
\{x<5\}
\end{array}
\]
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

\[
wp(x \leftarrow E, \psi) = [E/x] \psi \\
\{a + b < 5\} \\
x \leftarrow a + b \\
\{x < 5\}
\]
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions
  
  \[ \text{wp}(x \leftarrow E, \psi) = [E/x] \psi \]

  \[ \text{wp}(S; T, \psi) = \text{wp}(S, \text{wp}(T, \psi)) \]
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

\[
\begin{align*}
wp(x \leftarrow E, \psi) &= [E/x] \psi \\
wp(S; T, \psi) &= wp(S, wp(T, \psi))
\end{align*}
\]

\[
\begin{align*}
\{ \ ??? \} \\
b &\leftarrow 7; \\
x &\leftarrow a + b \\
\{x<5\}
\end{align*}
\]
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions
  \[ wp(x \leftarrow E, \psi) = [E/x] \psi \]
  \[ wp(S; T, \psi) = wp(S, wp(T, \psi)) \]

\[
\begin{align*}
\{ & \text{???, } \\
& b \leftarrow 7; \\
& \{a + b < 5\} \\
& x \leftarrow a + b \\
& \{x < 5\}
\end{align*}
\]
Inference rules for weakest preconditions

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

\[ \text{wp}(S; T, \psi) = \text{wp}(S, \text{wp}(T, \psi)) \]

\{a + 7 < 5\}

\text{b} \leftarrow 7; \quad \{a + b < 5\}

\text{x} \leftarrow \text{a} + \text{b} \quad \{\text{x}<5\}
Inference rules for weakest preconditions

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

\[
wp(S; T, \psi) = wp(S, wp(T, \psi))
\]

\[
wp(\text{if } B \text{ then } S \text{ else } T, \psi) = B \rightarrow wp(S, \psi) \land \neg B \rightarrow wp(T, \psi)
\]
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions
  
  \[ wp(x \leftarrow E, \psi) = [E/x] \psi \]
  
  \[ wp(S; T, \psi) = wp(S, wp(T, \psi)) \]
  
  \[ wp(\text{if } B \text{ then } S \text{ else } T, \psi) \]
  
  \[ = B \rightarrow wp(S, \psi) \land \neg B \rightarrow wp(T, \psi) \]
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions
  \[
  \begin{align*}
  \text{wp}(x \leftarrow E, \psi) &= [E/x] \psi \\
  \text{wp}(S; T, \psi) &= \text{wp}(S, \text{wp}(T, \psi)) \\
  \text{wp(\text{if } B \text{ then } S \text{ else } T, \psi)} &= B \rightarrow \text{wp}(S, \psi) \land \neg B \rightarrow \text{wp}(T, \psi)
  \end{align*}
  \]

  if c then
d = y + 2
else
d = y + 5
x/d
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions
  \[ \text{wp}(x \leftarrow E, \psi) = \left[ E/x \right] \psi \]
  \[ \text{wp}(S; T, \psi) = \text{wp}(S, \text{wp}(T, \psi)) \]
  \[ \text{wp}(\text{if } B \text{ then } S \text{ else } T, \psi) = B \rightarrow \text{wp}(S, \psi) \land \neg B \rightarrow \text{wp}(T, \psi) \]

\[
\begin{array}{c}
\text{if } c \text{ then} \\
\text{ } \\
\text{ } \\
\text{else} \\
\text{ } \\
\text{if c then} \\
\begin{array}{c}
\text{d }= y + 2 \\
\text{else} \\
\text{d }= y + 5 \\
x/d
\end{array}
\end{array}
\]

\[
\{d \neq 0\}
\]
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions
  \[ wp(x \leftarrow E, \psi) = \left[ E/x \right] \psi \]
  \[ wp(S; T, \psi) = wp(S, wp(T, \psi)) \]
  \[ wp(\text{if } B \text{ then } S \text{ else } T, \psi) = B \rightarrow wp(S, \psi) \land \neg B \rightarrow wp(T, \psi) \]

\[
\begin{array}{l}
\text{if } c \text{ then } \\
\quad d = y + 2 \\
\text{else } \\
\quad d = y + 5 \\
x/d \end{array} \quad \begin{cases} 
\{ ??? \} \\
\{ y + 2 \neq 0 \} \\
\{ y + 5 \neq 0 \} \\
\{ d \neq 0 \} 
\end{cases}
\]
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions
  
  \[ wp(x \leftarrow E, \psi) = [E/x] \psi \]
  
  \[ wp(S; T, \psi) = wp(S, wp(T, \psi)) \]
  
  \[ wp(\text{if } B \text{ then } S \text{ else } T, \psi) = B \rightarrow wp(S, \psi) \land \neg B \rightarrow wp(T, \psi) \]

\[
\begin{align*}
\text{if } c \text{ then } d &= y + 2 \\
\text{else } d &= y + 5
\end{align*}
\]

\[
\begin{align*}
x/d &\quad \{c \rightarrow y+2 \neq 0 \land \neg c \rightarrow y+5 \neq 0\} \\
&\quad \{y+2 \neq 0\} \\
&\quad \{y+5 \neq 0\} \\
&\quad \{d \neq 0\}
\end{align*}
\]
Hoare Logic

- Careful points
  - Redefinition of variables

Pre: \{a < 5, c < 2\}

\[
\begin{align*}
\text{b} & = \text{a} + 2 \\
\text{a} & = 3 \times c
\end{align*}
\]

Post: \{??\}
Hoare Logic

- Careful points
  - Redefinition of variables

\[
\begin{align*}
\text{Pre: } & \{a < 5, \; c < 2\} \\
\text{b} & = a + 2 \\
\text{a} & = 3*c \\
\text{Post: } & \{??\}
\end{align*}
\]

It can be necessary to rename variables that are redefined.
• **Careful points**
  - Redefinition of variables

  Pre: \(\{a < 5, c < 2\}\)
  
  \[
  \begin{align*}
  b & = a + 2 \\
  a & = 3 \times c
  \end{align*}
  \]

  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

\{φ\} C \{ψ\} \text{ while } B \text{ do } S \text{ done}
Hoare Logic

- Careful points
  - Redefinition of variables
  - Pointers
  - Loops

\[
\begin{align*}
&\{\varphi\}\mathcal{C}\{\psi\} \\
&\text{while } B \text{ do } S \text{ done}
\end{align*}
\]

\[
\text{Inv} \land \neg B \rightarrow \psi \quad \text{exit}
\]
Hoare Logic

- Careful points
  - Redefinition of variables
  - Pointers
  - Loops

\[
\{ \phi \} C \{ \psi \} \quad \text{while } B \text{ do } S \text{ done}
\]

\[
\{ \text{Inv } \land \neg B \rightarrow \psi \} \quad \text{exit}
\]

\[
\{ \text{Inv } \land B \} \quad S \quad \{ \text{Inv} \} \quad \text{continue}
\]
Hoare Logic

- **Careful points**
  - Redefinition of variables
  - Pointers
  - Loops

\[
\{\varphi\} C \{\psi\} \quad \text{while } B \text{ do } S \text{ done}
\]

\[
\{\text{Inv} \land \neg B \rightarrow \psi\} \quad \text{exit}
\]

\[
\{\text{Inv} \land B\} \quad S \quad \{\text{Inv}\} \quad \text{continue}
\]

\[
\{\varphi \rightarrow \text{Inv}\} \quad \text{enter}
\]
Hoare Logic

- Careful points
  - Redefinition of variables
  - Pointers
  - Loops

\[
\{\varphi\} \text{C}\{\psi\} \quad \text{while } B \text{ do } S \text{ done}
\]

\[
\{\text{Inv} \land \neg B \rightarrow \psi\} \quad \text{exit}
\]

\[
\{\text{Inv} \land B\} \quad S \quad \{\text{Inv}\} \quad \text{continue}
\]

\[
\{\varphi \rightarrow \text{Inv}\} \quad \text{enter}
\]

Automatically inferring such invariants is used for verifying safe:
- avionics
- machine learning
- ...
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 $\langle\rangle$ 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 $\langle \rangle$ 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 $\ast$.
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 *

\[
\{x \mapsto y \ast y \mapsto x\} x = z\{x \mapsto z \ast y \mapsto x\}
\]

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 $*$

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

Suppose we used $\land$ 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 $*$.

\[
\{x\mapsto y \ast y\mapsto x\}x = z\{x\mapsto z \ast y\mapsto x\}
\]

- 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

\[
\{ \varphi \} C \{ \psi \} \\
\{ \varphi \ast r \} C \{ \psi \ast r \}
\]
Separation Logic

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

\[
\begin{align*}
\{\varphi\} & C \{\psi\} \\
\hline
\{\varphi \ast r\} & C \{\psi \ast r\}
\end{align*}
\]

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

Can accessing the field g cause a null pointer exception?

[Margoor & Komondoor, 2015]
Discovering & Disproving Bugs

```java
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 ∧ d≠null}
{b.f=null}

[Margoor & Komondoor, 2015]
foo(a,b,c) {
    if (a != null) {
        b = c;
        t = new...;
        c.f = t;
    }
    d = a;
    if (d != null) {
        b.f.g = 10;
    }
}
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} v {b=c ∧ t=null ∧ a≠null}
{b.f=null ∧ a≠null}
{b.f=null ∧ d≠null}
{b.f=null}

[Margoor & Komondoor, 2015]
foo(a,b,c) {
  if (a != null) {
    b = c;
    t = new...;
    c.f = t;
  }
  d = a;
  if (d != null) {
    b.f.g = 10;
  }
}

{b\neq c \land b.f=\text{null} \land a\neq\text{null}}
{(b\neq c \land b.f=\text{null} \land a\neq\text{null}) \lor (b=c \land t=\text{null} \land a\neq\text{null})}
{b.f=\text{null} \land a\neq\text{null}}
{b.f=\text{null} \land d\neq\text{null}}
{b.f=\text{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 ≠ 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}
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\text{null} \rightarrow \#f) \lor (a=\text{null} \rightarrow b.f=\text{null} \land a\neq\text{null})}

{\#f}

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

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

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

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

{b.f=\text{null}}

[Margoor & Komondoor, 2015]
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}

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

Safe!

[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!=null → #f) v (a=null → b.f=null ∧ a!=null)} = #f

Note: this can be automated within a tool!
Localizing Bugs

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

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

assert(0 ≤ i < 3) should hold
assert(0 ≤ i < 3) should hold

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

assert(0 \leq i < 3) \text{ should hold}

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

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]);
```
Localizing Bugs

index₁ = 1

∧ (0 ≤ 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

\[
\text{assert}(0 \leq i < 3)
\]

should hold

When the starting index is 1, \(i\) is out of bounds

[Jose & Majumdar, 2011]

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

We will generate constraints in the forward direction

assert(0 \leq i < 3) should hold

When the starting index is 1, \(i\) is out of bounds
Localizing Bugs

index₁ = 1
∧ guard₁ = (index₁ ≠ 1)
∧ index₂ = 2
∧ (0 ≤ 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

index₁ = 1
∧ guard₁ = (index₁ ≠ 1)
∧ index₂ = 2
∧ index₃ = (index₁ + 2)
∧ (0 ≤ 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

\[
\begin{align*}
\text{index}_1 &= 1 \\
\land \text{guard}_1 &= (\text{index}_1 \neq 1) \\
\land \text{index}_2 &= 2 \\
\land \text{index}_3 &= (\text{index}_1 + 2) \\
\land (\text{guard}_1 \rightarrow i=\text{index}_2) \\
\land (\neg\text{guard}_1 \rightarrow i=\text{index}_3) \\
\land (0 \leq i < 3)
\end{align*}
\]

Code snippet:

```c
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

\[ \text{assert}(0 \leq i < 3) \text{ should hold}\]

When the starting index is 1, \(i\) is out of bounds

\[
\begin{align*}
\text{index}_1 &= 1 \\
\land \text{guard}_1 &= (\text{index}_1 \neq 1) \\
\land \text{index}_2 &= 2 \\
\land \text{index}_3 &= (\text{index}_1 + 2) \\
\land (\text{guard}_1 \rightarrow i = \text{index}_2) \\
\land (\neg \text{guard}_1 \rightarrow i = \text{index}_3) \\
\land (0 \leq i < 3)
\end{align*}
\]

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

[Jose & Majumdar, 2011]
Localizing Bugs

assert(0 ≤ i < 3) should hold

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

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

i = index;
print(arr[i]);

assert(0 ≤ i < 3) should hold

Index:

index₁ = 1
∧ guard₁ = (index₁ ≠ 1)
∧ index₂ = 2
∧ index₃ = (index₁ + 2)
∧ (guard₁ → i=index₂)
∧ (¬guard₁ → i=index₃)
∧ (0 ≤ i < 3)

snp(P,#t)
Localizing Bugs

\[
\begin{align*}
\text{index}_1 &= 1 \\
\text{guard}_1 &= (\text{index}_1 \neq 1) \\
\text{index}_2 &= 2 \\
\text{index}_3 &= (\text{index}_1 + 2) \\
(\text{guard}_1 \rightarrow \text{i} = \text{index}_2) \\
(\neg \text{guard}_1 \rightarrow \text{i} = \text{index}_3) \\
(0 \leq \text{i} < 3)
\end{align*}
\]

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

\[
\begin{align*}
\text{int arr}[3]; \\
... \\
\text{if (index} \neq 1) 
\{ \\
\quad \text{index} = 2; \\
\} \text{ else } 
\{ \\
\quad \text{index} = \text{index} + 2; \\
\}
\text{i} = \text{index}; \\
\text{print(arr[i]}); \\
\text{assert(0} \leq \text{i} < 3) \text{ should hold}
\end{align*}
\]

When the starting index is 1, \(i\) is out of bounds
Localizing Bugs

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

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

These constraints define our goal, so they are essential

[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\(_3\)) \land (0 \leq i < 3)\]

assert(0 \leq i < 3) should hold

[Jose & Majumdar, 2011]
Localizing Bugs

assert(0 ≤ i < 3) should hold when the starting index is 1, i is out of bounds.

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

Some of these constraints conflict with our goal.

These constraints define our goal, so they are essential.

index₁ = 1
∧ guard₁ = (index₁ ≠ 1)
∧ index₂ = 2
∧ index₃ = (index₁ + 2)
∧ (guard₁ → i=index₂)
∧ (¬guard₁ → i=index₃)
∧ (0 ≤ i < 3)

[Jose & Majumdar, 2011]
Localizing Bugs

assert(0 ≤ i < 3) should hold

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

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

assert(0 ≤ \(i\) < 3) should hold

When the starting index is 1, \(i\) is out of bounds

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

These constraints define our goal, so they are essential

Some of these constraints conflict with our goal

[Jose & Majumdar, 2011]
Localizing Bugs

When the starting index is 1, \( i \) is out of bounds

\[ \text{arr}[3]; \]

\[
\begin{align*}
\text{if} \ (\text{index} \neq 1) \ {\{} \\
\quad \text{index} = 2; \\
\quad \text{index} = \text{index} + 2; \\
\text{print(arr[1])}; \\
\text{assert(}0 \leq i < 3\text{) should hold}
\end{align*}
\]

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

Must SAT

[Jose & Majumdar, 2011]
Localizing Bugs

assert(0 ≤ i < 3) should hold

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

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

These constraints define our goal, so they are essential

Some of these constraints conflict with our goal

Must SAT → Max # satisfiable

index₁ = 1
∧ guard₁ = (index₁ ≠ 1)
∧ index₂ = 2
∧ index₃ = (index₁ + 2)
∧ (guard₁ → i=index₂)
∧ (¬guard₁ → i=index₃)
∧ (0 ≤ i < 3)

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

assert(0 ≤ i < 3) should hold when the starting index is 1, i is out of bounds

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

[Jose & Majumdar, 2011]

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

Could not SAT; Blame for inconsistency

Must SAT Max # satisfiable

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

index₁ = 1
∧ guard₁ = (index₁ ≠ 1)
∧ index₂ = 2
∧ index₃ = (index₁ + 2)
∧ (guard₁ → i=index₂)
∧ (¬guard₁ → i=index₃)
∧ (0 ≤ i < 3)
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)