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 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:

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:
Order Theory

- *Order theory* is a field examining how we compare elements of a set.
- Simplest example is numbers on a number line:
  
  \[ \leq \text{ is a total order on } \mathbb{Z}. \]
  
  - 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, ...

![Graph showing an order relation with a horizontal axis and a vertical axis, with points labeled 0, 1, 2, 3, 4 on the horizontal axis and 0, 1, 2 on the vertical axis. The graph illustrates an order relation between the points.]
We often want to compare complex data
- Ordinal, multidimensional, ...
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, ...
We often want to compare complex data
  - Ordinal, multidimensional, ...

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

- 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: $a \leq b \& b \leq c \Rightarrow a \leq c$
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 $\leq$ 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 $\sqsubseteq$
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 $\forall$ $a, b, c \in S$

  \[
  a \leq_{\text{str}} xabyz \\
  a \leq_{\text{seq}} xaybz \\
  \{a, b\} \subseteq \{a, b, x, y, z\}
  \]

- 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 ≤ a
  - Antisymmetric: a ≤ b & b ≤ a ⇒ a = b
  - Transitive: a ≤ b & b ≤ c ⇒ a ≤ c

- When reasoning about partial orders, we prefer ⊑.

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

- A relation $\leq$ is a **partial order** on a set $S$ if $\forall \ a, b, c \in S$

  \[
  (1,1) \sqsubseteq (1,2) \\
  (1,1) \sqsubseteq (2,2)
  \]

- 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 $\sqsubseteq$

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

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

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

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

- 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)
Partial Orders

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

- We can express the structure of partial orders as *(semi-)*lattices.
Partial Orders

- We can express the structure of partial orders as *(semi-)lattices.*
Partial Orders

- We can express the structure of partial orders as \((semi-)lattices\).
Partial Orders

- We can express the structure of partial orders as (semi-)lattices.
Partial Orders

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

- If unique least/greatest elements exist, we call them \( \bot \)(bottom)/\( \top \)(top).

\[
\begin{align*}
(0,0) &\quad (0,1) &\quad (0,2) \\
(1,0) &\quad (1,1) &\quad (1,2) \\
(2,0) &\quad (2,1) &\quad (2,2)
\end{align*}
\]
Partial Orders

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

- We are often interested in upper and lower bounds.
  - A *join* $a \sqcup b$ 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 $b$

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** \( a \lor b \) is the least upper bound of \( a \) and \( b \)
- A **meet** \( a \land b \) is the greatest lower bound of \( a \) and \( b \)

What is \( (0,1) \land (1,0) \)?
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 $b$
  - Bounds must be unique and may not exist.
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.
- $\forall S' \subseteq S$, 

```
(0,0) → (0,1) → (1,0) → (2,0) → (2,2)
```

```plaintext
(0,0) → (0,1) → (1,0) → (1,1) → (2,1) → (2,2)
```

```
(0,0) → (0,2) → (1,2) → (2,2)
```
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.
  - $\forall S' \subseteq S, \exists \sqcup S' \& \sqcap S' \Rightarrow \text{lattice}$
Partial Orders

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

\[
\begin{array}{ccccccc}
(0,0) & \rightarrow & (0,1) & \rightarrow & (1,0) & \rightarrow & (1,1) & \rightarrow & (2,0) \\
(0,1) & \rightarrow & (0,2) & \rightarrow & (1,2) & \rightarrow & (2,1) & \rightarrow & (2,2)
\end{array}
\]
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.
  - $\forall S' \subseteq S$, $\exists S'$ & $\sqcap S' \Rightarrow$ lattice, $\exists S'$ or $\sqcup S' \Rightarrow$ 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.

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

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

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

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

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

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

- 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
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 a search space
Regular Languages & Finite Automata

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

\[
\text{regex} \rightarrow \text{symbol} \\
\left| \left( `\text{regex}` `)` \right| \text{regex} `*` \\
\left| \text{regex} `|` `\text{regex}` \right|
\]
Regular Languages & Finite Automata

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

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

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.
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 \( abccdbce \)
A regular language can be expressed via a regular expression.

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

For example, the regular expression \( a(bc | cd)^*e \) recognizes the language \( L \) 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. \( a(bc \mid cd)^*e \) 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.

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

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

\[ce^n gf^n d\]
• Context free grammars add recursion and enable Dyck language recognition

Start = A
A → cBd
B → eBf
| g

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

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

Start = A
A → cBd
B → eBf
| g

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

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

```
Start = A
A → cBd
B → eBf | g
```

```
            A
           / \     / \
          c   B   d
          |     |   |
          e   B   f
```

$ce^n gf^n d$
Context Free Grammars & Pushdown Automata

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

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

  \[
  cBd^n g f^n d
  \]

  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

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

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

\[
\text{regex } \rightarrow \text{ symbol} \\
\phantom{regex } \quad \left| \ (` \text{regex } `) ` \right. \\
\phantom{regex } \quad \left| \ ` \text{regex } `* ` \right. \\
\phantom{regex } \quad \left| \ ` \text{regex } `| ` \text{regex} \right.
\]
Context Free Grammars & Pushdown Automata

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

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

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

**Grammar**

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

**Diagram**

- Transition from **S** to **x**
- Transition from **A** to **a**
- Transition from **B** to **b**
- Transition from **A** to **t**
- Transition from **B** to **u**
- Transition from **x** to **A**
- Transition from **y** to **B**
- Transition from **z** to **B**
- Transition from **t** to **A**
- Transition from **u** to **B**

**Stack**

- Initial stack: **xaatyy**
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*).

The grammar rules are:

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

The automaton transitions and stack are illustrated in the diagram. The input string is `xaaty`. The states and transitions are indicated with arrows labeled `x`, `y`, `z`, `t`, and `u`. The stack transitions are shown on the right side of the diagram.
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 states and transitions](image)
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)

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

\[
S \rightarrow xAy \mid zB
\]

\[
A \rightarrow aA \mid t
\]

\[
B \rightarrow bB \mid u
\]
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*).

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*)
- *Context free grammars* play a key role in
  - Precise static program analysis
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 play a key role in
  - Precise static program analysis
  - Program synthesis
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* play a key role in
  - Precise static program analysis
  - Program synthesis
  - Prediction and machine learning on programs
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
  - ¬ (Not), ∧ (And), ∨ (or)
  - → (Implies), ↔ (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
  - → (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. father(x), 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 judgement (\( \vdash \))
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 judgement ($\models$)

$I \models x$ and $I \models y$ iff $I \models x \land y$
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 $\iff \exists I (I \vdash s)$

- **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
Classical Logic

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

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

- 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 judgements enable others

\[
\frac{\Gamma \vdash x \quad \Delta \vdash y}{\Gamma, \Delta \vdash x \land y}
\]
Inference using classical logic

- Rules express how some judgements enable others

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

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

- Rules express how some judgements enable others

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

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

- Rules express how some judgements 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 judgements enable others

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

\[ \Gamma, \Delta \vdash x \land y \]

- Proofs can be written by stacking rules

\[ A \vdash A \]
\[ A \vdash B \vdash A \to B \]
\[ \text{Id} \]
\[ A \vdash A \]
\[ A \to B, A \vdash B \]
\[ \text{Id} \]
\[ A \vdash A \]
\[ \rightarrow\text{-E} \]
\[ A, A \to B, A \vdash A \times B \]
\[ \times\text{-I} \]
\[ A \to B, A, A \vdash A \times B \]
\[ \text{Exchange} \]
\[ A \to B, A, A \vdash A \times B \]
\[ \text{Contraction} \]

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`?
Intutionistic & 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 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

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

Law of excluded middle
Intuitionistic & Constructive Logic

- Classic logic includes several rules including

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

Double negation elimination
Intuitionistic & Constructive Logic

- Classic logic includes several rules including

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

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

- Classic logic includes several rules including

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

- *Intuitionistic logic* excludes these to require direct evidence

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

\[
sells\text{Burritos}(store) \Rightarrow sell\text{Burrito}(me,store) \\
\text{has10Dollars}(me) \land \text{buy\text{Burrito}(me,store)} \\
\text{buy\text{Burrito}(me,store)} \\
\text{buy\text{Burrito}(me,store)} \\
\text{buy\text{Burrito}(me,store)}
\]

Classical & intuitionistic logic have trouble expressing consumable facts
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
sellsBurritos(store) \implies buyBurrito(me, store)

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

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

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

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

\[
\Gamma,A,A,\Delta \vdash p \\
\Gamma,\Delta \vdash p \\
\Gamma,A,A,\Delta \vdash p \\
\Gamma,\Delta \vdash p
\]

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

- 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) 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
Linear & Substructural Logic

sellsBurritos(store) \implies 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;
 posed a:Thing
```
Linear & Substructural Logic

sellsBurritos(store) \implies 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;
```

Error
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:

  \[ x \land \neg y \land z \]

- 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?

```java
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?

```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?
  - 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;
}

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?

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

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

Precondition
Hoare Logic

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

![Diagram of Hoare Logic]

Precondition \( \{ \phi \} \) Command \( \{ \psi \} \)
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
  \[ \{ \varphi \} C \{ \psi \} \]

- 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

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

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

\[{x=3 \land y=2}x \gets 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

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

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

\[
\{ \#t \} \text{x\leftarrow5} \{ \text{x=5} \}
\]
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 \( C \).

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

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

\[ \{ x=3 \land \#t \} x \leftarrow 5 \{ x=5 \} \]

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

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

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

\[
\begin{align*}
\text{Store} & \quad \text{Command} \\
\sigma = \{x \mapsto 3, \ y \mapsto 1\} & \quad x \gets 5
\end{align*}
\]
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

\[
\begin{align*}
\text{Store} & \quad \text{Command} \\
\sigma = \{x \mapsto 3, \ y \mapsto 1\} & \quad x \leftarrow 5 \\
\sigma = \{x \mapsto 5, \ y \mapsto 1\}
\end{align*}
\]
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>
• 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
  $\sigma = \{x \mapsto 3, \ y \mapsto 1\}$  $x \leftarrow 5$  $\{x=3 \land y=2\}$
  $\sigma = \{x \mapsto 5, \ y \mapsto 1\}$  $\{x=5\}$
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?

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

\[ x \leftarrow 5 \]

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

- What do we really mean by captures all states?

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

\[ x \leftarrow 5 \]

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

- What do we really mean by captures all states?

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

\[ x \leftarrow 5 \]

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

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

\[ \{x = 3\} \]

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

\[ \{x = 5\} \]

\[ x \leftarrow 5 \]

Commands map sets to sets
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

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

\[ \{ x = 3 \} \]

\[ \{ x = 7 \} \]

\[ x \leftarrow 5 \]

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

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

- What do we really mean by captures all states?

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

\[ \{ \#t \} \]

\[ x \leftarrow 5 \]

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

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

- What do we really mean by captures all states?

σ = {x→3, y→1}

{x=5}

σ = {x→5, 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?

- \( \text{wp}(C, \psi) = \sqcap \{x \mid \{x\} C \{\psi\}\} \)
  - 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 \((A \rightarrow B) \vdash (A < B)\)
Hoare Logic – weakest preconditions

- What do we really mean by captures all states?

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

Intuitively, B is at least as general as A
Hoare Logic – weakest preconditions

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

- What do we really mean by captures all states?

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

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

- Technically, these are \textit{Weakest Sufficient Preconditions}

- We may also consider/compute other relationships

\[ \varphi \]

\[ C \]

\[ \psi \]
Hoare Logic

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

Technically, these are **Weakest Sufficient Preconditions**

We may also consider/compute other relationships
- Weakest Sufficient Preconditions (wsp)
  - Weakest Sufficient Preconditions (wsp)

What states \(\varphi\) lead to \(\psi\)?
Hoare Logic

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

What states $\psi$ must $\phi$ lead to?
Hoare Logic

- What do we really mean by captures all states?

- \( wp(C, \psi) = \bigcap \{x \mid \{x\} C \{\psi\}\} \)
  - 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 \(\varphi\) lead to \(\psi\)?
Hoare Logic

- What do we really mean by captures all states?
  \[ \text{wp}(C, \psi) = \bigcap \{x | \{x\} C \{\psi\}\} \]
  - Where \((A \rightarrow B) \vDash (A<\neg B)\)

- 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 \(\phi\) lead to \(\psi\)?

Then how does this differ from \(\text{wsp}\)?
Hoare Logic

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

- 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?

• \( \text{wp}(C, \psi) = \bigwedge \{x \mid \{x\} C \{\psi\}\} \)
  
  – Where \((A \rightarrow B) \vdash (A < B)\)

• Technically, these are \textit{Weakest Sufficient Preconditions}

• \textbf{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! (they are duals)
What do we really mean by captures all states?

$\text{wp}(C, \psi) = \bigsqcup \{x \mid \{x\} C \{\psi\}\}$
- 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 $\varphi$ lead to $\psi$ or *do not terminate*?
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

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

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

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

- Inference rules for weakest preconditions

\[ \text{wp}(x \leftarrow E, \varphi) = \left[ E/x \right] \varphi \]

\{ a + b < 5 \}
\xleftarrow{} x \leftarrow a + b
\{ x < 5 \}
Hoare Logic – weakest preconditions

- Inference rules for weakest preconditions

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

- Inference rules for weakest preconditions

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

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

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

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

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

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

\[
\begin{align*}
  \{a + 7 < 5\} \\
  b \leftarrow 7; \\
  \{a + b < 5\} \\
  x \leftarrow a + b \\
  \{x < 5\}
\end{align*}
\]
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

\[
\begin{align*}
wp(x ← E, ψ) &= [E/x]ψ \\
wp(S; T, ψ) &= wp(S, wp(T, ψ)) \\
wp(\text{if } B \text{ then } S \text{ else } T, ψ) &= B → wp(S, ψ) \land ¬B → wp(T, ψ)
\end{align*}
\]

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

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

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

{???

\{d \neq 0\}
```
Hoare Logic – weakest preconditions

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

  
  If \( c \) then
  
  \[
  d = y + 2
  \]
  
  Else
  
  \[
  d = y + 5
  \]
  
  \[
  \frac{x}{d}
  \]
  
  \[
  \{ \text{???} \}
  \]
  
  \[
  \{ y+2 \neq 0 \}
  \]
  
  \[
  \{ y+5 \neq 0 \}
  \]
  
  \[
  \{ d \neq 0 \}
  \]
Hoare Logic – weakest preconditions

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

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\}

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

Post: \{??\}
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
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.
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 $\langle\rangle$ or arbitrarily many times $\langle\rangle$.
- *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 *

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

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 *:**
  - This allows compositional reasoning about software.

\[
\{x \mapsto y * y \mapsto x\} x = z \{x \mapsto z * y \mapsto x\}
\]
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 *:
  - This allows compositional reasoning about software.

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

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 *
  - This allows compositional reasoning about software.
    \[
    \{x \mapsto y \ast y \mapsto x\} x = z \{x \mapsto z \ast y \mapsto x\}
    \]

- Separation logic enables efficient compositional reasoning
  - It is the backbone of Facebook’s Infer engine!
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

\[
\{\varphi\} C \{\psi\} \\
\hline
\{\varphi * r\} C \{\psi * r\}
\]

- 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.
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]
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 ∧ 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;
    }
}
```

{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 ≠ 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;
    }
}

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

```java
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) ∨ ( 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;
    }
}

{a!=null → #f ∧ 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}

[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\text{null} \implies \#f \land a=\text{null} \implies \#f\} = \#f
\{\#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}\}

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 ∧ a=null → #f} = #f
{#f}
{b≠c ∧ b.f=null ∧ a≠null}
{(b≠c ∧ b.f=null ∧ a≠null) ∨ (b=c∧t=null ∧ a≠null)}
{b.f=null ∧ a≠null}
{b.f=null ∧ d≠null}
{b.f=null}

Safe!

Note: this can be automated within a tool!

[Margoor & Komondoor, 2015]
Localizing Bugs

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

[Jose & Majumdar, 2011]
Localizing Bugs

assert(0 \leq i < 3) should hold

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

[Jose & Majumdar, 2011]
Localizing Bugs

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

[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]);

We will generate constraints in the forward direction

[Jose & Majumdar, 2011]
Localizing Bugs

\[ \text{index}_1 = 1 \]

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

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\((0 \leq i < 3)\) should hold

When the starting index is 1, \(i\) is out of bounds
We will generate constraints in the forward direction

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

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

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

\[ \begin{align*}
\text{int arr}[3]; \\
\text{...}
\text{if} (\text{index} \neq 1) \{ \\
\text{index} = 2; \\
\} \text{ else } \{ \\
\text{index} = \text{index} + 2; \\
\} \\
i = \text{index}; \\
\text{print(arr}[i]);
\end{align*} \]

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

When the starting \text{index} is 1, \text{i} is out of bounds
Localizing Bugs

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

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]);
```

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

assert(0 ≤ i < 3) should hold

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

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

```
int arr[3];
... 
if (index != 1) {
    index = 2;
} else {
    index = index + 2;
}
\text{i} = \text{index};
\text{print(arr}[	ext{i}]));
assert(0 \leq \text{i} < 3) \text{ should hold}
```

When the starting index is 1,
\text{i} is out of bounds

[Jose & Majumdar, 2011]
Localizing Bugs

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

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

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

\[ \text{snp}(P, \#t) \]

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

[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;
}
index = index + 2;

This is always false, but we can use that!

[Jose & Majumdar, 2011]
Localizing Bugs

assert(0 \leq 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]);
```

These constraints define our goal, so they are essential

$\exists \text{ index}_1 = 1$

$\land \text{ guard}_1 = (\text{index}_1 \neq 1)$

$\land \text{ index}_2 = 2$

$\land \text{ index}_3 = (\text{index}_1 + 2)$

$\land (\text{guard}_1 \rightarrow i = \text{index}_2)$

$\land (\neg\text{guard}_1 \rightarrow i = \text{index}_3)$

$\land (0 \leq i < 3)$

[Jose & Majumdar, 2011]
Localizing Bugs

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

\[
\begin{align*}
\text{index}_1 &= 1 \\
\land \ guard_1 &= (\text{index}_1 \neq 1) \\
\land \ \text{index}_2 &= 2 \\
\land \ \text{index}_3 &= (\text{index}_1 + 2) \\
\land \ (guard_1 \rightarrow i=\text{index}_2) \\
\land \ (\neg guard_1 \rightarrow i=\text{index}_3) \\
\land \ (0 \leq i < 3)
\end{align*}
\]

These constraints define our goal, so they are essential.

Some of these constraints \textit{conflict} with our goal.

```c
if (index != 1) {
    index = 2;
    index = index + 2;
} else {
    index = index + 2;
}
```

\[ i = \text{index}; \]

\[ \text{print(arr[i])}; \]

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

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

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

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

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

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

\[
\begin{align*}
\text{if} (\text{index} \neq 1) \{ \\
\quad \text{index} = 2; \\
\quad \text{index} = \text{index} + 2; \\
\quad \text{print} (\text{arr}[i]); \\
\}\end{align*}
\]

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

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

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

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.

Some of these constraints conflict with our goal.

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 \leq i < 3) should hold
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]);

[Jose & Majumdar, 2011]

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

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

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

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]);
```

These constraints define our goal, so they are essential

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

Some of these constraints conflict with our goal

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

Must SAT Max # satisfiable

```
assert(0 ≤ i < 3) should hold
When the starting index is 1, i is out of bounds
```
Localizing Bugs

(index_1 = 1 ∧ guard_1 = (index_1 ≠ 1) ∧ index_2 = 2 ∧ index_3 = (index_1 + 2) ∧ (guard_1 → i=index_2) ∧ (¬guard_1 → i=index_3) ∧ (0 ≤ i < 3))

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.
Further notes

- We will explore this further within Symbolic Execution
Further notes

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

- We will explore this further within Symbolic Execution
- Recognizing invariants & near 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)