Axiomatic Semantics

Programs → Theorems. Axiomatic Semantics

- Consists of:
  - A language for making assertions about programs
  - Rules for establishing when assertions hold

- Typical assertions:
  - During the execution, only non-null pointers are dereferenced
  - This program terminates with $x = 0$

- Partial vs. total correctness assertions
  - Safety vs. liveness properties
  - Usually focus on safety (partial correctness)

Partial Correctness Assertions

- The assertions we make about programs are of the form:
  $\{A\} c \{B\}$

  with the meaning that:
  - Whenever we start the execution of $c$ in a state that satisfies $A$, the program either does not terminate or it terminates in a state that satisfies $B$

- $A$ is called **precondition** and $B$ is called **postcondition**

  - For example:
    $\{y \leq x\} z := x; z := z + 1 \{y < z\}$

    is a valid assertion

  - These are called Hoare triples or Hoare assertions

Total Correctness Assertions

- $\{A\} c \{B\}$ is a partial correctness assertion. It does not imply termination

- $[A] c [B]$ is a total correctness assertion meaning that whenever we start the execution of $c$ in a state that satisfies $A$, the program **does terminate** in a state that satisfies $B$

  - Now let’s be more formal
    - Formalize the language of assertions, $A$ and $B$
    - Say when an assertion holds in a state
    - Give rules for deriving Hoare triples

Languages for Assertions

- A specification language
  - Must be easy to use and expressive (conflicting needs)
  - Most often only expression $\mathcal{E}$
  - Syntax: how to construct assertions
  - Semantics: what assertions mean

  - Typical examples
    - First-order logic
    - Temporal logic (used in protocol specification, hardware specification)
    - Special-purpose languages: $Z$, Larch, Java ML

State-Based Assertions

- Assertions that characterize the state of the execution
  - Recall: $\text{state} = \text{state of locals} + \text{state of memory}$

- Our assertions will need to be able to refer to
  - Variables
  - Contents of memory

  - What are not state-based assertions
    - Variable $x$ is live, lock $L$ will be released
    - There is no correlation between the values of $x$ and $y$
An Assertion Language

- We’ll use a fragment of first-order logic first
  - Formulas $A ::= O \mid T \mid \bot \mid P_1 \land P_2 \mid \forall x.P_1 \rightarrow P_2$
  - Atoms $O ::= f(O_1 \ldots, O_n) \mid E_1 = E_2 \mid \ldots$
  - Expressions $E ::= n \mid \text{true} \mid \text{false} \mid \ldots$
- We can also have an arbitrary assortment of function symbols
  - $\text{ptr}(E,t)$ - expression $E$ denotes a pointer to a $t$
  - $\text{reachable}(E_1, E_2)$ - list cell $E_2$ is reachable from $E_1$
  - these can be built-in or defined

Semantics of Assertions

- We introduced a language of assertions, we need to assign meanings to assertions.
  - We ignore for now references to memory
- Notation $\rho, \sigma \models A$ to say that an assertion holds in a given state.
  - This is well-defined when $p$ is defined on all variables occurring in $A$ and $\sigma$ is defined on all memory addresses referenced in $A$
- The $\models$ judgment is defined inductively on the structure of assertions.

Semantics of Assertions

- Formal definition (we drop $\sigma$ for simplicity):

  $$\rho \models e_1 = e_2 \iff \rho \models e_1 \parallel n_1 \land \rho \models e_2 \parallel n_2 \land n_1 = n_2$$
  $$\rho \models e_1 \geq e_2 \iff \rho \models e_1 \parallel n_1 \land \rho \models e_2 \parallel n_2 \land n_1 \geq n_2$$
  $$\rho \models A_1 \land A_2 \iff \rho \models A_1 \land \rho \models A_2$$
  $$\rho \models A_1 \lor A_2 \iff \rho \models A_1 \lor \rho \models A_2$$
  $$\rho \models A_1 \Rightarrow A_2 \iff \rho \models A_1 \Rightarrow \rho \models A_2$$
  $$\rho \models \forall x.A \iff \rho \models \exists \text{Z} \exists \text{Z}[x=n] \Rightarrow A$$

Semantics of Assertions

- Now we can define formally the meaning of a partial correctness assertion

  $$\models \{ A \} c \{ B \}$$

  $$\forall \rho, \sigma. (p, \rho \models A \land p, \sigma \models \lnot \parallel p', \sigma') \Rightarrow \rho', \sigma' \models B$$

- … and the meaning of a total correctness assertion

  $$\models \{ A \} c \{ B \}$$

  $$\forall \rho, \sigma. (p, \rho \models A \land p, \sigma \models \lnot \parallel p', \sigma') \Rightarrow \rho', \sigma' \models B$$

  $$\forall \rho, \sigma \models A \Rightarrow \exists \text{Z} \exists \text{Z}[x=n] \Rightarrow A$$

Why Isn’t This Enough?

- Now we have the formal mechanism to decide when
  $\models \{ A \} c \{ B \}$
  - Start the program in all states that satisfies $A$
  - Run the program
  - Check that each final state satisfies $B$
  - This is exhaustive testing
  - Not enough
    - Can’t try the program in all states satisfying the precondition
    - Can’t find all final states for non-deterministic programs
    - And also it is impossible to effectively verify the truth of a
      $\forall x.A$ postcondition (by using the definition of validity)

Derivations as Proxies for Validity

- We define a symbolic technique for deriving valid assertions from others that are known to be valid
  - We start with validity of first-order formulas
  - We write $\models A$ when we can derive (prove) assertion $A$
    - We wish that $\forall \rho, \sigma \models A$ if $\models (A)$
  - We write $\models \{ A \} c \{ B \}$ when we can derive (prove) the partial correctness assertion
    - We wish that $\models \{ A \} c \{ B \}$ if $\models \{ A \} c \{ B \}$
Derivation Rules for Assertions

- The derivation rules for $\vdash A$ are the usual ones from first-order logic with:
  - Natural deduction style axioms:
    
    \[
    \begin{align*}
    &\vdash A \\ &\vdash B \\ &\vdash A \land B \\ &\vdash [a/x]A \quad (a \text{ is fresh}) \\ &\vdash [E/x]A \\ &\vdash \forall x. A \\
    \end{align*}
    \]

Derivation Rules for Hoare Triples

- Similarly we write $\vdash \{A\} c \{B\}$ when we can derive the triple using derivation rules.
- There is one derivation rule for each command in the language.
- Plus, the rule of consequence:

\[
\begin{align*}
&\vdash A \Rightarrow A \quad (A \Rightarrow B \Rightarrow B) \\
&\vdash (A \Rightarrow B) \\
\end{align*}
\]

Derivation Rules for Hoare Logic

- One rule for each syntactic construct:

\[
\begin{align*}
&\vdash (A) \text{ skip } (A) \\
&\vdash (A) \text{ c } (B) \quad (B) \text{ c } (C) \\
&\vdash (A) \text{ c } (B) \quad (A \Rightarrow b) \text{ c } (B) \\
&\vdash (A) \text{ if } b \text{ then } c \text{ else } c \text{ (B)} \\
\end{align*}
\]

Hoare Rules: Assignment

- Example: $\{A\} \ x := x + 2 \ (x \geq 5)$. What is A?
- A has to imply $x \geq 3$
- General rule:

\[
\vdash ((e/x)A) \ x := e \ (A)
\]

Example: Assignment

- Assume that $x$ does not appear in $e$
  - Prove that $\text{true } x := e \ (x = e)$
  - We have

\[
\vdash (e = e) \quad (e \Rightarrow e \Rightarrow e)
\]

because $[e/x](x = e) = e = [e/x]e = e = e$

- Assignment + consequence:

\[
\vdash \text{true } \Rightarrow \ (e = e) \quad (e \Rightarrow e) \Rightarrow e \Rightarrow e
\]
The Assignment Axiom (Cont.)

- Hoare said: "Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics. It is surprising therefore that the axiom governing our reasoning about assignment is quite as simple as any to be found in elementary logic."

- Caveats are sometimes needed for languages with aliasing:
  - If $x$ and $y$ are aliased then
    $$\{\text{true}\} x := 5 \{x + y = 10\}$$
    is true.

Example: Conditional

\[
D_1 \vdash (\text{true} \land y \leq 0) \quad x := 1 \{x > 0\}
\]
\[
D_2 \vdash (\text{true} \land y > 0) \quad x := y \{x > 0\}
\]
\[
\vdash (\text{true}) \text{ if } y \leq 0 \text{ then } x := 1 \text{ else } x := y \{x > 0\}
\]

- $D_1$ is obtained by consequence and assignment
  $$\vdash (x > 0) \quad x := 1 \{x > 0\}$$
  $$\vdash (\text{true} \land y \leq 0) \quad x := 1 \{x > 0\}$$

- $D_2$ is also obtained by consequence and assignment
  $$\vdash (y > 0) \quad x := y \{x > 0\}$$
  $$\vdash (\text{true} \land y > 0) \quad x := y \{x > 0\}$$

Multiple Hoare Rules

- For some constructs multiple rules are possible:
  $$\frac{\vdash (A \land b \Leftarrow c \land B) \quad \vdash (A \land \neg b \Leftarrow B)}{\vdash (A \land \{b \Leftarrow c \land B\} \Leftarrow \{A \land \neg b \Leftarrow B\})}$$
  (This was the "forward" axiom for assignment before Hoare)

Example: Loop

- We want to derive that
  $$\vdash (x \leq 0) \text{ while } x \leq 5 \text{ do } x := x + 1 \{x = 6\}$$
- Use the rule for while with invariant $x \leq 6$
  $$\vdash (x \leq 6 \land x \leq 5) \quad x := x + 1 \{x \leq 6\}$$
  $$\vdash (x \leq 6) \text{ while } x \leq 5 \text{ do } x := x + 1 \{x \leq 6 \land x < 5\}$$
- Then finish-off with consequence
  $$\vdash (x \leq 0) \Rightarrow x \leq 6$$
  $$\vdash (x \leq 6 \land x \leq 5) \Rightarrow x = 6$$
  $$\vdash (x \leq 6) \Rightarrow (x \leq 6 \land x < 5)$$
  $$\vdash (x \leq 0) \Rightarrow (x = 6)$$

Another Example

- Verify that
  $$\vdash (A) \text{ while } \text{true} \text{ do } c \{B\}$$
  holds for any $A$, $B$ and $c$
- We must construct a derivation tree
  $$\vdash A \Leftarrow (\text{true} \land \text{true}) \quad c \{\text{true}\}$$
  $$\vdash \text{true} \land \text{false} \Leftarrow (\text{true}) \text{ while } \text{true} \text{ do } c \{\text{true} \land \text{false}\}$$
  $$\vdash (A) \text{ while } \text{true} \text{ do } c \{B\}$$

- We need an additional lemma:
  $$\forall c. (\text{true}) \quad c \{\text{true}\}$$
  - How do you prove this one?

GCD Example

- Let $c$ be the program:
  $$\text{while } (x \neq y) \text{ do }$$
  $$\text{if } (x \leq y) \text{ then } y := y - x$$
  $$\text{else } x := x - y$$
- We’ll derive that
  $$\vdash \{x = m \land y = n\} \quad c \{x = \text{gcd}(m, n)\}$$
GCD Example (2)

- Crucial to select good loop invariant
  - Let the precondition \( \text{Pre} \) be
    \[ x \equiv m \land y \equiv n \]
  - Let the postcondition \( \text{Post} \) be
    \[ x = \gcd(m, n) \]
  - We use the loop invariant
    \[ I \overset{df}{=} \gcd(x, y) = \gcd(m, n) \]

GCD Example (3)

We first use the rule of consequence to obtain the subgoal
\[
\vdash (I \land \neg(x \neq y)) \quad (1)
\]
But we also need to prove
\[
\vdash \text{Pre} \Rightarrow I \quad (2)
\]
\[
\vdash I \land \neg(x \neq y) \Rightarrow \text{Post} \quad (3)
\]
Subgoal 2 reduces to
\[
x = m \land y = n \Rightarrow \gcd(x, y) = \gcd(m, n)
\]
Subgoal 3 reduces to
\[
\gcd(x, y) = \gcd(m, n) \land x = y \Rightarrow x = \gcd(m, n)
\]

GCD Example (4)

Now we still have to derive subgoal 1:
\[
\vdash (I \land \neg(x \neq y)) \quad (4)
\]
We can apply the rule for \( \forall x \) and we get the subgoal
\[
\vdash (I \land x \neq y) \quad (5)
\]
where \( d \) is the body of the loop:
\[
\begin{align*}
\text{if } (x \leq y) & \\
\text{then } y & := y - x \\
\text{else } x & := x - y
\end{align*}
\]

GCD Example (5)

We can derive subgoal 4 using the rule for conditionals and we get two subgoals
\[
\vdash (I \land x \neq y \land x \leq y) \quad (5)
\]
\[
\vdash (I \land x \neq y \land x > y) \quad (6)
\]
Each of the subgoals 5 and 6 can be derived using the rule of consequence followed by assignment:
\[
\vdash I \land x \neq y \land x \leq y \Rightarrow \gcd(m, n) = \gcd(x, y - x) \quad (7)
\]
\[
\vdash I \land x \neq y \land x > y \Rightarrow \gcd(m, n) = \gcd(x - y, y) \quad (8)
\]

GCD Example (6)

\[
\vdash I \land x \neq y \land x > y \Rightarrow \gcd(m, n) = \gcd(x - y, y)
\]
- The above can be proved by realizing that
  \[ \gcd(x, y) = \gcd(x - y, y) \]
- Q.e.d.
- This completes the proof
- We used a lot of arithmetic
- We had to invent the loop invariants
- What about the proof for total correctness?

Hoare Rule for Function Call

- If no recursion we can inline the function call
  \[
  f(x_1, \ldots, x_n) = C f \text{ Program } (A) \in B[f/x] 
  \]
  \[
  \vdash (A[e_1/x_1, \ldots, e_n/x_n]) \times = f(e_1, \ldots, e_n)(8)
  \]
- In general,
  1. each function \( f \) has a \( \text{Pre}_f \) and \( \text{Post}_f \)
  \[
  \vdash (\text{Pre}_f[e_1/x_1, \ldots, e_n/x_n]) \times = f(e_1, \ldots, e_n)(\text{Post}_f[x/f])
  \]
  2. For each function we check \( (\text{Pre}_f) C_f (\text{Post}_f) \)
**Axiomatic Semantics in Presence of Side-Effects**

**Naïve Handling of Program State**
- We allow memory read in assertions: \( *x + *y = 5 \)
- We try:
  \[
  \{ A \} \quad *x = 5 \quad \{ *x + *y = 10 \}
  \]
- \( A \) ought to be \( *y = 5 \) or \( x = y \)
- The Hoare rule would give us:
  \[
  (*x + *y = 10)[5/*x] = 5 + *y = 10
  \]
  \[
  = *y = 5 \quad \text{(we lost one case)}
  \]
- How come the rule does not work?

**Handling Program State**
- We cannot have side-effects in assertions
  - While creating the theorem we must remove side-effects!
  - But how to do that when lacking precise aliasing information?
- Important technique: Postpone alias analysis
- Model the state of memory as a symbolic mapping from addresses to values:
  - If \( E \) denotes an address and \( M \) a memory state then:
  - \( \text{sel}(M, E) \) denotes the contents of memory cell
  - \( \text{upd}(M, E, V) \) denotes a new memory state obtained from \( M \) by writing \( V \) at address \( E \)

**More on Memory**
- We allow variables to range over memory states
  - So we can quantify over all possible memory states
- And we use the special pseudo-variable \( \mu \) in assertions to refer to the current state of memory
- Example:
  \[
  \forall i \cdot i \geq 0 \land i < 5 \Rightarrow \text{sel}(\mu, A + i) > 0
  \]
  \[= \text{allpositive}(\mu, A, 0, 5)\]
  says that entries 0, 4 in array \( A \) are positive

**Semantics of Memory Expressions**
- We need a new kind of values (memory values)
  \[
  \text{Values } v ::= n \mid a \mid \alpha
  \]
  \[
  \begin{array}{c}
  \rho, \alpha = \mu \Downarrow \alpha \\
  \rho, \alpha = E_1 \Downarrow \alpha' \\
  \rho, \alpha = E_2 \Downarrow a \\
  \rho, \alpha = \text{sel}(E_1, E_2) \Downarrow \psi(a)
  \end{array}
  \]
  \[
  \begin{array}{c}
  \rho, \alpha = E_1 \Downarrow a' \\
  \rho, \alpha = E_2 \Downarrow a \\
  \rho, \alpha = E_1 \Downarrow v
  \end{array}
  \]
  \[
  \begin{array}{c}
  \rho, \alpha = \text{upd}(E_1, E_2, E_3) \Downarrow \psi(a \Downarrow v)
  \end{array}
  \]

**Hoare Rules: Side-Effects**
- To correctly model writes we use memory expressions
  - A memory write changes the value of memory
    \[
    \{ (B(\text{upd}(\alpha, E_3, E_2)) \ast E_1 := E_2) \}
    \]
  - Important technique: treat memory as a whole
  - And reason later about memory expressions with inference rules such as (McCarthy):
    \[
    \begin{cases}
    \text{sel}(\text{upd}(M, E_2), E_3) = E_2 & \text{if } E_2 = E_3 \\
    \text{sel}(M, E_3) & \text{if } E_3 \neq E_3
    \end{cases}
    \]
Memory Aliasing

- Consider again: \( \{ A \} \times x = 5 \ ( \times x + \times y = 10 ) \)
- We obtain:
  \[
  A = \left( \times x + \times y = 10 \right) \uparrow \mu \times x, 5 \mu \] 
  \[
  = \left( \times \uparrow \mu, x, 5 \right) \times x + \left( \times \uparrow \mu, x, 5 \right) \times y = 10 \quad (*)
  = 5 + \left( \times \uparrow \mu, x, 5 \right) \times y = 10
  = x = y \text{ or } \times y = 5 \quad (**)
  
  To (*) is theorem generation
  From (*) to (**) is theorem proving

Alternative Handling for Memory

- Reasoning about aliasing is expensive (NP-hard)
- Sometimes completeness is sacrificed with the following (approximate) rule:

  \[
  \begin{align*}
  \text{If } E_1 = \text{obviously } E_3 & \quad \text{then } E_2 = \text{obviously } E_3 \\
  \text{sel}(M, E_1), E_2) = & \quad \text{if } p = \text{fresh new parameter}
  \end{align*}
  
  The meaning of "obvious" varies:
  - The addresses of two distinct globals are ≠
  - The address of a global and one of a local are ≠
  - PREFix and GCC use such schemes

Using Hoare Rules. Notes

- Hoare rules are mostly syntax directed
- There are three wrinkles:
  - When to apply the rule of consequence?
  - What invariant to use for while?
  - How do you prove the implications involved in consequence?
- The last one is how theorem proving gets in the picture
  - This turns out to be doable!
  - The loop invariants turn out to be the hardest problem!
    (Should the programmer give them? See Dijkstra.)

Where Do We Stand?

- We have a language for asserting properties of programs
- We know when such an assertion is true
- We also have a symbolic method for deriving assertions

Soundness of Axiomatic Semantics

- Formal statement
  \[ \text{If } \vdash \{ A \} \subset \{ B \} \text{ then } \vdash \{ A \} \subset \{ B \} \]
  or, equivalently
  \[ \text{For all } \rho, \sigma, \text{ if } \rho, \sigma = A \text{ and } D \vdash \rho, \sigma \downarrow \rho', \sigma' \text{ and } H \vdash \{ A \} \subset \{ B \} \text{ then } \rho', \sigma' = B \]

Completeness of Axiomatic Semantics

Weakest Preconditions
Completeness of Axiomatic Semantics

- Is it true that whenever \( \models \{ A \} \subseteq \{ B \} \) we can also derive \( \vdash \{ A \} \subseteq \{ B \} \)?
- If it isn’t then it means that there are valid properties of programs that we cannot verify with Hoare rules
- Good news: for our language the Hoare triples are complete
- Bad news: only if the underlying logic is complete (whenever \( \models A \) we also have \( \vdash A \)) - this is called relative completeness

Proof Idea

- Dijkstra’s idea: To verify that \( \{ A \} \subseteq \{ B \} \)
  a) Find out all predicates \( A' \) such that \( \models \{ A' \} \subseteq \{ B \} \)
  - call this set \( \text{Pre}(c, B) \)
  b) Verify for one \( A' \in \text{Pre}(c, B) \) that \( A \Rightarrow A' \)
- Assertions can be ordered:
  - false \( \Rightarrow \) true
  - strong \( \Rightarrow \) weakest
  - precondition: \( \text{WP}(c, B) \)
- Thus: compute \( \text{WP}(c, B) \) and prove \( A \Rightarrow \text{WP}(c, B) \)

Proof Idea (Cont.)

- Completeness of axiomatic semantics:
  - If \( \models \{ A \} \subseteq \{ B \} \) then \( \vdash \{ A \} \subseteq \{ B \} \)
  - Assuming that we can compute \( \text{wp}(c, B) \) with the following properties:
    1. \( \text{wp} \) is a precondition (according to the Hoare rules)
    \( \vdash \{ \text{wp}(c, B) \} \subseteq \{ B \} \)
    2. \( \text{wp} \) is the weakest precondition
    \( \models \{ A \} \subseteq \{ B \} \) then \( A \Rightarrow \text{wp}(c, B) \)
  - We also need that whenever \( A \) then \( A \)

Weakest Preconditions

- Define \( \text{wp}(c, B) \) inductively on \( c \), following Hoare rules:
  \[
  \begin{align*}
  \text{wp}(c_1 \cdot c_2, B) &= \text{wp}(c_1, \text{wp}(c_2, B)) \\
  \text{wp}(x := E, B) &= [E/x]B \\
  \text{wp}(\text{if } E \text{ then } c_1 \text{ else } c_2, B) &= \text{wp}(c_1, B) \land \neg E = \text{wp}(c_2, B)
  \end{align*}
  \]
- Weakest Preconditions for Loops
  - We start from the equivalence
    \( \text{while } b \text{ do } c \equiv \text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip} \)
  - Let \( w \) while \( b \text{ do } c \) and \( W = \text{wp}(w, B) \)
  - We have that
    \( W = b \Rightarrow \text{wp}(w, W) \land \neg b = B \)
  - But this is a recursive equation!
    - We know how to solve these using domain theory
  - We need a domain for assertions

A Partial-Order for Assertions

- What is the assertion that contains least information?
  - true - does not say anything about the state
- What is an appropriate information ordering?
  - \( A \leq A' \) iff \( A' \Rightarrow A \)
- Is this partial order complete?
  - Take a chain \( A_1 \leq A_2 \leq ... \)
  - Let \( \land A_i \) be the infinite conjunction of \( A_i \)
    \( \alpha = \land A_i \) iff for all \( i \) we have that \( \alpha = A_i \)
  - Verify that \( \land A_i \) is the least upper bound
- Can \( \land A_i \) be expressed in our language of assertions?
  - In many cases yes (see Winskel), we’ll assume yes
Weakest Precondition for WHILE

• Use the fixed-point theorem
  \[ F(A) = b \Rightarrow wp(c, A) \land \neg b \Rightarrow B \]
  - Verify that \( F \) is both monotonic and continuous

• The least-fixed point (i.e. the weakest fixed point) is
  \[ wp(w, B) = \bigwedge \{ F^i(\text{true}) | i \geq 0 \} \]

Weakest Precondition (Cont.)

• Define a family of wp's
  \[ \text{wp}(\text{while } e \text{ do } c, B) = \text{weakest precondition on which the loop if it terminates in } k \text{ or fewer iterations, it terminates in } B \]
  \[ \text{wp}_0 = \neg E \Rightarrow x \geq 7 \]
  \[ \text{wp}_1 = x \leq 5 \Rightarrow x + 1 \not\in \{5,6\} \land \text{wp}_0 = x \not\in \{4,5,6\} \]
  \[ \text{...} \]
  \[ \text{wp} = x \geq 7 \]

Weakest Precondition. Example 1

• Consider the code:
  \[ \text{while } x \leq 5 \text{ do } x := x + 1 \]
  with postcondition \( P = x \geq 7 \)
  • What is the weakest precondition?
  \[ \text{wp}(x := x + 1, A) = A[x+1/x] \]
  \[ \text{WP}_0 = \neg(x \leq 5) \Rightarrow x \geq 7 \Rightarrow x \not\in \{6\} \]
  \[ \text{WP}_1 = x \leq 5 \Rightarrow x + 1 \not\in \{5,6\} \land \text{WP}_0 = x \not\in \{4,5,6\} \]
  \[ \text{...} \]
  \[ \text{WP} = x \geq 7 \]

Weakest Precondition. Example 2

• Consider the code:
  \[ \text{while } x \geq 5 \text{ do } x := x + 1 \]
  with postcondition \( P = x \geq 7 \)
  • What is the weakest precondition?
  \[ \text{wp}(x := x + 1, A) = A[x+1/x] \]
  \[ \text{WP}_0 = \neg(x \geq 5) \Rightarrow x \geq 7 \Rightarrow x \geq 5 \Rightarrow x \not\in \{5,6\} \land \text{WP}_0 = x \not\in \{4,5,6\} \]
  \[ \text{...} \]
  \[ \text{WP} = x \geq 5 \]

Theorem Proving and Program Analysis (again)

• Predicates form a lattice:
  \[ WP(s, B) = \bigwedge \{ \text{Pre}(s, B) \} \]
• This is not obvious at all:
  - \[ \bigwedge \{ P_1, P_2 \} = P_1 \lor P_2 \]
  - \[ \bigwedge \{ P \} = \text{true} \]
  - But can we always write this with a finite number of \( \lor \)?
• Even checking implication can be quite hard
• Compare with program analysis in which lattices are of finite height and quite simple
  \[ \text{Program Verification is Program Analysis on the lattice of first order formulas} \]