Reasoning about code

- Determine what facts are true during execution — we've seen these as assertions, representation invariants, preconditions, postconditions, etc.
  - \( x > 0 \)
  - for all nodes \( n \): \( n\text{'s}.next.previous == n \)
  - array \( a \) is sorted
  - \( x + y == z \)
  - if \( x \neq \) null then \( x.a > x.b \)
- These can help...
  - ... increase confidence that code is correct
  - ... understand why code is incorrect

Forward reasoning

- Given a precondition, what is the postcondition?
- Example
  // precondition: \( x \) is even
  \( x = x + 3; \)
  \( y = 2x; \)
  \( x = 5; \)
  // postcondition: ??
- One use: rep invariant holds before running code, does it still hold after running code?

Backward reasoning

- Given a postcondition, what is the corresponding precondition?
- Example
  // precondition: ??
  \( x = x + 3; \)
  \( y = 2x; \)
  \( x = 5; \)
  // postcondition: \( y > x \)
- Uses include: what is needed to re-establish rep invariant, to reproduce a bug, to exploit a bug?

Ex: SQL injection attack

- SQL query constructed using unfiltered user input
  \[ \text{query} = \text{"SELECT * FROM users "} + \text{"WHERE name='"} + \text{userInput + "}=1'; \]
  - if the user inputs ' or '1'='1 this results in
  \[ \text{query} \to \text{SELECT * FROM users WHERE name='a' or '1'=1';} \]
  - This query returns information about all users — bad!

http://xkcd.com/327/
Forward vs. backward reasoning

- Forward reasoning is more intuitive for most people
  - Helps you understand what will happen (simulates the code)
  - Introduces facts that may be irrelevant to your task
- Backward reasoning is usually more helpful
  - Helps you understand what should happen
  - Given a specific task, indicates how to achieve it – for example, it can help creating a test case that exposes a specific error

Reasoning about code statements

- Convert assertions about programs into logic
- One logic representation is a Hoare triple:
  \[ P \{ \text{Java code} \} Q \]
  - \( P \) and \( Q \) are logical assertions about program values
  - The triple means “if \( P \) is true and you execute code, then \( Q \) is true afterward”
  - A Hoare triple is a boolean – true or false

Tiny examples

- \[ \text{true} \{ y = x \times x \} y \geq 0 \]
- \[ x \neq 0 \{ y = x \times x \} y > 0 \]
- \[ x > 0 \{ y = x + 1 \} y > 1 \]

Partial examples

- \[ x = k \{ \text{if } x < 0 \ x = -x \} \]
  - Replace \( ? \) with \( \text{what to get true} \)
- \[ x = 3 \{ x = 8 \} \]
  - Replace \( ? \) with \( \text{what to get true} \)

Longer example

- \[ x \geq 0 \{ \]
  - \[ z = 0; \]
  - \[ \text{if } (x \neq 0) \ z = x; \text{ else } z = z + 1; \}
  - \[ z > 0 \]
\[ \]
- \[ \text{assert } x >= 0; \]
  - \[ \text{assert } x > 0; \]
  - \[ \text{Reasoning: what we know after each program point} \]
  - \[ \Rightarrow z > 0 \]
    - \[ \text{QED} \]

Strongest or weakest conditions?

- \[ x = 5 \]
  - \[ (x = x + 2) \]
  - \[ x > 0 \]
  - \[ x = 5 \]
  - \[ (x = x + 2) \]
  - \[ x > 0 \]
  - \[ x = 5 \]
  - \[ (x = x + 2) \]
  - \[ x = 10 \]
  - \[ \text{All are true Hoare triples – which precondition is most valuable, and why?} \]
- \[ x = 5 \land y = 10 \]
  - \[ (z = x/y) \]
  - \[ z < 1 \]
  - \[ x < y \land y > 0 \]
  - \[ (z = x/y) \]
  - \[ z < 1 \]
  - \[ y \neq 0 \land x \land y < 1 \]
  - \[ (z = x/y) \]
  - \[ z < 1 \]
  - \[ \text{All are true Hoare triples – which postcondition is most valuable, and why?} \]
Weakest precondition

- $y \neq 0 \land x / y < 1$
  
  $z < 1$
  
  (the last one) is the most useful because it allows us to invoke the program in the most general condition

- It is called the weakest precondition, $\wp(S, Q)$ of $S$ with respect to $Q$

- If $P \{ S \} Q$ and for all $P'$ such that $P' \Rightarrow P$, then $P$ is $\wp(S, Q)$

A rule for each language construct

- The above examples use intuition to discuss the Hoare triples

- Specifically to understand how the code affects the precondition to determine the (strongest) postcondition, using forward reasoning

- postcondition to determine the (weakest) precondition, using backward reasoning

- To replace the intuition with a mechanical transformation — needed for precision and for automation — each language construct must be explicitly defined using the logic

Sequential execution or: What does ; really mean?

- $P \{ S_1; S_2 \} Q$

  - Compute the intermediate assertion $A = \wp(S_2, Q)$

  - This means that $P \{ S_1 \} (A \{ S_2 \} Q)$

  - Compute the assertion $T = \wp(S_1, A)$

  - This means that $T \{ S_1 \} (A \{ S_2 \} Q)$

  - If $P \Rightarrow T$ the triple is true

  - We reason backwards to compose the statements

Conditional execution

- $P \{ if \ C \ S_1 \ else \ S_2 \} Q$

  - Must consider both branches — consider

    - true
      
      $\{$
      
      - if $x \geq y$
        
        $z = x$
      
      - else
        
        $z = y$
      
      $\}$

    - $z = x \lor z = y$

  - But something is missing — knowledge about the value of the condition

Example

- $P \{ if \ C \ S_1 \ else \ S_2 \} Q$

  - The precise definition of a conditional (if-then-else) statement takes into account the condition’s value and both branches

    - $(P \land C \{ S_1 \} Q) \land (P \land \neg C \{ S_2 \} Q)$

  - Even though at execution only one branch is taken, the proof needs to show that both will satisfy $Q$

  - Or $\wp(if \ C \ S_1 \ else \ S_2 ; Q)$ is equal to

    - $C \Rightarrow \wp(S_1, Q) \land \neg C \Rightarrow \wp(S_2, Q)$

  - The precise definition of a conditional (if-then-else) statement takes into account the condition’s value and both branches

    - $(P \land C \{ S_1 \} Q) \land (P \land \neg C \{ S_2 \} Q)$

    - Even though at execution only one branch is taken, the proof needs to show that both will satisfy $Q$

    - Or $\wp(if \ C \ S_1 \ else \ S_2 ; Q)$ is equal to

      - $C \Rightarrow \wp(S_1, Q) \land \neg C \Rightarrow \wp(S_2, Q)$
Assignment statements

- What does the statement \( x = E \) really mean?
- \( Q(E) \{ x = E \} \ Q(x) \)
- That is, if we knew something to be true about \( E \) before the assignment, then we know it to be true about \( x \) after the assignment
- Assuming no side-effects
- \( \text{wp}(x=E;\ Q) \) is \( Q \) with \( x \) replaced by \( E \)

Examples

\[
Q(E) \{ x = E \} \ Q(x)
\]

\[
\begin{align*}
&y > 0 \\
&\{ x = y \} \\
x > 0
\end{align*}
\]

\[
\begin{align*}
&x > 0 \\
&\{ x = x + 1 \} \\
x > 1
\end{align*}
\]

More examples

\[
? \{ x = y + 5 \} \ x > 0
\]

\[
\begin{align*}
x &= A \land y = B \\
&\{ t = x; \\
&x = y; \\
y = t; \\
\} \\
x &= B \land y = A
\end{align*}
\]

Method calls

\[
? \{ x = \text{foo()} \} \ Q
\]

- If the method has no side effects, it's just like ordinary assignment
  \( (y = 22 \lor y = -22) \)
  \( \{ x = \text{Math.abs}(y) \} \)
  \( x = 22 \)

With side effects

- If it has side effects it also needs an assignment to every variable in \textit{modifies}
- Use the method specification to determine the new value

\[
z+1 = 22 \{ \text{incrZ()} \} \quad // \text{spec: } z_{\text{post}} = z_{\text{pre}} + 1
\]

Loops: \( P \{ \text{while } B \text{ do } S \} \ Q \)

- A loop represents an unknown number of paths (and recursion presents the same problem as loops)
- Cannot enumerate all paths – this is what makes testing and reasoning hard
- Trying to unroll the loop doesn’t work, since we don’t know how many times the loop can execute

\[
\begin{align*}
&P \land \neg B \ (S) \ Q \\
&P \land B \ (S) \ Q \land \neg B) \land \\
&P \land B \ (S) \ Q \land B) \ {S} \ Q \land \neg B \land \neg \ 
\end{align*}
\]
The most common approach to this is to find a loop invariant, a predicate that is true each time the loop head is reached (on entry and after each iteration) and helps us prove the postcondition of the loop. Essentially, we will prove the properties inductively.

Find a loop invariant \( I \) such that

1. \( P \implies I \) //Invariant is correct on entry
2. \( B \land I \implies I \) //Invariant is maintained
3. \( \neg B \land I \implies Q \) //Loop termination proves \( Q \)

An invariant that works: \( LI = x \geq y \)

1. \( x \geq 0 \land y = 0 \implies LI \)
2. \( LI \land x \neq y \{ y = y + 1 \} LI \)
3. \( (LI \land \neg(x \neq y)) \implies x = y \)

Proofs with loop invariants do not guarantee that the loop terminates, only that it does produce the proper postcondition if it terminates — this is called weak correctness. A Hoare triple for which termination has been proven is strongly correct.

Proofs of termination are usually performed separately from proofs of correctness, and they are usually performed through well-founded sets.

In the max example it’s easy, since \( i \) is bounded by \( n \), and \( i \) increases at each iteration.

Most of your loops need no proofs:

- for (String name : friends) { ... }

Write loop invariants and decrementing functions when you are unsure about a loop.

- If a loop is not working
  - Add invariant
  - Write code to check them
  - Understand why the code doesn’t work
  - Reason to ensure that no similar bugs remain
Next steps

- Wednesday: reasoning II; Friday: usability;
  Monday: UML; Wednesday: TBA
- A5 and A6