Reasoning About Code

New Word!

Loop

Say it with me: Loop

The action of doing something over and over again
Outline

• Intro to reasoning about code
  • Specifications
  • Preconditions and postconditions
  • Forward reasoning and backward reasoning

• Reasoning about code, formally; Hoare logic
  • Hoare Triples
  • Rules for assignment, sequence, if-then-else
Reasoning About Code

• Determines before execution what facts hold during program execution

• Reason about conditions:
  
  0 \leq index < \text{names}.length
  
  x > 0

  array \text{names} is sorted
  
  x > y

These are all condition which could be true or false
Why Reason About Code

- Our goal is to produce correct code!
- Two ways to ensure correctness
  - Testing
  - Reasoning about code
    - verification
- Reasoning about code
  - Verifies that code works correctly
  - Finds errors in code
  - Helps understand errors
    - E.g., what input caused division by zero?
Specifications

• What does it mean for code to be **correct**?
  • (Informally) Code is correct if it conforms to its **specification**

• A specification consists of a **precondition** and a **postcondition**
  • Precondition: conditions that must hold **before** code executes
  • Postcondition: conditions that must hold **after** code finishes execution (if precondition held!)

• **Precondition and Postconditions**
  • Logical constraint on values
Specifications

Precondition: \( \text{arr.length} = \text{len} \land \text{len} \geq 0 \)

Postcondition: \( \text{result} = \text{arr}[0] + \ldots + \text{arr}[\text{arr.length}-1] \)

```c
int sum(int[] arr, int len) {
    int result = 0;
    int i = 0;
    while (i < len) {
        result = result + arr[i];
        i = i+1;
    }
    return result;
}
```

To prove that \texttt{sum} is correct, we must prove that the implementation meets the specification. In other words, we must prove that if the precondition held, after code finishes execution, the postcondition holds. To do this, we must reason about code.

Notation:
- \( \land \) denotes logical AND
- \( \lor \) denotes logical OR
Specifications

• The specification is a **contract** between the function and its caller. Both caller and function have obligations:
  • Caller must pass arguments that obey the **precondition**.
  • If not, all bets are off --- function can break or return wrong result!
  • Function “promises” the **postcondition**

• In **sum**, how can the caller violate spec?
• How can **sum** violate spec?
Type Signature is a Form of Specification

• Type signature is a contract too!
  - int sum(int[] arr, int len) {...}
    - Precondition: arguments are an array of ints and an int
    - Postcondition: result is a int
  - Java enforces the type constraint

• We need more than type signatures! Why?
  - We need reasoning about behavior and effects (deeper properties)
Type Signature is a Specification

• Type checker (among other things) verifies that the parties meet the contract

• If language is type safe we can “trust” the type checker

• But if language is type unsafe would be possible for a caller to pass an argument of the wrong type!
  • E.g. Python allows you to pass an object that might not have the needed methods or worse have a method of the same name that does something different than expected.
Why Reason About Code

• Ensure code works correctly
  • Ensure that code meets the specification
  • we can prove that \texttt{sum} is correct by proving that \texttt{sum} meets its specification

• Find errors in code

• Understand errors
What is Wrong With this Code?

class NameList {
    int index;
    String[] names;

    // Precondition: 0 ≤ index < names.length
    void addName(String name) {
        index++;
        if (index < names.length)
            names[index] = name;
    }

    // Postcondition: 0 ≤ index < names.length
}

Is there a situation where the precondition holds, but postcondition is violated?
What Inputs Cause Wrong Output?

```java
String[] parseName(String name) {
    int comma = name.indexOf(“,“);
    String firstName = name.substring(0,comma);
    String lastName = name.substring(comma+2);
    return new String[] { lastName, firstName };
}
```

What input produces array ["Doe","Jane"]?
What input produces array ["oe","Jane"]?
What input produces StringIndexOutOfBoundsException?
Types of Reasoning

• **Forward reasoning**: given a precondition, does the postcondition hold?
  • Verify that code works correctly
  • Does the code produce output that matches the postcondition

• **Backward reasoning**: given a postcondition, what is the proper precondition?
  • Again, verify that code works correctly
  • What input caused an error
Forward Reasoning

• We know what is true \textbf{before} running the code. What is true \textbf{after} running the code?

```
// precondition: $x$ is even && $x \geq 0$

x = x + 3;
y = 2x;
x = 5;
```

// What is the postcondition here?
Strongest Postcondition

• Many postconditions hold from this precondition and code!

// precondition: x is even && x > 0

x = x + 3;
y = 2x;
x = 5;

// postcondition: x = 5 && y % 4 = 2
// postcondition: x = 5 && y is even
// postcondition: x > 0 && y is even

x=5 && y%4 = 2 is the strongest postcondition. It implies all other postconditions. More on stronger and weaker conditions later.
Forward Reasoning Example

// precondition: x>y
z = x;
x = y;
y = z;
// What is the postcondition ??
Backward Reasoning

• We know what we want to be true after running the code. What must be true beforehand to ensure that?

// precondition: ??

```
x = x + 3;
y = 2x;
x = 5;
```

// postcondition: y > x
Forward vs. Backward Reasoning

• Forward reasoning may be more intuitive, just simulates the code
  • Introduces facts that may be irrelevant to the goal
  • Takes longer to prove task or realize task is hopeless

• Backward reasoning is usually more helpful
  • Given a specific goal, shows what must hold beforehand in order to achieve this goal
  • Given an error, gives input that exposes error
Forward Reasoning: Putting Statements Together
Does the postcondition hold?

Precondition: $x \geq 0$; Postcondition: $z > 0$

$z = 0$; 

\[
\begin{align*}
\text{if} \ (x \neq 0) \ {&} \ \{ x \geq 0 \ \&\& \ x \neq 0 \ \&\& \ z = 0 \} \Rightarrow \{ x > 0 \ \&\& \ z = 0 \} \\
& \quad \{ x > 0 \ \&\& \ z = x \} \\
\text{else} \ {&} \ \{ x=0 \ \&\& \ x = 0 \ \&\& \ z = 0 \} \Rightarrow \{ x=0 \ \&\& \ z = 0 \} \\
& \quad \{ x = 0 \ \&\& \ z = 1 \}
\end{align*}
\]

Postcondition: $x > 0 \ \&\& \ z = x$) $||$ (x=0 $\&\&$ z=1)

either way $z > 0$;

Therefore, postcondition holds!
Reasoning About Loops

• A loop represents an unknown number of paths
  • Case analysis can be tricky
  • Recursion presents the same problem

• Can't enumerate all paths
  • Testing and reasoning about loops can be tricky
Forward Reasoning With a Loop

Does the postcondition hold?

Precondition: \( x \geq 0 \);

\[ i = x; \quad \{ x \geq 0 \land i = x \} \]

\[ z = 0; \quad \{ x \geq 0 \land i = x \land z = 0 \} \]

while (\( i \neq 0 \)) {

\[ z = z + 1; \quad ??? \]

\[ i = i - 1; \quad ??? \]

}

Postcondition: \( x = z \);

Yes. The key is to \textbf{guess} the loop invariant. Then prove by induction over the number of iterations of the loop.
Loop Invariant

• A loop invariant is any property that is preserved by execution of the loop body
• A condition that is true immediately before and immediately after each iteration of a loop
  • Doesn’t say anything about truth part way through
• We reason about loop invariants using induction
Forward Reasoning With a Loop

Precondition: \( x \geq 0; \)
\[
i = x;
\]
\[
z = 0;
\]
while (i != 0) {
    \[
z = z + 1;
\]
    \[
i = i - 1;
\]
}
Postcondition: \( x = z; \)

Invariant: \( i + z = x \)

Before:
\[
x + 0 = x
\]
Iteration 0:
\[
z_0 = z_0 + 1 = 1
\]
\[
i_0 = i_0 - 1 = x - 1
\]
\[
i_0 + z_0 = x
\]
Iteration n: (assume invariant holds for iteration n-1)
\[
z_n = z_{n-1} + 1
\]
\[
i_n = i_{n-1} - 1
\]
\[
i_n + z_n = i_{n-1} + z_{n-1} = x
\]
After:
\[
i == 0 \& \& i + z = x \rightarrow z = x
\]
Forward Reasoning With a Loop

• A loop invariant must be true before, after the loop exits, and after each iteration of the loop
  • Is it true before loop starts?
    • Base case
  • Assume the invariant is true for iteration n-1
  • Prove it is true for iteration n
  • Is the invariant true after the loop completes?

• A loop invariant must be useful/relevant

```java
while ( cond ) {
  statements
}
  <==> define loop invariant P
```
Forward Reasoning With a Loop

Before the loop begins

P (the invariant) holds here

true

P && cond

Cond?

does not make sense?

false

P && !cond
Outline

• Intro to reasoning about code
  • Specifications
  • Preconditions and postconditions
  • Invariants
  • Forward reasoning and backward reasoning

• Reasoning about code
  • formally: Hoare Logic
  • Hoare Triples
  • Rules for assignment, sequence, if-then-else
Hoare Logic

• Formal framework for reasoning about code
  • *mechanize* the process of reasoning about code

• Sir Anthony Hoare (Sir Tony Hoare or Sir C.A.R. Hoare)
  • Hoare logic
  • Quicksort algorithm
  • Other contributions to programming languages
  • Turing Award in 1980
Hoare Triples

• A Hoare Triple: \{ P \} code \{ Q \}
  • P and Q are logical statements about program values, and code is program code (in our case, Java code)

• “\{ P \} code \{ Q \}” means “if P is true and we execute code, then Q is true afterword”
  • “\{ P \} code \{ Q \}” is a logical formula, just like “0 ≤ index”
Examples of Hoare Triples

{ x>0 } x++ { x>1 } is true
{ x>0 } x++ { x>-1 } is true
{ x≥0 } x++ { x>1 } is false. Why?

{x>0} x++ {x>0} is ?
{x<0} x=x+1 {x<0} is ??
{x=a} if (x < 0) x=-x { x = | a | } is ??
{x=y} x=x+3 {x=y} is ??
Examples of Hoare Triples

• \{ x \geq 0 \} x++ \{ x > 1 \} is a logical formula

• The meaning of "\{ x \geq 0 \} x++ \{ x > 1 \}"
  - "If x>=0 and we execute x++, then x>1 will hold".
  - Counterexample
    - this statement is false because when x=0, x++ will be 1
    - x>1 won’t hold

• One way to show that a Hoare triple is false is to find a counterexample
Hoare Triples

• Why do we care?
  • We usually have some conclusion that we want to guarantee
    • postcondition
  • Given the code and the postcondition, what are the preconditions that guarantee the postcondition holds?
  • Typically requires backward reasoning
    • Can we reason about the code to find some precondition that will guarantee our postcondition?
    • Can we find a precondition that makes the Hoare triple true?
Rules for Backward Reasoning: Assignment

// precondition: ??
x = expression
// postcondition: Q

**Rule:** precondition is: Q with all occurrences of x in Q replaced by expression

// precondition:  y+1 > 0 => y>-1    // precondition:  z+1 > 0
x = y+1;                               z = z+1;
// postcondition: x>0                   // postcondition: z > 0
Weakest Precondition

Rule derives the weakest precondition

// precondition: y+1 > 0 (equiv. y > -1)

x = y+1

// postcondition: x>0

(y+1)>0 is the weakest precondition for code \(x=y+1\) and postcondition \(x>0\)

Notation: wp stands for weakest precondition

\(wp(\text{“x=expression;”},Q) = Q\) with all occurrences of \(x\) replaced by \(expression\)
Weaker and Stronger Conditions

• P is stronger than Q if P implies Q
  • P => Q
• If P is stronger than Q then P is more likely to be false than Q
• Example from politics:
  • “I will keep unemployment below 3%” is stronger than “I will keep unemployment below 15%”
• The strongest possible statement is always False
  • I will keep unemployment below 0%
• The weakest possible statement is always True
  • I will keep unemployment below 101%
Weaker and Stronger Conditions

• “P is stronger than Q” means “P implies Q”
• “P is stronger than Q” means “P guarantees more than Q”
  • x>0 is stronger than x>-1

Which one is stronger?

x > 0 && y = 0    or    x > 0 && y ≥ 0
0 ≤ x ≤ 10        or    0 ≤ x ≤ 1
x = 5 && y = 2 (mod 4)  or  x = 5 && y is even
Weaker and Stronger Conditions

Let the following be true:

\[ P \implies Q \quad Q \implies R \quad S \implies T \quad T \implies U \]

\{ Q \} code \{ T \}

“T \implies U” means “T implies U” or “T is stronger than U”

Then which of the following are true?

\{ P \} code \{ T \}
\{ R \} code \{ T \}
\{ Q \} code \{ S \}
\{ Q \} code \{ U \}
Weaker and Stronger Conditions

Let the following be true:

\[ P \implies Q \quad Q \implies R \quad S \implies T \quad T \implies U \]

\{ Q \} \text{ code } \{ T \}

“T \implies U” means “T implies U”
or “T is stronger than U”

Then which of the following are true?

\{ P \} \text{ code } \{ T \} \quad \text{true}
\{ R \} \text{ code } \{ T \} \quad \text{not necessarily}
\{ Q \} \text{ code } \{ S \} \quad \text{not necessarily}
\{ Q \} \text{ code } \{ U \} \quad \text{true}
Weaker and Stronger Conditions

• In **backward reasoning**, we determine the precondition, given code and a postcondition Q
  • We want the **weakest precondition**, \( \text{wp}(\text{code}, Q) \)
  • Find the minimal restriction the code places on the caller
  • We want the code to work in as many places as possible

• In **forward reasoning**, we determine the postcondition, given code and a precondition P
  • Normally we want the **strongest postcondition**
  • We want to guarantee as much as we can
Weakest Precondition

• Starting with a post-assertion, what is the weakest pre-condition that makes the assertion true?
  • What must be true beforehand to make the assertion true after
  • \([\text{WP} \&\& [\text{test} \&\& \text{action}] ] \rightarrow \text{Assertion}\)
  • Weakest preconditions yields the strongest specification for computation
• If \(A \Rightarrow B\) but not \((B \Rightarrow A)\), then \(B\) is “weaker” than \(A\), and \(A\) is “stronger” than \(B\)
• The weakest possible precondition is \(true\)
  • Since \(A \Rightarrow true\) is always true
• The strongest possible predicate is \(false\)
Weakest Precondition

• For each Q there are many P such that \{P\} code \{Q\}
• For each P there are many Q such that \{P\} code \{Q\}
• For each Q there is exactly one assertion \text{wp}(\text{code, Q})
• \{wp(S,Q)\} code \{Q\} is true
Weakest Precondition

• Consider \( x = x + 1 \) and postcondition \( x > 0 \)
• \( x > 0 \) is a valid precondition
  • \( \{x > 0\} \ x = x + 1 \ \{x > 0\} \) is true
• \( x > -1 \) is also a valid precondition
  • \( \{x > -1\} \ x = x + 1 \ \{x > 0\} \) is true
• \( x > -1 \) is weaker than \( x > 0 \)
  • \( x > 0 \implies x > -1 \)
• \( x > -1 \) is the weakest precondition
  • \( \text{wp}(x=x+1, \ x > 0) = x > -1 \)
Another Example

• Consider
  • \( a = a + 1 \)
  • \( b = b - 1 \)
  • Postcondition \( a \cdot b = 0 \)

• A very strong precondition
  • \((a=-1) \&\& (b=1)\)

• A weaker precondition
  • \( a = -1 \)

• Another weak precondition
  • \( b = 1 \)

• The weakest precondition
  • \((a=-1) \, || \, (b=1)\)

• \( \text{wp}(a = a + 1; \ b = b - 1, \ a \cdot b = 0) = (a=-1) \, || \, (b=1) \)
Backward Reasoning: Rule for Assignment

{ wp("x=<expression>", Q ) }
x = <expression>;
{ Q }

Rule: the weakest precondition wp( "x=expression", Q ) is Q with all occurrences of x in Q replaced by <expression>
Assignment Operations

• \( \wp(x = y + 5, (x > 5)) = \{y + 5 > 5\} \)  \( (\text{Substitute } y + 5 \text{ for } x) \)
  • \( \wp(x = y + 5, (x > 5)) = \{y + 5 > 5\} = \{y > 0\} \)  \( (\text{simplify}) \)

• \( \wp(x = x + 1, (x > 3)) = \{x + 1 > 3\} \)  \( (\text{substitute } x + 1 \text{ for } x) \)
  • \( \wp(x = x + 1, (x > 3)) = \{x + 1 > 3\} = \{x > 2\} \)  \( (\text{simplify}) \)
Rules for Backward Reasoning: Sequence

// precondition: ??
S1; // statement
S2; // another statement
// postcondition: Q
Work backwards:
precondition is \( \text{wp}("S1;S2;", Q) = \text{wp}("S1;", \text{wp}("S2;", Q)) \)

Example:
// precondition: ??
x = 0;
y = x+1;
// postcondition: y>0

// precondition: ??
x = 0;
// postcondition for x=0; same as
// precondition for y=x+1;
y = x+1;
// postcondition y>0
Example

\textit{precondition} : \textit{true}

\[ wp(x = 0; x + 1 > 0) = \{1 > 0\} = \{\text{true}\} \]

\[ x = 0 \]

\[ wp(y = x + 1; y > 0) = \{x + 1 > 0\} \]

\[ y = x + 1 \]

\textit{postcondition} : \textit{y > 0}
Exercise

// precondition: ??

x = x+1;

y = x + y;

// postcondition y>1
Exercise

precondition: \( x + y > 0 \)

\[
wp(x = x + 1; x + y > 1) = \{ x + 1 + y > 1 \} = \{ x + y > 0 \}
\]

\( x = x + 1 \)

\[
wp(y = x + y; y > 1) = \{ x + y > 1 \} \quad // \text{substitute for } y
\]

\( y = x + y \)

postcondition: \( y > 1 \)
Check by forward reasoning

precondition: $x_0 + y_0 > 0$

$x = x_0 + 1$

$$\{x = x_0 + 1 \& \& x_0 + y_0 > 0\} = \{x - 1 + y_0 > 0\} = \{x + y_0 > 1\}$$

$y = x + y_0$

$$\{y = x + y_0 \& \& x + y_0 > 1\} = \{y > 1\}$$

postcondition: $y > 1$
If-then-else Statement Example

// precondition: ??
if (x > 0) {
y = z;
}
else {
y = -z;
}
// postcondition: y>5

(z>5 && x>0) || (z<-5 && x≤0)

postcondition: y>5
If-then-else Statement Example

precondition: \((z > 5 \& \& x > 0) \| (z < -5 \& \& x \leq 0)\)

if \((x > 0)\) {
    wp(y = z; y > 5) = \{z > 5\}
    y = z
}
else {
    wp(y = -z; y > 5) = \{-z > 5\} = \{z < -5\}
    y = -z
}

postcondition: y > 5
Rules for Backward Reasoning: If-then-else

// precondition: ??
if (b) S1 else S2
// postcondition: Q

Case analysis, just as we did in the example:

wp("if (b) S1 else S2", Q)
    = { ( b && wp("S1",Q) ) || ( not(b) && wp("S2",Q) ) }
Exercise

Precondition: ??

\[ z = 0; \]

\[
\text{if} \ (x \neq 0) \ { \}

\qquad z = x;

\}

\text{else} \ { \}

\qquad z = z+1

\}

Postcondition: \( z > 0; \)
Exercise

precondition: \((x > 0) \land (x == 0)\)

\[ wp(z = 0; (x != 0 \land x > 0) \land (x == 0 \land z > -1)) = \{ (x > 0) \land (x == 0 \land 0 > -1) \} = \{ (x > 0) \land x == 0 \} \]

\( \text{postcondition: } z > 0 \)

if \((x != 0)\) {
    wp(z = x; z > 0) = \{ x > 0 \} // substitute for z
[
    z = x
\]
}
else {
    wp(z = z + 1; z > 0) = \{ z + 1 > 0 \} = \{ z > -1 \} // substitute for z
[
    z = z + 1
\]
}
Group Exercise

// precondition: ??
if (x < 5) {
    x = x*x;
}
else {
    x = x+1;
}

// postcondition: x ≥ 9
Group Exercise

// precondition \( x < 5 \& \& |x| \geq 3 \) \& \& \( x \geq 5 \& \& x \geq 8 \) = \{(x = 3 \| x = 4 \| x \leq -3) \| x \geq 8\}

if (x < 5) {
    wp(x = x \times x; x \geq 9) = \{x \times x \geq 9\} = \{|x| \geq 3\}
    x = x \times x;
}
else {
    wp(x = x + 1; x \geq 9) = \{x + 1 \geq 9\} = \{x \geq 8\}
    x = x + 1;
}
// postcondition x \geq 9
If-then-else Statement Review

Forward reasoning
{ P }
if b
{ P && b } S1
{ Q1 }
else
{ P && !b } S2
{ Q2 }
{ Q1 || Q2 }

Backward reasoning
{ (b && wp("S1",Q)) || ( !b && wp("S2",Q)) }
if b
{ wp("S1",Q) } S1
{ Q }
else
{ wp("S2",Q) } S2
{ Q }
{ Q }
If-then Statement

// precondition: ??

if (x > y) {
    z = x;
    x = y;
    y = z;
}

// postcondition: x < y
If-then Statement

// precondition $x > y \| x < y$

$\{(x > y \& \& x > y) \| (x \leq y \& \& x < y)\} = \{(x > y) \| (x < y)\}$

if $(x > y)$ {
    $wp(z = x; y < z) = \{y < x\} = \{x > y\}$
    
    $z = x$;
    $wp(x = y; x < z) = \{y < z\}$
    $x = y$;
    $wp(y = z; x < y) = \{x < z\}$
    $y = z$;
}

// postcondition: $x < y$
Backward Reasoning: Rule for Assignment

\[
\{ \text{wp}( \ "x=<expression>" , \ Q ) \} \\
x = <expression>; \\
\{ Q \}
\]

Rule: the weakest precondition \( \text{wp}( \ "x=expression" , \ Q ) \)
is \( Q \) with all occurrences of \( x \) in \( Q \) replaced
by \( <expression> \)
Backward Reasoning: Rule for Sequence

// find weakest precondition for sequence S1;S2 and Q

{ wp( S1, wp( S2, Q ) ) }
S1; // statement Postcondition for S1 is wp(S2, Q)
{ wp( S2, Q ) }
S2; // another statement
{ Q }
Backward Reasoning: Rule for If-then-else

\[
\{ ( b && \text{wp}( S1, Q ) ) \} || ( \text{not } b && \text{wp}( S2, Q ) ) \}
\]

if ( b ){
    S1;  // S1 and S2 could be multiple statements
}
else {
    S2;
}
{ Q }

... without the else:

\[
\{ ( b && \text{wp}( S1, Q ) ) \} || ( \text{not } b && Q ) \}
\]

if ( b ){
    S1;
}
{ Q }
Reasoning about Loops

Reasoning about loops is more difficult
-- Unknown number of iterations and unknown number of paths
-- Recursion adds an additional level of complexity

Key is loop invariant

Two things to prove about loops:
-- It computes correct values (partial correctness)
-- It terminates (does not go into an infinite loop)

Total correctness = Partial correctness + Loop termination
Reasoning about Loops

PRECONDITION: \{x \geq 0\}

\begin{align*}
i &= x \\
z &= 0
\end{align*}

while (i > 0) \{ LOOP INVARIANT (LI): i + z = x \}
\begin{align*}
&\{ \\
&\quad z = z + 1; \\
&\quad i = i - 1;
\end{align*}

POSTCONDITION: x = z && i=0

Questions:
(A) If the loop terminates, does POSTCONDITION x = z hold?
(B) Does the loop terminate?
Reasoning about Loops

Proof by Induction
(1) BASE CASE: Initially, i = x and z = 0 give us i+z=x, i.e., LI holds at iteration 0 (before the loop code executes)

(2) INDUCTION: Assuming i+z=x holds after iteration k, we show that i+z=x holds after iteration k+1

\[
z_{new} = z + 1 \quad \text{and} \quad i_{new} = i - 1
\]

therefore, \( i_{new} + z_{new} = i - 1 + z + 1 = i + z = x \)

(3) If the loop terminates, we know i = 0.
Since z+i=x holds, we have z = x (i.e., the POSTCONDITION)

How do we know if the loop terminates?
-- the PRECONDITION x \geq 0 guarantees that i \geq 0 before the loop.
At every iteration, i decreases by 1, thus it eventually reaches 0
Reasoning about Loops using Induction
-- i+z=x is a loop invariant, meaning that it holds true before
the loop and also after each/every iteration of the loop

-- even though i and z change within the loop code,
i+z=x stays true at the END of each iteration

-- Above we made an inductive argument over the number of iterations
of the given loop

-- Proof by Induction -- also called Computation Induction

-- Establish that the LI holds before iteration 0

-- Assuming LI holds after iteration k, show that it holds
after iteration k+1
Reasoning about Loops

Partial Correctness
-- Establish and prove the LI using computation induction

-- Loop exit condition and the LI must imply the desired postcondition
  -- i = 0 (loop exit condition) and i+z=x (LI) imply z=x

Termination
-- Establish some decrementing function $D$.
  Each loop iteration decrements $D$.
  LI and $D = \text{minimum value}$ imply loop exit condition
  LI and $D = \text{minimum}$ => not b
    b is the loop condition
Example

**precondition:** arr.length = len && len >= 0

```
int sum = 0;
int i = 0;
while ( i < len ) {
    sum = sum + arr[i];
    i = i + 1;
}
```

**postcondition:** (result is the sum of all elements on array arr)

```
sum = arr[0] + arr[1] + ... + arr[arr.length-1]
```
LI: $i \leq \text{len} \quad \&\& \quad \text{sum} = \text{arr}[0] + ... + \text{arr}[i-1]$

(1) BASE CASE: does the LI hold before the loop?

\begin{align*}
& i \leq \text{len} \quad \&\& \quad \text{sum} = \text{arr}[0] + ... + \text{arr}[i-1] \\
& \text{the LI does hold, given that } i = 0 \text{ and that no values from the array } \text{arr} \text{ have been summed yet}
\end{align*}

(2) INDUCTION: assume the LI holds at iteration $k$, does it hold at iteration $k+1$?

\begin{align*}
& \text{sum}_\text{new} = \text{sum} + \text{arr}[i] \\
& \text{i}_\text{new} = i + 1 \\
& \text{sum}_\text{new} = \text{arr}[0] + ... + \text{arr}[\text{i}_\text{new}-1] \\
& i \leq \text{len} \text{ also holds (if } i=\text{len} \text{ after iteration } k, \text{ there would be no iteration } k+1) \\
\end{align*}

To show that this loop does indeed terminate:

-- at each iteration, $i$ increases by one, while $\text{len}$ stays the same.
-- therefore, eventually, $i$ reaches $\text{len}$
-- and $i = \text{len}$, the loop exits (i.e., loop exit condition)
Partial correctness, more formally....

\{ P \} \textbf{while} \ ( b ) \ \textbf{S} \ \{ Q \}

We first need to "guess" a loop invariant (LI) such that

(1) \( P \Rightarrow LI \) \hspace{1em} \text{\textit{[BASE CASE]}} \hspace{1em} \text{\textit{[LOGIC]} \hspace{1em} [ASSUMPTIONS]}

(2) \{ b \&\& LI \} \ \textbf{S} \ \{ LI \} \hspace{1em} \text{Assuming that the LI held after iteration \( k \),}
\hspace{1em} \text{and execution performed iteration \( k+1 \), then}
\hspace{1em} \text{the LI holds after iteration \( k+1 \)}

(3) \{ \text{not} \ b \ \&\& LI \} \Rightarrow Q \hspace{1em} \text{The loop exit condition and the LI imply the desired postcondition}

Choosing the LI

\textbf{PRECONDITION:} \( x >= 0 \ \&\& \ y = 0 \)

\textbf{while} \ ( x != y ) \{} 
\hspace{1em} y = y + 1;
\}\n
\textbf{POSTCONDITION:} \( x = y \)
\hspace{1em} \text{what is the LI here?} 
\hspace{1em} y >= 0 
\hspace{1em} y leq x

if you guessed the LI to be \( y < x \), that's most of the way there
\hspace{1em} (i.e., it does not include when the loop will terminate)

the LI here is \( y leq x \) (because it includes/relates/comparces
\hspace{1em} both \( x \) and \( y \)
Example

PRECONDITION: $n \geq 0$

$i = 0$
$r = 1$

while ($i < n$) {
    $i = i + 1$
    $r = r \times i$
}

POSTCONDITION: $r = n!$ && $i = n$

what is the LI here?
PRECONDITION: \( n \geq 0 \)

\[
i = 0 \\
r = 1
\]

while ( \( i < n \) ) {
    \[
i = i + 1 \\
r = r * i
    \]
}

POSTCONDITION: \( r = n! \)

what is the LI here? \( i \leq n \) \&\& \( r = i! \)

show the above to be true in terms of Partial Correctness

BASE CASE: \( i = 0 \) and \( r = 1 \)

\[
i \leq n \) \&\& \( r = i! \\
0 \leq n \) \&\& \( 1 = 0! \) \ YES, base case holds (iteration 0)
\]

INDUCTIVE CASE:

\[
i_{\text{new}} = i + 1 \\
r_{\text{new}} = r * i_{\text{new}} \\
\]

// assume \( r = i! = (i_{\text{new}}-1)! \)

\[
r_{\text{new}} = (i_{\text{new}}-1)! * i_{\text{new}} = i_{\text{new}}! \\
r_{\text{new}} = i_{\text{new}}!
\]

TERMINATION: \( i \) increases by 1 at each iteration
Termination

Termination, more formally...

-- We need to "guess" a decrementing function D

\{ P \} \text{ while ( b ) S } \{ Q \}

We need D (with the range of natural numbers) such that

(1) \{ \text{ LI } \&\& b \} S \{ D_{\text{after}} < D_{\text{before}} \} \quad // \quad \text{One iteration of the loop reduces the value of } D

(2) ( \text{ LI } \&\& D=\text{min} ) \Rightarrow \text{ not b } \quad // \quad \text{LI combined with } D \text{ reaching minimum}

Note: In this case, if 0 is D's minimal value and must imply the loop exit condition. You can replace b with D > 0
Total correctness = Partial correctness + Loop termination

• Establish that the loop terminates
• Suppose the loop always reduces some variable’s value
  • Does the loop terminate if the variable is a
    • Natural number
    • Integer
    • Non-negative real
    • Boolean
    • List or Array
  • Loop terminates if the variable values are a subset of a well-ordered set
  • For an ordered set, every non-empty subset has a least element
Decrementing Function

• Decrementing function maps program variables to some well-ordered set

```c
// precondition: x ≥0 & & y = 0
// Loop invariant: x ≥ y
// D: (x-y)
while (x != y) {
    y = y + 1;
}
// postcondition: x = y
```

• Is x-y a good decrementing function?
Decrementing Function

• Does the loop reduce the decrementing function’s value?

\{ x! = y \}

\[ D_{\text{before}} = x_{\text{before}} - y_{\text{before}} \]

\[ y_{\text{after}} = y_{\text{before}} + 1 \]

\[ D_{\text{after}} = x_{\text{before}} - y_{\text{after}} = x_{\text{before}} - (y_{\text{before}} + 1) = x_{\text{before}} - y_{\text{before}} - 1 = D_{\text{before}} - 1 < D_{\text{before}} \]

• If the function is at a minimum does the loop exit?

\\{(x \geq y) \& \&(x - y = 0)\} = \{x = y\}
Example

PRECONDITION: \( x \geq 0 \)

\[
i = x
\]
\[
z = 0
\]

while ( \( i > 0 \) ) { LOOP INVARINT (LI): \( i + z = x \) }
{
    \[
z = z + 1;
\]
    \[
i = i - 1;
\]
}

POSTCONDITION: \( x = z \)

a decrementing function \( D \) is \( D = i \)
Exercise

precondition: \( \text{arr.length} = \text{len} \land \text{len} \geq 0 \)

```java
int sum = 0;
int i = 0;
while (i < len) {
    sum = sum + arr[i];
    i = i + 1;
}
```

postcondition: (result is the sum of all elements on array arr)

\[
\text{sum} = \text{arr}[0] + \text{arr}[1] + \ldots + \text{arr}[\text{arr.length}-1]
\]
PRECONDITION: \( x \geq 0 \)

\[
\begin{align*}
\text{zeros} &= 0; \\
y &= x; \\
\text{while ( } y \mod 10 == 0 \text{ ) } \\
    &\quad \{ \\
    &\quad \quad y = y / 10 \quad \text{ // integer division} \\
    &\quad \quad \text{zeros} = \text{zeros} + 1 \\
    &\quad \} \\
\end{align*}
\]

\( \text{zeros} \)

POSTCONDITION: \( x = y \times 10 \quad \&\& \quad y \mod 10 \neq 0 \)
Exercise

PRECONDITION: $x_1 > 0$ && $x_2 > 0$

$y_1 = x_1$
$y_2 = x_2$

while ( $y_1 != y_2$ ) {
  if ( $y_1 > y_2$ ) {
    $y_1 = y_1 - y_2$
  }
  else {
    $y_2 = y_2 - y_1$
  }
}

POSTCONDITION: $y_1 = \text{gcd}(x_1, x_2)$
Loops

Total correctness = Partial correctness + Loop termination

(1) Partial correctness

-- "Guess," then prove what the loop invariant (LI) is

-- Loop invariant and the loop exit condition must imply the given postcondition

-- This gives us:

"If the loop terminates, then the postcondition holds."

(2) Loop termination

-- "Guess" the decrementing function D. Each iteration of the loop decrements D, until D reaches a minimum. D at min along with the loop invariant must imply loop exit condition
Example

Before: \( x = 3, y = 2 \): \( r = 3, q = 0 \)
After: \( r = 1, q = 1 \)

integer division (calculate quotient \( q \) and remainder \( r \))

Precondition: \( x \geq 0 \) \&\& \( y > 0 \)
\( r = x \)
\( q = 0 \)
while ( \( y \leq r \) ) \quad \text{LI: } x = y \cdot q + r \quad ( \ y \leq x \quad q \geq 0 \ )
\{
    r = r - y \quad \text{(we aim to choose a LI that relates all of the}
    q = q + 1 \quad \text{"interesting" variables here)}
\}
Postcondition: \( x = y \cdot q + r \) \&\& \( r < y \)

(the LI often resembles the postcondition)

Decrementing function \( D \): the decrementing function \( D = r \)
(define the "0 case" as \( r > y \))

given that \( y > 0 \) and \( r = r - y \), we know that
\( r \) is decreasing after each iteration
Rules for Backward Reasoning: Method Call

// precondition: ??
x = foo()
// postcondition: Q
If method has no side-effects, just like assignment

// precondition: ??
x = Math.abs(y)
// postcondition: x = 1
Precondition is y = 1 || y = -1
Summary So Far

• Intro to reasoning about code. Concepts
  • Specifications, preconditions and postconditions, forward and backward reasoning

• Hoare triples

• Rules for backward reasoning
  • Rule for assignment
  • Rule for sequence of statements
  • Rule for if-then-else
In Practice

• Write loop invariants when unsure about a loop
• When you have evidence that a loop is not working
  • Add invariant and decrementing function
  • Write code to check them
  • Understand why the code doesn't work
  • fix
  • Reason to ensure that no similar bugs remain
In Practice

• Use the loop invariant to guide writing the loop
  • Determine the set of variables for the loop
  • Express the required condition at the end of the loop
    • Postcondition for the loop
  • Determine what holds before the loop executes
    • Precondition
  • Determine a decrementing function
    • What decreases with each iteration
      • Try to find a decrementing function with 0 as a minimum
  • Construct a loop invariant
    • What has to be true after each iteration
  • Use the loop invariant to construct the loop body
Why Do We Care?

• Correctness is important
  • Bugs are frustrating, expensive, and in some case dangerous

• Pre and postconditions for functions are specifications

• Optimizing compilers
  • Transform loops
  • Is the transformed loop the same as the original

• Thinking about code in a formal way leads to better code
  • Helps us solve problems
  • Helps us create code from specifications