

Program Correctness

OOSC2 Chapter 11



EECS3311 A: Software Design
Fall 2018

CHEN-WEI WANG

Weak vs. Strong Assertions

- Describe each assertion as **a set of satisfying value**.
 - $x > 3$ has satisfying values $\{ x \mid x > 3 \} = \{ 4, 5, 6, 7, \dots \}$
 - $x > 4$ has satisfying values $\{ x \mid x > 4 \} = \{ 5, 6, 7, \dots \}$
- An assertion p is **stronger** than an assertion q **if** p 's set of satisfying values is a subset of q 's set of satisfying values.
 - Logically speaking, p being stronger than q (or, q being weaker than p) means $p \Rightarrow q$.
 - e.g., $x > 4 \Rightarrow x > 3$
- What's the weakest assertion? [**TRUE**]
- What's the strongest assertion? [**FALSE**]
- In **Design by Contract** :
 - A **weaker invariant** has more acceptable object states
e.g., $balance > 0$ vs. $balance > 100$ as an invariant for ACCOUNT
 - A **weaker precondition** has more acceptable input values
 - A **weaker postcondition** has more acceptable output values

Motivating Examples (1)

Is this feature correct?

```
class FOO
  i: INTEGER
  increment_by_9
  require
    i > 3
  do
    i := i + 9
  ensure
    i > 13
  end
end
```

Q: Is $i > 3$ is too weak or too strong?

A: Too weak

\therefore assertion $i > 3$ allows value 4 which would fail postcondition.

Motivating Examples (2)

Is this feature correct?

```
class FOO
  i: INTEGER
  increment_by_9
    require
      i > 5
    do
      i := i + 9
    ensure
      i > 13
    end
  end
end
```

Q: Is $i > 5$ too weak or too strong?

A: Maybe too strong

∴ assertion $i > 5$ disallows 5 which would not fail postcondition.

Whether 5 should be allowed depends on the requirements.

- Correctness is a *relative* notion:
consistency of *implementation* with respect to *specification*.
⇒ This assumes there is a specification!
- We introduce a formal and systematic way for formalizing a program **S** and its *specification* (pre-condition **Q** and post-condition **R**) as a *Boolean predicate*: $\{Q\} S \{R\}$
 - e.g., $\{i > 3\} i := i + 9 \{i > 13\}$
 - e.g., $\{i > 5\} i := i + 9 \{i > 13\}$
 - If $\{Q\} S \{R\}$ **can** be proved **TRUE**, then the **S** is correct.
e.g., $\{i > 5\} i := i + 9 \{i > 13\}$ can be proved TRUE.
 - If $\{Q\} S \{R\}$ **cannot** be proved **TRUE**, then the **S** is incorrect.
e.g., $\{i > 3\} i := i + 9 \{i > 13\}$ cannot be proved TRUE.

Hoare Logic

- Consider a program **S** with precondition **Q** and postcondition **R**.
 - $\{Q\} S \{R\}$ is a **correctness predicate** for program **S**
 - $\{Q\} S \{R\}$ is TRUE if program **S** starts executing in a state satisfying the precondition **Q**, and then:
 - (a) The program **S** terminates.
 - (b) Given that program **S** terminates, then it terminates in a state satisfying the postcondition **R**.
 - Separation of concerns
 - (a) requires a proof of **termination**.
 - (b) requires a proof of **partial correctness**.
- Proofs of (a) + (b) imply **total correctness**.

Hoare Logic and Software Correctness

Consider the **contract view** of a feature f (whose body of implementation is S) as a **Hoare Triple**:

$$\{Q\} S \{R\}$$

Q is the **precondition** of f .

S is the implementation of f .

R is the **postcondition** of f .

- $\{true\} S \{R\}$
All input values are valid [Most-user friendly]
- $\{false\} S \{R\}$
All input values are invalid [Most useless for clients]
- $\{Q\} S \{true\}$
All output values are valid [Most risky for clients; Easiest for suppliers]
- $\{Q\} S \{false\}$
All output values are invalid [Most challenging coding task]
- $\{true\} S \{true\}$
All inputs/outputs are valid (No contracts) [Least informative]

Proof of Hoare Triple using wp

$$\{Q\} S \{R\} \equiv Q \Rightarrow wp(S, R)$$

- $wp(S, R)$ is the *weakest precondition for S to establish R*.
- S can be:
 - Assignments ($x := y$)
 - Alternations (**if** ... **then** ... **else** ... **end**)
 - Sequential compositions ($S_1 ; S_2$)
 - Loops (**from** ... **until** ... **loop** ... **end**)
- We will learn how to calculate the wp for the above programming constructs.

Hoare Logic A Simple Example

Given $\{??\}n := n + 9\{n > 13\}$:

- $n > 4$ is the **weakest precondition (wp)** for the given implementation ($n := n + 9$) to start and establish the postcondition ($n > 13$).
- Any precondition that is **equal to or stronger than** the wp ($n > 4$) will result in a correct program.
e.g., $\{n > 5\}n := n + 9\{n > 13\}$ can be proved **TRUE**.
- Any precondition that is **weaker than** the wp ($n > 4$) will result in an incorrect program.
e.g., $\{n > 3\}n := n + 9\{n > 13\}$ cannot be proved **TRUE**.
Counterexample: $n = 4$ satisfies precondition $n > 3$ but the output $n = 13$ fails postcondition $n > 13$.

Denoting New and Old Values

In the *postcondition*, for a program variable x :

- We write x_0 to denote its *pre-state (old)* value.
- We write x to denote its *post-state (new)* value.

Implicitly, in the *precondition*, all program variables have their *pre-state* values.

e.g., $\{b_0 > a\} b := b - a \{b = b_0 - a\}$

- Notice that:
 - We may choose to write “ b ” rather than “ b_0 ” in preconditions
 \therefore All variables are pre-state values in preconditions
 - We don't write “ b_0 ” in program
 \therefore there might be *multiple intermediate values* of a variable due to sequential composition

wp Rule: Assignments (1)

$$wp(x := e, R) = R[x := e]$$

$R[x := e]$ means to substitute all *free occurrences* of variable x in postcondition R by expression e .

wp Rule: Assignments (2)

Recall:

$$\{Q\} S \{R\} \equiv Q \Rightarrow wp(S, R)$$

How do we prove $\{Q\} x := e \{R\}$?

$$\{Q\} x := e \{R\} \iff Q \Rightarrow \underbrace{R[x := e]}_{wp(x := e, R)}$$

wp Rule: Assignments (3) Exercise

What is the weakest precondition for a program $x := x + 1$ to establish the postcondition $x > x_0$?

$$\{??\} x := x + 1 \{x > x_0\}$$

For the above Hoare triple to be **TRUE**, it must be that $?? \Rightarrow wp(x := x + 1, x > x_0)$.

$$\begin{aligned} & wp(x := x + 1, x > x_0) \\ = & \{Rule\ of\ wp:\ Assignment\} \\ & x > x_0[x := x_0 + 1] \\ = & \{Replacing\ x\ by\ x_0 + 1\} \\ & x_0 + 1 > x_0 \\ = & \{1 > 0\ always\ true\} \\ & True \end{aligned}$$

Any precondition is OK.

False is valid but not useful.

wp Rule: Assignments (4) Exercise

What is the weakest precondition for a program $x := x + 1$ to establish the postcondition $x > x_0$?

$$\{??\} x := x + 1 \{x = 23\}$$

For the above Hoare triple to be **TRUE**, it must be that $?? \Rightarrow wp(x := x + 1, x = 23)$.

$$\begin{aligned} & wp(x := x + 1, x = 23) \\ = & \{Rule\ of\ wp:\ Assignments\} \\ & x = 23[x := x_0 + 1] \\ = & \{Replacing\ x\ by\ x_0 + 1\} \\ & x_0 + 1 = 23 \\ = & \{arithmetic\} \\ & x_0 = 22 \end{aligned}$$

Any precondition weaker than $x = 22$ is not OK.

wp Rule: Alternations (1)

$$wp(\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end, } R) = \left(\begin{array}{l} B \Rightarrow wp(S_1, R) \\ \wedge \\ \neg B \Rightarrow wp(S_2, R) \end{array} \right)$$

The *wp* of an alternation is such that **all branches** are able to establish the postcondition ***R***.

wp Rule: Alternations (2)

Recall: $\{Q\} S \{R\} \equiv Q \Rightarrow wp(S, R)$

How do we prove that $\{Q\}$ if B then S_1 else S_2 end $\{R\}$?

```

{Q}
if B then
  {Q ∧ B} S1 {R}
else
  {Q ∧ ¬B} S2 {R}
end
{R}
  
```

$$\begin{aligned}
 & \{Q\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ end } \{R\} \\
 & \iff \left(\begin{array}{c} \{Q \wedge B\} S_1 \{R\} \\ \wedge \\ \{Q \wedge \neg B\} S_2 \{R\} \end{array} \right) \iff \left(\begin{array}{c} (Q \wedge B) \Rightarrow wp(S_1, R) \\ \wedge \\ (Q \wedge \neg B) \Rightarrow wp(S_2, R) \end{array} \right)
 \end{aligned}$$

wp Rule: Alternations (3) Exercise

Is this program correct?

```

{x > 0 ∧ y > 0}
if x > y then
  bigger := x ; smaller := y
else
  bigger := y ; smaller := x
end
{bigger ≥ smaller}
  
```

$$\left(\begin{array}{l} \{(x > 0 \wedge y > 0) \wedge (x > y)\} \\ \quad \text{bigger := x ; smaller := y} \\ \{bigger \geq smaller\} \end{array} \right)$$

$$\wedge$$

$$\left(\begin{array}{l} \{(x > 0 \wedge y > 0) \wedge \neg(x > y)\} \\ \quad \text{bigger := y ; smaller := x} \\ \{bigger \geq smaller\} \end{array} \right)$$

wp Rule: Sequential Composition (1)

$$wp(S_1 ; S_2, R) = wp(S_1, wp(S_2, R))$$

The *wp* of a sequential composition is such that the first phase establishes the *wp* for the second phase to establish the postcondition *R*.

wp Rule: Sequential Composition (2)

Recall:

$$\{Q\} S \{R\} \equiv Q \Rightarrow wp(S, R)$$

How do we prove $\{Q\} S_1 ; S_2 \{R\}$?

$$\{Q\} S_1 ; S_2 \{R\} \iff Q \Rightarrow \underbrace{wp(S_1, wp(S_2, R))}_{wp(S_1 ; S_2, R)}$$

wp Rule: Sequential Composition (3) Exercise

Is $\{ \text{True} \} \text{tmp} := x; x := y; y := \text{tmp} \{ x > y \}$ correct?
 If and only if $\text{True} \Rightarrow \text{wp}(\text{tmp} := x; x := y; y := \text{tmp}, x > y)$

$$\begin{aligned}
 & \text{wp}(\text{tmp} := x; \boxed{x := y; y := \text{tmp}}, x > y) \\
 = & \{ \text{wp rule for seq. comp.} \} \\
 & \text{wp}(\text{tmp} := x, \text{wp}(x := y; \boxed{y := \text{tmp}}, x > y)) \\
 = & \{ \text{wp rule for seq. comp.} \} \\
 & \text{wp}(\text{tmp} := x, \text{wp}(x := y, \text{wp}(y := \text{tmp}, x > \boxed{y}))) \\
 = & \{ \text{wp rule for assignment} \} \\
 & \text{wp}(\text{tmp} := x, \text{wp}(x := y, \boxed{x} > \text{tmp})) \\
 = & \{ \text{wp rule for assignment} \} \\
 & \text{wp}(\text{tmp} := x, y > \boxed{\text{tmp}}) \\
 = & \{ \text{wp rule for assignment} \} \\
 & y > x
 \end{aligned}$$

$\therefore \text{True} \Rightarrow y > x$ does not hold in general.

\therefore The above program is not correct.

- A loop is a way to compute a certain result by *successive approximations*.
e.g. computing the maximum value of an array of integers
- Loops are needed and powerful
- But loops **very hard** to get right:
 - Infinite loops [termination]
 - “off-by-one” error [partial correctness]
 - Improper handling of borderline cases [partial correctness]
 - Not establishing the desired condition [partial correctness]

Loops: Binary Search

<p style="text-align: center;">BS1</p> <pre> from i := 1; j := n until i = j loop m := (i + j) // 2 if t @ m <= x then i := m else j := m end end Result := (x = t @ i) </pre>	<p style="text-align: center;">BS2</p> <pre> from i := 1; j := n; found := false until i = j and not found loop m := (i + j) // 2 if t @ m < x then i := m + 1 elseif t @ m = x then found := true else j := m - 1 end end Result := found </pre>
<p style="text-align: center;">BS3</p> <pre> from i := 0; j := n until i = j loop m := (i + j + 1) // 2 if t @ m <= x then i := m + 1 else j := m end end if i >= 1 and i <= n then Result := (x = t @ i) else Result := false end </pre>	<p style="text-align: center;">BS4</p> <pre> from i := 0; j := n + 1 until i = j loop m := (i + j) // 2 if t @ m <= x then i := m + 1 else j := m end end if i >= 1 and i <= n then Result := (x = t @ i) else Result := false end </pre>

4 implementations for binary search: published, but *wrong!*

See page 381 in *Object Oriented Software Construction*

Correctness of Loops

How do we prove that the following loops are correct?

```

{Q}
from
  Sinit
until
  B
loop
  Sbody
end
{R}
  
```

```

{Q}
Sinit
while ( $\neg B$ ) {
  Sbody
}
{R}
  
```

- In case of C/Java, $\neg B$ denotes the **stay condition**.
- In case of Eiffel, B denotes the **exit condition**.

There is native, syntactic support for checking/proving the **total correctness** of loops.

Contracts for Loops: Syntax

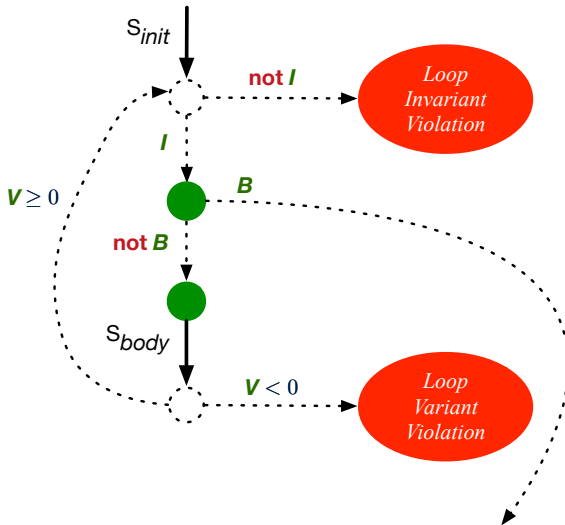
```
from
   $S_{init}$ 
invariant
  invariant_tag:  $I$  -- Boolean expression for partial correctness
until
   $B$ 
loop
   $S_{body}$ 
variant
  variant_tag:  $V$  -- Integer expression for termination
end
```


Contracts for Loops

- Use of **loop invariants (LI)** and **loop variants (LV)**.
 - **Invariants:** `Boolean` expressions for **partial correctness**.
 - Typically a special case of the postcondition.
e.g., Given postcondition “**Result is maximum of the array**”:
LI can be “**Result is maximum of the part of array scanned so far**”.
 - Established before the very first iteration.
 - Maintained `TRUE` after each iteration.
 - **Variants:** `Integer` expressions for **termination**
 - Denotes the **number of iterations remaining**
 - **Decreased** at the end of each subsequent iteration
 - Maintained **non-negative** at the end of each iteration.
 - As soon as value of **LV** reaches **zero**, meaning that no more iterations remaining, the loop must exit.
- Remember:

$$\text{total correctness} = \text{partial correctness} + \text{termination}$$

Contracts for Loops: Runtime Checks (1)



Contracts for Loops: Runtime Checks (2)

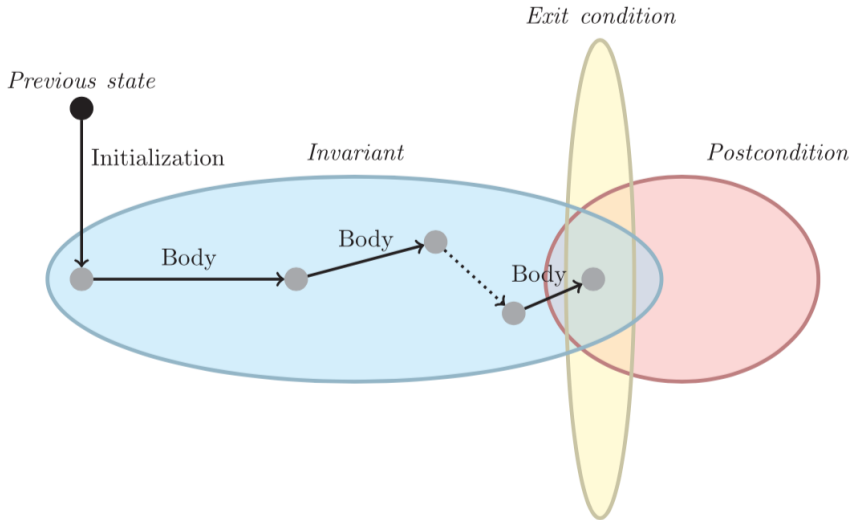
```
1 test
2   local
3     i: INTEGER
4   do
5     from
6       i := 1
7     invariant
8       1 <= i and i <= 6
9     until
10      i > 5
11    loop
12      io.put_string ("iteration " + i.out + "%N")
13      i := i + 1
14    variant
15      6 - i
16    end
17  end
```

L8: Change to $1 \leq i$ and $i \leq 5$ for a **Loop Invariant Violation**.

L10: Change to $i > 0$ to bypass the body of loop.

L15: Change to $5 - i$ for a **Loop Variant Violation**.

Contracts for Loops: Visualization



Contracts for Loops: Example 1.1

```

find_max (a: ARRAY [INTEGER]): INTEGER
  local i: INTEGER
  do
    from
      i := a.lower ; Result := a[i]
    invariant
      loop_invariant: --  $\forall j \mid a.lower \leq j \leq i \bullet Result \geq a[j]$ 
      across a.lower |..| i as j all Result >= a [j.item] end
    until
      i > a.upper
    loop
      if a [i] > Result then Result := a [i] end
      i := i + 1
    variant
      loop_variant: a.upper - i + 1
    end
  ensure
    correct_result: --  $\forall j \mid a.lower \leq j \leq a.upper \bullet Result \geq a[j]$ 
    across a.lower |..| a.upper as j all Result >= a [j.item]
  end
end

```

Contracts for Loops: Example 1.2

Consider the feature call `find_max(⟨⟨20, 10, 40, 30⟩⟩)`, given:

- **Loop Invariant:** $\forall j \mid a.lower \leq j \leq i \bullet Result \geq a[j]$
- **Loop Variant:** $a.upper - i + 1$

AFTER ITERATION	i	Result	LI	EXIT ($i > a.upper$)?	LV
Initialization	1	20	✓	×	—
1st	2	20	✓	×	3
2nd	3	20	×	—	—

Loop invariant violation at the end of the 2nd iteration:

$$\forall j \mid a.lower \leq j \leq 3 \bullet 20 \geq a[j]$$

evaluates to **false** $\because 20 \not\geq a[3] = 40$

Contracts for Loops: Example 2.1

```

find_max (a: ARRAY [INTEGER]): INTEGER
  local i: INTEGER
  do
    from
      i := a.lower ; Result := a[i]
    invariant
      loop_invariant: --  $\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]$ 
      across a.lower |..| (i - 1) as j all Result >= a [j.item] end
    until
      i > a.upper
    loop
      if a [i] > Result then Result := a [i] end
      i := i + 1
    variant
      loop_variant: a.upper - i
    end
  ensure
    correct_result: --  $\forall j \mid a.lower \leq j \leq a.upper \bullet Result \geq a[j]$ 
    across a.lower |..| a.upper as j all Result >= a [j.item]
  end
end

```

Contracts for Loops: Example 2.2

Consider the feature call `find_max(⟨⟨20, 10, 40, 30⟩⟩)`, given:

- **Loop Invariant:** $\forall j \mid a.lower \leq j < i$ • $Result \geq a[j]$
- **Loop Variant:** $a.upper - i$

AFTER ITERATION	i	Result	LI	EXIT ($i > a.upper$)?	LV
Initialization	1	20	✓	×	—
1st	2	20	✓	×	2
2nd	3	20	✓	×	1
3rd	4	40	✓	×	0
4th	5	40	✓	✓	-1

Loop variant violation at the end of the 2nd iteration

$\because a.upper - i = 4 - 5$ evaluates to **non-zero**.

Contracts for Loops: Example 3.1

```

find_max (a: ARRAY [INTEGER]): INTEGER
  local i: INTEGER
  do
    from
      i := a.lower ; Result := a[i]
    invariant
      loop_invariant: --  $\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]$ 
      across a.lower |..| (i - 1) as j all Result >= a [j.item] end
    until
      i > a.upper
    loop
      if a [i] > Result then Result := a [i] end
      i := i + 1
    variant
      loop_variant: a.upper - i + 1
    end
  ensure
    correct_result: --  $\forall j \mid a.lower \leq j \leq a.upper \bullet Result \geq a[j]$ 
    across a.lower |..| a.upper as j all Result >= a [j.item]
  end
end

```

Contracts for Loops: Example 3.2

Consider the feature call `find_max(⟨⟨20, 10, 40, 30⟩⟩)`, given:

- **Loop Invariant:** $\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]$
- **Loop Variant:** $a.upper - i + 1$
- **Postcondition:** $\forall j \mid a.lower \leq j \leq a.upper \bullet Result \geq a[j]$

AFTER ITERATION	i	Result	LI	EXIT ($i > a.upper$)?	LV
Initialization	1	20	✓	×	—
1st	2	20	✓	×	3
2nd	3	20	✓	×	2
3rd	4	40	✓	×	1
4th	5	40	✓	✓	0

Contracts for Loops: Exercise

```

class DICTIONARY[V, K]
feature {NONE} -- Implementations
  values: ARRAY[K]
  keys: ARRAY[K]
feature -- Abstraction Function
  model: FUN[K, V]
feature -- Queries
  get_keys(v: V): ITERABLE[K]
  local i: INTEGER; ks: LINKED_LIST[K]
  do
    from i := keys.lower ; create ks.make_empty
    invariant ??
    until i > keys.upper
    do if values[i] ~ v then ks.extend(keys[i]) end
    end
  Result := ks.new_cursor
ensure
  result_valid:  $\forall k \mid k \in \text{Result} \bullet \text{model.item}(k) \sim v$ 
  no_missing_keys:  $\forall k \mid k \in \text{model.domain} \bullet \text{model.item}(k) \sim v \Rightarrow k \in \text{Result}$ 
end
  
```

Proving Correctness of Loops (1)

```

{Q}   from
      Sinit
      invariant
      I
      until
      B
      loop
      Sbody
      variant
      V
      end   {R}
  
```

- A loop is **partially correct** if:
 - Given precondition Q , the initialization step S_{init} establishes $LI I$.
 - At the end of S_{body} , if not yet to exit, $LI I$ is maintained.
 - If ready to exit and $LI I$ maintained, postcondition R is established.
- A loop **terminates** if:
 - Given $LI I$, and not yet to exit, S_{body} maintains $LV V$ as non-negative.
 - Given $LI I$, and not yet to exit, S_{body} decrements $LV V$.

Proving Correctness of Loops (2)

$\{Q\}$ from S_{init} invariant I until B loop S_{body} variant V end $\{R\}$

- A loop is **partially correct** if:

- Given precondition Q , the initialization step S_{init} establishes LI .

$$\{Q\} S_{init} \{I\}$$

- At the end of S_{body} , if not yet to exit, LI is maintained.

$$\{I \wedge \neg B\} S_{body} \{I\}$$

- If ready to exit and LI maintained, postcondition R is established.

$$I \wedge B \Rightarrow R$$

- A loop **terminates** if:

- Given LI , and not yet to exit, S_{body} maintains LV V as non-negative.

$$\{I \wedge \neg B\} S_{body} \{V \geq 0\}$$

- Given LI , and not yet to exit, S_{body} decrements LV V .

$$\{I \wedge \neg B\} S_{body} \{V < V_0\}$$

Proving Correctness of Loops: Exercise (1.1)

Prove that the following program is correct:

```

find_max (a: ARRAY [INTEGER]): INTEGER
  local i: INTEGER
  do
    from
      i := a.lower ; Result := a[i]
    invariant
      loop_invariant:  $\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]$ 
    until
      i > a.upper
    loop
      if a [i] > Result then Result := a [i] end
      i := i + 1
    variant
      loop_variant: a.upper - i + 1
    end
  ensure
    correct_result:  $\forall j \mid a.lower \leq j \leq a.upper \bullet Result \geq a[j]$ 
  end
end
  
```

Proving Correctness of Loops: Exercise (1.2)

Prove that each of the following *Hoare Triples* is TRUE.

1. Establishment of Loop Invariant:

```
{ True }
  i := a.lower
  Result := a[i]
  {  $\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]$  }
```

2. Maintenance of Loop Invariant:

```
{  $(\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]) \wedge \neg(i > a.upper)$  }
  if a [i] > Result then Result := a [i] end
  i := i + 1
  {  $(\forall j \mid a.lower \leq j < i \bullet Result \geq a[j])$  }
```

3. Establishment of Postcondition upon Termination:

$$\begin{aligned}
 & (\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]) \wedge i > a.upper \\
 & \Rightarrow \forall j \mid a.lower \leq j \leq a.upper \bullet Result \geq a[j]
 \end{aligned}$$

Proving Correctness of Loops: Exercise (1.3)

Prove that each of the following **Hoare Triples** is TRUE.

4. Loop Variant Stays Non-Negative Before Exit:

```
{ (∀j | a.lower ≤ j < i • Result ≥ a[j]) ∧ ¬(i > a.upper) }  
  if a [i] > Result then Result := a [i] end  
  i := i + 1  
{ a.upper - i + 1 ≥ 0 }
```

5. Loop Variant Keeps Decrementing before Exit:

```
{ (∀j | a.lower ≤ j < i • Result ≥ a[j]) ∧ ¬(i > a.upper) }  
  if a [i] > Result then Result := a [i] end  
  i := i + 1  
{ a.upper - i + 1 < (a.upper - i + 1)0 }
```

where $(a.upper - i + 1)_0 \equiv a.upper_0 - i_0 + 1$

Proof Tips (1)

$$\{Q\} S \{R\} \Rightarrow \{Q \wedge P\} S \{R\}$$

In order to prove $\{Q \wedge P\} S \{R\}$, it is sufficient to prove a version with a **weaker** precondition: $\{Q\} S \{R\}$.

Proof:

- Assume: $\{Q\} S \{R\}$

It's equivalent to assuming: $\boxed{Q} \Rightarrow wp(S, R)$

(A1)

- To prove: $\{Q \wedge P\} S \{R\}$

- It's equivalent to proving: $Q \wedge P \Rightarrow wp(S, R)$
- Assume: $Q \wedge P$, which implies \boxed{Q}
- According to **(A1)**, we have $wp(S, R)$. ■

Proof Tips (2)

When calculating $wp(S, R)$, if either program S or postcondition R involves array indexing, then R should be augmented accordingly.

e.g., Before calculating $wp(S, a[i] > 0)$, augment it as

$$wp(S, a.lower \leq i \leq a.upper \wedge a[i] > 0)$$

e.g., Before calculating $wp(x := a[i], R)$, augment it as

$$wp(x := a[i], a.lower \leq i \leq a.upper \wedge R)$$

Index (1)

Weak vs. Strong Assertions

Motivating Examples (1)

Motivating Examples (2)

Software Correctness

Hoare Logic

Hoare Logic and Software Correctness

Proof of Hoare Triple using wp

Hoare Logic: A Simple Example

Denoting New and Old Values

wp Rule: Assignments (1)

wp Rule: Assignments (2)

wp Rule: Assignments (3) Exercise

wp Rule: Assignments (4) Exercise

wp Rule: Alternations (1)

Index (2)

wp Rule: Alternations (2)

wp Rule: Alternations (3) Exercise

wp Rule: Sequential Composition (1)

wp Rule: Sequential Composition (2)

wp Rule: Sequential Composition (3) Exercise

Loops

Loops: Binary Search

Correctness of Loops

Contracts for Loops: Syntax

Contracts for Loops

Contracts for Loops: Runtime Checks (1)

Contracts for Loops: Runtime Checks (2)

Contracts for Loops: Visualization

Contracts for Loops: Example 1.1

Index (3)

Contracts for Loops: Example 1.2

Contracts for Loops: Example 2.1

Contracts for Loops: Example 2.2

Contracts for Loops: Example 3.1

Contracts for Loops: Example 3.2

Contracts for Loops: Exercise

Proving Correctness of Loops (1)

Proving Correctness of Loops (2)

Proving Correctness of Loops: Exercise (1.1)

Proving Correctness of Loops: Exercise (1.2)

Proving Correctness of Loops: Exercise (1.3)

Proof Tips (1)

Proof Tips (2)