Program Correctness

OOSC2 Chapter 11

YORK UNIVERSITÉ UNIVERSITY

EECS3311 A: Software Design
Winter 2020

CHEN-WEI WANG

Assertions: Weak vs. Strong

- 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 $\boxed{\text{if}}$ 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

Assertions: Preconditions

Given *preconditions* P_1 and P_2, we say that

$\boxed{P_2 \textbf{ requires less} \text{ than } P_1}$ if

P_2 is ***less strict*** on (thus ***allowing more***) inputs than P_1 does.

$$\{ x \mid P_1(x) \} \subseteq \{ x \mid P_2(x) \}$$

More concisely:

$$P_1 \Rightarrow P_2$$

e.g., For command `withdraw(amount: INTEGER)`,

$\boxed{P_2 : amount \geq 0}$ ***requires less*** than $\boxed{P_1 : amount > 0}$

What is the *precondition* that ***requires the least***? [***true***]

Assertions: Postconditions

Given *postconditions* or *invariants* Q_1 and Q_2, we say that

$\boxed{Q_2 \textbf{ ensures more} \text{ than } Q_1}$ if

Q_2 is ***stricter*** on (thus ***allowing less***) outputs than Q_1 does.

$$\{ x \mid Q_2(x) \} \subseteq \{ x \mid Q_1(x) \}$$

More concisely:

$$Q_2 \Rightarrow Q_1$$

e.g., For query `q(i: INTEGER): BOOLEAN`,

$\boxed{Q_2 : \textbf{Result} = (i > 0) \wedge (i \textbf{ mod } 2 = 0)}$ ***ensures more*** than

$\boxed{Q_1 : \textbf{Result} = (i > 0) \vee (i \textbf{ mod } 2 = 0)}$

What is the *postcondition* that ***ensures the most***? [***false***]

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

$\because$ 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
```

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

A: Maybe too strong

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

Whether 5 should be allowed depends on the requirements.

Software Correctness

- Correctness is a *relative* notion:

  consistency of *implementation* with respect to *specification*.

  $\Rightarrow$ 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*: $\boxed{\{Q\}\ \mathbf{S}\ \{R\}}$
  - e.g., $\{i > 3\}$ `i := i + 9` $\{i > 13\}$
  - e.g., $\{i > 5\}$ `i := i + 9` $\{i > 13\}$
  - If $\boxed{\{Q\}\ \mathbf{S}\ \{R\}}$ **can** be proved **TRUE**, then the **S** is correct.

    e.g., $\{i > 5\}$ `i := i + 9` $\{i > 13\}$ can be proved TRUE.
  - If $\boxed{\{Q\}\ \mathbf{S}\ \{R\}}$ **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 $\boxed{\text{Hoare Triple}}$:

$$\{\boldsymbol{Q}\}\ \mathtt{S}\ \{\boldsymbol{R}\}$$

Q is the *precondition* of f.
$\mathtt{S}$ is the implementation of f.
R is the *postcondition* of f.

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

Proof of Hoare Triple using *wp*

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

- $wp(S, \boldsymbol{R})$ 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 $\{??\}n := n + 9\{n > 13\}$:

- $\boxed{n > 4}$ 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 $\boxed{x_0}$ to denote its ***pre-state (old)*** value.
- We write $\boxed{x}$ 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
    ∵ All variables are pre-state values in preconditions
  - We don't write "b_0" in program
    ∵ there might be *multiple intermediate values* of a variable due to sequential composition

wp Rule: Assignments (1)

$$wp(\texttt{x := e}, \boldsymbol{R}) = \boldsymbol{R}[x := e]$$

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

wp Rule: Assignments (2)

Recall:

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

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

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

wp Rule: Assignments (3) Exercise

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

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

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

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

Any precondition is OK. ***False*** is valid but not useful.

wp Rule: Assignments (4) Exercise

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

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

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

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

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

wp Rule: Alternations (1)

$$wp(\texttt{if } \boxed{B} \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end}, \boldsymbol{R}) = \begin{pmatrix} \boxed{B} \Rightarrow wp(S_1, \boldsymbol{R}) \\ \wedge \\ \neg \boxed{B} \Rightarrow wp(S_2, \boldsymbol{R}) \end{pmatrix}$$

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

wp Rule: Alternations (2)

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

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

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

$$\{\boldsymbol{Q}\}\ \texttt{if } \boxed{B} \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end}\ \{\boldsymbol{R}\}$$

$$\iff \begin{pmatrix} \{\ \boldsymbol{Q} \wedge \boxed{B}\ \}\ S_1\ \{\ \boldsymbol{R}\ \} \\ \wedge \\ \{\ \boldsymbol{Q} \wedge \neg \boxed{B}\ \}\ S_2\ \{\ \boldsymbol{R}\ \} \end{pmatrix} \iff \begin{pmatrix} (\boldsymbol{Q} \wedge \boxed{B}) \Rightarrow wp(S_1, \boldsymbol{R}) \\ \wedge \\ (\boldsymbol{Q} \wedge \neg \boxed{B}) \Rightarrow wp(S_2, \boldsymbol{R}) \end{pmatrix}$$

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}
```

$$\begin{pmatrix} \{(x > 0 \wedge y > 0) \wedge (x > y)\} \\ \quad \texttt{bigger := x ; smaller := y} \\ \{bigger \geq smaller\} \end{pmatrix}$$

$$\wedge$$

$$\begin{pmatrix} \{(x > 0 \wedge y > 0) \wedge \neg(x > y)\} \\ \quad \texttt{bigger := y ; smaller := x} \\ \{bigger \geq smaller\} \end{pmatrix}$$

wp Rule: Sequential Composition (1)

$$wp(S_1\ ;\ S_2, \boldsymbol{R}) = wp(S_1, wp(S_2, \boldsymbol{R}))$$

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:

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

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

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

wp Rule: Sequential Composition (3) Exercise

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

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

$\because$ $\boldsymbol{True} \Rightarrow y > x$ does not hold in general.
$\therefore$ The above program is not correct.

Loops

- 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

BS1

```
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)
```

BS2

```
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
```

BS3

```
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
```

BS4

```
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
```

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
  S_init
until
  B
loop
  S_body
end
{R}
```

```
{Q}
S_init
while(¬ B) {
  S_body
}
{R}
```

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

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

Contracts for Loops: Syntax

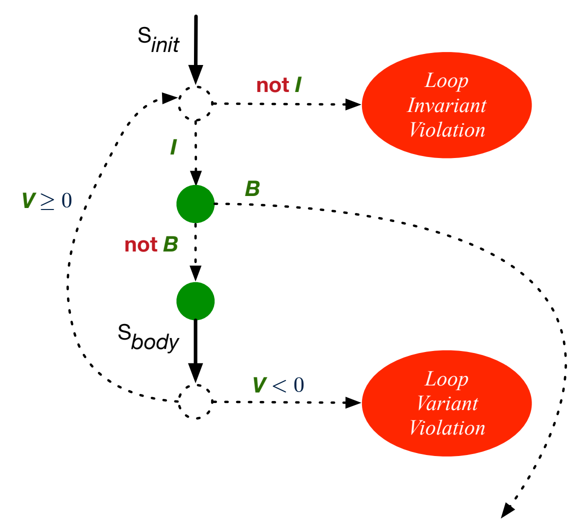
```
from
  S_init
invariant
  invariant_tag: I -- Boolean expression for partial correctness
until
  B
loop
  S_body
variant
  variant_tag: V -- Integer expression for termination
end
```


Contracts for Loops

- Use of ***loop invariants (LI)*** and ***loop variants (LV)***.
  - ***Invariants***: $\boxed{\text{Boolean}}$ 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***: $\boxed{\text{Integer}}$ 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:

  total *correctness* = ***partial*** *correctness* + ***termination***

Contracts for Loops: Runtime Checks (1)



Contracts for Loops: Runtime Checks (2)

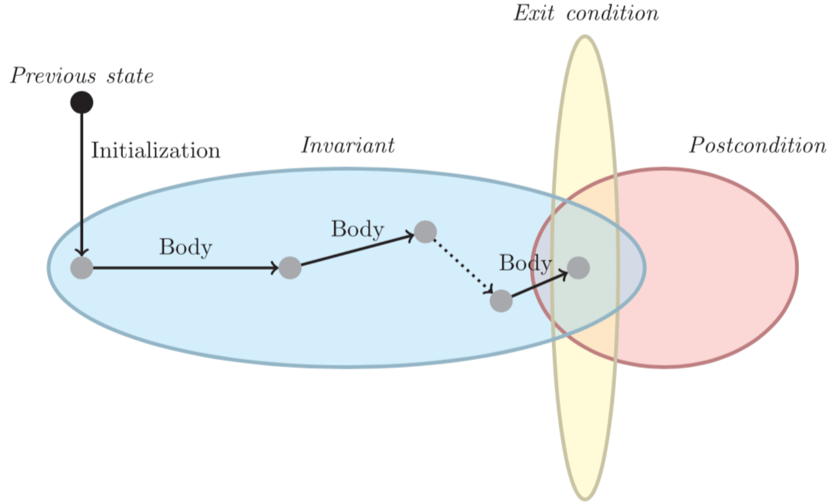
```
1  test
2    local
3      i: INTEGER
4    do
5      from
6        i := 1
7      invariant
8        1 <= i and i <= 6
9      until
10       i > 5
11     loop
12       io.put_string ("iteration " + i.out + "%N")
13       i := i + 1
14     variant
15       6 - i
16     end
17 end
```

L8: Change to `1 <= i and i <= 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



Digram Source: page 5 in *Loop Invariants: Analysis, Classification, and Examples*

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: -- ∀j | a.lower ≤ j ≤ i • Result ≥ 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: -- ∀j | a.lower ≤ j ≤ a.upper • Result ≥ 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 $\boxed{\text{find\_max}(\langle\langle 20, 10, 40, 30\rangle\rangle)}$, 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:

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

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: -- ∀j | a.lower ≤ j < i • Result ≥ 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: -- ∀j | a.lower ≤ j ≤ a.upper • Result ≥ 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***: $\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]$
- ***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: -- ∀j | a.lower ≤ j < i • Result ≥ 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: -- ∀j | a.lower ≤ j ≤ a.upper • Result ≥ 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: ∀k | k ∈ Result • model.item(k) ~ v
      no_missing_keys: ∀k | k ∈ model.domain • model.item(k) ~ v ⇒ k ∈ Result
    end
```


Proving Correctness of Loops (1)

```
{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** 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** I.

    $$\boxed{\{Q\}\ S_{init}\ \{I\}}$$
  - At the end of S_{body}, if not yet to exit, **LI** I is maintained.

    $$\boxed{\{I \wedge \neg B\}\ S_{body}\ \{I\}}$$
  - If ready to exit and **LI** I maintained, postcondition R is established.

    $$\boxed{I \wedge B \Rightarrow R}$$
- A loop *terminates* if:
  - Given **LI** I, and not yet to exit, S_{body} maintains **LV** V as non-negative.

    $$\boxed{\{I \wedge \neg B\}\ S_{body}\ \{V \geq 0\}}$$
  - Given **LI** I, and not yet to exit, S_{body} decrements **LV** V.

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

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: ∀j | a.lower ≤ j < i • Result ≥ 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: ∀j | a.lower ≤ j ≤ a.upper • Result ≥ 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]
{ ∀j | a.lower ≤ j < i • Result ≥ a[j] }
```

2. Maintenance of Loop Invariant:

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

3. Establishment of Postcondition upon Termination:

$$(\forall j \mid a.lower \leq j < i \bullet Result \geq a[j]) \land i > a.upper$$
$$\Rightarrow \forall j \mid a.lower \leq j \leq a.upper \bullet Result \geq a[j]$$

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)

$$\{Q\}\ \mathtt{S}\ \{R\} \Rightarrow \{Q \land P\}\ \mathtt{S}\ \{R\}$$

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

Proof:

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

Proof Tips (2)

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

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

$$wp(\mathtt{S}, a.lower \leq i \leq a.upper \land a[i] > 0)$$

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

$$wp(\mathtt{x := a[i]}, a.lower \leq i \leq a.upper \land R)$$

Index (1)

Index (2)

Index (3)

Index (4)