COSE419: Software Verification

Lecture 8 — Program Specification

Hakjoo Oh
2024 Spring

Program Verification

Techniques for specifying and verifying program properties:

- **Specification**: precise statement of program properties in first-order logic. Also called program annotations.
  - Partial correctness properties
  - Total correctness properties
- **Verification methods**: for proving partial/total correctness
  - Inductive assertion method
  - Ranking function method

Example 1: Linear Search

```
bool LinearSearch (int[] a, int l, int u, int e) {
  int i := l;
  while (i ≤ u) {
    if (a[i] = e) return true
    i := i + 1;
  }
  return false
}
```

Example 2: Binary Search

```
bool BinarySearch (int[] a, int l, int u, int e) {
  if (l > u) return false;
  else {
    int m := (l + u) div 2;
    if (a[m] = e) return true;
    else if (a[m] < e) return BinarySearch (a, m + 1, u, e)
    else return BinarySearch (a, l, m − 1, e)
  }
}
```

Example 3: Bubble Sort

```
int[] BubbleSort (int[] a_0) {
   int[] a := a_0
   for (int i := |a| − 1; i > 0; i := i − 1) {
      for (int j := 0; j < i; j := j + 1) {
         if (a[j] > a[j + 1]) {
            int t := a[j];
            int a[j] := a[j + 1];
            int a[j + 1] := t;
         }
      }
   }
   return a;
}
```

Example 3: Bubble Sort

2	3	4	1	2	5	6
j				i		
2	3	4	1	2	5	6
	j			i		
2	3	4	1	2	5	6
		j		i		
2	3	1	4	2	5	6
			j	i		
2	3	1	2	4	5	6
				j, i		
2	3	1	2	4	5	6
j			i			

Specification

- An annotation is a first-order logic formula F.
- An annotation F at location L expresses an *invariant* asserting that F is true whenever program control reaches L.
- Three types of annotations:
  - **Function specification**
  - **Loop invariant**
  - **Assertion**

Function Specifications

Formulas whose free variables include only the formal parameters and return variables.

- Precondition: Specification about what should be true upon entering the function.
- Postcondition: Specification about the expected output of the function. Postcondition relates the input and output of the function.

Example: Linear Search

The behavior of LinearSearch:

- It behaves correctly only when $l \geq 0$ and $u < |a|$.
- It returns true iff the array a contains the value e in the range $[l, u]$.

```
@pre : 0 ≤ l ∧ u < |a|
@post : rv ↔ ∃i.l ≤ i ≤ u ∧ a[i] = e
bool LinearSearch (int[] a, int l, int u, int e) {
  int i := l;
  while (i ≤ u) {
    if (a[i] = e) return true
    i := i + 1;
  }
  return false
}
```

Our goal is to prove the *partial correctness* property: if the function precondition holds and the function halts, then the function postcondition holds upon return.

Example: Binary Search

- It behaves correctly only when $l \geq 0$, $u < |a|$, and a is sorted.
- It returns true iff the array a contains the value e in the range $[l, u]$.

```
@pre : 0 ≤ l ∧ u < |a| ∧ sorted(a, l, u)
@post : rv ↔ ∃i. l ≤ i ≤ u ∧ a[i] = e
bool BinarySearch (int[] a, int l, int u, int e) {
  if (l > u) return false;
  else {
    int m := (l + u) div 2;
    if (a[m] = e) return true;
    else if (a[m] < e) return BinarySearch (a, m + 1, u, e)
    else return BinarySearch (a, l, m − 1, e)
  }
}
```

$$\textbf{sorted}(a, l, u) \iff \forall i, j.\, l \leq i \leq j \leq u \rightarrow a[i] \leq a[j]$$

Example: Bubble Sort

- Any array can be given.
- The returned array is sorted.

$$
\begin{array}{l}
\textbf{@pre}: |a_0| \geq 0 \\
\textbf{@post}: \textbf{sorted}(rv, 0, |rv| - 1) \\
\text{int[] BubbleSort (int[] } a_0) \; \{ \\
\quad \text{int[] } a := a_0 \\
\quad \text{for (int } i := |a| - 1;\ i > 0;\ i := i - 1) \; \{ \\
\quad\quad \text{for (int } j := 0;\ j < i;\ j := j + 1) \; \{ \\
\quad\quad\quad \text{if } (a[j] > a[j+1]) \; \{ \\
\quad\quad\quad\quad \text{int } t := a[j]; \\
\quad\quad\quad\quad \text{int } a[j] := a[j+1]; \\
\quad\quad\quad\quad \text{int } a[j+1] := t; \\
\quad\quad\quad \} \\
\quad\quad \} \\
\quad \} \\
\quad \text{return } a; \\
\}
\end{array}
$$

Loop Invariants

For proving partial correctness, each loop must be annotated with a loop invariant $\boldsymbol{F}$:

$$
\begin{array}{l}
\text{while} \\
\qquad @\boldsymbol{F} \\
\qquad (\langle \boldsymbol{condition} \rangle)\ \{ \\
\qquad \langle \boldsymbol{body} \rangle \\
\}
\end{array}
$$

Loop invariant is a property that is preserved by executions of the loop body; $\boldsymbol{F}$ holds at the beginning of every iteration. Therefore,

- $\boldsymbol{F} \wedge \langle \boldsymbol{condition} \rangle$ holds on entering the body.
- $\boldsymbol{F} \wedge \neg \langle \boldsymbol{condition} \rangle$ holds when exiting the loop.

Example: LinearSearch

```
@pre : 0 ≤ l ∧ u < |a|
@post : rv ↔ ∃i.l ≤ i ≤ u ∧ a[i] = e
bool LinearSearch (int[] a, int l, int u, int e) {
  int i := l;
  while
  @L : l ≤ i ∧ (∀j. l ≤ j < i → a[j] ≠ e)
  (i ≤ u) {
    if (a[i] = e) return true
    i := i + 1;
  }
  return false
}
```

Example: Bubble Sort

$$
\begin{array}{l}
\text{@pre} : |a_0| \geq 0 \\
\text{@post} : \mathsf{sorted}(rv, 0, |rv| - 1) \\
\text{int[] BubbleSort (int[] } a_0\text{) \{} \\
\quad \text{int[] } a := a_0 \\
\quad @L_1 \left[\begin{array}{l} -1 \leq i < |a| \\ \wedge\ \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge\ \mathsf{sorted}(a, i, |a|-1) \end{array} \right] \\
\quad \text{for (int } i := |a| - 1;\ i > 0;\ i := i - 1\text{) \{} \\
\qquad @L_2 \left[\begin{array}{l} 1 \leq i < |a| \ \wedge\ 0 \leq j \leq i \\ \wedge\ \mathsf{partitioned}(a, 0, i, i+1, |a|-1) \\ \wedge\ \mathsf{partitioned}(a, 0, j-1, j, j) \\ \wedge\ \mathsf{sorted}(a, i, |a|-1) \end{array} \right] \\
\qquad \text{for (int } j := 0;\ j < i;\ j := j + 1\text{) \{} \\
\qquad\quad \text{if (} a[j] > a[j+1]\text{) \{} \\
\qquad\qquad \text{int } t := a[j]; \\
\qquad\qquad \text{int } a[j] := a[j+1]; \\
\qquad\qquad \text{int } a[j+1] := t; \\
\qquad\quad \} \\
\qquad \} \\
\quad \} \\
\quad \text{return } a; \\
\}
\end{array}
$$

$$\mathsf{partitioned}(a, l_1, u_1, l_2, u_2) \iff \forall i, j.\ l_1 \leq i \leq u_1 < l_2 \leq j \leq u_2 \rightarrow a[i] \leq a[j]$$

Exercise 1

```
@pre : n ≥ 0
@post : rv = n
int SimpleWhile (int n) {
  int i := 0;
  while
  @L :
  (i < n) {
    i := i + 1;
  }
  return i
}
```

Exercise 2

$@\text{pre}: 0 \leq |a_0|$
$@\text{post}: \forall k.\ 0 \leq k < |rv| \rightarrow rv[k] \geq 0$

```
int[] AbsArray (int[] a₀) {
  int[] a := a₀;
  @L₁ :
  for (int i := 0; i < |a|; i := i + 1) {
    if (a[i] < 0) {
      a[i] := −a[i];
    }
  }
  return a;
}
```

Exercise 3

$@\text{pre} : 0 \leq |a_0|$
$@\text{post} :$

```
int LinearSearchIndex (int[] a, int key) {
  int idx := 0;
  while
  @L :
  (idx < |a|) {
    if (a[idx] = key) { return idx; }
    idx := idx + 1;
  }
  return −1;
}
```

Exercise 4

$$
\begin{array}{l}
\text{@pre} : 0 \leq |a| \land \textsf{sorted}(a, 0, |a|-1) \\
\text{@post} : (0 \leq rv \rightarrow (rv < |a| \land a[rv] = value)) \land \\
\qquad\quad (rv < 0 \rightarrow \forall k.\, 0 \leq k < |a| \rightarrow a[k] \neq value) \\
\text{int BinarySearchWhile (int[] } a\text{, int } value\text{) \{} \\
\quad \text{int } low := 0, high := |a|; \\
\quad \text{while} \\
\quad \text{@}L : \\
\\
\quad (low < high)\ \{ \\
\qquad mid := (low + high)/2; \\
\qquad \text{if } (a[mid] < value)\ \{\ low := mid + 1;\ \} \\
\qquad \text{else if } (value < a[mid])\ \{\ high := mid;\ \} \\
\qquad \text{else } \{\ \text{return } mid;\ \} \\
\quad \} \\
\quad \text{return } -1; \\
\}
\end{array}
$$

Exercise 5

$$@\text{pre} : |a| \geq 1$$
$$@\text{post} :$$

```
int FindMax (int[] a) {
  int i := 0, m := a[0];
  while
  @L :


  (i < |a|) {
    if (a[i] > m) { m := a[i]; }
    i := i + 1;
  }
  return m;
}
```

Assertions

- Programmers' formal comments on the program behavior
- Runtime assertions: division by 0, array out of bounds, etc

$@\text{pre} : 0 \leq l \land u < |a|$
$@\text{post} : \top$
bool LinearSearch (int[] a, int l, int u, int e) {
  int $i := l$;
  while
  $@L : \top$
  $(i \leq u)$ {
  $@0 \leq i < |a|$
    if $(a[i] = e)$ return true
    $i := i + 1$;
  }
  return false
}

Runtime Assertions: Binary Search

```
@pre : 0 ≤ l ∧ u < |a|
@post : ⊤
bool BinarySearch (int[] a, int l, int u, int e) {
  if (l > u) return false;
  else {
    @2 ≠ 0;
    int m := (l + u) div 2;
    @0 ≤ m < |a|;
    if (a[m] = e) return true;
    else if (a[m] < e) return BinarySearch (a, m + 1, u, e)
    else return BinarySearch (a, l, m − 1, e)
  }
}
```

Runtime Assertions: Bubble Sort

```
@pre : |a₀| ≥ 0
@post : ⊤
int[] BubbleSort (int[] a₀) {
  int[] a := a₀
  @⊤
  for (int i := |a| − 1; i > 0; i := i − 1) {
    @⊤
    for (int j := 0; j < i; j := j + 1) {
      @0 ≤ j < |a|
      @0 ≤ j + 1 < |a|
      if (a[j] > a[j + 1]) {
      @0 ≤ j < |a|
        int t := a[j];
      @0 ≤ j < |a|
      @0 ≤ j + 1 < |a|
        int a[j] := a[j + 1];
      @0 ≤ j + 1 < |a|
        int a[j + 1] := t;
      }
    }
  }
  return a;
}
```

Summary

Specifying partial correctness of programs:

- function pre/postconditions
- loop invariants
- runtime assertions