

AAA528: Computational Logic

Lecture 6 — Program Specification

Hakjoo Oh
2026 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 ( $i \leq u$ ) {  
        if ( $a[i] = e$ ) return true  
         $i := i + 1$ ;  
    }  
    return false  
}
```

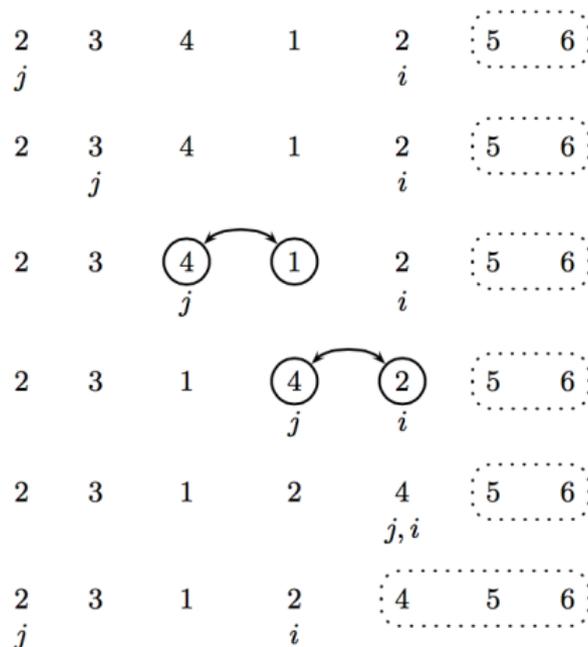
Example 2: Binary Search

```
bool BinarySearch (int[]  $a$ , int  $l$ , int  $u$ , int  $e$ ) {  
    if ( $l > u$ ) return false;  
    else {  
        int  $m := (l + u) \text{ div } 2$ ;  
        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



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 $[l, u]$.

$@pre : 0 \leq l \wedge u < |a|$

$@post : rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$

```
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 : $0 \leq l \wedge u < |a| \wedge \text{sorted}(a, l, u)$

@post : $rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$

```
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)  
    }  
}
```

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

Example: Bubble Sort

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

@pre : $|a_0| \geq 0$

@post : **sorted**($rv, 0, |rv| - 1$)

```
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  $a$ ;
}
```

Loop Invariants

For proving partial correctness, each loop must be annotated with a loop invariant F :

```
while
    @ $F$ 
    ( $\langle condition \rangle$ ) {
         $\langle body \rangle$ 
    }
```

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

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

Example: LinearSearch

```
@pre :  $0 \leq l \wedge u < |a|$   
@post :  $rv \leftrightarrow \exists i. l \leq i \leq u \wedge a[i] = e$   
bool LinearSearch (int[]  $a$ , int  $l$ , int  $u$ , int  $e$ ) {  
    int  $i := l$ ;  
    while  
        @L :  $l \leq i \wedge (\forall j. l \leq j < i \rightarrow a[j] \neq e)$   
        ( $i \leq u$ ) {  
            if ( $a[i] = e$ ) return true  
             $i := i + 1$ ;  
        }  
    return false  
}
```

Example: Bubble Sort

```
@pre :  $|a_0| \geq 0$ 
@post : sorted(rv, 0, |rv| - 1)
int[] BubbleSort (int[] a0) {
  int[] a := a0
  @L1 [
     $-1 \leq i < |a|$ 
     $\wedge$  partitioned(a, 0, i, i + 1, |a| - 1)
     $\wedge$  sorted(a, i, |a| - 1)
  ]
  for (int i := |a| - 1; i > 0; i := i - 1) {
    @L2 [
       $1 \leq i < |a| \wedge 0 \leq j \leq i$ 
       $\wedge$  partitioned(a, 0, i, i + 1, |a| - 1)
       $\wedge$  partitioned(a, 0, j - 1, j, j)
       $\wedge$  sorted(a, i, |a| - 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;
}
```

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

Exercise 1

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

Exercise 2

```
@pre :  $0 \leq |a_0|$   
@post :  $\forall k. 0 \leq k < |rv| \rightarrow rv[k] \geq 0$   
int[] AbsArray (int[]  $a_0$ ) {  
  int[]  $a := a_0$ ;  
  @L1 :  
  for (int  $i := 0$ ;  $i < |a|$ ;  $i := i + 1$ ) {  
    if ( $a[i] < 0$ ) {  
       $a[i] := -a[i]$ ;  
    }  
  }  
  return  $a$ ;  
}
```

Exercise 3

@pre : $0 \leq |a_0|$

@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

@pre : $0 \leq |a| \wedge \text{sorted}(a, 0, |a| - 1)$

@post : $(0 \leq rv \rightarrow (rv < |a| \wedge a[rv] = value)) \wedge$
 $(rv < 0 \rightarrow \forall k. 0 \leq k < |a| \rightarrow a[k] \neq value)$

int BinarySearchWhile (int[] *a*, int *value*) {

 int *low* := 0, *high* := |*a*|;

 while

@L :

 (*low* < *high*) {

mid := (*low* + *high*)/2;

 if (*a*[*mid*] < *value*) { *low* := *mid* + 1; }

 else if (*value* < *a*[*mid*]) { *high* := *mid*; }

 else { return *mid*; }

 }

 return -1;

}

Exercise 5

@pre : $|a| \geq 1$

@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

$@pre : 0 \leq l \wedge u < |a|$

$@post : \top$

```
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 : $0 \leq l \wedge u < |a|$

@post : \top

```
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 : T
int[] BubbleSort (int[] a0) {
  int[] a := a0
  @T
  for (int i := |a| - 1; i > 0; i := i - 1) {
    @T
    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