AAA528: Computational Logic

Lecture 8 — Partial Correctness Proof: Exercises

Hakjoo Oh
2025 Spring

Example 1: BinarySearchWhile

```
predicate sorted(a: array<int>, l: int, u: int) {
 forall j, k :: 0 <= l <= j <= k <= u < a.Length ==> a[j] <= a[k]
}

method BinarySearchWhile(a: array<int>, value: int) returns (index : int)
 requires 0 <= a.Length && sorted(a, 0, a.Length - 1)
 ensures (0 <= index ==> index < a.Length && a[index] == value)
 ensures (index < 0 ==> forall k :: 0 <= k && k < a.Length ==> a[k] != value) {
  var low : int := 0;
  var high : int := a.Length;
  var mid : int;
  while (low < high)
    invariant sorted(a, 0, a.Length - 1)
    invariant 0 <= low <= high <= a.Length
    invariant forall i :: 0 <= i < low ==> a[i] != value
    invariant forall i :: high <= i < a.Length ==> a[i] != value
  {
    mid := (low + high) / 2;
    if (a[mid] < value) { low := mid + 1; }
    else if (value < a[mid]) { high := mid; }
    else { return mid; }
  }
  return -1;
}
```

Example 2: FindMax

```
method FindMax (a: array<int>) returns (r: int)
 requires 0 < a.Length
 ensures forall k :: 0 <= k < a.Length ==> a[k] <= r
 ensures exists k :: 0 <= k < a.Length && a[k] == r
{
 var i : int := 0;
 r := a[0];

 while (i < a.Length)
  invariant a.Length > 0
  invariant 0 <= i <= a.Length
  invariant forall k :: 0 <= k < i ==> r >= a[k]
  invariant (a[0] == r || exists k :: 0 <= k < i && a[k] == r)
 {
  if(a[i] > r) { r := a[i]; }
  i := i + 1;
 }
}
```

Example 3: ReverseUptoK

```
method ReverseUptoK (s_in: array<int>, k: int) returns (s : array<int>)
    requires 2 <= k <= s_in.Length
    ensures forall i :: 0 <= i < k ==> s[i] == s_in[k - 1 - i]
    ensures forall i :: k <= i < s.Length ==> s[i] == s_in[i]
{
 var l : int := k - 1;
 var i : int := 0;
 var tmp : int;
 s := s_in;

 while (i < l - i)
  invariant 0 <= i <= (l + 1) / 2
  invariant forall p :: (0 <= p < i) || (l - i < p && p <= l) ==> s[p] == s_in[l-p]
  invariant forall p :: i <= p <= l - i ==> s[p] == s_in[p]
  invariant forall p :: k <= p < s.Length ==> s[p] == s_in[p]
  invariant l == k - 1
 {
  tmp := s[i];
  s[i] := s[l - i];
  s[l - i] := tmp;
  i := i + 1;
 }
}
```

Example 4: SelectionSort

```
predicate sorted(a: array<int>, l: int, u: int) {
 forall j, k :: 0 <= l <= j <= k <= u < a.Length ==> a[j] <= a[k]
}

predicate partitioned(a: array<int>, l1: int, u1: int, l2: int, u2: int) {
 forall x, y :: 0 <= l1 <= x <= u1 < l2 <= y <= u2 < a.Length ==> a[x] <= a[y]
}

method SelectionSort (a0: array<int>) returns (a: array<int>)
 requires a0.Length >= 0
 ensures sorted(a, 0, a.Length - 1)
{
 var i : int := 0;
 var tmp : int;
 var m : int;
 var j : int;
 a := a0;
 i := 0;
 while (i < a.Length)
  invariant 0 <= i <= a.Length
  invariant partitioned(a, 0, i-1, i, a.Length-1)
  invariant sorted(a, 0,i - 1)
 {
  m := i;
  j := i;

  while(j < a.Length)
   invariant 0 <= i <= a.Length
   invariant partitioned(a, 0, i-1, i, a.Length-1)
   invariant sorted(a, 0,i-1)
   invariant i <= j <= a.Length
   invariant i <= m < a.Length
   invariant forall k :: i <= k < j ==> a[k] >= a[m]
  {
   if(a[j] < a[m]) { m := j; }
   j := j + 1;
  }
  tmp := a[m];
  a[m] := a[i];
  a[i] := tmp;
  i := i + 1;
 }
}
```

Example 5: MergeSort

```
predicate sorted(a: array<int>, l: int, u: int)
{
 forall j, k :: 0 <= l <= j <= k <= u < a.Length ==> a[j] <= a[k]
}

predicate beq(a: array<int>, b: array<int>, k1: int, k2: int)
{
  forall i :: 0 <= k1 <= i <= k2 < a.Length && k2 < b.Length  ==> a[i] == b[i]
}

method MergeSort(a0: array<int>) returns (a: array<int>)
    requires 6 >= a0.Length && a0.Length >= 0
    ensures sorted(a, 0, a.Length - 1)
{
    a := ms(a0, 0, a0.Length - 1);
}
```

```
method ms(a0: array<int>, l : int, u : int) returns (a : array<int>)
    requires 0 <= l && u < a0.Length && 6 >= a0.Length && a0.Length >= 0
    ensures a.Length == a0.Length
    ensures beq(a, a0, 0, l - 1)
    ensures beq(a, a0, u + 1, a0.Length - 1)
    ensures sorted(a, l, u)
{
  var m : int;
  a := a0;
  if (l >= u) { return a; }
  else {
    m := (l + u) / 2;
    a := ms(a, l, m);
    a := ms(a, m + 1, u);
    a := merge(a, l, m, u);
    return a;
  }
}
```

```
method merge(a0 : array<int>, l : int, m : int, u : int) returns (a : array<int>)
    requires 6 >= a0.Length >= 0 && 0 <= l <= u < a0.Length && m == (l + u) / 2
    requires sorted (a0, l, m)
    requires sorted (a0, m + 1, u)
    ensures a.Length == a0.Length
    ensures sorted (a, l, u)
    ensures beq(a, a0, 0, l - 1)
    ensures beq(a, a0, u + 1, a0.Length - 1)
{
  var i : int;
  var j : int;
  var k : int;
  var buf: array<int>;

  a := a0;
  buf := new [u - l + 1];
  i := l;
  j := m + 1;
  k := 0;
```

```
while (k < buf.Length)
      invariant l <= i <= m + 1
      invariant m + 1 <= j <= u + 1
      invariant 0 <= k <= buf.Length
      invariant k == (i - l) + (j - (m + 1))
      invariant forall x:: 0 <= x < k && i <= m ==> buf[x] <= a[i]
      invariant forall x:: 0 <= x < k && j <= u ==> buf[x] <= a[j]
      invariant sorted (buf, 0, k - 1)

      invariant buf.Length == u - l + 1
      invariant a.Length == a0.Length
      invariant beq (a, a0, 0, a.Length - 1)

      invariant 6 >= a0.Length >= 0
      invariant 0 <= l <= u < a0.Length && m == (l + u) / 2
      invariant sorted (a0, l, m)
      invariant sorted (a0, m + 1, u)
```

```
{
  if (i > m) {
    buf[k] := a[j];
    j := j + 1;
  } else if (j > u) {
    buf[k] := a[i];
    i := i + 1;
  } else if (a[i] <= a[j]) {
    buf[k] := a[i];
    i := i + 1;
  } else {
    buf[k] := a[j];
    j := j + 1;
  }
  k := k + 1;
}

k := 0;
```

```
  while (k < buf.Length)
        invariant 0 <= k && k <= buf.Length
        invariant beq(a, a0, 0, l - 1)
        invariant beq(a, a0, u + 1, a0.Length - 1)
        invariant forall x :: 0 <= x < k ==> a[l + x] == buf[x]
        invariant sorted(a, l, l + k - 1)
        invariant sorted (buf, 0, buf.Length - 1) //

        invariant buf.Length == u - l + 1
        invariant a.Length == a0.Length

        invariant 6 >= a0.Length >= 0
        invariant 0 <= l <= u < a0.Length && m == (l + u) / 2
        invariant sorted (a0, l, m)
        invariant sorted (a0, m + 1, u)
  {
    a[l + k] := buf[k];
    k := k + 1;
  }

  return a;
}
```

Example 6: Partition

```
predicate partitioned(a: array<int>, l1: int, u1: int, l2: int, u2: int)
{
 forall x, y ::
    0 <= l1<= x <= u1 < l2 <= y <= u2 < a.Length ==> a[x] <= a[y]
}

predicate beq(a: array<int>, b: array<int>, k1: int, k2: int)
{
  forall i :: k1 <= i <= k2 ==> a[i] == b[i]
}

function random (l: int, u: int) : int
{
 l
}
```

```
method Partition(a0: array<int>, l: int, u: int) returns (pivot: int, a: array<int>
  requires 0 <= l <= u < a0.Length < 5
  requires partitioned(a0, 0, l - 1, l, u)
  requires partitioned(a0, l, u, u + 1, a0.Length - 1)
  ensures a.Length == a0.Length
  ensures beq(a, a0, 0, l - 1)
  ensures beq(a, a0, u + 1, a0.Length - 1)
  ensures l <= pivot <= u
  ensures partitioned(a, l, pivot - 1, pivot, pivot)
  ensures partitioned(a, pivot, pivot, pivot + 1, u)
{
  var pi : int := random(l, u);
  var pv : int;
  var i : int := l - 1;
  var j : int := l;

  a := a0;
  pv := a[pi];
  a[pi] := a[u];
  a[u] := pv;
  while (j < u)
    invariant a[u] == pv
    invariant a.Length == a0.Length && a0.Length < 5
    invariant beq(a, a0, 0, l - 1)
    invariant beq(a, a0, u + 1, a0.Length - 1)
    invariant l - 1 <= i < j <= u
    invariant forall k :: l <= k <= i ==> a[k] <= pv
    invariant forall k :: i + 1 <= k < j ==> a[k] > pv
  {
    if (a[j] <= pv) {
      i := i + 1;
      a[i], a[j] := a[j], a[i];
    }
    j := j + 1;
  }

  a[i+1], a[u] := a[u], a[i+1];
  return i+1, a;
}
```

Exercise Set 1

1. InsertionSort
2. InvertArray
3. IsPalindrome
4. PartitionOddEven
5. Union

Exercise Set 2

1. FindFirstRepeatedInt
2. FirstEvenOddDifference
3. LucidNumbers
4. mContained
5. TwoSum