

Program Verification - Assignment 7

Your Name(s)

November 12, 2025

1 T21: Warmup: Predicates and entailment

1.1 (a) Define the predicate `tree(x)`

We define the binary tree predicate recursively as follows:

```
1 predicate tree(this: Ref) {  
2   acc(this.value) && acc(this.left) && acc(this.right)  
3   &&  
4   (this.left != null ==> tree(this.left)) &&  
5   (this.right != null ==> tree(this.right))  
6 }
```

This predicate asserts that:

- We have access permissions to the `value`, `left`, and `right` fields of the node `this`
- If the left child exists (is not null), it must also satisfy the `tree` predicate recursively
- If the right child exists (is not null), it must also satisfy the `tree` predicate recursively

This captures the standard definition of a heap-based binary tree where each node has a value and pointers to left and right children.

1.2 (b) Implement the method `client`

The `client` method creates a tree with value 5 at the root, and two leaf children with values 1 (left) and 2 (right):

```
1 method client() {  
2   var t: Ref  
3   var leftChild: Ref  
4   var rightChild: Ref  
5  
6   // Create left leaf with value 1
```

```

7   leftChild := new(value, left, right)
8   leftChild.value := 1
9   leftChild.left := null
10  leftChild.right := null
11  fold tree(leftChild)
12
13  // Create right leaf with value 2
14  rightChild := new(value, left, right)
15  rightChild.value := 2
16  rightChild.left := null
17  rightChild.right := null
18  fold tree(rightChild)
19
20  // Create root with value 5
21  t := new(value, left, right)
22  t.value := 5
23  t.left := leftChild
24  t.right := rightChild
25  fold tree(t)
26
27  assert tree(t)
28 }

```

We construct the tree bottom-up: first creating the leaf nodes and folding the `tree` predicate for each, then creating the root and folding its `tree` predicate.

1.3 (c) Implement `left_leaf` and `right_leaf` functions

These functions return the leftmost and rightmost leaves respectively:

```

1 function left_leaf(this: Ref): Ref
2   requires tree(this)
3 {
4   unfolding tree(this) in (
5     this.left == null ? this : left_leaf(this.left)
6   )
7 }
8
9 function right_leaf(this: Ref): Ref
10  requires tree(this)
11 {
12   unfolding tree(this) in (
13     this.right == null ? this : right_leaf(this.right)
14   )
15 }

```

Both functions use recursion: `left_leaf` follows left pointers until reaching a node with no left child (the leftmost leaf), and `right_leaf` follows right

pointers until reaching a node with no right child (the rightmost leaf).

1.4 (d) Prove the entailment

We prove that $\text{tree}(x) \models \text{lsegl}(x, \text{left_leaf}(x)) * \text{lsegr}(x, \text{right_leaf}(x))$ by implementing the lemma method:

```

1 method tree_models_lsegs(this: Ref)
2   requires tree(this)
3   ensures lsegl(this, old(left_leaf(this)))
4   ensures lsegr(this, old(right_leaf(this)))
5 {
6   var ll: Ref := left_leaf(this)
7   var rl: Ref := right_leaf(this)
8
9   unfold tree(this)
10
11   if (this.left == null) {
12     fold lsegl(this, this)
13   } else {
14     tree_models_lsegs(this.left)
15     fold lsegl(this, ll)
16   }
17
18   if (this.right == null) {
19     fold lsegr(this, this)
20   } else {
21     tree_models_lsegs(this.right)
22     fold lsegr(this, rl)
23   }
24 }

```

The proof proceeds by structural induction: we unfold the tree predicate, recursively prove the property for the left and right subtrees, and then fold the segment predicates at the current node.

2 T22: Tree Rotations

Tree rotations are fundamental operations on binary trees that change the tree's structure while preserving the in-order traversal. A right rotation transforms a tree by promoting the left child to become the new root.

2.1 (a) Implement rotateRight

The right rotation operation is implemented as follows:

```

1 method rotateRight(root: Ref) returns (newRoot: Ref)
2   requires root != null

```

```

3   requires tree(root)
4   requires unfolding tree(root) in root.left != null
5   ensures tree(newRoot)
6   ensures inorder(newRoot) == old(inorder(root))
7 {
8   unfold tree(root)
9   var y: Ref := root
10  var x: Ref := y.left
11  var C: Ref := y.right
12
13  unfold tree(x)
14  var A: Ref := x.left
15  var B: Ref := x.right
16
17  // Perform rotation: x becomes new root
18  x.right := y
19  y.left := B
20
21  // Fold back the tree predicates
22  fold tree(y)
23  fold tree(x)
24
25  newRoot := x
26 }

```

The rotation transforms the structure from $(y, (x, A, B), C)$ to $(x, A, (y, B, C))$, where x becomes the new root.

2.2 (b) Call rotateRight with tree t

The `testRotation` method creates the tree from T21 and applies the rotation:

```

1 method testRotation() {
2   // Create tree: root=5, left=1, right=2
3   var t: Ref
4   // ... (construction as in T21.b)
5
6   var orderBefore: Seq[Int] := inorder(t)
7   var rotated: Ref := rotateRight(t)
8   var orderAfter: Seq[Int] := inorder(rotated)
9
10  assert orderBefore == orderAfter
11  assert orderBefore == Seq(1, 5, 2)
12 }

```

2.3 (c) Show memory-safety

Memory safety is guaranteed by Viper's verification system through the preconditions and postconditions of `rotateRight`:

- The precondition `requires tree(root)` ensures we have all necessary access permissions to the tree nodes
- The postcondition `ensures tree(newRoot)` proves that after rotation, all permissions are properly maintained
- The fold/unfold operations explicitly track permission transfer

Viper's verifier confirms that no invalid memory accesses occur during the rotation operation.

2.4 (d) Show height changes by at most 1

We prove that the height of the tree changes by at most 1 through mathematical case analysis. Let $h(T)$ denote the height of tree T .

Before rotation with root y and left child x :

$$h(y) = 1 + \max(h(x), h(C)) = 1 + \max(1 + \max(h(A), h(B)), h(C))$$

After rotation with new root x :

$$h(x) = 1 + \max(h(A), h(y')) = 1 + \max(h(A), 1 + \max(h(B), h(C)))$$

Case analysis:

1. If $h(A) \geq h(B)$ and $h(A) \geq h(C)$:
Before: $h = 2 + h(A)$, After: $h = 1 + h(A)$, Difference: -1
2. If $h(B) > h(A)$ and $h(B) \geq h(C)$:
Before: $h = 2 + h(B)$, After: $h = 2 + h(B)$, Difference: 0
3. If $h(C) > h(B)$ and $h(C) > h(A)$:
Before: $h = 1 + h(C)$, After: $h = 2 + h(C)$, Difference: $+1$

Therefore, $|h_{\text{before}} - h_{\text{after}}| \leq 1$. □

2.5 (e) Show in-order traversal preservation

The preservation of in-order traversal is formally verified in Viper through the postcondition:

`ensures inorder(newRoot) == old(inorder(root))`

The `inorder` function is defined recursively:

```

1 function inorder(this: Ref): Seq[Int]
2   requires tree(this)
3 {
4   unfolding tree(this) in (
5     (this.left == null ? Seq[Int]() : inorder(this.
6       left)) ++
7     Seq(this.value) ++
8     (this.right == null ? Seq[Int]() : inorder(this.
9       right))
10  )
11 }

```

Viper automatically verifies that the in-order sequence remains unchanged after rotation, confirming that rotations preserve the binary search tree property when applicable.

2.6 (f) Show in-order traversal is (1, 5, 2)

The `testRotation` method verifies this explicitly:

```

1 var orderBefore: Seq[Int] := inorder(t)
2 var rotated: Ref := rotateRight(t)
3 var orderAfter: Seq[Int] := inorder(rotated)
4
5 assert orderBefore == orderAfter
6 assert orderBefore == Seq(1, 5, 2)

```

Both assertions are verified by Viper, confirming that:

- The in-order traversal before rotation is (1, 5, 2)
- The in-order traversal after rotation is still (1, 5, 2)

3 T23: Separation Logic

3.1 (a) Prove soundness of frame rule

3.2 (b)-(h) Prove or disprove triples

4 T24: Partial Permissions

4.1 (a) Update operational semantics

4.2 (b)-(e) Prove or disprove triples