Program Verification - Assignment 7

Your Name(s)

November 15, 2025

1 T21: Warmup: Predicates and entailment

1.1 (a) Define the predicate tree(x)

We define the binary tree predicate recursively as follows:

```
predicate tree(this: Ref) {
    acc(this.value) && acc(this.left) && acc(this.right) &&
    (this.left != null ==> tree(this.left)) &&
    (this.right != null ==> tree(this.right))
}
```

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):

```
method client() {
    var t: Ref
    var leftChild: Ref
    var rightChild: Ref

    // Create left leaf with value 1
    leftChild := new(value, left, right)
    leftChild.value := 1
    leftChild.left := null
    leftChild.right := null
    fold tree(leftChild)

    // Create right leaf with value 2
    rightChild := new(value, left, right)
    rightChild.value := 2
    rightChild.left := null
    rightChild.right := null
    fold tree(rightChild)

    // Create root with value 5
    t := new(value, left, right)
    t.value := 5
    t.left := leftChild
    t.right := rightChild
    fold tree(t)

    assert tree(t)
}
```

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:

```
function left_leaf(this: Ref): Ref
    requires tree(this)
{
    unfolding tree(this) in (
        this.left == null ? this : left_leaf(this.left)
    )
}

function right_leaf(this: Ref): Ref
    requires tree(this)
{
    unfolding tree(this) in (
        this.right == null ? this : right_leaf(this.right
           )
    )
}
```

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 $\texttt{tree}(x) \models \texttt{lsegl}(x, \texttt{left\_leaf}(x)) * \texttt{lsegr}(x, \texttt{right\_leaf}(x))$ by implementing the lemma method:

```
method tree_models_lsegs(this: Ref)
    requires tree(this)
    ensures lsegl(this, old(left_leaf(this)))
    ensures lsegr(this, old(right_leaf(this)))
{
    var ll: Ref := left_leaf(this)
    var rl: Ref := right_leaf(this)

    unfold tree(this)

    if (this.left == null) {
        fold lsegl(this, this)
    } else {
        tree_models_lsegs(this.left)
        fold lsegl(this, ll)
    }

    if (this.right == null) {
        fold lsegr(this, this)
    } else {
        tree_models_lsegs(this.right)
        fold lsegr(this, rl)
    }
}
```

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:

```
method rotateRight(root: Ref) returns (newRoot: Ref)
    requires root != null
    requires tree(root)
    requires unfolding tree(root) in root.left != null
    ensures tree(newRoot)
    ensures inorder(newRoot) == old(inorder(root))
{
    unfold tree(root)
    var y: Ref := root
    var x: Ref := y.left
    var C: Ref := y.right

    unfold tree(x)
    var A: Ref := x.left
    var B: Ref := x.right

    // Perform rotation: x becomes new root
    x.right := y
    y.left := B

    // Fold back the tree predicates
    fold tree(y)
    fold tree(x)

    newRoot := x
}
```

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:

```
method testRotation() {
    // Create tree: root=5, left=1, right=2
    var t: Ref
    // ... (construction as in T21.b)

    var orderBefore: Seq[Int] := inorder(t)
    var rotated: Ref := rotateRight(t)
    var orderAfter: Seq[Int] := inorder(rotated)

    assert orderBefore == orderAfter
    assert orderBefore == Seq(1, 5, 2)
}
```

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:

```
function inorder(this: Ref): Seq[Int]
    requires tree(this)
{
    unfolding tree(this) in (
        (this.left == null ? Seq[Int]() : inorder(this.left)) ++
        Seq(this.value) ++
        (this.right == null ? Seq[Int]() : inorder(this.right))
    )
}
```

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

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

The `testRotation` method verifies this explicitly:

```
var orderBefore: Seq[Int] := inorder(t)
var rotated: Ref := rotateRight(t)
var orderAfter: Seq[Int] := inorder(rotated)

assert orderBefore == orderAfter
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

The frame rule is:

$$\frac{\{p\}\ C\ \{q\}}{\{p * r\}\ C\ \{q * r\}}$$

where no variable free in r is modified by C.
We prove soundness with respect to the operational semantics for each command.

Soundness: If $(s, h) \models p * r$ and $\langle C, (s, h)\rangle \Rightarrow \langle \text{done}, (s', h')\rangle$ and $\vdash \{p\}\ C\ \{q\}$, then $(s', h') \models q * r$.

Proof for allocation ($x :=$ cons e):
Assume $(s, h) \models p * r$. Then there exist disjoint heaps h_p and h_r such that $h = h_p \cup h_r$, $(s, h_p) \models p$, and $(s, h_r) \models r$.
By the allocation rule, $\{emp\}$ $x :=$ cons e $\{x \mapsto e\}$. Since $(s, h_p) \models emp$, we have $h_p = \emptyset$.
After allocation: $\langle x := \text{cons } e, (s, h)\rangle \Rightarrow \langle \text{done}, (s[x \leftarrow l], h[l \leftarrow [\![e]\!]_s])\rangle$ where $l \notin \text{dom}(h)$.
Since $h = h_r$ (as $h_p = \emptyset$), we have $l \notin \text{dom}(h_r)$. Let $h'_p = \{l \mapsto [\![e]\!]_s\}$ and $h'_r = h_r$.
Then $(s[x \leftarrow l], h'_p) \models x \mapsto e$ and $(s[x \leftarrow l], h'_r) \models r$ (since x is not free in r and $h'_r = h_r$).
Since $\text{dom}(h'_p) \cap \text{dom}(h'_r) = \{l\} \cap \text{dom}(h_r) = \emptyset$, we have $(s[x \leftarrow l], h'_p \cup h'_r) \models x \mapsto e * r$. □

Proof for deallocation (dispose e):
Assume $(s, h) \models p * r$ with $h = h_p \cup h_r$, $(s, h_p) \models p$, $(s, h_r) \models r$, and $\text{dom}(h_p) \cap \text{dom}(h_r) = \emptyset$.
By the deallocation rule, $\{e \mapsto -\}$ dispose e $\{emp\}$. So $(s, h_p) \models e \mapsto -$, meaning $h_p = \{[\![e]\!]_s\}$.
After deallocation: $\langle \text{dispose } e, (s, h)\rangle \Rightarrow \langle \text{done}, (s, h - [\![e]\!]_s)\rangle$ where $[\![e]\!]_s \in \text{dom}(h)$.
Since $h_p = \{[\![e]\!]_s\}$, we have $h - [\![e]\!]_s = h_r$. Thus $(s, h_r) \models emp * r = r$. □

Proof for mutation ($[e] := e'$):
Assume $(s, h) \models p * r$ with $h = h_p \cup h_r$, $(s, h_p) \models p$, $(s, h_r) \models r$.
By the mutation rule, $\{e \mapsto -\}$ $[e] := e'$ $\{e \mapsto e'\}$. So $(s, h_p) \models e \mapsto -$, meaning $h_p = \{[\![e]\!]_s \mapsto v\}$ for some v.
After mutation: $\langle [e] := e', (s, h)\rangle \Rightarrow \langle \text{done}, (s, h[[\![e]\!]_s \leftarrow [\![e']\!]_s])\rangle$.
Let $h'_p = \{[\![e]\!]_s \mapsto [\![e']\!]_s\}$ and $h'_r = h_r$. Then $(s, h'_p) \models e \mapsto e'$ and $(s, h'_r) \models r$.
Since $[\![e]\!]_s \in \text{dom}(h_p)$ and $\text{dom}(h_p) \cap \text{dom}(h_r) = \emptyset$, we have $[\![e]\!]_s \notin \text{dom}(h_r)$, so h'_p and h'_r are still disjoint.
Therefore $(s, h'_p \cup h'_r) \models e \mapsto e' * r$. □

Proof for lookup ($x := [e]$):
Assume $(s, h) \models p * r$ with $h = h_p \cup h_r$, $(s, h_p) \models p$, $(s, h_r) \models r$.
By the lookup rule, $\{e \mapsto v\}$ $x := [e]$ $\{\exists x'.e[x := x'] \mapsto v \wedge x = v\}$. So $(s, h_p) \models e \mapsto v$, meaning $h_p = \{[\![e]\!]_s \mapsto v\}$.
After lookup: $\langle x := [e], (s, h)\rangle \Rightarrow$
$\langle \text{done}, (s[x \leftarrow h([\![e]\!]_s)], h)\rangle$
$= \langle \text{done}, (s[x \leftarrow v], h)\rangle$.
Since x is not free in r (frame rule condition) and the heap is unchanged, $(s[x \leftarrow v], h_r) \models r$.
Also $(s[x \leftarrow v], h_p) \models \exists x'.e[x := x'] \mapsto v \wedge x = v$ (taking $x' = s(x)$).
Therefore $(s[x \leftarrow v], h) \models (\exists x'.e[x := x'] \mapsto v \wedge x = v) * r$. □

3.2 (b) Prove or disprove: $\{\{x \mapsto 0 \wedge y \mapsto 0\}\}\ [x] := 5\ \{\{x \mapsto 5 \wedge y \mapsto 0\}\}$

Disproven. The precondition $x \mapsto 0 \wedge y \mapsto 0$ is not a valid separation logic assertion.

In separation logic, $x \mapsto 0$ means the heap contains exactly one location (the value of x) mapping to 0. Similarly, $y \mapsto 0$ means the heap contains exactly one location (the value of y) mapping to 0.

The conjunction $\wedge$ requires both to hold on the *same* heap, which is impossible unless $x = y$. If $x \neq y$, no heap can satisfy this precondition.

The correct formulation would use separating conjunction: $\{\{x \mapsto 0 * y \mapsto 0\}\}\ [x] := 5\ \{\{x \mapsto 5 * y \mapsto 0\}\}$, which is valid.

3.3 (c) Prove or disprove: $\{\{e \mapsto v * e' \mapsto v\}\}$ dispose $e; [e'] := 5\ \{\{e' \mapsto 5\}\}$

Proven. We use the deallocation rule, mutation rule, and sequence rule.

$$
\begin{array}{l}
\{e \mapsto v * e' \mapsto v\} \\
\quad \text{dispose } e \\
\{emp * e' \mapsto v\} \quad \text{(by deallocation rule and frame rule)} \\
= \{e' \mapsto v\} \\
\quad [e'] := 5 \\
\{e' \mapsto 5\} \quad \text{(by mutation rule)}
\end{array}
$$

Therefore, by the sequence rule, $\{e \mapsto v * e' \mapsto v\}$ dispose $e; [e'] := 5\ \{e' \mapsto 5\}$. □

3.4 (d) Prove or disprove: $\{\{true\}\}\ [x] := 1\ \{\{true\}\}$

Disproven. The triple is invalid because the precondition *true* does not guarantee that location x is allocated in the heap.

By the operational semantics, $\langle [x] := 1, (s, h)\rangle \Rightarrow \langle \text{abort}, (s, h)\rangle$ if $[\![x]\!]_s \notin \text{dom}(h)$.

There exist states (s, h) where $(s, h) \models true$ but $[\![x]\!]_s \notin \text{dom}(h)$ (e.g., $h = \emptyset$). In such states, the command aborts, violating safety.

3.5 (e) Prove or disprove: $\{\{x \mapsto 3 * y \mapsto 9\}\}\ [x] := 2\ \{\{y \mapsto 9\}\}$

Disproven. While the triple is semantically valid (the mutation doesn't affect y), it cannot be derived using the standard separation logic proof rules.

The mutation rule gives us: $\{x \mapsto 3\}\ [x] := 2\ \{x \mapsto 2\}$.

Applying the frame rule with $r = y \mapsto 9$:

$$\{x \mapsto 3 * y \mapsto 9\}\ [x] := 2\ \{x \mapsto 2 * y \mapsto 9\}$$

The postcondition $\{x \mapsto 2 * y \mapsto 9\}$ is strictly stronger than $\{y \mapsto 9\}$ alone. To derive $\{y \mapsto 9\}$, we would need to "forget" the assertion $x \mapsto 2$, which requires the consequence rule with:

$$x \mapsto 2 * y \mapsto 9 \Rightarrow y \mapsto 9$$

However, this entailment is *not valid* in separation logic! The left side asserts ownership of both x and y, while the right side only asserts ownership of y. Separation logic does not allow "weakening" by discarding heap ownership.

Therefore, the triple cannot be proven.

3.6 (f) Prove or disprove: $\{\{x \mapsto 3*y \mapsto 9\}\}\ [x] := 2\ \{\{true*y \mapsto 9\}\}$

Proven. Using the mutation rule and frame rule:

$$\begin{aligned}
&\{x \mapsto 3 * y \mapsto 9\}\\
&\quad [x] := 2\\
&\{x \mapsto 2 * y \mapsto 9\} \quad \text{(by mutation + frame)}
\end{aligned}$$

Now, $x \mapsto 2 \Rightarrow true$ (any assertion implies $true$), so by the consequence rule:

$$\{x \mapsto 2 * y \mapsto 9\} \Rightarrow \{true * y \mapsto 9\}$$

Therefore, $\{x \mapsto 3 * y \mapsto 9\}\ [x] := 2\ \{true * y \mapsto 9\}$. □

3.7 (g) Prove or disprove: $\{\{x \mapsto -\}\}\ f; [x] := 3\ \{\{x \mapsto 3\}\}$, given $\vdash \{\{emp\}\}\ f\ \{\{emp\}\}$

Disproven. By the frame rule with $r = x \mapsto -$ and the given specification of f:

$$\{emp * x \mapsto -\}\ f\ \{emp * x \mapsto -\}$$

$$\{x \mapsto -\}\ f\ \{x \mapsto -\}$$

Then by sequence and mutation:

$$\{x \mapsto -\}\ f; [x] := 3\ \{x \mapsto 3\}$$

Wait - this actually **proves** the triple! The frame rule condition is satisfied since x is not free in emp (there are no free variables), so we can frame $x \mapsto -$ around the execution of f. □

3.8 (h) Prove or disprove: $\{\{x \mapsto -\}\}\ [x] := 3; g\ \{\{x \mapsto 3\}\}$, given $\vdash \{\{x \mapsto -\}\}\ g\ \{\{x \mapsto -\}\}$

Disproven. By mutation: $\{x \mapsto -\}\ [x] := 3\ \{x \mapsto 3\}$.

Then we would need: $\{x \mapsto 3\}\ g\ \{x \mapsto 3\}$.

However, we are only given $\{x \mapsto -\}\ g\ \{x \mapsto -\}$, which says g preserves *some* value at x, not necessarily the value 3.

The specification $\{x \mapsto -\}\ g\ \{x \mapsto -\}$ means: if x points to any value v before g, then after g, x points to some (possibly different) value v'.

Since the postcondition of the mutation is $x \mapsto 3$ (a specific value), but the precondition of g requires $x \mapsto -$ (any value), and the postcondition of g only guarantees $x \mapsto -$ (any value, not necessarily 3), we cannot conclude $x \mapsto 3$ after executing g.

Therefore, the triple cannot be proven.

4 T24: Partial Permissions

4.1 (a) Update operational semantics

We extend the operational semantics to support fractional permissions, where permissions are represented as fractions with shares of 10. Each location can have permissions split into fractions summing to at most 1 (or 10/10).

Extended heap model: A heap h now maps locations to pairs (v, π) where v is the value and π is the permission fraction. We write $h(l) = (v, \pi)$ to indicate location l has value v with permission fraction π.

Heap splitting: For fractional permissions, we can split a heap h into disjoint heaps h_1 and h_2 such that:

- If $l \in \text{dom}(h_1) \cap \text{dom}(h_2)$, then $h_1(l) = (v, \pi_1)$ and $h_2(l) = (v, \pi_2)$ where $h(l) = (v, \pi_1 + \pi_2)$ and the same value v is in both heaps
- If $l \in \text{dom}(h_1) \setminus \text{dom}(h_2)$, then $h(l) = h_1(l)$
- If $l \in \text{dom}(h_2) \setminus \text{dom}(h_1)$, then $h(l) = h_2(l)$

Updated operational semantics:
Allocation: $x := \text{cons } e$

$$\langle x := \text{cons } e, (s, h)\rangle \Rightarrow \langle \text{done}, (s[x \leftarrow l], h[l \leftarrow (\llbracket e \rrbracket_s, 10/10)])\rangle$$

where $l \notin \text{dom}(h)$. Allocation creates a new location with full permission (10/10).

Deallocation: dispose e

$$\langle \text{dispose } e, (s, h)\rangle \Rightarrow \langle \text{done}, (s, h \setminus \{\llbracket e \rrbracket_s\})\rangle$$

if $[\![e]\!]_s \in \mathrm{dom}(h)$ and $h([\![e]\!]_s) = (v, 10/10)$ for some v (full permission required).

Otherwise, $\langle \text{dispose } e, (s,h)\rangle \Rightarrow \langle \text{abort}, (s,h)\rangle$ if permission is not 10/10.

Mutation: $[e] := e'$

$$\langle [e] := e', (s,h)\rangle \Rightarrow \langle \text{done}, (s, h[[\![e]\!]_s \leftarrow ([\![e']\!]_s, 10/10)])\rangle$$

if $[\![e]\!]_s \in \mathrm{dom}(h)$ and $h([\![e]\!]_s) = (v, 10/10)$ for some v (full permission required for mutation).

Otherwise, $\langle [e] := e', (s,h)\rangle \Rightarrow \langle \text{abort}, (s,h)\rangle$ if permission is not 10/10.

Lookup: $x := [e]$

$$\langle x := [e], (s,h)\rangle \Rightarrow \langle \text{done}, (s[x \leftarrow v], h)\rangle$$

if $[\![e]\!]_s \in \mathrm{dom}(h)$ and $h([\![e]\!]_s) = (v, \pi)$ for some $\pi > 0$ (any positive permission allows reading).

Otherwise, $\langle x := [e], (s,h)\rangle \Rightarrow \langle \text{abort}, (s,h)\rangle$ if $[\![e]\!]_s \notin \mathrm{dom}(h)$.

Key properties:

- Reading requires any positive fraction $\pi > 0$
- Writing and deallocation require full permission $\pi = 10/10$
- Permissions are preserved during lookup (heap unchanged)
- Mutation updates value but maintains full permission

4.2 (b) Prove: $\{\{x \ 10/7 \mapsto v\}\}\ [x] := 5\ \{\{x \ 10/7 \mapsto 5\}\}$

Disproven. The triple cannot be proven because mutation requires full permission (10/10), but we only have fractional permission (10/7).

By the updated mutation rule, $[x] := 5$ requires the precondition to assert full permission: $x \ 10/10 \mapsto v$.

Since $10/7 \neq 10/10$ and $10/7 < 10/10$ ($10/7 \approx 1.43$, which is impossible for permissions that sum to at most 1), this appears to be a typo. If we interpret this as having permission 7/10 (less than full), then:

The operation $\langle [x] := 5, (s,h)\rangle$ will transition to $\langle \text{abort}, (s,h)\rangle$ because $h(x) = (v, 7/10)$ and $7/10 \neq 10/10$.

Therefore, the triple is **invalid** because the mutation cannot execute safely with only fractional permission.

4.3 (c) Prove: $\{\{x \ 10/7 \mapsto v\}\}$ `dispose5`(x); $y := [x]\ \{\{y = v\}\}$

Disproven. This triple is invalid for two reasons:

1. **Insufficient permission for disposal:** The operation `dispose5`(x) (assuming it means disposing location x) requires full permission 10/10, but we only have 10/7 (interpreted as 7/10 if correcting the notation).

2. **Use-after-free:** Even if disposal were possible, the sequence attempts to read from x after disposing it:
   - After `dispose5`(x), location x is removed from the heap
   - The subsequent lookup $y := [x]$ attempts to read from a deallocated location
   - This results in $\langle y := [x], (s, h)\rangle \Rightarrow \langle \text{abort}, (s, h)\rangle$

Therefore, the command sequence will abort and cannot establish any postcondition.

4.4 (d) Prove: $\{\{x\ 10/7 \mapsto v\}\}\ [x] := 3;\ f\ \{\{x\ 10/7 \mapsto 3\}\}$, given $\models \{\{x\ 1/7 \mapsto -\}\}\ f\ \{\{x\ 1/7 \mapsto -\}\}$

Disproven. Similar to part (b), the mutation $[x] := 3$ requires full permission 10/10.

Assuming the notation means we have permission 7/10 (less than full), the mutation rule requires:

$$\{x\ 10/10 \mapsto v\}\ [x] := 3\ \{x\ 10/10 \mapsto 3\}$$

Since we only have $x\ 7/10 \mapsto v$, we cannot perform the mutation. The operational semantics gives:

$$\langle [x] := 3, (s, h)\rangle \Rightarrow \langle \text{abort}, (s, h)\rangle$$

Therefore, the triple cannot be proven.

Alternative interpretation: If the notation 10/7 means we have *more* than full permission (which would be invalid in standard fractional permissions), this would also be impossible as total permissions cannot exceed 10/10.

4.5 (e) Disprove: $\{\{x\ 10/7 \mapsto v\}\}\ g;\ [x] := 3\ \{\{true\}\}$, given $\models \{\{x\ 1/7 \mapsto -\}\}\ g\ \{\{true\}\}$

Disproven. This triple fails for multiple reasons:

1. **Permission loss after g:** Given $\{x\ 1/7 \mapsto -\}\ g\ \{true\}$, the function g consumes the permission it receives and establishes only *true* (no heap assertion).

   If we split $x\ 10/7 \mapsto v$ as $x\ 1/7 \mapsto v * x\ 9/7 \mapsto v$ (again assuming notation correction), and frame g with the remaining permission:

   By the frame rule: $\{x\ 1/7 \mapsto v * x\ 9/7 \mapsto v\}\ g\ \{true * x\ 9/7 \mapsto v\}$

   But this gives us only $x\ 9/7 \mapsto v$ after g (interpreting as some fraction less than full).

2. **Insufficient permission for mutation:** After executing g, we do not have full permission 10/10 to perform $[x] := 3$.

3. **Given specification consumes permission:** The postcondition *true* of g indicates that g may deallocate or lose the permission to x, leaving no guarantee that x is still accessible.

Therefore, the triple cannot be proven because we cannot safely execute the mutation after g.