

Program Verification – Assignment 7

- Submit your solutions online in groups of 1-3 before the deadline.
- The tasks are numbered consecutively across all assignments. If you have questions, you can thus refer to the task number.
- You can use the concepts and notation from the course material without explanation. Any other definitions or notations must be defined before they are used. Similarly, theorems or mathematical properties that are neither introduced in the course nor well-known from typical undergraduate courses, for example discrete math or logic, need to be introduced and proven first. When in doubt about using a property you know from another course, ask us first.

T21: Warmup: Predicates and entailment (20 Points)

Consider the following template:

```
field value: Int; field left: Ref; field right: Ref

predicate lseg1(this: Ref, last: Ref) {
    this != last ==> acc(this.left) &&
    lseg1(this.left, last)
}

predicate lsegr(this: Ref, last: Ref) {
    this != last ==> acc(this.right) &&
    lsegr(this.right, last)
}

// TODO: implement
predicate tree(this: Ref)
function left_leaf(this: Ref): Ref
function right_leaf(this: Ref): Ref
method tree_models_lsegs(this: Ref)
    requires tree(this)
    ensures lseg1(this, old(left_leaf(this)))
    ensures lsegr(this, old(right_leaf(this)))

method client() {
    var t: Ref
    // TODO
    assert tree(t)
}
```

-
- (a) Define the predicate `tree(x)` such that it models heap-based binary trees with root x .
 - (b) Implement the method `client`, which creates a tree t with value 5 at the root and two children: a left leaf with value 1 and a right leaf with value 2.
 - (c) Implement the functions `left_leaf` and `right_leaf` such that they return the left-most and right-most leaf in the tree, respectively.
 - (d) Show that $\text{tree}(x) \vdash \text{lseg1}(x, \text{left_leaf}(x)) * \text{lseg1}(x, \text{right_leaf}(x))$ by implementing the lemma method `tree_models_lsegs`.

T22: Tree Rotations (50 Points)

Tree rotations are local transformations on binary trees that preserve the in-order structure of the tree while changing its shape¹. In this task, we study rotations for the binary tree predicate introduced in the previous task.

- (a) Implement a `rotateRight` method that, given a reference to the root of a tree, returns the pointer to the root of the right-rotated tree.
- (b) Call the method `rotateRight` with the tree `t` constructed in the previous task.
- (c) Show memory-safety of `rotateRight`.
- (d) Show that `rotateRight` changes the height of a given tree by at most 1.
- (e) Show that rotations performed by `rotateRight` do not change the order of an in-order tree traversal.
- (f) Show that an in-order traversal of the tree `t` from T21.b is (1, 5, 2) both before and after rotating it.

Hints:

- All proofs must be performed with Viper.
- You may want to implement some additional abstraction functions.

¹https://en.wikipedia.org/wiki/Tree_rotation

T23: Separation Logic (40 Points)

Separation logic is a popular extension of Floyd-Hoare logic to enable local reasoning about heap-manipulating programs. Nowadays, there exist many flavors and extensions of separation logics, including Viper's permission system (also known as implicit dynamic frames).² To sharpen our intuition on reasoning about heap-manipulating programs, we will study the original version of separation logic (SL for short) in this task. SL is less powerful than Viper's permission system, but also simpler to formalize. Furthermore, SL is often referenced in the verification literature, i.e. knowing SL is helpful when reading up on verification techniques. We first introduce the operational semantics of simple heap-manipulating commands. After that, we introduce corresponding Hoare-style proof rules for those commands. Your task will be to use those definitions to get a better understanding of heap-manipulating commands and framing.

Commands. The programs that we will consider are allowed to use the following four commands, other than the usual commands from our toy language:

$$C ::= x := \mathbf{cons} \ e \mid \mathbf{dispose} \ e \mid [e] := e' \mid x := [e] \mid \dots$$

Here, e are integer-valued terms like in the lectures. Their values can be interpreted as memory locations. The notation $[e]$ refers to the value at address e , similarly to pointer dereferences $*e$ in the C programming language.

We define operational semantics of the form $\langle C, (s, h) \rangle \Rightarrow \langle C, (s, h) \rangle$, where h is a mapping from a set of allocated *locations* to values, and s is a mapping from variables to values and locations. The command $x := \mathbf{cons} \ e$ allocates a new cell in the heap, initializing it to the evaluation of e and assigns its location to x . Formally,

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

where $l \notin \mathbf{dom}(h)$ (l is a fresh location), and where $\llbracket e \rrbracket_s$ is the evaluation of e under the stack s . The command $\mathbf{dispose} \ e$ deallocates the location obtained from the evaluation of e . If the location is not present in the heap, the program aborts:

$$\begin{array}{ll} \langle \mathbf{dispose} \ e, (s, h) \rangle \Rightarrow \langle \mathbf{done}, (s, h - \llbracket e \rrbracket_s) \rangle & \text{if } \llbracket e \rrbracket_s \in \mathbf{dom}(h) \\ \langle \mathbf{dispose} \ e, (s, h) \rangle \Rightarrow \langle \mathbf{abort}, (s, h) \rangle & \text{if } \llbracket e \rrbracket_s \notin \mathbf{dom}(h) \end{array}$$

²See <https://cacm.acm.org/research/separation-logic/> for an overview article.

Similarly, the command $[e] := e'$ assigns to the location obtained from the evaluation of e , the value obtained from the evaluation of e' . If the location $\llbracket e \rrbracket_s$ is not in the heap, the program aborts:

$$\begin{aligned} \langle [e] := e', (s, h) \rangle &\Rightarrow \langle \text{done}, (s, h[\llbracket e \rrbracket_s \leftarrow \llbracket e' \rrbracket_s]) \rangle & \text{if } \llbracket e \rrbracket_s \in \text{dom}(h) \\ \langle [e] := e', (s, h) \rangle &\Rightarrow \langle \text{abort}, (s, h) \rangle & \text{if } \llbracket e \rrbracket_s \notin \text{dom}(h) \end{aligned}$$

Lastly, the command $x := [e]$ assigns to the variable x the value in the location obtained from the evaluation of e . As before, if the location $\llbracket e \rrbracket_s$ is not in the heap, the program aborts:

$$\begin{aligned} \langle x := [e], (s, h) \rangle &\Rightarrow \langle \text{done}, (s[x \leftarrow h(\llbracket e \rrbracket_s)], h) \rangle & \text{if } \llbracket e \rrbracket_s \in \text{dom}(h) \\ \langle x := [e], (s, h) \rangle &\Rightarrow \langle \text{abort}, (s, h) \rangle & \text{if } \llbracket e \rrbracket_s \notin \text{dom}(h) \end{aligned}$$

Predicates. We consider, in addition to the usual pure predicates, three other predicates: **emp**, asserting that the heap is empty; $e \mapsto e'$, asserting that the heap is comprised only of one location (given by the evaluation of e) and contains the evaluation of e' ; and the separation conjunction $p * q$, which asserts that p and q are true in two disjoint partitions of the heap. Formally,

$$\begin{aligned} s, h \models \text{emp} &\iff \text{dom}(h) = \emptyset \\ s, h \models e \mapsto e' &\iff \text{dom}(h) = \{\llbracket e \rrbracket_s\} \text{ and } h(\llbracket e \rrbracket_s) = \llbracket e' \rrbracket_s \\ s, h \models e \mapsto - &\iff \text{dom}(h) = \{\llbracket e \rrbracket_s\} \\ s, h \models p * q &\iff \left(\begin{array}{l} \exists h_0, h_1. \text{dom}(h_0) \cap \text{dom}(h_1) = \emptyset \text{ and} \\ h_0 \cup h_1 = h \text{ and } s, h_0 \models p \text{ and } s, h_1 \models q \end{array} \right) \end{aligned}$$

Here, we write $e \mapsto -$ if the value that e maps to is not relevant (but e is indeed allocated in the heap).

Proof rules. In order to prove properties about heap-manipulating programs, we consider the following Hoare-style proof rules.

- Allocation rule:

$$\frac{}{\{\{\text{emp}\}\} x := \text{cons } e \{\{x \mapsto e\}\}} \text{[A]}$$

where x is not free in e .

- Deallocation rule:

$$\frac{}{\{\{e \mapsto -\}\} \text{ dispose } e \{\{\text{emp}\}\}} \text{[D]}$$

- Mutation rule:

$$\frac{}{\{\{e \mapsto -\}\} [e] := e' \{\{e \mapsto e'\}\}} [M]$$

- Lookup rule:

$$\frac{}{\{\{e \mapsto v\}\} x := [e] \{\{\exists x'. [e[x := x'] \mapsto v] \wedge x := v\}\}} [L]$$

- Frame rule:

$$\frac{\{\{p\}\} C \{\{q\}\}}{\{\{p * r\}\} C \{\{q * r\}\}} [F]$$

where no variable free in r is modified by C .

- Furthermore, the sequence and consequence rule are as in the lectures.

Tasks.

- (a) Prove that the frame rule is sound with respect to the four commands of allocation, deallocation, assignment and lookup.

Furthermore, prove the following triples using the Hoare-style proof rules, or disprove them by formally reasoning about the operational semantics.

- (b) $\{\{x \mapsto 0 \wedge y \mapsto 0\}\} [x] := 5 \{\{x \mapsto 5 \wedge y \mapsto 0\}\}$
- (c) $\{\{e \mapsto v * e' \mapsto v\}\} \text{ dispose } e; [e'] := 5 \{\{e' \mapsto 5\}\}$
- (d) $\{\{\text{true}\}\} [x] := 1 \{\{\text{true}\}\}$
- (e) $\{\{x \mapsto 3 * y \mapsto 9\}\} [x] := 2 \{\{y \mapsto 9\}\}$
- (f) $\{\{x \mapsto 3 * y \mapsto 9\}\} [x] := 2 \{\{\text{true} * y \mapsto 9\}\}$
- (g) $\{\{x \mapsto -\}\} f; [x] := 3 \{\{x \mapsto 3\}\}, \text{ given that } \vdash \{\{\text{emp}\}\} f \{\{\text{emp}\}\}$
- (h) $\{\{x \mapsto -\}\} [x] := 3; g \{\{x \mapsto 3\}\}, \text{ given that } \vdash \{\{x \mapsto -\}\} g \{\{x \mapsto -\}\}$

T24: Partial Permissions

(40 Points)

In this task, you will extend the classical heap model described in the previous task to account for *partial* ownership of heap cells, represented as shares of a fixed total. Concretely, each heap cell's ownership is divided into 10 equal parts. Allocating a cell grants ownership of all 10 parts, representing full ownership over the heap cell. Ownership can then be *partially* released by deallocating any subset of these parts. A program, may read from a heap cell whenever it owns at least one part, but writing requires full ownership.

Formally, we consider programs of the form

$$C ::= x := \mathbf{cons} \ e \mid \mathbf{dispose}_n \ e \mid [e] := e' \mid \dots$$

and assertions of the form

$$p, q ::= \mathbf{emp} \mid e \xrightarrow{n} e' \mid p * q \mid \dots$$

where $1 \leq n \leq 10$. We discuss a couple examples. The programs

$$x := \mathbf{cons} \ 15; [x] := 6 \quad \text{and} \quad x := \mathbf{cons} \ 15; \mathbf{dispose}_3 \ x; y := [x]$$

should successfully execute: the first program has permission over all 10 shares of the cell pointed by x at the time of mutation, while the second program has at least one (7) shares at the time of lookup. However, the programs

$$x := \mathbf{cons} \ 15; \mathbf{dispose}_3 \ x; [x] := 6 \quad \text{and} \quad x := \mathbf{cons} \ 15; \mathbf{dispose}_3 \ x; \mathbf{dispose}_7 \ x; y := [x]$$

should abort. Your task is to:

- (a) *Update the formal operational semantics of heap-manipulating programs, as well as the semantics of predicates, to allow for partial ownership.*

Then, use your operational semantics to prove the following triples:

- (b) $\{\{x \xrightarrow{10} v\}\} [x] := 5 \ \{\{x \xrightarrow{10} 5\}\}$
- (c) $\{\{x \xrightarrow{10} v\}\} \mathbf{dispose}_5(x); y := [x] \ \{\{y = v\}\}$
- (d) $\{\{x \xrightarrow{10} v\}\} [x] := 3; f \ \{\{x \xrightarrow{10} 3\}\}, \text{ where } \models \{\{x \xrightarrow{1} -\}\} f \ \{\{x \xrightarrow{1} -\}\}$

and disprove the following triple:

- (e) $\{\{x \xrightarrow{10} v\}\} g; [x] := 3; \ \{\{\mathbf{true}\}\}, \text{ where } \models \{\{x \xrightarrow{1} -\}\} g \ \{\{\mathbf{true}\}\}.$