Lab Session 7
More Exercises on Matrices & Verification of ADTs

Software Verification

Nova School of Science and Technology
Mário Pereira   mjp.pereira@fct.unl.pt

Version of October 20, 2025

This lab session is divided into two parts: first, some more exercises on matrices; second, implementing and proving correct a set ADT.

The goal of the second part is to simulate a small project where you have to implement a set ADT, using a Binary Search Tree data structure. This allows you to practice the design of ADTs, while at the same time practicing with functional programming and lemmas in Why3.

While working on the different exercises, try to change the specification of predicates, functions, and lemmas. This is crucial to understand why the given pre- and post-conditions are necessary to prove this ADT correct. After successfully completing this lab session, you will have a verified Why3 implementation of the interface given in Appendix A.

1 More Exercises on Matrices

Exercise 1. Define a Why3 predicate that states whether two given matrices `m1` and `m2` are *equal*. Your predicate shall have the following signature:

```
predicate equal_matrix (m1 m2: matrix int)
```

and must obey the following rules: matrices `m1` and `m2` are equal if:

1. the number of rows in `m1` is equal to the number of rows in `m2`;
2. the number of columns in `m1` is equal to the number of columns in `m2`;
3. for every indices `i` and `j` within the bounds of the matrices, then `m2[i][j]` is equal to `m2[i][j]`.

□

Exercise 2. Consider the specification of the following `sum_matrix` function, as defined in last lab session:

```
let sum_matrix (m1 m2: matrix int) : matrix int
  requires { m1.rows = m2.rows }
  requires { m1.columns = m2.columns }
  ensures  { result.rows = m1.rows }
  ensures  { result.columns = m1.columns }
```

```
    ensures  { forall i j. 0 <= i < result.rows -> 0 <= j < result.columns ->
                 get result i j = get m1 i j + get m2 i j }
```

Complete the definition of function `sum_assoc` that states the sum of matrices is associative. This function shall have the following signature:

```
    let sum_assoc (m1 m2 m3: matrix int) : (mr1: matrix int, mr2: matrix int)
    requires { m1.rows = m2.rows = m3.rows }
    requires { m1.columns = m2.columns = m3.columns }
    ensures  { equal_matrix mr1 mr2 }
```

The body of `sum_assoc` must first compute $M_1 + (M_2 + M_3)$ and then $(M_1 + M_2) + M_3$. Finally, it should return the two resulting matrices, so to prove the post-condition. □

Exercise 3. Specify and implement a function that *multiplies a matrix by a scalar value.* Your function shall have the following signature:

```
  let scalar_mul (v: int) (m: matrix int) : matrix int
```

□

Exercise 4. Specify and implement a function that *multiplies a matrix by a vector.* Your function shall have the following signature:

```
  let vector_mul (a: array int) (m: matrix int) : matrix int
```

□

2 Binary Search Trees

Consider the following Why3 definition for polymorphic Binary Trees:

```
module BST : Set
  use int.Int
  use set.Fset

  type tree 'a = Leaf | Node (tree 'a) 'a (tree 'a)
```

Exercise 5. On top of the `tree 'a` datatype, give a Why3 definition for a pure function that computes the number of occurrences for some value `x` in a tree `t`. Your function shall have the following signature:

```
  function occ (x: 'a) (t: tree 'a) : int
```

and must obey the following mathematical definition:

$$\mathtt{occ}(x, t) = \begin{cases} 0 & \text{if } t \equiv \mathtt{Leaf} \\ & \\ (\text{if } x = y \text{ then } 1 \text{ else } 0) + \mathtt{occ}\ x\ l + \mathtt{occ}\ x\ r & \text{if } t \equiv \mathtt{Node}\ l\ y\ r \end{cases}$$

Note that, since we define function `occ` as logical `function`, we are allowed to compare elements of type `'a` directly using operator `(=)`. □

Exercise 6. State and prove the following auxiliary lemma:

```
let rec lemma occ_nonneg (x: 'a) (t: tree 'a) : unit
  ensures { occ x t >= 0 }
  variant { t }
```

□

Exercise 7. Using the `occ` function from the previous question, give a Why3 definition for the relation *"a given element* `x` *belongs to a given tree* `t`*"*. Your predicate shall have the following signature:

```
predicate mem (x: 'a) (t: tree 'a)
```

□

Exercise 8. Using the `mem` predicate from the previous question, give a Why3 definition for the relation *"a given Binary Tree* `t` *is a Binary Search Tree (BST)"*. Your predicate shall have the following signature:

```
predicate bin_search_tree (t: tree int)
```

and must obey the following two rules:

$$\frac{}{\texttt{bin\_search\_tree}\ \mathrm{Leaf}}\ \text{BST\_LEAF}$$

$$\frac{\begin{array}{cc}\texttt{bin\_search\_tree l} & \texttt{bin\_search\_tree r} \\ \forall y, \texttt{mem y l} \implies y < x & \forall y, \texttt{mem y r} \implies y > x\end{array}}{\texttt{bin\_search\_tree}\ (\texttt{Nodelxr})}\ \text{BST\_NODE}$$

□

3 Set ADT – Type Invariant

Exercise 9. Having a definition of BSTs in Why3, we are now in position to build our Set ADT. Consider the following, as the beginning of the `set` datatype definition:

```
type t = {
        mutable data: tree int;
  ghost mutable view: fset int;
}
```

The field `data` stores the underlying data structure, in this case a Binary Tree of integer values. The field `view` is the abstract state, in this case a mathematical set of integer values.

Complete the definition for the type invariant, and the corresponding witness, according to the following rules:

1. the tree stored in `data` is a BST
2. every element that belongs to the tree `data` also belongs to the set `view`.

□

4 Set ADT – Operations

4.1 Creation

Exercise 10. Complete the definition of the `create` function for the `BST` module. This function must respect the following specification:

```
let create () : t
  ensures { result.view == empty }
```

Please, note the use of the `(==)` operator instead of `(=)`. The former stands for the *extensional equality* of sets: two sets are *equal* if both have the *same cardinality* and *contain the same elements*. The latter is the classical *polymorphic equality*. In practice, `(==)` can be seen as a *specific implementation* of `(=)` for sets. We use it in the implementation of the ADT since automated solvers deal much better with `(==)` than with `(=)`. Finally, the Why3 standard library of finite sets features the following lemma:

```
lemma extensionality:
  forall s1 s2: fset 'a. s1 == s2 -> s1 = s2
```

This is crucial to prove that a post-condition using `(==)` refines a post-condition using `(=)`. □

4.2 Insert an Element

Exercise 11. Give an implementation for a recursive function that inserts an element `x` in a BST `t`. Your function must respect the following specification:

```
let rec insert_aux (x: int) (t: tree int) : tree int
  requires { bin_search_tree t }
  ensures  { forall y. y <> x -> occ y t = occ y result }
  ensures  { mem x result }
  ensures  { bin_search_tree result }
  variant  { t }
```

Note that our representation of BSTs does not allow for multiple occurrences of the same element in the tree. Hence, when the element already belongs to the tree, `insert_aux` behaves as the identity function. □

Exercise 12. Using the `insert_aux` function from the previous question, complete the definition of function `insert` of the `BST` module, which inserts an element `x` in the ADT. Your function function must respect the following specification:

```
let insert (x: int) (t: t) : unit
  ensures { t.view == add x (old t.view) }
```

Note that `add` in the post-condition stands for the *mathematical addition of an element in a set*, which defined is defined in the Why3 standard library. □

4.3 Emptyness Check

Exercise 13. Complete the definition of function `is_empty` of the `BST` module, which checks whether the ADT represents the empty set. Your function must respect the following specification:

```
let function is_empty (t: t) : bool
  ensures { result <-> t.view == empty }
```

It is worth noting that by using `let function` and not only `function` or `let`, Why3 actually treats the name `is_empty` as both a predicate in the logic, as well as a function that returns a Boolean result. □

4.4 Minimum Element of the Set

Exercise 14. Give a Why3 definition for the relation *"a given value* `x` *is the minimum element of a tree* `t`*"*. Your predicate shall have the following signature:

```
predicate is_min_tree (x: int) (t: tree int)
```

A value `x` is the minimum element of a tree `t` if it obeys the following rules:

1. `x` belongs to `t`
2. for every other value `y` that belongs to `t`, then `x` is less than or equal to `y`.

□

Exercise 15. Using the `is_min_tree` predicate from the previous question, give a definition for the `is_min` predicate of the `BST` module. This predicate implements the relation *"*`x` *is the minimum element of the ADT"*. Your predicate shall have the following signature:

```
predicate is_min (x: int) (t: t)
```

□

Exercise 16. Give an implementation for a recursive function that removes the minimum element of a BST `t`. This function shall return the updated tree, as well as the removed minimum element. Your function must respect the following specification:

```
let rec remove_min_aux (t: tree int) : (rt: tree int, m: int)
  requires { t <> Leaf }
  requires { bin_search_tree t }
  ensures  { bin_search_tree rt }
  ensures  { is_min_tree m t }
  ensures  { not mem m rt }
  ensures  { forall y. y <> m -> occ y t = occ y rt }
  variant  { t }
```

□

Exercise 17. Using the `remove_min_aux` function from the previous question, complete the definition of function `remove_min` of the `BST` module, which removes the minimum element in the ADT. Your function must respect the following specification:

```
let remove_min (t: t) : int
  requires { not is_empty t }
  ensures  { is_min result (old t) }
  ensures  { t.view == remove result (old t.view) }
```

□

A Set ADT Interface

```
module Set
  use set.Fset

  (* private = ghost + private *)
  type t = abstract {
    mutable view : fset int;
  }

  val create () : t
    ensures { result.view = empty }

  val insert (x: int) (t: t) : unit
    writes  { t }
    ensures { t.view = add x (old t.view) }

  val function is_empty (t: t) : bool
    ensures { result <-> t.view = empty }

  predicate is_min (x: int) (t: t)

  val remove_min (t: t) : int
    requires { not is_empty t }
    writes   { t }
    ensures  { is_min result (old t) }
    ensures  { t.view = remove result (old t.view) }

end
```