Why3 Exercises: Verification of Functional Programs

I. Generic Programming

Create the following incomplete WhyML module containing definitions of polymorphic map and filter functions.

```
module FunctExerc
  use int.Int
  use list.List
  use list.Length
  use list.Mem
  use list.NthNoOpt
  type a
  type b

  let rec function map (f :a -> b) (l :list a) : list b
  =  match l with
     | Nil -> …
     | Cons h t -> …
     end

  let rec function filter (p :a -> bool) (l :list a) : list a
  = match l with
    | Nil -> …
    | Cons h t -> …
    end
```

```
end
```

The list library modules can be found here:
https://why3.lri.fr/stdlib/list.html

1. Complete the definition of the `map` function
2. Equip it with postconditions to express the following properties:
   a. For every n, the nth element of the result list and the nth element of the argument list are related by `f` as expected (use the `nth` library function)
   b. the length of the result list is the same as for the argument list (use the `length` library function)
3. Prove the resulting verification conditions

4. Complete the definition of `filter`
5. Equip it with postconditions to express the following property:
   a. any element x of type `a` is a member of the result list if and only if it is a member of the argument list and p(x) holds true

Now create a new module and define the `foldr` function:

```
module Foldr

  use int.Int
  use list.List
  use list.Permut
  use list.SortedInt
  use list.Sum

  let rec function foldr (f :'a -> 'b -> 'b) (z :'b) (l :list
'a) : 'b
  =
```

```
  match l with
  | Nil -> z
  | Cons h t -> f h (foldr f z t)
  end

...
end
```

1. One way in which we can use `foldr` is in specifications.
   a. Define a function `sumList` that sums a list of integers, using explicit recursion.
   b. Equip `sumList` with a specification stating that it returns the result of folding the binary sum operator over the list

2. Another way of course is to program with it. Let us use it to define the insertion sort algorithm.

   a. Define the "ordered insertion" function `insert` (or else, just write a spec for it)

   b. Now write the following definition, then include the usual spec in the function and try to prove the resulting VCs.

```
  let function iSort (l :list int) : list int
  = foldr insert Nil l
```

3. In principle your attempts have failed, because this definition is not explicitly recursive. But we can prove the result by writing a *lemma function*:

```
  let rec lemma iSort_sorts (l :list int)
    ensures { sorted (iSort l) }
  =
  match l with
  | Nil -> ()
  | Cons _ t -> iSort_sorts t
```

```
  end
```

This is a nice interplay between the logic and program levels of Why3: lemma functions allow for proofs to be written as programs!

- The lemma that one wants to prove is stated as the specification of the function
- The function body is like a proof script, describing the recursive argument that must be used

4. Now write a lemma function to prove that `iSort` produces a permutation of its argument.

Solutions
permalink

II. Binary Trees

Consider a (polymorphic) inductive type for (immutable) binary trees and the specification and implementation of an ordered insertion function on trees of type `int`.

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

let rec add (t : tree int) (v : int) : tree int =
    requires { sortedBT t }
    ensures  { sortedBT result }
    ensures  { size result = size t + 1 }
    ensures  { forall x : int. memt result x <-> (memt t x \/
x = v) }
    ensures  { forall x : int. num_occ x result =
                                if x = v then 1 + num_occ x t e
lse num_occ x t }
```

```
    match t with
     | Empty -> Node (Empty) v (Empty)
     | Node t1 x t2 ->
       if v <= x then Node (add t1 v) x t2
                 else Node t1 x (add t2 v)
    end
```

Define all the predicates and functions used in the specification and prove the correctness of `add`.

We note the following:

- The spec contains redundancy. Where?
- `add` can be extracted, and can also be used in logic if we include the keyword `function`, since it does not change the global state
- `sortedBT`, `size`, `memt`, and `num_occ` must be defined as pure functions in the logic namespace, and depending on how they are implemented, may also exist in the program namespace.

Recursive vs quantified logic definitions

Consider the definition of a `leq_tree` predicate to express that a given integer is not greater than any of the elements in a tree (required for definition of sorted tree). One way to express this is using a membership predicate and a universal quantifier:

```
predicate leq_tree (x : int) (t : tree int) =
     forall k : int. memt t k -> x <= k
```

A second way is to define the predicate recursively:

```
let rec predicate leq_tree (x : int) (t : tree int)
```

```
  = match t with
    | Empty -> true
    | Node t1 k t2 -> x <= k && leq_tree x t1 && leq_tree x t2
    end
```

Each definition may be more appropriate for proving different things.

But in fact in Why3 we can have both, by including the first definition as a postcondition of the second:

```
let rec predicate leq_tree (x : int) (t : tree int)
    ensures { result <-> forall k : int. not (memt t k) \/ x <
= k }
= match t with
  | Empty -> true
  | Node t1 k t2 -> x <= k && leq_tree x t1 && leq_tree x t2
  end
```

This kind of dual definition may be useful to facilitate automated proofs, but it has an additional advantage: the specifications become stronger and more trustable: since we will have to prove verification conditions to ensure that the postcondition holds, this will help us find any errors that might be present in either of the two versions of the definition.