Functional Programming and Verification

Prof. Dr. H. Seidl, N. Hartmann, R. Vogler

WS 2018/19

Exercise Sheet 8

Deadline: 16.12.2018

General Information

Detailed information about the lecture, tutorials and homework assignments can be found on the lecture website[1]. Solutions have to be submitted to Moodle[2]. Upload your solutions as a single file named 'hw08.ml' to moodle. Since submissions are tested automatically, solutions that do not compile or do not terminate within a given time frame cannot be graded and thus result in 0 points. If you do not manage to get all of your implementations compiling and/or terminating, comment them out and use the original definition (if you comment out the definition, the tests won't compile)! Use Piazza[3] to ask questions and discuss with your fellow students.

Functional Programming

Since this is a course about functional programming, we restrict ourselves to the functional features of OCaml. In other words: The imperative and object oriented subset of OCaml must not be used.

Assignment 8.5 (H) List Mishmash Reloaded [3 Points]

In assignment 5.6 you implemented the function

```
interleave3 : 'a list -> 'a list -> 'a list -> 'a list
```

Implement a tail recursive version of this function.

Assignment 8.6 (H) Lagrange [4 Points]

Given a set of points, polynome interpolation is the task of finding a polynomial with lowest degree that passes through all the points. Assume the set $\{(x_0, y_0), \ldots, (x_n, y_n)\}$ of points. Then a suitable polynomial is given by

$$L(x) := \sum_{j=0}^{n} y_j l_j(x)$$

where the Lagrange polynomials $l_j(x)$ are defined as follows:

$$l_j(x) := \prod_{\substack{0 \leq k \leq n \\ k \neq j}} \frac{x - x_k}{x_j - x_k}$$

Implement a function `lagrange : (float * float) list -> (float -> float)` that returns the interpolated polynomial L.

---

[1] https://www.in.tum.de/i02/lehre/wintersemester-1819/vorlesungen/functional-programming-and-verification/

[2] https://www.moodle.tum.de/course/view.php?id=44932

[3] https://piazza.com/tum.de/fall2018/in0003/home

Assignment 8.7 (H) Polymorphic Trees [6 Points]

So far, we used a very inflexible definition of binary trees for storing integer values. Now, we relax this limitation by defining the binary tree as a polymorphic type, such that arbitrary values may be stored at the nodes:

```
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree
```

1. Implement a function

   ```
   insert : 'a -> ('a -> 'a -> int) -> 'a tree -> 'a tree
   ```

   to insert a value (1st argument) into the tree (3rd argument) while preserving the binary search tree invariant. Note, since we do not know the type stored in the tree, the default comparison operators (<, <=, ...) cannot be used anymore to decide where to insert a new value. Imagine that we could use the tree to store `student` records ordered by `id`. Instead, a "compare" function has to be passed to `insert` as a 2nd argument. This function compares two values of type `'a` and has to return a negative number if the first value goes before the second, a positive number if the second goes before the first, and 0 if both values are considered equal. [2 Points]

2. Implement a function `string_of_tree : ('a -> string) -> 'a tree -> string` that constructs a string representation of the given tree. For `Node (7, Empty, Empty)`, the string shall be `"Node (7, Empty, Empty)"`. Again, in order to tell `string_of_tree` how to convert type `'a` to a `string`, a corresponding function has to be passed as the first argument. [1 Point]

   Hint: The tests compare the string case-sensitively, but space-insensitively.

3. Implement a function `inorder_list : 'a tree -> 'a list` that outputs all values inside the tree to a list in order. The function must be tail recursive. [3 Points]

Assignment 8.8 (H) Infinite Trees [7 Points]

The next restriction we are now going to relax is the limitation to finite trees. It is clear, that we cannot store infinite trees in finite memory, thus, we have to define the trees in a lazy fashion, such that only the subtree that is actually required is constructed, while the rest is not. Let the type

```
type 'a ltree = LNode of 'a * (unit -> 'a ltree) * (unit -> 'a ltree)
```

define polymorphic lazy (infinite) binary trees. Instead of storing a left and right child directly, we keep a function to construct them if ever needed. Note, that we do no longer need an `Empty` constructor.

Implement the following functions for infinite tree construction:

1. `layer_tree : int -> int ltree` when called with argument r constructs an infinite tree where all nodes of the nth layer store the value $r + n$. We consider the root as layer 0, so the root stores value r. [1 Point]

2. `interval_tree : (float * float) -> (float * float) ltree` constructs a tree where the left and right child of every node with interval (l, h) store the intervals $(l, \frac{l+h}{2})$ and $(\frac{l+h}{2}, h)$, respectively. The root stores the interval passed as the function's argument. [1 Point]

3. `rational_tree : unit -> (int * int) ltree` constructs a tree with root $(0, 0)$ and for every node with pair (n, d), the left child stores $(n, d+1)$ and the right child stores $(n+1, d)$. [1 Point]

Implement the following functions to work with infinite trees:

4. `top : int -> 'a ltree -> 'a tree` returns the top n (1st argument) layers of the given infinite tree as a finite binary tree. [1 Point]

5. `map : ('a -> 'b) -> 'a ltree -> 'b ltree` maps all elements of the tree using the given function (1st argument). [1 Point]

6. `find : ('a -> bool) -> 'a ltree -> 'a ltree` returns the infinite subtree rooted at a node that satisfies the given predicate (1st argument). Think about how to traverse the tree in order to make sure that every node is visited eventually. [2 Point]