CSCI 2041: Advanced Programming Principles
Fall 2017, University of Minnesota
Homework 7

Posted: Nov 27, 2017
Due: Dec 8, 2017 by 23:59

Note: Before you get started, you should copy the code in the file rbtree-sig.ml and rbtree-mod.ml. You will need to use this code in Problem 4.

Submission Protocol

You should submit two files containing the code you have written for this homework: the file `hw7.ml` containing the code for Problems 1 through 3 and the file `rbtree.ml` containing the code for Problem 4. *Mark the solution to each problem and each part both files clearly, through an OCaml comment line next to the solution you are providing.* Make sure also to stick to the names provided for the different things we want you to define, this is critical for our understanding and testing your code.

The files that you are to submit must be **pushed** to your private git repository in the github structure for the course, i.e., umn-csci-2041S15/repo-abcde001. As usualy, they must be placed at the top-level in this directory.

Before you start working on the homework, make sure to read the comments on the protocol for homeworks and the issues we consider when grading. Note in particular that you must pay attention to the structure of your programs, you must not have excessively long lines and you must use indentation to make your program text readable. Also avoid tabs: they show up differently under different editors and can make your code look ugly, something that *does matter* when we grade your work.

Finally, before submitting the final version of your homework, you should ensure it is well-formed and type-correct, i.e., contain no syntax or type errors. You may check this by running the following commands in the same directory as your homework submission on a cselabs machine:

```
ocamlbuild hw8.byte || echo "\n\n---INCORRECT SUBMISSION---\n\n"
ocamlbuild rbtree.byte || echo "\n\n---INCORRECT SUBMISSION---\n\n"
```

If you see:

```
---INCORRECT SUBMISSION---
```

for either of these commands, it means that the corresponding homework submission is not well-formed or well-typed. You should correct this problem before submission.

Problem 1 (3 + 3 points)

Consider the following function definitions

```
let rec map f l =
   match l with
   | [] -> []
   | (h::t) -> (f h) :: map f t

let sqr n = n * n

let rec take l n =
  match (l,n) with
  | ((_,0) | ([],_)) -> []
  | (h::t,n) -> h :: (take t (n-1))
```

1. Show the step-by-step evaluation of the expression

   ```
   take (map sqr [1;2;3]) 2
   ```

   under *call-by-name* evaluation. Recall that when we show the step by step evaluation, we show the state if the expression before and each function call; in particular, we do not show the process of evaluating match, if-then-else and related expressions. Note that, to get credit in this problem, you *must* show each step in the sequence so as to make clear how exactly computation proceeds. Note also that when a list is shown at the interactive level, each element of the list must be completely evaluated.

2. Repeat the above but this time using *call-by-value* evaluation.

The answers to each part should be included in comments in the file *hw7.ml* that you submit.

Problem 2 (4 + (3 + 3) + 4 points)

We will discuss in class the idea of representing an infinite object in a finite fashion in an eager language like OCaml; the technique for achieving this relies crucially on delaying evaluation by using what we will refer to as a "dummy" function. A generally useful example of an infinite object is a never-ending list, also called a *stream*. We can define a representation for such objects by means of the following type declaration and associated functions:

```
type 'a stream = Stream of (unit -> 'a * 'a stream)

let mkStream f = Stream f
let nextStream (Stream f) = f ()
```

As should be clear from looking at these definitions, a stream is essentially an encapsulation of a dummy function that generates the items in the stream, one item at a time. The function `mkStream` creates this encapsulation from a function of the required kind and the function `nextStream` probes the stream, each time giving you a pair consisting of the next item in it and an encapsulation of the rest of the stream.

Using these declarations, you can create the stream representing the infinite sequence of natural numbers as follows:

```
let rec fromNStream n = mkStream (fun () -> (n, fromNStream (n+1)))

let natStream = (fromNStream 1)
```

We assume here that the natural numbers start from 1. We can probe the members of this stream successively as shown below:

```
# let (x,rst) = nextStream natStream;;
val x : int = 1
val rst : int stream = Stream <fun>
# let (x,rst) = nextStream rst;;
val x : int = 2
val rst : int stream = Stream <fun>
# let (x,rst) = nextStream rst;;
val x : int = 3
val rst : int stream = Stream <fun>
# let (x,rst) = nextStream rst;;
val x : int = 4
val rst : int stream = Stream <fun>
#
```

This problem requires you to work with the idea of streams so as to gain some familiarity with using them.

1. Define the function

   ```
   val mapStream : ('a -> 'b) -> 'a stream -> 'b stream = <fun>
   ```

   that takes a function and a stream of suitable types and produces a new stream that corresponds to mapping the function over the items in the input stream. You should be able to produce the following kind of interaction using this function:

```
# let (x,rst) = nextStream (mapStream (fun x -> x+3) natStream);;
val x : int = 4
val rst : int stream = Stream <fun>
# let (x,rst) = nextStream rst;;
val x : int = 5
val rst : int stream = Stream <fun>
# let (x,rst) = nextStream rst;;
val x : int = 6
val rst : int stream = Stream <fun>
#
```

2. Provide definitions for `squareStream : int stream` and `cubeStream : int stream` that correspond to the sequences of the squares and the cubes of the natural numbers. Again some examples of their use:

```
# let (x,rst) = nextStream squareStream;;
val x : int = 1
val rst : int stream = Stream <fun>
# let (x,rst) = nextStream rst;;
val x : int = 4
val rst : int stream = Stream <fun>
# let (x,rst) = nextStream rst;;
val x : int = 9
val rst : int stream = Stream <fun>
# let (x,rst) = nextStream cubeStream;;
val x : int = 1
val rst : int stream = Stream <fun>
# let (x,rst) = nextStream rst;;
val x : int = 8
val rst : int stream = Stream <fun>
# let (x,rst) = nextStream rst;;
val x : int = 27
val rst : int stream = Stream <fun>
# let (x,rst) = nextStream rst;;
val x : int = 64
val rst : int stream = Stream <fun>
#
```

   Hint: you can provide very simple definitions for these streams by using `mapStream`.

3. Provide a definition for `squarecubeStream : int stream` that corresponds to a stream of numbers that are both the squares and the cubes of other numbers. Some examples of the use of this stream:

```
# let (x,rst) = nextStream squarecubeStream;;
val x : int = 1
val rst : int stream = Stream <fun>
# let (x,rst) = nextStream rst;;
val x : int = 64
val rst : int stream = Stream <fun>
# let (x,rst) = nextStream rst;;
val x : int = 729
val rst : int stream = Stream <fun>
#
```

I also want you to solve this problem in a particular way. The new stream should be generated by a function that does the following to determine the next element: it repeatedly generates items in the square and cube streams till it finds an item in common; the rest of the two streams will be kept around in the second "stream" component of the pair in a form needed to repeat the process.

Problem 3 ((2 + 4) + 3 points)

The way we defined streams in the previous problem was a bit inefficient: each time we looked at a stream, we would have to "evaluate" the function representing it to get the next item and the remaining stream. If this evaluation step is costly, we would like to avoid it. We can do this using the following idea: we allow the stream to be represented by either a pair or by a function that can be invoked to provide the pair. Moreover, we use a reference to store this object so that we can update it to avoid re-evaluation after we have done this once. To use this idea, we define the *stream* type as follows:

```
type 'a stream' = Stream' of 'a stream_aux ref
and  'a stream_aux =
       | Evald of ('a * 'a stream')
       | UnEvald of (unit -> 'a * 'a stream')
```

Your task in this problem is to understand this definition and show that you have understood it by using it in a few definitions.

1. Provide definitions for the following functions that can be used to create and probe streams using this new representation:

   ```
   mkStream' : (unit -> 'a * 'a stream') -> 'a stream'
   nextStream' : 'a stream' -> 'a * 'a stream'
   ```

   The first function should be easy to figure out. You have to be a bit careful in defining *nextStream'*. Here is why: if you "evaluate" the stream when you do a lookup, then you should modify it so that there is no evaluation needed the next time around.

2. Provide a definition for `natStream' : int stream'` that is like `natStream` from Problem 2, except that it uses the new representation of streams.

When you are all done with this problem, you should be able to produce interactions like the following:

```
# natStream';;
- : int stream' = Stream' {contents = UnEvald <fun>}
# let (x,rst) = nextStream' natStream';;
val x : int = 1
val rst : int stream' = Stream' {contents = UnEvald <fun>}
# natStream';;
- : int stream' =
Stream' {contents = Evald (1, Stream' {contents = UnEvald <fun>})}
```

```
# let (x,rst) = nextStream' rst;;
val x : int = 2
val rst : int stream' = Stream' {contents = UnEvald <fun>}
# natStream';;
- : int stream' =
Stream'
 {contents =
   Evald
    (1, Stream' {contents = Evald (2, Stream' {contents = UnEvald <fun>})})}
#
```

Observe how the structure of `natStream'` gets more and more elaborated as we look for items in it; your code should exhibit a similar behaviour, this is the purpose of using references in this problem.

Problem 4 (5 + 5 + 5 points)

A while ago, we looked at the binary search tree definition and saw how we could use the idea of embedding functions in data to provide a type declaration that allowed us to say what ordering relation was relevant to the tree. This allowed us to realize polymorphism that was truly parametric, i.e. we no longer had to cross our fingers and hope that the < relation "worked" on the type. In this problem, we will see how to realize this kind of polymorphism in a different way, using modules and functors. To allow you to focus on the parts of this exercise that are really important, I will give you almost all the code you have to write. Your task is to exhibit an understanding of modules and functors by adding a few lines of code to complete the definitions and by then using the definitions to create a few different kinds of binary search tree structures.

1. The file rbtree-sig.ml contains code defining two signatures: the `DATA` signature and the `BTREE` signature. The first signature identifies the kinds of things that are needed over the data that we want to use as the elements in a binary search tree. The `BTREE` signature identifies what functionality we will provide if we are able to define a binary search tree structure over the corresponding data. The file rbtree-mod.ml contains code for realizing the different functions based on a particular implementation---the red-black tree implementation---of binary search trees; you would have seen this code in Lab 13 and, hopefully, we will get a chance to talk about it in class, but the details are not so important to this problem. The key thing for us is that this code is incomplete in two respects. First, it assumes available a *Data* module that provides the functionality required by the `DATA` signature. Second, it is missing a surrounding structure that, after type checking, assures us that the code in *rbtree-mod.ml* will in fact realize all the expected functionality for binary search trees if it is given a *Data* module that satisfies the properties described by the `DATA` signature.

   Your task is to complete this missing pieces. More specifically, put all the code in the two files in a file called *rbtree.ml*. Then surround the code you have copied from the file *rbtree-mod.ml* in a way that converts it into the definition of a *functor* with the name *BTree* that takes in a structure called *Data* that satisfies the properties identified by the `DATA` signature

and produces a structure satisfying the properties identified by the `BTREE` signature. Then load the file into an OCaml interactive session and make sure that it compiles.

Note: that you **must** name the functor *BTree*, this is important for us to be able to test your code.

2. Now add to the file the definition of a structure called *IntData* that meets the requirements of `DATA`. Then, using this structure and the *BTree* functor, complete the following definition:

```
module IntBTree = ...
```

   to realize a binary search tree structure over integer data. Test your code by building some integer binary search trees, searching them and printing them. We will test your code in this way, so it is a good idea to make sure it works.

3. Finally, show the versatility of the code you have at this point by using it to generate another binary search tree structure, this time one that works over strings. Specifically, add to the file the definition of a structure called *StringData* that meets the requirements of `DATA` and use this together with the *BTree* structure to complete the following definition:

```
module StringBTree = ...
```

   Again, test your code by building, probing and printing a string binary search tree.

In case you were wondering about the kind of behaviour to expect once you have filled out the definitions, here is what I get from my program. Note that the tree is printed as if it is lying on its side, the root to the left and the branches extending rightwards.

```
# let inttree =
        IntBTree.insert (1,
         (IntBTree.insert (2,
            (IntBTree.insert (3,
               (IntBTree.insert (4,
                  (IntBTree.insert (5,
                      (IntBTree.insert (6,
                         IntBTree.insert (7,
                           IntBTree.insert (8,
                             IntBTree.initTree ()))))))))))));;
val inttree : IntBTree.btree = <abstr>
# IntBTree.print stdout inttree;;
      1
    2
  3
    4
5
    6
  7
    8
- : unit = ()
# let strtree =
        StringBTree.insert ("a",
```

```
          (StringBTree.insert ("b",
             (StringBTree.insert ("c",
                (StringBTree.insert ("d",
                   (StringBTree.insert ("e",
                       (StringBTree.insert ("f",
                          (StringBTree.insert ("g",
                             (StringBTree.insert ("h",
                                StringBTree.initTree ()))))))))))))));;
val strtree : StringBTree.btree = <abstr>
# StringBTree.print stdout strtree;;
      "a"
    "b"
  "c"
    "d"
"e"
    "f"
  "g"
    "h"
- : unit = ()
#
```

You can actually adapt the code a bit to understand how the balancing is done: change the tree printing routine a bit to also display the colour of the node. Then see what happens to the tree as you insert nodes in it successively. This can be fun to watch!

Last modified: Nov 26, 2017. Created by gopalan atsign cs dot umn dot edu. Copyright (c) Gopalan Nadathur.
