02157 Functional programming

Michael R. Hansen
DTU Compute
October 9, 2023

Exercises Week 8

This exercise set consists of 2 problems:

Problem 1 is the first problem from the exam set from May, 2022.

Problem 2 is the the second problem (Questions 1, 2, 3 and 7) from the exam set from May, 2020. Questions 4, 5 and 6 from that exam set concern tail-recursive functions and will be addressed later.

Problem 1

A teacher named Robin has a bookshelf with book that are lent to colleagues and students. The following models of the shelf and the loans are introduced to keep track of books:

```
type Book  = string
type Shelf = Book list   // ordered alphabetically

type Date  = int
type Name  = string
type Loan  = Book * Name * Date
```

Books are just modelled by strings and we assume below that the books appear in alphabetic order in a shelf. (Built-in orderings `<`, `<=`, etc. can be used to compare books.) A shelf may contain multiple copies of the same book.

A loan is modelled by a triple (b, n, d), where b is a book, n the name of the borrower and d the date when the book was borrowed. Names are strings and dates are integers. Consider, for example, the following declarations of a shelf `sh0` with three books and a list `ls0` containing four loans.

```
let sh0 = ["Introduction to meta-mathematics";
           "To mock a mockingbird";
           "What is the name of this book"];;

let ls0 = [("Communication and concurrency", "Bob", 4);
           ("Programming in Haskell", "Paul", 2);;
           ("Communicating Sequential processes", "Mary", 7);
           ("Elements of the theory of computation", "Dick", 1)];;
```

The questions 1. to 6. in this problem should be solved without using functions from the libraries `List`, `Seq`, `Set` and `Map`. That is, the requested functions should be declared using explicit recursion.

In the declarations you can assume that books are ordered alphabetically in shelf arguments to functions. It is required that books are ordered alphabetically in shelves returned by functions.

1. Declare a function `onShelf: Book -> Shelf -> bool` that can check whether a book is on a shelf.

2. Declare a function `toShelf: Book -> Shelf -> Shelf` so that `toShelf` b bs is the shelf obtained from bs by insertion of b in the right position.

3. Declare a function `fromShelf: Book -> Shelf -> Shelf option`. The value of the expression `fromShelf` b bs is `None` if bs does not contain b. Otherwise, the value is `Some` bs', where bs' is obtained from bs by deletion of one occurrence of b.

4. Declare a function `addLoan` b n d ls, that adds the loan (b, n, d) to the list of loans ls.

   Furthermore, declare a function `removeLoan` b n ls. The value of the function is the list obtained from the list of loans ls by deletion of the first element of the form (b, n, d), where d is some date, if such an element exists. Otherwise ls is returned. For example, `removeLoan "Programming in Haskell" "Paul" ls0` gives the list

```
[("Communication and concurrency", "Bob", 4);
 ("Communicating Sequential processes", "Mary", 7);
 ("Elements of the theory of computation", "Dick", 1)]|
```

5. Declare a function `reminders: Date -> Loan list -> (Name * Book) list`. The value of `reminders` d_0 ls is a list of pairs (n, b) from loans (b, n, d) in ls where $d < d_0$. We interpret $d < d_0$ as "date d is before date d_0".

   For example, `reminders 3 ls0` has two elements: `("Paul","Programming in Haskell")` and `("Dick", "Elements of the theory of computation")`.

6. In this problem, we consider a textual form of the reminders from Question 5, where, for example, a letter reminding Paul to return "Programming in Haskell" has the form:

```
"Dear Paul!
 Please return "Programming in Haskell".
 Regards Robin"
```

   Declare a function `toLetters: (Name * Book) list -> string list`, that transforms a list pairs (n, b) to a list of corresponding strings (letters). Notice, the escape characters `\n` and `\"` denote newline and citation quotation, respectively.

7. This question should be solved using functions from the `List` library. You should *not* use explicit recursion in the declarations.

   1. Give an alternative declaration of `toLetters` using `List.map`.
   2. Give an alternative declaration of `reminders` using `List.foldBack`.

Problem 2

The function `allPairs` from the `List` library could have the following declaration:

```
let rec f x = function
              | []    -> []
              | y::ys -> (x,y)::f x ys;;
val f : 'a -> 'b list -> ('a * 'b) list

let rec allPairs xs ys =
   match xs with
   | []       -> []
   | x::xrest -> f x ys @ allPairs xrest ys;;
val allPairs : 'a list -> 'b list -> ('a * 'b) list
```

where `f` is a helper function. Notice that the F# system automatically infers the types of `f` and `allPairs`.

1. Give an argument showing that `'a -> 'b list -> ('a * 'b) list` is indeed the most general type of `f` and that `'a list -> 'b list -> ('a * 'b) list` is indeed the most general type of `allPairs`. That is, any other type for `f` is an instance of `'a -> 'b list -> ('a * 'b) list`. Similarly for `allPairs`.

An example using `f` is:

```
f "a" [1;2;3];;
val it : (string * int) list = [("a", 1); ("a", 2); ("a", 3)]
```

2. Give an evaluation showing that `[("a", 1); ("a", 2); ("a", 3)]` is the value of the expression `f "a" [1;2;3]`. Present your evaluations using the notation $e_1 \rightsquigarrow e_2$ from the textbook, where you can use `=>` in your F# file rather than $\rightsquigarrow$. You should include at least as many evaluation steps as there are recursive calls.

3. Explain why the type of `f "a" [1;2;3]` is `(string * int) list`.

4. Give another declaration of `f` that is based on a single higher-order function from the `List` library. The new declaration of `f` should not be recursive.