F28PL – Coursework 2

Submission date

3:30pm (local time) Dec 08, Friday

Instructions

1. There are *five questions* totaling **50 marks**. You need to attempt all of them.
2. You cannot use library functions in your code. All helper functions you use need to be implemented by you in the same file.
3. Code should be clearly written and laid out and should include a brief explanation in English explaining the design of your code.
4. Your answers need to be valid OCaml and Python code. **Code that cannot compile may score zero marks.**
5. Submit your work by pushing your code to the Gitlab server. Only code that has been pushed to **your fork** of the project before the deadline will be marked. We are *not* using Canvas for coursework submission.

Question 1 (15 marks)

A filesystem consists of directories (folders) and files with their respective names, where each directory can subsequently consist of other arbitrary files and directories. A path to a file or directory is a list consisting of the names of the respective directories we are going down (and this list contains the name of the file at the end, if we are pointing to a file). We represent such a tree structure of the filesystem and the type of paths as the following OCaml types `ftree` and `path`.

```
type ftree = Dir of string * ftree list
           | File of string
type path = string list
```

For example, the following file tree

```
root/
|-- desktop/
|   |-- doc1
|   |-- notes
|   +-- emptydir/
|-- documents/
|   |-- notes
|   +-- text1
|-- file1
+-- file2
```

can be represented as a value of type `ftree` in the following way.

```
let fs =
  Dir ("root",
        [ Dir ("desktop", [ Dir ("emptydir", [])
                          ; File "doc1"
                          ; File "notes"
                          ])
        ; Dir ("documents", [ File "notes"
                            ; File "text1"
                            ])
        ; File "file1"
        ; File "file2"
        ])
```

(The order of the subsequent files and directories in the list in the `Dir` constructor does not matter.)

The path to the `notes` file in the desktop is the list `["root"; "desktop"; "notes"]` and the path to the `documents` directory is the list `["root"; "documents"]`, for example.

(a) Define a function `numFiles : string -> ftree -> int` which takes a name and a filesystem as input and returns the number of *files* with that name in the filesystem. Also define a function `numDirs : ftree -> int` which returns the total number of directories in the filesystem. For example, `numFiles "notes" fs` should evaluate to `2`, `numFiles "doc1" fs` should evaluate to `1`, `numFiles "desktop" fs` should evaluate to `0` (since we only care about files), and `numDirs fs` should evaluate to `4`. **(3 marks)**

(b) Above we gave two examples of *valid* paths. An example of an invalid path is `["root"; "desktop"; "homework"]`, since there is no file nor directory called `"homework"` in the desktop.

Define a function `validPath : path -> ftree -> bool` which takes as input a path and a filesystem and returns `true` if a file or directory with such a path exists in the filesystem, and `false` otherwise. We will consider the empty path `[]` to be *invalid*. **(4 marks)**

(c) Recall that normally, in a given directory, no two files and no two directories are allowed to have the same name. We call a filesystem *valid* if it satisfies this condition. Define a function `nodupes : 'a list -> bool` which returns `true` if there are no duplicate elements in the given list (i.e. each element occurs uniquely). Hence, or otherwise, define a function `validFTree : ftree -> bool` which returns `true` if the given filesystem is valid.

For example, `fs` in the example given at the start is a valid filesystem. Renaming `"desktop"` to `"documents"` results in an invalid filesystem (due to two directories having the same name) and similarly renaming `"text1"` to `"notes"` results in an invalid filesystem. **(5 marks)**

(d) Define the function `allFiles : ftree -> path list` which returns the list containing the full paths of all the files (and not directories) in the given filesystem. For example, `allFiles fs` should return the following list of paths.

```
[ ["root"; "desktop"; "doc1"]
```

```
  ; ["root"; "desktop"; "notes"]
  ; ["root"; "documents"; "notes"]
  ; ["root"; "documents"; "text1"]
  ; ["root"; "file1"]
  ; ["root"; "file2"]
  ]
```

The order of the paths in the resultant list does not matter. **(3 marks)**

Question 2 (5 marks)

Certain types, especially ones that are highly polymorphic, admit *natural* implementations of functions of that type. Intuitively, a natural implementation should not cause errors, exceptions or nontermination (other than possibly by calling its arguments), should not discard values, and the inferred type should be the same as specified (up to renaming of type variables).

(a) Provide natural implementations of the two functions `f1` and `f2` below.

```
f1 : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
f2 : (('a -> 'b) -> 'a) -> ('a -> 'b) -> 'b
```

(2 marks)

(b) In the comments, carefully explain the process you took to arrive at your implementations of `f1` and `f2`. Do there exist any natural implementations for `f1` and `f2` other than the ones you chose? **(3 marks)**

Question 3 (10 marks)

Consider the problem of *editing* a list — i.e., adding or removing elements at various positions, like you would when implementing (say) a text editor, where the edits generally take place at the cursor. In an imperative language, where the structure of the list is mutated by edits, it is easy to keep a pointer into a node of the list and perform multiple changes in its vicinity cheaply.

Things are not so simple in a purely functional setting, where a modification of a list needs to *preserve* the old copy, and thus copying may occur. This makes a naive implementation of editing procedures can be extremely inefficient. In this question, you need to figure out a way to perform edits efficiently in a functional setting.

To make the task easier, we provide an *interface* that you need to realize. The interface is stated in terms of a type `’a place`, which represents a place in a list, and consists of two functions that convert between lists and places, navigation functions for moving to the previous/next place, and element lookup/deletion/insertion at current place. Their types, intended meaning, and a few examples, are defined below:

- `getPlace : ’a list -> int -> ’a place` returns the view of the list *focused* on the given place. The index needs to be between `0` and `length xs`, inclusive(!), otherwise an exception should be raised.
- `collapse : ’a place -> ’a list` forgets the information about the place and returns the underlying list.
- The functions `isStart : ’a place -> bool` and `isEnd : ’a place -> bool` check whether we are at the start (index `0`) or at the end (index `length xs`), respectively.
- `next : ’a place -> ’a place` and `prev : ’a place -> ’a place` return the representation of next/previous place in the list, respectively.
- `lookup : ’a place -> ’a` returns the value of the element *after* the current place in the list; if we are at the end of the list, an exception should be raised.
- `delete : ’a place -> ’a place’` returns the representation of a place in a list where the element *after* the current one in the argument is deleted; if the place in the argument is at the end of the list, an exception should be raised.

- `insert : 'a -> 'a place -> 'a place` returns the representation of a place in a list, where the given element was inserted *before* the current place in the argument (mimicking the behaviour of a text editor).

For instance, the following expressions should return true:

- `getPlace [] 0 |> insert 1 |> insert 2 |> collapse = [1; 2]`
- `getPlace [1; 2; 2; 3] 2 |> delete |> collapse = [1; 2; 3]`
- `getPlace [1; 2; 2; 3] 2 |> delete |> lookup = 3`
- `getPlace [1; 2; 2; 3] 0 |> next |> delete |> next |> next |> isEnd`
- `p |> insert x |> prev |> lookup = x` for any `p : 'a place` and `x : 'a`

(a) Define the type `'a place` that makes editing of a list possible. **(2 marks)**

(b) Implement the interface by defining all the functions above. **(5 marks)**

(c) Explain your choice of representation of `'a place` and how it enables efficient implementation of the functions. **(3 marks)**

Question 4 **(12 marks)**

Recall that in the game of chess, a *rook* moves only either horizontally or vertically and a *knight* moves two squares vertically and one square horizontally, or two squares horizontally and one square vertically, jumping over other pieces.

In this question we will consider a variant of chess where the size of the board can be $n \times n$ for any n, and where we introduce a new piece called the *knook* – a piece which can move and attack as both a knight or a rook in any given turn. Given an $n \times n$ board with n knooks, we wish to find a configuration of the knooks on the board such that no two pieces are threatening each other (i.e. no two knooks are within attacking range of each other).

We set the convention that the bottom left square of the board is represented by the pair `(0,0)` and the square `(a,b)` is the square `a` squares from the left and `b` squares from the bottom.

(a) Define a function `threatening(s1,s2)` which takes as input the positions `s1` and `s2` of two knooks and returns `True` if they are within attacking range of each other and `False` otherwise.

For example, `threatening((0,0),(2,2))` is `False` but both `threatening((0,0),(1,2))` and `threatening((0,0),(4,0))` are `True`. **(2 marks)**

(b) Define a function `safeConfig(sqs)` which takes as input a list of positions `sqs` that knooks are placed on and returns `True` if no pair of knooks is threatening each other and `False` otherwise.

For example, `safeConfig([(0,0),(1,1),(2,2)])` is `True` and `safeConfig([(0,0),(1,2),(4,4)])` is `False`.

Write equivalent functions `threatening : (int * int) * (int * int) -> bool` and `safeConfig : (int * int) list -> bool` in OCaml. **(6 marks)**

(c) Define a Python function `allKnooks(n)` which takes as input the size of a chess board and returns a list of lists of positions of n knooks (i.e. a list of configurations) such that none of the knooks are threatening each other. Below are two examples.

```
allKnooks(2) == [ [(0, 1), (1, 0)]
                , [(0, 0), (1, 1)] ]
```

```
allKnooks(4) == [ [(0, 3), (1, 2), (2, 1), (3, 0)]
                , [(0, 1), (1, 2), (2, 3), (3, 0)]
                , [(0, 2), (1, 3), (2, 0), (3, 1)]
                , [(0, 0), (1, 3), (2, 2), (3, 1)]
                , [(0, 3), (1, 0), (2, 1), (3, 2)]
                , [(0, 1), (1, 0), (2, 3), (3, 2)]
                , [(0, 2), (1, 1), (2, 0), (3, 3)]
                , [(0, 0), (1, 1), (2, 2), (3, 3)]
                ]
```

Furthermore there are 20 possible configurations of 5 knooks on a 5×5 board and 94 possible configurations on a 6×6 board.

There are many ways of arriving at an answer. One possible method is to start with safe configurations on smaller subsets of the board and to incrementally add pieces, eventually covering the full board, such that the configurations still remain safe. Note that the order of the configurations does not matter.

Clearly detail your solution and the idea behind it in the comments, and mention any optimisations you implement towards making the search for solutions more efficient. **(4 marks)**

Question 5 (8 marks)

A *unit fraction* is a fraction where the numerator is 1. In this question we are concerned with the decimal expansion of unit fractions. For example, the decimal expansion of 1/4 is 0.25 and the decimal expansion of 1/12 is 0.08333.... This can also be written as 0.08(3), denoting that the 3 is a 1-digit repeating cycle. It can be seen that 1/7 has a 6-digit repeating cycle since it is equal to 0.(142857).

(a) Define a function `toDecimal(n)` which takes a number $n > 1$ as input and returns a pair of lists as output – the first list should contain the *non-repeating* digits of the decimal expansion (after the decimal point), and the second list should contain one copy of the *repeating* digits. For example, `toDecimal(4)` should return `([2,5],[])`, `toDecimal(12)` should return `([0,8],[3])`, and `toDecimal(7)` should return `([],[1,4,2,8,5,7])`. It may help to recall long division from school.

Explain how your code works in the comments. **(5 marks)**

(b) Define a function `toFrac(nonrep,rep)` which goes the other way – it takes as input a pair of lists representing the non-repeating and repeating segments of a decimal expansion (where the whole number part is 0) and returns a pair of numbers divided to obtain those digits. The fraction should be in its simplest form. For example, `toFrac([2,5],[])` should return `(1,4)`, `toFrac([0,8],[3])` should return `(1,12)`, and `toFrac([1],[3])` should return `(2,15)`. You may want to recall the usual technique for converting decimal expansions to fractions from your earlier years. **(3 marks)**