F28PL – Coursework 1

Submission date

3:30pm (local time) October 26, Thursday

Instructions

1. There are *three questions*, each worth **10 marks**, giving a total of **30 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 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

(a) Define the function `all : ('a -> bool) -> 'a list -> bool` such that `all p xs` is `true` if and only if the predicate `p : 'a -> bool` holds for all elements of `xs : 'a list`. You should define `all` using explicit recursion.

For example, if `isEven : int -> bool` is the predicate returning `true` if the input is even, `all isEven [4;2;4;8]` should evaluate to `true` and `all isEven [2;2;5;1]` should evaluate to `false`. **(3 marks)**

(b) Define the function `exists : ('a -> bool) -> 'a list -> bool` such that `exists p xs` is `true` if and only if there is an element of `xs : 'a list` for which the predicate `p : 'a -> bool` holds. You should define `exists` using either `foldLeft` or `foldRight`.

For example, `exists isEven [1;3;4;5]` should evaluate to `true` and `exists isEven [5;1]` should evaluate to `false`. **(3 marks)**

(c) Consider a version of `all` now implemented via a fold (like `exists` in part (b)). Call this function `allF : ('a -> bool) -> 'a list -> bool`. Do the functions `all` and `allF` behave the same on all inputs? If not, provide an example and explain the difference. **(2 marks)**

(d) Give specifications of the functions `all` and `exists` as predicates, calling them `allSpec` and `existsSpec` respectively. For this make use of the functions `all`, `exists`, and a membership function `member : 'a -> 'a list -> bool` in the implementation. (`member x xs` returns `true` if `x` is an element of the list `xs`.) What are the types of `allSpec` and `existsSpec`? **(2 marks)**

Question 2

(a) Define a function `extractMin : 'a list -> 'a * 'a list` that splits a given list into a pair containing its least element and a list containing all the other elements of the input list. For example, `extractMin [3;6;1;3;7]` should evaluate to the pair `(1,[3;6;3;7])`. **(3 marks)**

(b) Using `extractMin`, define a function `extractSort : 'a list -> 'a list` that sorts its argument. **(3 marks)**

(c) Describe the evaluation of your implementation of `extractSort` on the input `[2;3;1]` using the substitution model of computation. **(2 marks)**

(d) For each of your definitions state whether it is tail recursive and whether it generates garbage. **(2 marks)**

Question 3

Recall an old riddle: you have a 5 litre jar and a 2 litre jar, without any markings – can you measure out exactly 1 litre of water? The answer is, of course, to fill the 5 litre jar, transfer water to the 2 litre one until the latter is full, pour the water out of the small jar, and repeat the process, leaving you with 1 litre of water in the big jar. In this exercise, we will solve a generalised version of the riddle.

Our description of the game will consist of two pieces of data: a list of integers that represent the volume of jars we have at our disposal, and an integer that represents the target volume that we want to measure out. Moreover, we represent the *possible moves* with the following data type:

```
type move = Fill of int | Drain of int | Transfer of int * int
```

The three constructors represent the three possible actions that we can take:

- `Fill n` represents filling the nth jar with water (no matter how much water was in it to begin with);
- `Drain n` represents emptying the nth jar (emptying a jar is valid action even if it was already empty to begin with);
- `Transfer (m, n)` represents pouring water from the mth jar to the nth jar until either the mth jar is empty, or the nth jar is full, whichever comes first.

We index the jar starting from 0, and assume that we never ask for a jar with an index outside the range (i.e., the number `n` always satisfies `0 <= n < length jars`, where `jars` is the given list of jars), and the two indices given to `Transfer` are always distinct.

We split the task of solving the riddle into two parts.

1. Executing moves and checking the end-game condition (4 marks)

We begin with implementing the code to execute a strategy, expressed as a series of moves. To that end, implement the functions described below.

(a) Implement `doMove : int list -> int list -> move -> int list`. The first parameter is the list of volumes of the jars, the second — the list that describes the current contents of the jars, and the third — the move to be executed. The result is a list that describes the updated contents of the jars. You can assume that the lists are always the same length, no jar contains more water than its volume, and the indices in the move are valid. For example, calling `doMove [5; 2] [5; 0] (Transfer (1, 0))` should result in a list `[3; 2]`.

(b) Implement a function `check : int -> int list -> bool` that checks if any of the jars contains the required amount of water.

(c) Using the two functions defined above, implement `run : int list -> int -> move list -> bool`, that takes as parameters the list of volumes of jars, the target amount and a list of moves, executes the moves from the starting state (all jars empty), and checks whether in the final state at least one of the jars contains the required amount. For example, calling `run [5; 2] 1 [Fill 0; Transfer (0, 1); Drain 1; Transfer (0, 1)]` should return `true` — this is the solution of the original riddle.

2. Solving the riddle (6 marks)

The second part of the problem is more free-form, and will require you to come up with the solution on your own. Define a function `solve : int list -> int -> move list option` that takes as input the list of volumes and the desired amount, and tries to produce the shortest list of moves whose execution in the initial state results in a state with at least one jar containing the required amount of water. If such a sequence of actions is found (call it `ms`), the program returns `Some ms`, and if there is no solution, it should return `None`.

For instance, for the original riddle, the call `solve [5; 2] 1` should return the unique shortest solution, which is `Some [Fill 0; Transfer (0, 1); Drain 1; Transfer (0, 1)]`, while for the slightly modified riddle, `solve [6; 2] 1`, it should return `None` (since there is no way in which we can obtain an amount of 1). In a comment below your solution, explain your implementation: why will it find a solution, why it should be the shortest, how you will determine that no solution exists, etc.

Note: solutions that loop infinitely when no solution exists (but work correctly otherwise) will be docked 1 mark; similarly for solutions that return solutions that are correct, but not shortest — so do not worry overly if your solution falls into one (or both) of these categories.