F28PL – Resit Coursework

Submission date

3:30pm (local time) Aug 08, Thursday

Instructions

1. There are *four 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. All code should be clearly written and laid out and should include an 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 [OCaml] (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) -> ('a -> 'b) -> 'a -> 'c
f2 : ('a -> 'b -> 'c) -> 'a -> ('d -> 'b) -> 'd -> 'c
```

(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 2 [OCaml] (15 marks)

Consider the class of certain mystery functions of *one argument*. The *mystery* functions can be formed in one of four ways:

- the *variable* is a valid mystery function (note, there is only one variable, but it can possibly occur many times in a function);
- any integer *constant* is a valid mystery function;
- a *product* of two mystery functions is a mystery function;
- a *sum* of two mystery functions is a mystery function.

(a) Represent the syntax of mystery functions, as defined above, as an algebraic datatype `type myst`. **(3 marks)**

(b) Define a function `evalAt : myst -> int -> int`, which takes a mystery function and a point at which we want to compute its value, and evaluates the function (the point should be used as the value for the variable). **(2 marks)**

(c) It turns out that mystery functions are just another way of representing polynomials (see lecture notes for weeks 3/4), which we represented as lists of floats, with no trailing zeroes, which denoted the coefficients in increasing order. Here we shall just consider integer coefficients, giving us the alternative definition `type poly = int list`. Define a function `asPoly : myst -> poly` that converts a polynomial from our representation as mystery function to the one in the lecture notes. To help in this, define two helper functions: `addPoly : poly -> poly -> poly` and `mulPoly : poly -> poly -> poly`, which respectively add and multiply polynomials. **(6 marks)**

(d) Define a function `evalAtP : poly -> int -> int`, analogous to `evalAt` from part (b). Using this, define a predicate on `myst * int` (i.e., a function of type `myst * int -> bool`) which can be used to test the correctness of `asPoly` from part (c). In other words, assuming all functions have been correctly defined, this predicate should return `true` on all inputs. **(2 marks)**

(e) Define a function `eqMyst : myst -> myst -> bool` that checks whether two mystery functions are equal in the usual, functional sense (i.e., their results match for any arguments). **(2 marks)**

Hint: This is infeasible by checking all inputs, and difficult directly — but you can use your solution to one of the previous parts of this question!

Question 3 [OCaml] (10 marks)

In this question we will be working with infinite lists of values, which we hereby refer to as *inflists*. Examples of inflists of integers are the list of all non-negative integers $[0, 1, 2, 3, \ldots]$ and the list of all even integers $[0, 2, 4, 6, \ldots]$. An example of an inflist of booleans is the infinite list of alternating booleans [`true`, `false`, `true`, `false`, ...].

One way to represent an inflist of some type `'a` is as a function from (non-negative) integers to `'a`, where the value at the n^{th} position of the inflist can be obtained by evaluating the given function at n. In this representation the inflists $[0, 1, 2, 3, \ldots]$ and $[0, 2, 4, 6, \ldots]$ are given by the functions `l1 = fun n -> n` and `l2 = fun n -> n*2`, respectively. We take the definition of our type `inflist` as follows:

```
type 'a inflist = int -> 'a
```

We shall now define some functions to operate on inflists.

Note: for each question below ensure that you clearly explain how you arrived at all your respective implementations and demonstrate (by pasting the corresponding outputs of example computations from the interpreter) that all your functions work as intended.

(a) Define the inflist of alternating booleans [`true`, `false`, `true`, `false`, ...] as a value of type `bool inflist`. **(1 mark)**

(b) As with finite lists, we can consider analogues of the functions `take`, `drop`, and `append` on inflists. Define the following three functions.

- `take : int -> 'a inflist -> 'a list`
- `drop : int -> 'a inflist -> 'a inflist`
- `append : 'a list -> 'a inflist -> 'a inflist` **(3 marks)**

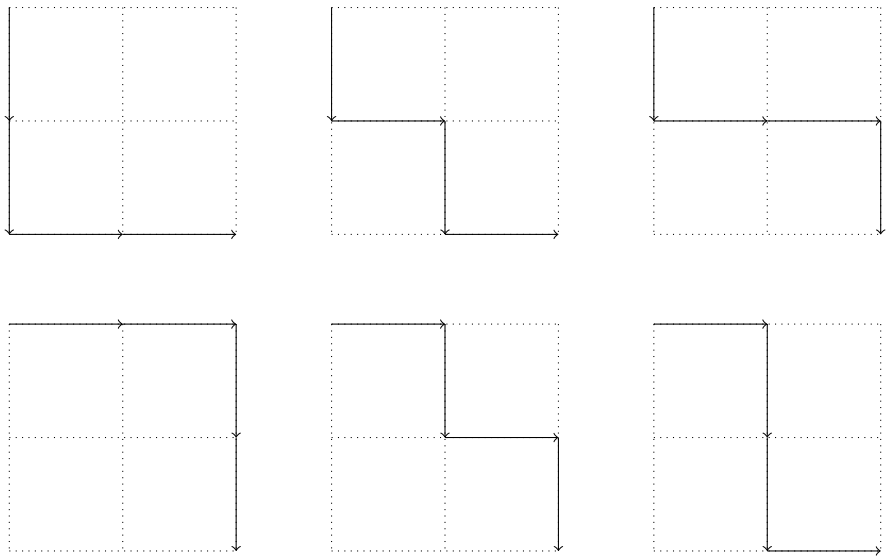
(c) Recall the functions `map` and `filter` for finite lists. Write down the corresponding types for the `inflist` versions of these two functions followed by their implementations. **(2 marks)**

(d) Consider inflists $[x_1, x_2, x_3, \ldots]$ and $[y_1, y_2, y_3, \ldots]$. These two lists can be *interleaved* to obtain the infinite list $[x_1, y_1, x_2, y_2, x_3, y_3, \ldots]$. Define the function `interleave : 'a inflist -> 'a inflist -> 'a inflist`. **(2 marks)**

(e) It is possible to produce inflists of type `'b` by starting with some given initial seed value `seed : 'a` and an iteration function `iter : 'a -> 'a * 'b`. From this we can obtain an inflist of `'b`s by repeatedly applying `iter` to `seed` and projecting out the appropriate value of type `'b` and using the value of type `'a'` as the next seed. Implement this process as the function `produce : 'a -> ('a -> 'a * 'b) -> 'b inflist` and show how the inflist of all odd integers can be defined using `produce`. **(2 marks)**

Question 4 [Python] (20 marks)

This question is about paths on a grid. The paths always start at the top-left corner and end at the bottom-right and at each step are only allowed to either go *down* or to the *right*. Below are *all six* such paths that one can get on a 2×2 grid.



We will write out paths as strings consisting of the letters "`d`" (down) and "`r`" (right). So, for example, the first path above is "`ddrr`" and the second path is "`rddr`".

(a) Write a function `isValidPath(m,n,p)` which takes the dimensions `m` (height), `n` (width), and a path `p` (as a string) and returns either `true` or `false` depending on whether or not the path is a valid one on a given $m \times n$ grid. **(3 marks)**

(b) Consider the function `allPaths(m,n)` which returns the list of all valid paths on an $m \times n$ grid. What should such a function return when one or both of the arguments is zero? Now suppose `m` and `n` are non-zero: how can we then express `allPaths(m,n)` in terms of the results obtained from computing `allPaths(m-1,n)` and `allPaths(m,n-1)`? Hence, or otherwise, implement `allPaths(m,n)`. **(7 marks)**

(c) Explain what you observe when you run `allPaths(30,30)`. **(2 marks)**

(d) We say a path has *turning number* k if it has k turning points. In the 2×2 example above, the first three paths have turning numbers 1, 3, and 2,

respectively. Write a function `numTurningPaths(m,n,k)` which returns the number of paths in an $m \times n$ grid with turning number k. **(4 marks)**

(e) We say path `p1` is *above* path `p2` if at no point along `p1` do we cross `p2`. In the 2×2 grid all the paths are above "`ddrr`" (including itself), and only "`rrdd`", "`rdrd`", and "`drrd`" are above "`drrd`". Write a function `pathsAbove(m,n,p)` which returns the list of all paths above path `p` in an $m \times n$ grid. **(4 marks)**