02157 Functional programming

Michael R. Hansen
DTU Compute
September 25, 2018

Mandatory assignment 1

This is the first of four mandatory assignments in 02157 Functional programming. It is a requirement for exam participation that 3 of the 4 mandatory assignments are approved. The mandatory assignments can be solved individually or in groups of 2 or 3 students.

Acceptance of a mandatory assignment from previous years does NOT apply this year.

- Your solution should be handed in **no later than Thursday, October 4, 2018**. Submissions handed in after the deadline will face an *administrative rejection.*
- The assignment has three tasks:

  Task 1: This task has the form of a paper and pencil exercise. You may consider scanning in you hand-written solution to this task and submit it in the form of a pdf-file.

  Tasks 2 and 3: These task concern a programming problem. A solution to them should have the form of a single F# file (*file*`.fsx` or *file*`.fs`). In your solution you are allowed to introduce helper functions; but you must also provide a declaration for each of the required functions, so that it has exactly the type and effect asked for.

  You should make structural tests (that is, white-box tests) for the declared functions belonging to questions 1. and 2. for Task 2. Thus, for each declared function you must give test cases showing that every branch of the function is executed and works correctly. The expected result for each test must be stated explicitly.
- Do not use imperative features, like assignments, arrays and so on in your programs. Failure to comply with that will result in an *administrative rejection of your submission.*
- To submit you should upload two files to Inside under Assignment 1: a pdf-fil containing the solution to Task 1 and a single F# file (*file*`.fsx` or *file*`.fs`) containing the solutions to Task 2 and Task 3. The files should start with full names and study numbers for all members of the group. If the group members did not contribute equally to the solution, then the role of each student must be explicitly stated.
- Your F# solution must be a complete program that can be uploaded to F# Interactive without encountering compilation errors. Failure to comply with that will result in an *administrative rejection of your submission.*
- Be careful that you submit the right files. A submission of a wrong file will result in an *administrative rejection of your submission.*
- **DO NOT COPY solutions** from others and **DO NOT SHARE your solution** with others. Both cases are considered as fraud and will be reported.

Task 1

Consider the following declarations of the `fold` and `foldBack` functions from the textbook:

```
let rec fold f e =
   function
   | x::xs -> fold f (f e x) xs
   | [] -> e;;
val fold: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

let rec foldBack f xlst e =
   match xlst with
   | x::xs -> f x (foldBack f xs e)
   | [] -> e;;
val foldBack:('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Notice that the F# system automatically infers that `foldBack` has the type:

```
('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

1. Give a rigorous argument showing that this type is indeed the most general type of `foldBack`. Support your argument by use of type assertions of the form $e : t$ and type constraints.

Two examples using the above functions are:

```
fold (fun a x -> x - 2*a) 0 [1;2;3];;
val it : int = 3

foldBack (fun x a -> x - 2*a) [1;2;3] 0;;
val it : int = 9
```

2. Give an evaluation showing that the value of `fold (fun a x -> x-2*a) 0 [1;2;3]` is 3 and one showing that the value of `foldBack (fun x a -> x-2*a) [1;2;3] 0` is 9. Present your evaluations using $e_1 \rightsquigarrow e_2$ and include at least as many steps as there are recursive calls and applications of the anonymous functions.

Task 2

Make a solution to the bird-observation problem, presented on the next page. You are NOT allowed to use functions from the `List` library in this Task.

Task 3

You shall now solve questions 1. and 3. from the bird-observation problem using higher-order functions from the list library. Attach in a comment to the programs justifications for your choices (of library programs).

Observation of birds

We consider now *observations* of birds, where an observation consists of the *species* and the *location* and *time* of the observation. This is captured in the following type declarations:

```
type Species     = string
type Location    = string
type Time        = int
type Observation = Species * Location * Time

let os = [("Owl","L1",3); ("Sparrow","L2",4); ("Eagle","L3",5);
          ("Falcon","L2",7); ("Sparrow","L1",9);  ("Eagle","L1",14)]
```

The value `os` is a list containing 6 observations. Time points are simplified to just being integers, as we just need simple comparisons of time points in this problem.

1. Declare a function `locationsOf: Species -> Observation list -> Location list`, so that `locationsOf` *s os* gives the list of locations from observations of species *s* in *os*. It is allowed to have repeated elements in the resulting list of locations.

We would like to get an *occurrence count* or just *count* of the different species in an observation list. To this end we introduce the type

```
type Count<'a when 'a:equality> = ('a*int) list
```

An element (a_i, c_i) of an occurrence count $[(a_1, c_1); \ldots; (a_n, c_n)]$, denotes that a_i has count c_i. We assume that the a_i's in an occurrence count are all different and that $c_i > 0$.

2. Declare a function `insert` *a occ* that gives the occurrence count obtained from *occ* by incrementing the count of *a* with 1. That is, if there is no count for *a* in *occ* then $(a, 1)$ is included in the resulting occurrence count. Otherwise, there is an element (a, c) in *occ* and that element is replaced by $(a, c + 1)$ in the result. Give the type of `insert`.
3. Declare a function `toCount: Observation list -> Count<Species>` that gives the occurrence count of species in a list of observations. For example, `toCount os` is an occurrence count with 4 elements: (`"Owl"`,1), (`"Sparrow"`,2"), (`"Eagle"`,2) and (`"Falcon"`,1).

A *time interval* (type `Interval`) is a pair (t_1, t_2) of time points, where we assume below that $t_1 \leq t_2$. A time point t is in this interval when $t_1 \leq t \leq t_2$.

```
type Interval = Time * Time
```

4. Declare a function `select` f *intv os* $= [f(o_{i_1}); \ldots; f(o_{i_k})]$, where f is a function on observations, *intv* is an interval and $o_{i_1}, \ldots, o_{i_k}$ are all observations from *os* having time points in *intv*.
5. Use `select` to give an expression e for a list of pairs of species and locations from observations in `os` that are made in the time interval (4,9). That is, the value of e is a list with 4 elements: (`"Sparrow"`,`"L2"`); (`"Eagle"`,`"L3"`); (`"Falcon"`,`"L2"`); (`"Sparrow"`,`"L1"`).