DTU Compute

Lyngby 03-11-2020
Michael R. Hansen

An exercise from the exam summer 2019

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 the most general type of `f` and that `'a list -> 'b list -> ('a*'b) list` is 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. 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. The declaration of `f` is *not* tail recursive. Explain briefly why this is the case.
5. Provide a declaration of a tail-recursive variant of `f` that is based on an accumulating parameter. Your tail-recursive declaration must be based on an explicit recursion.
6. 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.