exercise08-p

December 8, 2018

0.0.1 Assignment 8.1 (P) Partial Application

Types of n-ary functions are denoted as `arg_1 -> ... -> arg_n -> ret` in OCaml.

1) Discuss, why this notation is indeed meaningful.
2) Give the types of these expressions and discuss to what they evaluate:

```
In [ ]: let a (* : todo *) = (fun a b c -> c (a + b)) 3

        let b (* : todo *) = (fun a b -> (+) b)

        let c (* : todo *) = (fun a b c -> b (c a) :: [a]) "x"

        let d (* : todo *) = (fun a b -> List.fold_left b 1 (List.map ( * ) a))

        let e (* : todo *) = (let x = List.map in x (<))
```

0.0.2 Assignment 8.2 (P) Tail Recursion

1) Check which of the following functions are tail recursive:

```
In [ ]: let rec f a = match a with [] -> a
            | x::xs -> (x+1)::f xs

        let rec g a b = if a = b then 0
            else if a < b then g (a+1) b
            else g (a-1) b

        let rec h a b c = if b then h a (not b) (c * 2)
            else if c > 1000 then a
            else h (a+2) (not b) c * 2

        let rec i a = function [] -> a
            | x::xs -> i (i (x,x) [x]) xs
```

2) Write tail recursive versions of the following functions (without changing their types):

```
In [ ]: let rec fac n = if n < 2 then 1
            else n * fac (n-1)

        let rec remove a = function [] -> []
            | x::xs -> if x = a then remove a xs else x::remove a xs

        let rec partition f l = match l with [] -> [],[]
            | x::xs -> let a,b = partition f xs in
                if f x then x::a,b else a,x::b
```

```
In [ ]: let fac n = failwith "todo"

        let remove a l = failwith "todo"

        let partition f l = failwith "todo"
```

0.0.3 Assignment 8.3 (P) Lazy Lists

Infinite data structures (e.g. lists) can be realized using the concept of **lazy evaluation**. Instead of constructing the entire data structure immediately, we only construct a small part and keep us a means to construct more on demand.

```
In [ ]: type 'a llist = Cons of 'a * (unit -> 'a llist)
```

1) Implement the function `lnat : int -> int llist` that constructs the list of all natural numbers starting at the given argument.
2) Implement the function `lfib : unit -> int llist` that constructs a list containing the Fibonacci sequence.

```
In [ ]: let lnat i = failwith "todo"

        let lfib () = failwith "todo"
```

3) Implement the function `ltake : int -> 'a llist -> 'a list` that returns the first n elements of the list.
4) Implement the function `lfilter : ('a -> bool) -> 'a llist -> 'a llist` to filter those elements from the list that do not satisfy the given predicate.

```
In [ ]: let ltake n l = failwith "todo"

        let lfilter f l = failwith "todo"
```

0.0.4 Assignment 8.4 (P) Little Helpers

Consider the following functions.

- `(%) : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`
- `(@@) : ('a -> 'b) -> 'a -> 'b`

- `(|>) : 'a -> ('a -> 'b) -> 'b`

1) Try to find their implementation just from the types:

```
In [ ]: let (%) = failwith "todo"

        let (@@) = failwith "todo"

        let (|>) = failwith "todo"
```

2) When is it possible to derive the implementation from the type?

3) Give an example where these operators could be used.