exercise08-solution-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.

An n-ary function can be considered as an unary function that returns an (n-1)-ary function. Every function with multiple arguments can thus be treated as a sequence of applications of unary functions: `arg_1 -> (arg_2 -> ... (arg_n -> ret) ... )`. In fact, a definition `let foo a b c = a + b + c` is just a more convenient way to write `let foo = fun a -> fun b -> fun c -> a + b + c`.

2) Give the types of these expressions and discuss to what they evaluate:

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

        let b : 'a -> int -> int -> int = (fun a b -> (+) b)
            (* fun a b i1 -> b + i1 *)

        let c : ('a -> string) -> (string -> 'a) -> string list
            = (fun a b c -> b (c a) :: [a]) "x"
            (* fun b c -> [b (c "x"); "x"] *)

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

        let e : 'a list -> ('a -> bool) list = (let x = List.map in x (<))
            (* fun l -> List.map (<) l *)
```

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 =
            let rec impl n acc = if n < 2 then acc
                else impl (n-1) (acc * n)
            in
            impl n 1

        let remove a l =
            let rec impl l acc =
                match l with [] -> acc
                | x::xs -> if x = a then impl xs acc
                    else impl xs (x::acc)
            in
            List.rev (impl l [])
        (* or: *)
        let remove a l = List.rev (List.fold_left
            (fun acc x -> if x = a then acc else x::acc) [] l)

        let partition f l =
            let rec impl l (a,b) =
                match l with [] -> (a,b)
                | x::xs -> impl xs (if f x then x::a,b else a,x::b)
            in
            let r = impl l ([],[]) in
            List.rev (fst r), List.rev (snd r)
```

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 rec lnat i = Cons (i, (fun () -> lnat (i + 1)))

        let lfib () =
            let rec impl a b = Cons (a, fun () -> impl b (a+b))
            in
            impl 0 1
```

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 rec ltake n (Cons (h, t)) =
            if n <= 0 then [] else h::ltake (n-1) (t ())

        let rec lfilter f (Cons (h, t)) =
            if f h then Cons (h, fun () -> lfilter f (t ()))
            else lfilter f (t ())
```

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 (%) f g x = f (g x)

        let (@@) f x = f x

        let (|>) x f = f x
```

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

Assuming we use no side-effects: For types `x` and `y`, a pure function `x -> y` has `|x| * |y|` possible implementations (where `|x|` indicates the number of values of type `x`). Example for a function `bool -> bool`:

```
In [ ]: let implementations = [
            (fun _ -> true);
            (fun _ -> false);
            (fun x -> x);
            (fun x -> not x);
        ]
```

Since a polymorphic type could be any type (e.g. also `unit`), we don't know about its values, and since we can't inspect them, we also can't match on them. Therefore, for pure functions that don't have any concrete type in the signature, there is only one possible implementation.

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

```
In [ ]: List.map (string_of_int % fst) [(1,'a'); (2,'b'); (3,'c')]
        (* instead of
        List.map (fun x -> string_of_int (fst x)) [(1,'a'); (2,'b'); (3,'c')]
        *)

In [ ]: String.concat "" @@ List.map string_of_int
            @@ List.map fst [(1,'a'); (2,'b'); (3,'c')]
        (* instead of
        String.concat "" (List.map string_of_int
            (List.map fst [(1,'a'); (2,'b'); (3,'c')]))
        *)

In [ ]: List.map fst [(1,'a'); (2,'b'); (3,'c')]
            |> List.map string_of_int |> String.concat ""
        (* instead of
        String.concat "" (List.map string_of_int
            (List.map fst [(1,'a'); (2,'b'); (3,'c')]))
        *)
```