exercise06-solution-p

November 25, 2018

0.0.1 Assignment 6.1 (P) Explicit Type Annotation

In OCaml, types are inferred automatically, so there is no need to write them down explicitly. However, types can be annotated by the programmer. Discuss:

1) Annotate all types on the expression `let f = fun x y -> x, [y]`

```
In [ ]: let f : 'a -> 'a -> ('a * 'a list) =
            fun (x : 'a) (y : 'a) : ('a * 'a list) ->
                ((x : 'a), ([(x : 'a)] : 'a list) : ('a * 'a list))
```

2) When can explicitly annotated types helpful?

Consider the function `let make_list a b = [a]::[b]` that is supposed to build a list containing `a` and `b`. Now you use the function in `let x = make_list 1 2` and your compiler (or IDE) reports an error on constant `2`:

Error: This expression has type int but an expression was expected of type int list

With type inference, type errors may appear at places that may be arbitrarily far from the location where you used the wrong (unintended) types accidentally. Especially in larger programs, this can make finding the source of the error difficult if the wrong type propagates through a number of functions before it finally conflicts with some other type somewhere. In these cases, it can be useful to annotate types on function boundaries. In the above example the annotation in `let make_list (a : 'a) (b : 'a) : 'a list) = [a]::[b]` results in a much more helpful error on `[a]`:

Error: This expression has type 'a list but an expression was expected of type 'a

A clear disadvantage, however, is that you might have to change many annotations once you change (e.g. generalize) the types of your functions.

0.0.2 Assignment 6.2 (P) Local Binding

Local (variable) bindings can be used to give names to intermediate results or helper functions inside other expressions (e.g. functions) such that they can be reused. Furthermore, they allow you to split your computation into smaller steps.

In the following expression, mark all subexpressions and for each use of a variable, decide where this variable is defined and what its value is during the evaluation of this expression.

```
In [ ]: let x (*x1*) =                                    (* x1 = [1;3;2;2;3;2;2] *)
            let f x(*x2*) y(*y2*) =                        (* x2 = 1, y2 = 2 *)
```

```
    let x(*x3*) = 2 * x(*x2*) in                    (* x3 = 2 *)
    let y(*y4*) = y(*y2*)::[x(*x3*)] in              (* y4 = [2;2] *)
    let y(*y5*) x(*x5*) =                           (* x5 = 3, y5 = fun x -> ... *)
        let y(*y6*) = x(*x5*)::y(*y4*) in            (* y6 = [3;2;2] *)
        y(*y6*) @ y(*y6*)                           (* [3;2;2;3;2;2] *)
    in
    let x(*x8*) = y(*y5*) 3 in                      (* x8 = [3;2;2;3;2;2] *)
    1::x(*x8*)                                     (* [1;3;2;2;3;2;2] *)
in
f 1 2                                              (* [1;3;2;2;3;2;2] *)
```

0.0.3 Assignment 6.3 (P) Binary Search Trees

In this assignment, a collection to organize integers shall be implemented via binary search trees.

1) Define a suitable data type for binary trees that store integers.

```
In [ ]: type tree = Node of int * tree * tree | Empty
```

2) Define a binary tree `t1` containing the values $1, 6, 8, 9, 12, 42$

```
In [ ]: let t1 = Node (9,
                  Node (6,
                    Node (1, Empty, Empty),
                    Node (8, Empty, Empty)
                  ),
                  Node (42,
                    Node (12, Empty, Empty),
                    Empty
                  )
                )
```

3) Implement a function `to_list : tree -> int list` that returns an ordered list of all values in the tree.

```
In [ ]: let rec to_list t = match t with Empty -> []
          | Node (v, l, r) -> (to_list l) @ (v :: to_list r)
```

4) Implement a function `insert : int -> tree -> tree` which inserts a value into the tree. If the value exists already, the tree is not modified.

```
In [ ]: let rec insert x t = match t with Empty -> Node (x, Empty, Empty)
          | Node (v, l, r) -> if x < v then Node (v, insert x l, r)
                              else if x > v then Node (v, l, insert x r)
                              else t
```

5) Implement a function `remove : int -> tree -> tree` to remove a value (if it exists) from the tree.

```
In [ ]: let rec combine t1 t2 = match t1, t2 with Empty, t | t, Empty -> t
            | Node (v1, l1, l2), t -> Node (v1, combine l1 l2, t)

        let rec remove x t = match t with Empty -> Empty
            | Node (v, l, r) -> if x < v then Node (v, remove x l, r)
                                else if x > v then Node (v, l, remove x r)
                                else combine l r
```

0.0.4 Assignment 6.4 (P) The List Module (part 1)

Check the documentation of the OCaml `List` module and find out what the following functions do. Then implement them yourself. Make sure your implementations have the same type. In cases where the standard functions throw exceptions, you may just `failwith "invalid"`.

```
In [ ]: let hd l = match l with [] -> failwith "invalid" | x::_ -> x
        (* or: let hd (x::xs) = x *)

        let tl l = match l with [] -> failwith "invalid" | _::xs -> xs
        (* or: let tl (x::xs) = xs *)

        let rec length = function [] -> 0 | _::xs -> 1 + length xs

        let rec append l1 l2 = match l1 with [] -> l2 | x::xs -> x::append xs l2
        (* @ *)

        let rec rev = function [] -> [] | x::xs -> (rev xs)@[x]
        (* or better: *)
        let rev l =
            let rec impl acc = function [] -> acc | x::xs -> impl (x::acc) xs
            in impl [] l

        let rec nth l n = match l with [] -> failwith "invalid"
            | x::xs -> if n <= 0 then x else nth xs (n-1)
```