CS51 **Spring 2024**

Code Review 2: HOFs, Polymorphism, Anomalous Conditions (Solutions)

1 Practice Problems

Problem 1.1 (Type Inference)**.** For each of the following definitions of a function f, give its most general type (as would be inferred by OCaml) or explain briefly why no type exists for the function.[1]

1.
```
let f x =
  let a, b = x in
  b (a 5) ;;
```

Solution
`f : (int -> 'a) * ('a -> 'b) -> 'b`
We know that `x` is a tuple, so `f` takes in an argument whose type is formed by the `*` type constructor. Let's consider the type of `a`. `a` takes in a value of type `int`, and it's ambiguous what it could return. Thus, `a` is of type `int -> 'a`. `b` takes in a value of type `'a`, since its input type must be the same as the return type of `a`. It's ambiguous what `b`, so we assign the return type another type variable `'b`. Combining all this information, we get the type of `f`.

2.
```
let rec f x a =
  match x with
 | [] -> a
 | h :: t -> h (f t a) ;;
```

Solution
`f : ('a -> 'a) list -> 'a -> 'a`
Start with `x`. Since we match `x` to the empty list or `h :: t`, then we know `x` must be some sort of list. To figure out the elements of the list, we can see that in the second case, `h` (the head element of the list) is being used as a function that takes in a single argument, which is what the recursive function `f` returns.
Since `a` is just being returned and passed in as an argument to `a` (so nothing indicates that it's a particular type or value like a function), let `a` have type `'a`. So, this means that `f` must return an element of type `'a`, which also implies that `h` takes in a single argument of type `'a` and returns a value of type `'a`.
Putting this all together, `h` has type `'a -> 'a`, so `x` must be an `'a -> 'a list`. Then given that we know the type of `'a` and the output type of `f`, we arrive to our final answer.

3.
```
let rec f x =
  match x with
  | None
  | Some 0 -> None
```

[1] Exercise 63 in the textbook provides more exercises

```
| Some y -> f (Some (y - 1)) ;;
```

Solution

`int option -> 'a option`
`x` must be an option type as we see that in the first case of the `match` statement, `x` could take on a potential value of `None`. We then know that this is an `int` option specifically, as we match `x` to the value `Some 0`. The function must also return an option, as we see that it returns `None` when `x` matches the first two cases. We then see that in the last case, the result of `f` applied on an `int option` is returned. This result, based on all the potential outputs doesn't have a specified type, so we can say that it is a `'a option` (noting that it needs to be an option since one of the possible outputs is `None`.

4.
```
let f x y =
 if x then [x]
else [not x; y] ;;
```

Solution
`bool -> bool -> bool list`
`if x` implies that `x` is a boolean, as a condition (i.e. boolean statmement) must always follow an `if` in the if statement. We then return a list, which contains `x` or `not x` as the first element. Thus, this list must a `bool list`, which also implies that `y` must be a bool (since in the else branch `y` is the second element of the list with `not x`). Putting this all together, we get the type of `f` above.

Problem 1.2 (HOF Exercises)**.** Implement the following functions.

1. Write a function `remove_duplicates` that takes in a list and returns the duplicates from the list. Write a non-recursive solution. You may use `List.sort`. You may also assume that you have the function `to_run_length` (from problem set 1).

Solution

```
let remove_duplicates (lst : 'a list) : 'a list =
  let l = lst |> List.sort Stdlib.compare |> to_run_length |>
              List.filter (fun (count, _) -> count > 1)
  in List.map (fun (_, elt) -> elt) l ;;
```

2. Write a function `tranpose` that takes a list of lists that represents a matrix (each list corresponds to a row) and returns the transpose of the matrix. For instance, the list `[[1;2;3];[4;5;6]]` represents the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

The **tranpose** of the matrix is defined to be the matrix such that the rows and columns are flipped. For instance, the transpose of the matrix above is:

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Solution

```
let rec transpose (lst : 'a list list) : 'a list list =
  match lst with
  | [] -> []
  | h :: t -> List.fold_left (fun acc x ->
                              match x with
                              | [] -> acc
                              | hd :: _ -> acc @ [hd]) [] lst ::
                              (transpose (List.filter ((!=) []) (
                                 List.map List.tl lst))) ;;
```

3. Write a polymorphic function `matrix_mult` that takes in a function to add elements two elements, a function to multiply elements, an initial value used to add and multiply elements (i.e. think of this as the zero element), and two matrices. The result should be the matrix formed by multiplying the two matrices together.

   Given an $m \times n$ matrix A and an $n \times p$ matrix B, the matrix product $C = AB$ is an $m \times p$ matrix. If the two matrices that are being multiplied don't follow these dimensions, then they can't be multiplied together. For instance, a 3×5 can be multiplied with a 5×4 matrix. However, a 3×2 cannot be multiplied with a 3×4. In other words, the number of columns in A must be equal to the number of rows in B.

   When we multiply two matrix, C_{ij} (the entry in the *ith* row and *jth* column) is defined to be the **dot product** of the *ith* row in A and *jth* column in B. So, suppose

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$B = \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$$

   Then, C is

$$C = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$

   To see how C is calculated, look at the element in the **first row** and **first column** of C, 58. To obtain 58, we take the dot product of the first row of A (`[1 2 3]` and the first column of B (`7`

9 11), which gives us:

$$\begin{bmatrix}1\\2\\3\end{bmatrix}\begin{bmatrix}7\\9\\11\end{bmatrix} = 1\cdot 7 + 2\cdot 9 + 3\cdot 11 = 7 + 18 + 33 = 58$$

For this example, here's how our function should operate in OCaml.

Listing 1: Matrix Multiplication Example

```
(* Define A and B as lists of list *)
let a = [[1;2;3]; [4;5;6]] ;;
let b = [[7;8]; [9;10]; [11;12]] ;;

matrix_multiply (+) ( * ) 0 a b ;;

(*
matrix_multiply first takes in a function to add elements of the
   matrix. Since A and B are integer matrices, we can use the built
   -in OCaml add operator as an infix operator. Similarily, for the
    second argument, which is a function to multiply elements, we
   can use the multiply operator. The zero element of the set of
   integers is 0. Then we pass in A and B.
*)

(*
The result should C should be:

C = [[58; 64]; [139; 154]]
*)
```

(a) Create two functions: one function that multiplies integer matrices and another function that multiplies float matrices.

Solution

```
let rec matrix_multiply (add : 'a -> 'a -> 'a) (mult : 'a -> 'a
    -> 'a) (init : 'a) (x : 'a list list) (y : 'a list list): 'a
    list list =
  let rec dot_product (xs : 'a list) (ys : 'a list) : 'a =
    match xs, ys with
    | [], [] -> init
    | [], _ | _, [] -> raise (Invalid_argument "invalid dot
       product")
    | hx :: tx, hy :: ty -> add (mult hx hy) (dot_product tx ty)
      in
  let b = List.length (List.hd x) in
  let c = List.length y in
```

```
let y_t = transpose y in
if b != c then raise (Invalid_argument "Incompatible dimensions
   ")
else List.fold_left (fun acc row -> acc @  [(List.fold_left (
   fun acc col -> acc @ [(dot_product row col)])   [] y_t )])
   [] x ;;
```

Problem 1.3 (Repeated Application)**.** Define a function `repeat f x n` that applies `f` to `x` exactly `n` times. You can assume that $n \geq 1$.

Solution

```
let rec repeat (f : 'a -> 'b) (x : 'a) (n : int) : 'b =
 if n = 1 then f x ;;
```

Problem 1.4 (Function Composition)**.** Using list higher order functions, define a function `compose : ('a -> 'a) list -> ('a -> 'a)` that takes a list of functions and returns the composition of those functions. For example, suppose we had a list of functions $[f;g;h]$. `compose` should return the function $f \circ g \circ h$ or $\ell(x) = f(g(h(x)))$.

Solution

```
let compose (lst : ('a -> 'a) list) : ('a -> 'a) =
  List.fold_right (fun f acc -> fun x -> f (acc x)) lst (fun x -> x)
     ;;
```

Problem 1.5 (Option and Exception Conversion)**.** In this problem, the goal is to convert between functions that return option types and functions that raise exceptions.

1. Write a function `opt_to_ext : ('a -> 'b option) -> exn -> ('a -> 'b)` that takes in a function `f` of type `'a -> 'b option` and an exception. If the input function returns None for an argument, the output function should raise the exception passed into `opt_to_ext`. If the input function returns `Some v` on an argument, the output function should return `v` for that argument.

Solution

```
let opt_to_exp (f : 'a -> 'b option) (ex : exn) : 'a -> 'b =
  fun x ->
    match f x with
    | None -> raise ex
```

```
      | Some v -> v  ;;
```

2. Write a function `ext_to_opt : ('a -> 'b) -> ('a -> 'b option)` that converts a function that (potentially) raises an exception to a function that returns a value having an option type. If the input function raises any exception on an input, the output function should return None. If the input function returns a value `v` for an input, the output function should return `Some v`.

3. In lab 4, we wrote the following functions:

4. `ext_to_opt` returns a function that returns `None` on the proper input.

Solution

```
let ext_to_opt (f : 'a -> 'b) : 'a -> 'b option =
  fun x ->
  try
   Some (f x)
  with
   _ -> None ;;
```

Listing 2: Maximum of List

```
let rec max_list_opt (lst : int list) : int option =
  match lst with
  | [] -> None
  | head :: tail ->
     match (max_list_opt tail) with
     | None -> Some head
     | Some max_tail -> Some (max head max_tail) ;;

let rec max_list (lst : int list) : int =
  match lst with
  | [] -> raise (Invalid_argument "max_list: empty list")
  | [elt] -> elt
  | head :: tail -> max head (max_list tail) ;;
```

Write 4 unit tests that adequately test the behavior of `opt_to_ext` and `ext_to_opt` using the functions above. Specifically, make sure that:

(a) `opt_to_ext` returns a function that raises the appropriate exception.

(b) The functions returned from `opt_to_ext` and `ext_to_opt` exhibit proper behavior when a non-`None` value or an exception isn't raised.