Code Review 2 Solutions: Polymorphism, Higher-Order Functions, and Anomalous Conditions

CS51
Melissa Kwan

1 Strategy for recursion: assume that your function works on smaller inputs

Let's look at the function `partition`, which is very similar to `List.filter`, except instead of only returning a list of accepted elements, it returns a tuple of two lists: the accepted elements and the rejected elements. Here's an example of its behavior:

```
partition (fun x -> x mod 2 = 0) [1; 2; 3; 4; 5]
- : int list * int list = ([2; 4], [1; 3; 5])
```

Here is how such a function might be written in an imperative paradigm, in Python:

```
def partition(pred, lst):
  acc1, acc2 = [], []
  for el in lst:
    if pred(el):
        acc1.append(el)
    else:
        acc2.append(el)
  return acc1, acc2
```

1. Write `partition` in a functional paradigm, in OCaml.

```
# SOLUTION
let rec partition (pred : 'a -> bool) (lst : 'a list) : 'a list * 'a list =
  match lst with
  | [] -> [], []
  | hd :: tl -> let accepts, rejects = partition pred tl in
    if pred hd then hd :: accepts, rejects else accepts, hd :: rejects ;;
```

2 Polymorphic type puzzle

Write functions / expressions that will evaluate to the given type.

```
('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

```
# POSSIBLE SOLUTION
let f a b c = a (b c) ;;
```

3 Currying

1. Currying
   (a) **curry** is the function that takes in an uncurried function and returns a curried function.
   (b) **uncurried** is the function that takes in an input of type **(‘a * ‘b)** and returns an output of type **‘c**.
   (c) **(curry uncurried)** is the function that takes in an input of the form **‘a -> ‘b** and returns an output of type **‘c**.
2. Uncurrying
   (a) **uncurry** is the function that takes in an curried function and returns an uncurried function.
   (b) **curried** is the function that takes in an input of the form **‘a -> ‘b** and returns an output **‘c**.
   (c) **(uncurry curried)** is the function that takes in an input of type **(‘a * ‘b)** and returns an output of type **‘c**.

```
# Curry: first pass
let curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c =
    fun uncurried ->
        fun x ->
            fun y -> uncurried (x, y) ;;

# Uncurry: first pass
let uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c  =
    fun curried ->
        fun (x, y) -> curried x y ;;

# Curry: Final Writeation
let curry uncurried x y = uncurried (x, y) ;;

# Uncurry: Final Writeation
let uncurry curried (x, y) = curried x y ;;
```

4 Map, Filter, Fold, and Other List Module Functions

```
# Map: Use when you want to apply a function or transformation to every element of a list.
let rec map (f : 'a -> 'b) (lst : 'a list) : 'b list =
 match lst with
```

```
  | [] -> []
  | hd :: tl -> (f hd) :: (map f tl) ;;

# Filter: When you want to limit a list to only include specific elements.
let rec filter f lst =
  match lst with
  | [] -> []
  | hd :: tl -> if f hd
              then hd :: filter f tl
              else filter f tl

# Fold_right: Combine a list in a way that reduces it to a single value.
let rec fold_right (f : 'a -> 'b -> 'b) (lst : 'a list) (acc : 'b) : 'b =
    match lst with
    | [] -> acc
    | hd :: tl -> f hd (fold_right f tl acc) ;;

# Fold_left: Combine a list in a way that reduces it to a single value.
let rec fold_left (f : 'b -> 'a -> 'b) (acc : 'b) (lst : 'a list) : 'b =
    match lst with
| [] -> acc
| hd :: tl -> fold_left f (f acc hd) tl ;;
```

4.1 The Relative Power of Higher-Order functions

1. Can you write `filter` using `map`? What about vice versa?

   $\boxed{\text{No and no.}}$ You can't Write `filter` using `map`, because `map` is guaranteed to return a list with the same number of elements as the original. You can't Write `map` using `filter`, because `filter` is guaranteed to return a list with the same types of elements as the original.

2. Write `square_sum`, which squares all the elements in the list and returns the sum. (Hint: which higher-order function is appropriate here?)

   ```
   # SOLUTION
   let square_sum = List.fold_left (fun acc b -> acc + b*b) 0
   ```

3. Write `partition` using `fold`.

   ```
   # SOLUTION
   let partition_left pred =
       let separate (acc1, acc2) b =
           if pred b then (acc1, acc2 @ [b]) else (acc1 @ [b], acc2) in
       List.fold_left separate ([], []) ;;
   ```

4. **Challenge:** Write `map` using either `fold_left` or `fold_right`. Why is it possible to partially apply `map_left` but not `map_right`?

   ```
   # SOLUTION
   let map_left f = List.fold_left (fun acc b -> acc @ [f b]) [] ;;
   let map_right f lst = List.fold_right (fun b acc -> (f b) :: acc) lst [] ;;
   ```

5 Options and Error Conditions

1. Write the function `all_some : 'a option list -> bool`, which returns `true` if all elements in a list of options are of type `Some x` and `false` otherwise.

```
# SOLUTION
let all_some = List.fold_left (fun acc b -> acc && b <> None) true ;;
```

2. Write the function `assoc_opt : 'a -> ('a * 'b) list -> 'b option`, which returns the value associated with key `a` in the list of pairs `l`. That is, `assoc_opt a [ ...; (a,b); ...] = b` if `(a,b)` is the leftmost binding of `a` in list `l`. Returns `None` if there is no value associated with `a` in the list `l`.

```
# SOLUTION
let rec assoc_opt key lst =
    match lst with
    | [] -> None
    | (a, b) :: tl -> if a = key then Some b else assoc_opt key tl ;;
```

6 How to Ace Style + Design for Problem Set 2

Rewrite the following functions to be as clean as possible.

(a)
```
let concat_all lst = List.fold_left (fun s1 s2 -> s1 ^ s2) "" lst ;;
```

```
# SOLUTION
let concat_all = List.fold_left (^) "" ;;
```

We take advantage of multiple opportunities for partial application.

(b)
```
let mult_51 lst = List.map (fun x -> x * 51) lst ;;
```

```
# SOLUTION
let mult_51 = List.map (( * ) 51) ;;
```

Miscellaneous tips

- In Problem Set 2, the most important question is: "do I already have the pieces I need to solve this problem?" You don't want to reinvent the wheel. In addition to using higher-order functions from the List module, you'll also want to use functions that you wrote earlier in the problem set to help you solve problems later on.
- Take advantage of opportunities for partial application!
- Problem Set 3 is a lot longer than Problem Set 2, so try to finish Problem Set 2 by Sunday.