Functional and logic programming lab 1st report

Yoshiki Fujiwara, 05-191023

1 Q.3: Fix function

1.1 an example of operation

sum_to_n 3;; → 6

is_prime 1;; → false

is_prime 12;; → false

is_prime 13;; → true

gcd 12 6;; → 6

1.2 Discussion

1.2.1 About fix function

For example, in sum_to_n, fix function behaves as follows.

fix f 3 = f (fix f) 3

= 3 + fix f 2

= 3 + 2 + f (fix f) 1

= 3 + 2 + 1 + (fix f) 0

= 3 + 2 + 1 = 6

Fix function is called as a Fixed-point-combinator. Fix function expands as

$$fixf = f(f(\ldots f(fixf)\ldots))$$

which I used this time. This function is often defined as the recursive function in functional programming language that does not support recursion.

2 Q.4: fold_right and fold_left

2.1 an example of operation

fold_right(fun x y → 2*x + y) [1;2;3] 3;; → 15
fold_left(fun x y → 2*x + y) 3 [1;2;3];; → 35

2.2 Discussion

2.2.1 Difference between tail recursion and general recursion

Tail recursion is a special case of recursion where the calling function does not compute after the recursive call. Tail recursion can compute more efficiently than the general recursion. When we have a general recursion, we have to store the return address on the call stack before jumping to the called function. This means we perhaps need a long call stack. However, when we have tail recursion, as soon as we return from the recursive call we can immediately return it. This means we do not need a call stack. Tail recursion enables us to save a space.

2.2.2 Difference between fold_right and fold_left

There are two differences between them. Obvious difference between them is an order which they combine elements from the list. The other difference is that fold_left is a tail recursive but fold_right is not. When we want to use fold_right to a very long list, we should reverse the list and use fold_left.

3 Q.5: Append and filter

3.1 an example of operation

append [1;2;3;] [5;4;3;];; → int list = [1; 2; 3; 5; 4; 3]
filter (fun x → if(x = 0) then true else false) [1;0;2];; → int list = [0]

3.2 Discussion

3.2.1 Append function

Note that we cannot put an element into a list from the right. We have to put it from the left. In order to reach append function, we use recursive function. As I mentioned, we have to put an element from the left, so it is necessarily to split the first argument and leave the second as it is.

3.2.2 Filter function

It is also can be written in recursive function. In procedual programming, we would judge the each element one by one. In functional programming, we could do in the same way. If the element satisfies the given condition, let it add the rest list. That is all the filter function did.

4 Q.6: Append function using fold_right and fold_left

4.1 an example of operation

append_left is an append function written with fold_left and append_right is an append function written with fold_right append_left [1;2;3;] [4;5;6;];; → int list = [1; 2; 3; 4; 5; 6]

append_right [1;2;3;] [4;5;6;];; → int list = [1; 2; 3; 4; 5; 6]

4.2 Discussion

4.2.1 Difference between appned_right and append_left

append_right is easier to write because there is no need to reverse the list. However, append_left is better in the long length list because fold_left uses the tail recursion which is discussed in 2.2.1. As I mentioned, tail recursion does not need a call stack. General recursion, we have to store the return address every time the function called. This means that we need a call stack whose size is linear in the depth of the recursive calls. In functional programming, we should use tail recursion when we can use it.

5 Q.7: Permutation function

5.1 an example of operation

perm [1;2;3;];; $\rightarrow$
int list list = [[1; 2; 3]; [1; 3; 2]; [2; 1; 3]; [2; 3; 1]; [3; 1; 2]; [3; 2; 1]]

5.2 Discussion

5.2.1 Function's flow (Algorithm)

1. Get an element (a) from the list
2. Generate all permutations with the rest list.
3. Put (a) on the top of the each permutations (This operation generates all the permutations that starts with (a))
4. Do 1. - 3. with all the element of the list.

5.2.2 About the function in the code

- Filter
  - This function behaves as filter function. This receives a condition and a list as an input. This function trashes the element of the list that is not satisfies the given condition. This function is used in order to create a list that does not include (a).
- filterfunc
  - This function behaves as map function that applies a given function to each element of a list. This function is used in order to realize 5.2.1.3. (put (a) on the top of the each permutations)

5.2.3 About the permutation function

In order to realize the 5.2.1.4, I used the fold_left function. Type of "acc" is a list of a list.

6 Q.8: Reverse function

6.1 an example of operation

reverse [1;2;3;];; → int list = [3; 2; 1]

reverse_right [1;2;3;];; → int list = [3; 2; 1]

6.2 Discussion

6.2.1 Reverse function

In order to acheive the reverse function, inreverse function is created. Inreverse function is a function with three steps. First, this function takes one element from the first argument. Second, it connect the element and the second argument. Third, it calls the recursive function. This function moves as follows.

1. inreverse [1;2;3;] []
2. inreverse [2;3;] [1;]
3. inreverse [3;] [2;1;]
4. inreverse [] [3;2;1;]
5. return [3;2;1;]

6.2.2 Reverse function using fold_right

It is absolutely easy to create reverse function with fold_left, because fold_left operates from the top of the list. When we use fold_right as the fold_left, we should treat the element as a function.

Let $f(x, h) = h \circ x^*$ under x^* be a function that connects x with a given list, then we gain the following equation.

$$f(x_1, (f(x_2, e^*))) = f(x_1, (e^* \circ x_2^*)) = e^* \circ x_2^* \circ x_1^*$$

Let e^* be an identity function, we could gain the reverse function. What is point is that treating the element as the function.

Plus, we should be careful that function composition of f ang g is not (f g x) in ocaml. (fun y → f y)(g x) is the correct sentense in ocaml.

7 Q.9: fold_r and fold_l

7.1 an example of operation

fold_l_using_r (fun x y → 2*x + y) 3 [1;2;3:];;→ 35

fold_r_using_l (fun x y → 2*x + y) [1;2;3;] 3;;→ 15

7.2 Discussion

The point is same as the Q.8.

let e^* be an identity function, and x^* be a function that receives e as an input and returns $f(e, x)$.

$$g(x_1, g(x_2, e^*))e = g(x_1, e^* \circ x_2^*)e = (e^* \circ x_2^* \circ x_1^*)e$$

$$= e^* \circ x_2^*(f(e, x_1)) = e^* f(f(e, x_1), x_2) = f(f(e, x_1), x_2)$$

The above equation represents the way to express fold_l woth fold_r

8 Q.10: define add, mul, sub

8.1 an example of operation

add (fun f x → f x) (fun f x → f (f x)) (fun y → y + 1) 0;; → int = 3

mul (fun f x → f x) (fun f x → f (f x)) (fun y → y + 1) 0;; → int = 2

minus (fun f x → f (f x)) (fun f x → f x) (fun y → y + 1) 0;; → int = 1

8.2 Discussion

This is called Church encoding. add and mul function is easy to write. In order to make minus function, I created the pred function. Pred function receives n as input and return n-1 as an output.

(fun $g, h \to h(gf)$) in my code runs again and again. This "h" becomes id funciton only the last time. It achieve the pred function.