Higher Order Functions

Ruxandra-Stefania Tudose

Spring Term 2024

Introduction

This report aims to open a window onto the continuation of this week's first assignment, by implementing the given methods using higher order functions and by, therefore optimising the sketched code. The pipe operator and a new function which sums up the square of a number larger than a given integer has been implemented .

General approach

When it comes to implementing the given methods using higher order functions, as far as I am concerned, I began by taking a look at the patterns identified in the basic method implementation. In other words, I identified what they have in common and what do they exactly differ in. To put it briefly, these details were related either to the base cases, to the operation applied to the list's head or to the case statement.

Implementing the *map/2*

The *map/2* method is the base of the higher order implementation for those functions whose output is a modified version of the initial list, given an indicated operation, namely: *inc/2, dec/2, mul/2, rem/2.*

```
def inc(list, n) do
   map(list, fn(x) -> x + n end)
end

def map([],_) do [] end
 def map([h|t], op) do
   [op.(h)] ++ map(t,op)
 end
```

When calling any associated functions to this patter, they themselves call *map/2* with two parameters: the list and the operation that has to be

executed, namely what makes all these functions different. Since *op* uses the dot operator, it corresponds to the passed on function to which, we apply as argument the list's head after which we append it to the recursive call of the function. Therefore, it should be taken into account that this implementation uses the stack and is in other words, not tail recursive.

Depending on the operation that has to be executed the following changes will be made to the function argument:

- decrease: fn(x) -> x - n
- multiply: fn(x) -> x * n
- remainder: fn(x) -> rem(x,n); (the division by zero situation will also be added)

Implementing the *reduce/3*

The *reduce/3* method is the base of the higher order implementation for those functions whose output is one value only, namely: *sum/1, length/1, prod/1.* What's different this time is that for each of them, we have to send an additional parameter as the accumulator (1 for the product, zero for the rest of them). Apart from that, we directly apply the arguments to the function and no longer append anything.

One new feature I would like to highlight is that *reduce/2* has a more efficient implementation thanks to its tail recursion version. The main difference is the fact that we are not doing an operation and then call the recursive step. Instead, we do the operation within recursion and therefore, always have the final/partial desired result sent as parameter.

In order to come up with this implementation, I had to go initially go back to the basic assignment and create *sum/1 and prod/1* tail recursive so that I can spot the differences and common features.

```
def sumtail(list) do sumtail(list, 0) end
def sumtail([], y) do y end
def sumtail([h|t], y) do
  sumtail(t, h + y) #the addition is an argument, part of the recursion
end

def sumt(list) do reducel(list, 0, fn(x,y) -> x + y end) end
def reducel([], acc, _) do acc end
def reducel([h|t], acc, op) do reducel(t, op.(h,acc), op) end
```

Note: In the simple tail recursion sum implementation, *sumtail/1*, I could have skipped the first code line and make the initial call with two parameters (the list and the zero accumulator), but I wanted it to be as close as possible to the non-tail recursive version.

As it can be noticed, the recursive step $reducel(t, op.(h, acc), op)$ takes the rest of the remaining list, applies to the head the desired operation depending on the function and stops once there are no more elements to add to the final sum value in our case. That is precisely when the first pattern is matched and the accumulator, namely the final result is delivered. In other words, the stack is not used in this case.

Implementing the *filter/2* and *filter/3*

The *filter/2* method is the base of the higher order implementation for those functions whose output is one value only, namely: *even/1, odd/1, div/2*. Since the *filter/2* implementation is quite similar to what has so far been presented (the only difference is that operation is done as a case and depending on the result, the element is or not added to the final list. The function argument is a case statement which returns true or false depending on the remainder. Having said that, in order to make this function tail recursive, a third parameter has to be added (initially an empty list) to which in the recursive step, the elements that satisfy the given condition will be appended.

```
def list_divl(_, 0) do :error_division_by_zero end
def list_divl(list, n) do
  filterl(list,[], fn(x) -> case(rem(x,n)) do 0 -> true; _ -> false end end)
end

def filterl([], lst, _) do lst end
def filterl([h | t], lst, op) do
 case(op.(h)) do
   true -> filterl(t, lst ++ [h], op)
   false -> filterl(t,lst,op)
 end
end
```

Note: When using the appending operator '++' the order matters, therefore, *[h] ++ lst* is different from *lst ++ [h]*. For the division, the division by zero error atom has also been included. The *evenl/1 and oddl/1* have not been included since the only difference is the output of the function's case statement.

Combining *filter/2* and *reduce/3*

Before starting to actually write the Elixir code for the sum of squares of all numbers less than a given integer, I came to the conclusion that we have to make a choice like in *filter/2*, however the result would be one value only, similar to *reduce/3*. I then, realised I would directly use the higher order technique, by combining the two. I also made use of the pipe operator in order to test it out although it doesn't necessarily make a difference given this situation.

My implementation first started with filtering out the elements, namely by creating a new list with all the elements that are less than a given n, which will then be transmitted to a method so that the summation of the squared numbers can be done.

```
def sum_square(list, n) do
 filter(list, fn(x) -> case(x < n) do true -> true; false -> false end end)
                |> sum_sq() #use pipe operator
 #alternative:
 #sum_sq(filter(list, fn(x)->case(x < n) do true -> true; false -> false end end))
end
```

The list parameter is then sent to the method below, which calls *reducer*, whoses parameters are: the just created list, the accumulator zero (for the sum) and the defined function so that we can add up the squares of the numbers.

```
def sum_sq(list) do
 reducer(list, 0, fn(x, acc) -> x*x + acc end)
end

def reducer([], acc, _) do acc end
def reducer([h | t], acc, op) do
 op.(h, reducer(t, acc, op))
end
```

Once all the elements have been 'checked', the base case is met once the list is empty and the final result is built up by accessing the stack. Looking back, I could have called the tail recursion version of *reduce/3*: the output would have been similar, while the only difference would have been that the stack was not used at all.