Functional Programming Week 5

Bohdan Tymofieienko, Mikolaj Hilgert

Table of contents

1. Higher Order functions

For the first part of this week's assignment, it was required to implement a higher order function called repeatUntil. Higher-order functions can take functions as parameters and return functions as return values. As such, we need not define a type for the parameter but it indeed can be anything. Thus ‘a’. We structure the repeatUntil function in a way where an operation is repeated until a given predicate holds.

```
repeatUntil: (a -> Bool) -> (a -> a) -> a -> a
repeatUntil predicate operation input =
    if (predicate input) then
        input
    else
        repeatUntil predicate operation (operation input)
```

An example of a predicate is the following: Above val function.

```
aboveVal: Int -> Int -> Bool
aboveVal val x =
    x > val
```

For example, we are tasked to show the try: repeatUntil above100 ((+) 1) 42. In this case the +1 action will be repeated until the value 101 as at that point the above100 predicate will be True.

The next significant part of the assignment was the implementation for the Collatz conjecture using this aforementioned repeatUntil function. Namely for this we had to create a new predicate and operation.

For the predicate, the terminating condition to satisfy the conjecture is if we arrive at the number one. Thus using pattern matching we check if the head of the list is 1. If it is then we return True, else False.

```
contains1: List Int -> Bool
contains1 list =
    case list of
        [] -> False
        (x :: xs) -> x == 1
```

Next, we had to define the operation for the Collatz. We also have to keep track of all the previous values. So pattern matching and list concatenation was the perfect fit. As based on if the most recent value was even or odd- based on that we would calculate and concatenate it to the list of all previous values in the Collatz conjecture.

```
myCollatz: List Int -> List Int
myCollatz list =
    case list of
        [] -> []
        (x :: xs) ->
            if (modBy 2 x == 0) then
                x//2 :: x :: xs
            else
                1 + (3*x) :: x :: xs
```

2. Mathematical functions. Part 2

This week we expanded our solution from last week and new functions. In particular we worked on differentiation rules. We implemented all the basic rules such as addition, subtraction, multiplication and quotient rule. We also added the chain rule to make our solution complete. We added simplification rules to make the resulting expressions more human-readable.

Constant:	$\frac{d}{dx}(c) = 0$
Sum:	$\frac{d}{dx}(u + v) = \frac{du}{dx} + \frac{dv}{dx}$
Difference:	$\frac{d}{dx}(u - v) = \frac{du}{dx} - \frac{dv}{dx}$
Constant Multiple:	$\frac{d}{dx}(cu) = c\frac{du}{dx}$
Product:	$\frac{d}{dx}(uv) = u\frac{dv}{dx} + v\frac{du}{dx}$
Quotient:	$\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{v\frac{du}{dx} - u\frac{dv}{dx}}{v^2}$
Power:	$\frac{d}{dx}x^n = nx^{n-1}$
Chain Rule:	$\frac{d}{dx}(f(g(x)) = f'(g(x)) \cdot g'(x)$

We used pattern matching and recursion in our solution to differentiate the function.

You could observe from the code snippet below that the approach is to identify under which rule a given function falls and then substitute the elements of the given function to the new function according to the rule. Chain rule states that derivative of compound function is a derivative of “outside” function multiplied by derivative of “inside” function, which is applied for “Poly” function that represents function to the power.

```
derivative: Function -> Function
derivative func =
    case func of
         Poly base power -> Mult (Mult (Const power) (Poly base (power -
1))) (derivative base)
          Mult left right -> Plus (Mult (derivative left) right) (Mult
left (derivative right))
           Div top bottom -> Div (Minus (Mult bottom (derivative top))
(Mult top (derivative bottom))) (Poly bottom 2)
        Plus left right -> Plus (derivative left) (derivative right)
        Minus left right-> Minus (derivative left) (derivative right)
        Const value -> Const 0
        X -> Const 1
```

The base case is “Const” or “X”. In that case function cannot be differentiated further. However, you may guess that the resulting function would contain redundant constants and operations such as “x + 0”. This is due to the procedure of differentiation, when the function can no longer be simplified adding a constant of 0 would be a valid part of the process.

To get rid of that and as a result improve readability of an output we made a function “simplify” that removes such unnecessary elements of the function. Again we applied pattern matching to first identify what kind of function we are dealing with and then apply the operation needed. We also simplify addition of two constants by verifying what is the type of parameter. It is important to apply simplification for the “if” statement to avoid skipping simplification of inner parts of the functions. Obviously we have it implemented for all the operations (multiplication, division etc.). It follows the same logic and thus we decided not to include it in this document.

```
simplify: Function -> Function
simplify func =
    case func of
        Plus (Const left) (Const right) -> Const(left+right)
        Plus left right ->
            if ( simplify left ) == Const 0 then
                simplify right
            else if (simplify right) == Const 0 then
                simplify left
            else
                Plus (simplify left) (simplify right)

        Minus (Const left) (Const right) -> Const(left-right)
        Minus left right ->
            if ( simplify left ) == Const 0 then
                simplify right
            else if (simplify right) == Const 0 then
                simplify left
            else
                Minus (simplify left) (simplify right)
```