FUNCTIONAL PROGRAMMING MT2018

SHEET 1

GABRIEL MOISE

1.

1.1

1. (a plus (f x)) + ((x times y)*z) *(The functions plus and times are predefined to be the addition and multiplication?)*

2. 3 4 + 5 + 6 results in an error *(Shouldn't there be a sign between 3 and 4 for it to work?)*

3. (2^(2^(2^2))) = 65536

1.2 Suppose we have f, g, h functions and we want to show that f . (g . h) = (f . g) . h

First of all, we will have to specify the domains and codomains of each functions, so that we respect the rules of composition. So, f :: C -> D, g :: B -> C and h :: A -> B, where A, B, C, D are sets. Both (f . g) . h and f . (g . h) have the domain A and the codomain D, so we can check if they are indeed equal.

Let's take an arbitrary x from A. We will replace h(x) with y, with y from B. If we want to compose g and h, we will have g . h (x) = g (y) and we will replace g(y) with z, z from C. Now we will have:

f . (g . h) (x) = f . (g (h (x))) = f . (g (y)) = f (z) and

(f . g) . h (x) = (f . g) (y) = f (g (y)) = f (z)

So, f . (g . h) (x) = (f . g) . h (x) for every x from A, so we conclude that function composition is associative.

1.3 Suppose we have as = [a1,a2,…,an], bs = [b1,b2,…,bm] and cs = [c1,c2,…,cp].

Then, as ++ (bs ++ cs) = as ++ [b1,b2,…,bm,c1,c2,…,cp] = [a1,a2,…,an,b1,b2,…,bm,c1,c2,…,cp]

And (as ++ bs) ++ cs = [a1,a2,…,an,b1,b2,..bm] ++ cs = [a1,a2,…,an,b1,b2,…,bm,c1,c2,…,cp].

So, the ++ operation is associative.

However, the function is not also commutative because:

as ++ bs = [a1,a2,…,an] ++ [b1,b2,…,bm] = [a1,a2,…,an,b1,b2,…,bm] and

bs ++ as = [b1,b2,…,bm] ++ [a1,a2,…,an] = [b1,b2,…,bm,a1,a2,…,an] and these two lists are not equal for all a1,a2,…,an,b1,b2,…,bm.

The unit element in this case is e = [] because when we do as ++ e, or e ++ as, we obtain [a1,a2,…,an], which is in fact as.

Let's suppose that we have a zero element z = [z1,z2,…,zl]. From the properties of the zero element, we know that for all as = [a1,a2,…,an] we have as ++ z = z ++ as = z. So, we have that

[a1,a2,…,an,z1,z2,…,zl] = [z1,z2,…,zl,a1,a2,…,an] = [z1,z2,…,zl].

As we cannot have two equal lists with a different length, we need n+l=l, so we need n=0. However, we said that z is a zero element, so the property has to happen for all as, so for all possible lengths, which means n can also be >0 and this gets us to a contradiction. To sum up, the ++ operation has no zero element.

1.4 Starting from the fact that map takes a function and a list and applies that function to every element of the list, we can deduce that map double [3,7,4,2] will return [6,14,8,4], as we doubled every element from the initial list.

By applying map (double . double) [3,7,4,2], we change the double function into a composed one, let's call it quadruple :: Integer -> Integer, and it is the composition of two double functions. So, quadruple x = double (double x) = double (2*x) = 4*x. Then, map (double . double) [3,7,4,2] = map quadruple [3,7,4,2] = [12,28,16,8].

By applying map double [], the program will return [] because all it had to do was to double every element from [] and to create a list from the results. As we have no results to add in the list, which initially was [], the program will return [].

2. For the first equality sum . map double = double . sum, we will consider the argument to be [a1,a2,…,an], because both functions need a list as argument. By doubling the list we obtain [2*a1,2*a2,…,2*an], and after summing its elements we obtain 2*a1+2*a2+…+2*an. So, given a list [a1,a2,…,an], the function from the left returns 2*a1+2*a2+…+2*an. The second function sums all the elements of the list and then multiplies the result by 2, so we get 2*(a1+a2+…+an), which is also equal to 2*a1+2*a2+…+2*an. So, both functions return the same result given a random list. They also have the same domain [Integer] and codomain Integer, so they are equal.

For the second equality we need a list whose elements are also lists as the concat function takes the argument and creates a list with all the elements from the lists. The obtained list is needed for the sum function. The result is an Integer, so the domain of this function is [[Integer]] and the codomain is Integer. For the function from the left, we apply map sum so the result is a list and then we apply sum to it.

So, let's say we have [[a11,a12,…,a1n1],[a21,a22,…,a2n2],…,[am1,am2,…,amnm]] as argument for both functions.

If we apply sum.map sum to it we will transform it in sum [a11+a12+…+a1n1, a21+a22+…+a2n2, … , am1+am2+…+amnm]. Then, the result of this will be the sum of all the elements from the current list, which is (a11+a12+…+a1n1) + (a21+a22+…+a2n2) + … + (am1+am2+…+amnm).

If we apply sum.concat to the initial list we will obtain

sum [a11,a12,…,a1n1,a21,a22,…,a2n2,…,am1,am2,…,amnm]

which will then result in a11+a12+…+a1n1+ a21+a22+…+a2n2+…+am1+am2+…+amnm. As adding is associative and commutative, both functions give the same result for the same input. So, we can conclude that they're equal because they also have the same domain [[Integer]] and codomain Integer.

For the third equality we need a list as argument, let's say [a1,a2,…,an]. If we sort the list, the elements from it will only change their positions in the list depending on the order, but their sum remains the same.

As the domain and codomain of sum.sort and sum are [Integer] and Integer, respectively, we can conclude that these functions are equal, as they both return a1+a2+…+an.

2.1

```
factorial :: Integer -> Integer

factorial n

  | n<0 = error "Can't use this function for negative values!"

  | n==0 = 1

  | otherwise = product [1..n]
```

This way, we make sure we get an error if we try to calculate the factorial of a negative value, and the rest of the cases are respecting the definition of factorial.

The formula for the number of possibilities of choosing r objects from n is (n!) / ((r!) * ((n-r)!)). So, we will have the function:

```
choose :: Integer -> Integer -> Integer

choose n r = div (factorial n) ((factorial r) * (factorial (n-r)))
```

Let's now imagine we toss a coin n times. Every toss can be a Heads or a Tails. In total we have 2^n possible configurations for the sequence of results. Every configuration has a number of k Heads and (n-k) Tails, where 0<=k<=n. The 2^n configurations can be divided into n groups, each group k having configurations that have exactly k Heads and (n-k) Tails. For a group k, the number of configurations it has is equal to choose n k. So, the sum from r=0 to n of choose n r is always going to be equal to 2^n, so the function check will always return True, for any positive value of n (because that's how we declared factorial in order for it to work).

```
check :: Integer -> Bool

check n = if sum [choose n r | r <- [0..n]]  == 2^n then True

                                                  else False
```

2.2

```
not1 :: Bool -> Bool

not1 True = False

not1 False = True
```

~~~~~~~~~~~~~~~~~~~~~~~~

```
not2 :: Bool -> Bool

not2 x
```
~~~~~~~~~~~~~~~~~~~~~~~~

```
    | x == True = False
    | otherwise = True
~~~~~~~~~~~~~~~~~~~~~~~~
not3 :: Bool -> Bool
not3 x = if x==True then False
                    else True
~~~~~~~~~~~~~~~~~~~~~~~~
not4 :: Bool -> Bool
not4 x = case x of True -> False
                   False -> True
```

2.3 For every Bool type we can have up to 2 possible values (True and False), and for a domain containing n elements and a codomain containing m elements, we can have up to m^n possible functions from the domain to the codomain as for each x element from the domain we can have m possible values y from the codomain.

1. Bool -> Bool

The domain has 2 elements, the codomain has 2, so the number of functions is $2^2=4$.

2. Bool -> Bool -> Bool

This is equivalent to Bool -> (Bool -> Bool).

So, the function gets an argument and returns a function like the one from 1. Then, the domain is formed of 2 elements, and the codomain is formed of 4 elements, so the number of functions is $4^2=16$.

3. Bool -> Bool -> Bool -> Bool

This is equivalent to Bool -> (Bool -> (Bool -> Bool)).

So, this function gets an argument and returns a function like the one from case 2. Then, the domain is formed of 2 elements, and the codomain is formed of 16 elements, so the number of functions is $16^2=256$.

4. (Bool, Bool)

This type is a pair, so there is a number of 4 possibilities (two for each Bool) to form a pair using True and False.

5. (Bool, Bool) -> Bool

The domain has 4 elements and the codomain has 2, so the number of possible functions is $2^4=16$.

6. (Bool, Bool, Bool)

This type is a triplet, so there is a number of 8 possible triplets, because every Bool can be True or False.

7. (Bool, Bool, Bool) -> Bool

The domain is formed of 8 elements, and the codomain has 2 elements, so the number of possible functions is 2^8=256.

8. (Bool -> Bool) -> Bool

This function takes as argument a function of type 1. and returns a Bool value, so the domain is formed of 4 elements, and the codomain is formed of 2 elements, so the number of functions is 2^4=16.

9. (Bool -> Bool -> Bool) -> Bool

This is equivalent to (Bool -> (Bool -> Bool)) -> Bool. So, this function takes a function of type 2. as argument and returns a Bool value. The domain has 16 elements, and the codomain has 2, so the number of functions is 2^{16}=65536.

10. ((Bool -> Bool) -> Bool) -> Bool

The function takes as argument a function of type 8. and returns a Bool value. The domain has 16 elements and the codomain has 2, so the number of functions is 2^{16}=65536.