assignment01

May 8, 2019

1 Functional Programming SS19

2 Assignment 01 Solutions

2.1 Exercise 1

2.1.1 1a)

Give examples of Haskell function declarations with the following types and briefly explain their semantics. Your solutions must not ignore any of their arguments completely.

1. `Bool -> Bool -> Int`
2. `[Int] -> [Bool] -> Int`
3. `[Bool] -> (Bool -> Int) -> [Int]`

1. A simple example of a function with this signature would be one that converts the result of && to an integer.

```
andToInt :: Bool -> Bool -> Int
andToInt x y | x && y    = 1
             | otherwise = 0
```

2. An example of such a function would be one that sums up the values in the integer list for which the corresponding entries in the Boolean list are set to True. This could be used for filtering out data (e.g. sensor measurements).

```
sumValid :: [Int] -> [Bool] -> Int
sumValid [] [] = 0
sumValid x []  = 0
sumValid [] x  = 0
sumValid (x : xs) (y : ys) | y == True = x + sumValid xs ys
                           | otherwise = sumValid xs ys
```

3. A definition of a function with this signature is given below:

```
boolListToIntList :: [Bool] -> (Bool -> Int) -> [Int]
boolListToIntList xs f = [ f x | x <- xs ]
```

where a list comprehension is used for simplicity.

We could for instance use this function to obtain a list of zeros and ones from a list of Booleans, which could be used in a predicate learning scenario (depending on the learning algorithm, numeric rather than Boolean vectors may be required).

2.1.2 1b)

Suppose that `f` has the type `Bool -> [Int] -> Int`. What is the type of `\x y -> f ((f True x)>0) [y]`?

The type of the given lambda function is `[Int] -> Int -> Int`. We can find this by pattern matching on the right-hand side of the function. Since we know the type of `f`, it is clear that `x` is of type `[Int]`; otherwise, it would not be possible to apply the innermost `f`. `y` then has to be of type `Int`, otherwise the application of the outermost `f` would not be possible. Since `f` returns an `Int` value, the return type of the lambda function is also `Int`.

2.2 Exercise 2

2.2.1 2a)

For each of the following equations, if possible, give pairwise different example values for x, y, and z such that the equation holds. Otherwise explain why such an assignment is not possible.

1. `[[x],[y]] == [y]:z`
2. `([x] ++ z):y == (x:z):y`
3. `[[]] ++ ([x]:y) == ([x]:z)`
4. `(x:y):z == (y ++ [x]):z`

The operator ++ concatenates two lists. For example: `[1, 2, 3] ++ [2, 3] = [1, 2, 3, 2, 3]`.

2a1) In this case, the values of `x` and `y` on the left- and right-hand side respectively have to be equal; this however violates the given constraint of pairwise different examples, so there is no possible assignment that would satisfy this equation.

2a2) In this equation, we need the expressions in parentheses to be of type `a` and `y` to be of type `[a]`. An assignment that satisfies the equation would thus be

```
x = [1]
y = [[[2]]]
z = []
```

since in this case, `y` has the type `[[[Int]]]`, while the expressions in parentheses on both sides have the type `[[Int]]`, which matches the type of `y`'s elements.

```
In [1]: x = [1]
        y = [[[2]]]
        z = []
        ([x] ++ z):y == (x:z):y

Line 1: Use :
Found:
[x] ++ z
Why not:
x : z


True
```

2a3) This expression requires the following equality to hold

```
[[], [x], ...] == [[x], ...]
```

for which there is no possible assignment.

2a4) For this equation to hold, we need `x : y == y ++ [x]`; `x = [], y = [[]]` satisfies this equality since we obtain the list `[[], []]` on both sides, which is of type `[[a]]`. `z` then needs to be of type `[[[a]]]` for the overall equality to hold. An assignment that satisfies the equality is thus

```
x = []
y = [[]]
z = [[[1]]]
```

```
In [2]: x = []
        y = [[]]
        z = [[[1]]]
        (x:y):z == (y ++ [x]):z
```

```
True
```

2.2.2 2b)

Consider the following patterns:

`p1) ([x]++y):ys`

`p2) (x:y)++ys`

and the following terms:

`t1) [[]]`

`t2) [[1,2],[3]]`

For each pair of a pattern and a term, indicate whether the pattern matches the term. If so, provide the appropriate matching substitution. Otherwise, explain why the pattern does not match the term. Does there exist a term that is matched by `p1` but not by `p2`? Justify your answer.

2b1) In this pattern, `ys/[]` is the only possible substitution for `ys`; however, we also need `[x]++y == []`, for which there is no matching substitution. As a result, `t1` does not match `p1`.

2b2) `t2` matches `p1`; the matching substitution is $\sigma = [x/1, y/[2], ys/[[3]]]$.

```
In [3]: x = 1
        y = [2]
        ys = [[3]]
        ([x]++y):ys
```

```
Line 1: Use :
Found:
[x] ++ y
Why not:
x : y
```

```
[[1,2],[3]]
```

2b3) `t1` matches `p2`; the matching substitution is $\sigma = [x/[], y = [], ys = []]$.

```
In [4]: x = []
        y = []
        ys = []
        (x:y) ++ ys
```

```
[[]]
```

2b4) `t2` matches `p2`; the matching substitution is $\sigma = [x/[1,2], y/[], ys/[[3]]]$.

```
In [5]: x = [1,2]
        y = []
        ys = [[3]]
        (x:y) ++ ys
```

```
[[1,2],[3]]
```

2.3 Exercise 3

2.3.1 3a)

Write a Haskell–function `myrem`, where `myrem x y` is the remainder of the integer division when dividing x by y. So for example, `myrem 14 3 == 2`. If `y == 0` then `myrem x 0 == x`. If `y < 0` then `myrem x y == myrem x (-y)`.

```
myrem :: Int -> Int -> Int.
```

You may not use any predefined functions except comparisons, +, and -.

```
In [6]: normalise :: Int -> Int -> Int
        normalise x y | x < y  = x
                      | x >= y = normalise (x - y) y

        myrem :: Int -> Int -> Int
        myrem x 0 = x
        myrem x y | y < 0 = myrem x (-y)
                  | y > 0 = normalise x y
```

Test cases

```
In [7]: myrem 14 3
```

```
2
```

```
In [8]: myrem 10 0
```

```
10
```

```
In [9]: myrem 14 (-3)
```

```
2
```

2.3.2 3b)

Write a Haskell-function `count` that given a list `xs` and an element `x` returns the number of occurences of `x` in `xs`. E.g., `count 2 [0,2,2,0,2,5,0,2] == 4` wheras `count (-7) [0,2,2,0,2,5,0,2] == 0`.

```
count :: Int -> [ Int ] -> Int
```

You may not use any predefined functions except comparisons and +.

```
In [10]: count :: Int -> [Int] -> Int
         count x [] = 0
         count x (y : ys) | x == y = 1 + count x ys
                          | x /= y = count x ys
```

Test cases

```
In [11]: count 2 [0,2,2,0,2,5,0,2]
```

```
4
```

```
In [12]: count (-7) [0,2,2,0,2,5,0,2]
```

```
0
```

2.3.3 3c)

Write a Haskell-function `simplify` that given a list `xs` returns a list of pairs as follows. The resulting list contains the pair `(x,n)` if and only if `x` occurs in `xs` `n` times and `n > 0`. E.g., `simplify [0,2,2,0,2,5,0,2] == [(0,3),(2,4),(5,1)]`.

```
simplify :: [ Int ] -> [( Int , Int )]
```

You may not use any predefined functions except comparisons.

```
In [13]: {- |
           Counts the number of instances of a given number in a list.
           Implemented in exercise 3a).
         -}
         count :: Int -> [Int] -> Int
```

```haskell
count x [] = 0
count x (y : ys) | x == y = 1 + count x ys
                 | x /= y = count x ys

{- |
  Returns the first element in a tuple
-}
first :: (Int, Int) -> Int
first (x, y) = x

{- |
  Given a list of pairs (x, y), returns True if a given number
  is the first element of a pair and False otherwise.
-}
contained :: Int -> [(Int, Int)] -> Bool
contained x [] = False
contained x (y : ys) | x == y1 = True
                     | x /= y1 = contained x ys
                     where y1 = first y

{- |
  Utility function for simplify, i.e. given a list of numbers x,
  returns a list of (m, n) tuples in which m is a number
  and n is the number of times m occurs in x. This function avoids
  duplicate entries in the calculation, i.e. the resulting
  list includes a tuple containing m exactly once.
-}
countUnique :: [Int] -> [(Int, Int)] -> [(Int, Int)]
countUnique [] ys = []
countUnique (x : xs) ys | xcounted == True = z
                        | xcounted == False = (x, count x (x : xs))
                                              : countUnique xs w
                        where xcounted = contained x ys
                              z = countUnique xs ys
                              w = (x, count x (x : xs)) : ys
{- |
 Given a list of numbers x, returns a list of (m, n) tuples
 in which m is a number and n is the number of times m occurs in x.
-}
simplify :: [Int] -> [(Int, Int)]
simplify [] = []
simplify (x : xs) = countUnique (x : xs) []
```

Test cases

```
In [14]: simplify [0,2,2,0,2,5,0,2]
```

```
[(0,3),(2,4),(5,1)]
```

2.3.4 3d)

Write a Haskell-function `multUnion` that given two lists of pairs `xs` and `ys` concatenates these lists where each "multiple occurence" is simplified as follows: If `xs` contains a pair `(x,n)` and `ys` contains `(x,m)`, then the result contains `(x,n+m)`. You may assume that in both `xs` and `ys` an integer occurs at most once as first entry of a pair. Moreover, assume that the lists are sorted in ascending order w.r.t. the first entry of the pair. Make sure that the resulting list is sorted in ascending order w.r.t. the first entry of the pair as well. E.g., `multUnion[(0,3),(2,4),(5,1)] [(-1,1),(0,4)] == [(-1,1),(0,7),(2,4),(5,1)]`.

```
multUnion :: [( Int , Int )] -> [( Int , Int )] -> [( Int , Int )]
```

You may not use any predefined functions except comparisons and +.

```
In [15]: multUnion :: [(Int, Int)] -> [(Int, Int)] -> [(Int, Int)]
         multUnion [] [] = []
         multUnion x [] = x
         multUnion [] x = x
         multUnion (x : xs) (y : ys) | m == n = (m, k + l) : multUnion xs ys
                                     | m < n  = (m, k) : multUnion xs (y : ys)
                                     | m > n  = multUnion (y : ys) (x : xs)
                                    where (m, k) = x
                                          (n, l) = y
```

Test cases

```
In [16]: multUnion[(0,3),(2,4),(5,1)] [(-1,1),(0,4)]

[(-1,1),(0,7),(2,4),(5,1)]


In [17]: multUnion[(-1,1),(0,3),(5,1)] [(0,4),(2,4)]

[(-1,1),(0,7),(2,4),(5,1)]
```

2.4 Exercise 4

Define a Haskell function `^^^` in infix notation with the type declaration

```
(^^^) :: [Int] -> [Int] -> Int
```

such that the following holds for lists of equal length:

- The function call `xs ^^^ ys` evaluates to `xs` to the power of `ys` interpreted as vectors, where the negative entries of `ys` are ignored. In other words, `[x1, x2, ..., xn] ^^^ [y1, y2, ..., yn] == x1 ^ y1 * x2 ^ y2 * ... * xn ^ yn`. For example `[1, 4, 5] ^^^ [7, 2, 3]` evaluates to `1 ^ 7 * 4 ^ 2 * 5 ^ 3 == 2000` and `[1, 4, 5] ^^^ [5, -1, 0]` evaluates to `1 ^ 5 * 5 ^ 0 == 1`.
- `xs ^^^ ys * z`, where `xs` and `ys` have type `[Int]` and `z` has type `Int`, is a valid expression.

The function `^^^` may behave arbitrarily if the two arguments have different lengths. You may not use any predefined functions except *, ^, and comparisons. You may, of course, use constructors like [] and :. You can get one bonus point if you solve the exercise even without using the predefined function ^.

Hints:

- The binding priority of * is 7.
- The empty product defaults to 1, i.e. `[] ^^^ [] == 1`.
- Note that `0 ^ 0 == 1`.

```
In [18]: pow :: Int -> Int -> Int
         pow x 0 = 1
         pow x y = x * pow x (y - 1)

         (^^^) :: [Int] -> [Int] -> Int
         (^^^) [] []       = 1
         (^^^) (x : xs) [] = 1
         (^^^) [] (x : xs) = 1
         (^^^) (x : xs) (y : ys) | y < 0  = xs ^^^ ys
                                 | y >= 0 = (pow x y) * (xs ^^^ ys)
         infixl 7 ^^^
```

Test cases

```
In [19]: [1, 4, 5] ^^^ [7, 2, 3]
```

2000

```
In [20]: [1, 4, 5] ^^^ [5, -1, 0]
```

1

```
In [21]: [1, 4, 5, 3] ^^^ [5, -1, 0, 3]
```

27

```
In [22]: [4, 1, 5] ^^^ [-1, 5, 0]
```

1

```
In [23]: [2, 4, 5, 3] ^^^ [5, -1, 0]
```

32

```
In [24]: [2, 4, 5] ^^^ [5, -1, 0, 10]
```

```
32
```

```
In [25]: [1, 4, 5, 3] ^^^ [5, -1, 0, 3] * 4
```

```
108
```