[20231109] INFOFP - Functioneel programmeren - 1 - USP

Cursus: BETA-INFOFP Functioneel programmeren (INFOFP)

Tijdsduur: 2 uur en 30 minuten

Aantal vragen: 5

[20231109] INFOFP - Functioneel programmeren - 1 - USP

Cursus: Functioneel programmeren (INFOFP)

Aantal vragen: 5

1 In this question, we test your knowledge of equational reasoning about functional programs.

Using the following definitions

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z                            -- a
foldr f z (x:xs) = f x (foldr f z xs)       -- b

(++) :: [a] -> [a] -> [a]
[] ++ ys = ys                               -- c
(x:xs) ++ ys = x : (xs ++ ys)               -- d

swap :: (a, b) -> (b, a)
swap (a, b) = (b, a)                        -- e

flip :: (a -> b -> c) -> b -> a -> c
flip f b a = f a b                          -- f

curry :: ((a, b) -> c) -> (a -> b -> c)
curry f a b = f (a, b)                      -- g

(.) :: (b -> c) -> (a -> b) -> (a -> c)
(.) f g a = f (g a)                         -- h
```

please prove the following claims, where you justify every reasoning step by marking it with the letter (a - h) of some definition or by marking it as I.H. to refer to an induction hypothesis. Please clearly state any induction hypotheses you use.

a. [6pt] Claim:

```
curry (f . swap) = flip (curry f)
```

6 pt. **a.**

b. [8pt] Claim:

```
foldr f e (xs ++ ys) = foldr f (foldr f e ys) xs
```

8 pt. **b.**

2 In this question, we test your knowledge of lazy evaluation.

a. [5pt] Indicate, for each of the following expressions what their WHNF is. If the expression is already in WHNF, please copy the original expression. If the expression crashes in its evaluation to WHNF, please write "undefined".

```
filter even [1, undefined, 2]
```

a. ..... (1 pt.)

```
tail (filter odd [1, 1, undefined, 2])
```

b. ..... (1 pt.)

```
foldl (\(a,as) xs -> (length xs + a, xs ++ as) ) (0,[]) [[1]]
```

c. ..... (2 pt.)

```
Node (Node Leaf 5 Leaf) (3 + 5) Leaf
```

d. ..... (1 pt.)

in the above

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

b. [2pt] The function

```
const :: a -> b -> a
const a b = a
```

is strict in its first argument and non-strict in its second argument.
Write a function

```
const' :: a -> b -> a
```

that calculates the same value as const but is strict in both its arguments.

```
const' a b = e. ..... (2 pt.)
```

3 In this question, we test your knowledge of specification and testing.

Consider the function

```
gather :: Int -> (Int -> Int) -> [a] -> [a]
```

that reindexes a list based on a desired length of the output list and a function that remaps indices from positions in the output list to positions in the input list. For example,

```
> gather 5 (\n -> n * n) "Never gonna let you down."
"Nerny"
```

a. [3pt] Please implement gather without using direct recursion.

```
gather size f xs = a. ..........................................(3 pt.)
```

Hint: you may want to use the function

```
(!!) :: [a] -> Int -> a
(x:xs) !! n | n == 0 = x
            | otherwise = xs !! (n - 1)
```

b. [5pt] Please write a specification 'spec_gather' that can be used to check that 'gather n f xs' is a sublist of 'xs'. That is, check that the elements of 'gather n f xs' appear in the same order in 'xs'. Note that the sublist does not have to be contiguous.

```
spec_gather :: Eq a => Int -> (Int -> Int) -> [a] -> Bool
```

5 pt. **b.**

4 In this question, we consider a datatype that you have not seen before and test your knowledge of Functors, IO, and Monads.

Consider the following type to represent discrete probability distributions over a, in the form of value-weight pairs.

```
newtype DiscProb a = MkProb {probs :: [(a, Double)]} deriving (Show, Eq)
```

For example, we can represent the following common probability distributions using the type:

```
-- 0 <= p <= 1 is a probability
bernoulli :: Double -> DiscProb Int -- a biased coin flip that gives 1 with
probability p and 0 with probability 1 - p
bernoulli p = MkProb [(0, 1.0 - p), (1, p)]

-- ps is a list of probabilities that needs to sum up to 1
categorical :: [Double] -> DiscProb Int -- a distribution that assigns
probability ps !! k to the number k
categorical ps = MkProb $ zip [0..] ps
```

a. [3pt] Give a functor instance for 'DiscProb'

3 pt. **a.**

b. [4pt] Observe that our distributions 'd' are not guaranteed to be normalized (i.e. the weights may not sum up to one: 'totalWeight d' does not need to be one).
Complete this definition of 'totalWeight', which sums all the weights of a distribution.

```
totalWeight :: DiscProb a -> Double
totalWeight d = foldr (  b.  .....(2 pt.) )
                      (  c.  .....(1 pt.) )
                      (  d.  .....(1 pt.) )
```

c. [3pt] Observe that our representation of distributions may contain duplicate elements. This is uninteresting information when we think of them as distributions. For example,

```
MkProb [(True, 0.3), (False, 0.5), (True, 0.2)]
```

and

```
bernoulli 0.5 = [(False, 0.5), (True, 0.5)]
```

represent the same probability distribution. We can write a function

```
collect :: Eq a => DiscProb a -> DiscProb a
```

that collects elements of type 'a' into buckets and adds their weights to create a new 'DiscProb a' that represents the same distribution but without duplicate 'a's. For example,

```
> collect (MkProb [(True, 0.3), (False, 0.5), (True, 0.2)])
MkProb [(False, 0.5), (True, 0.5)]
```

Please complete this definition of 'collect':

```
collect (MkProb aws) = MkProb (foldr insert [] aws) where
 insert (a, w) [] = e.  ..........(1 pt.)
 insert (a, w) ((a', w') : rest) | a == a' = f.  ..........(1 pt.)
                                 | otherwise = g.  ..........(1 pt.)
```

d. [3pt] Consider the following helper function that selects an element from a weighted list (where we assume the weights sum to 1.0) based on a given number r between 0.0 and 1.0.

```
select :: [(a, Double)] -> Double -> a
select [] r = error "Empty list"
select ((x, p) : xs) r
  | r < p = x
  | otherwise = select xs (r - p)
```

Using select and the operation

```
randomRIO :: (Double, Double) -> IO Double
```

to draw a random number between two bounds (e.g. 'randomRIO (4.0, 6.0)' draws a random number between '4.0' and '6.0', from a uniform distribution), please define a function

```
sample :: DiscProb a -> IO a
```

such that 'sample d' actually performs a random draw from a normalized distribution 'd' (for which the weights sum up to 1.0).

3 pt. **h.**

(As an aside: we can normalize a distribution before sampling if needed, by using 'totalWeight' above and 'reweight' below).
We can also make 'DiscProb' a monad:

```
instance Monad DiscProb where
  return a = MkProb [(a, 1.0)]
  ma >>= f = MkProb [(b, p * p') | (a, p) <- probs ma, (b, p') <- probs $ f
a]
```

We can use monadic notation to build bigger distributions.
For example,

```
example = do c <- bernoulli 0.3
             c' <- bernoulli 0.4
             c'' <- bernoulli 0.7
             return $ (c && c') || c''
```

represents the distribution

```
MkProb
[(True,8.399999999999999e-2),(True,3.6000000000000004e-2),(True,0.126),(Fal
se,5.4000000000000006e-2),(True,0.19599999999999998),(False,8.4e-2),(True,0
```

```
.294),(False,0.126)]
```

which 'collect's to

```
MkProb [(False,0.264),(True,0.736)]
```

e. [4pt] Convert the example distribution below into bind-notation:

```
example2 :: DiscProb Int
example2 = do
 n <- bernoulli 0.3
 let m = n*3
 score (fromIntegral n)
 return $ n * m
```

Hint: for this question, it should not matter what 'score' is, but if you think it might help you, it is defined in the subquestion below.

4 pt. **i.**

f. [3pt] As observed earlier, 'DiscProb a' also contains representations of unnormalised distributions, other than the basic constructs for probability distributions. The simplest example is the distribution

```
score :: Double -> DiscProb ()
score w = MkProb [((), w)]
```

on the one-point type '()' that assigns its only element '()' weight 'w' (which may not be '1.0').

It turns out that 'score' together with normalized probability distributions is enough to get any unnormalised distribution. Indeed, show that we can implement the function

```
reweight :: (a -> Double) -> DiscProb a -> DiscProb a
```

such that 'reweight f d' multiplies the distribution 'd' with the density function 'f', using do-notation and 'score'. For example,

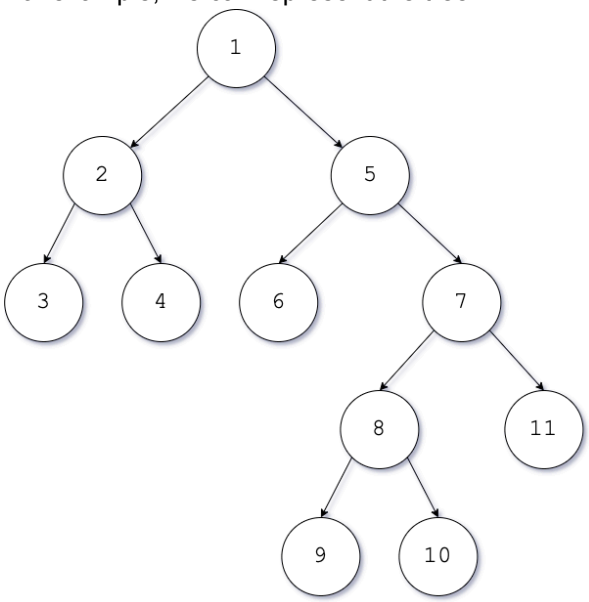
```
> reweight (\x -> fromIntegral $ x) (categorical [0.2, 0.3, 0.4, 0.1])
MkProb {probs = [(0,0.0),(1,0.3),(2,0.8),(3,0.3)]} -- i.e. categorical [0,
0.3, 0.8, 0.3]
```

3 pt. **j.**

5 In this question we ask you to write some basic code that traverses tree-shaped data structures. Consider the following tree type, in which we store values at both the leaves and the internal nodes.

```
data Tree a = Leaf a | Node (Tree a) a (Tree a) deriving (Eq, Show)
```

For example, we can represent the tree



as

```
exampleTree = Node (Node (Leaf 3) 2 (Leaf 4)) 1 (Node (Leaf 6) 5 (Node
(Node (Leaf 9) 8 (Leaf 10)) 7 (Leaf 11)))
```

Consider a function

```
levels :: Tree a -> [[a]]
```

that, given a tree 't', computes a list '[l_0,..,l_k]' of lists, such that the list 'l_i' contains all (values of) the nodes and leaves that are at distance 'i' from the root of 't', (and k is the maximum depth of the tree). 'l_i' contains the values from left to right. For example,

```
> levels exampleTree
[[1],[2,5],[3,4,6,7],[8,11],[9,10]]
```

a. [2pt] Use 'levels' to implement a function

```
bfs :: Tree a -> [a]
bfs = a. ..........................................................(2 pt.)
```

that computes all values in the tree in breath-first order, i.e. such that

```
> bfs exampleTree
[1,2,5,3,4,6,7,8,11,9,10]
```

b. [4pt] Now, please complete this implementation of 'levels':

```
levels = go . (:[]) where
  go :: [Tree a] -> [[a]]
  go [] = b.  ..................................................(1 pt.)
  go queue = let (lvl, queue') = foldr extend
                                      c.  ........................(1 pt.)
                                      queue
                 in lvl : go queue'
  extend :: Tree a -> ([a], [Tree a]) -> ([a], [Tree a])
  extend (Leaf x) (lvl, queue') = d.  ..............................(1 pt.)
  extend (Node l x r) (lvl, queue') = e.  ..........................(1 pt.)
```