[20221110] INFOFP - Functioneel programmeren - 1 - Online

Cursus: BETA-INFOFP Functioneel programmeren (INFOFP)

Tijdsduur: 2 uur

Aantal vragen: 5

[20221110] INFOFP - Functioneel programmeren - 1 - Online

Cursus: Functioneel programmeren (INFOFP)

Aantal vragen: 5

1 Consider the datatypes

```
type Guest = String
data Room = Room { roomNumber :: Int
                 , members :: [Guest]
                 }
              deriving (Show,Eq,Ord)

newtype Floor = Floor { rooms :: [Room] } deriving (Show)
```

a. [6pt] Write a function

```
consecutive :: (Enum a, Eq a) => [a] -> Bool
```

that tests if all values in a list are consecutive, i.e. every next element is the successor of the predecessor.

Hint: recall that the Enum typeclass implements a function

```
succ :: Enum a => a -> a
```

6 pt. **a.**

b. [4pt] Using function composition, write a property

```
noMissing :: Floor -> Bool
```

that tests if there are any missing rooms; i.e. if a floor has 'n' and 'm' as room numbers with 'n > m', then there is a room with number 'k' for any 'k' between 'n' and 'm'. You may use the function

```
sort :: Ord a => [a] -> [a]
```

```
noMissing = b.  ..........................................  (4 pt.)
```

c. [4pt] We call a floor valid, if no two rooms have the same number; please define a function

```
isValidFloor :: Floor -> Bool
```

that checks whether a floor is valid.
Hint: you may want to use the function

```
group :: Eq a => [a] -> [[a]]
```

that takes a list and returns a list of lists such that the concatenation of the result is equal to the argument and, moreover, each element list in the result contains only equal elements and consecutive element lists in the result do not contain equal elements. For example,

```
>>> group "Mississippi"
["M","i","ss","i","ss","i","pp","i"]
```

```
isValidFloor = c. ..... (4 pt.)
```

d. [8pt] Given an infinite list 'rms' of Rooms and an Int 'n', generate a valid floor that has 'n' rooms taken from the list 'rms'

```
genValidFloor rms n = Floor $ take n (validRms d. ..... (2 pt.) )

validRms acc (rm:rms) | elem (roomNumber rm) acc = e. .... (2 pt.)

                      | otherwise = f. ..... (4 pt.)
```

2 Using the following definitions

20 pt.

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

size :: Tree a -> Int
size Leaf         = 0                                          --
(a)
size (Node l v r) = size l + (1 + size r)                      --
(b)

combineTree :: Tree a -> Tree a -> Tree a
combineTree Leaf r'         = r'                               --
(c)
combineTree (Node l v r) r' = Node l v (combineTree r r')      --
(d)
```

and the laws

```
0 + i = i                                                      --
(e)
(i + j) + k = i + (j + k)                                      --
(f)
```

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

[20pt] Claim: for all Trees l' and r'

```
size (combineTree l' r') = size l' + size r'
```

3 [8pt] In this question, we test your knowledge of laziness.
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".

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

a. ..... (2 pt.)

```
map (+1) [0..2]
```

b. ..... (2 pt.)

```
length xs : map (+1) [0..5]
```

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

```
(\x -> Node Leaf (x + 1) Leaf) undefined
```

d. ..... (2 pt.)

4 a. [6pt] Write a function

```
countLinesAndWords :: FilePath -> IO Int
```

that takes a FilePath to some text file, prints the number of lines in the file, and returns the total number of words in the file. Remember that

```
readFile :: FilePath -> IO String
```

reads the given file, and

```
print :: Show a => a -> IO ()
```

prints a value to standard output.
Hint: remember that the functions

```
words :: String -> [String]
lines :: String -> [String]
```

split a String into words and lines, respectively.

6 pt. **a.**

The following funtion longestFile reads a directory name, and uses countLinesAndWords to compute the file with the most words (and lets the user know that it is working, by printing some console output).

```
longestFile :: FilePath -> IO Int
longestFile fp = do files <- listDirectory fp
  ls <- mapM countLinesAndWords files
  print "Working..."
  let l = maximum ls
  return l
```

b. [6pt] Rewrite 'longestFile' to use >>= directly rather than using do-notation.

6 pt. **b.**

5 Consider the following data types:

```
data Office = Office { _building :: String
                     , _floor :: Int
                     , _room :: Int
                     }
                     deriving (Show,Eq)

data Employee = Employee { _name :: String
                         , _age :: Int
                         , _office :: Office
                         }
                         deriving (Show,Eq)
```

with some example values

```
wrongFrank = Employee "Frank" 34 (Office "BBG" 4 9)
matthijs = Employee "Matthijs" 32 (Office "BBG" 5 65)
frank = moveToRoom 11 wrongFrank
```

a. [4pt] Write a function

```
moveToRoom :: Int -> Employee -> Employee
```

that takes a room number, and an employee, and updates the '_room' field of the office of that employee. Your function should use pattern matching to access the appropriate fields. So for example:

```
>>> moveToRoom 11 wrongFrank
Employee {_name = "Frank", _age = 34, _office = Office {_building =
"BBG", _floor = 4, _room = 11}}
```

4 pt. **a.**

Let us define a type level equivalent of the 'Const' function, which takes two type arguments, and only remembers the first one.

```
newtype Const c a = MkConst c
  deriving (Show,Eq)
```

For example, the value 'reallyJustAnInt' only stores an 'Int'; the 'String' in the type signature is not stored at all/there is no string at all:

```
reallyJustAnInt :: Const Int String
reallyJustAnInt = MkConst 5

>>> reallyJustAnInt
MkConst 5
```

b. [4pt] Let c be some type. Give the Functor instance for the type 'Const c' i.e. we should have that:

```
stillNoString = fmap (++ "and some extra string") reallyJustAnInt

>>> stillNoString
MkConst 5
```

```
instance b. ..... (2 pt.) where c. . (2 pt.)
```

While the solution to question a. is hopefully not too difficult, it is not very convenient to write the above code, (in particular if Employee and Office would both have many more fields). In this subquestion, we investigate an alternative, more composable, solution to the problem called Lenses.

Consider the following type 'LensF' which we can use as accessors of a particular field in a haskell data type. See also the example accessors below.
Note that technically, a 'LensF' is just a function with two parameters.

```
type LensF f p a = (a -> f a) -> p -> f p

name :: Functor f => LensF f Employee String
name f (Employee n a o) = fmap (\n' -> Employee n' a o) $ f n

age :: Functor f => LensF f Employee Int
age f (Employee n a o) = fmap (\a' -> Employee n a' o) $ f a

office :: Functor f => LensF f Employee Office
office f (Employee n a o) = fmap (\o' -> Employee n a o') $ f o

room :: Functor f => LensF f Office Int
room g (Office b f r) = fmap (\r' -> Office b f r') $ g r
```

c. [4pt] Write the function

```
view :: LensF (Const a) p a -> p -> a
```

so that we can use a

```
myLens :: LensF (Const a)
```

as a "getter" of a record. That is, so that 'view myLens p' to view/read/get the field 'myLens' of a record 'p':

```
>>> view name frank
"Frank"

>>> view office matthijs
Office {_building = "BBG", _floor = 5, _room = 65}

view lens p = d. ..... (4 pt.)
```

Using the following 'Identity' type, we can also write a "setter" function 'set':

```
newtype Identity a = Identity a

instance Functor Identity where
   fmap f (Identity x) = Identity (f x)

set           :: LensF Identity p a -> p -> a -> p
set lens p a' = let Identity p' = lens (\_ -> Identity a') p
                in p'
```

So for example, we can write:

```
>>> set name frank "FRANK"
Employee {_name = "FRANK", _age = 34, _office = Office {_building =
"BBG", _floor = 4, _room = 11}}
```

The nice thing about these Lenses is that they compose! To access the room number of a particular employee, we can now write:
(Note: Don't worry why this actually works the way it does/why we can write the lens this way.)

```
officeNumber :: Functor f => LensF f Employee Int
officeNumber = office . room
```

d. [4pt] Write a function

```
moveToRoom2 :: Int -> Employee -> Employee
```

that uses the 'officeNumber' lens to update the room of a particular employee:

```
>>> moveToRoom2 110 frank
Employee {_name = "Frank", _age = 34, _office = Office {_building =
"BBG", _floor = 4, _room = 110}}

moveToRoom2 n p = e. ..................................... (4 pt.)
```

e. [4pt, bonus] Please show how 'Identity' can be made an instance of 'Monad'.

```
instance f. ..................................... (1 pt.) where

g. ........................... (1 pt.) -- no identation necessary

h. ........................... (2 pt.) -- no identation necessary
```