[20201105] INFOFP - Functioneel programmeren - 1 - UITHOF

Cursus: BETA-INFOFP Functioneel programmeren (INFOFP)

Tijdsduur:	3 uur
Aantal vragen:	6

[20201105] INFOFP - Functioneel programmeren - 1 - UITHOF

Cursus: Functioneel programmeren (INFOFP)

Aantal vragen: 6

1 Lists allow efficient access to the front of the list, but not to the back of the list. A "Deque" (double ended queue) allows fast access to both the front and the back. The following data type models such a Deque:

```
data Deque a = Empty
             | Single a
             | Multiple (Access a) (Deque (a,a)) (Access a)
             deriving (Show,Eq)
```

where

```
data Access a = One a | Two a a deriving (Show,Eq)
```

So, for example a 'Single 1' corresponds to a Deque containing a single element (the Int '1'), and a

```
mySmallDeque :: Deque Int
mySmallDeque = Multiple (One 1)
                        (Single (2,3))
                        (Two 4 5)
```

is a Deque containing the elements '1', '2', '3', '4', '5' in that order. As a slightly larger example

```
myDeque :: Deque Int
myDeque = Multiple (Two 1 2)
                   (Multiple (One (3,4))
                             (Single ((5,6),(7,8)))
                             (One (9,10)))
                   (One 11)
```

contains all elements '1', ..., '11' (in that order).

a. [5pt] Write a function 'dequeToList' that converts a Deque into a list (containing the same elements in the same order). In particular, we have

```
dequeToList mySmallDeque == [1,2,3,4,5]
dequeToList myDeque == [1,2,3,4,5,6,7,8,9,10,11]
```

Hint: you may want to write a helper function 'flatten :: [(a,a)] -> [a]' first.

a.

b. [4pt] Write a total function 'safeLast' that gets the last (rightmost) element in the Deque (if the Deque is non-empty). For example, we have that

```
safeLast myDeque == Just 11
```

Your implementation should use constant (O(1)) time (so you cannot convert the Deque into a list first).

b.

c. [4pt] Implement a function 'cons' that takes an 'x :: a' and a 'dq :: Deque a' and adds the 'x' to the front of 'dq' (analogous to how
the '(:) :: a -> [a] -> [a]' function/constructor adds an element to the front of a list. For example,

```
cons 1 Empty        == Single 1
cons 0 mySmallDeque == Multiple (Two 0 1) (Single (2,3)) (Two 4 5)
```

Your implementation should run in O(log n) time. You do not have to prove/argue that your implementation achieves this O(log n) time bound; the most natural implementation will achieve this.

Hint: the function 'cons' will be a recursive function

c.

d. [4pt] Make the 'Deque' data type an instance of the 'Functor' typeclass. You may assume that 'Access' has already been made an instance of 'Functor'.

d.

2 In this question, we will ask you to perform some proofs about program equivalence.

Here, you may make use of the following definitions:

```
foldl :: (b -> a -> b) -> b -> [a] -> b

foldr :: (a -> c -> c) -> c -> [a] -> c

id :: d -> d

foldl2 :: (b -> a -> b) -> b -> [a] -> b

const :: a -> r -> a

bind :: (r -> a) -> (a -> r -> b) -> r -> b

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

help :: (b -> a -> b) -> a -> (b -> b) -> (b -> b)

(a) foldl op e [] = e

(b) foldl op e (x : xs) = foldl op (op e x) xs

(c) foldr op e [] = e

(d) foldr op e (x : xs) = op x (foldr op e xs)

(e) id x = x

(f) foldl2 op e bs = foldr (help op) id bs e

(g) const a _ = a

(h) bind f g r = g (f r) r

(i) flip f a b = f b a

(j) help op a g b = g (op b a)
```

Please mark every reasoning step in your proof either with the name of a definition (like (b)) or as the use of an induction hypothesis (I.H.). In case of a proof by induction, please explicitly state the induction hypothesis and mark it as such. In case you use extensional reasoning, please explain.

a. a. [6pt] Prove that

'flip bind const' = 'id'

Hint: note that this is an equation of expressions of type '(r -> b) -> r -> b' (or, equivalently, '(r -> b) -> (r -> b)')

b. b. [14pt] Prove that

'foldl2' = 'foldl'

3 Recall that a 'Map.Map k a' is a data structure that maps keys of type k to their associated values of type 'a', and that we can retrieve the value corresponding to a key (if it exists) using the 'Map.lookup :: k -> Map.Map k a -> Maybe a' function.

Consider the following type

```
type Graph v = Map.Map v [v]
```

which models directed graphs whose vertices are of type 'v'. In particular, the graph is stored using an adjacency-list representation where each vertex stores its (outgoing) neighbours.

This means we can report all vertices of the graph by retrieving all keys in the Map like:

```
vertices :: Graph v -> [v]
vertices = Map.keys
```

a. [2pt] Write the function edges, which returns a list of all edges in the graph. Each pair (u,v) in the output should be a directed edge from u to v. Your function should have type:

```
edges :: Graph v -> [(v,v)]
```

Hint: the function 'Map.assocs :: Map k v -> [(k,v)]' produces a list with all key,value pairs in a 'Map.Map'.

a.

b. [2pt] Write a function 'neighbours' that gives all neighbours of 'v', meaning those 'w' such that there is a directed edge from 'v' to 'w'. Your function should have type:

```
neighbours :: Ord v => Graph v -> v -> [v]
```

b.

Given a graph 'g', and a vertex 'v', we may want to compute all vertices reachable from 'v' by following directed edges. Here is a possible implementation of such a function:

```
reachablePure     :: Ord v => Graph v -> v -> [v]
reachablePure g v = let ws = neighbours g v
                    in v : concatMap (reachablePure g) ws
```

c. [2pt] Is this implementation correct? If yes: argue why; if no: explain why not.

c.

We will implement 'reachable' once more, this time using a 'State' monad.

Recall that:

1) 'State s a'

is a type of computations that maintain some state of type 's' and return an 'a'

2) 'runState :: State s a -> s -> (a,s)'
is a function that given a computation and an initial state performs that computation and returns the resulting 'a' and the final state.

3) 'get :: State s s'
is a stateful computation that returns the current state

4) 'put :: s -> State s ()'
is a function that takes an 's', and produces (a computation that) sets the state to the given 's', and

5) 'modify :: (s -> s) -> State s ()'
is a function that, given a function f produces (a computation that) modifies the current state by applying the function 'f' to it.

We can then implement reachable as follows:

```
reachable :: Ord v => Graph v -> v -> [v]
reachable g v = snd $ runState (markVisited g v) []
```

where 'markVisited g v' is a stateful computation that traverses 'g', starting at vertex 'v', while keeping track of a list of already visited vertices.

d. [4pt] Complete the following implementation of this function 'markVisited':

```
markVisited     :: Ord v => Graph v -> v -> State [v] ()
markVisited g v = do visited <- d. .......... ()
                     if v `elem` visited then
                        e. .......... ()
                     else do
                            f. .......... ()
                            mapM_ g. .......... ()  $ neighbours g v
```

4 Let 'extract :: Int -> [a] -> ([a],a,[a])' be a function that given an index 'i' and a list 'xs' extracts the i^{th} element from a list. More precisely, it can be implemented as

```
extract i xs = let pref = take i xs
                   (x:suf) = drop i xs
               in (pref,x,suf)
```

Given this function 'extract' we can implement the following function, which shuffles a list:

```
shuffle :: [a] -> IO [a]
shuffle [] = return []
shuffle xs = do i <- randomRIO (0,length xs - 1)
                let (pref,x,suf) = (extract i xs)
                xs' <- shuffle (pref ++ suf)
                return (x:xs')
```

a. [4pt] Rewrite the non-empty list case of 'shuffle' using 'return' and '>>=' (i.e. without using do-notation).

a.

b. [4pt] Write a function 'foo :: IO Int' that asks the user to input one or more Ints separated by spaces, and prints a random permutation of this list and returns its sum.

Hint: the function 'getLine :: IO String' reads a line from the standard input

b.

5 Consider the function words which "breaks a string up into a list of words, which were delimited by spaces".

We will write some some tests to verify whether a given implementation 'wordsImpl' of the words function is correct.

Note that throughout this exercise we will use a slightly simplified version of the 'words' function (compared to the one in Data.List) in that we will break the input string only at spaces (not at newlines and tabs). We will simply ignore newlines and tabs throughout the exercise.

a. [2pt] Write a quickcheck property 'noMoreSpaces' that checks that a given implementation does not "forget" any spaces; that is, if all Strings in the output list are free of spaces. Your function should have type signature:

```
noMoreSpaces :: (String -> [String]) -> String -> Bool
```

a.

b. [2pt] Given a function

```
removeInitialSpaces :: String -> String
removeInitialSpaces = dropWhile (== ' ')
```

that removes all spaces from the start of a 'String', please implement a function

```
removeFinalSpaces :: String -> String
```

that removes all spaces from the end of a 'String'.

So, for example,

```
removeFinalSpaces " Rick  Astley    " == " Rick  Astley"
```

b.

c. [4pt] Please implement a function

```
removeDuplicateSpaces :: String -> String
```

that replaces any number of consecutive spaces in a string by a single space, so, for example,

```
removeDuplicateSpaces "  Never   gonna  give  you   up   " == " Never
gonna give you up "
```

c.

Consider the property below.

```
recombines :: (String -> [String]) -> String -> Bool
recombines words' s = let removeRedundantSpaces = removeFinalSpaces .
                                                  removeInitialSpaces .
                                                  removeDuplicateSpaces
                      in (removeRedundantSpaces . unwords . words') s ==
removeRedundantSpaces s
```

Here, 'unwords' is the following Prelude function that creates a single String from a list of Strings by concatenating them while inserting spaces between them.

```
unwords [] = ""
unwords (w : ws) = w ++ go ws where
    go [] = []
    go (w : ws) = ' ' : w ++ go ws
```

d. [2pt] Is it true that 'recombines' complements 'noMoreSpaces' to give a correct specification of 'words' in the sense that any function 'wordsImpl' that satisfies both 'noMoreSpaces wordsImpl s' and 'recombines wordsImpl s' for all Strings 's' has the property that 'words s = wordsImpl s' for all Strings 's' ? If yes, please explain why; if no, please explain why and correct the definition of 'recombines' to make it true.

d.

6 a. [2pt] For each expression 'e' below, choose the expression 'f' that we obtain after evaluating it to WHNF.

More precisely, select the expression 'f' such that

1) by performing some finite number 'k' evaluation steps 'e' evaluates to 'f'
2) 'f' is in WHNF
3) there is no expression 'g' in WHNF such that 'e' evaluates to 'g' in fewer than 'k' steps.

A correct answer gives you 1 point. A wrong answer -1. Selecting I don't know gives you 0 points.

a. map (:[]) [1,2]

- **a.** [[1],[2]]
- **b.** [1] : map (:[]) [2]
- **c.** ((:[]) 1) : map (:[]) [2]
- **d.** map (:[]) [1,2]
- **e.** I don't know

b. fmap (+1) $ Just (3+2)

- **a.** Just 6
- **b.** Just ((3+2)+1)
- **c.** fmap (+1) $ Just (3+2)
- **d.** Just (fmap (+1) (3+2))
- **e.** I don't know

c. b. [3pt, bonus] Write a function 'force :: [a] -> [a]' that evaluates all the elements in the input list to WHNF.

Clearly, the use of force makes our code less lazy. For example, we can no longer safely write:

```
foo xs = length . force $ xs
```

when 'xs' is some list containing elements that would diverge (e.g. for 'xs = [1,2,3,4,undefined]')

c. [3pt, bonus] Write a pair of functions, such that together they can undo the effect of force. In particular, so that for finite lists 'xs' we have:

```
(unprotect . force . protect) xs == id xs
```

and thus we can safely write:

```
foo xs = length . unprotect . force . protect $ xs
```

d.