[20211111] INFOFP - Functioneel programmeren - 1 - UITHOF

Cursus: BETA-INFOFP Functioneel programmeren (INFOFP)

Tijdsduur: 3 uur

Aantal vragen: 5

[20211111] INFOFP - Functioneel programmeren - 1 - UITHOF

Cursus: Functioneel programmeren (INFOFP)

Aantal vragen: 5

1 In this question, we will ask you to perform some proofs of program equivalence.

Here, you may make use of the following definitions:

```
-- Types
type Writer a = (String, a)
data Tree a = Leaf a | Node (Tree a) (Tree a)

-- Function signatures
combine :: Writer a -> (a -> Writer b) -> Writer b
noWrite :: a -> Writer a
(++) :: [a] -> [a] -> [a]
flatten :: Tree a -> [a]
flattenAcc :: Tree a -> [a] -> [a]

-- Function definitions
(a) combine (m, a) f = (m ++ fst (f a), snd (f a))
(b) noWrite a = ("", a)
(c) [] ++ ys = ys
(d) (x:xs) ++ ys = x : (xs ++ ys)
(e) flatten (Leaf n) = [n]
(f) flatten (Node l r) = flatten l ++ flatten r
(g) flattenAcc (Leaf n) ns = n : ns
(h) flattenAcc (Node l r) ns = flattenAcc l (flattenAcc r ns)
```

as well as the following laws:

```
 -- Laws
(i) (fst x, snd x) = x       -- for any x :: (A, B) for any
(parameterized) types A, B
(j) fst (x, y) = x
(k) snd (x, y) = y
(l) xs ++ (ys ++ zs) = (xs ++ ys) ++ zs    -- for any xs, ys, zs :: [A]
for any (parameterized) type 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. Partial solutions can still be worth points.

4 pt. **a.** a. [4pt] Please prove that

'combine (noWrite a) f = f a'

9 pt. **b.** b. [9pt] Please prove that

'flatten t ++ ns = flattenAcc t ns'

for all 't :: Tree a' and <u>all</u> 'ns :: [a]'.

2 In this question, we test your knowledge of programming on data types and testing.

Consider (again) the following type of binary trees:

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

Let us say that such a tree is balanced if the number of leaves in the left and right subtree of every node differs by at most one, with leaves themselves being trivially balanced.
a. [4pt] Define a property

```
isBalanced :: Tree a -> Bool
```

that checks whether a binary tree is balanced.

4 pt. **a.**

b. [3pt] Define a function

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

that satisfies the specification that it converts a non-empty list 'l' into a balanced binary tree 't' such that 'flatten t = l' (where we use the function 'flatten' from question 1).

Hint: use the function 'split' that splits a list into two halves whose length differs by at most one

```
split :: [a] -> ([a], [a])
split xs = (take n xs, drop n xs) where
  n = length xs `div` 2
```

3 pt. **b.**

c. [2pt] Define a specification

```
propBalanceCorrect :: Eq a => ([a] -> Tree a) -> [a] -> Bool
```

that checks whether an implementation 'balanceImpl' of 'balance' is correct according to the specification that is given in subquestion b.

2 pt. **c.**

d. [3pt, bonus] Recall that

```
class Arbitrary a where
  arbitrary :: Gen a
```

is a type class that allows you to generate arbitrary values of type 'a' in the 'Gen' monad.

Write an 'Arbitrary' instance for 'Tree a' that generates balanced binary trees.

Hint: remember that the function 'balance' crashes for empty lists!

3 pt. **d.**

3 In this question, we test your knowledge of laziness and the evaluation of Haskell programs.

a. [3pt] One simple definition of a function that sorts a list is

```
sort [] = []
sort (x:xs) = insert x (sort xs) where
  insert x [] = [x]
  insert x (y:ys) = if x <= y then x:y:ys else y:insert x ys
```

This method is called insertion sort.
Reduce 'sort [4, 3]' to its WHNF under lazy evaluation (only give the final result: the WHNF).

a. .... (3 pt.)

b. [2pt] Evaluating the expression 'fmap (+1) $ Just (3+2)' to its WHNF under lazy evaluation yields the expression:

(b) **A.** Just (fmap (+1) (3+2)) **B.** fmap (+1) $ Just (3+2) **C.** Just 6 **D.** Just ((+1) (3+2)) **E.** I don't know (0 pt.)

(2pt for the correct answer, -1pt for a wrong answer, 0pt for "I don't know")

c. [2pt, bonus] Consider the definitions

```
length :: [a] -> Int
length = foldl (\n x -> n+1) 0

length2 :: [a] -> Int
length2 = lengthAcc 0

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

'length' and 'length2' both compute the length of a list.
Describe the difference in (asymptotic) space and time usage of 'length' compared to 'length2'.

2 pt. **c.**

4 In this question, we test a mixture of some basic programming skills and knowledge of functors, monads and IO. Knowledge of monads and IO is only needed for subquestions d and e.

a. [3pt] Write a function

```
lookUnsafe :: Eq k => k -> [(k, v)] -> v
```

such that 'lookUnsafe k kvs' returns the first 'v' such that '(k, v)' is in 'kvs' for some 'k'. 'lookUnsafe k kvs' may crash if there is no 'v' such that '(k, v)' is in 'kvs' for any 'k'.

3 pt. **a.**

b. [2pt] Consider the data type

```
data Reader s a = MkReader (s -> a)
```

We think of a value of type 'Reader s a' as an impure computation that computes a value 'a' with access to some read-only memory of type 's'.
Define an operation

```
get :: Reader s s
```

that returns the current value of the memory.

b. .... (2 pt.)

c. [2pt] Please give a 'Functor' instance for 'Reader s', such that 'fmap f m' applies a function 'f' to the result of a computation 'm' that reads state of type 's'.

2 pt. **c.**

d. [4pt] We can turn 'Reader s' into a monad. We will spare you the abstract definitions and instead explain what this monad it does by example.
The idea is that 'Reader s' behaves like the 'State s' monad with the restriction that we cannot write to the memory and we can only read it. That is, the 'Reader s' monad behaves exactly likes the 'State s' monad except that we cannot use the operation 'put' for writing state (or any other state writing operations that are derived from 'put'). For example, running 'lyrics' below returns "My name is What? My name is Who? My name is Chika-chika Slim Shady".

```
withEnv :: String -> Reader String String
withEnv s = do
  env <- get
  return (env ++ s)

wws :: Reader String String
wws = do
  w  <- withEnv "What? "
  w2 <- withEnv "Who? "
  ss <- withEnv "Chika-chika Slim Shady"
  return (w ++ w2 ++ ss)

runReader :: Reader s a -> s -> a
runReader (MkReader f) = f

lyrics :: String
```

```
lyrics = runReader wws "My name is "
```

Here, the operation 'get' returns "My name is " every time we call it, as that is the value we initialise the read-only state with using 'runReader'. There is no way for us to change the value of this state as we do not have a 'put' operation.

Using the Reader monad, we can write an evaluator

```
eval :: Expr -> Reader [(String, Int)] Int
```

for an expression of type

```
data Expr = Val Int          -- integer value
          | Mult Expr Expr  -- multiplication
          | Var String      -- variable
```

as a computation that has access to a read-only bit of memory that stores an association list 'm :: [(String, Int)]' in which we can look up the values of various variables.

Please write this evaluator.

Hint: you may want to use the function 'lookUnsafe' from subquestion a.

4 pt. **d.**

e. [3pt] Observe that we can define a function

```
evalR :: Expr -> [(String,Int)] -> Int
evalR = runReader . eval
```

that evaluates expressions, given a list of (variableName,value)-pairs.
Suppose that we further have a function

```
readKeyValues :: String -> [(String,Int)]
```

that reads (variableName,value)-pairs separated by spaces from a 'String'.
Please implement *without* using do-notation, i.e. by using return and >>= for the IO monad directly, a function

```
runWithEnv :: Expr -> IO ()
```

that takes an expression as an argument, and interactively takes from the user as input a list of (variableName,value)-pairs separated by spaces, and prints the value after evaluating the expression. For example

```
> runWithEnv (Var "v")
("v",5)
5

> runWithEnv (Mult (Var "v") (Mult (Val 2) (Var "x")))
("v",5) ("x",2)
```

```
20
```

Hint: you may want to make use of the functions

```
getLine :: IO String
```

which interactively takes a line of keyboard input from the user

and

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

which prints an 'a' to the standard output channel (assuming that we know how to 'show a').

3 pt. **e.**

5 In this question, you will work with an abstract data type and interpret and complete a piece of code that makes use of that data type. We also ask you to use and interpret some higher-order functions.
Consider the following (abstract) data type of 'Set's of 'a's.
1. We can form an empty set 'empty' (which does not have any elements).
2. We can use the function 'insert' on an element 'a' and an existing set 's' to obtain a new set 'insert a s', which contains all the elements that s did as well as 'a'.
3. We can use 'member a s' to check whether a given 'a' is an element of a set 's'.
That is 'Set a' is a type that implements the following functions

```
empty :: Ord a => Set a
insert :: Ord a => a -> Set a -> Set a
member :: Ord a => a -> Set a -> Bool
```

a. [3pt] Using a fold, write a function

```
fromList :: Ord a => [a] -> Set a
```

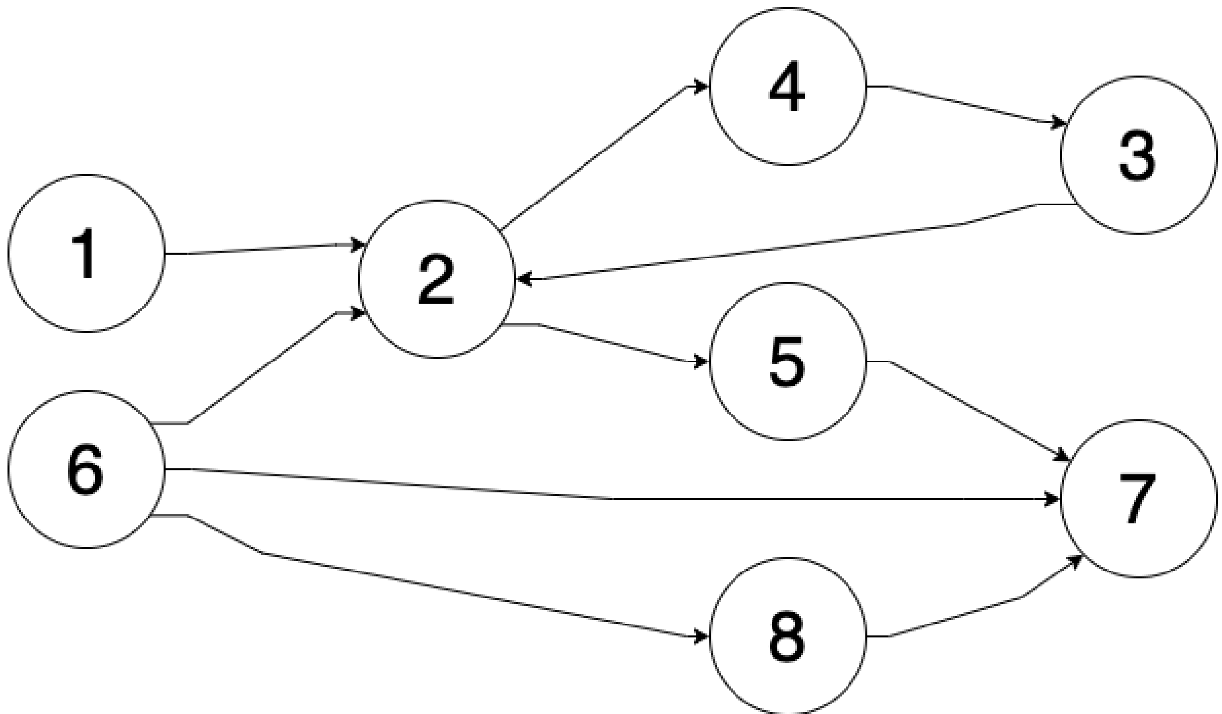
that converts a list of 'a's to a 'Set a' with the same elements.

3 pt. **a.**

b. [4pt] We can encode directed graphs whose nodes are 'a's as functions 'a -> [a]' that send a node to its successors. For example

```
testGraph :: Int -> [Int]
testGraph 1 = [2]
testGraph 2 = [4, 5]
testGraph 3 = [2]
testGraph 4 = [3]
testGraph 5 = [7]
testGraph 6 = [2, 7, 8]
testGraph 7 = []
testGraph 8 = [7]
```

represents the graph



Please complete the following code that traverses such a graph to compute the set of graph nodes reachable from the given starting node. Note that a node is always reachable from itself, so the resulting set should include the starting node and thus has size at least 1. For example, 'reachable testGraph 2' returns the set that contains 3, 4, 7, 5, and 2.

```
reachable :: Ord a => (a -> [a]) -> a -> Set a
reachable nexts node = reachableVis nexts node empty

reachableVis :: Ord a => (a -> [a]) -> a -> Set a -> Set a
reachableVis nexts node visited
  | node `member` visited = b. .............................. (2 pt.)
  | otherwise = foldr (reachableVis nexts)
c. .............................. (2 pt.)  (nexts node)
```

2 pt. **d.** c. [2pt] Does 'reachable' use a depth-first or breadth-first traversal?

(2pt for correct answer, -1pt for wrong answer, 0pt for "Don't know")

a. Breadth-first

b. Don't know

c. Depth-first