Chalmers | Göteborgs Universitet 2019-01-19

Examiner: Thomas Hallgren, D&IT,
Answering questions at approx 10.00 (or by phone)

Functional Programming TDA 452, DIT 143

2019-01-19 8:30 – 12:30 Lindholmen-salar

- There are 5 questions with maximum $8 + 8 + 12 + 8 + 4 = 40$ points. Grading:

  Chalmers: 3 = 20–26 points, 4 = 27–33 points , 5 = 34–40 points
  GU: G = 20–33 points, VG = 34–40 points

- Results: latest approximately 10 days.
- **Permitted materials:**
  - Dictionary
- **Please read the following guidelines carefully:**
  - Read through all Questions before you start working on the answers.
  - Begin each Question on a new sheet.
  - Write clearly; unreadable = wrong!
  - For each part Question, if your solution consists of more than a few lines of Haskell code, use your common sense to decide whether to include a short comment to explain your solution.
  - You can use any of the standard Haskell functions *listed at the back of this exam document.*
  - **Full points** are given to solutions which are **short**, **elegant**, and **correct**. Fewer points may be given to solutions which are unnecessarily complicated or unstructured.
  - You are **encouraged to use** the solution to an **earlier part** of a Question to help solve a **later part** — even if you did not succeed in solving the earlier part.

1. *(8 points)* For each of the following definitions, give the most general type, or write "No type" if the definition is not correct in Haskell.

```
fa x     = "Hello, "++x
fb (x:y) = (x,y)
fc x y z = x<=y && y<=z
fd       = map . map
```

Solution:

```
fa :: String -> String
fb :: [a] -> (a,[a])
fc :: Ord a => a -> a -> a -> Bool
fd :: (a->b) -> [[a]] -> [[b]]
```

2. *(8 points)*

   (a) *(3 points)* Define a function `subsequences` that computes all the subsequences of a list, i.e. all the list you get by keeping *some* of the elements in the list (including *none* of the elements and *all* of the elements).

   ```
   subsequences :: [a] -> [[a]]
   ```

   The elements within each subsequence should appear in the same order as in the argument list, but it doesn't matter in which order the subsequences appear, i.e. it could be the shortest subsequence first, the longest subsequence first, or some other order. Examples:

   ```
   subsequences [1]   == [[],[1]]
   subsequences [1,2] == [[],[1],[2],[1,2]]
   subsequences "abc" == ["abc","ab","ac","a","bc","b","c",""]
   ```

   (b) *(3 points)* Define a function `isSubsequenceOf` that checks if one list is a subsequence of another list.

   ```
   isSubsequenceOf :: Eq a => [a] -> [a] -> Bool
   ```

   Examples:

   ```
   ""   `isSubsequenceOf` "abc" == True
   "a"  `isSubsequenceOf` ""    == False
   "a"  `isSubsequenceOf` "abc" == True
   "ac" `isSubsequenceOf` "abc" == True
   "ad" `isSubsequenceOf` "abc" == False
   "cb" `isSubsequenceOf` "abc" == False
   ```

   (c) *(2 points)* Write a property that can be used with QuickCheck to test that all the subsequences returned by `subsequences` really are subsequences of the argument list.

Solution:

```
-- (a)
-- A list of length n has 2^n subsequences, since for each element you
-- can choose to include it or not (n choices with 2 possibilites).
subsequences [] = [[]]
subsequences (x:xs) = map (x:) pxs ++ pxs
  where pxs = subsequences xs

subsequences_v2 [] = [[]]
subsequences_v2 (x:xs) = [ss | ss' <- subsequences_v2 xs,
                               ss  <- [ss',x:ss']]

-- (b)
isSubsequenceOf []     ys                 = True
isSubsequenceOf xs     []                 = False
isSubsequenceOf (x:xs) (y:ys) | x==y      = isSubsequenceOf xs ys
                              | otherwise = isSubsequenceOf (x:xs) ys

-- (c)
prop_subsequences :: [Int] -> Bool
prop_subsequences xs = all (`isSubsequenceOf` xs) (subsequences xs)

prop_subsequences_v2 :: [Int] -> Property
prop_subsequences_v2 xs = length xs<15 ==>
                          all (`isSubsequenceOf` xs) (subsequences xs)
-- Since a list of length n has 2^n subsequences, we limit the length of
-- the lists we test, so that the tests complete in a reasonable
-- amount of time.
```

3. *(12 points)* Consider the following function definitions:

```
checkEqn :: Equation -> Bool
checkEqn (Eqn e1 e2) = eval e1 == eval e2

eval :: Expr -> Int
eval (Num x) = x
eval (Op op e1 e2) = evalOp op (eval e1) (eval e2)

evalOp :: Oper -> Int -> Int -> Int
evalOp Add = (+)
evalOp Sub = (-)
evalOp Mul = (*)
```

(a) *(3 points)* Give the data type definitions needed for the above function definitions to be correct.

(b) *(3 points)* Define a QuickCheck test data generator

```
rExpr :: Int -> Gen Expr
```

such that `rExpr n` generates random expressions containing `n` operators. For the numbers in the expressions, use random ones from the sequence `1`, `2` ... `10`.

(c) *(3 points)* Define a function

```
exprs :: [Int] -> [Expr]
```

that generates *all* expressions that contain the given numbers in the given order. Examples:

```
exprs [1] == [Num 1]
exprs [1,2] ==
      [Op Add (Num 1) (Num 2),Op Sub (Num 1) (Num 2), -- 1+2, 1-2
       Op Mul (Num 1) (Num 2)]                        -- 1*2,
```

(d) *(3 points)* Define a function

```
equations :: [Int] -> [Equation]
```

that generates *all* equations that are true and contain the given numbers in the given order. Examples:

```
equations [1,2,3] ==
          [Eqn (Op Add (Num 1) (Num 2)) (Num 3)]  -- 1 + 2 = 3
equations [3,2,1] ==
          [Eqn (Num 3) (Op Add (Num 2) (Num 1)),  -- 3 = 2 + 1
           Eqn (Op Sub (Num 3) (Num 2)) (Num 1)]  -- 3 - 2 = 1
```

Hints: (i) Generating a list of expressions with list comprehensions is very similar to generating a random expression with the `Gen` monad and the `do` notation. (ii) A helper function that generates pairs of expressions can be useful in both `exprs` and `equations`.

Solution:

```
-- (a)
data Equation = Eqn Expr Expr                deriving (Eq,Show)
data Expr = Num Int | Op Oper Expr Expr      deriving (Eq,Show)
data Oper = Add | Sub | Mul                  deriving (Eq,Show)
-- deriving (Eq,Show) can be useful for testing, but is not needed
-- for anything else in this solution

-- (b)
rExpr 0 = Num <$> choose (1,10)
rExpr n = do op <- elements [Add,Sub,Mul]
             m <- choose (0,n-1)
             e1 <- rExpr m
             e2 <- rExpr (n-1-m)
             return (Op op e1 e2)

-- (c)
exprs []  = []
exprs [x] = [Num x]
exprs xs  = [Op op e1 e2 | (e1,e2) <- twoExprs xs,
                           op <- [Add,Sub,Mul]]

twoExprs xs = [(e1,e2) | i <- [1..length xs-1],
                         let (xs1,xs2) = splitAt i xs,
                         e1 <- exprs xs1,
                         e2 <- exprs xs2]

-- (d)
equations xs = [eqn | (e1,e2) <- twoExprs xs,
                      let eqn = Eqn e1 e2,
                      checkEqn eqn]
```

4. *(8 points)* Consider the following data type for trees where the leaves contain one type of values and the internal nodes contain another type of values:

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

   (a) *(3 points)* Define a function that corresponds to `map` for lists:

```
mapTree :: (a1->a2) -> (b1->b2) -> Tree a1 b1 -> Tree a2 b2
```

   (b) *(2 points)* Define a function that "folds" a tree which has functions in the internal nodes. (So unlike `foldr` for lists, the function used to combine values is not given as an extra argument.)

```
foldTree :: Tree a (a->a->a) -> a
```

   (c) *(3 points)* Reimplement `eval` from Question 3 by first converting an `Expr` to a `Tree Int Oper`, then using `mapTree` and `foldTree`.

```
eval_v2 :: Expr -> Int
```

Solution:

```
-- (a)
mapTree lf nf (Leaf b) = Leaf (lf b)
mapTree lf nf (Node a t1 t2) = Node (nf a) (mapTree lf nf t1)
                                           (mapTree lf nf t2)
-- (b)
foldTree (Leaf a)       = a
foldTree (Node f t1 t2) = f (foldTree t1) (foldTree t2)

-- (c)
eval_v2 = foldTree . mapTree id evalOp . convert
  where
    convert :: Expr -> Tree Int Oper
    convert (Num x) = Leaf x
    convert (Op op e1 e2) = Node op (convert e1) (convert e2)

-- Extra:
prop_eval e = eval_v2 e == eval e

instance Arbitrary Expr where
  arbitrary = sized (\n->rExpr =<< choose (0,n))
```

5. *(4 points)* Write a function that reads lines of text form a number of files and outputs all the lines sorted.

```
sortFiles :: [FilePath] -> IO ()
```

Example: `sortFiles ["A.txt","B.txt"]`

A.txt	B.txt	Output
PHP	Lisp	C
Haskell	C	Erlang
Python	Java	Haskell
Erlang		Java
		Lisp
		PHP
		Python

In addition to the functions listed at the back of this exam, the following library function might be useful:

```
-- Functions to output text
putStr, putStrLn :: String -> IO ()
```

```
-- readFile reads the contents of a file
readFile :: FilePath -> IO String

-- File names are strings.
type FilePath = String
```

Solution:

```
sortFiles paths = do ls <- concatMap lines <$> mapM readFile paths
                     putStr (unlines (sort ls))
```

```haskell
{- This is a list of selected functions from the
   standard Haskell modules: Prelude Data.List
   Data.Maybe Data.Char Control.Monad -}
-----------------------------------------
-- standard type classes

class Show a where
  show :: a -> String

class Eq a where
  (==), (/=) :: a -> a -> Bool

class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

class (Eq a, Show a) => Num a where
  (+), (-), (*)    :: a -> a -> a
  negate           :: a -> a
  abs, signum      :: a -> a
  fromInteger      :: Integer -> a

class (Num a, Ord a) => Real a where
  toRational       ::  a -> Rational

class (Real a, Enum a) => Integral a where
  quot, rem        :: a -> a -> a
  div, mod         :: a -> a -> a
  toInteger        :: a -> Integer

class (Num a) => Fractional a where
  (/)              :: a -> a -> a
  fromRational     :: Rational -> a

class (Fractional a) => Floating a where
  exp, log, sqrt      :: a -> a
  sin, cos, tan       :: a -> a

class (Real a, Fractional a) => RealFrac a where
  truncate, round :: (Integral b) => a -> b
  ceiling, floor  :: (Integral b) => a -> b

---------------------------------------------
-- numerical functions

even, odd        :: (Integral a) => a -> Bool
even n           = n `rem` 2 == 0
odd              = not . even

---------------------------------------------
-- monadic functions
sequence     :: Monad m => [m a] -> m [a]
sequence     = foldr mcons (return [])
  where mcons p q = do x <- p
                       xs <- q
                       return (x:xs)

sequence_    :: Monad m => [m a] -> m ()
sequence_ xs = do sequence xs
                  return ()

liftM   :: (Monad m) => (a1 -> r) -> m a1 -> m r
liftM f m1   = do x1 <- m1
                  return (f x1)
-----------------------------------------
-- functions on functions

id             :: a -> a
id x           = x

const          :: a -> b -> a
const x _      = x

(.)            :: (b -> c) -> (a -> b) -> a -> c
f . g          = \ x -> f (g x)

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

($)            :: (a -> b) -> a -> b
f $  x         = f x
-----------------------------------------
-- functions on Bools

data Bool = False | True

(&&), (||)       :: Bool -> Bool -> Bool
True  && x       = x
False && _       = False
True  || _       = True
False || x       = x

not              :: Bool -> Bool
not True         = False
not False        = True
-----------------------------------------
-- functions on Maybe

data Maybe a = Nothing | Just a

isJust                 :: Maybe a -> Bool
isJust (Just a)        =  True
isJust Nothing         =  False

isNothing              :: Maybe a -> Bool
isNothing              =  not . isJust

fromJust               :: Maybe a -> a
fromJust (Just a)      =  a

maybeToList            :: Maybe a -> [a]
maybeToList Nothing    =  []
maybeToList (Just a)   =  [a]

listToMaybe            :: [a] -> Maybe a
listToMaybe []         =  Nothing
listToMaybe (a:_)      =  Just a

catMaybes              :: [Maybe a] -> [a]
catMaybes ls           = [x | Just x <- ls]

-----------------------------------------
-- functions on pairs

fst              :: (a,b) -> a
fst (x,y)        =  x

snd              :: (a,b) -> b
snd (x,y)        =  y

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

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

uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f p    =  f (fst p) (snd p)
-----------------------------------------
-- functions on lists

map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]

(++) :: [a] -> [a] -> [a]
xs ++ ys = foldr (:) ys xs

filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]

concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f

head, last       :: [a] -> a
head (x:_)       = x

last [x]         = x
last (_:xs)      = last xs

tail, init       :: [a] -> [a]
tail (_:xs)      = xs

init [x]         = []
init (x:xs)      = x : init xs

null             :: [a] -> Bool
null []          = True
null (_:_)       = False

length           :: [a] -> Int
length           = foldr (const (1+)) 0

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

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

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

iterate          :: (a -> a) -> a -> [a]
iterate f x      =  x : iterate f (f x)

repeat           :: a -> [a]
repeat x         =  xs where xs = x:xs

replicate        :: Int -> a -> [a]
replicate n x    =  take n (repeat x)

cycle            :: [a] -> [a]
cycle []         =  error "Prelude.cycle: empty list"
cycle xs         =  xs' where xs' = xs ++ xs'
```

```
tails           :: [a] -> [[a]]
tails xs        =  xs : case xs of
                          []      -> []
                          _ : xs' -> tails xs'

take, drop             :: Int -> [a] -> [a]
take n _      | n <= 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n-1) xs

drop n xs     | n <= 0 =  xs
drop _ []              =  []
drop n (_:xs)          =  drop (n-1) xs

splitAt                :: Int -> [a] -> ([a],[a])
splitAt n xs           =  (take n xs, drop n xs)

takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p []         =  []
takeWhile p (x:xs)
            | p x      =  x : takeWhile p xs
            | otherwise =  []

dropWhile p []         =  []
dropWhile p xs@(x:xs')
            | p x      =  dropWhile p xs'
            | otherwise =  xs

span :: (a -> Bool) -> [a] -> ([a], [a])
span p as = (takeWhile p as, dropWhile p as)

lines, words     :: String -> [String]
-- lines "apa\nbepa\ncepa\n"
--   == ["apa","bepa","cepa"]
-- words "apa  bepa\n cepa"
--   == ["apa","bepa","cepa"]

unlines, unwords :: [String] -> String
-- unlines ["apa","bepa","cepa"]
--       == "apa\nbepa\ncepa"
-- unwords ["apa","bepa","cepa"]
--       == "apa bepa cepa"

reverse          :: [a] -> [a]
reverse          =  foldl (flip (:)) []

and, or          :: [Bool] -> Bool
and              =  foldr (&&) True
or               =  foldr (||) False

any, all         :: (a -> Bool) -> [a] -> Bool
any p            =  or . map p
all p            =  and . map p

elem, notElem    :: (Eq a) => a -> [a] -> Bool
elem x           =  any (== x)
notElem x        =  all (/= x)

lookup      :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup key []    =  Nothing
lookup key ((x,y):xys)
    | key == x   =  Just y
    | otherwise  =  lookup key xys
```

```
sum, product     :: (Num a) => [a] -> a
sum              =  foldl (+) 0
product          =  foldl (*) 1

maximum, minimum :: (Ord a) => [a] -> a
maximum [] = error "Prelude.maximum: empty list"
maximum (x:xs) = foldl max x xs

minimum [] = error "Prelude.minimum: empty list"
minimum (x:xs) = foldl min x xs

zip              :: [a] -> [b] -> [(a,b)]
zip              =  zipWith (,)

zipWith          :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)
                 =  z a b : zipWith z as bs
zipWith _ _ _    =  []

unzip            :: [(a,b)] -> ([a],[b])
unzip            =
  foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])

nub              :: Eq a => [a] -> [a]
nub []           = []
nub (x:xs)       = x : nub [ y | y <- xs, x /= y ]

delete           :: Eq a => a -> [a] -> [a]
delete y []      = []
delete y (x:xs)  =
       if x == y then xs else x : delete y xs

(\\)             :: Eq a => [a] -> [a] -> [a]
(\\)             = foldl (flip delete)

union            :: Eq a => [a] -> [a] -> [a]
union xs ys      = xs ++ (ys \\ xs)

intersect        :: Eq a => [a] -> [a] -> [a]
intersect xs ys  = [ x | x <- xs, x `elem` ys ]

intersperse      :: a -> [a] -> [a]
-- intersperse 0 [1,2,3,4] == [1,0,2,0,3,0,4]

transpose        :: [[a]] -> [[a]]
-- transpose [[1,2,3],[4,5,6]]
--    == [[1,4],[2,5],[3,6]]

partition   :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs =
    (filter p xs, filter (not . p) xs)

group            :: Eq a => [a] -> [[a]]
group = groupBy (==)

groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
groupBy _  []     =  []
groupBy eq (x:xs) =  (x:ys) : groupBy eq zs
                     where (ys,zs) = span (eq x) xs

isPrefixOf :: Eq a => [a] -> [a] -> Bool
isPrefixOf []     _      =  True
isPrefixOf _      []     =  False
isPrefixOf (x:xs) (y:ys) =  x == y
                            && isPrefixOf xs ys
```

```
isSuffixOf :: Eq a => [a] -> [a] -> Bool
isSuffixOf x y = reverse x `isPrefixOf` reverse y

sort             :: (Ord a) => [a] -> [a]
sort             = foldr insert []

insert           :: (Ord a) => a -> [a] -> [a]
insert x []      = [x]
insert x (y:xs)  =
    if x <= y then x:y:xs else y:insert x xs

-----------------------------------------
-- functions on Char

type String = [Char]

toUpper, toLower :: Char -> Char
-- toUpper 'a' == 'A'
-- toLower 'Z' == 'z'

digitToInt :: Char -> Int
-- digitToInt '8' == 8

intToDigit :: Int -> Char
-- intToDigit 3 == '3'

ord :: Char -> Int
chr :: Int  -> Char

-----------------------------------------
-- Signatures of some useful functions
-- from Test.QuickCheck

arbitrary :: Arbitrary a => Gen a
-- the generator for values of a type
-- in class Arbitrary, used by quickCheck

choose :: Random a => (a, a) -> Gen a
-- Generates a random element in the given
-- inclusive range.

oneof :: [Gen a] -> Gen a
-- Randomly uses one of the given generators

frequency :: [(Int, Gen a)] -> Gen a
-- Chooses from list of generators with
-- weighted random distribution.

elements :: [a] -> Gen a
-- Generates one of the given values.

listOf :: Gen a -> Gen [a]
-- Generates a list of random length.

vectorOf :: Int -> Gen a -> Gen [a]
-- Generates a list of the given length.

sized :: (Int -> Gen a) -> Gen a
-- construct generators that depend on
-- the size parameter.
```