Chalmers | Göteborgs Universitet 2019-04-25

Examiner: Thomas Hallgren, D&IT,
Answering questions at approx 14.30 (or by phone)

Functional Programming TDA 452, DIT 143

2019-04-25 14.00 – 18.00 "Maskin"-salar (M)

- There are 4 questions with maximum $8 + 12 + 12 + 8 = 40$ 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 y     = y
fb x y     = x++[y]
fc f (x,y) = (f x,f y)
fd n k     = product [n | _ <- [1..k]]
```

Solution:

```
fa :: a -> b -> b
fb :: [a] -> a -> [a]
fc :: (a->b) -> (a,a) -> (b,b)
fd :: (Num a,Num b,Enum b) => a -> b -> a
```

2. *(12 points)*

   (a) *(3 points)* Define data types `Rank`, `Suit` and `Card` for representing the cards used in card games. There are four suits: hearts, spades, diamonds and clubs. There are 13 ranks: the numeric ranks 2, 3 ... 10, and the ranks of face cards: jack, queen, king and ace. Every card has a rank and a suit, so in total there are 52 different cards.

   Make sure that the data types are "junk free", i.e. all possible values of type `Card` should represent valid cards.

   (b) *(3 points)* Define random test data generators for suits, rands and cards:

   ```
   rSuit :: Gen Suit
   rRank :: Gen Rank
   rCard :: Gen Card
   ```

   (c) *(3 points)* Let a hand of cards be represented as a list of cards.

   ```
   type Hand = [Card]
   ```

   Write a test data generator that generates a random hand of a given size. The generated hand should not contain the same card more than once.

   ```
   rHand :: Int -> Gen Hand
   ```

   (d) *(3 points)* Write a property `prop_rHand_correct` to verify that hands generated by `rHand` have the correct size and does not contain the same card more than once.

   Hint: Use the QuickCheck function `forAll` to generate random sizes in a suitable range and random hands of that randomly chosen size.

   ```
   forAll :: (Testable prop, Show a) => Gen a -> (a -> prop) -> Property
   ```

Solution:

```
-- (a)
data Suit = Hearts | Spades | Diamonds | Clubs
            deriving (Eq, Bounded, Enum, Show)
data NumericRank = N2 | N3 | N4 | N5 | N6 | N7 | N8 | N9 | N10
            deriving (Eq, Bounded, Enum, Show)
data Rank = Numeric NumericRank | Jack | Queen | King | Ace
            deriving (Eq, Show)
data Card = Card Rank Suit
            deriving (Eq, Show)

-- (b)
rSuit = elements [minBound .. maxBound]
rRank = elements (map Numeric [N2 .. N10] ++ [Jack, Queen, King, Ace])
rCard = Card <$> rRank <*> rSuit

-- (c)
rHand = rHand' []
rHand' h 0 = return h
rHand' h n = do c <- rCard
                if c `elem` h      -- same card?
                  then rHand' h n  -- discard it and try again
                  else rHand' (c:h) (n-1)
-- (d)
prop_rHand_correct = forAll (choose (0,52)) $ \ n ->
                     forAll (rHand n) $ \ h ->
                     nub h == h && length h==n
```

3. *(12 points)* Consider the following data type for representing rectangular grids:

```
data Grid a = Grid [[a]]  deriving (Eq,Show)
type Pos    = (Int,Int)   -- (x,y)
type Size   = (Int,Int)   -- (width,height)

g1 :: Grid Int      -- Example
g1 = Grid [[3,4,5],
           [6,7,8]]
```

(a) *(2 points)* Define an indexing operator

```
(!) :: Grid a -> Pos -> a
```

that returns the element at the given coordinates in a grid. In a grid of size `(w,h)` the coordinates of the top left and bottom right corners are `(0,0)` and `(w-1,h-1)`, respectively. Examples:

```
g1 ! (0,0) == 3
g1 ! (2,1) == 8
```

(b) *(2 points)* Define a function that applies a function to every element of a grid.

```
mapGrid :: (a->b) -> Grid a -> Grid b
```

Example:

```
mapGrid even g1 == Grid [[False,True,False],[True,False,True]]
```

(c) *(3 points)* Define a function that, given the size of a grid and a position in the grid, computes the positions of the neighbours in the grid:

```
neighbours :: Size -> Pos -> [Pos]
```

A position has up to 8 neighbours (moving horizontally, vertically and diagonally), but positions in the corners and along the edges have fewer neighbours. Examples:

```
neighbours (3,3) (1,1) == [(0,0),(0,1),(0,2),(1,0),(1,2),(2,0),(2,1),(2,2)]
neighbours (3,3) (0,0) == [(0,1),(1,0),(1,1)]
neighbours (3,3) (2,1) == [(1,0),(1,1),(1,2),(2,0),(2,2)]
```

(d) *(3 points)* Define a function that computes neighbourhoods, i.e. a grid where every element is replaced with the list of its neighbour elements.

```
neighbourhoods :: Grid a -> Grid [a]
```

(e) *(2 points)* Define a function that counts how many of the neighbours of each element in a grid of booleans are `True`.

```
countNeighbours :: Grid Bool -> Grid Int
```

Example:

```
countNeighbours (mapGrid even g1) == Grid [[2,2,2],
                                           [1,3,1]]
```

Solution:

```
-- (a)
Grid rows ! (x,y) = rows !! y !! x

-- (b)
mapGrid f (Grid rows) = Grid $ map (map f) rows

-- (c)
neighbours (w,h) (x0,y0) =
    [(x,y) | x<-range x0 w, y<-range y0 h, (x,y)/=(x0,y0)]
  where
    range mid limit = [i | i<-[mid-1 .. mid+1], 0<=i && i<limit]

-- (d)
neighbourhoods g = mapGrid (map (g!) . neighbours s) (identityGrid s)
  where s = size g

size :: Grid a -> Size
size (Grid rows@(row:_)) = (length row,length rows)

-- | identityGrid s ! p == p
identityGrid :: Size -> Grid Pos
identityGrid (w,h) = Grid [[(x,y) | x<-[0..w-1]] | y<-[0..h-1]]

-- (e)
countNeighbours = mapGrid count . neighbourhoods
  where
    count = length . filter id
```

4. *(8 points)*

   (a) *(3 points)* Define a function `segments` that splits a list into segments.

   ```
   segments :: (a->Bool) -> [a] -> [[a]]
   ```

   Examples:

   ```
   segments (==';') "abc;def  ;g hi " = ["abc", "def  ", "g hi "]
   segments isSpace "abc;def  ;g hi " = ["abc;def", "", ";g", "hi"]
   ```

   (Note: there are four space characters in the example string: two after `f`, one after `g` and one after `i`.)

   (b) *(5 points)* Consider files containing the scores that some players obtained while playing a game:

   ```
   Player 1,10,30,40
   Player 2,30,20,15
   ```

Each line is a sequence of comma-separated values, where the first value identifies the player and the remaining values are scores.

Define the function `addSumsToFile`

```
addSumsToFile :: String -> IO ()
```

that reads a file containing scores as outlined above and writes a file where the sum of the scores for each player has been added as the second value in each line. For example, `addSumsToFile "scores"` should read the file `scores.csv` and write the output to `scores-sum.csv`. If `scores.csv` contains the data above, then the following data should be written to `scores-sum.csv`:

```
Player 1,80,10,30,40
Player 2,65,30,20,15
```

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

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

-- writeFile writes contents to a file
writefile :: FilePath -> String -> IO ()

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

Solution:

```
-- (a)
segments p [] = []
segments p xs = case break p xs of
                  (xs1,xs2) -> xs1:segments p (drop 1 xs2)

-- (b)
addSumsToFile path =
  do s <- readFile (path++".csv")
     let convert = toCSV . map addSum . fromCSV
     writeFile (path++"-sum.csv") (convert s)

addSum :: [String] -> [String]
addSum (name:scores) = name:show (sum (map read scores)):scores

fromCSV :: String -> [[String]]
fromCSV = map (segments (==',')) . lines

toCSV :: [[String]] -> String
toCSV = unlines . map (separate ',')

separate :: a -> [[a]] -> [a]
separate sep [] = []
separate sep [x] = x
separate sep (x:xs) = x++sep:separate sep xs
```

```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)
----------------------------------------
```

```haskell
-- 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)
```

```haskell
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]
```

```haskell
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
```

```haskell
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
```

```haskell
                       && 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.
```