Chalmers | Göteborgs Universitet 2016-04-07

Examiner: David Sands dave@chalmers.se.
Answering questions on the day of the exam (at approx 15.00):
Gregoire Detrez (tel: 073 55 69 550) – and at other times by phone.
, D&IT

Functional Programming TDA 452, DIT 142

2016-04-07 14.00 – 18.00 "Maskin"-salar (M)

Contact on the day, Gregoire: 073 55 69 550

- There are 4 Questions with maximum $10 + 10 + 12 + 8 = 40$ points; a total of 20 points definitely guarantees a pass.
- Results: latest Within approximately 20 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!
  - 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.
  - 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.*
  - 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.

Question 1. *(10 points)*

(i) *(4 points)* Give a definition of a function

```
range :: Ord a => [a] -> (a,a)
```

which computes the largest and smallest values (as a pair) of the values in the given list. You may assume that the argument to range is non-empty. Example:

```
prop_range = range [9,8,0,6] == (0,9) && range [1] == (1,1)
```

For full points your definition should use *a single tail-recursive helper function.* Other correct solutions give max 2 points. **Solution**

```
range (n:ns) = r (n,n) ns where
  r (s,b) []     = (s,b)
  r (s,b) (m:ms) = r (s `min` m, b `max` m) ms
```

(ii) *(4 points)* Define a function

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

which splits its second argument list into chunks at every element in the first list, and satisfies the following property:

```
prop_splitOneOf0 :: Eq a => [a] -> Bool
prop_splitOneOf0 as =
    splitOneOf ";:" "A;BB:;DDDD:"    == ["A","BB","","DDDD",""]
 && splitOneOf [2,3,4] [3,0,1,2,0,0] == [[],[0,1],[0,0]]
 && splitOneOf as []                 == [[]]
 && splitOneOf [] as                 == [as]
```

Hint: a recursive definition over the second argument using `span` may be simplest. **Solution**

```
splitOneOf sep xs =
     case span (`notElem` sep) xs of
         (c, [])  -> [c]
         (c, _:r ) -> c : splitOneOf sep r
```

(iii) *(2 points)* Describe the expected property of the expression `length (splitOneOf sep xs)` as a function

```
prop_splitOneOf :: [Int] -> [Int] -> Bool
```

Solution

```
prop_splitOneOf sep as =
  length (splitOneOf sep as) == length (filter (`elem` sep) as) + 1
```

Question 2. *(10 points)*

(i) *(6 points)* For each of the following functions, give the most general type.

```
fa m n l =  m `lookup` zip l n

fb []     = []
fb (b:c)  = (map b) : fb c

fc (a:b) (c:d) = a /= c && fc b d
fc _      _    = True
```

Solution

```
fa :: Eq a => a -> [b] -> [a] -> Maybe b
fb :: [a -> b] ->  [[a]->[b]]
fc :: Eq t => [t] -> [t] -> Bool
```

(ii) *(4 points)* Functions `fb` and `fc` are explicitly recursive. Rewrite them so that they use standard functions (listed at the back of this exam) and/or list comprehensions, instead of explicit recursion.

Solution

```
fb' = map map
fc' as bs = and $ zipWith (/=) as bs
```

Question 3. *(12 points)* A Sudoku puzzle consists of a 9x9 grid. Some of the cells in the grid have digits (from 1 to 9), others are blank. The objective of the puzzle is to fill in the blank cells with digits from 1 to 9, in such a way that every row, every column and every 3x3 block has exactly one occurrence of each digit 1 to 9.

In lab 3 you represented a sudoku board using the type `data Sudoku = Sudoku [[Maybe Int]]`. In this question you are to use a simpler type

```
type Sudoku = [[Int]]
```

In this representation, `0` represents the blank cell. An example sudoku is

```
ex = [[3,6,0,0,7,1,2,0,0],
      [0,5,0,0,0,0,1,8,0],
      [0,0,9,2,0,4,7,0,0],
      [0,0,0,0,1,3,0,2,8],
      [4,0,0,5,0,2,0,0,9],
      [2,7,0,4,6,0,0,0,0],
      [0,0,5,3,0,8,9,0,0],
      [0,8,3,0,0,0,0,6,0],
      [0,0,7,6,9,0,0,4,3]]
```

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

```
allBlanks :: Sudoku -> [(Int,Int)]
```

that returns the positions of all blank cells in the given Sudoku. For example, `allBlanks ex` includes (amongs others) the positions `[(2,0),(3,0),(7,0),(8,0)]` corresponding to the blanks in the first row.

You may assume that the sudoku is well-formed. **Solution**

```
allBlanks s =
  [(x,y) | (y,row) <- zip [0..] s,
           (x,0  ) <- zip [0..] row  ]
```

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

```
updateWith :: Int -> (Int,Int) -> Sudoku -> Sudoku
```

where `updateWith n (x,y) s` produces a Sudoku from `s` by updating the position `(x,y)` with the value `n`. So for example, `updateWith 4 (2,0) ex` would result in the sudoku which is like `ex` except that it has a 4 instead of a blank (0) at the first blank on the first row. You may assume that the position is always inside the given list of rows. **Solution**

```
updateWith n (x,y) rs =
 rowsBefore ++ [cellsBefore ++ [n] ++ cellsAfter] ++ rowsAfter
   where
    (rowsBefore, row:rowsAfter) = splitAt y rs
    (cellsBefore, _:cellsAfter) = splitAt x row
```

(iii) *(6 points)*

```
arbPuzzle :: Sudoku -> Int -> Gen Sudoku
```

Which given a Sudoku s and a positive integer n (where we assume that there are at least n non-blank cells in s) gives a quickCheck generator for random Sudoku obtained from s by making exactly n non-blank cells into blanks. **Solution**

```
arbPuzzle s n = do
   delList <- take n `fmap` shuffle nonblanks
   return $ foldr (updateWith 0) s delList

  where nonblanks = [(x,y) | x <- [0..8], y <- [0..8]] \\ allBlanks s

shuffle [] = return []
shuffle xs = do
  a <- elements xs
  as <- shuffle (delete a xs)
  return $ a : as
```

Question 4. *(8 points)*

(i) *(4 points)* Define a Haskell datatype `AExpr` to represet arithmetic expressions including the following kind of expression:

```
if x > 3.14 && y < 0 then sin (y + 1) else cos (y * 3)
```

Your type should permit any arithmetic expressions built from floating point numbers, variables (assumed to be of type Float), arithmetic operators (`+`,`*`,`-`,`^`), unary operators (`sin`, `cos`, `tan`, `abs`), if-then-else, comparison operators (`==`, $>$, $<$, $\leq$, $\geq$), and boolean combinations (`&&`, `||`).

You should make use the following types in your definition:

```
type VarName = String

data BinaryOp  = Add | Mul | Sub | Power
data UnaryOp   = Sin | Cos | Tan | Abs
data CompOp    = GT  | LT  | GEQ | LEQ -- (>, <, >=, <=)
data LogicOp   = And | Or
```

Solution

```
data AExp = Var VarName | If BExp AExp AExp | Num Double
          | Binary BinaryOp AExp AExp | Unary UnaryOp AExp

data BExp = Compare CompOp  AExp AExp
          | Combine LogicOp BExp BExp
```

Your type should *not* allow invalid arithmetic expressions, such as the following, to be represented:

`if x then 1 else 0` (since x represents a number)

`x > 3.14` (since the result is not a floating-point number)

`2 && 3` (since numbers are not boolean expressions)

`(1 > 0) > (1 < 0)` (since comparison is only allowed on numbers)

(ii) *(4 points)* Define a function

```
vars :: AExp -> [VarName]
```

which, given an expression computes a list of all the variable names appearing in that expression. A variable name should not appear in the result more than once.

Solution

```
vars e = nub $ avars e
   where avars (Var s)       = [s]
         avars (If b a1 a2)  = bvars b ++ avars a1 ++ avars a2
         avars (Binary _ a1 a2) = avars a1 ++ avars a2
         avars (Unary _ a)   = avars a

         bvars (Compare _ a1 a2) = avars a1 ++ avars a2
         bvars (Combine _ b1 b2) = bvars b1 ++ bvars b2
```

```
{-
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.
```