Chalmers | Göteborgs Universitet 2013-12-17

David Sands, D&IT

Functional Programming TDA 452/451, DIT 142/141

2013-12-17 08.30 – 12.30 *M/maskin*

David Sands, 0737 207663

- There are 4 Questions with maximum $8 + 9 + 15 + 16 = 48$ points; a total of 22 points definitely guarantees a pass.
- Results: latest within 21 days.
- The examiner will visit the examination rooms at approximately 09.30 - 09.45 and one more time during mid morning.
- **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.

There are only two difficulties with Haskell programming:
predicting space behaviour, monads, and off-by-one errors.

Question 1. In this question we assume that there exits a function `rainfall :: Int -> Float` which gives the weekly rainfall (in mm) in Gothenburg starting from a particular year, where weeks are numbered in sequence 1, 2,. . .

(a) *(5 points)* Define two versions of the function `maxRainfall :: Int -> Float` where `maxRainfall n` gives the maximum rainfall in any week in the range `[1..n]`. Your definition may assume that `n > 0`:

(i) Give a definition using `foldr`, simple enumeration lists (e.g., `[1..n]`), the function `max`, and no other recursive functions, list comprehensions, or prelude functions.

(ii) Give a definition using a *tail-recursive helper-function*, addition, and the function `max`, but *without* using any other recursive functions, list comprehensions, enumerations (like `[1..n]`) or prelude functions.

(b) *(3 points)* The function `maxWeeks :: Int -> [Int]` is supposed to behave as follows: given a week number n (assumed to be greater than zero) it returns the list of all week numbers in the range 1,...,n which had the maximal rainfall for that period. So for example if the largest rainfall in the first 9 weeks was 29.4, then `maxWeeks 9` should return all the week numbers in the range 1 to 9 which had rainfall of 29.4.

Define `maxWeeks` using only list comprehensions, the list `[1..n]`, the prelude function `maximum`, `==`, and no other recursive functions or prelude functions. You may not use the functions defined in part (a).

Solution

```
maxRainfall n = foldr maxrain 0 [1..n]
        where maxrain w r = rainfall w `max` r

maxRainfall' n = maxr 0 n
 where
  maxr maxSoFar w
         | w <= 0    = maxSoFar
         | otherwise = maxr (maxSoFar `max` rainfall w) (w-1)


maxWeeks n = [i | i <- [1..n], rainfall i == maximum [rainfall j | j <- [1..n]] ]
```

Question 2. (a) *(5 points)* Give the most general types of the following five functions:

```
fa x y z = x z && y z

fb m = Just (m+1)

fc (x:y:xs) | x > y = 999
fc _              = 0

fd (x:xs) (y:ys) = x == ys
fd []     ys     = ys == []

fe x y = do
     z <- x
     return $ z:y
```

Solution

```
fa :: (t -> Bool) -> (t -> Bool) -> t -> Bool
fb :: Num a => a -> Maybe a
fc :: (Num a, Ord b) => [b] -> a
fd :: Eq t => [[t]] -> [t] -> Bool
fe :: Monad m => m a -> [a] -> m [a]
```

(b) *(2 points)* Rewrite the following definition without do notation:

```
ff x y = do
  c <- readFile x
  d <- readFile c
  putStr (c ++ "\n" ++ d)
```

Solution

```
ff' x y = readFile x >>= \c ->
           readFile c >>= \d ->
             putStr (c ++ "\n" ++ d)
```

(c) *(2 points)* Rewrite the following using do notation:

```
fg a tb tc td
     = case lookup a tb of
             Nothing -> Nothing
             Just b -> case lookup b tc of
                  Nothing -> Nothing
                  Just c ->
                     case lookup c td of
                             Nothing -> Nothing
                             Just d -> Just [b,c,d]
```

Solution

```
fg' a tb tc td = do
  b <- lookup a tb
  c <- lookup b tc
  d <- lookup c td
  Just [b,c,d]
```

Question 3. A *bag*, or *multiset*, is a collection of elements like a set, but where an element can occur more than once. The following data type will be used to represent a bag:

```
data Bag a = EmptyBag | Node a Int (Bag a) (Bag a)
```

The idea is that any Bag containing a node of the form `Node elem count left right` represents a bag containing `count` copies of the element `elem`. The following data type invariants should be maintained (and thus may be assumed) for any such node in any Bag built in your implementation:

- all the elements in `left` are strictly smaller than `elem`,
- all the elements in `right` are strictly larger than `elem`, and
- the count `count` is greater than or equal to zero.

These invariants mean that a bag is a variant of a so-called *binary search tree*.

(a) *(1 points)* In what situation would it be useful to have the following definition:

```
emptyBag :: Bag a
emptyBag = EmptyBag
```

Solution

```
-- when you want to hide the implementation of the datatype
-- (by not exporting the constructors).
```

(b) *(2 points)* Define the following function, by recursion on the Bag argument, to count how many times a given element appears in a bag:

```
bcount :: Ord a => a -> Bag a -> Int
```

(c) *(3 points)* Define a function which converts a bag to a list, with the correct number of copies of each element. Your function should produce a sorted list whenever the bag satisfies the data type invariant (note: you should not use the sort function to ensure this).

```
bagToList :: Bag a -> [a]
```

(d) *(4 points)* Define the data type invariant

```
prop_Bag :: Ord a => Bag a -> Bool
```

which checks whether the given bag satisfies the data type invariant.

(e) *(5 points)* Define

```
bchange :: Ord a => a -> Int -> Bag a -> Maybe (Bag a)
```

which changes the number of occurrences of a given element by the given amount. If there are not sufficiently many elements in the bag to complete the operation then the function returns `Nothing`.

For example, if `bagToList b` is `[9,9,10]` then the following is True:

```
map (fmap bagToList)  [bchange 11 1 b, bchange 9 (-2) b, bchange 9 (-4) b]
      == [Just [9,9,10,11], Just [10], Nothing]
```

Solution

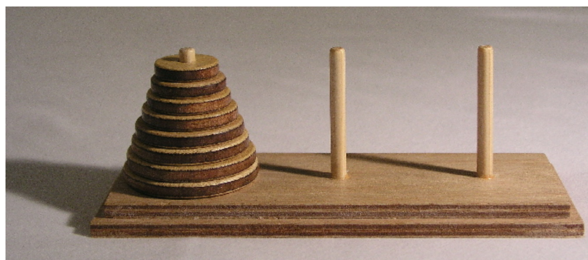
```
bcount x EmptyBag = 0
bcount x (Node e c left right)
      | x == e    = c
      | x <  e    = bcount x left
      | x >  e    = bcount x right

bagToList EmptyBag = []
bagToList (Node e c l r) = bagToList l ++ replicate c e ++ bagToList r

prop_Bag EmptyBag = True
prop_Bag (Node a c left right) = all (<a) (bagToList left)
                              && all (>a) (bagToList right)
                              && all prop_Bag [left,right] && c >= 0
-- Note single-traversal O(n) solution is more tricky;
-- you cannot just recursively compare each element
-- with the highest elements on the left and right!

bchange x n b | newcount >= 0 = Just $ bc b
              | otherwise     = Nothing
  where
        newcount = bcount x b + n
        bc EmptyBag = Node x n EmptyBag EmptyBag
        bc (Node e c left right)
           | e == x = Node e newcount left right
           | x < e  = Node e c (bc left) right
           | x > e  = Node e c left (bc right)
```

Question 4. [This question has four parts, (a)–(d) on two printed pages] The Towers of Hanoi is an ancient puzzle, consisting of a collection of rings of different sizes, and three posts mounted on a base. At the beginning all the rings are on the left-most post as shown, and the goal is to move them all to the rightmost post, by moving one ring at a time from one post to another. But, at no time may a larger ring be placed on top of a smaller one!



In this question we will model games with any number of discs (the game in the picure above has 8) and any number of pegs (the game in the picture has 3), using the following data type:

```
type Pegs     = Int
type NumDiscs = Int
type Disc     = Int
data Hanoi = Hanoi Pegs NumDiscs [[Disc]]
             deriving (Eq,Show)
```

`Hanoi 3 4 [[1,2,3,4],[],[]]`

`Hanoi 3 4 [[2,3,4],[1],[]]`

`Hanoi 3 4 [[3,4], [1], [2]]`

`Hanoi 3 4 [[3,4], [], [1,2]]`

`Hanoi 3 4 [[4],  [3], [1,2]]`

`Hanoi 3 4 [[1,4],[3], [2]]`

The image to the right above shows a sequence of moves for a game with 4 discs and three "pegs" and the corresponding Haskell representations of each board.

(a) *(3 points)* Define a function `prop_wellFormedHanoi :: Hanoi -> Bool` such that `prop_wellFormedHanoi (Hanoi p d ps)` checks that the Hanoi board is well-formed: it contains exactly the discs `[1..d]`, `p` pegs, and on any peg there is never a larger disc on top of a smaller one. **Solution**

```
prop_wellFormedHanoi (Hanoi p d ps) = sort (concat ps) == [1..d]
                                   && length ps == p
                                   && all (\a -> a == sort a) ps
```

(b) *(1 points)* Define a function

```
emptyHanoi :: Pegs -> NumDiscs -> Hanoi
```

which given a number of discs and number of pegs, creates an empty Hanoi of those dimensions. Note that this Hanoi will intentionally not be well-formed if the number of discs is nonzero. **Solution**

```
emptyHanoi np nd = Hanoi np nd $ replicate np []
```

(c) *(6 points)* Define two functions

```
addDisc :: Disc -> Pegs -> Hanoi -> Hanoi
removeDisc :: Pegs -> Hanoi -> Maybe (Disc, Hanoi)
```

`addDisc` given a disc, a peg number, and a Hanoi, creates a Hanoi by adding the disc to the peg of the Hanoi. If the peg is outside the range of the Hanoi then your definition should leave the Hanoi unchanged. `removeDisc p h` attempts to

return the pair of the top disc from peg `p` of `h`, together with the resulting Hanoi. It returns `Nothing` if this is not possible.

Neither of these functions should change the `NumDiscs` dimension of the Hanoi (so in practice one would expect that only the Hanoi result of `addDisc`, or the Hanoi argument to `removeDisc`, to be well-formed – but you do not need to check this).

Solution

```
addDisc p d (Hanoi p' d' ds) = Hanoi p' d' $ modify (d:) p ds

removeDisc p (Hanoi p' d' ds)
   | p <= p' && p > 0 && not (null peg) = Just  $ (head peg, h)
   | otherwise                          = Nothing
     where peg = ds !! (p - 1)
           h   = Hanoi p' d' $ modify tail p ds

modify f p =
  zipWith (\n ds -> if p == n then  f ds else ds) [1..]
```

(d) *(6 points)* Give the definitions necessary to enable quickCheck to run on properties of well-formed Hanois. You may assume that Hanoi boards have between 2 and 10 discs and between 3 and 5 pegs.

Hint: one strategy for generating a well-formed Hanoi is to first pick a random number of discs d and a random number of pegs p; then starting with a row of p empty pegs, recursively add the elements from the list of discs `[1..d]`, one at a time, to randomly chosen pegs in the range `[1..p]`. Finally fix each peg so that the discs are stacked in the right order.

Solution

```
instance Arbitrary Hanoi where
 arbitrary = do
     pegs <- elements [3..5]
     discs <- elements [2..10]
     dist <- distribute pegs [1..discs] $ replicate pegs []
     return $ Hanoi pegs discs $ map sort dist

distribute _ [] bs     = return bs
distribute ps (v:vs) bs = do
                 p <- choose(1,ps)
                 distribute ps vs (modify (v:) p bs)
```

```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               = iterate tail
```

```haskell
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 p
arameter.
```