Chalmers | Göteborgs Universitet 2015-01-15

Examiner: David Sands `dave@chalmers.se`, D&IT

Functional Programming TDA 452, DIT 142

2015-01-15 14.00 – 18.00 "Väg och vatten" (VV)

Course assistants Simon Huber and Anders Mörtberg will be available to answer questions on the day of the examination: 031 772 5410 or 0730423376 (Anders)

- There are 4 Questions with maximum $10 + 8 + 10 + 14 = 42$ points; a total of 21 points definitely guarantees a pass.
- Results: latest within 21 days.
- A course assistant (Anders and/or Simon) will visit the examination rooms at approximately 15–15.30, and again at approximately 16.30.
- **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)* These questions refer to standard library functions (listed at the back)

(i) Give a definition of the function `filter` using recursion.

(ii) The function `last` function can be defined as follows:

```
last (x:xs) = foldl g x xs
```

for some suitable function `g`. Define `g`.

(iii) Give a definition of `lookup` using `filter`, `map`, and `listToMaybe`. You may define small helper functions, but the helper functions should not use recursion or list comprehensions.

(iv) Give a recursive definition of the function `sequence_` **without** using do-notation.

(v) Define `unzip` using recursion.

Solution

```
filter' p [] = []
filter' p (x:xs) | p x = x : filter' p xs
                 | otherwise = filter' p xs

g _ x = x  -- flip const

lookup' x = listToMaybe . map snd . filter xfirst
  where xfirst (a,_) = a == x

sequence_' [] = return ()
sequence_' (i:is) = i >> sequence_' is

unzip' [] = ([],[])
unzip' ((a,b):xs) = let (as,bs) = unzip' xs in (a:as,b:bs)
```

Question 2. *(8 points)* Give the most general types of the following four functions:

```
fa m n = Just (m > n)

fb x y z = z y + z x

fc (x:xs) (y:ys) = x == ys
fc []     ys     = null ys

fd x = do
     z <- x
     return $ replicate z z
```

Solution

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

Question 3. *(10 points)* The function `permu` is intended to produce the list of all permutations of its argument. For example,

```
Main> permu [1,2,3]
[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
```

The exact order of lists in the output is not important, and if there are repeated elements in the input then there may be repeated lists in the output. The following is an incomplete recursive definition of `permu`:

```
permu :: [a] -> [[a]]
permu [] = [[]]
permu (x:xs) = concatMap (insertAll x) $ permu xs
```

(i) *(2 points)* Give the type of the missing function `insertAll`

(ii) *(4 points)* Provide the missing definition of `insertAll` **Solution**

```
-- recursive
insertAll :: a -> [a] -> [[a]]
insertAll x []     = [[x]]
insertAll x (y:ys) = (x:y:ys) : map (y:) (insertAll x ys)

-- or using list comprehensions:
insertAll' x ys =
  [take n ys ++ [x] ++ drop n ys | n <- [0..length ys] ]
```

(iii) *(4 points)* Define a quickCheck property which tests that every element of the result of function `permu` is indeed a permutation of its argument. Your solution should not use sorting (e.g. via the function `sort`). Include any type declarations you deem necessary.

Solution

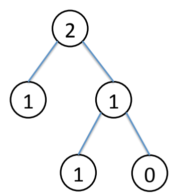
```
prop_permu xs = all (`isPermutationOf` xs) $ permu xs
  where types = xs :: [Bool] -- otherwise Haskel chooses [()]

isPermutationOf []     ys = null ys
isPermutationOf (x:xs) ys = x `elem` ys && xs `isPermutationOf` (delete x ys)
```

Question 4. *(14 points)* The following data type represents binary trees with elements of any type `a` at the nodes:

```
data T a = Leaf | Node a (T a) (T a)
  deriving Show
```

(i) *(2 points)* Write the expression which would be used to represent the tree pictured below. You are required to use code layout which make your solution easy to read!



Solution

```
ex = Node 2 t1 (Node 1 t1 t0)
  where t1 = Node 1 Leaf Leaf
        t0 = Node 0 Leaf Leaf
```

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

```
parents :: Eq a => a -> T a -> [a]
```

which computes the elements which are immediate parents of the given node element in the given tree. For example, in the tree pictured above, the parents of `1` are `[2,1]` and the parents of `0` are `[1]`, the parents of `2` is `[]`, and the parents of `3` is `[]`. The exact order in which the parents appear is not important. It is possible for a parent to appear in the result list more than once, but only because that parent occurs in more than one place in the tree. **Solution**

```
parents n Leaf = []
parents n (Node m t1 t2) = if n `isRootOf` t1 || n `isRootOf` t2 then [m] else []
                      ++ parents n t1 ++ parents n t2

n `isRootOf` Node m _ _ = n == m
n `isRootOf` Leaf       = False
```

(iii) *(4 points)* Define a quickCheck property for `parents` which specifies that each element of the result of `parents n t` should be found in `t`, and that the number of parents of an element is less than or equal to the number of times the element appears in the tree.

Solution

```
prop_parants n t =  all (`elem` ts) ps && length ps <= length (filter (==n) ts)
   where ps = parents n t
         ts = toList t
         toList Leaf = []
         toList (Node n t1 t2) = n : (toList t1 ++ toList t2)
         types = t :: T Integer
```

(iv) *(4 points)* We say that a tree has depth n if the longest path from the root of the tree to a leaf passes through n nodes. The depth of the tree in the above example is 3. Define a function

```
genT :: Int -> [a] -> Gen (T a)
```

such that `genT n es` is a QuickCheck generator for random trees of depth `n`, with elements in the list of elements `es`. You may assume that `n` is greater than or equal to 0. The tree pictured in the example above should be a possible sample of `genT 3 [0,1,2]`. As in the example, the trees you generate should have branches of various depths, but at least one branch should have the required depth.

Hint: It is probably *not* useful to use the quickCheck function `sized`. **Solution**

```
genT n els | n <= 0 = return Leaf
           | otherwise = do
              e <- elements els
              m <- choose (0,n-1)
              big   <- genT (n-1) els
              small <- genT m     els
              flip <- arbitrary
              return $ if flip then Node e big small
                                 else Node e small big
```

```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)
```

```haskell
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] ->
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.
```