Exam in Functional Programming TDA450/DIT140

Tuesday 24 October 2006, 8.30 – 12.30.
Examiner: Bengt Nordström, phone 1033 or 0730-79 42 89
Permitted aids: English-Swedish or English-other language dictionary.

You are free to use any Haskell standard functions, including the attached ones, unless the question specifically forbids you. You may use the solution of an earlier part of a question to solve a later part, even if you did not solve the earlier part. You may lose marks for complicated solutions. Your code should be written in such a way that it is quickly obvious to the reader what the code does and that it is correct. This means that you should strive for the right balance between conciseness and clarity. If any part of your code is not easy to understand, it should be commented in an appropriate way. Note that superfluous comments make the code more difficult to read.

This written exam is worth up to 180 points. It is possible to count up to 20 points for assignments handed in during this fall. We have the following limits for different grades: Chalmers students: 3 = 80 pts, 4 = 100 pts, 5 = 120 pts. University students: G = 100 pts, VG = 150 pts.

The exam will be shown on Monday 6 November at 15.30 in a location announced on the homepage of the course. Solutions to the exam will also be available from the homepage.

1. Consider the following data type for binary numbers:

   ```
   data Bin = Nil | O Bin | I Bin deriving (Show, Eq)
   ```

   The binary number 101 stands for the digital number 5, and 111 stands for 7. In our representation, the binary number will start with its least significant digit and the initial digit will always be removed (since all leading 0s are inignificant, all binary numbers can be considered to start with 1).

   For instance, the decimal number 5 is the binary number 101, which is represented by `I (O Nil)`. The binary number 100011 is represented by `I (I (O (O (O Nil))))`

So, we convert a binary number to our representation by taking away the first 1 and then reversing it and putting a Nil to the end.

The function which converts a binary number to an Integer is hence:

```
fromBin :: Bin -> Integer
fromBin Nil = 1
fromBin (O x) = 2 * (fromBin x)
fromBin (I x) = 2 * (fromBin x) + 1
```

(a) Define the function (10)

```
toBin :: Integer -> Bin
```

which converts an integer to a binary number, i.e. `fromBin (toBin n)` should compute to the value of n, for positive n.

(b) Define the function (20)

```
foldBin :: (t -> t) -> (t -> t) -> t -> Bin -> t
```

which should work for binary numbers in the same way as the function `foldr` works for lists. The first argument is a function which should be applied in the case the binary number starts with O and the second argument should be applied in the case the binary number starts with I. ou can look up the definition of `foldr` in the enclosed prelude, it is:

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

First, just to remind you about `foldr`: The intuition is that `foldr f a es` will - in the argument `es` - replace all `cons`-operations with `f` and the `nil` with `a`, such as:

```
  foldr f s a1:(a2:(a3:nil)) =
  foldr f s (cons a1 (cons a2 (cons a3 nil))) =
            ( f   a1 ( f   a2 ( f   a3  s )))
```

So our new function `foldBin` should work so that `foldBin f g a b` will in the argument `b` replace all occurrences of the constructor `O` by the function `f`, all occurrences of the constructor `I` by the function `g`, and the constructor `Nil` by the argument `a`.

```
foldBin f g a Nil   = a
foldBin f g a (O b) = f (foldBin f g a b)
foldBin f g a (I b) = g (foldBin f g a b)
```

An example how to use it:

```
fromBin2 :: Bin -> Integer
-- To convert a binary number to an integer replace all 0s with the
-- doubling operation and all 1s with the operation which doubles the
-- number and adds one, and replace the final Nil with 1, so for
-- instance Nil = 1, O Nil = 2*1, I (O Nil) = 2*(2*1) +1
fromBin2 x = foldBin (\x->2*x) (\x->2*x+1) 1 x
```

(c) Define the function (20)

```
reversBin :: Bin -> Bin
```

which reverses the digits of its input. So

```
*Main> reversBin (I (O (I (I Nil))))
I (I (O (I Nil)))
```

(d) If we use the definition of the data type Bin above, the Haskell interpreter will respond like this:

```
*Main> toBin 8
O (O (O Nil))
```

Instead, we would like to show binary numbers in the way we are used to:

```
*Main> toBin 8
IOOO
```

Show how the code must be changed so that all binary numbers are shown in the usual way. Explain your changes. irst, in the definition of the data type: (20)

```
data Bin = Nil | O Bin | I Bin deriving (Show, Eq)
```

we have to change it to

```
data Bin = Nil | O Bin | I Bin deriving (Eq)
```

so that we do not use the `show`-function which is automatically generated. Then we have to redefine the overloaded `show`- function:

```haskell
instance Show (Bin) where
  show = ppBin
ppBin :: Bin -> String
-- We insert the leading I and then replace all occurrences of the
-- constructor O with the character 0 and all occurrences of the
-- constructor I with the character 1 in the reverted binary number.
ppBin x = "I" ++ foldBin ('0':)
                         ('I':)
                         []
                         (revertBin x)
```

2. Consider now the definition of the data type of binary trees:

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

(a) Define the function (10)

```
depth ::  Tree a -> Integer
```

which computes the maximum depth of a tree. The depth of a leaf is defined to be 0.

(b) Define the infinite tree (5)

```
nats :: Tree Integer
```

which has 1 as the root, 2 as the root of the left subtree, 3 as the root of the right subtree, etc:

```
        1
      /   \
     2     3
   /  \   / \
  4    5 6   7
```

(c) Define the function (5)

```
root :: Tree a -> a
```

which computes the information in the root of its input, or the information in the leaf (if the tree is only a leaf).

(d) Explain why `root nats` does not lead to a nonterminating computation but to the result 1. In this explanation you must explain what a value in Haskell is and contrast this with the definition of values in eager languages. You must also explain how a function application is computed in Haskell and contrast this with other possible choices. (15)

If you did not define the constant `nats` above, you can instead use the constant `ones` defined by

```
ones = 1: ones
```

and explain the computation of the expression `head ones` . ee the lecture on coinductive data types.

(e) If we look at the tree `nats` above we see that we can use positive integers to index information in any finite tree. For instance, the number 2 indexes the root of the left tree of a tree, etc. If we now look at the binary representation of these numbers, we see that this represents the path from the root of the tree to the number (where O is the left subtree and I is the right subtree):

```
         I
        /\
      /O   \I
    /       \
   2         3
 /O  \I    /O  \I
4     5   6     7
```

For instance, the number 4 has the binary representation IOO, which is the path from the root to the number 4.

So we can look at a binary number as a path in a binary tree, a path reflecting the choices of subtrees.

The following function computes *one* of the longest paths of its arguments:

```
longest :: Tree a -> Bin
longest (Leaf v)       = Nil
longest (Tree v t1 t2) =
       if depth t1 > depth t2 then O l1 else I l2
       where l1 = longest t1
             l2 = longest t2
```

Define the function (20)

```
longests :: Tree a -> [Bin]
```

which computes the list of *all* longest paths of the argument.

(f) We can look at a vector as a list in Haskell. We compute the i:th element by traversing the list, and the length of the vector is the length of the list. This makes indexing of the vector a rather expensive operation: it will be proportional to the length of the list.

But instead we can implement a vector as a binary tree indexed by the corresponding binary number. The indexing operation can then be made more efficient, as there is no need to traverse the entire vector. Define now the function (20)

```
subBin     :: Tree a -> Bin -> Maybe a
```

which is such that `subBin t b` will look up the b-th element in the tree t. The exact meaning of this is seen when we see how the function is used to define the indexing operation for vectors:

```
sub     :: Tree a -> Integer -> Maybe a
sub t k = subBin t (toBin k)
```

So `sub t k` will look up the k-th element in t. For instance,

```
*Main> [(i, sub nats i) | i<-[1..10]]
[(1,Just 1),(2,Just 2),(3,Just 3),(4,Just 4),(5,Just 5),
(6,Just 6),(7,Just 7),(8,Just 8),(9,Just 9),(10,Just 10)]
```

e have to go down the tree along the path, the only trick is that we must not forget to reverse the binary number. Otherwise it is simple: we turn left or right depending on the digit in the path.

```
subBin t b = subr t (reversBin b)
subr (Leaf v)        Nil   = Just v
subr (Tree v t1 t2)  Nil   = Just v
subr (Tree v t1 t2) (O b) = subr t1 b
subr (Tree v t1 t2) (I b) = subr t2 b
subr _               _     = Nothing
```

(g) The length of a vector is the index of its last element. For binary trees, this is the same as the maximal binary number which indexes an element, so it is the maximal path in the tree. So we can define: (15)

```
lengthN :: Tree a -> Integer
lengthN t  = fromBin (lengthBin t)
lengthBin :: Tree a -> Bin
```

Your task is to define the function `lengthBin` above. otice that the maximal path in the tree (seen as a binary number) is always the rightmost longest paths. And it is exactly this path which is computed by the function `longest` earlier. Now, we must remember that this function computes the path *from* the root to the deepest element. But the binary index to the last element is the path from the element *to* the root. So we have to reverse the result:

```
lengthBin :: Tree a -> Bin
lengthBin t = reversBin (longest t)
```

3. Explain the difference between an overloaded function and a polymorphic function! Give examples (you can refer to earlier examples in this exam). ead about this in the lectures on overloading and type classes. There is also a description of this in the book. (20)

Good Luck!
Bengt
PS The next pages contains a list of function definitions.

```
-- Numerical functions: -------------------------------------
(^) :: (Num a, Integral b) => a -> b -> a
x ^ 0       =  1
x ^ n|n > 0  =  x * x^(n-1)
     |True   =  error "Prelude.^: negative exponent"
gcd :: Integral a => a -> a -> a
gcd 0 0        =  error "Prelude.gcd: gcd 0 0 is undefined"
gcd x y        =  gcd' (abs x) (abs y)
                    where gcd' x 0 = x
                          gcd' x y = gcd' y (x `mod` y)
sum, product   :: Num a => [a] -> a
sum            =  foldr (+) 0
product        =  foldr (*) 1
-- Char functions:-------------------------------------------
isAscii c              =  fromEnum c < 128
isControl c            =  c < ' ' ||  c == '\DEL'
isPrint c              =  c >= ' ' &&  c <= '~'
isSpace c              =  c == ' ' || c == '\t' || c == '\n' ||
                          c == '\r' || c == '\f' || c == '\v'
isUpper c              =  c >= 'A'   &&  c <= 'Z'
isLower c              =  c >= 'a'   &&  c <= 'z'
isAlpha c              =  isUpper c  ||  isLower c
isDigit c              =  c >= '0'   &&  c <= '9'
isAlphanum c           =  isAlpha c  ||  isDigit c
toUpper, toLower      :: Char -> Char
toUpper c | isLower c
        = toEnum (fromEnum c - fromEnum 'a' + fromEnum 'A')
          | otherwise  = c
toLower c | isUpper c
        = toEnum (fromEnum c - fromEnum 'A' + fromEnum 'a')
          | otherwise  = c
ord                   :: Char -> Int
ord                    = fromEnum
chr                   :: Int -> Char
chr                    = toEnum
-- List functions: ------------------------------------------------
null           :: [a] -> Bool
null []        =  True
null (_:_)     =  False
head           :: [a] -> a
head (x:_)     =  x
tail           :: [a] -> [a]
tail (_:xs)    =  xs

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

init             :: [a] -> [a]
init [x]          = []
init (x:xs)       = x : init xs
(++)           :: [a] -> [a] -> [a]
[]     ++ ys   =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)
concat         :: [[a]] -> [a]
concat         =  foldr (++) []
length         :: [a] -> Int
length   []    =  0
length (_:xs)  =  1 + length xs
reverse        :: [a] -> [a]
reverse []     =  []
reverse (x:xs) =  reverse xs ++ [x]
elem           :: Eq a => a -> [a] -> Bool
elem x []      =  False
elem x (y:ys)  =  x == y || elem x ys
take, drop     :: Int -> [a] -> [a]
take 0 _            = []
take _ []           = []
take n (x:xs) | n>0  = x : take (n-1) xs
take _ _            = error "PreludeList.take: negative argument"
drop 0 xs           = xs
drop _ []           = []
drop n (_:xs) | n>0  = drop (n-1) xs
drop _ _            = error "PreludeList.drop: negative argument"

replicate           :: Int -> a -> [a]
replicate 0 x       =  []
replicate n x | n>0 =  x : replicate (n-1) x

foldr             :: (a -> b -> b) -> b -> [a] -> b
foldr f e []      =  e
foldr f e (x:xs)  =  f x (foldr f e xs)
map            :: (a -> b) -> [a] -> [b]
map f []       =  []
map f (x:xs)   =  f x : map f xs
zip                :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs)  =  (a,b):zip as bs
zip _      _       =  []
zipWith                  :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith z (a:as) (b:bs)   =  z a b : zipWith z as bs
zipWith _ _      _        =  []
filter          :: (a -> Bool) -> [a] -> [a]
filter p []     =  []
filter p (x:xs) |  p x        =  x:filter p xs
                |  otherwise  =    filter p xs
takeWhile      :: (a -> Bool) -> [a] -> [a]
takeWhile p []       = []
takeWhile p (x:xs)
```

```
         | p x       = x : takeWhile p xs
         | otherwise = []
dropWhile      :: (a -> Bool) -> [a] -> [a]
dropWhile p []       = []
dropWhile p (x:xs)
         | p x       = dropWhile p xs
         | otherwise = x:xs
span, break            :: (a -> Bool) -> [a] -> ([a],[a])
span p []            = ([],[])
span p xs@(x:xs')
            | p x       =  (x:ys,zs)
            | otherwise =  ([],xs)
                           where (ys,zs) = span p xs'
break p                =  span (not . p)
-- lines breaks a string up into a list of strings at newline characters.
-- The resulting strings do not contain newlines.  Similary, words
-- breaks a string up into a list of words, which were delimited by
-- white space.  unlines and unwords are the inverse operations.
-- unlines joins lines with terminating newlines, and unwords joins
-- words with separating spaces.
lines            :: String -> [String]
lines ""         =  []
lines s          =  let (l, s') = break (== '\n') s
                      in  l : case s' of
                                []      -> []
                                (_:s'') -> lines s''
words            :: String -> [String]
words s          =  case dropWhile isSpace s of
                      "" -> []
                      s' -> w : words s''
                            where (w, s'') = break isSpace s'
unlines          :: [String] -> String
unlines          =  concatMap (++ "\n")
unwords          :: [String] -> String
unwords []       =  ""
unwords ws       =  foldr1 (\w s -> w ++ ' ':s) ws
until          :: (a -> Bool) -> (a -> a) -> a -> a
until p f x    =  if p x then x else until p f (f x)
unzip             :: [(a,b)] -> ([a],[b])
unzip []          =  ([],[])
unzip ((a,b):xs)  =  let (as,bs) = unzip xs
                     in (a:as,b:bs)
nub :: (Eq a) => [a] -> [a]
nub []     = []
nub (x:xs) = x : [ y | y <- xs, x /= y ]
sort :: (Ord a) => [a] -> [a]
sort []     = []
sort (x:xs) = sort smaller ++ [x] ++ sort bigger
 where
```

```
  (smaller, bigger) = partition (< x) xs
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
intersect               :: Eq a => [a] -> [a] -> [a]
intersect               =  intersectBy (==)
intersectBy             :: (a -> a -> Bool) -> [a] -> [a] -> [a]
intersectBy eq xs ys    =  [x | x <- xs, any (eq x) ys]
-- Standard combinators: ---------------------------------------------------
flip            :: (a -> b -> c) -> b -> a -> c
flip f x y      =  f y x
(.)             :: (b -> c) -> (a -> b) -> a -> c
(f . g) x       =  f (g x)
fst             :: (a,b) -> a
fst (x,_)        = x
snd             :: (a,b) -> b
snd (_,y)        = y
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)
id              :: a -> a
id    x          = x
const           :: a -> b -> a
const k _        = k
error  :: String -> a   -- primitive function, no definiton here
```