David Sands, D&IT

Functional Programming DIT 141 / TDA 451

2009-12-15 14.00 – 18.00 *M/maskin*

David Sands, 031 772 1059, 0737 207 663

- There are five Questions (with 11+4+10+4+11 = 40 points); a total of at least 17 points guarantees a pass.
- Results: latest 17 January.
- **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 marks are given to solutions which are short, elegant, efficient, and correct. Less marks are given to solutions which are unnecessarily complicated or unstructured
  - For each part Question, if your solution consists of more than 3 lines of Haskell code, include short comments to explain the intention.
  - You can use any standard Haskell function in your solution — a list of some useful functions is attached
  - 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.

This space is intentionally left non-blank

Question 1. In this question we suppose there exits some function `sales :: Int -> Int` which gives the weekly sales from a shop, where weeks are numbered in sequence 0, 1, 2,...

(a) *(8 points)* The function `zeroWeeks` is supposed to behave as follows: given a week number n (assumed to be non negative) it returns the number of weeks in the range 0,...,n for which the sales were zero.

Give *four* definitions of `zeroWeeks`:

i. using a list comprehension and the function `length`,

ii. using any of the standard Haskell functions, but *not* defining any new recursive functions, *not* using `foldr`/`foldl`, and *not* using list comprehensions,

iii. by defining a suitable tail-recursive helper-function, and

iv. using `foldr`, the list `[0..n]`, and no other recursive functions.

Solution

```
zeroWeeks1 n = length [() | i <- [0..n], sales i == 0]
zeroWeeks2 n = (length . filter (==0) . map sales) [0..n]

count n = if sales n == 0 then 1 else 0

zeroWeeks3 n = zW 0 n
  where zW k n | n < 0     = k
        zW k n | otherwise = zW (k + count n) (n-1)

zeroWeeks4 n = foldr zeroC 0 [0..n]
               where zeroC week total = count n + total
```

(b) *(3 points)* Explain what the function maxWeeks below computes (not *how* it computes).

```
maxWeeks n = mW n (sales n) []
   where mW n max weeks | n < 0          = weeks
         mW n max weeks | sales n > max  = mW (n-1) (sales n) [n]
                        | sales n == max = mW (n-1) max (n:weeks)
                        | sales n < max  = mW (n-1) max weeks
```

Define a function which computes the same result (for all n greater than or equal to zero), but which does not use explicit recursion. **Solution**

```
-- Computes the list of week numbers which have the largest sales in the period
maxWeeks' n = filter ((==m) . sales) weeks
              where weeks = [0..n]
                    m     = maximum (map sales weeks)
```

Question 2. (a) *(2 points)* Define a datatype (any helper types you might need) to represent a Ticket. A Ticket any one of

- A train ticket from a city to a city, either first- or second-class
- A bus ticket from a city to a city

- A flight ticket from a city to a city, travelling either business class, super economy, or economy.

Cities may be represented as strings. **Solution**

```
data Ticket = Train City City TClass | Bus City City | Flight City City FClass
type City = String
data TClass = First | Second
data FClass = Business | SuperEcon | Econ
```

(b) *(2 points)* For any ticket, the first City is called the start city and the second city is called the destination. We represent a journey by a list of tickets

```
type Journey = [Ticket]
```

Journey is valid if for any consecutive tickets in the list, the destination city for the first ticket is the same as the start city for the second ticket.

Define a function

```
valid :: [Ticket] -> Bool
```

which determines whether a journey is valid. You may assume the Journey is non empty. **Solution**

```
valid [_] = True
valid xs  = tail starts == init dests
          where (starts,dests) = unzip $ map cities xs
                cities (Train c d _)  = (c,d)
                cities (Bus c d)      = (c,d)
                cities (Flight c d _) = (c,d)
```

Question 3. A basic datatype for logical (boolean) expressions, `Logic` is given below:

```
data Logic = Const Bool | And Logic Logic | Not Logic | Var String
```

To compute the value of such an expression we need an environment which provides values for each variable:

```
type Env = [(String,Bool)]
```

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

```
eval :: Env -> Logic -> Maybe Bool
```

A `Maybe Bool` result is used here because a variable might not appear in the environment. However, your eval function should implement left-to-right shortcut (lazy) evaluation of `And`. For example, suppose that

```
p = And (Var "x") (Const False)
q = And (Const False) (Var "x")
```

Then `eval [] p` should give `Nothing` and `eval [] q` should give `Just False`. **Solution**

```
eval t (Var s) = lookup s t
eval t (Const b) = Just b
eval t (Not p) = case (eval t p) of
                   Just b  -> Just (not b)
                   Nothing -> Nothing
eval t (And p q) = case (eval t p) of
```

```
                    Just False -> Just False
                    Just True  -> eval t q
                    Nothing    -> Nothing
```

(b) *(2 points)* Sometimes programming with Maybe types gets messy, for example when we require nested case expressions of the form

```
case ... of
  Nothing -> Nothing
  Just x  -> case ... of
               Nothing -> Nothing
               Just y  -> ...
```

This situation can sometimes be improved by programming in monadic style, since `Maybe` is in fact an instance of `Monad`, as given by the following definition:

```
instance Monad Maybe where
    return        = Just
    fail          = Nothing
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
```

Rewrite your definition of `eval` above to make use of `do` notation.

Solution

```
eval' t (Var s) = lookup s t
eval' t (Const b) = Just b
eval' t (Not p) =
            do b <- eval' t p
               return (not b)
eval' t (And p q) =
            do b <- eval' t p
               if b then eval' t q else return False
```

(c) *(5 points)* A logical expression e is called a *tautology* if `eval t e` gives `Just True` for *any* environment t which contains a value for each variable in `e`. Define a function

`taut :: Logic -> Bool`

which computes whether a logic expression is a tautology. (Note that this question has nothing to do with QuickCheck!).

Hint: it will be useful to be able to generate a list of all possible environments for a given list of variable names.

Solution

```
vars :: Logic -> [String]
vars (Var s)   = [s]
vars (And p q) = vars p ++ vars q
vars (Not p)   = vars p
vars _         = []

allenvs :: [String] -> [Env]
allenvs []     = [[]]
allenvs (x:xs) = let es = allenvs xs in
```

```
                        map ((x,True):) es ++ map ((x,False):) es

taut l = and [b | Just b <- map (flip eval l) es]
  where es = allenvs (vars l)
```

Question 4. *(4 points)* Define a QuickCheck generator for permutations of a given list

```
perm :: [a] -> Gen [a]
```

For example

```
Main> sample (perm [1,1,2,3])
[1,1,2,3]
[1,2,1,3]
[2,1,1,3]
[3,1,2,1]
[3,1,1,2]
[1,2,1,3]
```

For full marks your definition should have exactly the type given. **Solution**

```
perm' :: Eq a => [a] -> Gen [a]
perm' [] = return []
perm' xs = do a <- elements xs
              as <- perm' (delete a xs)
              return (a:as)

perm xs = do as <- perm' [0..length xs - 1]
             return (map (xs!!) as )
```

Hint: you may find that the QuickCheck function `elements :: [a] -> Gen a` is useful here:

```
Main> sample (elements ["and", "a","happy","new","year"])
"happy"
"new"
"a"
"year"
"happy"
"and"
```

Question 5. A rose tree is a tree with data items at the nodes, and having zero or more branches. This can be represented in haskell using the following type:

```
data Rose a = R a [Rose a]
```

(a) *(1 points)* Give a Haskell definition `example :: Rose Int` which represents the tree

```
     1
    /|\
   2 0 3
  /|\  |
 4 5 6 7
```

Solution

```
example = R 1 [left, mid, right]
   where leaf n = R n []
         left   = R 2 [leaf 4, leaf 5, leaf 6]
         mid    = leaf 0
         right  = R 3 [leaf 7]
```

(b) *(2 points)* Define a function `roseMap` such that `roseMap f` is a function which applies the function `f` to each node in the rose tree to obtain a new rose tree. You should also give the most general type of the function. For example, `roseMap (+2) example` would compute a rose tree representing

```
     3
    /|\
   4 2 5
  /|\  |
 6 7 8 9
```

Solution

```
roseMap :: (a -> b) -> Rose a -> Rose b
roseMap f (R a xs) = R (f a) (map (roseMap f) xs)
```

(c) *(2 points)* Define a function `level :: Int -> Rose a -> [a]` such that `level n r` computes the elements, form left to right, at the n'th level of the tree. So for example, `level 3 example == [4,5,6,7]` and `level 4 example == []`. You may assume that `n` is non negative. **Solution**

```
level 0 r        = []
level 1 (R a _)  = [a]
level n (R _ rs) = concatMap (level (n-1)) rs
```

(d) *(2 points)* A simple computer file system can be viewed as a directory which has a name, and contains zero or more files, and zero or more directories (usually referred to as "sub-directories"). Each file has a name and some contents. With the help of the following definitions

```
type DirName  = String
type FileName = String
type Contents = String
data File     = File FileName Contents
```

define a type `Dir` for a Directory. Your definition should use the rose tree datatype, and not introduce any new recursive types. **Solution**

```
type Dir = Rose (DirName,[File])
```

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

```
find:: String -> Dir -> [String]
```

which searches for all the files within a directory which contain (in their Contents) the given string. The result is a list of all the *paths* to the files. For example, suppose that we have a directory "C" containing just two sub-directories called "Programs" and "Documents". In the "Programs" directory there is a file,

```
File "test.hs" "solution = x\n where x = x"
```

and in the "Documents" directory there is a file

```
File "lyric.txt" "He's a real nowhere man, sitting in his nowhere land"
```

If `c` represents the directory called "C", then `find "where" c` should produce

```
["/C/Programs/test.hs", "/C/Documents/lyric.txt"]
```

Solution

```
find s (R (dir,files) fs)
      = map extendPath $ ["/" ++ f | File f c <- files, s `occursIn` c]
                         ++ concatMap (find s) fs
        where extendPath path = "/" ++ dir ++ path
              occursIn s c = any (isPrefixOf s) (tails c)
              -- Data.List.tails was not included in the supplied function list
              -- tails [] = [[]]
              -- tails (x:xs) = (x:xs): tails xs
```