Chalmers | Göteborgs Universitet 2011-12-13

David Sands, D&IT

Functional Programming TDA 452/451, DIT 142/141

2011-12-13 14.00 – 18.00 VV ("Väg och Vatten")

David Sands, 0737 207663

- There are 4 Questions with maximum $11 + 9 + 16 + 6 = 42$ points; a total of 20 points definitely guarantees a pass.
- Results: latest within 21 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*, plus any functions of the QuickCheck library.
  - 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.

A computer once beat me at chess. But it was no match for me at kick boxing.

Question 1. (a) *(2 points)* Give the type of the following function:

```
q1 [] _           = []
q1 ((x:xs):xss) y = (x < y, True) : q1 xss y
```

Solution

```
q1 :: (Ord t) => [[t]] -> t -> [(Bool,Bool)]
```

(b) *(2 points)* Redefine `q1` without using recursion (but you may use any recursive functions defined in the Prelude). **Solution**

```
q1' xss y = map (\(x:_) -> (x < y, True)) xss
```

(c) *(3 points)* Simplify the following function definition as much as possible and give its type:

```
q1c x y
  | x /= False               = odd y : [ ]
  | x == True && even y      = [False]
  | otherwise                = [] ++ [False]
```

Solution

```
q1c' :: Integral a => Bool -> a -> [Bool]
q1c' x y = [x && odd y]
```

(d) *(4 points)* Define a function `maxDiff` which, given a list of Integers, returns the largest difference between any two consecutive integers in the list. For example,

```
maxDiff [3,1,-3,0,1,3,5]
```

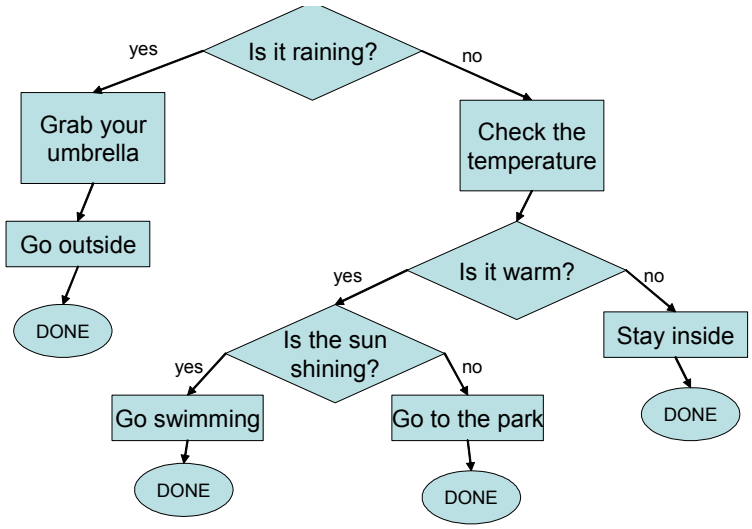
should return 4. `maxDiff [1]` and `maxDiff []` are both 0. (Note: the "difference" between 1 and -3 is the same as between -3 and 1).

Your definition must use a single tail-recursive helper function, and no other recursive functions.

Solution

```
maxDiff :: [Integer] -> Integer
maxDiff = md 0
  where
  md a (x:y:xs) = let d = abs (x - y) in md (max a d) (y:xs)
  md a _          = a
```

Question 2. In this Question, you will design a Haskell datatype to model action diagrams. An example of an action diagram is given here:



We can see that action diagrams contain three kinds of elements: Choice elements (diamond shapes) that contain a yes/no question, Action elements (rectangular shapes) that contain an action, and Terminal elements (round shapes) where the action diagram stops. One can take a path through an action diagram by starting at the top, answering all the questions with either yes or no, and following all arrows until arriving at a terminal DONE.

(a) *(3 points)* Define a recursive Haskell datatype `Diagram` that models the concept of action diagrams as described above. You may represent questions and actions simply as Strings. **Solution**

```
data Diagram = Done | Action String Diagram | Q String Diagram Diagram
```

(b) *(2 points)* Give the definition of a Haskell value

```
example :: Diagram
```

Solution

```
example = Q "Raining?" umbrella (Action "Temp" warm)
  where umbrella = Action "Umbrella" (Action "Outside" Done)
        warm     = Q "Warm?"  sunny (Action "StayIn" Done)
        sunny    = Q "Sunny?" (Action "Swim" Done) (Action "Park" Done)
```

that represents the above picture. You should use local definitions to improve readability. You may abbreviate the strings involved e.g. "Raining?".

(c) *(4 points)* Define the following function:

```
actions :: Diagram -> [Bool] -> [String]
```

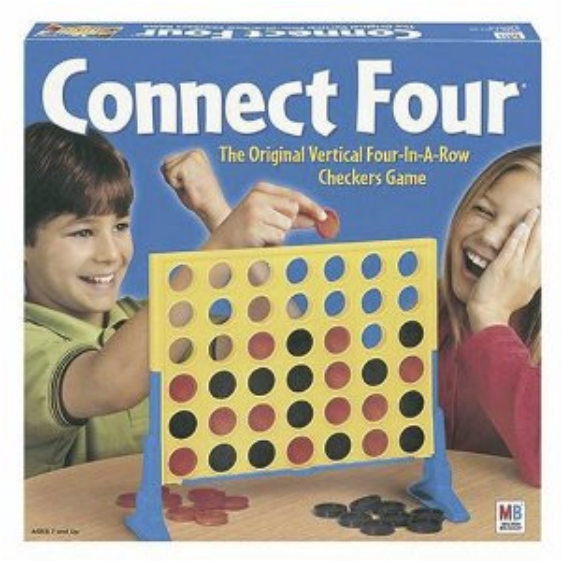
This function, given an action diagram, and a list of the answers to all questions on the way (False=no, True=yes), produces a list of actions that we need to take. You may assume that there are enough answers to the questions in the list, and that unused anwers are ignored. Example:

```
 Main> actions example [True]
["Grab your umbrella", "Go outside"]
 Main> actions example [False,True,True]
["Check the temperature", "Go swimming"]
```

Solution

```
actions Done _ = []
actions (Action s d) bs = s: actions d bs
actions (Q _ yes no) (b:bs)
            | b         = actions yes bs
            | otherwise = actions no bs
```

Question 3. The two-player game known as *Connect 4* involves a 7×6 grid consisting of 7 columns, each holding at most 6 tokens. One player has black tokens and the other has red. The grid is placed in a vertical plane as illustrated.



Players take it in turns to drop a token into any non-full column; the token falls to the lowest empty slot in that column.

The objective of the game is to become the first player to get four tokens in a straight line.

For the purposes of the following questions the exact definition of a "win" will not be important, since we will ignore this aspect of the game.

In the questions that follow we will use the following to model a C4 board:

```
data C4 = C4 [Column]
   deriving (Eq,Show)

type Column = [Player]

data Player = Red | Black
   deriving (Eq,Show)

example1 = C4 [[],[],[Red],[Black,Black,Red],[Black],[],[]]
```

Each column is represented by a list where the *last* element in each list is the most recent play. In `example1` above, if player Red decides to play in column 5 then the result would be

```
example2 = C4 [[],[],[Red],[Black,Black,Red],[Black,Red],[],[]]
```

[The questions begin on the next page]

(a) *(4 points)* The datatype invariant for a `C4` is that (i) no column has more than 6 tokens, (ii) there are at most 7 columns, (iii) the difference between the number of tokens that each player has played is at most one. We will call a `C4` with these properties a *legal* `C4`. Give a definition of a function

```
legalC4 :: C4 -> Bool
```

which checks whether the given `C4` is legal.

Solution

```
legalC4 (C4 cs) = length cs == 7 && all ((<= 6) . length) cs && balanced
    where balanced = number Red - number Black `elem` [-1,0,1]
          number p = length $ filter (==p) $ concat cs
```

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

```
play :: Player -> Int -> C4 -> Maybe C4
```

where `play p i c4` will try to create the C4 resulting from dropping a token of player `p` into column `i` of `c4`. So for instance

```
play Red 5 example1 == Just example2
```

If the resulting C4 is not legal (does not satisfy the invariant) for whatever reason then the result is `Nothing`, so for example

```
play Black 1 example1 == Nothing
```

Solution

```
play p ci (C4 cs)
    | ci < 1 || ci > 7 = Nothing
    | legalC4 cs'      = Just cs'
    | otherwise        = Nothing
     where
        cs' = let (f,i:b) = splitAt (ci-1) cs
                      in C4 $ f ++ (i ++ [p]):b
```

(c) *(4 points)* A run of a game can be represented by a list of column numbers, representing the list of alternating moves by the players (so the odd-indexed elements of the list are the moves from the first player, and the even-indexed elements are the moves from the second player).

Define a function

```
run :: C4 -> Player -> [Int] -> C4
```

where `run c p m` calculates the final C4 obtained after playing the sequence of moves `m` starting on the board `c` where player `p` makes the first move. If any move results in an illegal C4 then that move is discarded and the current player replays with the next move in the sequence. **Solution**

```
run c4 p [] = c4
run c4 p (m:ms) = case play p m c4 of
                      Just c4' -> run c4' (otherPlayer p) ms
                      Nothing  -> run c4 p ms

otherPlayer Red = Black
otherPlayer Black = Red
```

(d) *(4 points)* Define a QuickCheck generator for legal C4s, and make C4 an instance of class Arbitrary. Hints: use the function `run`. QuickCheck function

`vectorOf :: Int -> Gen a -> Gen [a]`

(among others) may be useful here. **Solution**

```
instance Arbitrary C4 where
  arbitrary = do
     n <- choose (1,6*7)
     moves <- vectorOf n (elements [1..7])
     player1 <- elements [Black,Red]
     return $ run emptyC4 player1 moves

emptyC4 = C4 (replicate 7 [])
```

Question 4. The datatype `Maybe` is used to model a computation which might fail. For example, the function `lookup :: Eq a => [(a,b)] -> a -> Maybe b` tries to lookup a key in a key-value table, and returns `Nothing` if the lookup fails.

One problem with Maybe is that it has no way to carry any useful information about the reason for the failure. Instead we could use the standard datatype `Either` instead of `Maybe`:

```
data Either a b = Left a | Right b    deriving (Eq, Show)
```

The idea is to use the type `Either String a` to model a computation of `a` which might fail, where the string is a suitable error message. For convenience we define:

```
type MayErr a = Either String a
```

As an example consider a safe division function:

```
safeDiv :: Integral a => a -> a -> MayErr a
safeDiv i j | j /= 0    = Right (i `div` j)
            | otherwise = Left ("Error: divide " ++ show i ++ " by zero")
```

(a) *(2 points)* Since many standard functions already use `Maybe`, it is useful to define a function which when given an error string, converts a `Maybe a` to an `MayErr a`. Define the following function to achieve this:

```
failsWith :: Maybe a -> String -> MayErr a
```

Solution

```
failsWith Nothing  s = Left s
failsWith (Just a) _ = Right a
```

(b) *(4 points) In the exam the crucial definition of the monad itself was missing. Unfortunately there were no questions about this so it didn't get spotted.*

We can make MayErr an instance of Monad as follows

```
instance Monad MayErr
```

Note that this is very similar to the instance for Maybe. (It is in fact the standard instance definition provided for any type `Either a` and not just for `Either String` as given here).

The following code uses three lookup tables to compute a triple consisting of the name, personal number, and crime associated with a given car licence number.

```
suspiciousCar :: LicenceNr -> Maybe (Name, Pid, Crime)
```

The details of the various types are not important, but you may assume they are in class Show.

Question: Rewrite this definition to have type

```
suspiciousCar' :: LicenceNr -> MayErr (Name, Pid, Crime)
```

making use of `failsWith` to provide better information, and do-notation to simplify the code. (Suggestion: use failsWith in infix-form)

Solution

```
-- Worse case: the three lookups are dependent on each other:
suspiciousCar car =
  case lookup car carRegister of
     Nothing -> Nothing
     Just pnr  -> case lookup pnr nameRegister of
             Nothing -> Nothing
             Just name  -> case lookup name crimeRegister of
                  Nothing -> Nothing
                  Just crime  -> Just (name,pnr,crime)

-- which could be rewritten as:
suspiciousCar' car = do
      pnr   <- lookup car carRegister
                `failsWith` ("No pnr found for " ++ show car)
      name  <- lookup pnr nameRegister
                `failsWith`  ("No name found for " ++ show pnr)
      crime <- lookup name crimeRegister
                `failsWith`  ("No crime found for " ++ show name)
      return (name,pnr,crime)
{- Other correct solutions assumed the lookups were independent and
tried to report all errors. -}
```