

David Sands , D&IT

Functional Programming DIT 141 / TDA 451

2010-12-14 14.00 – 18.00 VV (“Väg och Vatten”)

David Sands , [Answering questions on the day:David Sands 0737 207663]

- There are four Questions (with $11 + 11 + 18 + 12 = 52$ points); a total of 25 points definitely guarantees a pass.
- Results: latest 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 points are given to solutions which are short, elegant, and correct. Fewer points may be given to solutions which are unnecessarily complicated or unstructured, or unnecessarily inefficient.
 - 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.

How many programmers does it take to change a light bulb? None Its a hardware problem.

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

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

Solution

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

(b) (3 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]) xss
```

(c) (2 points) Simplify the following function definition as much as possible:

```
q1b :: Bool -> Int -> String
q1b x y
  | x == False      = "True"
  | x == True && even y = "False"
  | otherwise       = "True"
```

Solution

```
q1c x y = show $ not x || odd y
```

(d) (4 points) Define a function `minmax` (including its type) which given a non-empty list returns a pair of the smallest and the largest element in the list. Your definition should use a single tail-recursive helper function which computes the pair, and no other recursive functions.

Solution

```
minmax :: Ord a => [a] -> (a,a)
minmax (x:xs) = mm (x,x) xs where
  mm (s,b) [] = (s,b)
  mm (s,b) (y:ys) = mm (min s y, max b y) ys
-- alt:
minmax' (x:xs) = foldl (\(s,b) y -> (min s y, max s y)) (x,x)
```

Q 2. This question is about representing and writing a type checker for a tiny language of Haskell-like expressions.

The subset of Haskell expressions, `Hexp`, has expressions of just the following kinds: variables (identifiers) such as `x`, `y` and `z`, integer literals such as `42` and `-1`, boolean literals `True` and `False`, equality expressions of the form `e1 == e2`, and conditionals of the form `if e1 then e2 else e3`, where `e1`, `e2`, and `e3` stand for any `Hexp` expressions.

- (a) (3 points) Define a datatype to represent the above language of `Hexp` expressions. You should allow any expressions to be built, not just type correct ones. For simplicity you may assume that variables can be any string. **Solution**

```
data Hexp = Var String | HB Bool | HI Int | Hif Hexp Hexp Hexp
          | Heq Hexp Hexp
          deriving Show
```

- (b) (2 points) Give definitions for `example1`, and `example2` which should represent the following two `Hexp` expressions (one of which is badly typed!):

```
if x == False then 2 else 3
if 54 == x then 42 else True

example1 = Hif (Var "x" 'Heq' HB False) (HI 2) (HI 3)
example2 = Hif (HI 54 'Heq' Var "x") (HI 42) (HB True)
```

- (c) (6 points) To determine whether a given `Hexp` expression is type correct we need to know the type of the variables it contains. The following types can be used to represent these things:

```
data HType = HBool | HInt deriving (Eq,Show) -- the type of an Hexp
type TEnv = [(String,HType)]              -- a type environment
```

Define a function

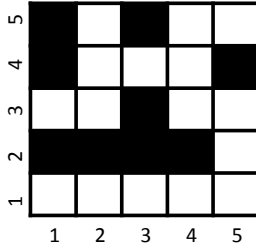
```
hType :: TEnv -> Hexp -> Maybe HType
```

For example, `hType [("x",HBool)] example1` should give `Just HInt` but both `hType [("x",HBool)] example2` and `hType [("x",HInt)] example2` should give `Nothing`.

You may decide for yourself what your function does in the case that the type environment does not have types for all variables in the expression. **Solution**

```
hType env = h where
  h (Var s) = lookup s env
  h (HB _)  = Just HBool
  h (HI _)  = Just HInt
  h (Heq e1 e2) = case (h e1, h e2) of
    (Just t1, Just t2) | t1 == t2 -> Just HBool
    -                               -> Nothing
  h (Hif b e1 e2) = case (h b, h e1, h e2) of
    (Just HBool, Just t1, Just t2) | t1 == t2 -> Just t1
    -                                           -> Nothing
```

Q 3. A maze consists of an $n \times n$ grid in which some squares are black. Here is an example of a 5×5 maze:



A maze is represented by its size and a list of the positions of its black squares:

```
type Position = (Int,Int)
type Maze = (Int, [Position])
```

For example the maze above could be represented by

```
maze :: Maze
maze = (5, [(1,2), (1,4), (1,5), (2,2), (3,2), (3,3), (3,5), (4,2), (5,4)])
```

A *path* through a maze is a sequence of positions of white squares:

```
type Path = [Position]
```

The first position represents the *end* of the path and the last position represents its *start*. In a path any two consecutive positions are either side-by-side or one above the other. No position can occur more than once in a path.

An example path from the south-east corner $(5,1)$ to the north-east corner $(5,5)$ of maze is

```
path = [(5,5), (4,5), (4,4), (4,3), (5,3), (5,2), (5,1)]
```

This question is about defining a function which can find a path from a given start position to a given end position.

(a) (6 points) Define a function

```
neighbour :: Maze -> Position -> [Position]
```

where `neighbour m p` provides a list of all the white squares which are either to the left or right, or above or below `p`. For example `neighbour maze (4,3)` could give `[(4,4), (5,3)]`.

Solution

```
neighbour m (x,y)
  = filter (isWhite m) [(x-1,y), (x+1,y), (x,y-1), (x,y+1)]
```

```
isWhite (n,ps) q = q 'inside' n && q 'notElem' ps
```

```
inside (a,b) n = all (\x -> x>0 && x<=n) [a,b]
```

(b) (3 points) Define a function

```
extend :: Maze -> Path -> [Path]
```

which gives all the possible ways to extend the given set of non-empty paths in the given maze with one square at the beginning. For example `extend maze [(4,4),(4,3)]` could give `[[(3,4), (4,4), (4,3)], [(4,5), (4,4), (4,3)]]`

Solution

```
extend maze (e:ps) = [p:e:ps | p <- neighbour maze e, not $p `elem` ps]
```

(c) (6 points) Define a function

```
allpaths :: Maze -> Position -> [Path]
```

which computes the list of all paths in the given maze starting at the given position.

Hints: You might consider first defining a recursive function which computes all paths of length k from a given start position. An alternative approach is to make use of the function `iterate`. **Solution**

```
extendAll = concatMap . extend
```

```
allpaths m p = concat [paths k p | k <- [1..(fst m)^2]]
```

```
  where paths 1 p = [[p]]
```

```
        paths n p = extendAll m (paths (n-1) p)
```

```
allpaths' m p = concat $ takeWhile (not . null) $
  iterate (extendAll m) [[p]]
```

(d) (3 points) Define a function

```
fromto :: Maze -> Position -> Position -> [Path]
```

where `fromto m p q` computes all paths from a start position `p` to an end position `q` in a maze `m`. **Solution**

```
fromto m start end = filter ( (==end) . head ) $ allpaths m start
```

Q 4. (continuation from Question 3)

- (a) (6 points) In order to make a maze an instance of class `Arbitrary` we need to make a new data type thus:

```
data TestMaze = M Maze deriving (Eq,Show)
instance Arbitrary TestMaze where
  arbitrary = ...
```

Provide a definition for `arbitrary`, ensuring that mazes are well formed: all squares are within the given dimensions of the maze, and no square appears more than once in the list of positions of black squares. **Solution**

```
instance Arbitrary TestMaze where
  arbitrary = do a <- arbitrary
                let size = abs a + 1 -- non-empty size
                    ls <- listOf (square size)
                    return (M (size, nub ls))
                where square s = do -- generate a square in an s x s maze
                                      x <- choose (1,s)
                                      y <- choose (1,s)
                                      return (x,y)
```

Hint: functions from `Test.QuickCheck` such as `choose :: (Int,Int) -> Gen Int` (for generating a number in a given range) and `listOf :: Gen a -> Gen [a]` (for converting a generator of things into a generator of lists of things) could be useful.

- (b) (6 points) Define a `quickCheck` property which tests a useful relationship between all the paths in `fromto m start end` and those in `fromto m end start`. Since such a test requires taking `start` and `end` to be white squares within `m`, in your property you should take `start` to be the lowest numbered white square in `m` and `end` to be the highest numbered (the exact ordering you use for squares is not important).

Solution

```
prop_fromto (M (size,b)) = not (null white) ==>
  sort forwards == sort (map reverse backwards)
  where forwards = fromto (size,b) sp ep
        backwards = fromto (size,b) ep sp
        white = [(x,y) | x <- [1..size], y <- [1..size]] \\ b
        sp = head white
        ep = last white
```