Chalmers | Göteborgs Universitet 2012-12-18

David Sands, D&IT

Functional Programming TDA 452/451, DIT 142/141

2012-12-18 14.00 – 18.00 "Väg och vatten"-salar

David Sands, 0737 207663

- There are 4 Questions with maximum of 13 + 13 + 11 + 8 = 45 points; a total of 22 points definitely guarantees a pass.
- Results: latest within 21 days.
- The examiner will visit the examination rooms at approximately 15.00–15.15 and again at around 16.15–16.30; at other times he will be available by phone to answer queries about the questions.
- **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.

Two bytes meet. The first byte asks, "Are you ill?" The second byte replies, "No, just feeling a bit off."

Question 1. Consider the following function:

```
chat 0 f (x:xs) = f x : xs
chat _ _ []     = []
chat n f (x:xs) = x:chat (n-1) f xs
```

You may assume that the first argument to chat will be a non-negative Int.

(a) *(2 points)* Give the type of `chat`. **Solution**

```
chat :: Int ->  (a -> a) -> [a] -> [a]
```

(b) *(3 points)* Give a definition for a function `chat'` which is equivalent to `chat` (under the assumption about the first argument), but which is defined using only the standard functions (as listed at the back). **Solution**

```
chat' n f xs =
   [if m == n then f x else x | (m,x) <- zip [0..] xs]
-- or
chat'' n f xs =
   case drop n xs of
           [] -> xs
           (z:zs) -> take n xs ++ [f z] ++ zs
```

(c) *(2 points)* Define a quickCheck property that could be used to test the equivalence of `chat` and `chat'`. In your test you may use a specific function for the second parameter of `chat`.

Solution

```
prop_chat n xs = let n' = abs n in
    chat n' not xs == chat' n' not xs
```

(d) *(3 points)* A function `findIn` tries to find the earliest index at which its first argument can be found as a sublist of the second argument. It satisfies the following property:

```
prop_findIn0 = findIn "Hell" "Hello"      == Just 0
            && findIn "ell" "Hello Jello" == Just 1
            && findIn "Hell" "Helan"      == Nothing
```

With the help of the function `isPrefixOf`, give a definition of `findIn`, including its most general type, using a tail-recursive helper function. **Solution**

```
findIn :: Eq a => [a] -> [a] -> Maybe Int
findIn xs = f 0
  where f n ys | xs `isPrefixOf` ys = Just n
        f _ []                      = Nothing
        f n (_:ys)                  = f (n+1) ys
```

(e) *(0 points)* Check that you remembered to include the type of the function in your answer to the previous question.

(f) *(3 points)* Define a quickCheck property which checks that whenever a list `ys` definitely contains `xs` as a sublist, then `findIn xs ys` will not give `Nothing`. Note: it is not necessary to create a new generator for lists to answer this question.

Solution

```
prop_findIn xs ys zs = isJust $ findIn ys (xs ++ ys ++ zs)
   where types = xs :: [Bool]
```

Question 2. In this Question, you will design a Haskell datatype to model a journey. A journey is a non-empty list of *legs*. For example a journey from Halmstad to London might consist of three legs: a train from Halmstad to Gothenburg, a bus from Gothenburg to Landvetter Airport, and a flight from Landvetter to London. Suppose that we begin to model this by defining

```
type Journey = [Leg]
```

As in the example above leg consists of a *mode of transport*, which is either bus, train, or flight, the place of origin, and the destination. Here we will model places as strings:

```
type Place = String
```

(a) *(2 points)* Complete the definition of the data type for a Journey.
Solution

```
data Leg = Leg Mode Place Place
  deriving Show

data Mode = Bus | Train | Flight
  deriving Show
```

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

```
connected :: Journey -> Bool
```

which computes whether the places in the journey are all connected (so that the destination of one leg will always be the origin for the next leg). *Your solution should not define any new recursive function, but should make use of standard functions.* Hint: you might find it useful to use the list

```
zip (init journey) (tail journey)
```

in your solution. **Solution**

```
connected j =
 and [ p == q | (Leg _ _ p , Leg _ q _) <- zip (init j) (tail j)]
```

(c) *(4 points)* Define, *using recursion and none of the standard functions except for those in the* `Eq` *class*, a function

```
missingLegs :: Journey -> [(Place,Place)]
```

which computes the pairs of places that are not connected in the given Journey. This should satisfy:

```
prop_missingLegs j = not(null j) ==> connected j == null (missingLegs j)
```

Solution

```
missingLegs (Leg _ _ q : Leg x r s : ls)
     | q /= r    = (q,r):rest
     | otherwise = rest
   where rest = missingLegs (Leg x r s : ls)
missingLegs _    = []
```

(d) *(4 points)* Add appropriate instance declarations so that quickCheck can be run on `prop_missingLegs`. **Solution**

```
instance Arbitrary Leg where
  arbitrary = do
            let places = elements ["A","B","C"]
            p <- places
            q <- places
            mode <- arbitrary
            return $ Leg mode p q

instance Arbitrary Mode where
  arbitrary = elements [Bus,Train,Flight]
```

Question 3. The map of a simple text-based adventure game is modelled as

```
data Map = Map PlaceName [(Dir,Map)]
data Dir = N | S | E | W  deriving (Eq,Show)
type PlaceName = String
```

An example of a map consisting of three places is given below; the "Hogwarts" castle has a lake to the north and a forest to the south:

```
hogwarts = Map "Castle" [(N,forest),(S,lake)]
forest   = Map "Forest" [(S,hogwarts)]
lake     = Map "Lake"   [(N,hogwarts)]
```

In the questions that follow you may assume that a direction appears at most once in a list of direction-map pairs, and that every distinct place in a map has a unique place name.

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

```
travel ::  Map -> [Dir] -> Maybe Map
```

which returns the map (if there is one) obtained after following the given sequence of directions. So for example `travel hogwarts [N,S,S]` would give a result equivalent to `Just lake`, but `travel hogwarts [N,E]` or `travel hogwarts [N,N]` would both give `Nothing`. Hint: the function lookup can be useful here.

Solution

```
travel m [] = Just m
travel (Map _ dirs) (d:ds) = do m <- lookup d dirs -- case lookup ...
                                travel m ds
```

(b) *(1 points)* If we add `deriving Show` to the definition of `Map`, what happens when we try to print `hogwarts`?

(c) *(6 points)* Make Map an instance of class Show in a way that allows maps to be displayed in the following way:

Main> lake

```
You are at the Lake. Go N to Castle
Castle. Go N to Forest, Go S to Lake
Forest. Go S to Castle
```

Main> forest

```
You are at the Forest. Go S to Castle
Castle. Go N to Forest, Go S to Lake
Lake. Go N to Castle
```

Hints: the function `intersperse` could come in handy. As a wise man once said, to avoid going round in circles, it can be useful to remember where you've been.

Solution

```
instance Show Map where
  show m = "You are at the " ++ showMap [] m

inter as = concat . intersperse as
here (Map p _) = p
```

```
showMap seen (Map p ds) =
  showHere ++ showDirections ++ "\n" ++ showRestofMap
          where showHere = p ++ ". "
                showDirections =
                    inter ", " $ map showDir ds
                showDir (dir,m) =
                    "Go " ++ show dir ++ " to " ++ here m
                showRestofMap =
                    inter "\n" $ map (showMap (p:seen)) notseen
                notseen = [Map q d |(_,Map q d) <- ds, q`notElem`(p:seen)]
```

Question 4. (a) *(3 points)* Rewrite the following definition without using do notation:

```
backup f = do
      a <- readFile f
      let backup = f ++ ".bac"
      putStrLn $ "Creating backup in " ++ backup
      writeFile backup a
```

Solution

```
backup' f = readFile f >>= \a ->
               let backup = f ++ ".bac" in
               putStrLn "..." >> writeFile backup a
```

(b) *(2 points)* For-loops found in typical imperative programs are not part of Haskell, but there is nothing to stop us from defining our own imperative-style control structures. In this question you should define a function `for_` of type

```
for_ :: [a] -> (a -> IO()) -> IO()
```

which can represent simple for loops. For example a (psudocode) for loop

```
 for i = i to 10 {
      print i
}
```

could be written in Haskell as

```
for_ [1..10]  $ \i ->
     print i
```

Solution

```
for_ range f = sequence_ (map f range)
```

(c) *(1 points)* The above function assumes that the loop body does not produce any result. Give a definition for a more general function

```
for :: [a] -> (a -> IO b) -> IO [b]
```

which collects the results of each iteration.

Solution

```
for range f = sequence (map f range)
```

(d) *(2 points)* Sometimes a large file (such as a video) needs to be split into a collection of smaller files. Suppose that these smaller files are named *f.part1*, *f.part2*, ... This question is about joining them back together again to get the original file *f*.

Use the function `for` to define the function `join :: FilePath -> Int -> IO()` such that `join f i`, when run, concatenates the contents of the `i` parts of file `f` together and writes them back into file `f`. You may assume that `f` and `i` are correctly specified. `FilePath` is equivalent to `String`.

Solution

```
join f i = do
  parts <- for [1..i] $ \p ->
                    readFile (f ++ ".part" ++ show p)
  writeFile f (concat parts)
```