Chalmers | Göteborgs Universitet 2020-01-18

Examiner: David Sands, dave@chalmers.se, D&IT,
Answering questions at 10 – 10.30 (or by phone 031 772 1059)

Functional Programming TDA 452, DIT 143

2020-01-18 8:30 – 12:30 Lindholmen-salar

031 772 1059

- There are 5 questions with maximum $12 + 14 + 6 + 8 = 40$ points. Grading:
  Chalmers: 3 = 20–26 points, 4 = 27–33 points , 5 = 34–40 points
  GU: G = 20–33 points, VG = 34–40 points
- Results: latest approximately 15 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!
  - 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.*
  - **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.
  - 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.

1. *(12 points)*

   (a) *(8 points)* For each of the following definitions, give the most general type, or write "No type" if the definition is not correct in Haskell.

```
fa x         = unlines ("Hello, " : x)
fb (x:y)     = (x,x,y)
fc x y (z:_) = x<=y && not z
fd a         = do s <- a
                  putStrLn $ "Answer: " ++ s
```

   (b) *(4 points)* Define a function `merge` that merges two sorted lists into a sorted list.

```
merge     :: Ord a => [a] -> [a] -> [a]
```

2. *(14 points)* The following types are used to represent a card game *Poopamon*. Each playing card is either a *Trainer* or a *Monster*. A trainer card has between 1 and 5 "hit points". A monster card has a particular *Species* and a *Kind*, which is either *Nature*, *Fire*, or *Water*. A monster card has 1, 2, or 3 hit points. Note that not all combinations of *Species* and *Kind* are valid.

   A hand of cards consists of one or more cards. These are modelled in Haskell using the following data types:

```
type HP      = Int -- the number of "hit points"
data Card    = Trainer HP | Monster Species Kind HP  deriving Show
data Species = Poopachu | Fartle | Blub              deriving (Eq,Show)
data Kind    = Nature | Water | Fire                 deriving (Eq,Show)

data Hand = Last Card | Several Card Hand            deriving Show
```

   The combinations of Species and Kinds that are valid are given by the following function:

```
validKinds :: Species -> [Kind]
validKinds Poopachu  = [Fire]
validKinds Blub      = [Water]
validKinds Fartle    = [Nature,Fire,Water]
```

   Cards should only have valid combinations according to this function. So for example the following is a valid hand:

```
exHand = Several (Trainer 5)
         (Several (Monster Fartle Nature 2)
         (Last    (Monster Poopachu Fire 3)))
```

   Note for example that a *Poopachu* can only have kind *Fire*, and no monster can have more than three hit points.

   (a) *(2 points)*
   Define the data type invariant

```
prop_Card :: Card -> Bool
```

   which checks that a card is valid according to the constraints on hit points and kinds described above.

   (b) *(2 points)*
   Define a *recursive* function

```
prop_Hand :: Hand -> Bool
```

   which checks that a hand only contains valid cards.

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

```
foldHand ::(Card -> a) -> (a -> a -> a) -> Hand -> a
```

such that `foldHand f op hand` combines all the cards in a hand by applying function `f` to each card, and combining all the results with the function `op`. For example, with this function the previous question could be deined by

```
prop_Hand' h = foldHand prop_Card (&&) h
```

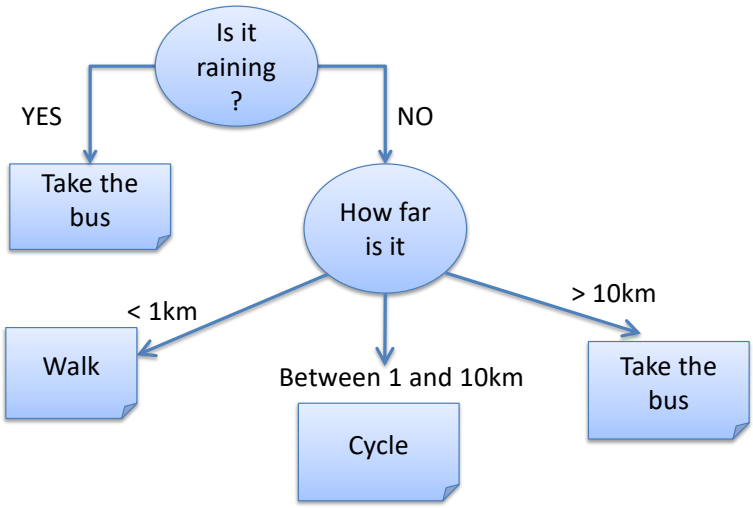
(d) *(2 points)* Use `foldHand` to define a function

```
hitPoints :: Hand -> Int
```

which calculates the total number of hit points in a given hand. You may use non-recursive helper-functions if you need.

(e) *(5 points)* Write the code necessary to be able to sucessfully run (and pass) the test `quickCheck prop_Card`, and other useful tests on cards. You do not need to write any code for type `Hand`.

3. *(6 points)* The picture below is what is sometimes called a *decision tree*.



The following data type can be used to represent such a decision tree:

```
data DTree = Q Question [(Answer,DTree)] deriving Show
type Question = String
type Answer   = String
```

Each node of the tree has a question, and a list of pairs of answers and decision trees. A final decision (a square box in the picture) is represented as a question with no alternatives; for convenience we define a function for this:

```
decision :: Question -> DTree
decision q = Q q []
```

(a) *(3 points)*
Give a definition of a decision tree

```
travel :: DTree
```

which represents the tree pictured above. You can use local definitions for parts of your answer to make your solution more readable!

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

```
allAnswers :: DTree -> [Answer]
```

which computes a list of all the answers handled by a given decision tree.
For example `allAnswers travel` should satisfy:

```
prop_allAnswers =
  allAnswers travel == ["Yes", "No", "<1km", "Between 1 and 10km", ">10km"]
```

4. *(8 points)* One way to represent a Sudoku puzzle is as a list-of-lists:

```
type Sudoku = [Row]
type Row = [Maybe Int]
```

Here is an example of a mini 3x3 sudoku

```
exampleSud :: Sudoku
exampleSud = [[blank,  blank,  Just 4]
             ,[blank,  Just 5, blank ]
             ,[Just 6, blank,  blank ]]   where blank = Nothing
```

The following definitions represent a Sudoku in a different way, by giving the position and the number contained in the non-blank squares

```
type Sudoku' = [FilledCell]
type Pos = (Int,Int)

data FilledCell = Filled Pos Int  deriving (Eq,Show)
```

We will assume that positions are always positive, and the position (0,0) refers to the top left square of the Sudoku, and the y-coordinates grow downwards.

Note also that the order of the elements in the list is unimportant, but there are no repeated coordinates. The example Sudoku above could be represented as:

```
exampleSud' :: Sudoku'
exampleSud' = [Filled (2,0) 4, Filled (1,1) 5, Filled (0,2) 6]
```

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

```
fromSudoku :: Sudoku -> Sudoku'
```

that converts from the first representation to the second. You may assume that every row and column in your Sudoku has the same number of elements, but your definition should work for any size of Sudoku (not just 9 x 9).
Your definition should convert `exampleSud` into something equivalent to `exampleSud'` (i.e. the elements should be the same, but the order might be different).

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

```
update :: Sudoku' -> Pos -> Maybe Int -> Sudoku'
```

which updates the given `Sudoku'` at the given position. For example the following should be true for your solution (although you might need to reorder the lists):

```
prop_update =
    update exampleSud' (2,0) Nothing  == [Filled (1,1) 5, Filled (0,2) 6]
 && update exampleSud' (2,0) (Just 1) ==
                           [Filled (2,0) 1, Filled (1,1) 5, Filled (0,2) 6]
 && update exampleSud' (0,0) Nothing  == exampleSud'
```

You may assume that the `Sudoku'` is valid (e.g. does not have any repeated positions).