Till Mossakowski
Benjamin Junge

9. Excercise sheet

Issued: 2023-12-13
Due: 2023-12-18 & 2023-12-19

9.1 *ID and a Monad for counting binds*

The lecture has introduced the general concept of the monad typeclass. In order to get better acquainted with it lets write some instances of Monad for relatively simple/trivial types:

- Consider the Type data Identity a = Identity a. While not very interesting or useful it still a good exer-cise to write a Monad instance for this type. You will also need to write instances of Functor and Appli-cative.
- Consider now the Type data CountBinds a = CountBinds (Integer , a) . This type counts the number of »=and » used allowing to count how many things are sequenced in a do block.
  Write a selector binds :: CountBinds a → Integer that extracts the bind count and write the Monad (and Functor/Applicative) instance that allows for counting binds.
  Note on <> instance: apply will combine two CountBinds elements: one with a function, one with a value. Propagate the count of the vale element. A combination of both counts is not necessary.*
  e.g:

```
two :: CountBinds Int
two = do
x ← return 1
y ← return 2
return (x + y)
```

  binds two ⇝ 2

9.2 *making Bingos*

The traditional Bingo game is very simple: every participant gets a sheet with a 5x5 table of numbers, usually between 0 and 99 (the middle square is a free space). It is played by drawing numbers from a source and everyone marking it if it is on their table. If a full row/column/diagonal is marked the player won. The game only makes sense/is fun if every Bingo card is different so lets write a simple generator for them: Write the function bingoMaker :: IO ( ) which is a interactive generator tool. To keep it simple the bingoMaker will only ask for the amount of cards that should be generated and wait for the user input. After a valid input number n the bingoMaker will generate n unique1 lists of 24 Integers between 0 and 99. Then it will format each list into the bingo format:

```
26 | 22 | 64 | 18 | 48 |
-----------------------
| 1 | 26 | 28 | 82 | 21 |
-----------------------
| 63 | 31 | free | 8 | 98 |
-----------------------
| 96 | 44 | 98 | 19 | 99 |
-----------------------
| 19 | 66 | 19 | 38 | 53 |
```

After that it will write the formatted bingos into an output file "Bingos.txt"where Bingo cards should be separated by 3 empty lines. Test your function by generating a Bingos.txt file with 10 Bingo cards.

9.3 *making Bingos*

Buzzword Bingo is a derivative of the traditional Bingo. The main difference is that instead of numbers each field has a buzzword (or trope) associated with e.g. a movie genre or something similar. The game is then played by watching a movie of that genre and marking a field whenever its buzzword is used or trope is seen. We now want to create a generator for any Buzzword Bingo based on the generator idea of 9.2: The bingoMaker :: IO ( ) function will this time ask for a source txt file for Buzzwords (1 word per line) and a number of cards that should be generated. Then it will generate the cards in a similar fashion to 9.2 using the buzzwords from the file instead of integers from 0 to 99 and write them (with 3 empty lines in-between two cards) into a outputfile BBingos.txt.
Padding out cells like in 9.2 is not necessary for these cards, a row might look like: |optionOne|what|free|t|empty|
Test your generator by generating 10 cards with a custom selfmade source file.
Hint: using a Data.Map might allow reuse of the 9.2 bingoMaker

9.4 **The state of Ants*

Langton's ant is a two-dimensional universal Turing machine that shows that emergent behaviour can arise from a set of very simple rules. A brief description of the ant: consider an infinite grid in the plane where each grid cell can be either coloured black or white. At the start the entire grid is white and the ant is placed onto a starting position facing up/north. Each step of the simulation consists of the ant changing the colour of the current cell it is in, moving one step into the current orientation direction and changing the orientation according to the following rule: if the destination cell is white turn clockwise, if its black turn counter clockwise. While we will not take a look at the theory behind it we will try and create/simulate such an ant by using the state monad.
The state type State s a = State runState :: s → (a, s) consists of two type arguments, s being the type for the states and a being the type for some output.
So lets start by outlining what the state type of the Ant machine should consist of: The grid itself [A Map of (Int,Int) Bool should suffice] and the Ant which has a current position in the grid and a orientation (Up, Down, Left or Right).
Based on those types implement

```
step :: State Field String
walk :: Field → (String , Field )
```

Where walk describes the transition from state to state based on the relevant information of the current state (ants current position, the Bool value of the destination cell and the current orientation). Since the state monad (import Control.Monad.State.Lazy) hides the constructor for State you will need to use the
state :: (s → (a, s)) → State s a function for step. The string in walk should encode which direction the ant turned and its orientation before and after the step.
To test your ant use:

```
g = fromList [ ( ( i , j ) , False ) | i ← [ 10 .. 10], j ← [ 10 .. 10]]
f st ( runState ( replicateM 10 step ) ( Field g (Ant (0,0) Up) ) )
```

which should return something like:["cw U-R","cw R-D","cw D-L","cw L-U","ccw U-L","ccw L-D","cw D-L","cw L-U","cw U-R","cw R-D"]. Your result may of course vary depending on your encoding choice. In this example every step consists of the rotation (cw = clockwise, ccw=counter-clockwise) and the ants orientation before and after the step separated by '-'.