Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Datenbanksysteme · Prof. Dr. Grust

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming

WS 2019/20

Benjamin Dietrich, Denis Hirn

Assignment #8

Submission Deadline: Thu, 19.12.2019

Wichtig: Im Zeitraum vom 6.12.2019 bis zum 19.12.2019 (16 Uhr) findet die Lehrevaluation zu dieser Veranstaltung statt. Ihr wurdet dazu per E-Mail an eure studentische Adresse eingeladen. Bitte nehmt daran teil, das Feedback ist uns sehr wichtig!

Exercise 1: Functor Instances (4 Points)

1. Make the following data type for rose trees an instance of `Functor`:

   ```
   data RoseTree a = RoseTree a [RoseTree a]
   ```

2. Consider a data type for simple key-value maps:

   ```
   data Map a b = Map [(a, b)]
   ```

   The following instance cannot be defined:

   ```
   > instance Functor Map where
   error:
       * Expecting one more argument to 'Map'
         Expected kind '* -> *', but 'Map' has kind '* -> * -> *'
       * In the first argument of 'Functor', namely 'Map'
         In the instance declaration for 'Functor Map'
   ```

   Explain the problem. Define a slightly different but reasonable `Functor` instance for `Map`, instead.

Exercise 2: Weak Head Normal Form (4 Points)

Enter the following lines of code into your GHCi REPL and observe the values bound to `search`, `text` and `finding`, as printed with `:sprint` in lines 9, 10, 12, 13, 14, 16, 17 and 18.

- Test your intuition! Before you enter a `:sprint` command, guess what the result will look like.
- For each `:sprint`, explain briefly, why the expressions have (only) been reduced to the printed form. Based on the values printed in lines 17 and 18, try to infer exactly how function `intersect` proceeds to compute its result.

Note:

- Don't copy code from the PDF document. You can also find it in a separate file `sprint.hs`.
- To write *mulit-line commands* in GHCi, enter the command `:{`. Then, paste or write multiple lines of code and close the input with `:}`.

```
1  import Data.List
2  import Data.Char

3  let text = cycle [ "stop", "reading", "!", "Because", "this", "is", "an"
4                   , "endless", "text", "that", "-", "once", "in", "a"
5                   , "loop", "-", "will", "never", "allow", "you", "to"
6                   ]
7  let search = map (map toLower) ["A", "text", "loop", "!"]

8  length search
9  :sprint text
10 :sprint search

11 let finding =  search `intersect` text
12 :sprint finding
13 :sprint text
14 :sprint search

15 length finding
16 :sprint finding
17 :sprint text
18 :sprint search
```

Exercise 3: Simple Sudoku Solver **(12 Points)**

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 1: Grid of a difficult Sudoku puzzle unsolved (left) and solved (right).

The file `sudoku.hs` contains a couple of type and function definitions you have to use in this exercise to write a simple Sudoku[1] solver. The basic type definitions of interest are:

```
type Matrix a = [Row a]
type Row a = [a]

type Grid = Matrix Digit
type Digit = Char
```

A Sudoku grid is a 9×9 matrix of characters `['1' .. '9']` or a character `'0'` which encodes an *empty* cell. To *solve* a Sudoku puzzle, all empty cells have to be filled with the digits 1 to 9 such that each column, row and 3×3 box contains all of the numbers 1 to 9, each only once. Your task is to write a function `solve :: Grid -> [Grid]` which computes all the ways a given Sudoku grid may be completed without collisions.

First implement a naive algorithm based on the following approach:

```
solve :: Grid -> [Grid]
solve = filter valid . expand . choices
```

1. A function `choices :: Grid -> Matrix [Digit]` fills each cell of the given grid with all available choices. For non-empty cells there is no choice: its digit is given by the grid. For empty cells, all digits are a possible choice.
2. A function `expand :: Matrix [Digit] -> [Grid]` then expands this matrix to a list of all possible grids by combining the available choices in all possible ways.
3. Finally, each grid is tested by a function `valid :: Grid -> Bool` to be a valid Sudoku solution, without any forbidden duplicates of digits 1 to 9 in any row, column or box.

[1] https://en.wikipedia.org/wiki/Sudoku

This naive approach is obviously not efficient. So we are going to improve it utilizing a simple idea: Before expansion we can remove a lot of impossible choices that conflict with a digit that was already given by the grid.

```
solve :: Grid -> [Grid]
solve = filter valid . expand . prune . choices
```

4. Function `prune :: Matrix [Digit] -> Matrix [Digit]` removes all digits from each row, column or box which are already contained as fixed choices in this row, column or box.

Pruning may cause further fixed choices, so we can even try to apply it more than once:

```
solve :: Grid -> [Grid]
solve = filter valid . expand . many prune . choices
```

5. Implement function `many :: Eq a => (a -> a) -> a -> a` applies a function `f :: a -> a` to a value *a* (of type `a`) multiple times, just as long as `f` *a* changes the value of *a*.