EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming
Wintersemester 2021/2022
Prof. Dr. T. Grust, D. Hirn

Assignment 8 (17.12.2021)

Handin until: Friday, 06.01.2022, 00:00

Die Vorlesungsevaluation im WS 2021/2022 läuft – Bitte gebt uns eure Rückmeldung. Danke!

Bitte nehmt euch ein paar Minuten Zeit und meldet euch zurück. Jedes Bit an Rückmeldung zählt – das gilt insbesondere für die Freitextkommentare. Die gewonnenen Einsichten sind Gold wert für uns, um einzuschätzen, ob wir die *Functional Programming* in die richtige Richtung steuern und wie wir die Veranstaltung in den verbleibenden Wochen des Semesters noch besser machen können.
Die Feedbackformulare der Lehrevaluation können **bis zum 21. Dezember 2021, 18:00 Uhr** ausgefüllt werden. Nochmals: Von uns schon jetzt ganz herzlichen Dank dafür!

Exercise 1: Lazy Evaluation **[4 Points]**

Function `sumAcc` (defined below) performs list summation using an accumulating argument `acc`. We have `sumAcc 0 [3..9] ≡ 42`, for example. `sumAcc` works fine but exhibits problems once the list `xs` gets large:

```
> sumAcc 0 [1..50000000]
*** Exception: stack overflow  [Haskell's evaluation memory is exhausted]
```

1. Show the reduction steps performed when Haskell reduces `sumAcc 0 [1,2,3]` (use → to show the individual reduction steps just like with example `min = head . isort` in the lecture). You reduction trace will look as follows:

```
sumAcc 0 [1,2,3]  →  ...                    [sumAcc.2]
                  →  ...                        ⋮
                  ⋮
                  →  6
```

2. Study your reduction trace and briefly explain your hypothesis of why `sumAcc 0 [1..50000000]` leads to a memory overflow.

   A function `f` that might not use its argument `x` to produce a result is **non-strict** in `x` (see function `f` below). When `f` is applied to argument `e` (via `f e` or, equivalently, `f $ e`), `e` will not be evaluated. This changes with Haskell's *strict application operator* `$!`: in `f $! e`, argument `e` will *always* be evaluated before `f` is applied, no matter what (if `e` is already in WHNF, then `f $! e ≡ f e`):

```
> f e
42
it :: Integer
> f $! e
e has been evaluated
42
it :: Integer
```

3. Define `sumAcc'` that follows the *same accumulating argument principle* like `sumAcc` but uses strict application `$!` where necessary to avoid the memory exhaustion problem:

```
> sumAcc' 0 [1..50000000]                  [runs for about 10 secs]
1250000025000000
it :: Integer
```

4. Show the reduction steps performed when Haskell reduces your `sumAcc' 0 [1,2,3]`.

```haskell
import Debug.Trace (trace)

sumAcc :: Integer -> [Integer] -> Integer
sumAcc acc []     = acc                          -- [sumAcc.1]
sumAcc acc (x:xs) = sumAcc (acc+x) xs            -- [sumAcc.2]

e :: Integer
e = trace "e has been evaluated" 1

f :: Integer -> Integer
f x = 42
```

Exercise 2: Weak Head Normal Form [3 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 `:}`.

```haskell
import Data.List
import Data.Char

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

length search
:sprint text
:sprint search

finding =  search `intersect` text
:sprint finding
:sprint text
:sprint search

length finding
:sprint finding
:sprint text
:sprint search
```

Exercise 3: Simple Sudoku Solver [13 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.

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*.

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