EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming
Summer term 2023
Prof. Torsten Grust, Denis Hirn
WSI — Database Systems Research Group

Assignment 8

Hand in this assignment until **Friday, 30. June 2023, 10:00** at the latest.

Ⓔxam-style Exercises

Exercises marked with Ⓔ are similar in style to those you will find in the exam. You can use these to hone your expectations and gauge your skills.

Running out of ideas?

Are you hitting a roadblock? Are some of the exercises unclear? Do you just need that one hint to get the ball rolling? Refer to the `#forum` channel on our Discord server and check the tag for this assignment—maybe you'll find just the help you need.

Task 1: Lazy Evaluation

The `sumAcc` function (defined below) performs list summation with an accumulating argument `acc`. We have `sumAcc 0 [3..9] ≡ 42`, for example. `sumAcc` works fine, but has problems when 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, as in the `min = head . isort` example in the lecture). Your reduction trace will look like this:

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

2. Study your reduction trace and briefly explain your hypothesis as to 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` is not evaluated. This changes with Haskell's *strict application operator* `$!`: in `f $! e`, the argument `e` is *always* 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'`, which 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]`.

```
1  import Debug.Trace (trace)
2
3  sumAcc :: Integer -> [Integer] -> Integer
4  sumAcc acc []     = acc                          -- [sumAcc.1]
5  sumAcc acc (x:xs) = sumAcc (acc+x) xs            -- [sumAcc.2]
6
7  e :: Integer
8  e = trace "e has been evaluated" 1
9
10 f :: Integer -> Integer
11 f x = 42
```

Task 2: Weak Head Normal Form Ⓔ

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 11, 12, 15, 16, 17, 20, 21 and 22.

- Test your intuition! Before you enter a **:sprint** command, guess what the result will look like.
- For each **:sprint**, briefly explain why the expressions were (only) reduced to the printed form. From the values printed in lines 21 and 22, try to deduce exactly how the **intersect** function calculates its result.

> Note
>
> - Don't copy code from the PDF document. You can also find it in a separate file **sprint.hs**.
> - To write *multi-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
4  text = cycle [ "stop", "reading", "!", "Because", "this", "is", "an"
5               , "endless", "text", "that", "-", "once", "in", "a"
6               , "loop", "-", "will", "never", "allow", "you", "to"
7               ]
8  search = map (map toLower) ["A", "text", "loop", "!"]
9
10 length search
11 :sprint text
12 :sprint search
13
14 finding =  search `intersect` text
15 :sprint finding
16 :sprint text
17 :sprint search
18
19 length finding
20 :sprint finding
21 :sprint text
22 :sprint search
```

Task 3: Simple Sudoku Solver

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 some type and function definitions you will 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 must be filled with the numbers 1 to 9, so that every column, every row and every 3×3 box contains all of the numbers from 1 to 9, each only once. Your task is to write a function `solve :: Grid -> [Grid]` that calculates all ways to complete a given Sudoku grid 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 every cell of the given grid with all available choices. For non-empty cells, there is no choice: their 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 into a list of all possible grids by combining the available choices in all possible ways.
3. Finally, each grid is tested with the function `valid :: Grid -> Bool` to be a valid Sudoku solution, without any forbidden duplicates of the digits 1 to 9 in any row, column, or box.

This naive approach is obviously inefficient. So we are going to improve it using 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 that are already contained as fixed choices in that 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