Technical University of Munich
Chair for Logic and Verification
Prof. Tobias Nipkow, Ph.D.
J. Rädle, L. Stevens, K. Kappelmann, MC Sr Eberl

WS 2020/21
20.11.2020
Deadline: 30.11.2020, 23:59

Functional Programming and Verification

Sheet 3

Sustainability Exercises

The **Public Climate School** is taking place this week (23.–27.11). How about you watch some of the lectures as part of this exercise sheet? ;) You can find the programme here `https://studentsforfuture.info/public-climate-school/`

Tutorial Exercises

Exercise T3.1 Good Style, Bad Style

Your tutor will give you a brief introduction about how to and how not to do case distinctions and recursions on lists. They will implement a recursive function on lists in two ways: using pattern matching and using library functions (`null`, `head`, `tail`), etc.

For the future: never use the bad style again.

Exercise T3.2 Matrices

a) Write a function `dimensions :: [[a]] -> (Int,Int)` that determines the dimensions of its input matrix encoded as a list. For example, the matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

will be encoded as `[[1,2,3,4],[5,6,7,8],[9,10,11,12]]`. Calling `dimensions` on this matrix should return `(3,4)`. If the input is not a valid matrix, e.g. if one row contains fewer elements than the other rows, the function should return `(-1,-1)`.

b) Define a predicate `isSquare :: [[a]] -> Bool` that returns true iff its input is a square matrix. Also define predicates

```
canAdd :: [[a]] -> [[a]] -> Bool
canMult :: [[a]] -> [[a]] -> Bool
```

that determine whether their input matrices have the right dimensions to be added or multiplied together.

c) Write a function `diagonal :: [[a]] -> [a]` that returns the diagonal of a square matrix encoded as a list. For example, the diagonal line of the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

above matrix is `[1,5,9]`.

(Optional) Can you extend this to a function that takes the diagonal line of a cube?

d) Define functions

```
matrixAdd :: [[Integer]] -> [[Integer]] -> [[Integer]]
matrixMult :: [[Integer]] -> [[Integer]] -> [[Integer]]
```

which add/multiply two matrices.

Hint: for multiplication, you may want to use the `transpose` function from the `List` library.

Exercise T3.3 Merge Sort

In the lecture you have seen a Haskell implementation of Quicksort. In this assignment you will have to implement *Merge Sort* in Haskell.

Recall: Merge Sort is based on the divide-and-conquer principle. First, it splits a list in two halves and sorts these lists separately. In the conquer step, it merges the two sorted lists. Note that this can be done recursively by comparing the two heads of the lists.

- Implement a Haskell function `mergeSort :: [Integer] -> [Integer]` that sorts an integer list in ascending order by using Merge Sort. To split the list, you can use the function `splitAt`.
- Implement a function `adjacentPairs :: [a] -> [(a,a)]` that generates all adjacent pairs of elements from a given list.
- Test your sorting function by checking whether all adjacent pairs of its result are indeed in the correct order.

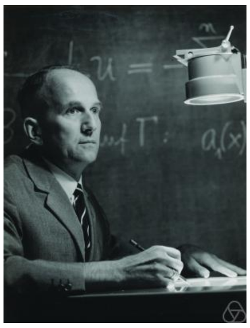
Exercise T3.4 (Optional) Collatz Conjecture

We define the following function $f : \mathbb{N} \to \mathbb{N}$

$$f(n) = \begin{cases} 1, & \text{if } n = 1 \\ \frac{n}{2}, & \text{if } n \text{ is even} \\ 3n+1, & \text{otherwise} \end{cases}$$

and, for every $n \in \mathbb{N}_+$, the sequence

$$c_0 = n, \quad c_{i+1} = f(c_i).$$


Lothar Collatz

The longstanding Collatz conjecture states that for every $n \in \mathbb{N}_+$, the sequence $(c_i)_{i\in\mathbb{N}}$ stabilises to 1.

a) Write a function `collatz :: Integer -> [Integer]` such that `collatz n` computes the sequence $(c_i)_{i\in\mathbb{N}}$ until it stabilises at 1. For example,

```
collatz 12 = [12,6,3,10,5,16,8,4,2,1].
```

b) Write a quickCheck test that checks whether the collatz conjecture holds for $1 \leq n \leq 100$.

Homework

You need to collect 7 out of 9 points (P) to collect a coin.

Exercise H3.1 Sudo ku –solve

In this exercise, you will implement a backtracking based Sudoku solver. This means that you start inserting numbers until you either found a solution or the Sudoku is in an illegal state. In the latter case, you go back (backtrack) until the Sudoku is legal again and try the next number, eventually arriving at the right combination.

A Sudoku is described as a list of list of Ints (`[[Int]]`). Sudokus always have a square shape, but their size can vary. For example, there are 4×4, 9×9, and 16×16 Sudokus. Every $n \times n$ Sudoku will be a list containing n lists of length n, where each entry/cell is a number in `[0..n]`. The number 0 represents an empty cell.

A given $n \times n$ Sudoku is further divided into $\sqrt{n}$ subgrids/squares. The usual rules of Sudoku then apply:

1. Each row must contain the numbers `[1..n]`.
2. Each column must contain the numbers `[1..n]`.
3. Each square must contain the numbers `[1..n]`.

Hint: You can use the function `intRoot` from the template to obtain the square root of an `Int`.

a) We start by defining different Methods to access our Sudoku:

   i) Implement a function `selectRow :: [[Int]] -> Int -> [Int]`, such that `selectRow sudoku n` returns the nth row of the Sudoku (index starts at 0).

   ii) Implement a function `selectColumn :: [[Int]] -> Int -> [Int]`, such that `selectColumn sudoku n` returns the nth column of the Sudoku (index starts at 0).

   iii) Implement a Function `selectSquare :: [[Int]] -> Int -> [Int]`, such that `selectSquare sudoku n` returns the nth square of the sudoku. Squares are numbered from left to right, top to bottom and their content is read in the same way.

   Example: Given the following `sudoku`:

```
8 9 6|7 5 2|4 1 3
5 2 3|6 1 4|9 8 7
4 7 1|8 9 3|2 6 5
-----+-----+-----
9 5 4|3 6 7|8 2 1
3 1 8|2 4 5|7 9 6
```

```
7 6 2|1 8 9|5 3 4
-----+-----+-----
6 8 9|5 7 1|3 4 2
2 4 7|9 3 6|1 5 8
1 3 5|4 2 8|6 7 9
```

we have

i) `selectRow sudoku 0 = [8,9,6,7,5,2,4,1,3]`,

ii) `selectColumn sudoku 0 = [8,5,4,9,3,7,6,2,1]`,

iii) `selectSquare sudoku 3 = [9,5,4,3,1,8,7,6,2]`, and

iv) `selectSquare sudoku 2 = [4,1,3,9,8,7,2,6,5]`.

Hint: You can use execute `putStr (showSudoku s)` from the template to pretty-print the Sudoku `s`.

b) Next we want to check if a given (not necessarily completety filled) Sudoku is in a valid state, that is in every row, column, and square, every number appears only once. Reminder: 0 is not an actual entry but represents an empty cell.

Do so by implementing the function `isValidSudoku :: [[Int]] -> Bool`.

Hint: You might want to use a function `isValidSubsection [Int] -> Bool` that checks if a given row, column, or square follows the rules.

c) As a next step, we want to modify a Sudoku. To do so, you have to implement a function `setCell :: [[Int]] -> (Int, Int) -> Int -> [[Int]]`, such that `setCell sudoku (x, y) n` returns a Sudoku with the cell at position `(x,y)` set to `n`.

d) Finally – time to start solving Sudokus! Your final task is to write a function `solveSudoku :: [[Int]] -> [[Int]]` that returns a solution for a given Sudoku. If the given Sudoku is not solvable, it returns the empty list instead. You can test your implementation with the sudoku provided in the template; it should not take more than a few seconds. Your implementation should use a backtracking-based approach – or something even better! The MCs have no time to waste – no homework points are awarded for slow, brutforce implementations.

Hint: If you are motivated, consider using the more idiomatic approach of Zippers to mutate a list instead of relying on `setCell`.

Wettbewerb: As the fundamental theorem of modern society tells us: time is money. The MCs hence ask you to maximise money by not minimising tokens for a change, but by minimising the time burnt solving Sudokus. Only the fastest solvers must come to fame.

If you want to participate in the *Wettbewerb* as a guest, please send an e-mail to fpv@in.tum.de to make us aware of it.

This exercise was designed and implemented in coorporation with our tutors. Special thanks to all of them!

Haskell is concise
Functional, well-typed, and neat
It is like Haiku

— Haskell's Haiku webpage