185.A05 Advanced Functional Programming SS 21

Monday, 12 April 2021

Assignment 3

on Chapter 7 and Chapter 4

Topics: Functional Arrays and Functional Pearls

Submission deadline: Monday, 19 April 2021, 12am

Regarding the deadline for the second submission: Please, refer to „Hinweise zu Organisation und Ablauf der Übung“ available at the homepage of the course.

Important:

1. Carefully read and follow the instructions outlined in the complementary files provided with assignment 0. If you have any questions regarding them, ask your questions in the TISS forum. Following these instructions is essential to ensure a smooth processing of your submitted file with the test system.

2. Store all functions to be written for this assignment in a top-level file named

   `Assignment3.hs`

   of your group directory. Take advantage of the template file `Assignment3.hs` and extend it as required. Comment your program meaningfully, and use auxiliary functions and constants, where reasonable. The very same file name shall be used for the second submission of assignment 3.

3. *Do not use self-defined modules!* If you want to re-use functions (written for other assignments), copy them to `Assignment3.hs`. Import declarations for self-defined modules will fail: Only `Assignment3.hs` will be copied for the (semi-automatical) evaluation procedure, no other ones.

4. Your programs will be (semi-automatically) evaluated on `g0` using the there installed GHC interpreter of GHC version 8.65. If you use a different tool or a different version of GHC for program development, please, double-check well in time before the submission deadline that your programs behave on `g0` and the there installed GHC interpreter version as you expect.

Programming assignments:

In this assignment we explore and compare the relative merits of lists and functional arrays considering typical matrix operations and Binoxxo puzzles as examples. Binoxxo puzzles give us additionally the opportunity of exerimenting with functional pearl programming.

1. We consider integer matrices and addition and multiplication of integer matrices using the following notations:

   - *Matrix:* An (integer) matrix $\mathcal{M}$, $m, n \in \mathsf{IN}_1$, is a non-empty two-dimensional scheme of integers:

   $$\mathcal{M} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

   M is called *rectangular* of type (m, n), if $m \neq n$; it is called *quadratic* of type (m, m), if $m = n$.

   - *Matrix addition:* If M, N are two matrices of type (m, n), their sum S is a matrix of the same type defined by:

   $$M + N =_{df} \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix} = S$$

   with

   $$S =_{df} \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{pmatrix}$$

   and

   $$c_{ij} =_{df} a_{ij} + b_{ij},\ i = 1, \ldots, m,\ j = 1, \ldots, n$$

   If M, N are not of the same type, adding them and their sum is undefined.

   - *Matrix multiplication:* If M, N are two matrices of type (m, n) and (n, p), respectively, their product P is a matrix of type (m, p) defined by:

   $$M \cdot N =_{df} \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \cdots & \cdots & \cdots & \cdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix} = P$$

   with

   $$P =_{df} \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \cdots & \cdots & \cdots & \cdots \\ c_{m1} & c_{n2} & \cdots & c_{mp} \end{pmatrix}$$

   and

   $$c_{q,r} =_{df} \Sigma_{j=1}^{n} a_{qj} b_{jr},\ q = 1, \ldots, m,\ r = 1, \ldots, p$$

   If M, N are not of 'fitting' types, multiplying them and their product is undefined.

(a) Mofelling matrices as lists of lists:

```
type Nat1 = Int

type Row    = [Int]
type Column = [Int]

type MatrixLoR = [Row]      -- Matrices as lists of rows
type MatrixLoC = [Column]   -- Matrices as lists of columns
```

Implement the following functions (only for matrices as lists of rows):

```
is_wellformed :: MatrixLoR -> Bool
add  :: MatrixLoR -> MatrixLoR -> Maybe MatrixLoR
mult :: MatrixLoR -> MatrixLoR -> Maybe MatrixLoR
```

where

i. `is_wellformed` returns `True`, if its argument is a matrix of type (m, n) for some $m, n \in \mathbb{N}_1$; otherwise, it returns `False`.

ii. `add` returns the sum of its arguments (as `Just` value), if they are well-formed matrices of the same type; otherwise, it returns `Nothing`.

iii. `mult` returns the product of its arguments (as `Just` value), if they are well-formed matrices of fitting types; otherwise, it returns `Nothing`.

(b) Modelling matrices as two-dimensional functional arrays:

```
import Data.Array

type Matrix = Array (Nat1,Nat1) Int
```

Implement the following functions:

```
plus  :: Matrix -> Matrix -> Maybe Matrix
times :: Matrix -> Matrix -> Maybe Matrix
```

where

i. `plus` returns the sum of its arguments (as `Just` value), if they are matrices of the same type; otherwise, it returns `Nothing`.

ii. `times` returns the product of its arguments (as `Just` value), if they are of fitting types; otherwise, it returns `Nothing`.

(c) **Without submission:**

i. Do you think it would make a (substantial) difference for implementing `is_wellformed`, `add`, and `mult`, if matrices were modelled as lists of columns instead of lists of rows?

ii. Why can we omit implementing a function `is_wellformed` for matrices modelled as two-dimensional functional arrays?

iii. Do you consider the implementation of `add` and `mult` or of `plus` and `times` conceptually and technically easier? Or do you think it does not make a substantial difference?

iv. If you consider the list of list or the array implementation of matrices as superior over the other one, can you imagine'real world' examples, where your judgement would be the other round? In particular, of examples, which do not rest on the 'unlimited growth' ability of lists?

v. Test the correctness of the various matrix operations by matrices of your own choice. Are their performance differences between the operations working on list and array-modelled arrays?

Explain your reasoning and experimental observations.

2. A Binoxxo puzzle is given by a rectangular or quadratic grid with even side-lengths that is partially filled with crosses and circles:

Bodoxxo 1

			X		X		O		
		O			X				X
									X
		O						O	
	X							O	
				X		X			
			X			X		X	
	O	O			O				
				X	X				
	O							O	

Bodoxxo 2

			X		O		O		O
				O					
X	O						O		
		X	X				O	X	
					X				
X						O			
	O				X				
		O					O		
X					X				X
	O		O			X			

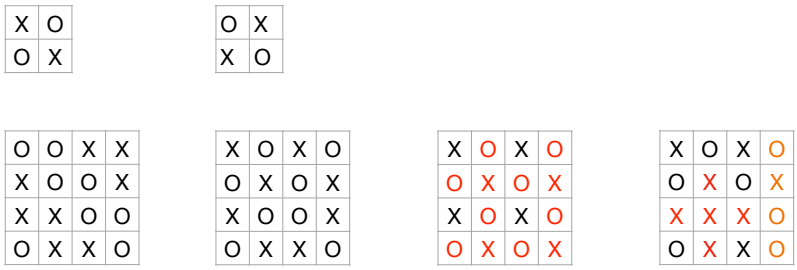
Solving a Binoxxo puzzle means filling all its free entries with either a cross or a circle such that the following three *wellformedness conditions* hold:

(wf1) In every row and in every column the number of crosses and circles is the same.

(wf2) All rows and all colums are pairwise disjoint.

(wf3) At most two crosses or circles are horizontally or vertically immediately adjacent.

The following figure shows at the top all correctly solved Binoxxo puzzles of size 2×2, and at the bottom two correctly solved puzzles of size 4×4 on the left, and two incorrectly filled ones on the right.



In this assignment we consider rectangular and quadratic Binoxxo puzzles of even side-lengths in the range of 2 to 10. As for the matrix operations, we explore on-the-fly the relative merits of modelling Binoxxo puzzles as lists of lists and two-dimensional functional arrays, respectively.

(L) Modelling Binoxxo puzzles as lists of rows:

```
data Entry = B     -- Blank (or empty)
           | X     -- Cross
           | O     -- Circle
           deriving (Eq,Show)

type BRow = [Entry]

type BinoxxoL = [BRow]

is_wellformed_L :: BinoxxoL -> Bool
is_complete_L   :: BinoxxoL -> Bool

solve_naively_L :: BinoxxoL -> Maybe BinoxxoL
solve_smartly_L :: BinoxxoL -> Maybe BinoxxoL
```

(A) Modelling Binoxxo puzzles as two-dimensional functional arrays:

```
type Number = One | Two | Three | Four | Five
            | Six | Seven | Eight | Nine | Ten
            derivng (Eq,Ord,Enum,Show)

instance Ix Number where
 ...

type BinoxxoA = Array (Number,Number) Entry

is_wellformed_A :: BinoxxoA -> Bool
```

```
is_complete_A   :: BinoxxoA -> Bool

solve_naively_A :: BinoxxoA -> Maybe BinoxxoA
solve_smartly_A :: BinoxxoA -> Maybe BinoxxoA
```

The various functions shall have the following meaning:

(a) `is_wellformed_L` and `is_wellformed_A` return `True`, if their arguments model a rectangular or quadratic Bionoxxo puzzle with at most 10 rows and 10 columns enjoying additionally the Binoxxo well-formedness conditions $(wf1)$, $(wf2)$, $(wf3)$; otherwise, they return `False`.

(b) `is_complete_L` and `is_complete_A` return `True`, if their arguments do not contain any empty entries; otherwise, they return `False`.

(c) `solve_naively_L` and `solve_naively_A` shall solve well-formed Binoxxo puzzles straightforwardly in an obviously correct manner at the expense of being possibly (very) inefficient and poorly performing. The naive solvers play the rôle of the initial algorithms solving a functional pearl problem (cf. Chapter 4 on functional pearls). If there is more than one solution of a puzzle, it does not matter which of them the implementations of `solve_naively_L` and `solve_naively_A` yield. If there is no solution, the naive solvers return `Nothing`.

(d) `solve_smartly_L` and `solve_smartly_A` shall solve wellformed Binoxxo puzzles as fast as possible! If there is more than one solution of a puzzle, it does not matter which of them the implementations of `solve_smartly_L` and `solve_smartly_A` yield. If there is no solution, the smart solvers return `Nothing`.

Try making your implementation a functional pearl! Try developing the smart solvers by systematically and stepwise transforming their naive counterparts. Make sure (at least by convincing yourself somehow), that every transformation step preserves the semantics of the solver of the current step, ensuring thus the correctness of the implementation of the final smart solvers.

3. **Without submission:**

(a) Note: Wellformed and complete Binoxxo puzzles are correctly solved.

(b) For matrices we could omit implementing a well-formed test of array-modelled matrices. Does this apply to Binoxxo puzzles, too?

(c) In case a well-formed test for array-modelled Binoxxo puzzles can not be dropped, does the list-modelled Binoxxo puzzles provide an advantage over the array-based ones or vice versa for implementing the well-formed test?

(d) Do you consider the list-modelled or the array-modelled of Binoxxo puzzles advantageous for implementing Binoxxo puzzle solvers? Or do you think it does not make a substantial difference?

(e) Test the correctness and performance of the various solvers by Binoxxo puzzles of your own choice.

(f) If you developed your smart solvers in several steps, extend the tests to the intermediate solvers. Which of the heuristics you added for performance improvement had the biggest impact? Were there some that had little or no impact? What do you think are the reasons for the success of failure of the heuristics?

(g) Did algorithms patterns turn out to be useful for your implementations?

Explain your reasoning and experimental observations.

Iucundi acti labores.
Getane Arbeiten sind angenehm.

Cicero (106 - 43 v.Chr.)
röm. Staatsmann und Schriftsteller