Sjaak Smetsers

Radboud University
Institute for Computing and Information Sciences

Exam
Functional Programming 1 (NWI-IBC029)

07.01.2019, 12:30 – 14:30

Note: Please complete in block letters.

Surname: ....

First name: ....

Student number: ....

Email: ....@student.ru.nl

Please read carefully before answering the questions:

- Write your name and student number on *each* sheet.
- Check that the exercise set is complete: the exam consists of 8 questions.
- Read the questions carefully.
- Write your answers *on* the question sheets. Feel free to use the back pages, as well. Two blank pages are included at the rear. Additional paper is available on request.
- Use a pen with a permanent ink.
- Write legibly. Be concise and precise.
- This exam is "gesloten boek". You are not allowed to use lecture notes, personal notes, books, a notebook, or any other electronic devices.

Question	**1**	**2**	**3**	**4**	**5**	**6**	$\sum$
max. points:	10	22	14	10	8	12	**76**
obtained:							

Result: ....

Question		10 points
1	Name:	(5×2)
	Student number:	Σ: .... / S: ....

Polymorphism

Define a *total* function for each of the following types, that is, provide a binding for each of the type signatures. (You may assume that functions passed as arguments are total.)

a) $g1 :: (a, b, c, d) \to (b, c, d, a)$

b) $g2 :: a \to (b \to (a, (b, b), a))$

c) $g3 :: ((a, b, c) \to d) \to c \to b \to a \to d$

d) $g4 :: (Int \to a) \to a$

e) $g5 :: (((Int \to a) \to a) \to a) \to a$

Question		22 points
2	Name: .... Student number:	$(4 \times 2 + 2 \times 4 + 6)$ Σ: .... / S: ....

Programming with lists

A *circular list* is a list-like data structure in which the last node refers to the beginning. Hence, a circular list has no end. Although it is possible to build cyclic data structures directly in Haskell, we will represent a circular list by a standard list, and pretend as if the tail of the list is connected to the head. Therefore we introduce the following datatype.

$$\textbf{newtype}\ \mathit{CircList}\ a = \mathit{CL}\ \{\mathit{fromCL} :: [a]\}$$

Define for this type the following functions.

$$\begin{array}{l}
\mathit{size} :: \mathit{CircList}\ a \to \mathit{Int} \\
\mathit{current} :: \mathit{CircList}\ a \to \mathit{Maybe}\ a \\
\mathit{insert} :: [a] \to \mathit{CircList}\ a \to \mathit{CircList}\ a \\
\mathit{delete} :: \mathit{Int} \to \mathit{CircList}\ a \to \mathit{CircList}\ a \\
\mathit{rotate} :: \mathit{Int} \to \mathit{CircList}\ a \to \mathit{CircList}\ a \\
\mathit{takeFrom} :: \mathit{Int} \to \mathit{CircList}\ a \to [a]
\end{array}$$

The function *size* returns the number of elements in the list; *current* returns the element that is currently the head of the list; *insert xs* extends a list by inserting the elements *xs* right before the current element (i.e. the head); *delete* n deletes n elements from the list. The operations so far can be implemented without taking into account the fact that the lists should be considered as circular. This, however, is different for the other two functions. The function *rotate* n rotates the list by moving the elements n positions to the *left*. Of course, all elements disappearing on the left should reappear on the right; *takeFrom* takes the first n elements of the list (after unrolling if necessary).

The following examples illustrate the intended functionality:

$$\begin{array}{l}
\mathit{size}\ (\mathit{CL}\ [1,2,3,4,5]) = 5 \\
\mathit{current}\ (\mathit{CL}\ [1,2,3,4,5]) = \mathit{Just}\ 1 \\
\mathit{current}\ (\mathit{CL}\ [\,]) = \mathit{Nothing} \\
\mathit{rotate}\ 3\ (\mathit{CL}\ [1,2,3,4,5]) = \mathit{CL}\ [4,5,1,2,3] \\
\mathit{current}\ \$\ \mathit{rotate}\ 4\ (\mathit{CL}\ [1]) = \mathit{Just}\ 1 \\
\mathit{insert}\ [7,6]\ (\mathit{CL}\ [1,2,3,4,5]) = \mathit{CL}\ [7,6,1,2,3,4,5] \\
\mathit{delete}\ 2\ (\mathit{CL}\ [1,2,3,4,5]) = \mathit{CL}\ [3,4,5] \\
\mathit{takeFrom}\ 7\ (\mathit{CL}\ [1,2,3]) = [1,2,3,1,2,3,1]
\end{array}$$

Question		
2	Name: .... Student number:	

Programming with lists — *continued*

newtype $CircList\ a = CL\ \{fromCL :: [a]\}$

a) $size :: CircList\ a \rightarrow Int$

b) $current :: CircList\ a \rightarrow Maybe\ a$

c) $insert :: [a] \rightarrow CircList\ a \rightarrow CircList\ a$

d) $delete :: Int \rightarrow CircList\ a \rightarrow CircList\ a$

e) $rotate :: Int \rightarrow CircList\ a \rightarrow CircList\ a$

f) $takeFrom :: Int \rightarrow CircList\ a \rightarrow [a]$

Question		
2	Name: .... Student number:	

Programming with lists — *continued*

Define a predicate

$$equalCL :: (Eq\ a) \Rightarrow CircList\ a \to CircList\ a \to Bool$$

yielding *True* if and only if both lists contain the same elements in the same order, but not necessarily with the same current element. For example:

$$equalCL\ (CL\ [1,2,3])\ (CL\ [1,2,3]) = True$$
$$equalCL\ (CL\ [1,2,3,4,5])\ (CL\ [3,4,5,1,2]) = True$$
$$equalCL\ (CL\ [1,2,3])\ (CL\ [2,3]) = False$$
$$equalCL\ (CL\ [1,1])\ (CL\ [1]) = False$$
$$equalCL\ (CL\ [1,2,3,4,5])\ (CL\ [1,2,3,5,4]) = False$$

g) $equalCL :: CircList\ a \to CircList\ a \to Bool$

Question		14 points
3	Name: .... Student number:	$(3+5+6)$ Σ: .... / S: ....

Datatypes

Consider the following datatype of quadtrees and two auxiliary functions.

$$\mathbf{data}\ QTree = Black \mid White \mid Node\ (QTree, QTree, QTree, QTree)$$
$$listToT4\ [a, b, c, d] = (a, b, c, d)$$
$$t4ToList\ (a, b, c, d) = [a, b, c, d]$$

The size of a quadtree is the total number of nodes of that tree (leafs as well as internal nodes). The function that calculates the size is defined by:

$$size :: QTree \to Int$$
$$size\ Black\ = 1$$
$$size\ White = 1$$
$$size\ (Node\ sts) = 1 + sum\ (map\ size\ (t4ToList\ sts))$$

a) Define a single recursive function that computes the difference between the number of white leafs and the number of blacks leaves.

$$diff :: QTree \to Int$$

Assume that the following datatype of binary digits is given.

$$\textbf{data}\ Bit = O \mid I\ \textbf{deriving}\ (Eq, Ord)$$

A quadtree can be represented as a sequence of bits. This is done by preorder traversal. Each internal node is rendered as a I and each leaf as a O. In the case of an internal node, the subsequent bits are representations of the four subtrees. For a leaf, the following bit is a O for a black leaf and I for a white leaf. Take, for example, the following quadtrees.

$t1 = White$
$t2 = Node\ (Black, White, Black, White),$
$t3 = Node\ (Black, Node\ (Black, White, White, White), Node\ (White, Black, Black, Black), White)$

These trees are converted into

$[O, I]$ — t1
$[I, O, O, O, I, O, O, O, I]$ — t2
$[I, O, O, I, O, O, O, I, O, I, O, I, I, O, I, O, O, O, O, O, O, O, I]$ — t3

b) Write a function *compress* that encodes a given quatree as a bit string.

$$compress :: QTree \to [Bit]$$

c) Write a function *decompress* that converts a bit string back to the original quadtree. You may assume that the bit string correctly represents a quadtree.

$$decompress :: [Bit] \rightarrow QTree$$

Question		**10 points**
	Name:	$(3 + 7)$
4	Student number:	Σ: .... / S: ....

foldr (induction)

In this exercise we assume that the following definition of *reverse* in terms of *foldr* is given:

$$reverse = foldr\ (\backslash x\ r \to r +\!\!+ [x])\ [\,]$$

The function *foldr* is defined as usual:

$$foldr\ op\ e\ [\,] = e$$
$$foldr\ op\ e\ (x : xs) = x\ `op`\ (foldr\ op\ e\ xs)$$

Use induction over the structure of the list xs to show:

$$reverse\ (xs +\!\!+ ys) = reverse\ ys +\!\!+ reverse\ xs$$

If needed you may assume that $xs +\!\!+ [\,] = xs = [\,] +\!\!+ xs$.

a) **Case $xs = [\,]$:**

b) **Case** $\boldsymbol{xs = a : as}$ **:**

Question		8 points
5	Name: .... Student number:	Σ: .... / S: ....

foldr (fusion)

Recall the fusion law from the lectures:

$$f \circ \mathit{foldr}\ (\triangleright)\ e = \mathit{foldr}\ (\blacktriangleright)\ (f\ e) \qquad \Longleftarrow \qquad f\,(a \triangleright b) = a \blacktriangleright f\,b$$

a) Use the fusion law to show

$$\mathit{reverse} \circ \mathit{reverse} = \mathit{id}$$

where *reverse* is defined in the same way as in the previous question. *Hint:* use the fact that $\mathit{foldr}\ (:)\ [\,] = \mathit{id}$. You also may need the property of *reverse* from the previous question.

Question		**12 points**
	Name:	$(4 + 6 + 2)$
6	Student number:	Σ: .... / S: ....

Program derivation

Reconsider the definition of *reverse* of the previous questions, repeated below

$$reverse = foldr\ (\backslash x\ r \to r +\!\!+ [x])\ [\,]$$

This function has quadratic running time because of the repeated invocations of $+\!\!+$. To improve the running time we eliminate these calls by specifying

$$revApp\ xs\ ys = foldr\ (\backslash x\ r \to r +\!\!+ [x])\ [\,]\ xs +\!\!+ ys$$

Derive an implementation of *revApp* from this specification.

a) case $xs = [\,]$:

$$\begin{aligned} & revApp\ [\,]\ ys \\ = \ & ... \end{aligned}$$

b) case $xs = a : as$:

$$\begin{aligned} & revApp\ (a : as)\ ys \\ = \ & ... \end{aligned}$$

c) Express *reverse* in terms of *revApp*

Question	Name:	
	Student number:	

Question	Name: .... Student number:	

Question	Name: .... Student number:	