EXAM FUNCTIONAL PROGRAMMING
Tuesday the 5th of November 2013, 9.00 h. - 12.00 h.

Name: Student number:

Before you begin: Do not forget to write down your name and student number above. If necessary, explain your answers (in English or Dutch). For multiple choice questions, clearly circle what you think is the best answer. Use the empty boxes under the other questions to write your answer and explanations in. Use the empty paper provided with this exam only as scratch paper (kladpapier). At the end of the exam, only hand in the filled-in exam paper. Answers will not only be judged for correctness, but also for clarity and conciseness. A total of one hundred points can be obtained; divide by 10 to obtain your grade. Good luck!

1. We want to implement a function that computes the size of a value of a given type by means of type classes. Hence, we introduce:

```
class Sizable t where
  size :: t -> Int
  -- An Int is an atomic value:
instance Sizable Int where
  size i = 1
```

(i) Give an instance declaration for sizing lists. The size of a list is obtained by adding the sum of the sizes of its elements to the number of elements in the list.

$\boxed{\ldots/\mathbf{8}}$ The simplest implementation is:

```
instance Sizable a => Sizable [a] where
  size xs = length xs + sum (map size xs)
}
```

Essential here is that you also compute the *size* of all the elements in the list, and to understand that you also need to add *Sizable a* as a condition in the instance declaration.

An alternative that avoids *map* and *sum* is

```
instance Sizable a => Sizable [a] where
  size []       = 0
  size (x : xs) = 1 + size x + size xs
```

And with a *foldr* you get

```
instance Sizable a => Sizable [a] where
  size = foldr (\x r -> size x + 1 + r)
```

Score break down: 3 for the condition, 5 for the code. In the code you get 2 for counting the elements in the list, and 3 for computing the sizes of the elements and adding them up. For small syntactic issues you can get the occasional -1.

2. In this question we deal with a function $perms :: [a] \to [[a]]$ which returns all the permutations (i.e., all possible orderings) of the argument list.

(i) What are the permutations of [1,2,3,4] that start with 1?

$\boxed{\dots/\mathbf{4}}$ [1,2,3,4], [1,2,4,3], [1,3,2,4], [1,3,4,2], [1,4,2,3], [1,4,3,2].
If you show that you understand what a permutation is, you still get 3 points.

(ii) Explain how you can compute $perms\ (x : xs)$ from $perms\ xs$ (for example by using concrete values for x and xs)

$\boxed{\dots/\mathbf{6}}$ Here you should explain that given a result, say $rs = [[2,3],[3,2]] = perms\ [2,3]$, you can extend it to an answer for $[1,2,3]$, by taking every element in rs, and put 1 into that list in every possible position. For $[2,3]$ you then get $[[1,2,3],[2,1,3],[2,3,1]]$, and similarly for $[3,2]$. The results is then the concatenation of these two.

(iii) Now, write the function $perms :: [a] \to [[a]]$

$\boxed{\dots/\mathbf{9}}$ The function and the helper function it needs can be found in the slides, and are:

$$
\begin{aligned}
&between && :: a \to [a] \to [[a]] \\
&between\ e\ [\,] && = [[e]] \\
&between\ e\ (y : ys) && = (e : y : ys) : map\ (y{:})\ (between\ e\ ys)
\end{aligned}
$$

$$
\begin{aligned}
&perms\ [\,] && = [[\,]] \\
&perms\ (x : xs) && = concat\ (map\ (between\ x)\ (perms\ xs))
\end{aligned}
$$

The code for *perms* gets 4 points (1 for the base case, 3 for the recursive), the code for *between* 5 (2 for the base case, 3 for the recursive). General rule: no code but a correct signature gets you one point.

3. In this question we again see the *perms* function of the previous question. Even if you failed to come up with an implementation for *perms*, you may still be able to answer this one.

   (i) Give a QuickCheck property that can check that all lists in the outcome of *perms* have the right length.

   $\boxed{\ldots/\mathbf{8}}$

   $propi\ xs = all\ (\equiv length\ xs)\ (map\ length\ (perms\ xs))$

   (ii) Give the same property as at (i), but now so that it will only check this for input lists that have no duplicate elements.

   $\boxed{\ldots/\mathbf{6}}$

   $propii\ xs = allUnique\ xs \Longrightarrow all\ (\equiv length\ xs)\ (map\ length\ (perms\ xs))$
   where
   $allUnique\ xs = length\ (nub\ xs) \equiv length\ xs$
   $allUnique'\ [\,] = True$
   $allUnique'\ (x : xs) =$ **if** $elem\ x\ xs$ **then** $False$ **else** $allUnique'\ xs$

   $allUnique'$ is an alternative for $allUnique$. You get 3 points for the $allUnique\ xs{\Longrightarrow}$, and 2 for the implementation of $allUnique$ itself, 1 point for keep the rest of the property as it is.

   (iii) Chances may be small that a randomly generated list of *Int*s has no duplicates, and QuickCheck may give up in despair. Define a generator to generate random lists with no duplicate elements.

   $\boxed{\ldots/\mathbf{5}}$ For writing a generator like this you get 3 points:

   $genNoDups :: Gen\ [Int]$
   $genNoDups =$ **do**
   $xs \leftarrow arbitrary$ -- Can generate any list
   $return\ (nodup\ xs)$

   Now there are many possibilities for *nodup*. One is to have $nodup = nub$. Also possible is function like the *mksorted* from the slides like this (but the disadvantage is that the results is less random, since it is monotonically increasing):

   $nodup :: [Int] \rightarrow [Int]$
   $nodup\ [\,] = [\,]$
   $nodup\ [x] = [x]$
   $nodup\ (x : y : ys) = x : nodup\ ((x + 1 + abs\ y : ys))$

   Either of these gets you 2 points.

4. The following multiple choice questions are each worth 5 points.

$\boxed{\ldots/\mathbf{20}}$ **(i).c, (ii).a, (iii).d, (iv).b**. Some explanations on (iii) and (iv) are given below. The rest you can check for yourself in a ghci session.

(i) Someone tries to write a function $\mathit{revDigits} :: \mathit{Int} \to \mathit{Int}$ that "reverses the digits in an *Int*"; so 123 is mapped onto 321. Which is the correct solution?

a. $\mathit{revDigits}\ i = \mathit{foldl}\ (\lambda \mathit{ds}\ x \to x : \mathit{ds})\ \texttt{""}\ (\mathit{show}\ i)$

b. $\mathit{revDigits}\ i = \mathit{foldr}\ (\lambda x\ \mathit{ds} \to \mathit{ds} +\!\!+ [x])\ \texttt{""}\ (\mathit{show}\ i)$

c. $\mathit{revDigits}\ i = \mathit{revDigits}'\ i\ 0$
   where $\mathit{revDigits}'\ 0\ r = r$
   $\mathit{revDigits}'\ i\ r = \mathit{revDigits}'\ (i\ \text{‘}\mathit{div}\text{‘}\ 10)\ (r * 10 + i\ \text{‘}\mathit{mod}\text{‘}\ 10)$

d. $\mathit{revDigits}\ i = \mathit{revDigits}'\ i\ 0$
   where $\mathit{revDigits}'\ 0\ r = r$
   $\mathit{revDigits}'\ i\ r = \mathit{revDigits}'\ (i\ \text{‘}\mathit{mod}\text{‘}\ 10)\ (r * 10 + i\ \text{‘}\mathit{div}\text{‘}\ 10)$

(ii) What is the type of *concat* . *concat*?

a. $[[[a]]] \to [a]$
b. $[a] \to [[a]] \to [a]$
c. $[[a]] \to [[a]] \to [[a]]$
d. none of the above

(iii) Given are the following two statements:

I The side effects possible because of IO make Haskell an impure language
False. Monads are Haskell's way of dealing with IO in a way that is pure. The monads prevent the programmer from being able to access the IO monads internal state, and from duplicating the world

II Using *seq* can never make your program slower
Of course it can. The expression fib 20000 ‘*seq*‘ 2 *is really much slower to compute than just returning* 2.

a. Both I and II are true
b. Only I is true
c. Only II is true
d. Both I and II are false

(iv) Given the definition of strict function application \$! from the slides, consider the following two statements:

I $\mathit{snd}\ \$!\ (\bot_1, \bot_2)$ equals $\bot_2$
This is true, although having \$! *here does not really play a role, since the pair is already in WHNF.*

II $\mathit{length}\ \$!\ \mathit{map}\ \bot\ [1, 2]$ equals $\bot$
This is incorrect. length does not need the values of the elements of the mapped list, so the answer is simply 2.

a. Both I and II are true
b. Only I is true
c. Only II is true
d. Both I and II are false

5. A *heap* is a data structure described by a data type quite similar to a search tree:

$$\textbf{data}\ \mathit{Heap}\ a = \mathit{Top}\ a\ (\mathit{Heap}\ a)\ (\mathit{Heap}\ a)$$
$$\mid \mathit{Leaf}$$

with the so called *heap property* that the a value in a *Top* node is larger than or equal to the values in the roots of its child *Heaps*, which have this property themselves too. An example of a heap is

```
aHeap = Top 10 (Top 7 Leaf
                      (Top 4 Leaf Leaf))
               (Top 3 (Top 2 Leaf Leaf)
                      Leaf)
```

(i) Write a function $\mathit{checkHeap} :: \mathit{Ord}\ a \Rightarrow \mathit{Heap}\ a \rightarrow \mathit{Bool}$ which returns *True* if its argument has the required propery, and *False* otherwise. **Hint**: you may want to write a helper function $\mathit{checkHeap}' :: \mathit{Ord}\ a \Rightarrow a \rightarrow \mathit{Heap}\ a \rightarrow \mathit{Bool}$.

…/**8** The tricky part here is that you cannot access the values of the children easily to compare with. The solution below sends the value from the root to the children and makes the comparison there.

```
checkHeap Leaf = True
checkHeap h@(Top a _ _)
 = checkHeap' a h
   where checkHeap' f (Top c l r) = f ≥ c
                                    ∧ checkHeap' c l
                                    ∧ checkHeap' c r
         checkHeap' _ Leaf =          True
```

Having the *Leaf* case correctly handled (1pt), recursing both subtrees, on the right arguments (2pts each), doing the right comparisons (3 pts)

(ii) Write a function $\mathit{getFromHeap} :: \mathit{Ord}\ a \Rightarrow \mathit{Heap}\ a \rightarrow \mathit{Maybe}\ (a, \mathit{Heap}\ a)$, which – provided the heap is non-empty– returns the largest a value from its *Heap a* argument, together with a *Heap a* containing the rest of the values of its *Heap a* argument, and *Nothing* if the *Heap a* argument is a *Leaf*. Make sure the resulting *Heap a* again satisfies the heap property! **Hint**: the new root of the heap is either the root of the left subtree or the root of the right subtree.

…/**8**

```
getFromHeap Leaf        = Nothing
getFromHeap (Top a l r) = Just (a, mergeHeaps l r)
mergeHeaps Leaf r       = r
mergeHeaps l      Leaf  = l
mergeHeaps l@(Top lv ll lr) r@(Top rv lr rr)
   | lv ≥ rv            = Top lv (mergeHeaps ll lr) r
   | otherwise          = Top rv l                  (mergeHeaps lr rr)
```

The *getFromHeap* part gets you 3 points, reconstructing the heap after deleting the maximum gets you 5. Of these 5, the right recursive calls gives you 2 points, a correct comparison 1 points, constructing the right tree 2.

If you do not assume the heap to be a heap in the first place, reconstructing a heap becomes much harder.

6. Given the following definitions of *reverse*, and $(+\!\!+)$:

$$\begin{aligned}
&\mathit{reverse} :: [a] \rightarrow [a] \\
&\mathit{reverse}\ [\,] = [\,] \\
&\mathit{reverse}\ (x : xs) = \mathit{reverse}\ xs +\!\!+ [x]
\end{aligned}$$

$$\begin{aligned}
&(+\!\!+) :: [a] \rightarrow [a] \rightarrow [a] \\
&[\,] +\!\!+ ys = ys \\
&(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys),
\end{aligned}$$

(i) Prove by induction that $xs = xs +\!\!+ [\,]$. Do not forget to explain every step in your proof!

$\boxed{\ldots/\mathbf{5}}$ This is part of 5.20 in the lecture notes.
We need to prove:

$$xs = xs +\!\!+ [\,]. \tag{1}$$

With induction on the structure of xs.
Base case:

$$\begin{aligned}
& [\,] \\
= \ & [\,] +\!\!+ [\,] && \{\text{definition van } (+\!\!+)\}.
\end{aligned}$$

Induction step:

$$\begin{aligned}
& x : xs \\
= \ & x : (xs +\!\!+ [\,]) && \{\text{induction hypothesis}\} \\
= \ & (x : xs) +\!\!+ [\,] && \{\text{definition of } (+\!\!+)\}.
\end{aligned}$$

(ii) Prove by induction that $\mathit{reverse}\ (xs +\!\!+ ys) = \mathit{reverse}\ ys +\!\!+ \mathit{reverse}\ xs$. You may use the result of part (i), and a Lemma that says that $(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$ in your proof. Again, do not forget to explain every step in your proof!

$\boxed{\ldots/\mathbf{13}}$ Again with induction over the structure of xs.
Base case:

$$\begin{aligned}
& \mathit{reverse}\ ([\,] +\!\!+ ys) \\
= \ & \mathit{reverse}\ ys && \{\text{definition van } (+\!\!+)\} \\
= \ & \mathit{reverse}\ ys +\!\!+ [\,] && \{\text{Exercise (i))}\} \\
= \ & \mathit{reverse}\ ys +\!\!+ \mathit{reverse}\ [\,] && \{\text{definitie van } \mathit{reverse}\}.
\end{aligned}$$

Induction step:

$$\begin{aligned}
& \mathit{reverse}\ ((x : xs) +\!\!+ ys) \\
= \ & \mathit{reverse}\ (x : xs +\!\!+ ys) && \{\text{definition } (+\!\!+)\} \\
= \ & \mathit{reverse}\ (xs +\!\!+ ys) +\!\!+ [x] && \{\text{definition } \mathit{reverse}\} \\
= \ & (\mathit{reverse}\ ys +\!\!+ \mathit{reverse}\ xs) +\!\!+ [x] && \{\text{induction hypothese}\} \\
= \ & \mathit{reverse}\ ys +\!\!+ (\mathit{reverse}\ xs +\!\!+ [x]) && \{\text{Lemma})\} \\
= \ & \mathit{reverse}\ ys +\!\!+ \mathit{reverse}\ (x : xs) && \{\text{definitie van } \mathit{reverse}\}
\end{aligned}$$

Non-inductive proofs obtain no points, but if you have an equation in a non-inductive proof that also is present above, then you get a point.