**EXAM FUNCTIONAL PROGRAMMING**
Thursday the 6th of November 2014, 13.30 h. - 16.30 h.

| |
|---|
| Name:<br>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 (one and only) 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!

In any of your answers below you may (but do not have to) use the following well-known Haskell functions/operators: *id*, *concat*, *foldr* (and variants), *map*, *filter*, *const*, *flip*, *fst*, *snd*, *not*, (.), *elem*, *take*, *drop*, *takeWhile*, *dropWhile*, *head*, *tail*, (++), *lookup* and all members of the type classes *Eq*, *Num*, *Ord*, *Show* and *Read*.

1. (i) Define a type class *Finite a* (eindig), that has one member *values* that enumerates all (finitely many) values of type *a*.

   | . . . /**5** |
   |---|

   (ii) Define a suitable instance for *Finite Bool*.

   | . . . /**3** |
   |---|

   (iii) Define a suitable instance for *Finite* $(a, b, c)$ with a list comprehension, for the case that $a$, $b$ and $c$ are instances of *Finite*.

   | . . . /**6** |
   |---|

   (iv) Why is it not possible to add a member *size* :: *Int* (that returns the length of *values*) to the *Finite* type class?

   | . . . /**5** |
   |---|

2. In this question we deal with a function *segs* :: [ *a* ] −> [[ *a* ]] which returns all the segments of the argument list. A list *L1* is a segment of another list *L2*, if you can obtain *L1* from *L2* by dropping any number of elements (including 0) at the beginning of *L2*, and dropping any number of elements (including 0) at the end of *L2*.

   (i) What are the segments of [1,2,3,4]?

   | . . . /**4** |

   (ii) Explain how you can compute *segs* ( *x* : *xs* ) from *segs xs* (for example by using concrete values for *x* and *xs*)

   | . . . /**6** |

   (iii) Now, write the function *segs* :: [ *a* ] −> [[ *a* ]]

   | . . . /**6** |

(iv) Write a QuickCheck property *numberProp* :: [*Int*] −> *Property* that tests whether *segs xs* has the correct number of segments, but only for input lists of length at least 3.

| .../**6** |
|---|
| |

3. Given is the following datatype for trees:

   **data** *Tree* = *Leaf* | *Bin Tree Int Tree* **deriving** *Eq*

(i) Define a function *listLike* :: *Tree* −> *Bool* that returns *True* if every *Bin* node has at most one non-Leaf child.

| .../**7** |
|---|
| |

(ii) Assume that an instance *Arbitrary Tree* has been defined, write a generator *genNLLTree* :: *Gen Tree* for arbitrary trees that are *not* list-like.

| .../**7** |
|---|
| |

4. Given are the following definitions (with line numbers):

$$
\begin{aligned}
&(1)\ id\ x &&= x\\
&(2)\ flip\ f\ x\ y &&= f\ y\ x\\
&(3)\ reverse\ [\,] &&= [\,]\\
&(4)\ reverse\ (x : xs) &&= reverse\ xs\ {+}{+}\ [x]\\
&(5)\ foldr\ f\ e\ [\,] &&= e\\
&(6)\ foldr\ f\ e\ (x : xs) &&= f\ x\ (foldr\ f\ e\ xs)\\
&(7)\ foldl\ f\ e\ [\,] &&= e\\
&(8)\ foldl\ f\ e\ (x : xs) &&= foldl\ f\ (f\ e\ x)\ xs
\end{aligned}
$$

(i) Prove by induction that $foldr\ (:)\ [\,] = id$ (use the line numbers above when you refer to a particular given equation in your proof):

.../**7**

(ii) Prove by induction that $foldr\ f\ e\ (reverse\ xs) = foldl\ (flip\ f)\ e\ xs$ for all $f$, $e$ and $xs$ of the right type. You may use (without proof) the following lemma: $foldr\ f\ e\ (as ++ [b]) = foldr\ f\ (f\ b\ e)\ as$ for all suitable $f$, $e$, $as$ and $b$ (again, use the line numbers when you refer to a particular given equation in your proof).

| .../**13** |
|---|

5. ... /**25** The following multiple choice questions are each worth 5 points.

(i) Which of the following is true?

a. The function *return* is idempotent (i.e. *return* (*return a*) can safely be replaced by *return a*).

b. There exist expressions of type *IO* (*IO Int*).

c. If you define an instance of the class *Eq* you have at least to specify the function (==).

d. The class *Enum* has a fixed number of instances.

(ii) I A jargon is a special kind of domain-specific language.

II It is easier to achieve fluency with a deeply embedded DSL than with a shallowly embedded DSLs.

a. Both I and II are true

b. Only I is true

c. Only II is true

d. Both I and II are false

(iii) Which observation is correct when comparing the types of (*map map*) *map* and *map* (*map map*)?

a. The type of the first is less polymorphic than the type of the second.

b. The type of the first is more polymorphic than the type of the second.

c. The types are the same, since function composition is associative.

d. One of the expressions is type incorrect.

(iv) What is the type of *foldr flip*?

a. $b \rightarrow (b \rightarrow a \rightarrow b) \rightarrow [a] \rightarrow b$

b. $(a \rightarrow b) \rightarrow [b \rightarrow b] \rightarrow a \rightarrow b$

c. $(a \rightarrow b) \rightarrow [a \rightarrow (a \rightarrow b) \rightarrow b] \rightarrow a \rightarrow b$

d. The expression is type incorrect

(v) In the Haskell prelude the list constructor [] has been made an instance of the class *Monad*:

```
instance Monad [] where
  ma >= a2mb = concat (map a2mb ma)
  return a = [a]
```

Which of the following equals [*f x y* | *x* <- *expr1*, *y* <- *expr2*]?

a.
```
do return (f x y)
   where do x <- expr1
            y <- expr2
```

b.
```
do x <- expr1
   y <- expr2
   f x y
```

c.
```
do x <- expr1
   y <- expr2
   return (f x y)
```

d.
```
do y <- expr2
   x <- expr1
   return (f x y)
```