EXAM FUNCTIONAL PROGRAMMING

Thursday the 10th of November 2016, 13.30 h. - 16.30 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 (one and only) best answer. Use the empty boxes under the other questions to write your answer and explanations in. Use the blank 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 100 points can be obtained. Good luck!

Unless stated otherwise, in any of your answers below you may (but do not have to) use well-known Haskell functions/operators like: *replicate*, *id*, *concat*, *foldr* (and variants), *map*, *filter*, *const*, *all*, *any*, *flip*, *fst*, *snd*, *not*, (.), *elem*, *take*, *drop*, *takeWhile*, *dropWhile*, *head*, *tail*, *zip*, *reverse*, (++), *lookup*, *max*, *min* and all members of the type classes *Eq*, *Num*, *Ord*, *Show* and *Read*. For the QuickCheck questions you can use anything from the QuickCheck library, like *quickcheck*, *arbitrary*, *choose*, *forAll*, *oneOf*, *sized*, (===) and (==>).

1. TYPE CLASSES

(i) We define a datatype for describing a single communication in a network:

data *LogEntry t adr m* = *E t adr adr m*

The t represents a time stamp, the type *adr* represents some kind of address (eg. an IP address), and m represents the type of the messages. All these types are kept abstract. We have two fields of type *adr* as arguments to E because communication involves two parties.

Multiple machines collect this kind of logging data, and some of the entries on one machine have a corresponding entry on another. To integrate lists of *LogEntry*s for multiple machines, we would like to get rid of duplicates. Because the values of the time stamp are likely to be different, these should be ignored in the equality comparison, while the order of the *adr* fields may differ between two log entries that we still consider equal. The messages of the two log entries must however be exactly the same.

For example, if we choose *String* for t and *adr* and [*String*] for m, the following two values

le1 = *E* "12:00:00.00" "128.1.1.0:2000" "255.192.8.1:3000" ["There", "there"]
le2 = *E* "12:00:00.05" "255.192.8.1:3000" "128.1.1.0:2000" ["There", "there"]

should be considered equal.

Give the corresponding instance definition for *LogEntry* for the *Eq* class.

.../**8**

```
instance (Eq adr, Eq m) => Eq (LogEntry t adr m) where
  (E t1 adr11 adr12 m1) == (E t2 adr21 adr22 m2) =
    m1 == m2 && ((adr11 == adr21 && adr12 == adr22) ||
      (adr11 == adr22 && adr12 == adr21))
```

(ii) For some applications, a reasonable ordering for a series of *LogEntry*s is to order by time stamp, using the ordering relation of the time stamp type. Give an instance declaration of *LogEntry* for the *Ord* type class, by giving the correct definition for <=.

$\boxed{\ldots/\mathbf{5}}$

instance (*Ord t*, *Eq adr*, *Eq m*) => *Ord* (*LogEntry t adr m*) **where**
  (*E t1* _ _ _) <= (*E t2* _ _ _) = *t1* <= *t2*

(iii) A maybe surprising aspect is that *Eq adr* and *Eq m* have to be part of the context of the instance declaration above (so, if you forgot these, go back and add them now). Explain why these need to be present.

$\boxed{\ldots/\mathbf{3}}$ The *Ord* class uses equality to compute $x < y$ from knowing just $x <= y$ and then that $x < y$ if and only if $x <= y$ and not $x == y$.

(iv) Can you explain what is problematic in our current definitions of equality and ordering *LogEntry*s using the two example log entries given earlier? What is the cause of this problem?

$\boxed{\ldots/\mathbf{3}}$ The two definitions are inconsistent with each othere. Practically, it means that *le1* == *le2* but not both *le1* <= *le2* and *le1* >= *le2* which is what you would expect of a sensible combination.

2. MULTIPLE CHOICE $\boxed{\ldots/\mathbf{22}}$

a, a, a, b, a, b

Multiple choice questions with four choices are worth 5 points, those with two choices 3.

(i) **let** $i = \backslash\, y \rightarrow y$ **in** $i\ i$ is well-typed.

  a. The statement is true
  b. The statement is false

(ii) Which of the following is true?

  a. There exist expressions of type *IO* (*IO Int*).
  b. The function *return* is idempotent (i.e. *return* (*return a*) can safely be replaced by *return a*).
  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.

(iii) ["BO" ++ "OM" ‘*seq*‘ *sqrt* 16, *sin* 5.2] is well-typed.

  a. The statement is true
  b. The statement is false

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

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

(v) An advantage of deeply embedded DSLs is that DSL programs can be analyzed and optimized before being run.

  a. The statement is true
  b. The statement is false

(vi) External DSLs can be more easily combined than internal (aka embedded) DSLs.

  a. The statement is true
  b. The statement is false

3. LAWS

(i) Consider the following statement: *foldr op e xs* = *foldl op e xs* for all type compatible *op*, *e* and *xs*. Come up with a choice for *op*, *e* and *xs* that shows this theorem does not hold in general.

$\boxed{\ldots/\mathbf{4}}$ *foldl* $(-)\ 0\ [1, 2, 3] = -6$ and *foldr* $(-)\ 0\ [1, 2, 3] = 2$

(ii) Formulate conditions on *op* and *e* so that the above theorem becomes true.

$\boxed{\ldots/\mathbf{4}}$ First of all, the type of *op* should be $b \rightarrow b \rightarrow b$ for some b. e is a unit of *op*, and *op* should be associative. We get (for $xs = [x1, x2, x3]$) $((e\ op\ x1)\ op\ x2)\ op\ x3) == x1\ op\ (x2\ op\ (x3\ op\ e))$. So, e is a unit of *op* will allow us to drop e on both sides, and then we are left with a demand for associativity of *op*.
Another correct answer is to say that *op* should be commutative.

4. THE PIANO

We encode the clavier of a piano with its black and white keys as follows:

data *Piano* = *Black Piano* | *White Piano* | *Silence*

(i) Write a function *cv2bs* that converts a *Piano* to a list of booleans, where every black key becomes *True*, and every white key *False*.

.../**5**

```haskell
cv2bs :: Piano -> [Bool]
cv2bs Silence = []
cv2bs (Black p) = True : cv2bs p
c22bs (White p) = False : cv2bs p
```

(ii) Write a generator *genPiano* :: *Gen Piano*. You may reuse Arbitrary instances for all well-known types like *Int*, *Integer*, *Bool*, lists and tuples.

.../**4**

```haskell
genPiano :: Gen Piano
genPiano = do
  bs <- arbitrary :: Gen [Bool]   -- Integer is also okay, but how big is then the piano
  return (cv2p bs)

cv2p [] = Silence
cv2p (True : xs) = Black (cv2p xs)
cv2p (False : xs) = White (cv2p xs)
```

(iii) Give the *Haskell* code that makes *Piano* a member of the *Arbitrary* class (with the above generator, of course).

.../**3**

```haskell
instance Arbitrary Piano where
  arbitrary = genPiano
```

(iv) *ebonyAndIvory* :: *Piano* –> *Piano* –> *Piano* is a function that puts two *Piano*s side by side for a duet. Stevie Wonder and Paul McCartney insisted however that the function should be tested in the situation that both piano's have as many white keys as black keys; we call such a *Piano* balanced. Given a function *balanced* :: *Piano* –> *Bool* that checks that its argument *Piano* is balanced, define the QuickCheck property *balProp* :: *Piano* –> *Piano* –> *Property* that, given two balanced piano's, the result of *ebonyAndIvory* is a balanced piano.

.../**4**

```
balProp :: Piano -> Piano -> Property
balProp p1 p2 = (balanced p1 && balanced p2) ==> balanced (ebonyAndIvory p1 p2)
```

(v) What is the problem that arises when applying *quickcheck* to *balProp*?

.../**3** Generating two balanced piano's is not likely going to succeed quickly when we generate simply arbitrary piano's. So QuickCheck will give up trying.

(vi) Write a generator *genBalancedPiano* :: *Gen Piano* that yields arbitrary balanced *Piano*s by construction.

.../**6**

```
genBalancedPiano :: Gen Piano
genBalancedPiano = do
    i <- arbitrary  -- integer, half of the "length" of the piano
    genHelp i i  -- Black, White
  where
    genHelp :: Int -> Int -> Gen Piano
    genHelp 0 n = return (cv2p (replicate n False))  -- rep
    genHelp n 0 = return (cv2p (replicate n True))
    genHelp n m = do
      b <- arbitrary
      let newn = if b then n - 1 else n
      let newm = if b then m else m - 1
      r <- genHelp newn newm
      if b then
        return (Black r)
      else
        return (White r)
```

(vii) Adapt *balProp* from above to use the specialized generator.

.../**3**

```
balProp :: Property
balProp = forAll genBalancedPiano (\p1 -> \p2 -> balanced (ebonyAndIvory p1 p2))
```

Alternatively, you could wrap the piano in a different datatype and add an instance with the above generator. Then you get 2 pts.

5. TERMINATION AND STRICTNESS

(i) In the lecture we discussed *seq* and strict application (\$!). Define (\$!) in terms of *seq*.

$\boxed{\ldots/\mathbf{4}}$

$f \,\$!\, x = x \text{ `}seq\text{` } f\ x$

(ii) Give a (small) expression that includes a single *seq* and that does not terminate, but that does terminate when you remove the *seq* and its first argument.

$\boxed{\ldots/\mathbf{4}}$ This is not as easy as it seems, since *seq* only evaluates to weak head normal form. But *until* (*const False*) *id* 1 ‘*seq*‘ 2 does the job. Note that $[1\,..]$ ‘*seq*‘ 2 does not! For that, we need deepseq.

6. INDUCTION

Given is the following code:

$$
\begin{aligned}
&(1)\ map\ f\ [\,] &&= [\,]\\
&(2)\ map\ f\ (x : xs) &&= f\ x : map\ f\ xs\\
&(3)\ foldr\ f\ e\ [\,] &&= e\\
&(4)\ foldr\ f\ e\ (x : xs) &&= f\ x\ (foldr\ f\ e\ xs)
\end{aligned}
$$

Prove by induction that for all type compatible f, g, e and xs, $foldr\ f\ e\ (map\ g\ xs) = foldr\ h\ e\ xs$, where $h = \backslash\, x\ r \mathrel{-\!\!>} f\ (g\ x)\ r$.

$\boxed{\ldots/\mathbf{15}}$ We perform induction on xs (duh).

Case xs is of the form []: $foldr\ f\ e\ (map\ g\ [\,]) = (1) = foldr\ f\ e\ [\,] = (3) = e$. Form the other direction: $foldr\ h\ e\ [\,] = (3) = e$.

Case xs is of the form (x:xs): $foldr\ f\ e\ (map\ g\ (x{:}xs)) = (2) = foldr\ f\ e\ (g\ x{:}(map\ g\ xs)) = (4) = f\ (g\ x)\ (foldr\ f\ e\ (map\ g\ xs)) = IH = f\ (g\ x)\ (foldr\ h\ e\ xs)$.

From the other side $foldr\ h\ e\ (x : xs) = (4) = h\ x\ (foldr\ h\ e\ xs) = (unfold\ h) = f\ (g\ x)\ (foldr\ h\ e\ xs)$.