FP 2005-2006, Eindtoets

April 19, 2006, 14.00-17.00

The exam consists of 4 multiple choice questions (1 point each) and three open questions (2 points each). Not answering a multiple choice question will give you 0.25 point. Hand in the second and third page, with choices made in the corresponding box and open questions answered.

1. Which of the following is a correct definition of *inits*?
   (a) $foldr\ (\lambda x\ r \rightarrow [[\,]] : map\ (x{:})\ r)\ [[\,]]$
   (b) $foldr\ (\lambda x\ r \rightarrow [\,] : map\ (x{:})\ r)\ [[\,]]$ **CORRECT**
   (c) $foldr\ (\lambda x\ r \rightarrow map\ (x{:})\ r)\ [\,]$
   (d) $foldr\ (\lambda x\ r \rightarrow map\ (x{:})\ r)\ [[\,]]$
2. Which of the following is a correct definition of *tails*?
   (a) *map reverse.reverse.tails.reverse*
   (b) *reverse.inits.map reverse.reverse*
   (c) *reverse.map (inits.reverse).reverse*
   (d) *map reverse.reverse.inits.reverse* **CORRECT**
3. Which of the following is a correct definition of *transpose*?
   (a) $foldr\ (zipWith\ (+\!\!+))\ (repeat\ [\,])$
   (b) $foldr\ (zipWith\ (:))\ (repeat\ [[\,]])$
   (c) $foldl\ (zipWith\ (+\!\!+))\ (repeat\ [\,])$
   (d) $foldr\ (zipWith\ (:))\ (repeat\ [\,])$ **CORRECT**
4. Which of the following is true?
   (a) The expressions **do** *return a* is the same as *return a*. **CORRECT**
   (b) Expressions of type *IO a* cannot occur inside other expressions.
   (c) Expressions of type *IO a* can only occur as a subexpression of expressions of type *IO b* for a suitable *b*.
   (d) The expression *return* (*return a*) can be replaced by *return a*.

With respect to alternative (c) notice that when we have:

$$\begin{array}{l}
\mathit{ringBell} :: \mathit{IO}\ () \\
\mathit{times}\ \mathit{action}\ 0 = \mathit{return}\ () \\
\mathit{times}\ \mathit{action}\ n = \mathbf{do}\ \mathit{action} \\
\qquad\qquad\qquad\qquad\quad \mathit{times}\ \mathit{action}\ (n-1)
\end{array}$$

that the expression *times ringBell* has a subexpression of type $\mathit{IO}\ ()$ while its type is not of the form $\mathit{IO}\ ()$.

With respect to alternative (d) notice that when $\mathit{return}\ a$ has type $\mathit{IO}\ a$ for some a, then $\mathit{return}\ (\mathit{return}\ a)$ has type $\mathit{IO}\ (\mathit{IO}\ a)$. Since these types are different they will definitely not be the same.

5. Someone wants to compare search trees, which are defined as:

$$\begin{array}{l}
\mathbf{data}\ \mathit{SearchTree}\ a = \mathit{Branch}\ (\mathit{SearchTree}\ a)\ a\ (\mathit{SearchTree}\ a) \\
\qquad\qquad\qquad\qquad\quad \mid \mathit{Leaf}
\end{array}$$

and requires the tree $\mathit{insert}\ 3\ (\mathit{insert}\ 5\ t)$ to be equal to $\mathit{insert}\ 5\ (\mathit{insert}\ 3\ t)$, i.e. two trees are the same if they contain the same elements irrespective of the order in which they were inserted.

(a) Write a function $\mathit{flattenTree} :: \mathit{SearchTree}\ a \rightarrow [a]$ which computes a list of all the values in the tree in increasing order, and does so in linear time.

(b) Use this function to define the overloaded functions $\equiv$ and $\not\equiv$ fpr search trees.

We start with the straightfoward *flattenTree*, from now on called *ft*:

$$\begin{array}{ll}
\mathit{ft}\ \mathit{Leaf} & = [\,] \\
\mathit{ft}\ (\mathit{Branch}\ l\ v\ r) & = \mathit{ft}\ l +\!\!\!+ (v : \mathit{ft}\ r)
\end{array}$$

Unfortunately this solution may be expensive, so we may use the trick with an accumulation parameter:

$$\begin{array}{lll}
\mathit{ft}'' :: \mathit{SearchTree}\ a \rightarrow [a] \rightarrow [a] & & \\
\mathit{ft}'\ \mathit{Leaf} & \mathit{rest} & = \mathit{rest} \\
\mathit{ft}'\ (\mathit{Branch}\ l\ v\ r) & \mathit{rest} & = \mathit{ft}'\ l\ (v : \mathit{ft}'\ r\ \mathit{rest}) \\
\mathit{ft}\ t & & = \mathit{ft}'\ t\ [\,]
\end{array}$$

Another way of writing essentially the same is to use the efficient way for concatenating lists:

$$\begin{array}{ll}
\mathit{ft}'' :: \mathit{SearchTree}\ a \rightarrow ([a] \rightarrow [a]) & \\
\mathit{ft}''\ \mathit{Leaf} & = \mathit{id} \\
\mathit{ft}''\ (\mathit{Branch}\ l\ v\ r) & = \mathit{ft}''\ l.(v:).\mathit{ft}''\ r \\
\mathit{ft}\ t = (\mathit{ft}''\ t)\ [\,] &
\end{array}$$

An elegant solution given by two people is:

$$\begin{array}{ll}
\mathit{ft}\ \mathit{Leaf} & = [\,] \\
\mathit{ft}\ (\mathit{Branch}\ \mathit{Leaf}\qquad\qquad v\ \ r) & = v : \mathit{ft}\ r \\
\mathit{ft}\ (\mathit{Branch}\ (\mathit{Branch}\ l\ v\ r)\ w\ \mathit{rr}) & = \mathit{ft}\ (\mathit{Branch}\ l\ v\ (\mathit{Branch}\ r\ w\ \mathit{rr}))
\end{array}$$

We can now use an **instance** declaration to get the required eauality functions for bags that are represented by *SearchTree*'s:

```
instance Eq a ⇒ Eq (SearchTree a)
  a ≡ b = ft a ≡ ft b
```

Note that the default definition of $\not\equiv$ in the class definition takes care of the $\not\equiv$ case. Furthermore many people have forgotten the *Eq a* part of the instance definition, or have just writte **instance** *Eq SearchTree* **where**....!

Many, many people still write code like:

```
ft l v r = l : v : r
ft (SearchTree l) v (SearchTree r) = ft (SearchTree a) ...    -- find all the mistakes!!
```

In the next test such incorrect usage of syntax, and mixing up : and ++ will no longer be tolerated. I accept that you have difficulty in inventing smart algorithms, but at least you should by now know how to corrently use patterns and how to write simple type correct code.

6. The function *numberOfCombinations* : $[Int] \to Int \to Int$ computes the number of combinations in which we can select elements from the first list that sum up to the second argument. You may assume that all the elements in the first list are different and positive. Elements may be selected more than once. Zo is *numberofCombinations* $[5, 10, 20]$ 40 gelijk aan 9, want dat is de lengte van de lijst [$[5, 5, 5, 5, 5, 5, 5, 5]$, $[5, 5, 5, 5, 5, 5, 10]$, $[5, 5, 5, 5, 10, 10]$, $[5, 5, 5, 5, 20]$,$[5, 5, 10, 10, 10]$, $[5, 5, 10, 20]$, $[10, 10, 10, 10]$, $[10, 10, 20]$, $[20, 20]$].

   (a) Define the function *numberOfCombinations*

   (b) Redefine the function in such a way that it uses arrays to remember common calls, and the overall complexity becomes linear in the second argument, assuming that indexing takes constant time.

Solution:

```
noc _ 0                 = 1   -- Note that this should come before the next alternative!!
noc [] _                = 0
noc (x : xs) n | n < 0  = 0
               | otherwise = noc (x : xs) (n − x) + noc xs n
```

Notice that in this code there may be many calls to the function *noc* with the same arguments, so we decide to remember them using arrays:

```
module TestTest where
import Array
moc :: [Int] → Int → Int
moc l n =
  let last = length l
      lv = array (1, last) [(i, l !! (i − 1)) | i ← [1 .. last]]
      res :: Array (Int, Int) Int
      res = array ((0, 0), (n, last))
            ([((0, i), 1)                        | i ← [0 .. last]] ++
             [((j, 0), 0)                        | j ← [1 .. n]    ] ++
             [((j, i), res ! (j, i − 1) + if j ⩾ lv ! i
                                          then res ! (j − lv ! i, i)
```

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ **else** $0) \mid j \leftarrow [1\mathinner{\ldotp\ldotp}n], i \leftarrow [1\mathinner{\ldotp\ldotp}\mathit{last}]]$

$\qquad\qquad)$

$\quad$ **in** $\mathit{res}\,!\,(n, \mathit{last})$

$\mathit{main} = \mathit{moc}\ [5, 10, 20]\ 40$

Since almost no solution coming close to this has been given this part has not been taken into account for the final mark. However bonusses were given for those who at least tried.

7. Give an inductive proof of the fact that $\mathit{map}\ f\ (\mathit{xs} +\!\!+\ \mathit{ys}) \equiv \mathit{map}\ f\ \mathit{xs} +\!\!+\ \mathit{map}\ f\ \mathit{ys}$.

   See lecture notes **page 167**.

Name:

Student nr:

Bachelor program: Inf/CKI/...

Answers to the mutiple choice questions:

1	2	3	4

QUESTION 5, the functions *flattenTree* and the instance of *Eq*:

(see other side/ zie andere zijde)

QUESTION 6, the functions *numberOfCombinations*:

Name:

Student nr:

QUESTION 7, the proof of $map\ f\ (xs \mathbin{+\!\!+} ys) \equiv map\ f\ xs \mathbin{+\!\!+} map\ f\ ys$

vervolg 7