FP 2009-2010, Eindtoets
2010, Feb 3 , 14.00-17.00

Hand in the separate sheet with the 6 solutions. Don't forget to fill out your name! If you are sure your solution does not even come near to what is expected leave the corresponding entry blank; this will considerably speed up the marking process.

1. **Flattening search trees (1)**
   Consider the type of binary search trees:

   $$\begin{aligned}&\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\\ &\qquad\qquad\ \mid\ \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)\end{aligned}$$

   and assume that for each node it holds that the values in the left subtree are strictly smaller than the value of type a in that node, and that the values in the right subtree are strictly larger than that value.

   Write a function $\mathit{rft} :: \mathit{Tree}\ a \to [\,a\,]$ (rft stands for *reverse flatten tree*), which returns a descending list containing the a-values from the nodes of the tree. Do not use the function *reverse*!

   $$\begin{aligned}&\mathit{rft} = \mathit{rft}'\ [\,]\\ &\quad\textbf{where}\ \mathit{rft}'\ \mathit{Leaf} = \mathit{id}\\ &\qquad\qquad\ \mathit{rft}'\ (\mathit{Node}\ l\ v\ r) = \mathit{rft}'\ r\ .\ (v{:})\ .\ \mathit{rft}'\ l\end{aligned}$$

   Some people still write things like $\mathit{rft}\ r : v : \mathit{rft}\ l$.

2. **Induction Proof (1)**
   Prove by induction that $\mathit{map}\ f\ (x +\!\!+ y) = \mathit{map}\ f\ x +\!\!+ \mathit{map}\ f\ y$.

   $$\begin{aligned}&\mathit{map}\ f\ ([\,] +\!\!+ \mathit{ys}) = \mathit{map}\ f\ [\,]\ +\!\!+ \mathit{map}\ f\ \mathit{ys}\\ &\mathit{map}\ f\ \mathit{ys} = [\,]\ +\!\!+ \mathit{map}\ f\ \mathit{ys}\\ &\mathit{map}\ f\ \mathit{ys} = \mathit{map}\ f\ \mathit{ys}\end{aligned}$$

   $$\begin{aligned}&\mathit{map}\ f\ ((x : \mathit{xs})\ +\!\!+ \mathit{ys}) &&= \{\text{-unfold ++ -}\}\\ &\mathit{map}\ f\ (x : (\mathit{xs}\ +\!\!+ \mathit{ys})) &&= \{\text{-definition of map -}\}\\ &f\ x : \mathit{map}\ f\ (\mathit{xs}\ +\!\!+ \mathit{ys}) &&= \{\text{-induction hypothesis -}\}\\ &f\ x : (\mathit{map}\ f\ \mathit{xs} +\!\!+ \mathit{map}\ f\ \mathit{ys}) &&= \{\text{-definition of map -}\}\end{aligned}$$

   $$\begin{aligned}&\mathit{map}\ f\ (x : \mathit{xs})\ +\!\!+ \mathit{map}\ f\ \mathit{ys} &&= \{\text{-definition of map -}\}\\ &(f\ x : \mathit{map}\ f\ \mathit{xs}) +\!\!+ \mathit{map}\ f\ \mathit{ys} &&= \{\text{-definition of append -}\}\\ &f\ x : (\mathit{map}\ f\ \mathit{xs}\ +\!\!+ \mathit{map}\ f\ \mathit{ys})\end{aligned}$$

3. **Permutations (2)**
   Write a functions $\mathit{perms} :: [\,a\,] \to [[\,a\,]]$ which returns a list containing all permutations of the input list.

   see lecture notes

4. **Substitution and Evaluation (2)**
   The following data type was used to represent terms in the Prolog interpreter:

   $$\begin{aligned}&\textbf{type}\ \mathit{Ident} = \mathit{String}\\ &\textbf{data}\ \mathit{Term} = \mathit{Con}\ \mathit{Int}\\ &\qquad\qquad\ \mid\ \mathit{Var}\ \mathit{Ident}\\ &\qquad\qquad\ \mid\ \mathit{Fun}\ \mathit{String}\ [\,\mathit{Term}\,]\\ &\qquad\qquad\ \textbf{deriving}\ \mathit{Eq}\end{aligned}$$

   The function $\mathit{subst} :: [(\mathit{Ident}, \mathit{Term})] \to \mathit{Term} \to \mathit{Maybe}\ \mathit{Term}$ tries to replace all variables in a *Term* by the *Term* associated with that variable in the first parameter. In case the *Term* contains variables which cannot be found in the $[(\mathit{Ident}, \mathit{Term})]$ the whole subsititution returns *Nothing*, and otherwise *Just* t in which t stands for the result of the substitution.

(a) Define the function *subst*.

```
subst _ t@(Con _)       = Just t
subst env (Var v)       = lookup v env
subst env (Fun f args)  = do as ← lookupList env args
                             return (Fun f as)

lookupList' env []      = Just []
lookupList' env (a : as) = do r  ← subst env a
                              rs ← lookupList env as
                              return (r : rs)
  -- or
subst' _   t@(Con _)    = Just t
subst' env (Var v)      = lookup v env
subst' env (Fun f args) = fmap (Fun f) (mapM (subst' env) args)
```

(b) Write a function *evalTerm* :: *Term* → *Int* which evaluates a *Term*, assuming that the *Term* does not contain free variables anymore, and that the function symbols which occur are either `"+"` or `"-"`. You may assume these operators have indeed the right number of arguments in the list.

```
eval (Con i) = i
eval (Fun f (x : y: _)) = case f of
                            "+" → eval x + eval y
                            "-" → eval x − eval y
```

5. **Classes (2)**

The operators for parser combinators we have seen are introduced in a Haskell module *Control.Applicative* by the classes:

```
class Applicative f where
  (<*>) :: f (b → a) → f b → f a
  pure :: a                → f a   -- like pSucceed
class Alternative f where
  (<|>) :: f a → f a → f a
  fail  ::             f a
```

Give corresponding instances of these classes for the type constructor *Maybe*. Keep in mind that a *Maybe* result resembles the list of successes method, with the difference that we return at most one succesful result.

```
instance Applicative Maybe where
  Just f <*> Just a = Just (f a)
  _ <*> _ = Nothing
  pure a  = Just a
instance Alternatve Maybe where
  l@(Just a) <|> _ = l
  _ <|> r          = r
  fail             = Nothing
```

(see other side for exercise 6)

6. **Maximal Segment Sum (2)**

(a) Complete the following definition, for the function which computes all consecutive sequences of elements from a list:

$$\begin{aligned}
&\mathit{segs}\ \mathit{xs} &&= \mathit{fst}\ .\ \mathit{segsinits}\ \$\ \mathit{xs}\\
&\mathit{segsinits} :: [a] \to &&([[a]], [[a]])\\
&\mathit{segsinits}\ [\,] &&= ([[\,]], [[\,]])\\
&\mathit{segsinits}\ (x : \mathit{xs}) &&= \ldots
\end{aligned}$$

Example of the use of segs:

$$\begin{aligned}
&\gg \mathit{segs}\ [1, 2, -3, 4, 1]\\
&[[\,], [1], [4], [4, 1], [-3], [-3, 4], [-3, 4, 1], [2], [2, -3], [2, -3, 4], [2, -3, 4, 1],\\
&[1], [1, 2], [1, 2, -3], [1, 2, -3, 4], [1, 2, -3, 4, 1]]
\end{aligned}$$

(b) The specification of the *maximal segment sum problem* is:

$$\mathit{mss}\ \mathit{xs} = \mathit{foldr}\ \mathit{max}\ 0\ .\ \mathit{map}\ \mathit{sum}\ .\ \mathit{segs}\ \$\ \mathit{xs}$$

Note that in the evaluation of this specification a lot of common computation is going on. Give a solution for *mss* which does not construct the intermediate list structures, and which takes time linear in the length of the list.

The following hints may be useful:

- First modify your *segsinits* definition of the previous item into a definition of:

$$\begin{aligned}
&\mathbf{type}\ \mathit{Sum}\ a = a\\
&\mathit{segsinitis}' :: \mathit{Num}\ a \Rightarrow [a] \to ([(\mathit{Sum}\ a, [a])],\\
&\qquad\qquad\qquad\qquad\qquad\qquad ,[(\mathit{Sum}\ a, [a])]\\
&\qquad\qquad\qquad\qquad\qquad\qquad)
\end{aligned}$$

  which tuples each list occurring in the resulting lists with its sum.
- Apply the same method once more for the lists of segments and inits, and thus tuple those lists with the maximum value of the individual sums:

$$\begin{aligned}
&\mathbf{type}\ \mathit{Max}\ a = a\\
&\mathit{segsinitis}'' :: \mathit{Num}\ a \Rightarrow [a] \to ((\mathit{Max}\ a, [(\mathit{Sum}\ a, [a])])\\
&\qquad\qquad\qquad\qquad\qquad\qquad ,(\mathit{Max}\ a, [(\mathit{Sum}\ a, [a])])\\
&\qquad\qquad\qquad\qquad\qquad\qquad)
\end{aligned}$$

- Now remove the superfluous computations from the program.
- Keep in mind that $\mathit{sum}\ (x : \mathit{xs}) \equiv x + \mathit{sum}\ \mathit{xs}$
- Realise that $(a + b)\ `\mathit{max}`\ (a + c) = a + (b\ `\mathit{max}`\ c)$

You do not have to give the intermediate results. The hints are just there to help you to derive the solution.

$$\mathit{mss} = \mathit{foldr}\ \mathit{max}\ 0\ .\ \mathit{map}\ \mathit{sum}\ .\ \mathit{fst}\ .\ \mathit{segsinits}$$

We start out by completing the *segsinits* function, which tuples the computation of the segments themselves with the initial segments of the tails.

$$\begin{aligned}
&\mathit{segsinits}\ [\,] &&= ([[\,]], [[\,]])\\
&\mathit{segsinits}\ (x : \mathit{xs}) &&= \mathbf{let}\ (\mathit{segs}, \mathit{inits}) = \mathit{segsinits}\ \mathit{xs}\\
&&&\qquad\quad \mathit{newinits} = [\,] : \mathit{map}\ (x{:})\ \mathit{inits}\\
&&&\quad\ \mathbf{in}\ \ (\mathit{segs} +\!\!+ \mathit{newinits}, \mathit{newinits})
\end{aligned}$$

Following the hint we now change this function so it tuples each computed segment and each initial part with its sum:

$$\begin{aligned}
&\mathit{segsinits}' :: \mathit{Num}\ a \Rightarrow [a] \to ([(a, [a])], [(a, [a])])\\
&\mathit{segsinits}'\ [\,] &&= ([(0, [\,])], [(0, [\,])])\\
&\mathit{segsinits}'\ (x : \mathit{xs}) &&= \mathbf{let}\ (\mathit{segs}, \mathit{inits}) = \mathit{segsinits}'\ \mathit{xs}\\
&&&\qquad\quad \mathit{newinits} = (0, [\,]) : \mathit{map}\ (\lambda(s, \mathit{is}) \to (x + s, x : \mathit{is}))\ \mathit{inits}\\
&&&\quad\ \mathbf{in}\ \ (\mathit{segs} +\!\!+ \mathit{newinits}, \mathit{newinits})
\end{aligned}$$

Since the lists are now tupled with their sums we do not have to compute the sums anymore in *mss*, but can take just the first element from each tuple:

$$mss' = foldr\ max\ 0\ .\ map\ fst\ .\ fst\ .\ segsinits'$$

Now note that the lists are not needed for the final result anymore, so we can remove the code which computes them.

$$\begin{aligned}
&segsinits'' :: Num\ a \Rightarrow [\,a\,] \to ([\,a\,], [\,a\,]) \\
&segsinits''\ [\,] \qquad\quad = ([0], [0]) \\
&segsinits''\ (x : xs) = \mathbf{let}\ (segs, inits) = segsinits''\ xs \\
&\qquad\qquad\qquad\qquad\quad newinits = 0 : map\ (+x)\ inits \\
&\qquad\qquad\qquad\qquad\ \mathbf{in}\ \ (segs +\!\!+ newinits, newinits)
\end{aligned}$$

$$mss'' = foldr\ max\ 0\ .\ fst\ .\ segsinits''$$

We apply the same method once more, and now tuple each list with the maximal value of the list

$$\begin{aligned}
&segsinits''' :: (Num\ a, Ord\ a) \Rightarrow [\,a\,] \to ((a, [\,a\,]), (a, [\,a\,])) \\
&segsinits'''\ [\,] \qquad\quad = ((0, [0]), (0, [0])) \\
&segsinits'''\ (x : xs) = \mathbf{let}\ ((ms, segs), (mi, inits)) = segsinits'''\ xs \\
&\qquad\qquad\qquad\qquad\quad newinits = ((mi + x)\ `max`\ 0, 0 : map\ (+x)\ inits) \\
&\qquad\qquad\qquad\qquad\ \mathbf{in}\ \ ((ms\ `max`\ fst\ newinits, segs +\!\!+ snd\ newinits), newinits)
\end{aligned}$$

$$mss''' = fst\ .\ fst\ .\ segsinits'''$$

Now we also the computations of the lists of maxima themselves has become superfluous:

$$\begin{aligned}
&segsinits'''' :: (Num\ a, Ord\ a) \Rightarrow [\,a\,] \to (a, a) \\
&segsinits''''\ [\,] \qquad\quad = (0, 0) \\
&segsinits''''\ (x : xs) = \mathbf{let}\ (ms, mi) = segsinits''''\ xs \\
&\qquad\qquad\qquad\qquad\quad newinitsmax = (mi + x)\ `max`\ 0 \\
&\qquad\qquad\qquad\qquad\ \mathbf{in}\ \ (ms\ `max`\ newinitsmax, newinitsmax)
\end{aligned}$$

$$mss'''' = fst\ .\ segsinits''''$$

$$mss''''' = fst\ .\ foldr\ (\lambda x\ (ms, mi) \to \mathbf{let}\ ni = mi + x\ `max`\ 0\ \mathbf{in}\ (ms\ `max`\ ni, ni))\ (0, 0)$$

Finally we notice that we initialise the maxima with 0 already, so we simplify:

$$mss'''''' = fst\ .\ foldr\ (\lambda x\ (ms, mi) \to \mathbf{let}\ ni = mi + x\ \mathbf{in}\ (ms\ `max`\ ni, ni))\ (0, 0)$$