FP 2010-2011, Tussentoets
2010, Oct 4 , 8.30-10.30, EDUC Gamma

Only hand in the sheet with the answers, while showing your legitimation. Don't forget to fill out your name! Do not forget necessary parentheses!

1 The function *intersperse*

Write a function $intersperse :: a \to [a] \to [a]$, which places its first argument between the elements of its second argument; i.e. `intersperse 'a' "xyz"` should return `"xayaz"`. (2 points)

There are quite a few possible solutions. The most straightforward one is e.g.:

$intersperse\ a\ (x : y : ys) = x : a : intersperse\ a\ (y : ys)$
$intersperse\ \_\ xs \qquad\quad = xs$

The second case takes care of the empty xs and an xs od length 1. Unfortunately, as I explained in the last lecture, this function is not as productive as it could be.

Alternative solutions are:

$intersperse\ a\ xs = tail\ (\ foldr\ (\lambda x\ r \to a : x : r)\ [\,]\ xs)$
$intersperse\ a \quad = tail\ .\ foldr\ (\lambda x\ r \to a : x : r)\ [\,]$
$intersperse\ a \quad = tail\ .\ foldr\ (\lambda x \to (a:)\ .\ (x:))\ [\,]$
$intersperse\ a \quad = tail\ .\ foldr\ ((a:)\ .\ )\ .\ (:))\qquad [\,]$
$intersperse\ a \quad = tail\ .\ concat\ .\ map\ (\lambda x \to [a, x])$

2 Type inference

1. What is the type of *map map* (1 point)
2. Explain in at most 40 words why *foldl map* is not well-typed (1 point).

Let us assume that the first map has type: $(a \to b) \to [a] \to [b]$ and the second one $(c \to d) \to [c] \to [d]$, which has to match with (a-¿ b). Hence we infer that $a = c \to d$ and b = $[c] \to [d]$; we conclude that the result type is thus $[a] \to [b]$ with the above substitution: $[[c \to d] \to [[c] \to [d]]$.

We know that $foldl :: (b \to a \to b) \to b \to [a] \to b$, and that $map :: (c \to d) \to [c] \to [d]$ which has to line up with the $b \to a \to b$. We conclude both $b = c \to d$ and $b = [d]$, whch cannot both be the case.

3 The function *foldl*

1. What is the type of the expression *foldl* (*flip* (:)) [ ], where *flip* $f\ x\ y = f\ y\ x$ (1 point). Hint: first write down the types of (:), *foldl* and *flip* on your piece of scrap paper.
2. Give a definition of the function $reverse :: [a] \to [a]$ which uses an *accumulating parameter* in order to avoid inefficiencies.

$(:) \quad :: \quad a \quad \to [a] \to [a]$
$flip \ :: (\ a \quad \to b \quad \to c) \to (b \to a \to c)$
$flip\ (:) :: [a] \to a \quad \to [a]$
$foldl :: (\ b \quad \to a \quad \to b) \to b \to [a] \to b$
$foldl\ (flip\ (:)\ [\,] :: \qquad\qquad\qquad [a] \to [a]$

Actually this expression is equivalent to the function *reverse*:

$reverse\ xs = reverse'\ xs\ [\,]$
  where $reverse'\ [\,] \qquad r = r$
  $reverse'\ (x : xs)\ r = reverse'\ xs\ (x : r)$

which you can get by substituting the arguments (*flip* (:) and [ ] in the definition of *foldl*.

Many, many of you have written something like:

$reverse\ [\,] = [\,]$
$reverse\ (x : xs) = reverse\ xs : x$

This does not use an accumulating parameter, besides that you cannot use : to put an extra element at the end of the list (use $reverse\ xs +\!\!+ [a]$ instead), and the parentheses are missing, etc.

4 Input and output

Write a function $ask :: [String] \to IO\ [String]$ which takes a list of questions (each a *String*), poses those question on the terminal, reads an answer (assume 'y' or 'n' as the first character in the answer line) and *return*-s only those questions to which a positive answer was given (2 points).

$ask\ [\,] = return\ [\,]$
$ask\ (q : qs) = \mathbf{do}\ putStrln\ q$
$\qquad\qquad (a{:}\ \_) \leftarrow getLine$
$\qquad\qquad as \leftarrow ask\ qs$
$\qquad\qquad \mathbf{if}\ a \equiv \texttt{'y'}\ \mathbf{then}\ return\ (q : as)\ \mathbf{else}\ return\ as$

or

$return\ (\mathbf{if}\ a \equiv \texttt{'y'}\ \mathbf{then}\ (q{:})\ \mathbf{else}\ id)\ as)$

5 List based functions

1. Write the function $subs :: [a] \to [[a]]$ which returns all sublists of the argument list (1 point).
2. Change this function into a function *subs_asc* such that it only return those lists which are ascending (stijgend) (1 point). Do not use the function *filter* since this may be very inefficiënt; make the filtering part of the generation process.

$subs\ [\,] = [[\,]]$
$subs\ (x : xs) = map\ (x{:})\ subsxs +\!\!+ subsxs$
$\qquad\qquad \mathbf{where}\ subsxs = subs\ xs$

$subs\_asc\ [\,] = [[\,]]$
$subs\_asc\ (x : xs) = [x : l \mid l \leftarrow subsxs, null\ l \vee x \leqslant head\ l] +\!\!+ subsxs$
$\qquad\qquad\qquad \mathbf{where}\ subsxs = subs\_asc\ xs$