FP 2010-2011, Proeftussentoets
2010, Sept 29

1 *foldl*

1. What is the type of the function *foldl* (1 point)
2. Give the definition of the function *foldl* (1 point)

Het type van *foldl* is $(b \to a \to b) \to b \to [a] \to b$. We beginnen helemaal links met een waarde van type b en verwerken dan alle elementen uit de lijst van type $[a]$ stuk voor stuk met de functie van type $b \to a \to b$. Het uiteindelijke resulaattype is b.

De definitie van *foldl* staat in het dictaat:

$$\begin{aligned} &\mathit{foldl}\ \mathit{op}\ e\ [\,] &&= e \\ &\mathit{foldl}\ \mathit{op}\ e\ (x : xs) &&= \mathit{foldl}\ \mathit{op}\ (e\ `\mathit{op}`\ x)\ xs \end{aligned}$$

2 Tupling *dropWhile* and *takeWhile*

(2 points)

The function $\mathit{dropWhile} :: (a \to \mathit{Bool}) \to [a] \to [a]$ drops the longest prefix of a list in which all elements obey the passed predicate, whereas $\mathit{takeWhile} :: (a \to \mathit{Bool}) \to [a] \to [a]$ returns the part that is not returned by *dropWhile*. Write a function $\mathit{takeAndDropWhile} :: (a \to \mathit{Bool}) \to [a] \to ([a], [a])$ which computes the combined result of this function and applies the predicate only once to elements of the list.

oplossing:

$$\begin{aligned} &\mathit{takeAndDropWhile}\ p\ [\,] = ([\,], [\,]) \\ &\mathit{takeAndDropWhile}\ p\ t@(x : xs) \mid p\ x &&= \mathbf{let}\ (t, f) = \mathit{takeAndDropWhile}\ p\ xs \\ &&&\quad \mathbf{in}\ (x : t, f) \\ &\qquad\qquad \mid \mathit{otherwise} &&= ([\,], t) \end{aligned}$$

3 Type inference

What is the type of the following expressions:

1. *foldr map* (1 point)
2. *map foldr* (1 point)

We geven eerst eens even verschillende namen aan de verschillende polymorfe variabelen:

$$\begin{aligned} &\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b \\ &\mathit{map}\ :: (c \to d) \to [c] \to [d] \end{aligned}$$

We voorzien dit eerst eens van wat extra haakjes:

$$\begin{aligned} &\mathit{foldr} :: (a \to (b \to b)) \to b \to [a] \to b \\ &\mathit{map}\ :: (c \to d\ \ \ \ \ \ ) \to ([c] \to [d]) \end{aligned}$$

Als we nu de functie *map* meegeven op de eerste parameterplaats van de *foldr* dan zien we dat $a \to (b \to b)$ hetzelfde type moet zijn als $(c \to d) \to ([c] \to [d])$. Dit kan als we kiezen $a \equiv c \to d$ en $b \to b \equiv [c] \to [d]$. Uit dit laatste concluderen we dat $b \equiv [c]$ en ook $b \equiv [d]$ moeten gelden. Hieruit leiden we af dat dus $c \equiv d$ moet gelden. Nu gebruiken we deze informatie om te kijken wat we aanextra informatie over het type $b \to [a] \to b$ te weten zijn gekomen. We verwerken eerst het feit dat $c \equiv d$ in onze vergelijkingen, en krijgen dan $a \equiv c \to c$ en $b \equiv [c]$. Hiermee gewapend kunnen we nu afleiden dat het gezochte resultaattype $[c] \to [c \to c] \to [c]$ is.

Nu het tweede geval. We zetten eerst de haakjes goed in *foldr* zodat we niet in de war raken:

$foldr :: (a \to (b \to b)) \to (b \to [a] \to b)$
$map\ :: (c \to d\ \ \ \ \ \ \ ) \to ([c] \to [d])$

Hieruit leiden we af dat $c = a \to b \to b$ en dat $d = b \to [a] \to b]$. Substitueren we dit in in het type $[c] \to [d]$ dan krijgen we $[a \to b \to b] \to [b \to [a] \to b]$. Iets ingewikkelder resultaat, maar een iets eenvoudiger afleiding.

Let op dat als je zoiets doet voor een expressie als *map map* dat je dan voor beide instanties van *map* dus nieuwe variabelen kiest:

$map1 :: (a \to b) \to [a] \to [b]$
$map2 :: (c \to d) \to [c] \to [d]$

En nu lezen we *map map* als *map1 map2*, en dus moet gelden $a \equiv c \to d$ en $b \equiv [c] \to [d]$, en is dus het resultaattype $[c \to d] \to [[c] \to [d]]$.

4 IO

(2 points)

Write a function $table :: IO\ ()$ which reads a number (say 4) from the terminal (you may use the function $read : String \to Int$), and which prints the lines `"1*4 = 4"` upto `"10*4=40"`.

module *TestIO* **where**

$main = \mathbf{do}\ l \leftarrow getLine$
$\qquad sequence\_\ (map\ (singleLine\ (read\ l))\ [1\,..\,10])$
$singleLine\ i\ j = putStrLn\ (show\ j\ ++\texttt{" * "}\ ++show\ i\ ++\texttt{" = "}\ ++show\ (i * j))$

5 List based functions

1. Write the function $splits :: [a] \to [(a, [a])]$ which splits a list into all possible combinations of a single element and the rest of the elements. (1 point)
2. Use this fuction to write a function *permute* which returns all possible permutations of a list. (1 point)

Voor oplossing zie slides van afgelopen maandag.