FP 2010-2011, Eindtoets
2010, Nov 8 , 8.30-10.30,Jaarbeurs

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

1 The functies *sum* and *and* (1 point)

The functions *sum* and *and* can both be expressed using either *foldr*, or *foldl* or *foldl'*. Give the appropriatate definitions and explain your choice in a few words.

Possible answer:

Since the operator + is strict in both arguments we choose the function *foldl'* to combine all the elements; this function evaluates the final sum in its accumulating argument, and thus prevents the build-up of large expressions in memory. Furthermore we avoid the overhead associated with lazy evaluation, hence: $sum = foldl'\ (+)\ 0$. For the *and*-case the situation is different, since $\wedge$ not strict in its second argument: once we have evaluated a left argument and have discovered that its result equals *False* we can skip the rest of the evaluation, and hence we choose *foldr*: $and = foldr\ (\wedge)\ True$.

2 Types

1. What is the type of $last\ .\ repeat$? (0.5 point)
2. What is the type of $foldr\ map$? (0.5 point)

Possible answer:

The function *repeat* has type $a \to [a]$ and *last* has type $[a] \to a$. Hence the composition has type $a \to a$. This despite the fact that this expression will never terminate.

Furthermore we have $foldr\ map :: [a] \to [a \to a] \to [a]$.

3 Inductiebewijs (2 points)

Proof with induction that $sum\ (map\ ((k+)\ .\ (n*))\ xs) = k * len\ xs + n * sum\ xs$

Possible answer:

In case *xs* equals the empty list the results follows immediately. In the non-empty case we proceed as follows:

$$
\begin{array}{ll}
sum\ (map\ ((k+)\ .\ (n*))\ (x : xs)) & k * length\ (x : xs) + n * sum\ (x : xs) \\
= definition\ map\ and\ . & = definition\ sum\ and\ length \\
sum\ (k + n * x : map\ ((k+)\ .\ (n*))\ xs) & k + length\ xs + n * (x + sum\ xs) \\
= def\ sum & = *is\ associative, +commutative, +is\ associative \\
k + n * x + sum\ (map\ ((k +. (n*))\ xs) & k + n * x + (length\ xs + sum\ xs) \\
= use\ induction\ hypothesis & \\
k + n * x + (length\ xs + sum\ xs) &
\end{array}
$$

4 Unificatie

Give the code which completes the function *unify* from the Prolog interpreter at the ... positions (1 point) and define the function >=> (1 point):

```
unify ::      Term → Term → Bdgs → Maybe Bdgs
unify t u e
   = uni (lookUp e t) (lookUp e u) e
  where uni t@(Var x) y      e = bind t y e
```

```
uni x t@(Var y)       e = bind t x e
uni (Con x) (Con y) e = ...
uni (Fun x xs) (Fun y ys) e
  | x ≡ y ∧ length xs ≡ length ys
                = foldr (>=>) return ...
  | otherwise = Nothing
uni _ _ _       = Nothing
```

Possible answer:

The constants only uniify if the have the same value. If this is the case this does not teach us anything new about the environment, so we get *uni* (*Con x*) (*Con y*) = **if** $x \equiv y$ *the Just e* **else** *Nothing*.

In the case of the function symbol we require that the subtrees (the arguments) can be unified pairwise: *foldr* (>=>) (*zipWith unify*) \$ *e*. Since each unification has to start with the environment resulting from the previous one, provided that onne succeeded, we have:

```
(>=>) :: (Monad m) ⇒ (a → m b) → (b → m c) → a → m c
  -- in our intepreter a, b anc c are all Bdgs
(a2mb >=> b2mc) a = do b ← a2mb a
                       b2mc b
```

5 River crossing

This code gives a default definition for the *Problem* class:

```
class Eq c ⇒ Problem c where
  isSolution  :: c → Bool
  successors  :: c → [c]
  successors' :: c → [c]   -- transitive closure of 'successors'
  successors' initial = generate [] [initial] where
    generate done [] = []
    generate done (x : xs) = let done' = x : done
                                 succX = successors x \\ done'
                             in x : generate done' (xs ++succX)
```

Give a better definition of the function *generate* does not generate duplicate solutions (1 point), which does not use concatenations (0.5 point) and performs a depth-first search (0.5 point).

Possible answer:

A problem with the solution given is that it may produce duplicate solutions. Although we never add elements to the worklist which have already been used for the computation of their successors, we may still add elements which have been added to the worklist already. As an example consider the following successor relation $\{1 \to [2,3], 2 \to [4], 3 \to [4]\}$. After we have dealt with 1 the worklist contains $[2,3]$. We now inspect 2 and add 4, so the worklist becomes $[3,4]$. Inspecting the 3 adds another 4.

Since we want to avoid the repeated concatenation of lists, and especially at the end of the worklist, we mainatin a list of lists as the worklist, and add new sublists to the beginning of this list:

```
generate done []              = []
generate done ([] : xs)       = generate done xs
generate done ((x : xs) : ys) = if x ∈ done
                                then generate done (                xs : ys)
                                else x : generate done (successors x : xs : ys)
```

6 Lijst gebaseerde functies

1. Write the function *perms* :: $[a] \to [[a]]$ which returns all permutations of the argument list. (1 point)

2. Change this function into a function $perms_smooth :: Num\ a \Rightarrow a \rightarrow [a] \rightarrow [[a]]$ such that only those permutations are returned for which all consecutive elements do not differ more than the value given by the first parameter. (1 point)

   $perms_smooth\ 1\ [3, 4, 5] = [[3, 4, 5], [5, 4, 3]]$

Possible answer:

The definition of *perms* which provides the easiest solution to the second question is the one based on the function *splits*, which splits a list in a list of individual elements, each paired with the remaining elements:

$perms\ [\,] = [[\,]]$

$perms\ xs = [\,y : p \mid (y, ys) \leftarrow splits\ xs, p \leftarrow perms\ ys\,]$

Now the filtering gets very easy:

$perms_smooth\ _\ [\,] = [[\,]]$

$perms_smotth\ d\ xs = [\,y : p \mid (y, ys) \leftarrow splits\ xs, p \leftarrow perms\ ys, null\ p \vee abs\ (head\ p - y) \leqslant d\,]$