Haskell Live

[06] Aufgabenblatt 3 & CityMaut

Bong Min Kim
e0327177@student.tuwien.ac.at

Christoph Spörk
christoph.spoerk@inode.at

Florian Hassanen
florian.hassanen@googlemail.com

Bernhard Urban
lewurm@gmail.com

12. November 2010

Aufgabenblatt 3

Mögliche Lösungswege für die Aufgaben 3. Übungsblatts:

```
import Data.Char
import Data.List
```

dreiNplusEins

```
dreiNplusEins :: Integer -> [Integer]
```

```
dreiNplusEins 1 = [1]
dreiNplusEins x = x : (dreiNplusEins next)
  where
    next = if x `mod` 2 ≡ 0
      then x `div` 2
      else (x ∗ 3) + 1
```

maxZyklus

```
  -- wrapper for correct type
mylen :: [a] → Integer
mylen = fromIntegral ∘ length
```

```
type UntereGrenze = Integer
type ObereGrenze = Integer
type MaxZykLaenge = Integer
```

```
maxZyklus :: UntereGrenze → ObereGrenze → (UntereGrenze, ObereGrenze, MaxZykLaenge)
maxZyklus m n
  | m > n = (m, n, 0)
  | otherwise = (m, n, maxLength)
  where
    maxLength = maximum [length | i ← [m . . n], let length = mylen (dreiNplusEins i)]
```

```
  -- another solution (which can exceed the 4000 limit by balancing the search tree)
maxZyklusBalanced :: UntereGrenze → ObereGrenze → (UntereGrenze, ObereGrenze, MaxZykLaenge)
maxZyklusBalanced u o =
(u,
```

```
    o,
    max 0 (zyklenStep u o)
  )

zyklenStep :: UntereGrenze -> ObereGrenze -> MaxZykLaenge
zyklenStep u o
 | u == o = mylen (dreiNplusEins u)
 | u > o = -1
 | otherwise = max (zyklenStep u mid) (zyklenStep (mid + 1) o)   -- "balancing" happens here
   where mid = u + ((o - u) `div` 2)
```

anzNachbarn

```
anzNachbarn :: [[Bool]] -> (Integer, Integer) -> Integer
anzNachbarn matrix (m, n)
    | not ((m, n) `inMatrix` matrix) = -1
    | otherwise = anzahl
  where (hoehe, breite) = sizeOfMatrix matrix
        offsets = [(z, s) | z <- [-1, 0, 1],    -- all offsets
                  s <- [-1, 0, 1],
                  not (z == 0 && s == 0)]
        koords = [(z, s) | (zoff, soff) <- offsets,    -- add offsets to coords
                  let z = m + zoff,
                  let s = n + soff,
                  (z, s) `inMatrix` matrix]
        trues  = [(z, s) | (z, s) <- koords,    -- filter for "true" neighbours
                  matrix !! (fromIntegral z) !! (fromIntegral s)]
        anzahl = fromIntegral (length (trues))   -- and count them

inMatrix :: (Integer, Integer) -> [[Bool]] -> Bool
```

```
inMatrix (a, b) matrix =
  a ⩾ 0 ∧ a < hoehe ∧ b ⩾ 0 ∧ b < breite
  where (hoehe, breite) = sizeOfMatrix matrix
```

```
sizeOfMatrix :: [[Bool]] → (Integer, Integer)
sizeOfMatrix matrix@(ersteZeile: _) = (mylen (matrix), mylen (ersteZeile))
```

transform

```
transform :: [[Bool]] → [[Integer]]
transform matrix =
  [[anzNachbarn matrix (z, s)
      | s ← [0 . . (breite − 1)]
     ]
   | z ← [0 . . (hoehe − 1)]
  ]
    where (hoehe, breite) = sizeOfMatrix matrix
```

CityMaut

```
type Bezirk = Char
type AnzBezirke = Integer
type Route = (Bezirk, Bezirk)
type Weg = [Bezirk]
type CityMap = (AnzBezirke, [Route])

angabeCity = (6, [
  ('B', 'C'),
  ('A', 'B'),
  ('C', 'A'),
  ('D', 'C'),
  ('D', 'E'),
  ('E', 'F'),
  ('F', 'C')])
```

a function of the type “BezirkMapping” shall return all neighbouring bezirks to a given bezirk

```
type BezirkMapping = (Bezirk → [Bezirk])
```

bezirkMapping generates a “BezirkMapping” based on the route information of a CityMap

```
bezirkMapping :: [Route] → BezirkMapping
bezirkMapping [] _ = []
bezirkMapping ((from, to) : rest) x
  | x ≡ from = to : recursion
  | x ≡ to = from : recursion
  | otherwise = recursion
  where recursion = bezirkMapping rest x

paths1 :: BezirkMapping → Bezirk → Bezirk → [Weg]
paths1 mapping v1 v2 = paths1_ mapping v1 v2 []
```

```
type TabuList = [Bezirk]
paths1_ :: BezirkMapping → Bezirk → Bezirk → TabuList → [Weg]
  -- explicit tabulist
paths1_ mapping start dest tabulist
   | (start ≡ dest) = [[dest]]
   | ¬ tabu      = [(start : tailpath) | next ← neighbours,
                    tailpath ← (paths1_ mapping next dest (start : tabulist))]
   | otherwise = []
                      -- "\\" means "without" (see Data.List for definition)
  where neighbours = (mapping start) \\ [start]
     tabu = (elem start tabulist)


bezirkMappingMinus :: BezirkMapping → Bezirk → BezirkMapping
  -- hint: f(g(x)) = (f.g)(x)
bezirkMappingMinus mapping bezirkToDelete = (filter (bezirkToDelete ≢)) ∘ mapping
  -- this is equivalent:
bezirkMappingMinus′ mapping bezirkToDelete askedBezirk = filter (bezirkToDelete ≢) (mapping askedBezirk)


  -- define operator \\\
(\\\) = bezirkMappingMinus


paths2 :: BezirkMapping → Bezirk → Bezirk → [Weg]
  -- "implicit tabulist" via "\\\" operator
paths2 mapping start dest
   | (start ≡ dest)      = [[dest]]
   | next_bezirks ≡ [] = []
   | otherwise            = [start : tailpath | next ← next_bezirks,
                              tailpath ← paths2 (mapping \\\ start) next dest]
  where next_bezirks = (mapping \\\ start) start
```

```haskell
paths3 :: BezirkMapping -> Bezirk -> Bezirk -> [Weg]
paths3 mapping v1 v2 = paths3_ mapping [v1] v2 []

paths3_ :: BezirkMapping -> [Bezirk] -> Bezirk -> [Bezirk] -> [Weg]
  -- without list comprehension
paths3_ _ [] _ _ = []
paths3_ mapping (current : neighbours) dest partial
   | (dest == current) = [partial ++ [current]] ++ recursion_excl_current
   | not tabu          = recursion_incl_current ++ recursion_excl_current
   | otherwise         = recursion_excl_current
  where recursion_excl_current = paths3_ mapping neighbours dest partial
        recursion_incl_current = paths3_ mapping currents_neighbours dest (partial ++ [current])
        currents_neighbours = (mapping current) \\ [current]
        tabu                = elem current partial

angabeMapping = bezirkMapping $ snd angabeCity

allRoutes :: BezirkMapping -> Bezirk -> Bezirk -> [Weg]
allRoutes = paths1
  -- allRoutes = paths2

  -- simple testsuite
equivTest = [(xy, res1 == res2, res2 == res3, res3 == res1)
   | x <- ['A' .. 'F'],
     y <- ['A' .. 'F'],
     let p1 = paths1 angabeMapping,
     let p2 = paths2 angabeMapping,
     let p3 = paths3 angabeMapping,
     let res1 = p1 x y,
```

```
    let res2 = p2 x y,
    let res3 = p3 x y,
    let xy = (x, y)
  ]
equivTestSuccess = all (λ(_, t1, t2, t3) → (t1 ∧ t2 ∧ t3)) equivTest

nadeloehrs :: BezirkMapping → Bezirk → Bezirk → Maybe [Bezirk]
  -- returns Nothing on invalid parameters
  -- returns Just <list> where list is the [potentially empty] list of nadeloehrs
nadeloehrs mapping start end
   | no_route = Nothing
   | otherwise = Just (intersectAll routes)
  where routes = allRoutes mapping start end
    no_route = routes ≡ [] ∨ start ≡ end

intersectAll = intersectAll1

  -- using foldl' (assumption: all bezirk names are upper case letters)
  -- (init.tail) strips the first and last element of a list
intersectAll1 :: [Weg] → [Bezirk]
intersectAll1 routes = foldl' (intersect) ['A' .. 'Z'] (map (init ∘ tail) routes)

  -- using list comprehension (same assumption as before)
intersectAll2 :: [Weg] → [Bezirk]
intersectAll2 routes = [n | n ← ['A' .. 'Z'], and (map ((elem n) ∘ init ∘ tail) routes)]

  -- using simple recurions (do note: no assumptions on names of bezirks needed)
intersectAll3 :: [Weg] → [Bezirk]
intersectAll3 [route] = route   -- try to figure out, why this case is needed
intersectAll3 []      = []   -- and this one as well
intersectAll3 (route : routes) = intersect route (intersectAll3 routes)
```