Fakultät für Informatik und Mathematik
Prof. Dr. Martin Griebl
Summer semester 2023

Exercises for Lecture: Functional Programming

Exercise Sheet 9 (Homomorphisms (2), Optimization Rules)

Problem 1 (Skeleton for List Homomorphisms)

`hom` is a skeleton for list homomorphisms, which is parametrized with a function `f`, the operator `op` in the image set (of the homomorphism) together with its neutral element `e`.

```
hom :: (a->b) -> (b->b->b) -> b -> [a] -> b
hom f op e = foldr op e . map f
```

Define the following functions using `hom`:

(a) exchanging neighboring elements: `twistH :: [a] -> [a]`
`twistH "ABCDEF"` ⇝ `"BADCFE"`
`twistH "ABCDEFG"` ⇝ `"BADCFEG"`

(b) transposition of a two-dimensional list:

```
transpose          :: [[a]] -> [[a]]
transpose []       = []
transpose ([]:xss) = transpose xss
transpose xss      = [x | (x:_) <- xss] : transpose [xs | (_:xs) <- xss]
```

Examples:

- `transpose [[1,2,3],[4,5,6]]` ⇝ `[[1,4],[2,5],[3,6]]`
- `transpose [[1,2,3],[4,5],[],[6,7,8,9]]` ⇝ `[[1,4,6],[2,5,7],[3,8],[9]]`

Hint:

- First, think about the set M and the neutral element `e` in the target monoid $(M, \texttt{op}, \texttt{e})$ of `transpose`.
- Now, try to find a suitable definition for `op`. The following might help you (focus on the last two lines):

```
   transpose [[1,2,3], [4,5], [], [6,7,8,9]]
== transpose ([[1,2,3], [4,5]]  ++   [[], [6,7,8,9]])
== transpose [[1,2,3], [4,5]]  `op`  transpose [[], [6,7,8,9]]
== [[1,4], [2,5], [3]]         `op`  [[6], [7], [8], [9]]
== [[1,4,6],[2,5,7],[3,8],[9]]
```

- Finally, find a suitable mapping operation `f` for the `hom` skeleton.

Problem 2 (List Homomorphisms)

Are the following functions list homomorphisms? Give either a proof sketch (if the respective function is a list homomorphism) or a counter example (if it is not).

(a) `mss :: [Integer] -> Integer`
(maximum segment sum, cf. lecture slides 4–87 and following), where

```
mss = let segs = concat . map inits . tails
      in maximum . map sum . segs
```

(b) `s :: [a] -> [a]` with `s = scanl g e` where `g :: a -> a -> a` is an associative function and `e :: a` is neutral w.r.t. `g`.

Problem 3 (Optimization Rules)

Laws for list functions and other equalities can be used directly by compilers in purely functional languages as optimization rules for program transformations. In Stud.IP, you can find the file `RulesDemo.hs`, which demonstrates how to apply an optimization rule in GHC. Since many optimizations are only turned on by GHC when compiling programs (not in GHCi), the program has to be compiled using

```
ghc -o rulesdemo RulesDemo.hs
```

and run with `./rulesdemo`.

The file `BTreeOpt.hs` contains the data type `BTree a` known from an earlier exercise sheet together with functions `mapBTree` and `foldBTree`. Compile using `ghc -o btreeopt BTreeOpt.hs` and run it with `./btreeopt` *N*, where *N* denotes the size of the computation performed (cf. function `bigComputation`). Values near 25 should lead to a run time of a few seconds.

(a) Add optimization rules in `BTreeOpt.hs` for `mapBTree`/`foldBTree`, which speed up the computation `bigComputation` by avoiding temporary data structures.

(b) (more difficult:) Write a function to create a `BTree` in a "generic way", e.g.,

```
unfoldBTree :: (b -> Either a (b,a,b)) -> b -> BTree a
```

`unfoldBTree f x` starts a recursion from `x` to build a tree. `f x` returns either a value `Left v`, when a leaf is to be created, or `Right (x1,v,x2)` when an inner node shall be created. `x1` and `x2` are then used in recursive calls to generate the left and right sub tree, respectively. (`unfoldBTree` is the analogue of `unfoldr` from module `Data.List`.)

Define `bigTree` using `unfoldBTree` and write a rule that permits the fusion of `foldBTree` and `unfoldBTree` (eliminating all intermediate tree data structures).

Due date: Tuesday, June 20, 2023 at 16:00

Upload your Haskell source codes (`*.hs`) to the `Submissions` folder for this exercise in Stud.IP.