Programmierung

04. Übungsblatt

Zeitraum: 11. – 15. Mai 2015

Übungsverlegungen zu Himmelfahrt (14.05.):

- *1. DS verlegt auf Fr., 15.05., 3. DS, APB/E023*
- *2. DS verlegt auf Di., 12.05., 1. DS, APB/E009*
- *4. DS verlegt auf Mi., 13.05., 1. DS, APB/E010*

Übung 1

Wir stellen Binärbäume ohne Knotenbeschriftungen durch den Typ

```
data Tree = Leaf | Branch Tree Tree
```

dar. Schreiben Sie eine Funktion `mkBalanced :: Int -> [Tree]`, welche alle balancierten Binärbäume einer vorgegebenen Höhe berechnet.

Zur Erinnerung: Ein Binärbaum ist dann balanciert, wenn sich an jedem seiner Knoten die Höhen der beiden Kindbäume um höchstens 1 unterscheiden.

Übung 2 (AGS 12.1.40)

(a) Gegeben sei der polymorphe algebraische Datentyp

```
data Tree a = Node a (Tree a) (Tree a) | Leaf a
```

Geben Sie eine Funktion `path :: Tree a -> Tree Int` an, die die Beschriftung jedes Knotens durch die Anzahl der Knoten auf dem Pfad zum Wurzelknoten ersetzt. Dem Wurzelknoten wird also die Zahl 1 zugeordnet, seinen direkten Nachfolgern die Zahl 2, deren direkten Nachfolgern die Zahl 3, und so weiter.

(b) Geben Sie eine Funktion `nodes :: Tree a -> Tree [a]` an, die die Beschriftung jedes inneren Knotens eines Eingabebaums durch die (dreielementige) Liste der Beschriftungen an dem Knoten selbst, an seinem linken und an seinem rechten Nachfolger ersetzt. Ist der Knoten stattdessen ein Blatt, so soll die Liste nur aus seiner Beschriftung bestehen.

Übung 3

(a) Implementieren Sie die Funktion `f` aus Aufgabe 2 vom ersten Übungsblatt aufs neue. Nutzen Sie dazu Higher-Order-Funktionen aus der Prelude-Bibliothek!

(b) Schreiben Sie eine Funktion `pow :: Int -> (a -> a) -> a -> a`, so dass der Aufruf `pow n f a` der `n`-fachen Anwendung von `f` auf `x` entspricht.

(c) Schreiben Sie eine Funktion `pleat :: (a -> a -> b) -> [a] -> [b]`, so dass für eine Liste `xs = [x0, x1, x2, ...]` gilt, dass

```
pleat f xs = [f x0 x1, f x1 x2, f x2 x3, ...]  .
```

Übung 4 (AGS 12.1.19)

Gegeben sei der polymorphe Typ:

```
data Tree a = Leaf a | Branch a (Tree a) (Tree a)
```

(a) Schreiben Sie eine polymorphe Funktion `inorder` (einschließlich Typdefinition), die aus jedem Baum `t :: Tree a` jeweils eine Liste des Typs `[a]` aller Knotenbewertungen des Baumes `t` in der Reihenfolge linker Teilbaum, Wurzelbewertung, rechter Teilbaum erzeugt („in-order"–Durchlauf).

(b) Seien `f :: a -> b` eine beliebige Funktion, `t :: Tree a` ein Baum und `map` die Ihnen bekannte Haskell-Funktion. Schreiben Sie eine Funktion `mapTree` (einschließlich Typdefinition), so dass

```
map f (inorder t) == inorder (mapTree f t)  .
```

Zusatzaufgabe 1 (AGS 12.1.15)

(a) Programmieren Sie eine Funktion `filter2 :: [Int] -> [Int]`, die aus einer Liste von Zahlen diejenigen herausstreicht, welche kleiner als 0 sind.

(b) Programmieren Sie eine polymorphe Funktion `count :: [a] -> Int`, die eine Liste beliebigen Typs als Argument nimmt und die Anzahl der darin enthaltenen Elemente liefert.

(c) Programmieren Sie eine Funktion `check :: [Int] -> Bool`, die genau dann den Wahrheitswert `True` liefert, wenn die Eingabeliste mindestens 5 nicht-negative Zahlen enthält. Verwenden Sie dazu die Funktionen `filter2` und `count`.

(d) Gegeben sei der folgende polymorphe algebraische Datentyp:

```
data Tree a = Node (Tree a) (Tree a) | Leaf a
```

mit dessen Hilfe sich binäre Bäume darstellen lassen, die an den Blättern Werte des Typs `a` speichern. Geben Sie als Beispiel für ein Element des Typs `Tree [Int]` einen Baum mit genau 3 Blättern an; dabei sollen die Blätter unterschiedlich beschriftet sein.

(e) Programmieren Sie eine Funktion `trans :: Tree [Int] -> Tree Bool`, die ihren Eingabebaum transformiert, indem sie die Funktion `check` auf die jeweils an den Blättern gespeicherten Werte anwendet und ansonsten die Struktur des Eingabebaumes erhält.

Zusatzaufgabe 2 (AGS 12.1.38 ⋆)

(a) Der algebraische Datentyp `Lst` sei durch folgenden Haskell-Code gegeben.

```
data Lst a = Snoc Lst a | Nil deriving Show
```

In welcher Beziehung steht der Datentyp `Lst a` zu dem Typ `[a]`? Geben Sie Funktionen vom Typ `[a] -> Lst a` bzw. `Lst a -> [a]` an, welche diese Beziehung veranschaulichen.

(b) Welches Problem fällt Ihnen sowohl bei `[a]` als auch bei `Lst a` auf, wenn wir das Laufzeitverhalten beim Einfügen von Elementen am Anfang bzw. am Ende der Datenstruktur betrachten?

(c) Implementieren Sie nun mit den Listen von Haskell einen *Queue*-Datentyp. Eine Queue enthält eine Folge von Werten und bietet darüber hinaus Funktionen

- zum Prüfen ob die Queue leer ist:
  `isEmpty :: Queue a -> Bool`,
- zum (laufzeit-)effizienten Einfügen eines Elements ans *Ende* der Queue:
  `enqueue :: a -> Queue a -> Queue a`
- zum effizienten Zugriff auf das *Anfangs*element der Queue:
  `first :: Queue a -> a`
- zum Abspalten der Rest-Queue vom ersten Element:
  `rest :: Queue a -> Queue a`

Sie dürfen dabei die Funktion `reverse` zum Umkehren von Listen verwenden!