Christoph Lüth
Marcus Ermler
Christian Maeder
Martin Ring
Diedrich Wolter

5. Übungsblatt

Ausgabe: 21.11.12
Abgabe: 30.11.12

Rekursion ist ein mächtiges Werkzeug, und wie alle mächtigen Werkzeuge in den falschen Handen nicht ungefährlich. Im sicherheitskritischen Bereich wird Rekursion deshalb manchmal schlicht verboten; die MISRA Programmierrichtlinien für den Einsatz von C in sicherheitskritischen Systemen [1] fordern deshalb:

> **Rule 16.2 (Required) Functions shall not call themselves, either directly or indirectly.**
> This means that recursive function calls cannot be used in safety-related systems. Recursion carries with it in the danger of exceeding available stack space, which can be a serious error. Unless recursion is very tightly controlled, it is not possible to determine before execution what the worst-case stack usage could be.

Nun ist Haskell nicht C, aber das Argument gilt *mutatis mutandis* trotzdem. In Haskell bedeutet der Verzicht auf Rekursion statt dessen auf die zum größten Teil vordefinierten Funktionen höherer Ordnung (`map`, `fold`, `filter` und Freunde) zurückzugreifen, welche unter anderem die Gefahr versehentlicher Nicht-Termination minimieren. In diesem Übungsblatt soll dieses Vorgehen geübt werden, und deshalb gilt für alle zu implementierenden Funktionen wenn nicht *ausdrücklich* anderweitig angegeben:

*Es sind **keine rekursiven** Funktionsdefinitionen zugelassen.*

5.1 *Binäre Bäume* — *5 Punkte*

Es gibt viele Arten von Bäumen, in der Natur wie in der Informatik. Hier ist eine Spezies von binären Bäumen, die auch in den Blättern Daten trägt — und zwar von einem anderen Typ als in den Knoten:

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

- Geben Sie Definitionen für `foldT` und `mapT` für diesen Datentyp an; diese sind notwendigerweise rekursiv (als einzige Funktionen auf diesem Übungsblatt).

Mit dem so definierten `foldT` implementieren Sie folgende Funktionen:

- `height` berechnet die Höhe eines Baumes berechnen:

  ```
  height :: Tree a b→ Int
  ```

- `numLeaves` und `numNodes` berechnen die Anzahl der Blätter bzw. Knoten des Baumes:

  ```
  numLeaves :: Tree a b→ Int
  numNodes  :: Tree a b→ Int
  ```

- `preorder` gibt Blätter und Knoten in Präorder-Traversion zurück. Welche Signatur kann `preorder` haben?

5.2 *Vordefiniertes Neu Definiert* — *3 Punkte*

Implementieren Sie folgende bereits vordefinierte Funktionen auf Listen *nur mit* **foldr** *oder* **foldl**:

- take und drop für endliche Listen:

  ```
  take :: Int→ [a]→ [a]
  drop :: Int→ [a]→ [a]
  ```

- last gibt das letzte Element einer Liste zurück:

  ```
  last :: [a]→ a
  ```

Um die vordefinierten Funktionen zu verstecken und mit Ihren Definition zu überschreiben benutzen Sie die folgende magische Inkantation am Anfang Ihres Quellcodes:

```
import Prelude hiding (take,drop,last)
```

5.3 *Allerlei Nützliches* — *6 Punkte*

Definieren Sie folgende nützliche Funktionen auf Listen:

- updListWith p n as ersetzt in der Liste as alle Elemente a, für die p a gilt, durch n. Die Reihenfolge der Elemente soll gleich bleiben.

  ```
  updListWith :: (a→ Bool)→ a→ [a]→ [a]
  ```

- flattenPair macht aus einer Liste von Paaren eine Liste der Elemente:

  ```
  flattenPair :: [(a, a)]→ [a]
  ```

- deleteFindMin liefert aus einer Liste von vergleichbaren Elementen das Minimum, und den Rest der Liste:

  ```
  deleteFindMin :: Ord a⇒ [a] → (a, [a])
  ```

- avg berechnet den arithmetischen Mittelwert aus einer Liste von numerischen Werten:

  ```
  avg :: Fractional a⇒ [a]→ a
  ```

- averages „glättet“ eine Liste, indem es von jeweils drei benachbarten Werten den Mittelwert bildet:

  ```
  averages :: [Double]→ [Double]
  ```

  Sei $\mathsf{avg}[x_1, x_2, \ldots, x_n] = [y_1, \ldots, y_{n-2}]$ mit $n >= 2$, dann ist $y_i = \mathsf{avg}[x_i, x_{i+1}, x_{i+2}]$

- Die vorherige Funktion kann noch verallgemeinert werden zu

  ```
  averages_k :: Int→ [Double]→ [Double]
  ```

  welches statt von dreien von k benachbarten Elementen den Mittelwert bildet, oder mit anderen Worten averages ≡ averages_k 3.

5.4 *Bitlisten revisited* — *6 Punkte*

Im 2. Übungsblatt gab es noch keine Listen, und das Leben war etwas mühsam. Mit unserem neuen Kenntnissen können wir die dort implementierten Funktionen einfacher gestalten.

- Wir fangen an mit der einfachen Typdefinition

  ```
  data Bit = O | I    deriving (Eq, Show)
  ```

Zum Einstieg implementieren wir die Konversion von und nach Strings, und die Kodierung/Dekodierung von ganzen Zahlen in Bitlisten:

```
display     :: [Bit] → String
readBitlist :: String→ [Bit]
encode      :: Int→ [Bit]
decode      :: [Bit]→ Int
```

Hinweise:

- Wenn in den Bitlisten das vorderste Element das LSB ist, erleichtert dies die Implementation ganz erheblich.
- Für encode ist die vordefinierte Funktion unfoldr unerläßlich; sie wird mit **import** Data.List(unfoldr) importiert.

- Jetzt benötigen wir eine Hilfs-Funktionn, die von zwei Listen die kürzere von beiden von hinten (d.h. für Bitlisten in der MSB-Position) mit dem Wert eines Default-Arguments auf die Länge der längeren Liste auffüllt:

  ```
  pad :: a→ [a]→ [a]→ ([a], [a])
  ```

- Mit den zwei Funktionen für Volladdierer können wir jetzt die Additionsfunktion über foldl implementieren:

  ```
  fullAddC :: Bit→ Bit→ Bit→ Bit
  fullAddS :: Bit→ Bit→ Bit→ Bit

  add :: [Bit] → [Bit] → [Bit]
  ```

  Die Definitionen von fullAddC und fullAddS können Sie aus Ihrer Lösung des 2. Übungsblattes übernehmen.

? *Verständnisfragen*

1. foldr ist die „kanonische einfach rekursive Funktion“ (Vorlesung). Was bedeutet das, und warum ist das so? Für welche Datentypen gilt das?
2. Wann kann foldr f a xs auch für ein zyklisches Argument xs (bspw. eine zyklische Liste) terminieren?
3. Was ist die Grundidee hinter Parserkombinatoren, und funktioniert diese Idee nur für Parser oder auch für andere Problemstellungen?

Literatur

[1] *MISRA-C:2004 – Guidelines for the use of the C language in critical systems*. Motor Industry Research Association (MIRA) Limited, Nuneaton, UK, 2004.