Christoph Lüth
Marcus Ermler
Christian Maeder
Martin Ring
Diedrich Wolter

# 6. Übungsblatt

Ausgabe: 28.11.12
Abgabe: 07.12.12

Wir wollen das Rad nicht neu erfinden, aber trotzdem den *Heureka*-Effekt einer fundierten Erkenntnis erleben. Am Beispiel der (unterschiedlichen) Berechnung von Unterschieden zwischen Listen beziehungsweise Bäumen wollen wir zeigen, wo und wie Haskellprogramme den Unterschied machen.

## 6.1 *Diffenzen von Listen* *8 Punkte*

Implementieren Sie eine Funktion

```
diff :: Eq a ⇒ [a] → [a] → [(Ordering, a)]
```

welche die Differenz zweier Listen berechnet und zwar in ähnlicher Weise, wie das Unix-Utility `diff` die Differenz von Textzeilen berechnet.[1] Beispiel: die Dateien `A.txt` und `B.txt` enthalten Zeilen `A`, `B`, `C`, `D`, `E` respektive `A`, `D`, `E`, `B`. Das Unix-Kommando `diff -u A.txt B.txt` liefert dazu u.a. die Ausgabe:

```
 A
-B
-C
 D
 E
+B
```

Die Zeilen `A`, `D`, `E` sind dabei die gemeinsamen Zeilen; die mit `-` gekennzeichneten Zeilen `B` und `C` kommen aus der Datei `A.txt`, und die mit `+` gekennzeichnete Zeile `B` am Ende stammt aus der Datei `B.txt`. Die Ausgabe Ihrer Haskellfunktion `diff` soll — angewendet auf entsprechende Eingabelisten — folgendes Ergebnis liefern: [2]

```
[(EQ,"A"), (LT,"B"), (LT,"C"), (EQ,"D"), (EQ,"E"), (GT,"B")]
```

Die Implementierung von `diff` basiert auf einer Berechnung der längsten gemeinsamen Teilsequenz[3] (abgekürzt `lcs`) der Eingaben. Dazu vergleicht man die Anfangselemente beider Listen. Sind diese gleich (Fall 1) kann man zunächst die `lcs` von beiden Resten berechnen und später das Anfangselement hinzufügen. Sind die Anfangselemente verschieden (Fall 2), dann probiert man zwei (unabhängige) Berechnungen von `lcs` mit jeweils dem Anfangselement einer Liste weniger. Von diesen beiden Berechnungen wählt man dann die längere als Ergebnis. (Sind beide Berechnungen gleich lang, hat man die freie Wahl.)

Hinweis: Da die Berechnung der Länge einer Liste (durch `length` mit linearen Aufwand) nicht besonders effizient ist, bietet es sich an, zusammen mit der Ergebnisliste auch deren Länge zu berechnen.

Ihre (temporäre) Implementierung von `lcs` erweitern Sie danach zu einer Implementierung von `diff`, indem Sie die gemeisamen Elemente der längsten Teilsequenz in der Ausgabe mit `EQ` und zusätzliche Elemente der ersten bzw. zweiten Eingabe mit `LT` respektive `GT` kennzeichnen.

Das Ergebnis der `diff`-Ausgabe soll nun kompaktifiziert werden, indem benachbarte gleichmarkierte Elemente zusammengefasst werden. Dieses soll die Funktion `groupChunks` leisten, für die `groupBy` aus `Data.List` hilfreich sein könnte:

```
groupChunks :: [(Ordering, a)] → [(Ordering, [a])]
```

Die Ausgabe für obiges Beispiel wäre dann:

[1] Der Unterschied zu `diff` wird sein, dass unsere Funktion sowohl Unterschiede als auch Gemeinsamkeiten berechnet und zurückgibt, während `diff` hautpsächlich die Differenz zurückgibt. Daher sind aus der Ausgabe unserer Funktion beide Eingabedateien rekonstruierbar, während allein aus der Ausgabe von `diff` die Eingabe nicht immer rekonstruiert werden kann.

[2] Statt `Ordering` können Sie auch einen eigenen Datentyp mit 3 Werten verwenden.

[3] `http://en.wikipedia.org/wiki/Longest_common_subsequence_problem`

```
[(EQ,["A"]), (LT,["B", "C"]), (EQ,["D", "E"]), (GT,["B"])]
```

Implementieren Sie nun noch mit möglichst wenig Haskell-Code (und vor allem ohne Codeduplikation) die Funktionen

```
lcs, list1, list2 :: [(Ordering, [a])] → [a]
```

welche die längste gemeinsame Teilsequenz sowie die ursprünglichen Eingabelisten aus dem kompaktifizierten Ergebnis einer `diff`-Berechnung rekonstruieren.

Implementieren Sie außerdem eine rekursive Testfunktion

```
isSubSeqOf :: Eq a ⇒ [a] → [a] → Bool
```

die testet, ob die erste Liste eine Teilsequenz der zweiten ist. In wieweit unterscheidet sich diese Funktion von `isInfixOf` aus `Data.List`? Ferner implementieren Sie die Funktion

```
equal :: Eq a ⇒ [a] → [a] → Bool
```

welche anhand Eingaben aus dem kompaktifizierten `diff`-Ergebnis testet, ob die Listen gleich sind.

## 6.2 *Baumdifferenzen* — *5 Punkte*

Bäume sind etwas völlig anderes als Listen. Daher implementieren wir hier auch einen völlig anderen Algorithmus zur Berechnung von Differenzen, den wir aber später für die *Unifikation* nutzen können. Unseren Baumdatentyp `Tree` importieren wir aus `Data.Tree` oder definieren ihn als:

```
data Tree a = Node a [Tree a] deriving (Show, Eq)
```

Zwei Bäume sind *elementar* unterschiedlich, wenn ihre Werte an der Wurzel oder die Anzahl ihrer Unterbäume unterschiedlich sind. Nur wenn die Wurzelwerte und Anzahl der Unterbäume gleich sind, werden korrespondierende Unterbäume (also jeweils n-te Unterbäume von beiden) weiter auf elementare Unterschiede untersucht.

Implementieren Sie eine Funktion

```
diffPairs :: Eq a ⇒ Tree a → Tree a → [(Tree a, Tree a)]
```

welche eine Liste aller *elementar unterschiedlichen* Baumpaare (sogenannter *disagreement pairs*) aus den Eingaben berechnet. Vergewissern Sie sich, dass die Ergebnisliste leer ist, wenn beide Eingabebäume gleich sind, und einelementig, wenn beide Eingaben schon an der Wurzel elementar unterschiedlich sind.

Wir betrachten unsere Liste von elementar unterschiedlichen Baumpaaren weiter. Zwei Bäume eines solchen Paares sind *potenziell* unifizierbar, wenn einer der beiden Bäume ein *Blatt* (also ein Knoten ohne Unterbäume) ist und dieses Blatt im anderen Baum nicht vorkommt.

Implementieren Sie eine Testfunktion, die testet ob alle Paare *potenziell unifizierbar* sind. Schreiben Sie dazu eine Hilfsfunktion (den sogenannten *occurs check*), die testet, ob ein Blatt in einem Baum vorkommt.

```
mayUnify :: Eq a ⇒ [(Tree a, Tree a)] → Bool
occurs :: Eq a ⇒ a → Tree a → Bool
```

## 6.3 *Unifikation* — *7 Punkte*

Der Unifikationsalgorithmus versucht zu seiner Eingabe (einer Liste von Baumpaaren) eine *Substitution* zu berechnen, die die beiden Komponenten eines Paares gleich macht.

Wie am Ende der vorherigen Aufgabe angedeutet, ersetzt eine Substitution $t\left[\begin{smallmatrix}s\\a\end{smallmatrix}\right]$ ein Blatt mit dem Knotenwert `a` im Baum `t` durch den Baum `s`. Implementieren Sie eine Substitutionfunktion

```
subst :: Eq a ⇒ a → Tree a → Tree a → Tree a
```

so dass `subst a s t` die Substitution $t\left[\begin{smallmatrix}s\\a\end{smallmatrix}\right]$ berechnet. Enthält `t` kein Blatt `a`, liefert die Substitution `t` unverändert zurück.

Implementieren Sie nun die Unifikation wie folgt:

```
unify :: Eq a ⇒ [(Tree a, Tree a)] → Maybe [(a, Tree a)]
```

Für alle Paare berechnen sie die *elementar unterschiedlichen* Paare durch `diffPairs` und fügen diese zu einer neuen Paarliste zusammen. Ist diese Liste nicht *potenziell unifizierbar* (s.o.), dann schlägt die Unifikation mit `Nothing` fehl. Ist die elementare Paarliste leer, dann sind die Bäume schon gleich und das Ergebnis ist eine leere Substitutionsliste, `Just []`. Andernfalls betrachten Sie das erste Paar: eine Komponente dieses Paars ist ein Blatt, das durch den Baum der anderen Komponente, die das Blatt selbst nicht enthält, ersetzt werden kann. Die entsprechende Substitution wenden Sie nun auf alle Bäume der Paarliste an und erhalten eine (weniger elementare) Paarliste, die rekursiv durch `unify` weitere Substitutionen liefert, oder eben fehl schlägt.

Zum Überprüfen unserer Funktionen implementieren Sie nun noch die Funktion

```
substList :: Eq a ⇒ [(a, Tree a)] → Tree a → Tree a
```

um das Ergebnis der Unifikation auf die Eingabebäume anzuwenden. Falls also die Unifikation nicht fehl schlägt, muss gelten:

```
Just sl == unify [(t1, t2)] ⟹ substList sl t1 == substList sl t2
```

Geben Sie Beispiele für Baumpaare an, deren Unifikation aus unterschiedlichen Gründen fehl schlägt bzw. mit mehrelementigen Substitutionslisten erfolgreich ist.

## ? *Verständnisfragen*

1. Warum ist es hilfreich, Typen abzuleiten und nicht nur die gegebene Typisierung zu überprüfen?
2. Welches sind drei charakteristische Eigenschaften von Haskells Typsystem (Hindley-Milner)?
3. Was ist Unifikation, und welche Rolle spielt sie bei der Typinferenz? Wann kann sie fehlschlagen, und was sind Beispiele (Haskell-Programme, deren Typinferenz fehlschlägt) dafür?