Algebraische Datentypen – Binärbäume

Übungsblatt 2

Eric Kunze — 25. April 2020

Keine Garantie auf Vollständigkeit und/oder Korrektheit!

Aufgabe 3

Teilaufgabe (a)

Wir betrachten den algebraischen Datentyp `BinTree` definiert durch

```
data BinTree = Branch Int BinTree BinTree | Nil deriving Show
```

Wie versteht man nun diese Definition: entweder wir haben einen Knoten mit Beschriftung und zwei Kindern vorliegen oder der Baum ist leer bzw. als Nil kodiert.

Aufgabe. Geben Sie eine Funktion `insert :: BinTree -> [Int] -> BinTree` an, die alle Werte einer Liste von Integer-Zahlen in einen bereits bestehenden Suchbaum des Typs `BinTree` so einfügt, dass die Suchbaumeigenschaft erhalten bleibt. In einem Suchbaum muss für jeden Knoten `x` gelten, dass seine Beschriftung größer oder gleich (bzw. kleiner oder gleich) allen Beschriftungen im linken (bzw. rechten) Teilbaum von `x` ist.

Wir wollen die Rekursion über die Listenstruktur laufen lassen. Eine Liste ist entweder

- die leere Liste oder
- sie hat mindestens ein (erstes) Element

Wenn wir die leere Liste einfügen wollen, dann musst man natürlich nichts machen außer den Baum wieder ausgeben. So dann zum Rekursionsfall, d.h. wir haben wirklich Elemente in der Liste, die wir einfügen wollen. Dazu kann man sich erst einmal das etwas leichtere Problem ansehen, nämlich anstatt einer ganzen Liste nur ein einzelnes Element einzufügen. Dies erledigt die Funktion

```
insertSingle :: BinTree -> Int -> BinTree
```

für uns. Der Basisfall dort ist wieder relativ einfach einzusehen, d.h. wenn wir in einen leeren Baum ein Element einfügen wollen, dann erstellen wir einen Knoten mit entsprechender Beschriftung und leeren Kindern:

```
insertSingle :: BinTree -> Int -> BinTree
insertSingle Nil            x = Branch x Nil Nil
```

Betrachten wir den Rekursionsfall von `insertSingle`. Nehmen wir an, dass wir im Baum mindestens einen Knoten vorliegen haben; dieser trägt eine Beschriftung `y` und einen linkes Kind `l` sowie ein rechtes Kind `r`. Wollen wir nun dort ein Schlüssel (= Knotenbeschriftung) `x` korrekt einfügen.

```
insertSingle :: BinTree -> Int -> BinTree
insertSingle Nil            x = Branch x Nil Nil
insertSingle (Branch y l r) x = ...
```

Nun machen wir uns die Eigenschaft eines Suchbaums zu Nutze, dass in `l` alle Schlüssel kleiner sind als `y` und in `r` alle Schlüssel größer sind als `y`. Dementsprechend müssen wir an jedem Knoten entscheiden, ob wir in den linken oder rechten Teilbaum einfügen wollen.

- Ist das einzufügende Element `x` kleiner als `y`, dann gehört es per Definition des Suchbaums in den linken Teilbaum. Also gestalten wir uns einen "neuen" Knoten, der die gleiche Beschriftung `y` trägt und auch den gleichen rechten Teilbaum `r`, da wir dort ja nichts verändert haben. Den linken Teilbaum müssen wir aber verändern, nämlich so, dass dort `x` eingefügt wird -– das ist aber das bekannte Problem "Einfügen eines Schlüssels in einen Baum" und das macht uns die Funktion `insertSingle`. Daher kommt also die Zeile

  ```
  insertSingle :: BinTree -> Int -> BinTree
  insertSingle Nil            x = Branch x Nil Nil
  insertSingle (Branch y l r) x
    | x < y     = Branch y (insertSingle l x) r
  ```

- In dem Fall, dass der einzufügende Schlüssel x größer ist als y, dann gehen wir analog vor, verändern jedoch nicht den linken, sondern den rechten Teilbaum, also:

  ```
  insertSingle :: BinTree -> Int -> BinTree
  insertSingle Nil            x = Branch x Nil Nil
  insertSingle (Branch y l r) x
    | x < y     = Branch y (insertSingle l x) r
    | otherwise = Branch y l (insertSingle r x)
  ```

Damit haben wir also das Problem "Einfügen eines Schlüssels in einen Baum" mithilfe von insertSingle gelöst. Dann müssen wir uns jetzt noch darum kümmern, dass wir von der Liste in `insert` zum einzelnen Element in `insertSingle` kommen. Das läuft aber relativ einfach als Rekursion über die Listenstruktur. Wir teilen die Liste also auf in `(x:xs)`, spalten also ein erstes Element ab. Nun wollen wir das `x` als einzelnes Element einfügen via `insertSingle t x` und bekommen dann aber schon einen neuen Baum, den wir `t'` nennen. Wenn wir die restlichen Elemente von `xs` einfügen wollen, dann müssen wir aufpassen und müssen diese in `t'` und nicht in `t` einfügen. Da entsteht dann der rekursive Aufruf `insert t' xs`. Die Auslagerung der Berechnung von `t'` ist einfach eine kleine Feinheit, die ich für bisschen verständlicher halte, man kann auch die rechte Seite von `t'` direkt in den rekursiven Aufruf packen wie in der Musterlösung.

Damit erhalten wir also als Lösung der Aufgabe:

```
1 insert :: BinTree -> [Int] -> BinTree
2 insert t []     = t
```

```
insert t (x:xs) = insert t' xs
  where
    t' = insertSingle t x
    insertSingle :: BinTree -> Int -> BinTree
    insertSingle            Nil x = Branch x Nil Nil
    insertSingle (Branch y l r) x
      |     x < y = Branch y (insertSingle l x)                  r
      | otherwise = Branch y                  l (insertSingle r x)
```

Um das ganze zu testen, legen wir uns mal einen Beispielbaum an. Das funktioniert im Prinzip so, dass man die Typdefinition nimmt und dann für die Typen darin konkrete Werte einsetzt. Also:

```
testTree :: BinTree
testTree = Branch 5 Nil Nil
```

steht zum Beispiel für den Baum, der nur eine `5` enthält. Den können wir nun immer weiter erweitern, indem wir die `Nil`'s ersetzen und dabei bisschen auf die Klammerung achten.

```
testTree = Branch 5
       (Branch 3 Nil Nil )
       (Branch 8 Nil Nil )
```

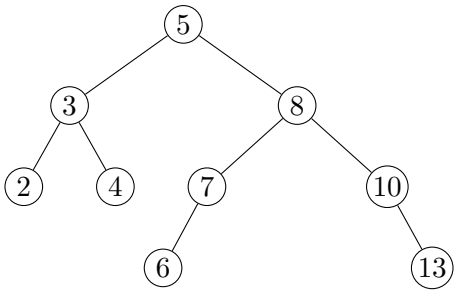
Um das bisschen übersichtlich zu machen, schreibt man das dann so bisschen gestaffelt untereinander (es wird zwangsweise immer bisschen unübersichtlich). Dann kann man das immer weiter ausbauen und ich hab jetzt mal einen Suchbaum konstruiert, der etwas größer ist.

```
testTree :: BinTree
testTree = Branch 5
      (Branch 3
        (Branch 2 Nil Nil)
        (Branch 4 Nil Nil)
      )
      (Branch 8
        (Branch 7
          (Branch 6 Nil Nil)
          (Nil)
        )
        (Branch 10
          (Nil)
          (Branch 13 Nil Nil)
        )
      )
```

Schreibt man das in gewohnter Schreibweise, so erhält man den Baum:

Damit dann auch die Ausgabe klappt, wenn wir unsere Funktion mit `ghci` testen, müssen wir noch eine klein wenig die Typdefinition ändern:

```
data BinTree = Branch Int BinTree BinTree | Nil deriving Show
```



Die Direktive `Show` sorgt einfach dafür, dass eine standardmäßige (trotzdem relativ hässliche) Ausgaberoutine bereitgestellt wird. Damit können wir jetzt in `ghci` testen, z.B. mit dem Aufruf `insert testTree [9,12]`

Bemerkung (Suchbäume). *In der Regel betrachtet man Suchbäume ohne Dopplungen, d.h. jede Zahl sollte nur einmal vorkommen. Dementsprechend macht es wenig Sinn eine bereits bestehende Zahl einzufügen.*

Teilaufgabe (b)

Aufgabe. Geben Sie eine Haskell-Funktion einschließlich der Typ-Definition an, die testet, ob zwei Binärbäume des Typs `BinTree` identisch sind.

Man überlegt sich hier leicht, dass zwei Bäume genau dann gleich sind, wenn beide Wurzelknoten die gleiche Beschriftung tragen und die beiden Kinder jeweils übereinstimmen. Somit erhält man für den Fall, dass beide Bäume mindestens einen Knoten tragen, den folgenden Rekursionsfall

```
equal :: BinTree -> BinTree -> Bool
equal (Branch x l1 r1) (Branch y l2 r2) = (x == y] &&
                                          (equal l1 l2) &&
                                          (equal r1 r2)
```

Als Basisfall würde man zunächst den offensichtlichen nehmen: beide Bäume sind leer und damit auch gleich:

```
equal :: BinTree -> BinTree -> Bool
equal Nil               Nil               = True
equal (Branch x l1 r1) (Branch y l2 r2) = (x == y) &&
                                          (equal l1 l2) &&
                                          (equal r1 r2)
```

Aber hier gibt es eine kleine Gemeinheit. Die Funktion führt eine Rekursion über beide Bäume aus. Jetzt kann es aber auch passieren, dass beide Bäume unterschiedlich groß sind. Wenn wir nun immer wieder den Rekursionsfall anwenden, dann werden die Bäume zwar immer kleiner, aber aufgrund der unterschiedlichen Größe kann einer eher leer werden als der andere. Das ist dann in noch keinem Fall abgedeckt, weil der leere Baum dann nicht mehr die `Branch`-Struktur besitzt (Pattern Matching schlägt fehl), anderseits auch der Basisfall mit zwei leere Bäumen

nicht passt, da ja einer von beiden noch nicht leer ist. Dementsprechend gibt es noch die beiden Fälle

```
equal Nil               (Branch y l2 r2) = False
equal (Branch x l1 r1) Nil               = False
```

Ich denke, dass die Bäume dann nicht gleich sind, ist klar (leer und nichtleer kann halt nicht gleich sein). Entweder man schreibt nun die beiden Fälle noch als Basis dazu oder man sagt, dass alles, was durch die bereits bestehenden Fälle (leer-leer und voll-voll) noch nicht abgedeckt ist (und das sind genau die beiden Fälle leer-voll und voll-leer), das wird zu `False`. Genau diese Variante ist in der Musterlösung mit den Wildcards `_` angegeben. Wildcards sind dabei Platzhalter für beliebige Werte; man könnte auch stets Variablen `t1` und `t2` anstelle derer schreiben, jedoch werden diese auf der rechten Seite ohnehin nicht zu Berechnung benötigt.

Man erhält somit eine der beiden folgenden Lösungen:

```
1 equal :: BinTree -> BinTree -> Bool
2 equal Nil               Nil               = True
3 equal Nil               (Branch y l2 r2) = False
4 equal (Branch x l1 r1) Nil               = False
5 equal (Branch x l1 r1) (Branch y l2 r2) = (x == y) &&
6                                            (equal l1 l2) &&
7                                            (equal r1 r2)
```

oder

```
1 equal :: BinTree -> BinTree -> Bool
2 equal Nil               Nil               = True
3 equal (Branch x l1 r1) (Branch y l2 r2) = (x == y) &&
4                                            (equal l1 l2) &&
5                                            (equal r1 r2)
6 equal _                 _                 = False
```