Algebraische Datentypen – Binärbäume

Übungsblatt 3

Eric Kunze — 30. April 2022

Keine Garantie auf Vollständigkeit und/oder Korrektheit!

Aufgabe 2

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.

Teilaufgabe (a)

Wir wollen einen Beispielbaum anlegen. Das funktioniert im Prinzip so, dass man die Typdefinition nimmt und dann für die Typen darin konkrete Werte einsetzt. Also:

```
mytree :: BinTree
mytree = Branch 0 Nil Nil
```

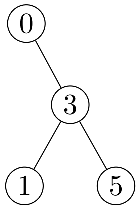
steht zum Beispiel für den Baum, der nur eine `0` enthält. Den können wir nun immer weiter erweitern, indem wir die `Nil`’s ersetzen und dabei bisschen auf die Klammerung achten.

```
mytree = Branch 0
            ( Nil )
            ( Branch 3 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 bis wir den geforderten Baum erhalten.

```
mytree = Branch 0
            ( Nil )
            ( Branch 3
              ( Branch 1 Nil Nil )
              ( Branch 5 Nil Nil )
             )
```

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



Teilaufgabe (b)

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

Die Typdefinition unserer Funktion `equal` ist schnell gefunden: wir brauchen zwei Bäume vom Typ `BinTree`, die miteinander verglichen werden sollen, als Argumente; und geben am Ende einen Wahrheitswert `Bool` zurück. Demnach ist die Typdefinition gegeben durch:

```
equal :: BinTree -> BinTree -> Bool
```

Man überlegt sich 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:

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

oder

```
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)
equal _                 _               = False
```

Teilaufgabe (c)

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:

```
insert :: BinTree -> [Int] -> BinTree
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)
```

Wir wollen nun diese Funktion mit unserem Beispielbaum `mytree` aus Teil (a) testen. Damit dann auch die Ausgabe klappt, wenn wir unsere Funktion mit `ghci` testen, müssen wir noch eine klein wenig die Typdefinition ändern:

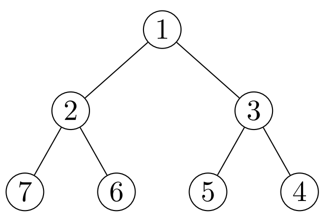
```
data BinTree = Branch Int BinTree BinTree | Nil deriving Show
```

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.*

0.1 Teilaufgabe (d)

Aufgabe. Geben Sie eine Funktion `unwind :: BinTree -> [Int]` an, welche einen gegebenen Baum ebenenweise traversiert und dabei die Liste der Knotenbeschriftungen in der Reihenfolge der Traversierung berechnet. Dabei soll zunächst die Wurzel besucht werden, dann die direkten Nachfolger der Wurzel von links nach rechts, dann deren direkte Nachfolger von links nach rechts, usw. Zum Beispiel soll für den unten stehenden Baum `tree` gelten, dass `unwind tree == [1,2,3,7,6,5,4]`.



Zunächst einmal ist der dargestellte Baum `tree` in Haskell wie folgt zu notieren (vgl. **??**):

```
tree :: BinTree
tree = Branch 1
      ( Branch 2
        ( Branch 7 Nil Nil )
        ( Branch 6 Nil Nil ) )
      ( Branch 3
```

```
    ( Branch 5 Nil Nil )
    ( Branch 4 Nil Nil ) )
```

Für Umsetzung der ebenenweisen Traversierung nutzen wir folgende Idee: wir speichern uns stets eine Liste von Bäumen. In diese Liste nehmen wir uns immer den ersten Baum heraus, fügen dessen Wurzel zur Ausgabeliste hinzu und hängen die Kindbäume hinten an die Speicherliste an – diese werden in der nächsten Runde betrachtet.

Wir beginnen mit der Hauptfunktion `unwind` und übergeben das Problem vollständig an die Hilfsfunktion, die wir in diesem Fall `go` nennen. Gemäß der beschriebenen Idee müssen wir dazu das Argument von `unwind`, ein Binärbaum, in eine Liste eintragen, also

```
undwind :: BinTree -> [Int]
unwind t = go [t]
```

Nun können wir uns mit der `go`-Funktion beschäftigen. Wie bereits erwähnt ist das Argument von `go` eine Liste von Bäumen, d.h. `[BinTree]`; die Rückgabe soll und muss um mit dem Ergebnistyp von `unwind` übereinzustimmen, vom Typ `[Int]` sein. Damit ist der Typ der Funktion gegeben durch

```
go :: [BinTree] -> [Int]
```

Nun müssen wir `go` rekursiv definieren. Jedoch haben wir hier zwei rekursive Datenstrukturen vorliegen: Listen und Bäume. In diesem Beispiel wollen wir auch beide Rekursionen parallel ablaufen lassen. Beginnen wir dazu mit der Rekursion auf der Liste (von Bäumen). Diese müssen wir für den Rekursionsfall aufspalten in Head und Tail, also in der Form `(t:ts)`. Der Basisfall für Listen sieht wie üblich die leere Liste vor. Wir brauchen also für die Listenrekursion folgende zwei Zeilen:

```
go []     = ...
go (t:ts) = ...
```

Nun bringen wir die zweite Rekursion ins Spiel: die Rekursion auf der Baumstruktur - und zwar um genau zu sein auf der Baumstruktur des ersten Baumes der Liste, also der Struktur von `t`. Im Rekursionsfall zerlegen wir diesen in einen Knoten mit Beschriftung `x` und zwei Kindern `l` und `r`, d.h. in `t` = `Branch x l r`; im Basisfall ist dieser Baum leer, d.h. `t` = `Nil`. Damit zerlegt sich die bisherige zweite Zeile in zwei weitere Zeilen:

```
go []     = ...
go ( Nil            : ts) = ...
go ((Branch x l r) : ts) = ...
```

Damit ist das Grundgerüst für die Funktion gebaut. Wir müssen uns "nur" noch Gedanken für die jeweiligen Rückgaben auf der rechten Seite machen. Also los geht's ...

Wir erinnern uns, dass jeder nichtleere Baum in genau drei Teile zerlegt werden kann: ein Wurzelknoten und ein linkes sowie ein rechtes Kind. Auf diese drei Teile haben wir im Rekursionsfall Zugriff und können sie neu anordnen. Der Wurzelknoten `x` soll natürlich direkt in die Ausgabeliste wandern. Diese Ausgabeliste ist die Rückgabe von `go`, also können wir dieses `x` als Head der Ausgabeliste “konstruieren”. Die beiden Kindbäume müssen wir an unsere Baum-Speicher-Liste anhängen. Bisher bestand diese Liste aus dem ersten Baum, den wir gerade zerlegen, und den Restbäumen in der Liste `ts`. Den ersten Baum haben wir soeben abgearbeitet, also entfällt dieser; `ts` verbleibt; die Bäume `l` und `r` hängen wir als zweielementige Liste `[l, r]` hinten an `ts` an. Insgesamt haben wir bisher folgendes Ergebnis:

```
go []                   = ...
go ( Nil          : ts) = ...
go ((Branch x l r) : ts) = x : go (ts ++ [l, r])
```

Treffen wir in unserer Merkliste auf einen leeren Baum, dann hat dieser schon keinen Knoten mehr, den wir der Ausgabeliste hinzufügen könnten, also können wir den leeren Baum einfach ignorieren:

```
go []                   = ...
go ( Nil          : ts) = go ts
go ((Branch x l r) : ts) = x : go (ts ++ [l, r])
```

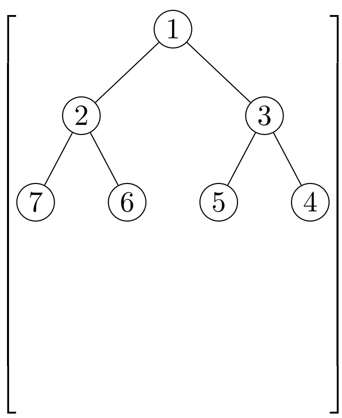
Ist die Merkliste von noch verbleibenden Bäumen schließlich leer, so sind alle Knoten abgearbeitet und wir sind fertig. Da wir im Rekursionsfall immer mit dem cons-Operator einzelne Elemente hintereinander gehangen haben, müssen wir zum Abschluss noch eine leere Liste anhängen. Also:

```
go []                   = []
go ( Nil          : ts) = go ts
go ((Branch x l r) : ts) = x : go (ts ++ [l, r])
```

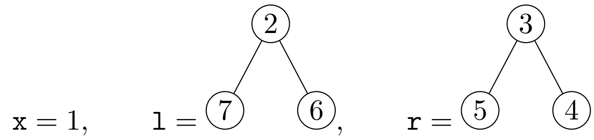
Die Hilfsfunktion `go` können wir nun noch in eine lokale Definition schieben und erhalten schließlich die vollständige ebenenweise Traversierung:

```
undwind :: BinTree -> [Int]
unwind t = go [t]
  where
    go []                   = []
    go ( Nil          : ts) = go ts
    go ((Branch x l r) : ts) = x : go (ts ++ [l, r])
```

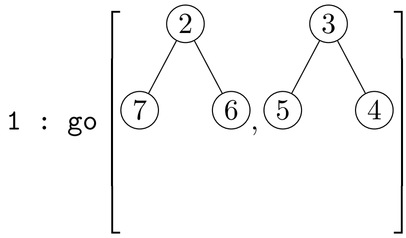
Um diese Funktion noch besser zu verstehen, wollen wir anhand des Beispielbaums `tree` die Abarbeitung veranschaulichen. Wir starten in `unwind` und packen den vollständigen Baum in eine Liste, also starten wir mit dem Aufruf

go


Wir zerlegen den Baum in Wurzelknoten und die beiden Kinder:



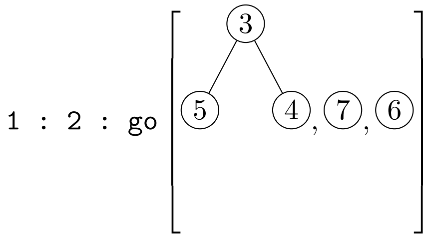
Damit folgt der nächste rekursive Aufruf von `go` mit



oder in Haskell-Schreibweise

```
1 : go [Branch 2 (Branch 7 Nil Nil) (Branch 7 Nil Nil) ,
        Branch 3 (Branch 5 Nil Nil) (Branch 4 Nil Nil) ]
```

Nun spalten wir wieder den ersten Baum der Speicherliste in `go` ab, packen die Wurzel zur Ausgabe und die beiden Kinder hinten an die Speicherliste, also



Im Anschluss wird auch der nächste Baum nach diesem Prinzip zerlegt und wir erhalten

$$\texttt{1 : 2 : 3 : go}\left[7, 6, 5, 4\right]$$

Nun kommen wir an den ersten kleinen kritischen Punkt, denn die Kinder des nächsten Baumes sind leer. Das macht in diesem Moment aber noch nichts aus, da wir dennoch den Knoten 7 in `Branch 7 Nil Nil` aufspalten können. Damit erhalten wir im nächsten rekursiven Aufruf

$$\texttt{1 : 2 : 3 : 7 : go}\left[6, 5, 4, \texttt{Nil}, \texttt{Nil}\right]$$

Für die nächsten drei Bäume erfolgt das Spiel komplett analog und wir rufen rekursiv wie folgt auf:

$$\texttt{1 : 2 : 3 : 7 : 6 : 5 : 4 : go}[\texttt{Nil}, \texttt{Nil},\ \texttt{Nil}, \texttt{Nil},\ \texttt{Nil}, \texttt{Nil},\ \texttt{Nil}, \texttt{Nil}]$$

Nun wenden wir den Baum-Basisfall an, da das erste Element der Speicherliste auf `Nil` passt und keine `Branch`-Struktur mehr aufweist. Damit entfernen wir dieses erste `Nil` und rufen `go` einfach wieder mit der verbleibenden Restliste auf ohne etwas zur Ausgabe hinzuzufügen:

$$\texttt{1 : 2 : 3 : 7 : 6 : 5 : 4 : go}\,[\texttt{Nil},\ \texttt{Nil},\texttt{Nil},\ \texttt{Nil},\texttt{Nil},\ \texttt{Nil},\texttt{Nil}]$$

Das gleiche Spiel passiert nun noch sieben Mal und wir erhalten schließlich

`1 : 2 : 3 : 7 : 6 : 5 : 4 : go []`

Damit können wir noch den Listen-Basisfall anwenden, der `go []` zu `[]` evaluiert:

`1 : 2 : 3 : 7 : 6 : 5 : 4 : []`

Führen wir nun noch alle cons-Operatoren aus, so erhalten wir am Ende die finale Ergebnisliste `[1,2,3,7,6,5,4]`.