Sven Eric Panitz

Lehrbriefe
Funktionale Programmierung

3 Binäre Suchbäume

Binäre Suchbäume

Sven Eric Panitz

24. März 2025

Inhaltsverzeichnis

1 Binäre Suchbäume

Eine häufig benötigte Datenstruktur stellen Mengen dar. Es bieten sich zwei prinzipiell unterschiedliche Umsetzungen dafür an, Mengen effizient zu implementieren.

- als ein HashSet, bei dem die Elemente entsprechend eines Hash-Wertes in einem Array an einem bestimmten Index gespeichert werden,
- oder als binärer Suchbaum. Dabei ist eine Ordnung auf den zu speichernden Elementen notwendig. Entsprechend der Ordnung findet sich ausgehend vom Wurzelelement das gesuchte Element dann im linken bzw. rechten Teilbaum.

In dieser Aufgabe soll die Implementierung über einen Binärbaum in Haskell realisiert werden.

Wir werden dabei die Haskellumsetzung einer Lösung in Java gegenüberstellen. Dieses kann helfen, die Gemeinsamkeiten und Unterschiede zwischen einem funktionalen und einem objektorientierten Paradigma zu verstehen.

Tatsächlich sind im Standard-Modul `Data.List` Funktionen definiert, die Listen als Mengen betrachten. Die Funktion

Tree.lhs

```
nub :: Eq a => [a] -> [a]
```

löscht alle doppelten aus einer Liste, macht sie also zur Menge.

Zusätzlich sind Funktionen für die Vereinigung und dem Schnitt implementiert:

Tree.lhs

```
union :: Eq a => [a] -> [a] -> [a]
intersect :: Eq a => [a] -> [a] -> [a]
```

Diese Versionen haben natürlich bei größeren Mengen Performance-Probleme.

2 Der Datentyp für Binärbäume

Wir definieren das Modul für Binärbäume:

Tree.lhs

```
> module Tree where
```

Eine Menge ist ein Datentyp, der einen beliebigen aber festen Elementtypen hat. In objektorientierten Programmiersprachen spricht man dabei von einem generischen Typ mit einer Typvariablen als Elementtyp. In funktionalen Sprachen ist hier der Begriff ›polymorpher Typ‹ etabliert.

Unser Binärbaum ist ein Typ, der einen variablen aber festen Elementtypen hat. Auch in Haskell wird hierfür eine Typvariable eingeführt. Die Bezeichner für Typvariablen müssen in Haskell mit einem Kleinbuchstaben beginnen. Zumeist wird nur ein einzelner Buchstabe als Bezeichner genommen.

Somit definieren wir einen Typ `Tree`, der polymorph über eine Variable `a` ist.

Tree.lhs

```
> data Tree a =
```

Dieses entspricht ein wenig der folgenden Klassendefinition in Java:

Tree.java

```
class Tree<A extends Comparable<? super A>>{
```

Hier ist der erste Unterschied, dass bereits in der Klassendefinition spezifiziert ist, dass die Elemente eine Ordnung brauchen. Java-technisch verlangen wir, dass die Elemente die Schnittstelle `Comparable` implementieren sollen.

Diese Einschränkung braucht im Haskell-Datentyp `Tree` noch nicht gemacht zu werden. Hier wird erst in Funktionen, die die Sortiereigenschaft benötigen, der Typ der Elemente entsprechend eingeschränkt.

Wir wollen zwei Arten von Bäumen definieren. Einmal einen Baumknoten, der den leeren Baum repräsentiert. Hierzu gibt es einen Konstruktor, der kein Argument hat.

Tree.lhs

```
>   Empty
```

In einer Java-Implementierung wäre dieses also auch ein Konstruktor ohne Argument. Dabei merkt sich die Java-Klasse in einem Feld, ob die leere Menge dargestellt wird.

Tree.java

```
boolean isEmpty = true;
Tree(){}
```

Der zweite Konstruktor verbindet einen linken Teilbaum mit einem rechten Teilbaum und hat ein neues Wurzelelement.

Tree.lhs

```
>   |Branch (Tree a) a (Tree a)
```

In der Java-Umsetzung braucht man denselben Konstruktor und Felder, um die drei Argumente in dem Baum zu speichern. In Java bietet sich an, diesen Konstruktor privat zu machen, damit nur über die add-Methode neue Element in einen Baum eingetragen werden.

Tree.java

```
Tree<A> left = null;
A element = null;
Tree<A>  right = null;
private Tree(Tree<A> l,A e,Tree<A> r){
  left = l;
  element = e;
  right = r;
  isEmpty = false;
}
```

Für schnelle Ausgabe als String lassen wir die Typklasse `Show` von Haskell automatisch implementieren.

Tree.lhs

```
>     deriving (Show)
```

2.1 Funktionen auf Mengen

Für die Datenstruktur können jetzt Funktionen definiert werden. Eine erste Funktion soll die Anzahl der Elemente in der Menge angeben. Hierzu ist im Baum rekursiv abzusteigen und es sind die nichtleeren Baumknoten zu zählen.

In Java ist diese Funktion relativ geradlinig umzusetzen. Leere Bäume habe eine Größe von 0. Ansonsten gibt es ein Element an der Wurzel und zusätzlich die Elemente in den zwei Kindbäumen.

Tree.java

```
long size(){
  if (isEmpty) return 0;
  return 1 + left.size() + right.size();
}
```

In Haskell kann die Unterscheidung, ob es sich um einen leeren Baum oder um einen verzweigenden Knoten handelt, mit Pattern-Matching realisiert werden. Wir erhalten zwei Funktionsgleichungen. Die erste für den leeren Baum definiert das Ergebnis 0:

Tree.lhs

```
> size :: Num a => Tree t -> a
> size Empty = 0
```

Die zweite Gleichung ist für das Pattern eines verzweigenden Knotens. Hierbei werden in dem Pattern die Kindbäume und das Element entsprechend der Reihenfolge im Konstruktor `Branch` an Variablen gebunden. Das Ergebnis ist dann die Summe der Größen der Teilbäume plus 1.

Tree.lhs

```
> size (Branch left element right) = 1+size left+size right
```

3 Aufgaben

Jetzt sind Sie dran, um für diese Datenstruktur eine Reihe weiterer Funktionen umzusetzen. Als Leitfaden sind jeweils die Umsetzungen in Java gegeben.

Aufgabe 1

a) Schreiben Sie jetzt eine Funktion, die es erlaubt, ein neues Element in den Baum einzufügen. Da Sie wegen der referientiellen Transparenz den Baum nicht verändern können, ist ein neuer Baum als Ergebnis zu erzeugen. Er können aber Teile des Argumentbaumes verwendet werden, da sich ja keine Daten verändern können. Es kann somit keine Seiteneffekte geben.

Die Funktion benötigt nun eine Einschränkung auf die Elemente des Baumes. Jetzt sind nicht mehr beliebige Typen für die Typvariable a erlaubt, sondern nur solche Typen, für die eine Ordnung existiert, also Vergleichsoperatoren definiert sind.

Tree.lhs

```
> add :: Ord a => a -> Tree a -> Tree a
> add _ t = t
```

Als kleine Richtlinie für die Umsetzung kann die Implementierung in Java dienen.

Tree.java

```
void add(A e){
  if (isEmpty) {
    left = new Tree<>();
    right = new Tree<>();
    element = e;
  }else if (e.compareTo(element)<0){
    left.add(e);
  }else {
    right.add(e);
  }
}
```

Für einen leeren Baum ist ein Blatt zu erzeugen, ansonsten wird das Einfügen im linken bzw. rechten Teilbaum durch einen rekursiven Aufruf gemacht.

Für eine bequeme Anwendung dieser Funktion, können wir auch einen binären Operator definieren, der als Infixoperator dient, um ein Element einer Menge hinzuzufügen.

Tree.lhs

```
> infixr 5 +>
> (+>)  el t = add el t
```

Mit diesen Definitionen sollten Sie nun in der Lage sein, erste kleine Aufrufe im Interpreter zur Auswertung zu bringen.

Shell

```
00002BinTree$ ghci solution/Tree.lhs
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Tree             ( solution/Tree.lhs, interpreted )
Ok, one module loaded.
*Tree> let t1 = 17 +> 42 +> -1 +> 100 +> 576576576 +> Empty
*Tree> t1
Branch (Branch (Branch Empty (-1) (Branch (Branch Empty 17 Empty) 42 Empty))
100 Empty) 576576576 Empty
```

```
*Tree> size t1
5
*Tree>
```

b) Schreiben Sie jetzt eine Funktion, die testet, ob ein Element in der Menge enthalten ist.

Tree.lhs

```
> contains :: Ord a => a -> Tree a -> Bool
> contains _ _ = False
```

An der entsprechenden Umsetzung in Java kann man erkennen, dass die Haskell-Lösung auch die vier Fälle enthalten wird:

- Der Baum ist leer und enthält nichts.
- Bereits an der Wurzel ist das gesuchte Element.
- Das gesuchte Element kann nur im linken Kind enthalten sein.
- Das gesuchte Element kann nur im rechten Kind enthalten sein.

Tree.java

```
boolean contains(A e){
  if (isEmpty) return false;
  if (e.compareTo(element)==0) return true;
  if (e.compareTo(element)<0) return left.contains(e);
  return right.contains(e);
}
```

c) Schreiben Sie jetzt eine Funktion, die aus der Menge ein Element löscht. Machen Sie hierzu eine lokale Funktion, die den Baum neu organisiert, wenn an der Wurzel das zu löschende Element steht.

Tree.lhs

```
> remove :: Ord a => a -> Tree a -> Tree a
> remove _ t = t
```

Als Vorlage auch hier eine Implementierung in Java. Zunächst wird ein Knoten gesucht, an dem das zu löschende Element ist.

Tree.java

```
void remove(A e){
  if (isEmpty) return;
  if (e.compareTo(element)<0) left.remove(e);
  else if (e.compareTo(element)>0) right.remove(e);
  else rebuild(left,right);
```

```
}
```

Die private Hilfsmethode zum Umstellen des Baumes, sodass das Wurzelelement nicht mehr im Baum ist. Wir haben also zwei binäre Suchbäume, aus denen einer zu bauen ist. Einfach ist das, wenn einer der beiden Bäume leer ist. Ansonsten können wir für den Ergebnisbaum vom ersten Baum den linken Teilbaum und das Element nehmen und den rechten Teilbaum dann mit dem zweiten Baum durch einen rekursiven Aufruf verbinden.

Tree.java

```
private void rebuild(Tree<A> l,Tree<A> r){
  if (l.isEmpty) {
    left = r.left;
    element = r.element;
    right = r.right;
    isEmpty = r.isEmpty;
  }else if (r.isEmpty) {
    left = l.left;
    element = l.element;
    right = l.right;
    isEmpty = l.isEmpty;
  }else {
    element = l.element;
    left = l.left;
    right.rebuild(l.right,r);
  }
}
```

Die private Methode `rebuild` wird in Haskell am besten als lokale Funktion der Funktion `remove` definiert. Hierzu kann man die `where`-Klausel verwenden.

So wie für die Funktion `add` können wir auch einen Infixoperator für die Funktion `remove` definieren.

Tree.lhs

```
> infixr 5 >-
> (>-) t el = remove el t
```

d) Schreiben Sie eine Funktion, die eine Liste der Elemente in sortierter Reihenfolge erzeugt.

Tree.lhs

```
> inorder :: Tree t -> [t]
> inorder _ = []
```

Auch hier sei eine Umsetzung in Java als kleiner Leitfaden gegeben.

Tree.java

```
java.util.List<A> inorder(){
  var result = new java.util.ArrayList<A>();
  inorder(result);
  return result;
}
void inorder(java.util.List<A> result){
  if (isEmpty) return;
  left.inorder(result);
  result.add(element);
  right.inorder(result);
}
```

e) Schreiben Sie eine Faltungsfunktion, die alle Elemente der Menge mit einer Funktion verknüpft.

Tree.lhs

```
> reduce :: Ord t => t1 -> (t1 -> t -> t1) -> Tree t -> t1
> reduce x _ _ = x
```

Als Anleitung sei eine falsche Implementierung in Java gegeben. Die Funktion soll alle Elemente mit einem Startobjekt über eine mitgegebene Funktion verrechnen. Hierzu kann man das Wurzelelement mit dem Startobjekt verrechnen. Das Ergebnis als Startobjekt nehmen und mit den übrigen Elementen des Baumes verrechnen. Hierzu ist das Wurzelelement aus dem Baum herauszunehmen, denn es wurde ja schon verrechnet. Die folgende Java-Implementierung verändert also das `this`-Objekt.

Tree.java

```
  <B> B reduce(B start,java.util.function.BiFunction<B,A,B> f){
    if (isEmpty) return start;

    remove(element);
    return reduce(f.apply(start,element),f);
  }
} //end of class Tree
```

In Haskell können Sie hingegen die obige Java-Lösung direkt umsetzen, denn dort wird nie ein Objekt modifiziert.

f) Setzen Sie den Operator \/ um, der die Vereinigung zweier Mengen realisieren soll:

Tree.lhs

```
> infixl 5 \/
> (\/) :: Ord a => Tree a -> Tree a -> Tree a
> xs \/ ys = Empty
```

Wenn Sie die Funktion `reduce` verwenden, so lässt sich die Funktion in einer Zeile lösen.

g) Setzen Sie den Operator `/\` um, der den Schnitt zweier Mengen realisieren soll:

Tree.lhs

```
> infixl 5 /\
> (/\) :: Ord a => Tree a -> Tree a -> Tree a
> xs /\ ys = Empty
```

Auch hier findet sich mit der Funktion `reduce` wieder ein Einzeiler. Versuchen Sie es.

h) Implementieren Sie die Gleichheit auf Mengen:

Tree.lhs

```
> instance (Ord a) => Eq (Tree a) where
>   _ == _ = False
```

Zwei Mengen sind gleich, wenn sie gleich viel Elemente haben und auch der Schnitt beider Mengen die unveränderte Anzahl von Elementen.

i) Schreiben Sie jetzt eine Funktion, die eine neue Menge erzeugt, deren Elemente aus der Argumentmenge durch Anwendung einer Funktion entstehen. Auch hier können Sie mit der Funktion `reduce` eine Lösung in nur einer Zeile finden.

Tree.lhs

```
> tmap :: (Ord a,Ord b) => (a -> b) -> Tree a -> Tree b
> tmap f xs = Empty
```

4 Beispielaufrufe

Es folgen ein paar Beispiele und weitere Funktionsdefinitionen für die Mengenimplementierung.

Beginnen wir mit der Definition einer Menge von Zahlen

Tree.lhs

```
> s1 = 56 +> 5 +> 67 +> 17 +> 4
>       +> -576 +> 42 +> 10000000789798876657 +> Empty
```

Und um zu sehen, dass Binärbäume polymorph mit unterschiedlichen Elementtypen verwendet werden können eine zweite Menge, nun mit Strings als Elemente:

Tree.lhs

```
> t2  = "hallo" +> "welt" +> "witzebitzelbritz"
>         +> "pandudel" +> "sowieso" +> "klaro" +> Empty
```

Für Mengen von numerischen Typen können wir die Summe und das Produkt der Typen definieren:

Tree.lhs

```
> summe = reduce 0 (+)
>
> produkt = reduce 1 (*)
```

Wende eine Funktion auf alle Elemente an. Danach resultierende Baum ist anders als bei der Funktion `tmap` dann allerdings kein binärer Suchbaum mehr.

Tree.lhs

```
> mappe _ Empty = Empty
> mappe f (Branch l el r) = Branch (mappe f l)(f el)(mappe f r)
```

Eine kombinierte Funktion von `reduce` und `mappe`:

Tree.lhs

```
> mapReduce f s op Empty = s
> mapReduce f s op (Branch l el r)
>   = (f el) `op`  (mapReduce f s op l) `op` (mapReduce f s op r)
```

Eine zweite Version dieser Funktion. Aber es wird hier nicht neu implementiert, sondern die beiden Funktionen aufgerufen. Der Nachteil ist, dass dann erst ein kompletter temporärer Baum erzeugt wird, um dann dessen Elemente mit einer Funktion zu verknüpfen:

Tree.lhs

```
> mapReduce2 f s op tree = reduce s op $ mappe f tree
```

Eine Alternative für die Baumgröße, in der erst ein Baum mit lauter Elementen der Zahl 1 erzeugt wird und diese dann aufaddiert werden:

Tree.lhs

```
> size2 = \t -> summe $ mappe (\x->1) t
```

Dieses Funktion mit mapReduce umgesetzt:

Tree.lhs

```
> size3 = mapReduce (\_->1) 0 (+)
```

Ist ein Element mit einer bestimmten Eigenschaft enthalten?

Tree.lhs

```
> containsWithProp p tree = reduce False (||) $ mappe (\x-> p x) tree
```

Zum Beispiel, ob eine Liste mit mehr als 10 Elemente in der Liste enthalten ist.

Tree.lhs

```
> haslongelement :: Tree [a] -> Bool
> haslongelement = containsWithProp (\x->length x > 10)
```

5 Lernzuwachs

- Eine polymorphe (generische) Datenstruktur als sogenannter algebraischer Datentyp.
- Pattern Matching für eine Datenstruktur.
- Beispiele für Infix-Operatoren.
- Guards für die Fallunterscheidung.
- Einführung lokaler Funktionen in einer `where`-Klausel.
- Verwendung der Standardtypklassen `Eq` und `Ord` für eigene Datenstruktur.

algebraischer Datentyp
Pattern Matching
Polymorphie
Higher Order