Universität Bremen

Praktische Informatik 3 WS 2023/24

Thomas Barkowsky
Tede von Knorre
Yaroslav Purgin
Tim Sperling

9. Übungsblatt

Ausgabe: 15.01.2024
Abgabe: 29.01.2024

Nachdem ihr nun schon in Haskell mit Abbildungen gearbeitet habt, wollt ihr euch daran versuchen, Mengen zu implementieren.

Zuerst brauchen wir dafür die Spezifikationen für die Operationen, die man auf einer Menge anwenden kann. Um die Spezifikation von der Implementation abzukoppeln, definieren wir das Interface als Typklasse Set:

```
class Set s where
  empty :: s a
  elem :: Ord a ⇒ a → s a → Bool
  insert :: Ord a ⇒ a → s a → s a
  remove :: Ord a ⇒ a → s a → s a
  size :: s a → Int
  toList :: s a → [a]
  union :: Ord a ⇒ s a → s a → s a
  intersection :: Ord a ⇒ s a → s a → s a
  difference :: Ord a ⇒ s a → s a → s a
```

Eine simple Beispielimplementierung ist im `WrongSet` Module vorgegeben. Wie der Name es schon verrät, ist diese Implementierung fehlerhaft. Nun müsst ihr sowohl selber eine richtige (und etwas performantere) Mengenimplementierung schreiben (9.2) als auch Tests schreiben, die zeigen dass `WrongSet` falsch und euer `RightSet` richtig implementiert sind (9.1). Nachdem ihr die Tests geschrieben habt, solltet ihr auch `WrongSet` korrigieren, so dass am Ende alle Tests durchlaufen (9.3).

Da 9.1 und 9.2 nicht aufeinander aufbauen, könnt ihr beide auch parallel implementieren. Allerdings wird es irgendwann sinnvoll sein, mit Hilfe der Tests eure Implementierung zu überprüfen.

9.1 *Mengen testen* — *9 Punkte*

Weil ihr es gerade in der Vorlesung hattet, wollt ihr die Tests mit QuickCheck in `src/Tests.hs` machen.

(Bitte lest die Hinweise unten bevor ihr euch den Kopf über die einzelnen Punkte zerbrecht.)

Hierfür brauchen wir Eigenschaften, die die Menge haben soll. Wir haben drei Wege, Informationen über die Menge zu bekommen: Die Größe, ob zwei Mengen gleich sind und ob ein Element in einer Menge enthalten ist.

Zu der Größe haben wie die Eigenschaften, dass eine leere Menge die Größe 0 hat, dass das Hinzufügen eines Elementes, das nicht in der Menge ist, die Größe um 1 erhöht, und dass das Entfernen eines Elementes, das in der Menge ist, die Größe um 1 verringert. Das mit der leeren Menge ist hier ein Spezialfall, bei dem ihr keine Mengen generieren lassen müsst, weil es ja nur eine leere Menge pro Implementierung gibt und die als Parameter übergeben wird.

Der Fall, dass das Element schon enthalten bzw. noch nicht enthalten ist, so dass sich die Größe nicht ändert, ist Teil einer anderen Eigenschaft: Zwei Änderungen an der gleichen Stelle ergeben die gleiche Menge wie nur die zweite Änderung. Ähnlich dazu ist die Eigenschaft, dass die Reihenfolge von Änderungen an unterschiedlichen Elementen keine Rolle spielt. Beispielsweise ergibt `insert 1 ∘ insert 2` das gleiche wie `insert 2 ∘ insert 1`. Tipp: Diese zwei Eigenschaften gelten ähnlich auch in Abbildungen, wie in der Vorlesung gezeigt.

Das waren Eigenschaften, die mit der Größe oder der Gleichheit arbeiten. Nun geht es mit den Eigenschaften weiter, die `elem` nutzen. Es gibt keinen Wert, der Element der leeren Menge ist; falls ein Wert mit `insert` eingefügt wurde, ist er danach Element der Menge; und falls ein Wert aus der Menge entfernt wird, ist er danach kein Element der Menge. Zu diesen drei relativ offensichtlichen Eigenschaften kommt noch hinzu, dass eine Änderung an einem Element der Menge keine Auswirkung auf das Enthaltensein eines anderen Elements hat. Wenn man beispielsweise 2 hinzufügt, ist nicht plötzlich 3 kein Element der Menge mehr.

Jetzt fehlen nur noch die Eigenschaften zu den drei Mengenoperationen `union`, `intersection`, und `difference`. Hierbei beschränken wir uns auf Tests mit `elem` und lassen `size` erst mal außen vor. Bei der Vereinigung ist es so, dass alle Elemente der Ergebnismenge Element der ersten oder der zweiten Eingabemenge sein müssen. Das alleine reicht aber

noch nicht, denn das würde auch zutreffen, falls Elemente der Eingabemengen weggelassen werden. Also muss noch die umgekehrte Richtung getestet werden: Alle Elemente der Eingabemengen sind Elemente der Ausgabemengen. Die Eigenschaftspaare der anderen zwei Operationen sind ähnlich: intersection erfordert, dass alle Elemente der Ausgabemenge in beiden Eingabemengen sind und umgekehrt, dass alle Elemente, die in beiden Eingabemengen sind, in der Ausgabemenge sind. difference erfordert, dass alle Elemente der Ausgabemenge in der ersten, aber nicht in der zweiten Eingabemenge sind, und umgekehrt, dass alle Elemente, die in der ersten, aber nicht der zweiten Eingabemenge sind, Element der Ausgabemenge sind.

Hinweise für diese Aufgabe:

Da einige Funktionen von Set auch in Prelude definiert sind, wird Set in `Tests.hs` mit dem **qualified** Schlüsselwort importiert. Dadurch müsst ihr S.foo nutzen, wenn ihr die Funktion foo für Mengen nutzen wollt.

Damit ihr die Tests nur einmal schreiben müsst und sie dann für beide Mengen genutzt werden können, sind die Funktionen für die Eigenschaften mit polymorphischen Signaturen versehen. Damit QuickCheck trotzdem weiß, von welchen konkreten Typen es die Mengen generieren soll, gibt es forAllSet. In der exampleForAllSet Funktion wird ein Beispiel für die Nutzung gezeigt: hinter dem forAllSet gibt man das leere Set an, das mit dem Parameter übergeben wurde, um den Typ festzulegen. Dahinter folgt `$ \set ... ->`, damit QuickCheck die Menge und potentiell weitere Parameter generieren kann. Diesen Stil solltet ihr überall nutzen, wo ihr eine zufällig generierte Menge braucht. Falls ihr das nicht braucht, kann die Funktion property nützlich sein.

Um ähnlich zu der Vorlesung die Anzahl der Kombinationen bei manchen Eigenschaften zu reduzieren, gibt es die Hilfsfunktion update, die je nachdem ob der erste Parameter True oder False ist, insert oder remove ausführt.

Falls ihr wissen wollt, wie genau das Set innerhalb des Testes aussah, ist counterexample eine nützliche Funktion. Hierfür ist auch ein Beispiel in exampleForAllSet.

Falls euch interessiert, wie das Generieren oder Verkleinern von Sets funktioniert, könnt ihr euch den Abschnitt unter der forAllSet Funktion angucken.

9.2 *Eine Menge implementieren* *7 Punkte*

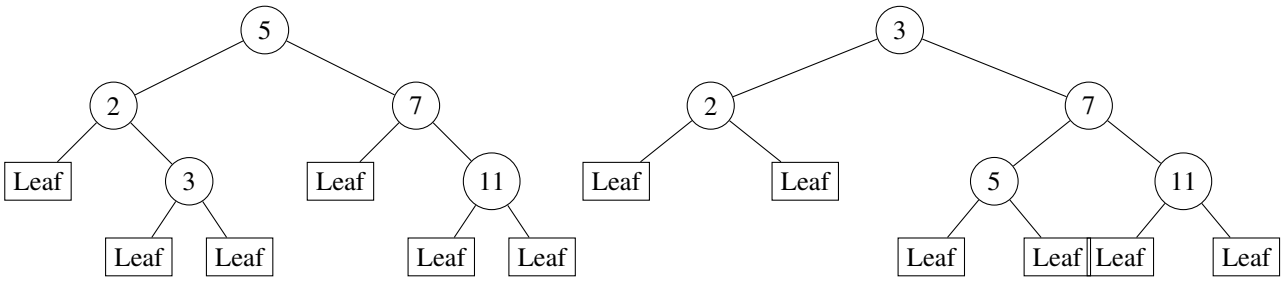
Mengen können unterschiedlich implementiert werden.

Am simpelsten, aber ineffizientesten, ist es, eine Liste zu verwenden, die man bei jeder Abfrage mit == oder elem prüft. Dies hat offensichtlich eine Laufzeitkomplexität von O(n), was zwar bei kleineren Datenmengen kein Problem ist, aber ihr wollt eine Menge implementieren, die schneller ist.

Aus Java kennt ihr wahrscheinlich schon das `HashSet`, das mit Hashcodes und einem Array arbeitet. Um das nutzen zu können, muss man aber erst eine Funktion definieren, die Hashcodes berechnet. Außerdem benötigt man eine Strategie für kollidierende Hashcodes.

Eine simplere Möglichekeit ist es, einen Binärbaum und eine Ordnung auf den Elementen zu nutzen. Hierfür definieren wir in `RightSet.hs` den Datentyp **data** RightSet a = Node a (RightSet a) (RightSet a) | Leaf. Dabei ist a das Element, das erste (RightSet a) enthält Elemente die kleiner sind, und das zweite (RightSet a) Elemente die größer sind.

Als Beispiel nehmen wir eine Menge mit den Zahlen 2, 3, 5, 7 und 11. In der folgenden Grafik werden die Nodes und Leafs für zwei mögliche Bäume mit diesen Werten dargestellt.



Damit muss man beim Überprüfen, ob ein Element in der Menge enthalten ist, in jedem Schritt nur prüfen, ob der aktuelle Knoten leer, kleiner, gleich oder größer ist. In Fall 1 und 3 ist die Suche direkt beendet, bei Fall 2 und 4 muss der jeweilige Teilbaum rekursiv weiter durchsucht werden.

Nun zu den Funktionen:

empty erzeugt eine leere Menge, was hier ein Blatt ist. elem x set prüft analog zu Listen, ob ein Element x in der Menge

`set` enthalten ist.

Mit `insert x set` und `remove y set` können Elemente hinzugefügt und entfernt werden. Falls bei `insert` das Element schon enthalten ist oder bei `remove` das Element nicht enthalten ist, wird die Menge nicht verändert. Bei `insert` müsst ihr nur darauf achten, das Element an eine freie Stelle (also an ein Blatt) zu packen. Bei `remove` hingegen kann es vorkommen, dass ihr den Baum darunter verändern müsst, um die Sortierung im Baum beizubehalten.

`size` berechnet rekursiv die Größe der Menge. Eine leere Menge, also ein `Leaf`, enthält keine Elemente, hat also die Größe 0. Ein `Node` hingegen hat sein eigenes Element plus die Elemente des linken und rechten Teilbaumes.

Mit `toList` kann man eine Menge in eine Liste umwandeln, die alle Elemente der Menge enthält. Als Reihenfolge bietet sich bei dieser Implementierung eine aufsteigende Liste an, aber die Reihenfolge kann auch anders sein.

`union`, `intersection`, und `difference` berechnen die Vereinigung, die Schnittmenge, und die Differenzmenge von 2 Mengen analog zu `union`, `intersect` und `\\` für Listen. Da das Nutzen der Funktionen der Standardbibliothek aber eine Laufzeitkomplexität von mindestens $O(n^2)$ hat, können wir (dürft ihr) diese Funktionen hier **nicht** nutzen, sondern müsst sie selber implementieren.

Was man nicht vergessen sollte, ist auch die Instanz von `Eq`, damit in den Tests die Mengen auch verglichen werden können. Eine einfache strukturelle Gleichheit ist hier keine Option, da, wie in der Grafik zu sehen, unterschiedliche Bäume gleiche Mengen repräsentieren können.

9.3 *WrongSet korrigieren* — *4 Punkte*

Da ihr nun getestet habt, was am `WrongSet` so falsch ist, könnt ihr es berichtigen. Behebt alle Fehler, die eure Tests anzeigen, ohne den grundlegenden Datentypen zu ändern.

Allerdings gibt es dort auch Implementierungsfehler, die nicht durch die oben beschriebenen Eigenschaften aufgedeckt werden, also guckt etwas genauer hin oder überlegt, welche Eigenschaften fehlen.

Insgesamt sollten dort 8 Fehler versteckt sein.