Universität Bremen

Praktische Informatik 3 WS 22/23

Christoph Lüth
Raphael Baass
Thomas Barkowsky
Tede von Knorre
Alexander Krug
Tarek Soliman

6. Übungsblatt

Ausgabe: 28.11.2022
Abgabe: 04.12.2022

In dieser Übung wollen wir unsere Haskell-Kenntnisse in den Dienst der Analyse von ÖPNV-Netzwerken (U-Bahn, Straßenbahn, Bus, etc.) stellen. Solche Netzwerke bestehen aus Stationen und Linien mit einer eindeutigen ID-Nummer:

```
type StationID = Int
type LineID    = Int
```

Das gesamte Netzwerk können wir nun als Liste von Tripeln repräsentieren, die angeben, welche Stationspaare auf welcher Linie miteinander verbunden sind:

```
type TNet = [(StationID, LineID, StationID)]
```

Hierbei handelt es sich um einen einen *ungerichteten Graphen*. Im Unterschied zu einem gerichteten Graphen, wie wir ihn aus dem vorherigen Übungsblatt kennen, ist hier eine Kante `(m, e, n)` sowohl eine Verbindung von `m` nach `n` als auch von `n` nach `m`. Wir wollen diesen Graphen in einen Baum umwandeln, der für einen gegebenen Startknoten die von dort erreichbaren Stationen darstellt. Hierfür modellieren wir zunächst eine Station:

```
data Station = Station { station_id    :: StationID
                       , station_name  :: String
                       , station_lines :: [LineID] }
```

Nun können wir unseren Baum für das Netzwerk definieren:

```
data NetTree a = NT {entry :: a, connections :: [NetTree a]} deriving Show
```

Es handelt sich hierbei um einen *variadischen* Baum, bei dem jeder Knoten mit mehreren weiteren Knoten direkt verbunden sein kann. Wir benötigen hier nur einen Konstruktor: wenn ein Knoten keine Folgeknoten besitzt (Blatt), dann ist die Liste der Verbindungen leer.

6.1 *Der Netz-Baum* — *4 Punkte*

Jetzt können wir unter Verwendung einer Liste von Verbindungen unseren Baum befüllen. Im Folgenden kümmern wir uns zunächst nicht um die Namen von Stationen und um die Linien; das machen wir später. Im Moment arbeiten wir hierfür mit leeren Strings bzw. der leeren Liste.

Eine Reihe von Hilfsfunktionen können wir jetzt gut gebrauchen:

```
startStation :: Station → NetTree Station
```

wandelt eine Station in die Wurzel unseres Baums um:

```
startStation (Station 7 "" []) ⇝
    NT {entry = Station {station_id = 7, station_name = "", station_lines = []},
        connections = []}
```

Die Funktion

```
stationConnectsTo :: StationID → TNet→ [StationID]
```

liefert aus einer Verbindungsliste die IDs derjenigen Stationen, die von der gegebenen Station (ID) direkt erreichbar sind:

```
stationConnectsTo 2 smallNet ⇝ [1,3,11,13]
```

mit

```
smallNet = [(1,1,2),(2,1, 3),(3,1,4),(11,2,2),(2,2,13)] :: TNet
```

Die Funktion

```
enterConnections :: [StationID] → NetTree Station → NetTree Station
```

fügt die Stationen mit den angegebenen IDs an der Wurzel eines Netzbaums hinzu:

```
enterConnections [2,4] (startStation (Station 3 "" [])) ⇝
NT {entry = Station {station_id = 3, station_name = "", station_lines = []}, connections =
[NT {entry = Station {station_id = 2, station_name = "", station_lines = []}, connections = []}
,NT {entry = Station {station_id = 4, station_name = "", station_lines = []}, connections = []}
]}
```

Jetzt können wir den eigentlichen Aufbau eines kompletten Netz-Baums in Angriff nehmen und schreiben dafür die Funktion:

```
enterConnectionsFromTNet :: TNet →  NetTree Station → NetTree Station
```

Beispiel:

```
s1= enterConnectionsFromTNet smallNet (startStation (Station 2 "" [])) ⇝
NT {entry = Station {station_id = 2, station_name = "", station_lines = []},
connections = [NT {entry = Station {station_id = 1, station_name = "", station_lines = []},
connections = []},NT {entry = Station {station_id = 3, station_name = "", station_lines = []},
connections = [NT {entry = Station {station_id = 4, station_name = "", station_lines = []},
connections = []}]},NT {entry = Station {station_id = 11, station_name = "", station_lines =
[]}, connections = []},NT {entry = Station {station_id = 13, station_name = "",
station_lines = []}, connections = []}]}
```

Hier beginnen wir mit einem ausgewählten *Startknoten*, den wir zusammen mit der Liste *aller* Verbindungen übergeben. Der Aufbau unseres variadischen Baums erfolgt rekursiv: Für den Startknoten werden alle Verbindungsknoten eingetragen; danach muss dasselbe für alle Folgeknoten in der connections-Liste getan werden. Wichtig ist hierbei, dass jede Verbindung nur *einmal* eingetragen wird. Es empfiehlt sich, hierfür mindestens eine Hilfsfunktion zu schreiben, die neben dem jeweils aktuellen Baum auch die um die eingetragenen Knoten reduzierte Verbindungsliste zurückliefert.

6.2 *Linien und Namen von Bahnhöfen* *3 Punkte*

Jetzt wird es aber Zeit, dass wir den Stationen in unserem Baum echte Namen geben und die Information, auf welchen Linien sie jeweils liegen, hinzufügen. Das wollen wir natürlich elegant mit `fmap` machen, weshalb wir unseren Datentyp `NetTree` in die Typklasse `Functor` aufnehmen:

```
instance Functor NetTree where
  fmap = ...
```

Für die Eintragung der Linien an einer Station schreiben wir und eine Hilfsfunktion, die zu einer gegebenen Stationsnummer alle Linien dieser Station ermittelt:

```
linesOfANode :: StationID → TNet → [LineID]
```

Natürlich soll jede Linie nur je einmal in der Liste enthalten sein. Hier kann die Funktion `nub` aus `Data.List` nützlich sein. Beispiel:

```
linesOfANode 2 smallNet ⇝ [1,2]
```

Das eigentliche Hinzufügen der Linien (unter Verwendung von `fmap`) erledigt

```
addLinesToNodes :: NetTree Station → TNet → NetTree Station
```

Für das Einfügen der Namen benötigen wir ein entsprechendes Verzeichnis des Typs

```
type StationDir = [(StationID, String)]
```

Die Funktion

```
nameOfStation :: StationID → StationDir → String
```

liefert uns den passenden Namen zu einer Stations-ID. Ist keine Station mit der gegebenen ID vorhanden, soll ein Leerstring zurückgegeben werden:

```
nameOfStation 2 smallNetStations ⇝ "Piccadilly␣Circus"
nameOfStation 9 smallNetStations ⇝ ""
```

Die Funktion

```
addNamesToNodes :: NetTree Station → StationDir → NetTree Station
```

fügt dann die Namen der Stationen in unserem Netzbaum ein. Damit könnte unser obiges Beispielnetz nun vervollständigt werden:

```
s2= addNamesToNodes (addLinesToNodes s1 smallNet) smallNetStations ⇝
NT {entry = Station {station_id = 2, station_name = "Piccadilly␣Circus", station_lines = [1,2]},
connections = [NT {entry = Station {station_id = 1, station_name = "Green␣Park",
station_lines = [1]}, connections = []},...
```

6.3 *Falten des Netzbaums* *3 Punkte*

Unser Streckennetz-Baum ist nun vollständig. Wir können ihn also nun verwenden, um Fragen zu beantworten wie:

- Wie viele Stationen sind in unserem Netz-Baum enthalten?
- Welche Stationen liegen auf der Linie x?
- An welchen Stationen laufen die meisten Linien zusammen?

Derartige Aufgaben rufen natürlich nach der Möglichkeit, unsere Datenstruktur zu falten. Wir definieren uns also die passende Faltungsfunktion

```
foldNetTree :: (a → [b] → b) → NetTree a → b
```

Unter Verwendung von `foldNetTree` können wir nun die Funktion

```
numberOfStations :: NetTree Station → Int
```

schreiben. Für das obige Beispielnetz würden wir erhalten:

```
numberOfStations s2 ⇝ 6
```

Als nächstes bauen wir die Funktion

```
stationsOnLine :: LineID → NetTree Station → String
```

die uns einen String mit passenden Zeilenumbrüchen erzeugt:

```
putStr (stationsOnLine 1 s2) ⇝
Piccadilly Circus
Green Park
Leicester Square
Covent Garden
```

Die letzte Funktion für dieses Aufgabeblatt ermittelt für jede Station, wieviele Linien dort vorhanden sind:

```
stationsWithNumberOfLines :: NetTree Station → [(String,Int)]
```

Für unser kleines Beispielnetz erhalten wir:

```
stationsWithNumberOfLines s2 ⇝
[("Piccadilly␣Circus",2),("Green␣Park",1),("Leicester␣Square",1),("Covent␣Garden",1),
("Oxford␣Circus",1),("Charing␣Cross",1)]
```

Zum Testen und Spielen findet ihr in der Vorlage das komplette Londoner U-Bahn-Netz. Unter Verwendung der obigen Funktionen können wir hierin nun die *wirklich* großen Umsteigebahnhöfe ermitteln:

```
filter (> 4) $ stationsWithNumberOfLines s3 ⇝
[("King's␣Cross␣St.␣Pancras",6),("Baker␣Street",5)]
```