Praktische Informatik 3WS 04/05

4. Übungsblatt

Ausgabe: 13.12.04
Abgabe: 11/12.01.05

Christoph Lüth <cxl>
Shi Hui <shi>
Klaus Lüttich <luettich>
Wolfgang Machert <wmachert>
Christian Maeder <maeder>
Hennes Märtins <maertins>
Kai-Florian Richter <richter>
Dennis Walter <dw>
Diedrich Wolter <dwolter>

8 *Endlich!* — *8 Punkte*

Abstrakte Datentypen sind ein grundlegendes Konzept der Informatik, und Informatik ist eine Schlüsseltechnologie in der modernen Wirtschaft. Leider sagen Wirtschaftsexperten der OECD in einigen Jahren einen entscheidenden Mangel an abstrakten Datentypen voraus. Um unseren Wohlstand auch für zukünftige Generationen zu sichern, müssen wir daher den Nachschub an abstrakten Datentypen sicherstellen.

Ein bekannter abstrakter Datentyp sind *endliche Abbildungen*, auf gut Altdeutsch auch als *finite maps* oder *lookup tables* bekannt, die wir in dieser Aufgabe implementieren werden, um unseren Teil beizutragen. Sie finden z.B. Verwendung im Compilerbau, wo sie als Symboltabellen benutzt werden. Ein endliche Abbildung ist parametrisiert über einem Definitionsbereich (*domain*) `a` und einem Wertebereich (*codomain*) `b`, und ordnetet jedem Wert aus `a` höchstens einen Wert aus `b` zu (d.h. die Abbildung kann auch undefiniert sein — sie ist *partiell*). Als der *Graph* einer endlichen Abbildung wird die Liste `[(a, b)]` aus Paaren von `a` und dem zugeordneten Wert aus `b` bezeichnet. Es kann davon ausgegangen werden, dass auf dem Definitionsbereich `a` eine Ordnung definiert ist (`a` ist in der Typklasse `Ord`).

Erstellen Sie zuerst die *Signatur* des abstrakten Datentyps. Wir wollen folgende Operationen bereitstellen (die Namen müssen evtl. angepasst werden, damit sie nicht mit vordefinierten Funktionen kollidieren):

- die leere Abbildung (`empty`);
- den Test auf die leere Abbildung (`isEmpty`);
- eine Abbildung um die Zuordnung eines `a` zu einem `b` erweitern (`add`; eine evtl. vorhandene vorherige Zuordnung wird überschrieben);
- die Abbildung auf `a` anwenden (ergibt ein `Maybe b`);
- prüfen, ob ein Element `a` abgebildet wird (`elem`);
- einen Wert `a` aus der Abbildung löschen (`delete`);
- die Anzahl der Werte in der Abbildung (`size`);
- den Graph der Abbildung (eine Liste `[(a, b)]`) erzeugen (`mapToList`);
- aus dem Graphen eine Abbildung erzeugen (`listToMap`);

- zu einer Abbildung eine zweite hinzufügen (`addMap`), wobei die Zuordnung in der zweiten Abbildung Priorität haben sollen;
- zwei Abbildungen zusammenfügen (`merge`), dabei soll eine Funktion `b-> b-> b` als Parameter übergeben werden, die die Werte kombiniert, wenn sie in beiden Abbildungen definiert ist;
- ein Funktional `fold`, welches gegeben eine Funktion `a-> b-> c-> c` und einen Startwert `c`, über die Abbildung iteriert und einen Wert vom Typ `c` zurückgibt;
- ein Funktional `map` welches, gegeben eine Funktion `b-> c` eine Abbildung von `a` nach `b` in eine Abbildung von `a` nach `c` überführt.

Danach *implementieren* Sie die endlichen Abbildungen mit denen in der Vorlesung vorgestellten ausgeglichenen Bäumen. Sie können dabei den Quellcode des Moduls `Set` als Ausgangspunkt nehmen.

9 *Oh Du Fröhliche!* *12 Punkte*

Bald ist ja nun Weihnachten, und der gute alte Weihnachtsmann hat wie immer ein Problem: das ganze Jahr nichts zu tun, und nun soll er überall gleichzeitig sein.

Damit er uns dieses Mal überaus reichlich beschenkt, wollen wir dem Weihnachtsmann etwas zur Hand gehen, und einen Algorithmus implementieren, der eine möglichst gute (d.h. möglichst kurze) Route zwischen allen guten Kindern (plus Studenten, Tutoren und Dozenten) berechnet.

Formal gesehen handelt es sich hierbei um das bekannte *Travelling Santa-Claus Problem (TSP)*. Abstrakt gesehen soll der Weihnachtsmann den optimalen Weg durch eine Graphen finden, dessen Knoten die zu besuchenden Stationen sind, und dessen Kanten mit der Entfernung gewichtet sind. Wir suchen den kürzesten Hamilton-Zyklus, d.h. den kürzesten Zyklus in dem Graphen, der alle Knoten besucht.

Wir können uns dabei zu Nutze machen, dass in dem Graphen die *Dreiecksungleichung* gilt, d.h. der Weg zwischen zwei Knoten ist immer kürzer als der Weg über einen anderen Knoten. In diesem Fall läßt sich der kürzeste Zyklus mit guter Näherung wie folgt finden: man berechnet den minimalen aufspannenden Baum des Graphen, zählt die Knoten des Graphen in Präorder-Reihenfolge in diesem Baum auf, und findet den Zyklus, der die Knoten des Baumes in dieser Reihenfolge durchläuft. Dieser Weg ist zwar nicht optimal, aber höchstens doppelt so lang wie der optimale; das allgemeine Problem ist, wie wir wissen, NP-vollständig.

Die Berechnung des mimimal aufspannenden Baumes erfolgt durch den *Prim*-Algorithmus (benannt nach dem Erfinder der Primzahlen): wir benötigen dazu eine Vorrangwarteschlange (s.u.) mit (mindestens) den folgenden Operationen:

- leere Schlange erzeugen (`emptyQ`);
- auf leere Schlange testen (`isEmptyQ`);
- Element einfügen (`addQ`);
- minimales Element entnehmen (`getMin`);

- Priorität der Elemente verringern (`decreaseP`).

Wir können einen beliebigen Knoten als Wurzel des aufspannenden Baumes wählen. Zur Initialisierung werden alle Knoten des Graphen mit maximaler Priorität in die Warteschlange eingefügt, lediglich der Wurzelknoten erhält Priorität 0. Jetzt wird, solange die Vorrangwarteschlange nicht leer ist, aus der Vorrangwarteschlange das minimale Element entnommen. Dieses ist die nächste Station, und für alle Knoten der Vorrangwarteschlange wird die Priorität auf die Entfernung zwischen dem entnommenen Knoten und diesem Knoten verringert, wenn sie nicht schon kleiner ist.

Wir erhalten damit eine Liste der Knoten des Graphen in der gewünschten Reihenfolge. Da alle Knoten des Graphen verbunden sind, ist dieses genau der gesuchte Zyklus.

Die Implementation dieses Algorithmus erfolgt in drei Schritten:

1. Zuerst implementieren wir die Vorrangwarteschlangen als ein Modul `PQueue` mit den Operationen wie oben und der naheliegenden Signatur. Die Priorität ergibt sich durch eine Ordnung auf den Elementen, d.h. diese sind ein Element der Typklasse `Ord`. Die Funktion `getMin` soll ein Tupel aus dem minimalen Element, und der um dieses Element verringerten Warteschlange zurückgeben. Die Funktion `decreaseP` soll die Priorität nur verringern, d.h. wir übergeben eine Funktion welche auf jedes Element angewandt wird, und nur wenn das Ergebnis der Funktion in der definierten Ordung kleiner ist als das bisherige wird der Eintrag geändert.

   Die Implementation erfolgt als sortierte Liste. Das ist zwar im allgemeinen eine suboptimale Lösung, weil das Hinzufügen und die Änderung der Priorität linearen Aufwand $O(n)$ haben; mit Datenstrukturen wie binomialen Heaps kann der Aufwand hier auf logarithmisch $O(\log n)$ verringert werden. In diesem Fall amortisiert sich das, da wir sehr oft die Prioritäten ändern.

   Als sortierte Liste ist das minimale Element immer vorne. Entscheidend ist, dass wir jetzt die Funktion `decreaseP` effizient implementieren können: wir traversieren die geordnete Liste rekursiv, und fügen jedes Element mit entweder der neuen Priorität (falls diese sich verringert hat) oder der alten Priorität an die entsprechende Stelle in die Liste ein, so dass sie geordnet bleibt. (*4 Punkte*)

2. Ein Graph (`Graph a b`) ist parametrisiert über die Knoten `a` und die Gewichtung `b` der Kanten. Er kann repräsentiert werden als eine Datenstruktur, die die Liste der Knoten enthält sowie eine Funktion, die für jeweils zwei Knoten die Entfernung zurückgibt (da in unserem Fall alle zwei Knoten verbunden sind).

   Für solch einen Graphen implementieren wir eine Funktion

   ```
   walk :: Ord b=> Graph a b -> [a]
   ```

   die unter Benutzung der Vorrangwarteschlangen der vorherigen Teilaufgabe die Näherung des kürzesten Rundweges wie oben beschrieben berechnet und zurückgibt. (*4 Punkte*)

3. Die zu implementierende Hauptfunktion hat folgende Signatur:

```
type Longitude   = (Int, Int, Int)
type Latitude    = (Int, Int, Int)
type Coordinates = (Longitude, Latitude)
xmasTour :: [(String, Coordinates)] -> [String]
```

Wir erhalten die zu besuchenden Orte als eine Liste aus Namen und Koordinaten, bestehend aus Längen- und Breitenangabe in jeweils Grad, Minuten, Sekunden. Hierbei sind die Minuten und Sekunden immer positiv; negative Längengrade entsprechen westlicher Länge, und negative Breitengrade entsprechen südlicher Breite.

Um die Entfernung zwischen zwei Punkten zu berechnen, konvertieren wir zuerst die Angaben für Länge und Breite in Gradangaben in Radian. Für d Grad, m Minuten und s Sekunden ergibt sich in Radian

$$d_{rad} = d_{deg}\frac{\pi}{180}, d_{deg} = d + \frac{m}{60} + \frac{s}{3600}.$$

Für die eigentlich Entfernungsberechnung nehmen wir näherungshalber an, dass die Erde eine Kugel mit dem Radius $R = \frac{40003.2}{2\pi} = 6367\ km$ sei. Die Entfernung d von zwei Punkten p_1 und p_2 mit Breite δ_1, δ_2 und Länge ϕ_1, ϕ_2 (Gradangaben in Radian) berechnet sich mit der *Semiversusformel* (*haversine formula*):

$$d = 2\,R\,\sin^{-1}(\sqrt{\sin^2(\frac{\delta_2 - \delta_1}{2}) + \cos\delta_1 \cos\delta_2 \sin^2(\frac{\phi_2 - \phi_1}{2})}). \tag{1}$$

Jetzt können wir einen Graphen generieren, dessen Knoten die zu besuchenden Orte sind, und dessen Kanten jeweils die Entfernung zwischen zwei Orten sind. Da der Weihnachtsmann bekanntlich in einem von Rentieren gezogenen Schlitten durch die Luft fliegt, sind alle zwei Knoten des Graphen verbunden, und die Entfernung zwischen ihnen ist genau die Luftlinie, berechnet durch (1).

Für diesen Graphen berechnen wir die Näherung des kürzesten Weges durch die Funktion `walk` aus der vorherigen Teilaufgabe. Dieses ist die Route des Weihnachtsmannes für dieses Jahr. (*4 Punkte*)

Fröhliche Weihnachten und guten Rutsch!