Christoph Lüth <cxl>
Felix Beckwermert <foetus>
Michael Skibbe <mskibbe>
Dennis Walter <dw>
Rafael Trautmann <pirate>
Thomas Meyer <mclee>

4. Übungsblatt

Ausgabe: 02.12.2002
Bearbeitungszeit: Zwei Wochen

$\boxed{7}$ *Alles ist Endlich* — *5 Punkte*

Eine *endliche Abbildung* ist ein über zwei Typen A und B parametrisierter abstrakter Datentyp, der endlich viele Zuordnungen von Elementen von A auf Elemente von B beschreibt (d.h. eine Verallgemeinerung des in der Vorlesung vorgestellten ADT `Store`).

Beispiele für endliche Abbildungen sind z.B. die Symboltabellen eines Übersetzers, welches für jeden in dem Programm auftretenden Bezeichner Informationen wie Typ und Wert beinhaltet, oder ein Lexikon, welches jedes Wort einem erklärenden Eintrag zuordnet.

Implementieren Sie ein Modul `FiniteMap`, welches eine endliche Abbildung möglichst effizient beschreibt. Das Modul soll folgende Konstruktionen auf endlichen Abbildungen beschreiben:

- die leere endliche Abbildung `empty`;
- Hinzufügen einer Zuordnung von `a` zu `b`;
- Entfernen einer Zuordnung zu `a`;
- Finden eines zu `a` zugeordneten `b`, falls vorhanden.

Definieren Sie zuerst die Schnittstelle des Moduls, und geben Sie danach eine Implementation.

Hinweis: Eine endliche Abbildung von `A` nach `B` kann effizient durch ausgewogene Bäume, deren Knoten mit Elementen aus `A` und `B` dekoriert sind, implementiert werden.

In diesem Sinne bietet sich hier ein *refactoring* des in der Vorlesung vorgestellten Moduls `Set` an. Wie kann danach das Modul `Set` wiederum durch eine geeignete Instantiierung `FiniteMap` implementiert werden?

$\boxed{8}$ *Noblesse oblige* — *15 Punkte*

Graphen sind besonders zur Weihnachtszeit eine wichtige Datenstruktur, denn wie sonst soll der Weihnachstmann seine Routenplanung durchführen?

Daher beschäftigen wir uns in dieser Aufgabe mit der Modellierung von Graphen in Haskell.

Graphen kann man auf vielerlei Arten definieren. Hier soll ein Graph $\mathcal{G}$ gegeben sein als

$$\mathcal{G} = (V, \{E_{(v,w)}\}_{v,w \in V}),$$

das heißt ein Graph besteht aus

- einer Menge V von Knoten, und
- einer Familie von Mengen $E_{(v,w)}$ von Kanten zwischen den Knoten v und w.

Graphen sind also über die Knoten und Kanten parametrisiert. In Haskell werden die Knoten als endliche Menge modelliert, und die Kanten als endliche Abbildung vom Kreuzprodukt der Knoten in Mengen von Kanten. Zusätzlich sollen in dem Graphen die eingehenden und ausgehenden Kanten jedes Knoten separat gehalten werden, als endliche Abbildung von den Knoten auf eine endliche Menge von Knoten und Kanten (wobei bei den ausgehenden Kanten die Menge von Knoten und Kanten die ausgehende Kante zusammen mit dem Zielknoten der Kante ist, und entsprechend bei den eingehenden Kante der Startknoten der Kante). Die Operationen auf dem Graphen müssen sicherstellen, dass diese Daten konsistent bleiben.

Definieren Sie diese Datenstruktur in Haskell, und implementieren Sie darauf folgende Funktionen:

- der leere Graph als Konstante;
- Test auf den leeren Graph;
- Hinzufügen eines Knoten;
- Hinzufügen einer Kante zwischen zwei Knoten;
- Löschen eines Knoten (dabei sollen auch alle ein- und ausgehenden Kanten gelöscht werden);
- Löschen einer Kante;
- Rückgabe der Menge der Knoten, der Menge der Kanten zwischen zwei Knoten, oder der Menge der aus- oder eingehenden Kanten (zusammen mit dem Ziel- bzw. Startknoten) eines Knoten.

Beachten Sie die Sonderfälle (z.B. was soll passieren, wenn Kanten zwischen nicht existierenden Knoten hinzugefügt oder gelöscht werden?)

Mit all diesen Funktionen definieren sie eine Funktion

```
transitive :: (Ord a, Ord b)=>
              (b-> b-> b)-> Graph a b-> Graph a b
```

die den *transitiven Abschluß* bezüglich der Funktion f des Graphen G berechnet. Dabei heißt $\mathcal{G} = (V, E)$ bezüglich f transitiv abgeschlossen, wenn für alle Knoten x, y, z und Kanten $e_1 \in E_{(x,y)}, e_2 \in E_{(y,z)}$ gilt, dass $f\ e_1\ e_2 \in E_{(x,z)}$; d.h. f ist eine Funktion, die für Knoten x, y, z und Kanten e_1 zwischen x und y und e_2 zwischen y und z eine neue Kante berechnet, und der Graph ist transitiv bezüglich f abgeschlossen, wenn es diese neu berechnete Kante zwischen x und z schon gibt. (Man beachte, dass `transitive` nicht immer terminieren muß.)

In diesem abstrakten Rahmenwerk können Sie jetzt eine Funktion

```
allShortestPaths :: Ord a=> Graph a Int-> Graph a Int
```

implementieren, die für einen Graphen mit beliebigen Knoten, aber ganzzahlwertigen Kanten den jeweils kürzeste Entfernung zwischen zwei Knoten berechnet, indem `transitive` mit einer geeigneten Funktion aufgerufen wird. Die Entfernung zwischen zwei Knoten x und y ist hierbei die Summe der Kanten auf dem Pfad zwischen x und y.

Genauer gesagt berechnet `allShortestPaths` einen Graphen, in dem für zwei Knoten x und y die kleinste Kante zwischen x und y (das minimale Element in $E_{(x,y)}$) die kürzeste Entfernung zwischen x und y angibt, wenn diese existiert; eine letzte Funktion

```
shortestPath :: Ord a=> Graph a Int-> (a, a)-> Maybe Int
```

berechnet unter Zuhilfenahme von `allShortestPaths` genau diesen Wert.

Dieses ist Version 1.0 von 2. Dezember 2002.