Praktische Informatik 3 WS 06/07

4. Übungsblatt

Ausgabe: 12.12.06
Abgabe: 09.01.07

Christoph Lüth <cxl>
Matthias Berger <tokio>
Klaus Hartke <hartke>
Cui Jian <ken>
Friederike Jolk <rikej>
Christian Maeder <maeder>
Diedrich Wolter <dwolter>

10 *Endliche Mappen* *5 Punkte*

Erweitern Sie den auf AVL-Bäuumen basierenden ADT `Set` aus der Vorlesung zu dem ADT `Map`, dessen Signatur am Ende der Vorlesung vorgestellt wurde, und der folgende Funktionen implementiert:

- `empty :: Map k a` erstellt die leere (überall undefinierte) Abbildung;
- `isEmpty :: Map k a-> Bool` prüft ob die Abbildung leer ist;
- `insert :: Ord k=> Map k a-> k-> a-> Map k a` trägt an dem Schlüsselwert den neuen Wert ein, und überschreibt den bisherigen Wert;
- `remove :: Ord k=> Map k a-> k-> Map k a` entfernt den Schlüsselwert aus der Abbildung, oder gibt die Abbildung unverändert zurück, falls sie den Schlüssel nicht enthält;
- `lookup :: Ord k=> Map k a-> k-> Maybe a` liest die Abbildung an der gegeben Stelle aus;
- `enum :: Ord k=> Map k a-> [(k, a)]` liefert den Graph der Abbildung (die Paare aus Schlüssel und Wert) zurück, aufsteigend sortiert nach den Schlüsselwerten.

11 *Das Haskell-Orakel* *10 Punkte*

Unser Textverarbeitungsprogramm (letztes Übungsblatt) ist ein großer 'Verkaufsschlager', und deshalb wollen wir jetzt den Erfolg konsolidieren (und den Bankrott abwenden, bevor das Wagniskapital zu Ende ist). Nach kurzer Überlegung wenden wir uns daher dem Datenbankbereich zu, der schon andere Firmen groß machte: in dieser Aufgabe soll eine funktional-objekt-relationale datawarehousing-Lösung für den backoffice-Bereich implementiert werden.

Eine *Datenbank* besteht aus einem *Datenmodell*, was das Format der Daten beschreibt, und den eigentlichen Daten. Die Daten bestehen aus einer *Tabelle*, welche *Einträge* (Zeilen) enthält, die aus einer Reihe von Datenfeldern (oder einfach nur Feldern) bestehen, von denen jeder einen Namen hat, und einen entweder alphanumerischen oder numerischer Wert. Das Datenmodell spezifiziert den Namen der Felder dieser Tabelle, sowie den Typ (numerisch oder alphanumerisch). Zusätzlich können Felder als *Schlüsselfelder* gekennzeichnet sein (das wird erst in der Bonusaufgabe X-1 relevant)

1. Implementieren Sie die Typen `Database`, `Datamodel`, `Entry` und `Value` sowie gegebenfalls weitere Datentypen, welche die Datenbank, das Datenmodell (für eine einzelne Tabelle), Einträge in der Datenbank sowie den Wert für einzelne Einträge beschreiben.

2. Um eine Datenbank aufzubauen, definieren wir (u.a.) folgende Funktionen:

```
data Answer a  = Error String | Ok a
                 deriving Show

create   :: Datamodel-> Answer Database
add      :: Database-> Entry-> Answer Database
remove   :: Database-> Entry-> Answer Database

entry    :: [(String, Value)]-> Entry
select   :: Entry-> String-> Maybe Value
```

   `create` legt eine neue, leere Datenbank mit dem angegeben Datenmodell an. Die Funktion muss prüfen, ob das Datenmodell konsistent ist, d.h. keine doppelten Einträge enthält.

   `add` fügt der Datenbank einen neuen Eintrag hinzu. Der Eintrag muss vorher auf Konsistenz mit dem Datenmodell geprüft werden, d.h. ob alle gegebenen Datenfelder im Datenmodell deklariert sind, und der Typ korrekt ist.

   `remove` löscht den Eintrag aus der angegebenen Datenbank.

   `entry` erzeugt einen Eintrag (aus Paaren von Namen und Wert), und `select` selektiert aus einem Eintrag den Wert eines Feldes.

3. In einer Datenbanken gibt es keine Zinsen, dafür kann man beliebig oft abheben, ohne dass sie leer wird. Mit einer *einfachen Abfrage* kann für alphanumerische Felder nach einem exakten Wert oder einem Substring gesucht weden, oder bei numerischen Feldern nach exakten Werten, Obergrenzen, Untergrenzen oder Werten innerhalb eines vorgegeben Intervalls. Eine *Abfrage* ist dann eine Verknüpfung von Abfragen mittels logischer Negation, Konjunktion, Disjunktion oder Äquivalenz. Das Ergebnis der Abfrage ist eine Liste von Tabellenzeilen, die die Abfrage erfüllen.

   Eine Abfrage soll modelliert werden durch einen algebraischen Datentyp `Query`. Implementieren Sie diesen zusammen mit der Funktion `query`, welche die Abfrage durchführt. Die Funktion `query` muss dabei prüfen, ob die Abfrage konsistent mit dem Datenmodell ist:

```
data Query
query :: Database -> Query-> Answer [Entry]
```

Das Ergebnis von `query` ist eine Liste von Tabelleneinträgen, welche die Abfragebedingung erfüllen.

In dieser Aufgabe ist es hinreichend, einen Prototyp der Datenbank zu implementieren, der noch nicht performanceoptimiert ist — das passiert dann in der Bonusaufgabe X-1. (*Hinweis:* eine naive Implementierung als Liste bietet sich an.)

12 *Es weihnachtet sehr...* *5 Punkte*

Bald ist Weihnachten, aber der Weihnachtsmann hat ein Problem: dieses Jahr sind nämlich die Säcke etwas knapp. Deshalb muss der Weihnachtsmann dieses Jahr die Geschenksäcke besonders effizient packen. Wenn man nämlich Geschenksäcke einfach nur der Reihe nach füllt, dann werden — weil ja die Geschenke unterschiedlich groß sein können — einige Säcke nur halbvoll. Deshalb müssen Sie dem Weihnachtsmann helfen, sonst sind eventuell die Säcke alle, und es gibt keine Geschenke mehr!

Das Problem stellt sich wie folgt dar: jedes Geschenk p hat einen Namen (irgendeine Bezeichnung), eine Größe (Volumen) $s(p)$, und ein Gewicht $w(p)$. Jeder Sack hat das gleiche Fassungsvermögen S. Zu finden ist jetzt aus einer Menge P von Geschenken eine Teilmenge $\{p_1, \ldots, p_n\} \subseteq P$, deren Gesamtvolumen kleiner oder gleich dem Volumen des Sackes ist

$$\sum_{i=1}^{n} s(p_i) \leq S$$

aber deren Gesamtgewicht optimal ist, d.h. alle anderen Mengen von Geschenken, die auch in den Sack passen würden, haben weniger (oder gleich viel) Gewicht.

In Haskell ausgedrückt, modellieren Sie zuerst den Datentyp `Present`, der Geschenke repräsentiert, sowie die Funktionen

```
data Present
name   :: Present -> String
size   :: Present -> Double
weight :: Present -> Double
```

Die eigentliche Packfunktion ist dann

```
pack :: [Present]-> Double-> [Present]
```

die aus einer Menge von Geschenken für ein gegebenes Sackvolumen die in Hinsicht auf das Geschenkgewicht optimale Bepackung auswählt.

X1 *Das Haskell-Orakel II* *5 Bonuspunkte*

In dieser Bonusaufgabe wird eine optimierte Retail-Version der Datenbank aus Aufgabe 11 implementiert. Der wesentliche Unterschied ist hier, dass wir die Schlüsselfelder nutzen, um einen schnellen Zugriff auf die Daten zu erlauben.

Dazu nutzen wir die in der Aufgabe 10 implementieren endlichen Abbildungen `Map k a`. Für jedes deklarierte Schlüsselfeld definieren wir jetzt eine Abbildung von Schlüsselwerten auf alle Zeilen, die für diesen Eintrag genau diesen Schlüsselwert enthalten. Wenn mehrere Schlüsselwerte deklariert sind, entspricht das mehreren Abbildungen — die Funktionen `add` und `remove` müssen diese Abbildungen dann konsistent halten.

Die Optimierung soll die Schnittstelle und das Verhalten des Prototypen beibehalten, nur schneller sein.

Hinweis: Zur Realisierung können Sie auch den abstrakten Datentyp `Data.Map` aus der erweiterten Standardbücherei verwenden, der eine leicht andere Signatur und wesentlich mehr Funktionen bietet als `Map` aus Aufgabe 10.

Dies ist Version 1.1 vom 2006/12/12 16:04:37.