Universität Bremen

Praktische Informatik 3 WS 16/17

Christoph Lüth
Tobias Brandt
Tristan Bruns
Johannes Ganser
Berthold Hoffmann
Alexander Kurth

5. Übungsblatt

Ausgabe: 22.11.16
Abgabe: 02.12.16, 12:00 Uhr

5.1 *Binäre Suchbäume als Endliche Abbildungen* *10 Punkte*

Eine oft gebrauchte Datenstruktur in der Informatik sind *endliche Abbildungen* (auch als Assoziativlisten oder *dictionaries* bekannt) von einem Schlüsselbereich α auf einen Wertebereich β. Bisher haben wir diese in Haskell durch eine Liste $[(\alpha, \beta)]$ repräsentiert, was aber nicht sehr effizient ist. Deshalb wollen wir Maps durch binäre Suchbäume implementieren. Ausgangspunkt ist der algebraische Datentyp Map, parametrisiert über einem Schlüsseltyp α und einem Wertetyp β:

```
data Map α β = Branch (Map α β) α β (Map α β) | Leaf deriving Show
```

Die Invariante hier soll sein, dass für jede Map `Branch l k v r` alle Schlüssel in `l` kleiner, und alle Schlüssel in `r` größer als `k` sind. (Daraus folgt, dass jeder Schlüssel höchstens einmal in der Map enthalten sein darf.)

Um den Umgang mit Map zu erleichtern, sollen Sie zunächst Instanzen für die Typklassen Functor und Foldable für Map α und Eq für Map implementieren. Damit Map α eine Instanz von Functor werden kann, muss die Funktion fmap implementiert werden:

```
fmap :: Functor f ⇒ (α → β) → f α → f β
```

fmap soll dabei die übergebene Funktion auf alle Werte innerhalb von Map anwenden. Die Struktur der Map soll jedoch beibehalten werden.

Für die Typklasse Foldable muss die Funktion `foldr` implementiert werden. Diese soll die rechtsassoziative strukturelle Rekursion auf den Werten der Map darstellen.

```
foldr :: Foldable t ⇒ (α → β → β) → β → t α → β
```

Die Typklasse Eq fordert die Funktion (==). Zwei Maps sind gleich, wenn alle ihre Abbildungstupel gleich sind; die Struktur des internen Baumes soll keine Rolle spielen.

```
(==) :: Eq α → α → α → Bool
```

Jetzt implementieren wir die Funktionen auf dem Datentyp:[1]

```
empty    :: Map α β
insert   :: Ord α ⇒ α → β → Map α β → Map α β
delete   :: Ord α ⇒ α → Map α β → Map α β
lookup   :: Ord α ⇒ α → Map α β → Maybe β
size     :: Ord α ⇒ Map α β → Int
fromList :: Ord α ⇒ [(α, β)] → Map α β
toList   :: Ord α ⇒ Map α β → [(α, β)]
elems    :: Ord α ⇒ Map α β → [β]
```

- empty liefert die leere Map.
- insert fügt ein neues Abbildungspaar in die Map ein, und überschreibt ein vorhandenes.

[1] Die Benennung orientiert sich an dem Datentyp Data.Map aus der GHC-Standardbücherei. Um einen Namenskonflikt zwischen lookup und der gleichnamigen vordefinierten Funktion zu verhindern, verstecken Sie diese mit **import** Prelude **hiding** (lookup) am Anfang ihres Moduls. Nächste Woche lernen wir, wie wir mit solchen Namenskonflikten strukturiert umgehen.

- `lookup` liefert anhand eines Schlüssels den Wert dieses Schlüssels innerhalb der Map. Sollte der Schlüssel nicht in der Map vorhanden sein, soll `Nothing` zurückgegeben werden.
- `delete` entfernt ein Element aus der Map anhand eines Schlüssels. Ist das Element nicht vorhanden, bleibt die Map unverändert.
- `size` liefert die Anzahl der Elemente in der Map.
- `fromList` erstellt eine neue Map anhand einer Liste von Schlüssel-Wert-Tupel, und `toList` liefert die Liste von Schlüssel-Wert-Tupeln einer Map (in den Schlüsseln aufsteigend sortiert).
- `elems` erstellt die Liste von Werten anhand einer Map.

Nutzen Sie dabei Funktionen höherer Ordnung wo immer sinnvoll.

5.2 *Toni liefert wieder...(erneut!)* *10 Punkte*

Gianluigi "Toni" Corleone hat wieder ein Problem. Sein Pizzaunternehmen *Pizza-Toni* kann jetzt schnell backen und ausliefern, aber das im zweiten Übungsblatt implementierte Lagerhaltungssystem ist einfach zu ineffizient (neulich hat Tante Guilia zwei Stunden gebraucht, um die Oliven zu finden). Deshalb hat Toni Sie wieder um Hilfe gebeten: Sie sollen sein Lagersystem effizienter machen. Zum Glück können Sie jetzt auf die Abbildungen aus Aufgabe 5.1 zurückgreifen, und definieren folgende Typen:

```
data Artikel = Artikel { aName :: String, beschreibung :: String, aPreis :: Int } deriving (Eq, Show)
data Pizza = Pizza { pName :: String, zutaten :: [String] } deriving (Eq, Show)
type Katalog = Map String Artikel
type Lager = Map String Int
type Speisekarte = Map String Pizza
```

- `Artikel` repräsentiert eine Pizza-Zutat, mit einem Namen, einer Beschreibung und einem Preis.
- `Pizza` repräsentiert eine Pizza, mit einem Namen und einer Liste von Zutaten.
- `Katalog` ist eine Abbildung von Zutatennamen auf eine Zutat. Ein Katalog enthält alle auf dem Markt verfügbaren Zutaten.
- `Lager` repräsentiert das Lager von *Pizza-Toni*, als Abbildung von Zutatenname auf den Lagerbestand.
- `Speisekarte` repräsentiert die Speisekarte von *Pizza-Toni*; also eine Zuordnung von Pizzaname auf Pizza.

Mit diesen bestehenden Typen sollen Sie nun das Lagersystem von Toni um folgende Funktionen erweitern:

```
pizzaHinzufuegen  :: String → [String] → Speisekarte → Speisekarte
preisZutaten      :: [String] → Katalog → Int
preisPizza        :: Pizza → Katalog → Int
zutatenVorhanden  :: Lager → Pizza → Bool
entnehmeZutat     :: String → Lager → Lager
entnehmeZutaten   :: Pizza → Lager → Lager
bestellen         :: [String] → Speisekarte → Lager → Katalog → Maybe (Lager, Int)
```

- `pizzaHinzufuegen` soll eine Pizza mit ihrem Namen und ihrer Zutaten in die Speisekarte einfügen.
- `preisZutaten` soll den Gesamtpreis der übergebenen Zutatennamen ermitteln.
- `preisPizza` gibt den Preis einer Pizza zurück. Wir nehmen dabei an, dass die Pizza genau so viel wie die Summe ihrer Zutatenpreise kostet.
- `zutatenVorhanden` überprüft, ob im Lager alle Zutaten für eine Pizza vorhanden sind. Dabei wird angenommen, dass jeder Eintrag in der Zutatenliste der Pizza eine Einheit der entsprechenden Zutat verbraucht.
- `entnehmeZutat` entnimmt eine Zutat, die durch ihren Namen übergeben wird, aus dem Lager.
- `entnehmeZutaten` entnimmt alle Zutaten, die zum Backen einer Pizza benötigt werden, aus dem Lager.
- `bestellen` soll alle Pizzen, die per Pizzanamen übergeben werden, backen. Unter der Voraussetzung, dass alle Zutaten für die Pizzen vorhanden sind, soll der neue Lagerstand kombiniert mit dem Gesamtpreis der Pizzen zurückgegeben werden.

Hinweise: Die folgenden Funktion können hilfreich sein:

```
sequence :: [Maybe a] → Maybe [a]
```

? *Verständnisfragen*

1. `foldr` ist die „kanonische einfach rekursive Funktion“ (Vorlesung). Was bedeutet das, und warum ist das so? Für welche Datentypen gilt das?
2. Wann kann `foldr f a xs` auch für ein zyklisches Argument `xs` (bspw. eine zyklische Liste) terminieren?
3. Warum sind endrekursive Funktionen im allgemeinen schneller als nicht-endrekursive Funktionen? Unter welchen Voraussetzungen kann ich eine Funktion in endrekursive Form überführen?