Till Mossakowski
Stephan Günther

4. Übungsblatt

Ausgabe: 2019-11-09
Abgabe: 2019-11-18

4.1 *Lagern will strukturiert sein*

In Anlehnung an "Uncle Bob's Auld-Time Grocery Shoppe's" Lager implementieren wir in dieser Aufgabe ein Lager für einen Essenslieferservice. Wir gehen dabei wie folgt vor:

1. In unserem Lager werden die eingelagerten Objekte als Artikel hinterlegt. Jeder Artikel besteht aus einer Beschreibung `a` und seiner Anzahl im Lager als `Natural` (wobei zwei Artikel gleich sind, wenn ihre Beschreibung gleich ist):

   ```
   data Article a = Article a Natural
   ```

   Das Lager besteht aus einer einfachen Liste mit Artikeln:

   ```
   type Storage a = [Article a]
   ```

   Produkte stellen Objekte dar, die aus Artikeln aus unserem Lager hergestellt werden können. Sie enthalten jeweils einen Namen und eine Liste mit den benötigten Zutaten in Form einer Liste von Artikeln. (Auch hier sind zwei Produkte gleich, wenn ihre Beschreibung (`name`) gleich ist.)

   ```
   data Product a b = Product {name :: b, ingredients :: [Article a]}
   ```

   Die wesentlichen Funktionen unseres Lagers werden das Hinzufügen und Entfernen von Artikeln sein, sowie die Überprüfung, wie viel von einem bestimmten Artikel vorhanden ist. Implementieren Sie hierzu die folgenden Funktionen:

   ```
   contains :: Eq a ⇒ Storage a → a → Natural
   store :: Eq a ⇒ Storage a → a → Natural → Storage a
   remove :: Eq a ⇒ Storage a → a → Natural → Storage a
   ```

2. Implementieren Sie nun eine Funktion die mehre Produktionsaufträge gleichzeitig Verarbeitet. Implementieren Sie hierzu die Funktionen:

   ```
   ingredients_available :: (Eq a, Eq b) ⇒ Product a b → Storage a → Bool
   remove_product :: (Eq a, Eq b) ⇒ Product a b → Storage a → Storage a
   ```

   Dabei überprüft `ingredients_available`, ob sich im Lager alle Artikel in ausreichender Anzahl für das übergebene Produkt befinden und `remove_product` entfernt alle Artikel eines Produkts aus dem Lager.

   Implementieren Sie nun die Funktion

   ```
   produce :: (Eq a, Eq b) ⇒ [Product a b] → Storage a → Storage a
   ```

   die `ingredients_available` und `remove_product` für eine Liste von Produkten nutzt, und dabei die Artikel von allen Produkten aus dem Lager entfernen, für die sie vorhanden sind.

Hinweis: um den Datentyp `Natural` nutzen zu können, müssen Sie das Modul `Numeric.Natural` via

```
import Numeric.Natural
```

importieren.

4.2 *Polymorphism Quirks*

(1) Evaluieren Sie folgende Ausdrücke in `ghci`:

```
"" == []
tail [1] == ""
tail [1] == []
```

Erläutern Sie die Ergebnisse der Ausdrücke und erklären Sie insbesondere die jeweiligen Unterschiede und Gemeinsamkeiten.

(2) In der Vorlesung wurde die Funktion `read` aus der Typklasse `Read` vorgestellt. Diese Funktion stellt in gewisser Weise das Inverse zu `show` aus der Typklasse `Show` dar. Evaluieren Sie folgende Ausdrücke in `ghci`:

```
read "10" == 10
read "10"
```

Erläutern Sie das Resultat der beiden Ausdrücke. Fällt Ihnen eine Möglichkeit ein im `ghci` mit Hilfe des Ausdruckes `read "10"` ohne einen Fehler zu Produzieren den Wert `10` zu erzeugen? (D.h.: Fällt Ihnen eine Zeile Code ein, die mit `let x =` beginnt, in der auf der Rechten Seite des = unter Anderem `read "10"` steht und nach deren Auswertung `x` zu `10` evaluiert?)

4.3 *Stringspalterei*

In der Vorlesung wurde die Funktion `splitAt` aus dem `Prelude` erwähnt, die einen String in ein Paar aus Strings aufspaltet. Hierbei besteht die erste Komponente des Ergebnispaars aus dem Substring vom Anfang des Arguments bis zum letzten Zeichen vor dem gegebenen Index (0-basiert), d.h. zum Beispiel:

```
splitAt 10 "Praktische␣Informatik" == ("Praktische" ,"␣Informatik")
```

Schreiben Sie eine eigene Implementierung der Funktion `splitAt`. Verallgemeinern sie die Implementierung danach auf multiple Eingabeindizes. Schreiben Sie also eine Funktion

```
splitAtMultiple :: [Int] → String → [String]
```

die den gegebenen String an allen gegebenen Indizes aufspaltet. D.h.:

```
length(splitAtMultiple xs s) == length(xs) +1
concat(splitAtMultiple xs s) == s
splitAtMultiple [1, 3, 5] "foobar" == ["f", "oo", "ba", r]
```

4.4 * *Hilfreiche Typen: Funktionskomposition*

Das Haskell `Prelude` enthält einen Operator für Funktionskomposition, der, in Anlehnung an die mathematische Notation $\circ$ in z.B. $(f \circ g)(x) = f(g(x))$, einfach nur aus dem Punkt `.` besteht. In Haskell bedeutet `(f . g) x` also soviel wie `f (g x)`. Erläutern Sie den Typen `(.) :: (b -> c) -> (a -> b) -> a -> c` des Kompositionsoperators und geben Sie zwei kurze Beispiele für die Nutzung des Operators.

Der Typ des Kompositionsoperators sieht auf den ersten Blick danach aus, als würde der Operator nur auf einargumentigen Funktionen arbeiten. Dies ist aber nicht der Fall.

Evaluieren Sie in `ghci` folgende Ausdrücke:

```
let f = undefined :: b → c1 → c
:t (f .)
let g = undefined :: a → a1 → b
:t (. g)
```

und erläutern Sie die jeweiligen Resultate.