Aufgaben für die 2. Sitzung

20. August 2017

Entwurf von ADTs

(1) Bei Abstimmungen im Plenum gibt es Für-Stimmen, Gegen-Stimmen und Enthaltungen. Entwirf einen neuen Datentypen Vote, der das Abstimmungsverhalten einer Person bei einer Entscheidung darstellt.

Kennst du einen bereits bestehenden Datentypen, der zu Vote isomorph ist? Schreibe den Isomorphismus.

(2) Entwirf einen Datentypen Tree a für einen (unbalancierten) Binärbaum, der in den Blättern Werte vom Typ a enthält.

Schreibe im Anschluss eine Funktion traverse :: Tree a → [a], die von links nach rechts die Werte aus den Blättern holt.

(3) Du stehst in einer Fabrik vor einem Förderband. An dem Förderband sind nacheinander einige Maschinen angebracht. Jede der Maschinen ist in einem der folgenden Zustände:

- Sie ist angehalten.
- Sie gibt gerade eine fertige Ware vom Typ b aus. In diesem Fall ist auch bereits der Nachfolgezustand bekannt.
- Sie wartet gerade auf eine ankommende Ware vom Typen a. Sobald sie die Ware erhalten hat, wechselt sie in einen von der Ware abhängigen Nachfolgezustand.

Entwirf einen Datentypen Machine a b, der eine solche Maschine adäquat darstellt.

Vergleichbarkeit

(4) In manchen Typen sind Vergleiche definiert (z.B. Int, Bool), in anderen nicht (z.B. a → b). Hierzu gibt es die Typenklasse **Eq**:

```
class Eq a where
  (==) :: a → a → Bool

a /= b = not (a == b)
```

Implementiere die Instanz **Eq** a ⇒**Eq** [a].

(5) Da es mit Eq noch keinen Vergleich auf kleiner oder größer gibt, ist in der Standardbibliothek auch noch die Typenklasse **Ord**:

```
data Ordering = LT | EQ | GT deriving Eq

class Eq a ⇒ Ord a where
  compare :: a → a → Ordering

(<) :: Ord a ⇒ a → a → Bool
a < b = compare a b == LT
```

Die anderen Operatoren <=, > und >= sind analog definiert. Instanziiere Eq und Ord für Nat.

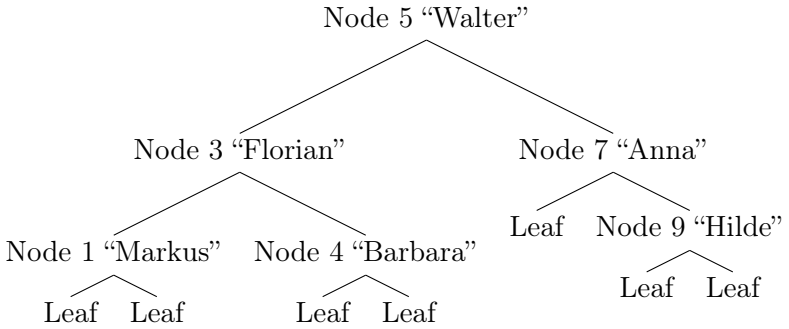
Wörterbücher

(6) Sei ListMap key value wie folgt definiert:

```
data ListMap key value = LM [(key, value)]
```

Implementiere die Funktion lmLookup :: ListMap key value →key → **Maybe** value. Musst du key auf eine Klasse einschränken? Auf welche?

(7) Entwirf einen ADT TreeMap key value als binären Suchbaum, d.h. als Binärbaum mit Schlüssel-Wert-Paaren in den inneren Knoten, wobei die Schlüssel im linken Teilbaum kleiner sind als der Schlüssel in der Wurzel, und die Schlüssel im rechten Teilbaum größer sind. Zum Beispiel ist der folgende Baum ein binärer Suchbaum:



(8) Implementiere tmLookup :: TreeMap key value →key →**Maybe** value. Musst du key auf eine Klasse einschränken? Auf welche?

Monoide und Funktoren

(9) Gibt es eine sinnvolle Monoid-Instanz für Vote? Wenn ja, implementiere sie.

Gibt es eine Functor-Instanz für Vote? Wenn ja, implementiere sie.

(10) Schreibe eine Monoid-Instanz für Tree a, sodass gilt:

- Assoziativgesetz: a <> (b <> c) === (a <> b) <> c
- Linearität der Traversierung: traverse (a <> b) === traverse a <> traverse b

(11) i. Schreibe eine Functor-Instanz für Tree a, sodass die Struktur des Baumes bei fmap nicht verändert wird.

ii. Schreibe eine Applicative-Instanz für Tree a.

(12) Schreibe eine Functor-Instanz für Machine a b, bei der mit fmap Funktionen auf die ausgegebenen Waren angewandt werden.

Warum können wir keine Functor-Instanz schreiben, die Funktionen auf die ankommenden Waren anwendet?

(13) Kannst du die folgenden Klassen für TreeMap k v instanziieren? Wenn ja, schreibe die Instanz. Wenn nein, was ist das Problem?

i. **Functor** (TreeMap k v)

ii. Applicative (TreeMap k v)

iii. Monoid (TreeMap k v)

iv. (**Eq** k, **Eq** v) ⇒**Eq** (TreeMap k v)

(14) Für Funktoren werden die folgenden (vom Compiler nicht prüfbaren) Regeln vorausgesetzt:

- Linearität bzgl. Komposition: fmap (f . g) === fmap f . fmap g
- Neutrales Element: fmap **id** === **id**

Konstruiere einen Datentypen und eine Functor-Instanz, die vom Compiler akzeptiert wird, aber mindestens eine der Regeln verletzt.

(15) Für Int gibt es mehrere sinnvolle Monoiden, z.B. mappend = (+), (∗), **min** oder **max**. Definiere zwei **newtype**s Sum und Product, und schreibe für beide die Monoid-Instanz.

(16) Zu jedem Monoid mit mappend gibt es ein duales Monoid mit **flip** mappend. Beispielsweise könnte statt [1,2,3] <> [4,5,6] = [1,2,3,4,5,6] mit einer alternativen Monoid-Instanz [1,2,3] <> [4,5,6] = [4,5,6,1,2,3] gelten.

Definiere den **newtype** Dual a und die zugehörige Monoid-Instanz.

(17) Betrachte den Typen a → b.

i. Schreibe eine sinnvolle Monoid-Instanz mit der Voraussetzung, dass bereits b ein Monoid ist.

ii. Schreibe eine sinnvolle Functor-Instanz.

iii. Schreibe eine sinnvolle Applicative-Instanz.

Besondere Abbildungen

(18) Ein Endomorphismus über a ist eine Abbildung aus dem Typen a in sich selbst, also eine Funktion f :: a → a. In der Standardbibliothek gibt es den Typen Endo a, der Endomorphismen kapselt und wie folgt definiert ist:

```
newtype Endo a = Endo { appEndo :: a → a }
```

newtype ist eine „light"-Version von **data**, mit der Einschränkung, dass der neue Typ nur einen Konstruktur haben darf, und dieser Konstruktor nur einen Parameter. Der neue Typ ist also isomorph zum Typen des Konstrukturparameters (Isomorphismus hier: Endo :: (a → a) → Endo a, mit Umkehrfunktion appEndo :: Endo a →a → a).

Kannst du eine Monoid-Instanz für Endo a schreiben? Wenn ja, tue dies. Wenn nein, warum nicht?

Kannst du eine Functor-Instanz für Endo a schreiben? Wenn ja, tue dies. Wenn nein, warum nicht?

(19) Eine Abbildung f :: a → b heißt Homomorphismus, falls a und b Monoide sind und gilt:

- Linearität: f (a <> b) = f a <> f b
- Neutrales Element: f mempty === mempty

Sind folgende Funktionen Homomorphismen?

- traverse :: Tree a -> [a]
- fmap :: Functor f => (a -> b) -> (f a -> f b) mit dem Monoid aus (17)
- fmap :: Functor f => (a -> a) -> (f a -> f a) mit dem Monoid aus (18)
- length :: [a] -> Int
- sum :: [Int] -> Int
- fib :: Int -> Int
- (*2) :: Int -> Int

Suche dir für Int eine Monoid-Instanz aus (15) aus, wie es dir beliebt.

(20) *Für Mathematiker*innen:*

- Inwiefern unterscheidet sich diese Homomorphismusdefinition von der aus der Algebra? Warum ist das sinnvoll?
- Inwiefern unterscheidet sich unsere Isomorphismusdefinition von der aus der Algebra? Warum ist das sinnvoll?

(21) FYI:

- Ein Endomorphismus, der gleichzeitig Isomorphismus ist, heißt Automorphismus.
- Ein Automorphismus, der identisch mit seiner Umkehrfunktion ist, heißt Involution.

- Eine Abbildung, die für jedes Argument definiert ist ($\neq \bot$), heißt total. Eine nicht totale Abbildung heißt partiell.
- Eine Abbildung f :: a → b heißt injektiv oder Injektion, falls aus f x == f y folgt, dass x == y.
- Eine Abbildung f :: a → b heißt surjektiv oder Surjektion, falls jedes Element des Zieltyps durch irgendein Argument erreicht werden kann.
- Eine partielle, nicht-injektive Funktion heißt Yolomorphismus[1].

Fallen dir Beispiele für Yolomorphismen ein? Warum können Yolomorphismen Probleme bereiten?

Numerische Typen

(22) Bisher haben wir als numerischen Typ grundsätzlich Int oder Double verwendet. Es gibt jedoch eine ganze Reihe weiterer numerischer Typen in der Standardbibliothek und die grundlegenden arithmetischen Funktionen sollen auf allen funktionieren. Dafür wurde die Typenklasse **Num** erfunden:

```
class Num a where
  (+), (-), (*) :: a → a → a
  negate, abs, signum :: a → a
  fromInteger :: Integer → a
```

Implementiere die Num-Instanz für Nat. Sollten die Werte unter Null fallen, wirf einfach einen Fehler, z.B. **error** "Arithmetic␣underflow".

(23) Definiere einen ADT **Complex** a für komplexe Zahlen, bei denen die beiden Anteile jeweils vom Typen a sind. Unter der Voraussetzung, dass Num (und Floating) bereits für a implementiert ist, schreibe eine Num-Instanz für **Complex** a. Solltest du komplexe Zahlen noch nicht kennen, kannst du in der Wikipedia nachschauen, wie die Operationen aussehen müssen. Kannst du Eq und Ord instanziieren?

(24) Definiere einen ADT **Ratio** a für rationale Zahlen, mit Zähler und Nenner vom Typ a. Unter der Voraussetzung, dass für a bereits die Klasse Integral instanziiert ist (und damit Num, sowie die Funktionen gcd und lcm), schreibe eine Num-Instanz für **Ratio** a. Kannst du Eq und Ord instanziieren?

Eigene Typenklassen

(25) Definiere eine Typenklasse Map mit einer Methode **lookup**.

(26) Schreibe die Instanzen für Map ListMap und Map TreeMap.

[1] `http://curry-club-augsburg.de/posts/2016-02-27-yolomorphismus.html`