Typklassen und ihre Gesetze

Wir haben im Laufe der Vorlesung sowie in der Übung eine ganz Reihe an Typklassen (und Typkonstruktorklassen) kennen gelernt und auch Gesetze formuliert, die die entsprechenden Instanzen erfüllen sollen.

Wir wollen nun die Beziehungen zwischen den Klassen sowie die Intuition hinter den Gesetzen noch einmal betrachten und unsere Erkenntnisse zusammenfassen.

```
import Prelude hiding (Functor (..), Monad (..))
```

Monoide

Monoide sind eine mathematische Struktur $(M, *, e)$ bestehend aus einer Menge M, einer assoziativen Verknüpfung $*$ und einem bezüglich dieser Verknüpfung neutralen Element e.

Wir können für diese Struktur in Haskell eine Typklasse definieren, die Implementierungen von $*$ und e vorsieht:

```
-- From Data.Monoid
class Monoid m where
  -- |Identity of 'mappend'
  mempty  :: m
  -- |An associative operation
  mappend :: m -> m -> m
  -- |Fold a list using the monoid.
  -- For most types, the default definition for 'mconcat' will be
  -- used, but the function is included in the class definition so
  -- that an optimized version can be provided for specific types.
  mconcat :: [m] -> m
  mconcat = foldr mappend mempty
```

Die Funktion `mconcat` ist bereits vordefiniert, allerdings erlaubt die Aufnahme in die Typklasse die Angabe einer effizienteren Implementierung.

Für die Instanzen fordern wir die folgenden Gesetze:

```
-- ‘mempty‘ ist links-neutral bzgl. ‘mappend‘
mempty ‘mappend‘ m           = m

-- ‘mempty‘ ist rechts-neutral bzgl. ‘mappend‘
m ‘mappend‘ mempty           = m

-- Assoziativität von ‘mappend‘
(m ‘mappend‘ n) ‘mappend‘ o = m ‘mappend‘ (n ‘mappend‘ o)
```

```haskell
-- Verhalten von ‘mconcat‘
mconcat = foldr mappend mempty
```

Als Instanzen von Monoiden haben wir u.a. `Maybe a`, `[a]`, `DList a`, aber auch `Integer` kennen gelernt.

```haskell
instance Monoid [a] where
  mempty  = []
  mappend = (++)

instance Monoid a => Monoid (Maybe a) where
  mempty                  = Nothing
  Nothing ‘mappend‘ m       = m
  m       ‘mappend‘ Nothing = m
  Just m1 ‘mappend‘ Just m2 = Just (m1 ‘mappend‘ m2)
```

Funktoren

Unter einem Funktor verstehen wir einen Typ (genauer: einen einstelligen Typkonstruktor), der es erlaubt eine Funktion auf alle Werte des Elementtyps anzuwenden.

```haskell
-- aus Prelude bzw. Data.Functor
class  Functor f  where
  fmap :: (a -> b) -> f a -> f b
```

Wir fordern die folgenden Gesetze:

```haskell
-- ‘fmap‘ ist strukturerhaltend
fmap id = id

-- Distributivität von ‘fmap‘
fmap (f . g) = fmap f . fmap g
```

Wir fordern also, dass `fmap` ein Monoid-Homomorphismus ist.

Instanzen von `Functor` sind beispielsweise `Maybe`, `[]` und `IO`.

```haskell
instance Functor [] where
  fmap = map

instance Functor Maybe where
  _ ‘fmap‘ Nothing = Nothing
  f ‘fmap‘ Just x  = Just (f x)
```

Applikative Funktoren

Applikative Funktoren sind Funktoren, die auch die Applikation der Elemente eines Fuktor auf die Elemente eines anderen Funktor erlauben:

```
-- from Control.Applicative
class Functor f => Applicative f where
  -- | Lift a value.
  pure :: a -> f a

  -- | Sequential application.
  (<*>) :: f (a -> b) -> f a -> f b

  -- | Sequence actions, discarding the value of the first argument.
  (*>) :: f a -> f b -> f b
  u *> v = pure (const id) <*> u <*> v

  -- | Sequence actions, discarding the value of the second argument.
  (<*) :: f a -> f b -> f a
  u <* v = pure const <*> u <*> v
```

Der Ausdruck `const id` ist übrigens äquivalent zu `flip const` und entspricht einer zweistelligen Funktion, die das erste Argument ignoriert und das zweite Argument zurückgibt.

Für die Instanzen fordern wir die folgenden Gesetze:

```
-- pure id ist die Identität, d.h. insbesondere pure ist effektfrei
pure id <*> v = v

-- Komposition/Assoziativität
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)

-- Homomorphismus
pure (f x) = pure f <*> pure x

-- Austausch
u <*> pure x = pure ($x) <*> u

-- Verhalten von `(*>)`
u *> v = pure (const id) <*> u <*> v

-- Verhalten von `(<*)`
u <* v = pure const <*> u <*> v
```

Als Konsequenz dieser Gesetze gilt auch:

```
fmap f u = pure f <*> u
```

Instanzen von `Functor` sind beispielsweise `Maybe` und `[]`.

```
instance Applicative [] where
  pure      = (:[])
  fs <*> xs = [ f x | f <- fs, x <- xs ]

instance Applicative Maybe where
  pure          = Just
  Just f  <*> Just x = Just (f x)
  _       <*> _      = Nothing
```

Monaden

Monaden in der Mathematik sind ein Konzept aus der *Kategorientheorie*. Aus Sichtweise eines Haskell-Programmierers ist es jedoch häufig sinnvoller, Monaden als einen *abstrakten Datentyp* von Aktionen zu betrachten.

```
class Monad m where
  -- |Inject a value into the monadic type.
  return :: a -> m a
  -- |Sequentially compose two actions, passing any value produced
  -- by the first as an argument to the second.
  (>>=)  :: m a -> (a -> m b) -> m b
  -- |Sequentially compose two actions, discarding any value produced
  -- by the first, like sequencing operators (such as the semicolon)
  -- in imperative languages.
  (>>)   :: m a -> m b -> m b
  -- |Fail with a message.  This operation is not part of the
  -- mathematical definition of a monad, but is invoked on pattern-match
  -- failure in a @do@ expression.
  fail        :: String -> m a

  m >> k      = m >>= \_ -> k
  fail s      = error s

infixl 1 >>=
```

Instanzen von `Monad` sollten die folgenden Gesetze erfüllen:

```
-- return ist links-neutral
return a >>= k   ==  k a
-- return ist rechts-neutral
m >>= return     ==  m
```

```haskell
-- Assoziativität
(m >>= k) >>= h  ==  m >>= (\x -> k x >>= h)
```

Um die Assoziativität deutlich zu machen, schreiben wir dieses Gesetz um als:

```haskell
(m >>= \x -> k x) >>= \y -> h y  ==  m >>= (\x -> k x >>= \y -> h y)
```

bzw. in `do`-Syntax:

```haskell
y <- do x <- m        x <- m
        k x      ==   y <- k x
h y                   h y
```

Wir erkennen nun, dass durch dieses Gesetz `do`-Sequenzen gewissermaßen "flachgeklopft" werden können.

Beispielinstanzen für Monaden sind:

```haskell
instance Monad [] where
  return     = (:[])              -- return == pure
  []   >>= _ = []
  x:xs >>= f = f x ++ (xs >>= f) -- (>>=)  == concatMap

instance Monad Maybe where
  return        = Just
  Nothing >>= _ = Nothing
  Just x  >>= f = f x
```

MonadPlus

`MonadPlus` ist eine Erweiterung von Monaden, die auch Alternativen und Fehlschläge bereitstellen und sich somit gut für Suchen eignen.

```haskell
-- | Monads that also support choice and failure.
class Monad m => MonadPlus m where
  -- | the identity of 'mplus'
  mzero :: m a
  -- | an associative operation
  mplus :: m a -> m a -> m a
```

Instanzen sollten die folgenden Gesetze erfüllen:

```haskell
-- mzero ist links-neutral
mzero `mplus` m         ==  m
-- mzero ist rechts-neutral
m `mplus` mzero         ==  m
-- Assozativität
m `mplus` (n `mplus` o) ==  (m `mplus` n) `mplus` o
```

Als Instanzen haben wir kennen gelernt:

```
instance MonadPlus [] where
   mzero = []
   mplus = (++)

instance MonadPlus Maybe where
   mzero             = Nothing
   Nothing `mplus` y = y
   x       `mplus` _ = x
```