Typeclassopedia
Funktoren, Monaden, Arrows
Typklassen für Typkonstruktoren

Robert Steuck

11.05.2011

Was sind Typklassen?

Was sind Typklassen?

Beispiel:

```
class Eq a where
  (==) :: a -> a -> Bool
```

Was sind Typklassen?

- Interface
- Überladung von Funktionen
- Vererbung?

Was sind Typklassen?

```
class Eq a where
  (==) :: a -> a -> Bool

data Ordering = LT | EQ | GT

class (Eq a) => Ord a  where
  compare :: a -> a -> Ordering
```

Funktoren

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Funktoren

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

data Maybe a = Just a | Nothing
```

Funktoren

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

data Maybe a = Just a | Nothing

instance Functor Maybe where
  fmap g Nothing = Nothing
  fmap g (Just x) = Just (g x)
```

Funktoren

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

data Maybe a = Just a | Nothing

instance Functor Maybe where
  fmap g Nothing = Nothing
  fmap g (Just x) = Just (g x)

instance Functor [] where
  fmap g [] = []
  fmap g (x:xs) = (g x) : (fmap g xs)
```

Funktoren

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

data Maybe a = Just a | Nothing

instance Functor Maybe where
  fmap g Nothing = Nothing
  fmap g (Just x) = Just (g x)

instance Functor [] where
  fmap g [] = []
  fmap g (x:xs) = (g x) : (fmap g xs)

-------------------------------------

  fmap :: (a -> b) -> (f a -> f b)
```

Funktoren

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Gesetze:

```
fmap id = id
fmap (g . h) = fmap g . fmap h
```

Funktoren

Zwei Sichtweisen auf Funktoren

- `fmap` wendet eine Funktion auf alle Elemente eines Containers an, ohne die Containerstruktur zu verändern.
- `fmap` wendet eine Funktion auf einen Wert an, ohne den Kontext des Wertes zu ändern.

Applicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (<$>) :: (a -> b) -> f a -> f b
  (<$>) = fmap
```

Applicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (<$>) :: (a -> b) -> f a -> f b
  (<$>) = fmap

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

Applicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (<$>) :: (a -> b) -> f a -> f b
  (<$>) = fmap

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

instance Applicative [] where
  pure x = [x]
  gs <*> xs = [g x | g <- gs, x <- xs]
```

Applicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (<$>) :: (a -> b) -> f a -> f b
  (<$>) = fmap
```

Gesetze:

```
fmap g x = pure g <*> x
```

Applicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  (<$>) :: (a -> b) -> f a -> f b
  (<$>) = fmap
```

Gesetze:

```
fmap g x = pure g <*> x
```

bzw.

```
g <$> x = pure g <*> x
```

Monaden

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

  (>>)   :: m a -> m b -> m b
  m >> n = m >>= \_ -> n
  fail   :: String -> m a
  fail e = error e
```

Monaden: return

```
return :: a -> m a
```

Monaden: return

```
return :: a -> m a
```

- gleiche Signatur wie `pure :: a -> f a`

Monaden: return

```
return :: a -> m a
```

- gleiche Signatur wie `pure :: a -> f a`
- gleiche Aufgabe

Monaden: return

```
return :: a -> m a
```

- gleiche Signatur wie `pure :: a -> f a`
- gleiche Aufgabe
- gleiche Implementierung

Monaden: >>= (bind)

```
(>>=) :: m a -> (a -> m b) -> m b
```

Monaden: >>= (bind)

```
(>>=) :: m a -> (a -> m b) -> m b
```

Beispiel für `Maybe`:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

Monaden: >>= (bind)

```
(>>=) :: m a -> (a -> m b) -> m b
```

Beispiel für Maybe:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
instance Monad Maybe where
  return x = Just x
  (Just x) >>= g = g x
  Nothing  >>= g = Nothing
```

Monaden: >>= (bind)

```
(>>=) :: m a -> (a -> m b) -> m b
```

Beispiel für Maybe:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
instance Monad Maybe where
  return x = Just x
  (Just x) >>= g = g x
  Nothing  >>= g = Nothing
```

- Berechnung mit Fehlerfall

Monaden: >>= (bind)

```
(>>=) :: m a -> (a -> m b) -> m b
```

Beispiel für Maybe:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b

instance Monad Maybe where
  return x = Just x
  (Just x) >>= g = g x
  Nothing  >>= g = Nothing
```

- Berechnung mit Fehlerfall

```
Just 3
>>= (\x -> Just (x + 1))
>>= (\x -> if odd x then Just x else Nothing)
>>= (\x -> Just (x - 1))
```

Monaden: >>= (bind)

```
(>>=) :: m a -> (a -> m b) -> m b
```

Aufgabe:

- Kombination von zwei Berechnungen
- erste Berechnung: `m a`
- zweite Berechnung abhängig vom Ergebnis der ersten: `a -> m b`

Applicative vs Monad

Applicative:

- feste Berechnungsstruktur
- vollständig unabhängig von Eingabe und Funktionswert

Monad:

- dynamische Berechnungsstruktur
- Zwischenergebnisse beeinflussen Berechnung

Applicative vs Monad

Nachbau von bind durch `fmap`
Signaturen:

```
(>>=) :: m a -> (a -> m b) -> m b
fmap  :: (a -> b) -> m a -> m b
```

Applicative vs Monad

Nachbau von bind durch `fmap`
Signaturen:

```
(>>=) :: m a -> (a -> m b) -> m b
fmap  :: (a -> b) -> m a -> m b

newBind :: m a -> (a -> m b) -> ???
newBind m g = (fmap g) m
```

Applicative vs Monad

Nachbau von bind durch `fmap`
Signaturen:

```
(>>=) :: m a -> (a -> m b) -> m b
fmap  :: (a -> b) -> m a -> m b
```

```
newBind :: m a -> (a -> m b) -> ???
newBind m g = (fmap g) m
```

```
newBind :: m a -> (a -> m b) -> m (m b)
```

Monaden: join

```
newBind :: m a -> (a -> m b) -> m (m a)

join :: Monad m => m (m a) -> m a
join mm = mm >>= id

newBind' :: m a -> (a -> m b) -> m b
newBind' m g = join (fmap g m)
```

Monaden: alternative Definition

```
class Applicative m => Monad' m where
  return   :: a -> m a
  return x = pure x
  (>>=)    :: m a -> (a -> m b) -> m b
  m >>= g  = join (fmap g m)
  join     :: m (m a) -> m a
  join mm  = mm >>= id
```

Monaden: alternative Definition

```
class Applicative m => Monad’ m where
  return   :: a -> m a
  return x = pure x
  (>>=)    :: m a -> (a -> m b) -> m b
  m >>= g  = join (fmap g m)
  join     :: m (m a) -> m a
  join mm  = mm >>= id
```

Beispiel:

```
instance Monad’ [] where
  []     >>= g = []
  (x:xs) >>= g = (g x) ++ (xs >>= g)
  join xs      = concat xs
```

Monaden: Hilfsfunktionen

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM g m = m >>= (\x -> return (g x))

ap :: Monad m => m (a -> b) -> m a -> m b
ap mg m = mg >>= (\g -> m >>= (\x -> return (g x)))
```

Monaden: Hilfsfunktionen

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM g m = m >>= (\x -> return (g x))

ap :: Monad m => m (a -> b) -> m a -> m b
ap mg m = mg >>= (\g -> m >>= (\x -> return (g x)))
```

- `liftM` entspricht `fmap`
- ap entspricht <*>

Monaden: Hilfsfunktionen

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM g m = m >>= (\x -> return (g x))

ap :: Monad m => m (a -> b) -> m a -> m b
ap mg m = mg >>= (\g -> m >>= (\x -> return (g x)))
```

- `liftM` entspricht `fmap`
- ap entspricht <*>
- `return` entspricht pure

Monaden als Applicative Functoren

WrappedMonad:

```
newtype WrappedMonad m a =
  WrapMonad { unwrapMonad :: m a }

instance Monad m => Functor (WrappedMonad m) where
  fmap g (WrapMonad v) = WrapMonad (liftM g v)

instance Monad m => Applicative (WrappedMonad m) where
  pure = WrapMonad . return
  WrapMonad g <*> WrapMonad v = WrapMonad (g `ap` v)
```

Monoid

- Menge
- zweistellige innere assoziative Operation
- neutrales Element

Monoid

- Menge
- zweistellige innere assoziative Operation
- neutrales Element

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
```

Monoid

- Menge
- zweistellige innere assoziative Operation
- neutrales Element

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a

instance Monoid [a] where
  mempty = []
  mappend xs ys = xs ++ ys
```

Monoid: Beispiele

```
newtype Sum a = Sum {getSum :: a}

instance (Num a) => Monoid (Sum a) where
  mempty = Sum 0
  (Sum x) `mappend` (Sum y) = Sum (x + y)

newtype Product a = Product {getProduct :: a}

instance (Num a) => Monoid (Product a) where
  mempty = Product 1
  (Product x) `mappend` (Product y) = Product (x * y)
```

Monoid: Beispiele

```
newtype Any = Any {getAny :: Bool}

instance Monoid Any where
  mempty = Any False
  (Any x) `mappend` (Any y) = Any (x || y)

newtype All = All {getAll :: Bool}

instance Monoid All where
  mempty = All True
  (All x) `mappend` (All y) = All (x && y)
```

Monoid

Gesetze:

```
mempty `mappend` x = x
x `mappend` mempty = x
(x `mappend` y) `mappend` z =
  x `mappend` (y `mappend` z)
```

Monoide Typklassen: Alternative

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

Monoide Typklassen: MonadPlus

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Foldable

```
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
```

Foldable

```
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
```

- `foldMap` wandelt alle Elemente eines Containers in Monoide um
- diese werden dann mit `mappend` zusammengefasst

Foldable

```
class Foldable t where
  foldMap :: Monoid m => (a -> m) -> t a -> m
```

- `foldMap` wandelt alle Elemente eines Containers in Monoide um
- diese werden dann mit `mappend` zusammengefasst

Beispiel:

```
instance Foldable [] where
  foldMap g [] = mempty
  foldMap g (x:xs) = (g x) `mappend` (foldMap g xs)
```

Category

```
class Category cat where
  id  :: cat a a
  (.) :: cat b c -> cat a b -> cat a c
```

Category

```
class Category cat where
  id  :: cat a a
  (.) :: cat b c -> cat a b -> cat a c
```

- Verallgemeinerung der Funktionskomposition

Category

```
class Category cat where
  id  :: cat a a
  (.) :: cat b c -> cat a b -> cat a c
```

- Verallgemeinerung der Funktionskomposition

Beispiel

```
instance Category (->) where
  id x  = Prelude.id x
  g . h = g Prelude.. h
```

Arrow

```
class Category a => Arrow a where
  arr :: (b -> c) -> (a b c)
  first :: (a b c) -> (a (b,d) (c,d))

  second :: (a b c) -> (a (d,b) (d,c))
  (***) :: (a b c) -> (a b' c') -> (a (b,b') (c,c'))
  (&&&) :: (a b c) -> (a b c') -> (a b (c,c'))
```

Arrow

```
class Category a => Arrow a where
  arr :: (b -> c) -> (a b c)
  first :: (a b c) -> (a (b,d) (c,d))

  second :: (a b c) -> (a (d,b) (d,c))
  (***) :: (a b c) -> (a b' c') -> (a (b,b') (c,c'))
  (&&&) :: (a b c) -> (a b c') -> (a b (c,c'))
```

- Abstraktion für Berechnungen

Arrow

```
class Category a => Arrow a where
  arr :: (b -> c) -> (a b c)
  first :: (a b c) -> (a (b,d) (c,d))

  second :: (a b c) -> (a (d,b) (d,c))
  (***) :: (a b c) -> (a b' c') -> (a (b,b') (c,c'))
  (&&&) :: (a b c) -> (a b c') -> (a b (c,c'))
```

- Abstraktion für Berechnungen

Beispiel

```
instance Arrow (->) where
  arr f = f
  first g = \(x,y) -> (g x,y)
```

ArrowChoice

```
class Arrow a => ArrowChoice a where
  left :: (a b c) -> (a (Either b d) (Either c d))

  right :: (a b c) -> (a (Either d b) (Either d c))
  (+++) :: (a b c) -> (a b' c') ->
    (a (Either b b') (Either c c'))
  (|||) :: (a b d) -> (a c d) -> (a (Either b c) d)
```

ArrowChoice

```
class Arrow a => ArrowChoice a where
  left :: (a b c) -> (a (Either b d) (Either c d))

  right :: (a b c) -> (a (Either d b) (Either d c))
  (+++) :: (a b c) -> (a b' c') ->
    (a (Either b b') (Either c c'))
  (|||) :: (a b d) -> (a c d) -> (a (Either b c) d)
```

- Flexibilisierung von Berechnungen
- Berechnungspfade komplett vordefiniert

ArrowApply

```
class Arrow a => ArrowApply a where
  app :: a (a b c , b) c
```

ArrowApply

```
class Arrow a => ArrowApply a where
  app :: a (a b c , b) c
```

- Flexibilisierung von Berechnungen
- Berechnungspfade dynamisch

ArrowApply

```
class Arrow a => ArrowApply a where
  app :: a (a b c , b) c
```

- Flexibilisierung von Berechnungen
- Berechnungspfade dynamisch
- `ArrowApply` und `Monad` sind gleich mächtig

ArrowApply

```
class Arrow a => ArrowApply a where
  app :: a (a b c , b) c
```

- Flexibilisierung von Berechnungen
- Berechnungspfade dynamisch
- `ArrowApply` und `Monad` sind gleich mächtig

Beispiel:

```
instance ArrowApply (->) where
  app (f,x) = f x
```

Extend und Comonad

```
class Functor w => Extend w where
  duplicate   :: w a -> w (w a)
  duplicate w = extend id w
  extend      :: (w a -> b) -> w a -> w b
  extend g w  = fmap g . duplicate w

class Extend w => Comonad w where
  extract   :: w a -> a
```

Extend und Comonad

```
class Functor w => Extend w where
  duplicate   :: w a -> w (w a)
  duplicate w = extend id w
  extend      :: (w a -> b) -> w a -> w b
  extend g w  = fmap g . duplicate w

class Extend w => Comonad w where
  extract   :: w a -> a
```

Gegenfunktionen aus Monad:

- `extract` und `return`
- `duplicate` und `join`
- `extend` und `>>=`

Extend und Comonad

Beispiel:

```
instance Extend Maybe where
  duplicate Nothing = Nothing
  duplicate j = Just j
```