Haskell Workshop - Typklassen

Carsten König

Juni 2021

Typ-Klassen

Deriving Via

```haskell
data DangerLevel
  = AllOk
  | NotGreat
  | UhOh
  | RealProblemHere
  | Catastrophe

instance Semigroup DangerLevel where
  Catastrophe <> _ = Catastrophe
  RealProblemHere <> Catastrophe = Catastrophe
  RealProblemHere <> _ = RealProblemHere
  ...
```

langwierig - besser

```haskell
data DangerLevel
  = AllOk
  | NotGreat
  | UhOh
  | RealProblemHere
  | Catastrophe
  deriving (Eq, Ord)

instance Semigroup DangerLevel where
  (<>) = max

instance Monoid DangerLevel where
  mempty = AllOk

data MovieRating
  = G
  | PG
  | PG13
  | R
  deriving (Eq, Ord)

instance Semigroup MovieRating where
  (<>) = max
instance Monoid MovieRating where
  mempty = G
```

mit `derivingvia`

```haskell
{-# LANGUAGE DerivingStrategies, DerivingVia #-}

newtype Supremum a = MkS a
```

```
  deriving (Eq, Ord, Bounded)

instance Ord a => Semigroup (Supremum a) where
  (<>) = max
instance (Ord a, Bounded a) => Monoid (Supremum a) where
  mempty = minBound

data DangerLevel
  = AllOk
  | NotGreat
  | UhOh
  | RealProblemHere
  | Catastrophe
  deriving (Eq, Ord, Bounded)
  deriving (Semigroup, Monoid) via (Supremum DangerLevel)

data MovieRating
  = G
  | PG
  | PG13
  | R
  deriving stock (Eq, Ord, Bounded)
  deriving (Semigroup, Monoid) via (Supremum MovieRating)
```

coerce die Implementierung von `Supremum`

```
instance Semigrou DangerLevel where
 (<>) = coerce ((<>) @(Supremum DangerLevel))
 -- geht wenn im wesentlichen Supremum DangerLevel nur ein newtype um DangerLevel ist

{-# LANGUAGE DerivingStrategies, DerivingVia, GeneralizedNewtypeDeriving  #-}

import Data.Ord (Down)

newtype Supremum a = MkS a
  deriving (Eq, Ord, Bounded)

instance Ord a => Semigroup (Supremum a) where
  (<>) = max
instance (Ord a, Bounded a) => Monoid (Supremum a) where
  mempty = minBound

data DangerLevel
  = AllOk
  | NotGreat
  | UhOh
  | RealProblemHere
  | Catastrophe
  deriving (Eq, Ord, Bounded)
  deriving (Semigroup, Monoid) via (Supremum DangerLevel)

data MovieRating
  = G
  | PG
  | PG13
  | R
  deriving stock (Eq, Ord, Bounded)
  deriving (Semigroup, Monoid) via (Supremum MovieRating)

newtype FloodLevel = MkFl Int
  deriving newtype (Eq, Ord, Bounded)
  deriving (Semigroup, Monoid) via (Supremum (Down FloodLevel))
```

Funktoren

Intuition

- *Container* mit einem oder mehr Werte
- *Berechnung* die einen Wert produziert
- `fmap` bringt eine Funktion, die Werte transformiert in den Funktor

Typklasse

```
:i Functor
class Functor (f :: * -> *) where
    fmap :: (a -> b) -> f a -> f b
    (<$) :: a -> f b -> f a
    {- MINIMAL fmap #-}
```

`<$>` ist Operator für `fmap`

Laws

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

Beispiele

- Listen
- `Maybe`
- `IO`
- `Either a`
- `((->) r)`
- `((,) a)`

```
> fmap succ [1..5]
[2,3,4,5,6]

> fmap succ (Just 5)
Just 6

> fmap succ Nothing
Nothing

> fmap (++ "!") getLine
Hello
"Hello!"

> fmap succ (Right 5)
Right 6

> fmap succ (Left "argh")
Left "argh"

> fmap succ (*5) 2
11

> fmap succ ("Hallo", 41)
("Hallo",42)
```

Extension

mit `{- LANGUAGE DeriveFunctor #-}` kann fast jeder ADT zu einem Funktor gemacht werden (mittels `deriving Functor`)

Übungen

- Funktor Instanz für `data Pair a = Pair a a`
- Funktor Instanz für `data ITree a = Leaf (Int -> a) | Node [ITree a]`
- Gesucht Typ mit Kind `* -> *`, der kein Functor ist

Funktoren sind sind *komposierbar*

Komposition

```haskell
newtype Compose f g a = Compose (f (g a))
  deriving Show


instance (Functor f, Functor g) => Functor (Compose f g) where
  fmap f (Compose x) = Compose $ fmap (fmap f) x


type MaybeList a = Compose [] Maybe a

example :: MaybeList String
example = fmap show $ Compose [Just 5, Nothing, Just 4]
```

Coprodukt

```haskell
newtype Coproduct f g a = Coproduct (Either (f a) (g a))
  deriving Show


instance (Functor f, Functor g) => Functor (Coproduct f g) where
  fmap f (Coproduct (Left x)) = Coproduct (Left $ fmap f x)
  fmap f (Coproduct (Right y)) = Coproduct (Right $ fmap f y)


type MaybeOrList a = Coproduct Maybe [] a

exampleL :: MaybeOrList String
exampleL = fmap show $ Coproduct (Left $ Just 5)

exampleR :: MaybeOrList String
exampleR = fmap show $ Coproduct (Right $ [7])
```

exotische Funktoren

```haskell
newtype Ident a = Ident { runId :: a }

instance Functor Ident where
  fmap f (Ident a) = Ident (f a)


newtype Const b a = Const { runConst :: b }

instance Functor (Const b) where
  fmap f (Const b) = Const b
```

Einschub: `fix`

```
fix :: (a -> a) -> a
```

Beispiele:

```
fix (1:) == [1,1,1,..]
fix (\c x -> if x <= 1 then 1 else x * c (x-1)) 5 == 120
```

Für Typen:

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

heftiges Beispiel 'Recursion Schemes'

```
{- LANGUAGE DeriveFunctor #-}

module RecSchemes where


data ExprF a
  = ValF Int
  | AddF a a
  | MulF a a
  deriving (Show, Eq, Functor)


type Expr = Fix ExprF


data Fix f = Fix { unFix :: f (Fix f) }


val :: Int -> Expr
val n = Fix $ ValF n

add :: Expr -> Expr -> Expr
add a b = Fix $ AddF a b

mul :: Expr -> Expr -> Expr
mul a b = Fix $ MulF a b


type Algebra f a = f a -> a


evalAlg :: Algebra ExprF Int
evalAlg (ValF n) = n
evalAlg (AddF a b) = a + b
evalAlg (MulF a b) = a * b


eval :: Expr -> Int
eval = cata evalAlg


cata :: Functor f => Algebra f a -> Fix f -> a
cata alg =
  alg . fmap (cata alg) . unFix
```

```haskell
showAlg :: Algebra ExprF String
showAlg (ValF n) = show n
showAlg (AddF a b) = "(" ++ a ++ " + " ++ b ++ ")"
showAlg (MulF a b) = a ++ " * " ++ b


showExpr :: Expr -> String
showExpr = cata showAlg
```

Semigroup / Monoid

Intuition

Ein **Monoid** ist mathematisch eine Menge + Operation auf dieser Menge. Übersetzt in Haskell geht es natürlich eher um eine Funktion/Operation auf einem (festen) Typ `a`:

`op :: a -> a -> a`

diese muss ein zwei Regeln erfüllen:

- Die Operation ist Assoziativ: `(a `op` b) `op` c == a `op` (b `op` c)`
- (nur für Monoid) Es gibt ein neutrales Element `e` mit `a `op` e == e `op` a == a` für alle `a`

Typklasse

```haskell
class Semigroup a where
  (<>) :: a -> a -> a

class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
```

`mappend` hat eine default-Implementation mit `(<>)`

`mconcat` hat eine default-Implementation mit `foldr`:

`mconcat = foldr mappend mempty`

Übung

Implementiere Monoid-Instanz für `newtype Endo a = Endo { appEndo :: a -> a }`

Lösung

```haskell
newtype Endo a = Endo { appEndo :: a -> a }

instance Semigroup (Endo a) where
  Endo f <> Endo g = Endo (f . g)

instance Monoid (Endo a) where
  mempty                  = Endo id
```

Applicative Funktoren

Intuition

was, wenn die Funktion, die wir anwenden wollen, selbst im *Kontext verpackt ist*?

mit `Applicative` können *verpackte* Funktionen auf *verpackte* Werte anwenden

Typklasse

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

Laws

- Identity: `pure id <*> v = v`
- Homomorphism: `pure f <*> pure x = pure (f x)`
- Interchange: `u <*> pure y = pure ($ y) <*> u` (die Reihenfolge der Auswertung ist egal)
- Composition: `u <*> (v <*> w) = pure (.) <*> u <*> v <*> w` (ein wenig wie Assoziativität)

mit Funktor:

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

Beispiele

Maybe

```
> Just (+) <*> Just 5 <*> Just 6
Just 11

> Just (+) <*> Nothing <*> Just 6
Nothing

> Nothing <*> Just 5
Nothing
```

Listen

Es gibt zwei Möglichkeiten Listen zu einer Instanz von `Applicative` zu machen:

- Funktionen-Liste und Werte-Liste paarweise anwenden
- jede Funktion in der Funktionen-Liste mit jedem Wert in der Werte-Liste applizieren

---

Übung

Newtypes (`ZipList` / `CompList`) für die beiden Möglichkeiten und Applicative implementieren

Lösung

```
newtype ZipList a = ZipList [a]
  deriving (Show, Functor)

instance Applicative ZipList where
  pure = ZipList . repeat
  ZipList fs <*> ZipList xs = ZipList $ zipWith ($) fs xs


newtype CompList a = CompList [a]
  deriving (Show, Functor)

instance Applicative CompList where
  pure a = CompList [a]
  CompList fs <*> CompList xs =
    CompList $ [ f x | f <- fs, x <- xs ]
```

Paare

Wie könnte eine Instanz für `(,) a` aussehen?

- Für `pure x :: x -> (a0,x)` brauchen wir einen Wert für `a0 :: a`
- Für `(a1,f) <*> (a2,x) = (a1 ? a2, f x)` müssen wir uns etwas für `? :: a -> a -> a` ausdenken
  - `pure id <*> (a,x) = (a0, id) <*> (a, x) = (a0 ? a, id x) = (law)= (a, x)` - d.h. `a0 ? a = a`
  - analog wegen *interchange* für `a ? a0 = a`
  - Composition schließlich heißt `a1 ? (a2 ? a3) = (a1 ? a2) ? a3`
  - eine derartige Operation heißt **Monoid** und dafür gibt es bereits eine Typklasse

```haskell
newtype Paar a b = Paar (a, b)
  deriving (Show, Functor)

instance Monoid a => Applicative (Paar a) where
  pure a = Paar (mempty, a)
  Paar (a,f) <*> Paar (b,x) = Paar (a `mappend` b, f x)
```

Übung

Implementiere `sequenceAL :: Applicative f => [f a] -> f [a]`

Lösung

```haskell
sequenceAL :: Applicative f => [f a] -> f [a]
sequenceAL = foldr (liftA2 (:)) (pure [])
```

Foldable

*WTF*s

```
> length (1,2)
1

> sum (4,2)
2
```

Intuition

bekanntes Aggregieren/Fold Muster

Typklasse

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

Übung

implementiere `foldR` nur mit `foldMap` (Tipp: `Endo`)

Lösung

```
foldR :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldR f b = flip appEndo b . foldMap (Endo . f)
```

Frage

Was wäre nötig um `foldMap` mit `fold :: Monoid m => t m -> m` zu implementieren?

```
import Control.Monad (join)
import Data.Foldable

foldMap' :: (Foldable t, Monoid m, Monad t) => (a -> t m) -> t a -> m
foldMap' f as = fold $ join (fmap f as)
```

Komposierbar?

```
:t foldMap . foldMap
```

Traversable

Typklasse

```
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
```

Intuition

Verallgemeinert den Funktor - jetzt kann die Abbildung Seiteneffekte haben die durch `traverse` zusammengefasst werden.

Schaut man sich `sequenceA` an, kann man sich das als Möglichkeit vorstellen Funktoren zu kommutieren (herauszuziehen)

schau `traverse . traverse` an

Laws

- `traverse Identity = Identity`
- `traverse (Compose . fmap g . f) = Compose . fmap (traverse g) . traverse f`

Beispiele

Implementiere für `MyList`

```
newtype MyList a = MyList [a]
  deriving (Functor, Foldable, Show, Eq)

instance Traversable MyList where
  traverse _ (MyList []) =
    pure (MyList [])
  traverse f (MyList (x:xs)) =
    pure (\y (MyList ys) -> MyList (y:ys)) <*> f x <*> traverse f (MyList xs)
```

Übung (hart)

implementiere `foldMap` nur mit Traversable-Methoden

Hinweis: `(,)` hat eine seltsame `Applicative` Instanz

Lösung

```
fmap' :: Traversable t => (a -> b) -> t a -> t b
fmap' f xs = runIdentity $ traverse (Identity . f) xs


foldMap' :: (Monoid m, Traversable t) => (a -> m) -> t a -> m
foldMap' f xs = fst $ traverse (\a -> (f a, ())) xs
```

siehe auch

the Essence of the Iterator Pattern