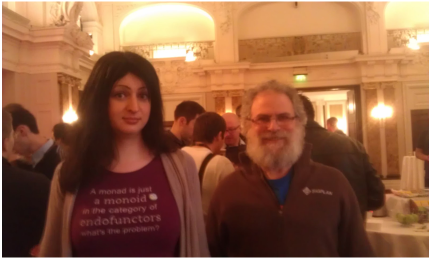
# ☺ Freie Monaden ☺




Ingo Blechschmidt
<iblech@speicherleck.de>

10. September 2015
Augsburger Curry-Club

1. Monoide
   - Definition und Beispiele
   - Nutzen
   - Freie Monoide
2. Funktoren
   - Definition und Beispiele
   - Funktoren als Container
3. Monaden
   - Definition und Beispiele
   - „Monoid in einer Kategorie von Endofunktoren“
4. Freie Monaden
   - Definition
   - Konstruktion
   - Nutzen

# Monoide

```haskell
class Monoid m where
    (<>) :: m -> m -> m
    unit :: m
```

Ein **Monoid** besteht aus

- einer Menge $M$,
- einer Abbildung $(\circ) : M \times M \to M$ und
- einem ausgezeichneten Element $1 \in M$,

sodass die **Monoidaxiome** gelten: Für alle $x, y, z \in M$

- $x \circ (y \circ z) = (x \circ y) \circ z$,
- $1 \circ x = x$,
- $x \circ 1 = x$.

# Monoide

```
class Monoid m where
    (<>) :: m -> m -> m
    unit :: m
```

Ein **Monoid** besteht aus

- einer Menge $M$,
- einer Abbildung $(\circ) : M \times M \to M$ und
- einem ausgezeichneten Element $1 \in M$,

sodass die **Monoidaxiome** gelten: Für alle $x, y, z \in M$

- $x \circ (y \circ z) = (x \circ y) \circ z$,
- $1 \circ x = x$,
- $x \circ 1 = x$.

**Beispiele:**

natürliche Zahlen, Listen, Endomorphismen, Matrizen, …

**Nichtbeispiele:**

natürliche Zahlen mit Subtraktion, nichtleere Listen, …

# Axiome in Diagrammform

$$
\begin{tikzcd}
M \times M \times M \arrow[r, "\mathrm{id}\times(\circ)"] \arrow[d, "(\circ)\times\mathrm{id}"'] & M \times M \arrow[d, "(\circ)"] \\
M \times M \arrow[r, "(\circ)"'] & M
\end{tikzcd}
$$

$$
\begin{tikzcd}
 & M & \\
M \arrow[ur] \arrow[r] & M \times M \arrow[u] & M \arrow[ul] \arrow[l]
\end{tikzcd}
$$

# Monoidhomomorphismen

Eine Abbildung $\varphi : M \to N$ zwischen Monoiden heißt genau dann **Monoidhomomorphismus**, wenn

- $\varphi(1) = 1$ und
- $\varphi(x \circ y) = \varphi(x) \circ \varphi(y)$ für alle $x, y \in M$.

**Beispiele:**

```
length :: [a] -> Int
sum :: [Int] -> Int
```

**Nichtbeispiele:**

```
reverse :: [a] -> [a]
head :: [a] -> a
```

# Wozu?

- Allgegenwärtigkeit
- Gemeinsamkeite und Unterschiede
- Generische Beweise
- Generische Algorithmen

- Monoide gibt es überall.
- Das Monoidkonzept hilft, Gemeinsamkeiten und Unterschiede zu erkennen und wertschätzen zu können.
- Man kann generische Beweise für beliebige Monoide führen.
- Man kann generische Algorithmen mit beliebigen Monoiden basteln.

# Freie Monoide

Gegeben eine Menge $X$ ohne weitere Struktur. Wie können wir auf möglichst ökonomische Art und Weise daraus einen Monoid $F(X)$ gewinnen?

# Freie Monoide

Gegeben eine Menge $X$ ohne weitere Struktur. Wie können wir auf möglichst ökonomische Art und Weise daraus einen Monoid $F(X)$ gewinnen?

Spoiler: Der **freie Monoid** $F(X)$ auf $X$ ist der Monoid der **Listen** mit Elementen aus $X$.

# Freie Monoide

Gegeben eine Menge $X$ ohne weitere Struktur. Wie können wir auf möglichst ökonomische Art und Weise daraus einen Monoid $F(X)$ gewinnen?

Spoiler: Der **freie Monoid** $F(X)$ auf $X$ ist der Monoid der **Listen** mit Elementen aus $X$.

Essenz der Freiheit: Jede beliebige Abbildung $X \to M$ in einen Monoid $M$ stiftet genau einen Homomorphismus $F(X) \to M$.

$$\begin{tikzcd} X \arrow[rr] \arrow[dr] & & M \\ & F(X) \arrow[ur, dashed] & \end{tikzcd}$$

# Freie Monoide

Freie Monaden sind …

… frei wie in Freibier?

# Freie Monoide

Freie Monaden sind …

~~… frei wie in Freibier?~~

# Freie Monoide

Freie Monaden sind …

~~… frei wie in Freibier?~~

… frei wie in Redefreiheit?

# Freie Monoide

Freie Monaden sind …

~~… frei wie in Freibier?~~

~~… frei wie in Redefreiheit?~~

# Freie Monoide

Freie Monaden sind …

~~… frei wie in Freibier?~~

~~… frei wie in Redefreiheit?~~

… frei wie in **linksadjungiert**! ✓

# Freie Monoide

Freie Monaden sind …

~~… frei wie in Freibier?~~

~~… frei wie in Redefreiheit?~~

… frei wie in **linksadjungiert**! ✓

Der Funktor $F : \mathrm{Set} \to \mathrm{Mon}$ ist **linksadjungiert** zum Vergissfunktor $\mathrm{Mon} \to \mathrm{Set}$.

In $F(X)$ sollen neben den Elementen aus $X$ nur so viele weitere Elemente enthalten sein, sodass man eine Verknüpfung $(\circ)$ und ein Einselement definieren kann.

In $F(X)$ sollen nur die Rechenregeln gelten, die von den Monoidaxiomen gefordert werden.

Die universelle Eigenschaft kann man schön in Haskell demonstrieren:

```
can :: (Monoid m) => (a -> m) -> ([a] -> m)
can phi []     = mzero
can phi (x:xs) = phi x <> can phi xs
```

Wenn man tiefer in das Thema einsteigt, erkennt man, dass endliche Listen noch nicht der Weisheit letzter Schluss sind: `http://comonad.com/reader/2015/free-monoids-in-haskell/`

# Funktoren

Ein **Funktor** $F : \mathcal{C} \to \mathcal{D}$ zwischen Kategorien $\mathcal{C}$ und $\mathcal{D}$ ordnet

- jedem Objekt $X \in \mathcal{C}$ ein Objekt $F(X) \in \mathcal{D}$ und
- jedem Morphismus $f : X \to Y$ in $\mathcal{C}$ ein Morphismus $F(f) : F(X) \to F(Y)$ in $\mathcal{D}$ zu,

sodass die **Funktoraxiome** erfüllt sind:

- $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$,
- $F(f \circ g) = F(f) \circ F(g)$.

In Haskell kommen Funktoren Hask $\to$ Hask vor:

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

# Beispiele für Funktoren

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

instance Functor [] where fmap f = map f

data Maybe a = Nothing | Just a
instance Functor Maybe where
    fmap f Nothing  = Nothing
    fmap f (Just x) = Just (f x)

data Id a = MkId a
instance Functor Id where
    fmap f (MkId x) = MkId (f x)

data Pair a = MkPair a a
instance Functor Pair where
    fmap f (MkPair x y) = MkPair (f x) (f y)
```

# Funktoren als Container

Ist `f` ein Funktor, so stellen wir uns den Typ `f a` als einen Typ von Containern von Werten vom Typ `a` vor.

Je nach Funktor haben die Container eine andere Form.

Die Vorstellung ist aus folgendem Grund plausibel: Aus einer Funktion `a -> b` können wir mit `fmap` eine Funktion `f a -> f b` machen. Also stecken wohl in einem Wert vom Typ `f a` irgendwelche Werte vom Typ `a`, die mit der gelifteten Funktion dann in Werte vom Typ `b` umgewandelt werden.

# Monaden

Eine **Monade** besteht aus

- einem Funktor $M$,
- einer natürlichen Transformation $M \circ M \Rightarrow M$ und
- einer natürlichen Transformation $\mathrm{Id} \Rightarrow M$,

sodass die **Monadenaxiome** gelten.

# Monaden

Eine **Monade** besteht aus

- einem Funktor $M$,
- einer natürlichen Transformation $M \circ M \Rightarrow M$ und
- einer natürlichen Transformation $\mathrm{Id} \Rightarrow M$,

sodass die **Monadenaxiome** gelten.

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

**Listen**

```
concat    :: [[a]] -> [a]
singleton :: a     -> [a]
```

**Maybe**

```
join :: Maybe (Maybe a) -> Maybe a
Just :: a -> Maybe a
```

Kein Beispiel: `Pair` von eben.

# Weitere Beispiele

```haskell
class (Functor m) => Monad m where
    join   :: m (m a) -> m a
    return :: a -> m a
```

## Reader

```haskell
type Reader env a = env -> a

instance Functor Reader where
    fmap f k = f . k

instance Monad Reader where
    return x = \_   -> x
    join   k = \env -> k env env
```

## State

```haskell
type State s a = s -> (a,s)

instance Monad State where
    return x = \s -> (x,s)
    join   k = \s -> let (k',s') = k s in k' s'
```

# Die Monadenaxiome

Sprechweise: Ein Wert vom Typ `m (m (m a))` ist ein (äußerer) Container von (inneren) Containern von (ganz inneren) Containern von Werten vom Typ `a`.

$$
\begin{tikzcd}
M \circ M \circ M \arrow[r, Rightarrow, "\text{innen join}"] \arrow[d, Rightarrow, "\text{außen join}"'] & M \circ M \arrow[d, Rightarrow, "\text{join}"] \\
M \times M \arrow[r, Rightarrow, "\text{join}"'] & M
\end{tikzcd}
$$

$$
\begin{tikzcd}
 & M & \\
M \arrow[ur, Rightarrow] \arrow[r, Rightarrow, "\text{return}"'] & M \circ M \arrow[u, Rightarrow, "\text{join}"'] & M \arrow[ul, Rightarrow] \arrow[l, Rightarrow, "\text{innen return}"]
\end{tikzcd}
$$

# Freie Monaden

Gegeben ein Funktor `f` ohne weitere Struktur. Wie können wir auf möglichst ökonomische Art und Weise daraus eine Monade **`Free`** `f` konstruieren?

$$\begin{tikzcd} F \arrow[rr, Rightarrow] \arrow[dr, Rightarrow] & & M \\ & \mathrm{Free}\, F \arrow[ur, Rightarrow, dashed] & \end{tikzcd}$$

```haskell
can :: (Functor f, Monad m)
    => (forall a. f a -> m a)
    -> (forall a. Free f a -> m a)
```

```haskell
import Control.Monad (join)

data Free f a
    = Pure a
    | Roll (f (Free f a))

liftF :: (Functor f) => f a -> Free f a
liftF = Roll . fmap Pure

instance (Functor f) => Functor (Free f) where
    fmap h (Pure x) = Pure (h x)
    fmap h (Roll u) = Roll (fmap (fmap h) u)

instance (Functor f) => Monad (Free f) where
    return x = Pure x
    m >>= k  = join_ $ fmap k m
        where
        join_ (Pure u) = u
        join_ (Roll v) = Roll (fmap join_ v)

can :: (Functor f, Monad m)
    => (forall a. f a -> m a)
    -> (forall a. Free f a -> m a)
can phi (Pure x) = return x
can phi (Roll u) = join $ phi . fmap (can phi) $ u
          -- oder: join $ fmap (can phi) . phi $ u
```

- Free Void ist die Maybe-Monade.
- Free Pair ist die Tree-Monade.

# Anwendungen freier Monaden

- Viele wichtige Monaden sind frei.
- Freie Monaden kapseln das „Interpreter“-Muster.
- Freie Monaden können zur Konstruktion weiterer Monaden genutzt werden, etwa zum Koprodukt zweier Monaden.