☺ Freie Monaden ☺

Ingo Blechschmidt
<iblech@speicherleck.de>

10. September 2015 und 8. Oktober 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 $e \in M$,

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

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

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 $e \in M$,

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

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

Beispiele:

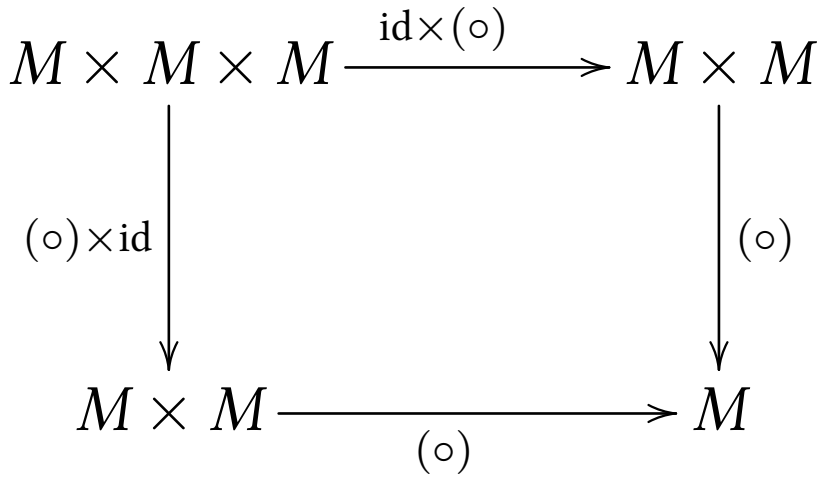
natürliche Zahlen, Listen, Endomorphismen, Matrizen, ...

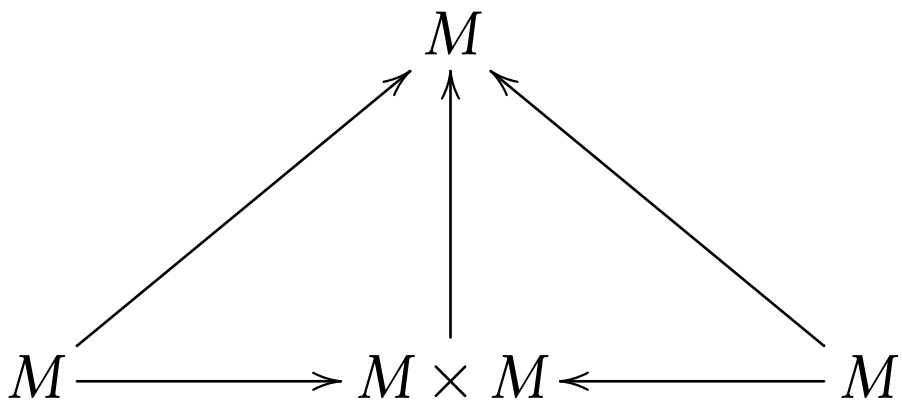
Nichtbeispiele:

natürliche Zahlen mit Subtraktion, nichtleere Listen, ...

- Die natürlichen Zahlen bilden mit der Addition als Verknüpfung und der Null als ausgezeichnetes Element einen Monoid.
- Die natürlichen Zahlen bilden mit der Multiplikation als Verknüpfung und der Eins als ausgezeichnetes Element einen Monoid.
- Die Menge $X^\star$ der endlichen Listen mit Einträgen aus einer Menge X (in Haskell also sowas wie `[X]`, wobei da auch unendliche und partiell definierte Listen dabei sind) bildet mit der Konkatenation von Listen als Verknüpfung und der leeren Liste als ausgezeichnetes Element einen Monoid.
- Ist X irgendeine Menge, so bildet die Menge aller Abbildungen $X \to X$ mit der Abbildungskomposition als Verknüpfung und der Identitätsabbildung als ausgezeichnetes Element einen Monoid.
- Die natürlichen Zahlen bilden mit der Subtraktionen keinen Monoid, da die Differenz zweier natürlicher Zahlen nicht immer wieder eine natürliche Zahl ist. Auch mit den ganzen Zahlen klappt es nicht, da die Subtraktion nicht assoziativ ist: $1 - (2 - 3) \neq (1 - 2) - 3$.

Axiome in Diagrammform





Beide Diagramme sind Diagramme in der Kategorie der Mengen, d. h. die beteiligten Objekte M, $M \times M$ und $M \times M \times M$ sind Mengen und die vorkommenden Pfeile sind Abbildungen. Wer möchte, kann aber auch vorgeben, dass M ein Typ in Haskell ist (statt $M \times M$ muss man dann `(M,M)` denken) und dass die Pfeile Haskell-Funktionen sind.

Das obere Diagramm ist wie folgt zu lesen.

Ein beliebiges Element aus $M \times M \times M$ hat die Form (x, y, z), wobei x, y und z irgendwelche Elemente aus M sind. Bilden wir ein solches Element nach rechts ab, so erhalten wir $(x, y \circ z)$. Bilden wir dieses weiter nach unten ab, so erhalten wir $x \circ (y \circ z)$.

Wir können aber auch den anderen Weg gehen: Bilden wir (x, y, z) erst nach unten ab, so erhalten wir $(x \circ y, z)$. Bilden wir dieses Element weiter nach rechts ab, so erhalten wir $(x \circ y) \circ z$.

Fazit: Genau dann *kommutiert das Diagramm* – das heißt beide Wege liefern gleiche Ergebnisse – wenn die Rechenregel $x \circ (y \circ z) = (x \circ y) \circ z$ für alle Elemente $x, y, z \in M$ erfüllt ist.

Die Pfeile auf dem unteren Diagramm sind nicht beschriftet. Hier die Erklärung, was gemeint ist.

Wir betrachten dazu ein beliebiges Element $x \in M$ (untere linke Ecke im Diagramm). Nach rechts abgebildet erhalten wir (e, x). Dieses Element weiter nach oben abgebildet ergibt $e \circ x$.

Der direkte Weg mit dem Nordost verlaufenden Pfeil ergibt x.

Das linke Teildreieck kommutiert also genau dann, wenn die Rechenregel $e \circ x = x$ für alle $x \in M$ erfüllt ist. Analog kommutiert das rechte Teildreieck genau dann, wenn $x \circ e = x$ für alle $x \in M$.

Wieso der Umstand mit der Diagrammform der Axiome? Das hat verschiedene schwache Gründe (es ist cool), aber auch einen starken inhaltlichen: Ein Diagramm dieser Art kann man nicht nur in der Kategorie interpretieren, in der es ursprünglich gedacht war (also der Kategorie der Mengen). Man kann es auch in anderen Kategorien interpretieren. Wählt man dazu speziell eine Kategorie von Endofunktoren, so erhält man die Definition einer Monade.

Monoidhomomorphismen

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

- $\varphi(e) = e$ 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
```

- Es gelten die Rechenregeln

$$\texttt{length []} = 0,$$
$$\texttt{length (xs ++ ys)} = \texttt{length xs + length ys}$$

  für alle Listen `xs` und `ys`. Das ist der Grund, wieso die Längenfunktion ein Monoidhomomorphismus ist.

  Achtung: Bevor man eine solche Aussage trifft, muss man eigentlich genauer spezifizieren, welche Monoidstrukturen man in Quelle und Ziel meint. Auf dem Typ `[a]` soll die Monoidverknüpfung durch Konkatenation gegeben sein, auf dem Typ `Int` durch Addition.

- Es gilt die Rechenregel

$$\texttt{sum (xs ++ ys) = sum xs + sum ys}$$

  für alle Listen `xs` und `ys` von Ints. Das ist die halbe Miete zur Begründung, wieso `sum` ein Monoidhomomorphismus ist.

- Die Rechenregel

$$\texttt{reverse (xs ++ ys) = reverse xs ++ reverse ys.}$$

  gilt *nicht* für alle Listen `xs` und `ys`. Daher ist `reverse` kein Monoidhomomorphismus.

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 unspektakuläre Art und Weise daraus einen Monoid $F(X)$ gewinnen?

Freie Monoide

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

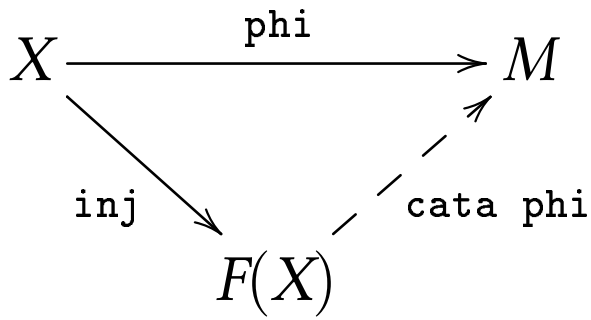
Spoiler: Der gesuchte **freie Monoid** $F(X)$ auf X ist der Monoid der endlichen **Listen** mit Elementen aus X.

Freie Monoide

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

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

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



In $F(X)$ sollen neben den Elementen aus X gerade nur so viele weitere Elemente enthalten sein, sodass man eine Verknüpfung $(\circ)$ und ein Einselement definieren kann. Es soll sich also jedes Element aus $F(X)$ über die Monoidoperationen aus denen von X bilden lassen.

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

Die nach „Essenz der Freiheit“ angegebene Forderung heißt auch *universelle Eigenschaft*. Präzise formuliert lautet sie wie folgt: Ein Paar $(F, \texttt{inj})$ bestehend aus einem Monoid F und einer Abbildung $\texttt{inj} : X \to F$ heißt genau dann *freier Monoid auf* X, wenn für jedes Paar $(M, \texttt{phi})$ bestehend aus einem Monoid M und einer Abbildung $\texttt{phi} : X \to M$ genau ein Monoidhomomorphismus $\texttt{cata phi} : F \to M$ existiert, sodass $\texttt{phi} = \texttt{cata phi.inj}$.

Man kann zeigen – über ein kategorielles Standard-Argument – dass je zwei Paare $(F, \texttt{inj})$, welche diese universelle Eigenschaft erfüllen, vermöge eines eindeutigen Isomorphismus zueinander isomorph sind. Das rechtfertigt, von *dem* freien Monoid über X zu sprechen.

Man kann die universelle Eigenschaft schön in Haskell demonstrieren:

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

Ist `phi` irgendeine Abbildung `a -> m`, so ist `cata phi` eine Abbildung `[a] -> m`. Diese ist nicht nur irgendeine Abbildung, sondern wie gefordert ein Monoidhomomorphismus.

Und mehr noch: Das Diagramm kommutiert tatsächlich, das heißt `cata phi` und `phi` haben in folgendem präzisen Sinn etwas miteinander zu tun: `cata phi . inj = phi`. Dabei ist `inj :: a -> [a]` die Abbildung mit `inj x = [x]`.

Unsere Definition von „unspektakulär“ soll sein: Alle Elemente in $F(X)$ setzen sich mit den Monoidoperationen aus denen aus X zusammen, und in $F(X)$ gelten nur die Rechenregeln, die von den Monoidaxiomen erzwungen werden, aber keine weiteren.

Das ist bei der Konstruktion von $F(X)$ als Monoid der endlichen Listen über X auch in der Tat der Fall: Jede endliche Liste mit Einträgen aus X ergibt sich als wiederholte Verknüpfung (Konkatenation) von Listen der Form `inj x` mit `x` aus X. Und willkürliche Rechenregeln wie `[a,b] ++ [b,a] == [c]` gelten, wie es auch sein soll, *nicht*.

Wieso fängt die universelle Eigenschaft genau diese Definition von Unspektakularität ein? Wenn man zu $F(X)$ noch weitere Elemente hinzufügen würde (zum Beispiel unendliche Listen), so wird es mehrere oder keinen Kandidaten für `cata phi` geben, aber nicht mehr nur einen. Genauso wenn man in $F(X)$ durch Identifikation gewisser Elemente weitere Rechenregeln erzwingen würde.

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/`

Kurz zusammengefasst: Es stimmt schon, dass der Monoid der endlichen Listen über X der freie Monoid auf X ist, sofern man als Basiskategorie in der Kategorie der Mengen arbeitet. Wenn man dagegen in Hask arbeitet, der Kategorie der Haskell-Typen und -Funktionen, so benötigt man eine leicht andere Konstruktion.

Fun Fact: Die in dem Blog-Artikel angegebene Konstruktion ergibt sich *unmittelbar* aus der universellen Eigenschaft. Es ist keine Überlegung und manuelle Suche nach einem geeigneten Kandidaten für $F(X)$ nötig. (Bemerkung für Kategorientheorie-Fans: Das es so einfach geht, liegt an einer gewissen Vollständigkeitseigenschaft von Hask.)

Freie Monoide

Freie Monoide sind …

… frei wie in Freibier?

Freie Monoide

Freie Monoide sind …

~~… frei wie in Freibier?~~

Freie Monoide

Freie Monoide sind …

~~… frei wie in Freibier?~~

… frei wie in Redefreiheit?

Freie Monoide

Freie Monoide sind …

~~… frei wie in Freibier?~~

~~… frei wie in Redefreiheit?~~

Freie Monoide

Freie Monoide sind …

~~… frei wie in Freibier?~~

~~… frei wie in Redefreiheit?~~

… frei wie in **linksadjungiert**! ✓

Freie Monoide

Freie Monoide 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 $\mathrm{Mon} \to \mathrm{Set}$.

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

Fun Fact am Rande: Ein Theorem für lau garantiert, dass in Haskell das zweite Funktoraxiom aus dem ersten folgt.

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.

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

data List  a = Nil     | Cons a [a]
data Maybe a = Nothing | Just a
data Id    a = MkId    a
data Pair  a = MkPair a a
data Unit  a = MkUnit
data Void  a
```

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.

Im Folgenden ist Hask eine geeignete Kategorie von Haskell-Typen und -Funktionen. Die Objekte sollen also Haskell-Typen sein, und Morphismen zwischen je zwei Typen sollen durch Haskell-Funktionen des entsprechenden Typs gegeben sein.

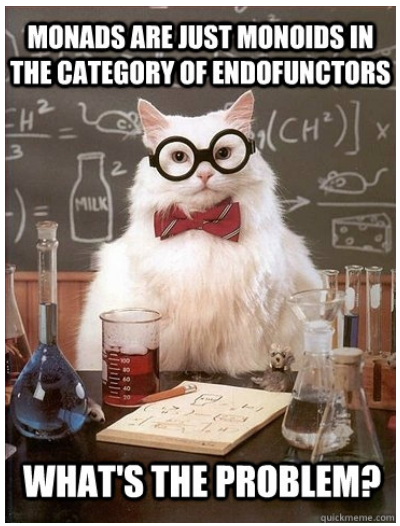
Die Kategorie End(Hask) ist die Kategorie der *Endofunktoren* von Hask. Deren Objekte sind Funktoren Hask $\to$ Hask (also das, was man in Haskell gewöhnlich als „Funktor“ bezeichnet) und deren Morphismen sind natürliche Transformationen (siehe übernächste Folie).

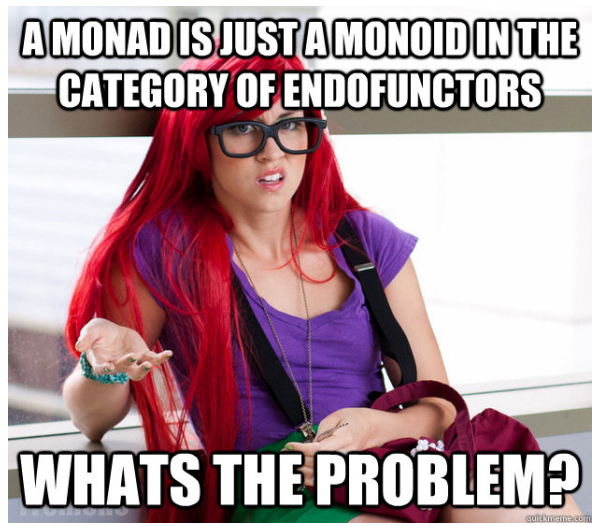
Monoidale Kategorien

Set	End(Hask)
Menge M	Funktor M
Abbildung $M \to N$	natürliche Transformation $\forall a.Ma \to Na$
$M \times N$	$M \circ N$ mit $(M \circ N)a = M(Na)$
nur linke Komponente ändern	nur äußere Schicht ändern
nur rechte Komponente ändern	nur innere Schicht ändern

Monoidale Kategorien

Set	End(Hask)
Menge M	Funktor M
Abbildung $M \to N$	natürliche Transformation $\forall a.Ma \to Na$
$M \times N$	$M \circ N$ mit $(M \circ N)a = M(Na)$
nur linke Komponente ändern	nur äußere Schicht ändern
nur rechte Komponente ändern	nur innere Schicht ändern
Monoid	**Monade**





In der Kategorientheorie ist eine natürliche Transformation $\eta : F \Rightarrow G$ zwischen zwei Funktoren $F, G : \mathcal{C} \to \mathcal{D}$ eine Familie von Morphismen $\eta_X : F(X) \to G(X)$ (ein Morphismus für jedes Objekt $X \in \mathcal{C}$), sodass für jeden Morphismus $f : X \to Y$ in $\mathcal{C}$ das Diagramm

$$\begin{array}{ccc} F(X) & \xrightarrow{\eta_X} & G(X) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(Y) & \xrightarrow[\eta_Y]{} & G(Y) \end{array}$$

kommutiert.

Spezialisiert auf den Spezialfall $\mathcal{C} = \mathcal{D} =$ Hask [und ein externes Hom in ein internes verwandelt, was auch immer das heißt] ist eine natürliche Transformation `eta` : $F \Rightarrow G$ eine polymorphe Haskell-Funktion

```
eta :: F a -> G a
```

mit

```
fmap f . eta = eta . fmap f
```

für alle Funktionen `f :: a -> b`.

Ein Theorem für lau garantiert aber, dass jede Funktion `eta` des richtigen Typs automatisch diese Kompatibilitätsbedingung mit dem `fmap` von `F` bzw. dem von `G` erfüllt, weswegen man diese Bedingung nicht explizit fordern muss. (Vereinfacht gesprochen liegt der Grund dafür darin, dass eine Haskell-Funktion nicht in Abhängigkeit der konkreten Instanz der Typvariablen `a` ihr Verhalten ändern kann.)

Beispiele für natürliche Transformationen gibt es in Haskell viele:

- `maybeToList :: Maybe a -> [a]`
- `eitherToMaybe :: Either E a -> Maybe a`

  Dabei ist `E` ein bestimmter fester Typ.
- `listToMaybe :: [a] -> Maybe a`

  Dieses Beispiel zeigt, dass natürliche Transformationen durchaus Informationen vernichten dürfen.
- `sequence :: [M a] -> M [a]`

  Dabei ist `M` eine bestimmte feste Monade.
- `tail :: [a] -> [a]`
- `length :: [a] -> Int`

  Dabei steht auf der rechten Seite die Anwendung von `a` auf den *konstanten Funktor bei `Int`*.

Die Kategorien Set und End(Hask) können beide mit einer *monoidalen Struktur* versehen werden. In der Kategorie der Mengen ist das die Operation $(M, N) \mapsto M \times N$. In der Kategorie der Endofunktoren von Hask ist es $(M, N) \mapsto M \circ N$.

In jeder monoidalen Kategorie (also einer Kategorie zusammen mit einer monoidalen Struktur) kann man von *Monoid-Objekten* sprechen. Monoid-Objekte in Set sind einfach ganz gewöhnliche Monoide. Monoid-Objekte in End(Hask) sind Monaden.

Siehe auch:
`http://blog.sigfpe.com/2008/11/from-monoids-to-monads.html`.

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
```

Anschauliche Interpretation: Monaden sind spezielle Containertypen, die zwei Zusatzfähigkeiten haben: Ein Container von Containern von Werten soll man zu einem einfachen Container von Werten „flatten“ können (`join`), und einen einzelnen Wert soll man in einen Container packen können (`return`).

Weitere Beispiele

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

Reader

```
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
```

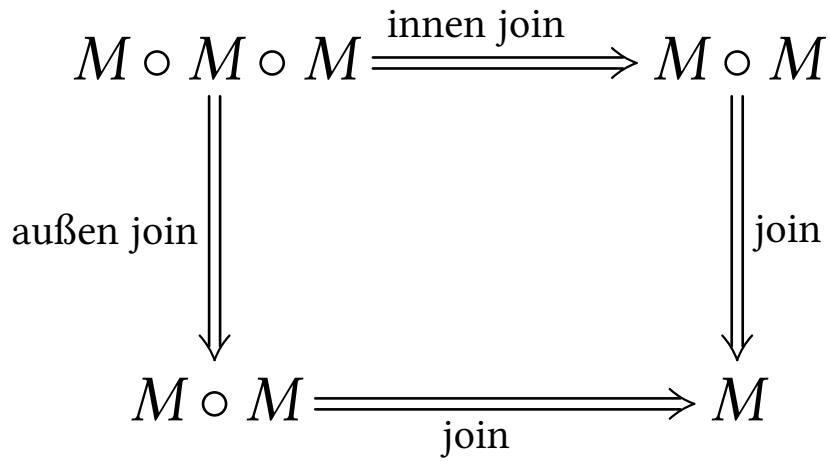
State

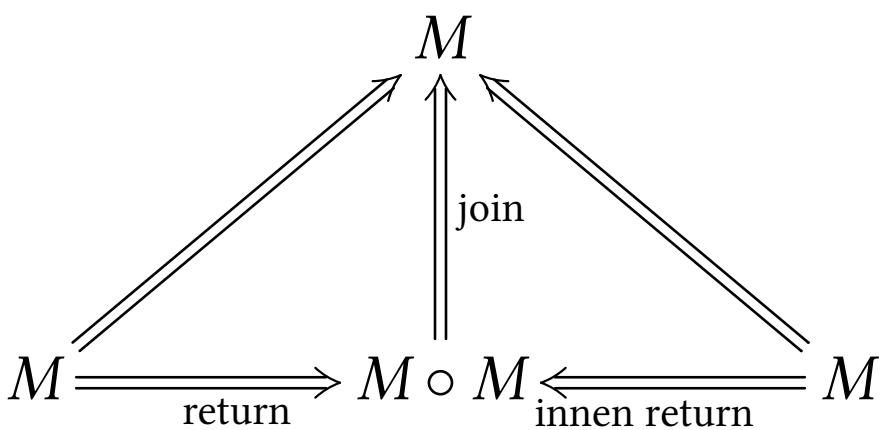
```
type State s a = s -> (a,s)

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

Auch `State` und `Reader` passen in die „Container mit Zusatzfähigkeit“-Intuition, wenn man sich hinreichend verbiegt. Schaffst du das?

Die Monadenaxiome

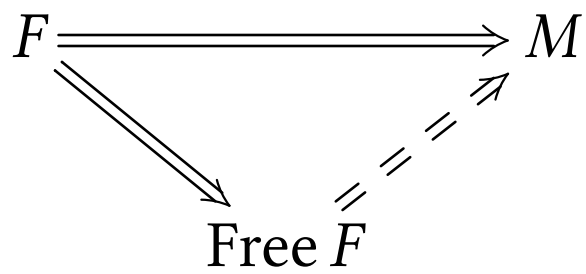
Sprechweise: Ein Wert vom Typ `m (m (m a))` ist ein (äußerer) Container von (inneren) Containern von (ganz inneren) Containern von Werten vom Typ `a`.





Freie Monaden

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




```
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 f a` ist der Typ der „`f`-förmigen“ Bäume mit Blättern vom Typ `a`. Dabei flattened `join` einen `f`-förmigen Baum von `f`-förmigen Bäumen von irgendwelchen Werten zu einem einfachen `f`-förmigen Baum dieser Werte.

- `Free Void` ist die `Id`-Monade.
- `Free Unit` ist die `Maybe`-Monade. Hier hat jeder Mutterknoten kein einziges Kind. Die Bezeichnung „Mutterknoten“ ist also etwas euphemistisch.
- `Free Pair` ist die `Tree`-Monade. Hier hat jeder Mutterknoten genau zwei Kinder.
- `Free Id` ist die `(Writer Nat)`-Monade. Hier hat jeder Mutterknoten genau ein Kind.

Die Listen-Monade ist übrigens nicht frei. Übungsaufgabe: Beweise das!

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, welches die Fähigkeiten zweier gegebenen Monaden vereint.

Die in diesen Folien gegebene Konstruktion der freien Monade hat ein Effizienzproblem – genau wie bei der linksgeklammerten Verkettung von Listen. Mit der sog. Kodichtemonade, einer speziellen Links-Kan-Erweiterung, kann man das Problem lösen.

Mehr zum Interpreter-Muster gibt es in einem Folgevortrag zu Effektsystemen. Kurz zusammengefasst: Wenn man maßgeschneidert eine Monade konstruieren möchte, welche genau einen gewissen Satz von Nebenwirkungen unterstützt, dann kann man das über freie Monaden. Und noch einfacher über „freiere Monaden“ (da ist der Ausgangspunkt nur ein Typkonstruktor `F`, der *kein* Funktor sein muss).

Ein Vorgeschmack in Form von Haskell-Code gibt es unter `https://github.com/iblech/vortrag-haskell/blob/master/freie-monaden.hs`.

Nach der Schule trifft Barbie Paul und Peter in der Bibliothek. „Funktionale Programmierung ist fantastisch!“, freut sich Barbie. „Ich weiß nicht", sagt Paul, „was zum Teufel ist eine Monade noch mal?“. „Eine Monade ist einfach ein Monoid in einer Kategorie von Endofunktoren – wo liegt das Problem?“, antwortet Barbie.