Fortgeschrittene Funktionale Programmierung mit Haskell

Applikative Funktoren & Monaden

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

12. November 2015

Either

Polymorphe Datentypen können auch mehrere Typparameter haben. `Either` ist ein wichtiges Beispiel:

```
data Either a b  =  Left a | Right b
```

`Either` ist Typ für disjunkte Vereinigungen:

$$A_1 \dot\cup A_2 = \{(i, a) \mid a \in A_i\}$$

Beispiel:

$$\{\heartsuit, \diamondsuit\} \dot\cup \{\heartsuit, \clubsuit, \spadesuit\} = \{(1, \heartsuit), (1, \diamondsuit), (2, \heartsuit), (2, \clubsuit), (2, , \spadesuit)\}$$

Für endliche Mengen gilt: $|A_1 \dot\cup A_2| = |A_1| + |A_2|$

Man spricht auch von "Summentypen" bei Alternativen

Modul Data.Either

```
module Data.Either where
  data Either a b  =  Left a | Right b

  isRight :: Either a b -> Bool
  isRight (Left  _) = False
  isRight (Right _) = True

  lefts   :: [Either a b] -> [a]
  lefts x = [a | Left a <- x]

  partitionEithers :: [Either a b] -> ([a],[b])
  partitionEithers []       = ([],[])
  partitionEithers (h : t)
    | Left  l <- h = (l:ls,   rs)
    | Right r <- h = (  ls, r:rs)
    where (ls,rs) = partitionEithers t
```

Modul Data.Either

```
module Data.Either where
  data Either a b  =  Left a | Right b

  isRight :: Either a b -> Bool
  isRight (Left  _) = False
  isRight (Right _) = True

  lefts   :: [Either a b] -> [a]
  lefts x = [a | Left a <- x]

  partitionEithers :: [Either a b] -> ([a],[b])
  partitionEithers []       = ([],[])
  partitionEithers (h : t) =
    let (ls,rs) = partitionEithers t
    in case h of  Left l  -> (l:ls,   rs)
                  Right r -> (  ls, r:rs)
```

Either für Fehler-Kommunikation

Wir haben `Maybe a` für Berechnungen verwendet, welche fehlschlagen können.
Stattdessen bietet sich auch `Either String a` an, wenn man noch Fehlermeldungen mitgeben möchte, z.B.:

```
myDiv :: Double -> Double -> Either String Double
myDiv x 0 = Left "Error: Division by 0"
myDiv x y = Right $ x / y

firstDiff :: Eq a => [a] -> [a] -> Either String a
firstDiff [] [] = Left "No Difference found."
firstDiff (x:xs) (y:ys)
  | x /= y    = Right x
  | otherwise = firstDiff xs ys
```

`Maybe` war eine Instanz von `Functor`. Können wir `Either` auch zu einer Instanz von `Functor` machen?

Typklasse Functor

Im Module `Data.Functor` findet sich folgende Definition:

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

(<$>) :: (a -> b) -> f a -> f b
(<$>) = fmap        -- Infix-Synonym
```

Gesetze:

Identität: `fmap id == id`

Komposition: `fmap f . fmap g == fmap (f . g)`

Der Parameter `f` der Typklasse `Functor` steht also nicht für einen konkreten Typ wie z.B. `Tree Int`, sondern für einen Typkonstruktor wie z.B. `Tree`.
Die Typklasse `Functor` ist also die Klasse aller Typen-in-Kontext, welche es erlauben Ihre Inhalte auf andere im gleichen Kontext abzubilden.

Functor Either ?

Nein, `Either` selbnst können wir so nicht zu einer Instanz von `Functor` machen:

```
instance Functor Either where
    fmap f (Left  a) = Left  $ f a
    fmap f (Right b) = Right $ f b
```

```
Error:
  Expecting one more argument to `Either'
  In the instance declaration for `Functor Either'
```

`Functor` erwartet einen Typkonstruktor mit einem Argument, aber `Either` ist ein Tykonstruktor mir zwei Argumenten.
Es liegt ein **Kind**-Fehler vor!

Kind

Der **Kind** eines Typen beschreibt die Art des Typen, also die Anzahl und Art der Argumente eines Typkonstruktors.

Ein Kind ist entweder `*` oder aus zwei Kinds per Pfeil zusammengesetzt:

$$\kappa ::= * \mid \kappa \to \kappa$$

`(*)` steht für alle konkreten Datentypen, z.B. `Int`, `Bool`, `Double` und auch `[Int]`, `Maybe Bool`, `Either String Double`

`(* -> *)` steht für alle Typkonstruktoren mit genau einem Argument, z.B. `[]`, `Maybe` und auch `Either String`.

`(* -> (* -> *))` steht für alle Typkonstruktoren mit genau zwei Argumenten, z.B. `Either`.

Wie bei Funktionstypen ist die Rechtsklammerung implizit, d.h.

`* -> (* -> *)` = `* -> * -> *` ≠ `(* -> *) -> *`

Kind

GHCi kann den Kind eines Typen mit `:kind` anzeigen:

```
> :kind Integer
Integer :: *
> :ki Maybe
Maybe :: * -> *
> :k Either
Either :: * -> * -> *
```

`Maybe` und `Either` haben also unterschiedliche Kinds, da diese Typkonstruktoren unterschiedliche viele Argumente verlangen.

Nicht vergessen, Funktionstypen sind auch Typen:

```
> :k (Int -> Int)
(Int -> Int) :: *
> :k ((->) Int)
((->) Int) :: * -> *
> :k (->)
(->) :: * -> * -> *
```

Kind of Functor

Angewendet auf die Typklasse `Functor` gibt GHCi folgendes heraus:

```
> :k Functor
Functor :: (* -> *) -> Constraint
```

Funktoreninstanzen benötigen ein Typargument mit Kind `* -> *`

```
> :k Functor Maybe
Functor Maybe :: Constraint

> :k Functor f => (a -> b) -> f a -> f b
Functor f => (a -> b) -> f a -> f b :: *
```

Die GHC-spezielle Antwort "Constraint" bedeutet dabei, dass es sich nicht um einen Typen, sondern um eine Typklasse(-nbeschränkung) handelt, wie wir sie vor dem Doppelpfeil `=>` schreiben.

Funktor-Instanz für Either

Auch wenn wir `Either` nicht zur Funktoreninstanz machen können, so können wir immerhin eine für `Either a` definieren:

```
data  Either a b  =  Left a | Right b
  deriving (Eq, Ord, Read, Show, Typeable)

instance Functor (Either a) where
  fmap _ (Left x)  = Left x
  fmap f (Right y) = Right (f y)
```

So ist es im Modul `Data.Either` auch definiert.

Beispiel:

```
> fmap (*3) $ Left "Error"
Left "Error"
> fmap (*3) $ Right 23
Right 69
```

Functor-Instanz für Either II

Ein `flip`-äquivalent gibt es nicht, aber wie können einen äquivalenten Datentyp mit eigener Instanz einführen: dazu müssen wir `newtype` bemühen:

```
newtype FlipEither b a = FE (Either a b)
  deriving Show

instance Functor (FlipEither b) where
  fmap f (FE (Left  y)) = FE $ Left $ f y
  fmap _ (FE (Right x)) = FE $ Right x
```

Dies ist auch zwingend notwendig, da ja anhand des Typen entschieden wird, welcher Code ausgeführt wird!

Immer nur eine Instanz pro Typ und Klasse!

```
> fmap (*2) $ Left 21
Left 21
> fmap (*2) $ FE $ Left 21
FE (Left 42)
```

Newtype Deklaration

`newtype`-Deklaration ein optimierter Speziallfall zum "Kopieren" einen existierenden Typ. Das Typsystem unterscheidet beide Typen!

```
newtype Typname = Konstruktor <typ> deriving <Typklassen>
```

- Frischer Typname und frischer Konstruktor mit 1 Argument
- Typ und Konstruktor werden oft gleich benannt, da Typen und Werte in getrennten Namensräumen leben `Bool` $\neq$ `True`

Unterschied zur herkömmlichen Deklaration

```
data Typname = Konstruktor <typ>     deriving <Typklassen>
```

- `newtype` effizienter, da Konstruktor zur Laufzeit nicht existiert
- `newtype` ist fauler, da kein Pattern-Match ausgeführt wird

Beide Varianten erlauben eigen Typklassen-Instanzen!

Beispiel: Down

Modul `Data.Ord` definiert:

```
newtype Down a = Down a deriving (Eq, Show, Read)

instance Ord a => Ord (Down a) where
  compare (Down x) (Down y) = y `compare` x
```

Damit können wir für jeden Typ `a` aus der Klasse `Ord` die Ordnung leicht umdrehen:

```
> sort [5,3,1,4,2]
[1,2,3,4,5]
> sort $ Down <$> [5,3,1,4,2]
[Down 5,Down 4,Down 3,Down 2,Down 1]
```

Mit `newtype` können wir also leicht alternative Instanzdeklarationen definieren, also die Beschränkung, pro Typ und Klasse immer nur Instanz zu haben, quasi umgehen.

Laziness von Newtype

`newtype` ist *fauler*, da keine Pattern-Matches ausgeführt werden – der Konstruktor existiert ja zur Laufzeit gar nicht:

```
data    Foo a = Foo a deriving Show
newtype Baz a = Baz a deriving Show

foo :: Foo a -> String
foo (Foo _) = "OK!"

baz :: Baz a -> String
baz (Baz _) = "OK!"

> foo undefined
"*** Exception: Prelude.undefined
> baz undefined
"OK!"
> foo (Foo undefined)
"OK!"
```

Mehrstellige Funktionen und Funktoren

```
> fmap (*2) (Just 21)
Just 42
> :t fmap (*) (Just 2)
fmap (*) (Just 2) :: Num a => Maybe (a -> a)
```

- Wie können wir z.B. die binäre Operation `(*)` auf zwei Werte des Typs `Maybe Int` anwenden?
- Was bedeutet ein Funktionstypen-mit-Kontext, wie etwa `Maybe (a -> a)`?
- Wie wenden wir Funktionstypen-mit-Kontext an?

Applikative Funktoren

Modul `Control.Applicative` bietet speziellere Funktoren:

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

Ein **Applikativer Funktor** erlaubt die Anwendung einer Funktion-mit-Kontext auf einen Wert-mit-Kontext!

Folgende Gesetze sollten gelten:

- Identität `v == pure id <*> v`
- Komposition `u <*> (v<*>w) == pure (.) <*> u <*> v <*> w`
- Homomorphie `pure f <*> pure x == pure (f x)`
- Interchange `u <*> pure y == pure (\f -> f y) <*> u`

Für die Definition `f <$> x = pure f <*> x` folgen daraus auch alle Gesetze für Funktoren!

Beispiel: Applicative-Instanz für Maybe

```
instance Applicative Maybe where
  -- pure  :: a -> Maybe a
  pure = Just
  -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
  Nothing  <*> _           = Nothing
  (Just f) <*> something = f <$> something
```

Wenn wir keine Funktion bekommen, dann können wir auch keine Funktionsanwendung durchführen.

```
> Just (+3) <*> Just 8
Just 11
> pure (+) <*> Just 3 <*> Just 8
Just 11
> (+) <$> Just 3 <*> Just 8
Just 11
```

Ist einer der Werte `Nothing`, dann kommt `Nothing` heraus.

Beispiel: Applicative-Instanz für Maybe

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

Wenn wir keine Funktion bekommen, dann können wir auch keine Funktionsanwendung durchführen.

```
> Just (+3) <*> Just 8
Just 11
> pure (+) <*> Just 3 <*> Just 8
Just 11
> (+) <$> Just 3 <*> Just 8
Just 11
```

Ist einer der Werte `Nothing`, dann kommt `Nothing` heraus.

Beispiel: Applicative-Instanz für Listen

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

Ist der Kontext eine Vielzahl von Möglichkeiten, dann werden einfach alle Möglichkeiten kombiniert.

```
> pure (++) <*> ["H","P"] <*> ["i","a"]
["Hi","Ha","Pi","Pa"]

> [(*0),(+10),(*7)] <*> [1,2,3]
[0,0,0,11,12,13,7,14,21]

> [(+),(*)] <*> [1,2] <*> [3,5]
[4,6,5,7,3,5,6,10]
```

Beispiel: Applicative-Instanz für Listen

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

Ist der Kontext eine Vielzahl von Möglichkeiten, dann werden einfach alle Möglichkeiten kombiniert.

```
>       (++) <$> ["H","P"] <*> ["i","a"]
["Hi","Ha","Pi","Pa"]

> [(*0),(+10),(*7)] <*> [1,2,3]
[0,0,0,11,12,13,7,14,21]

> [(+),(*)] <*> [1,2] <*> [3,5]
[4,6,5,7,3,5,6,10]
```

Beispiel: Applicative-Instanz für Listen II

Listen kann man auch durch *zippen* zu einer gültigen Instanz von `Applicative` machen. Wir verwenden dazu erneut den newtype-Trick von Folie 04-12:

```
newtype ZipList a = ZipList [a] deriving Show

instance Applicative ZipList where
  pure x = ZipList $ repeat x
  ZipList fs <*> ZipList xs = ZipList $
                              zipWith (\f x -> f x) fs xs
```

Beispiele

```
> (*) <$>  [1,2,3] <*>  [1,10,100,1000]
[1,10,100,1000,2,20,200,2000,3,30,300,3000]
> (*) <$> ZipList [1,2,3] <*> ZipList [1,10,100,1000]
ZipList [1,20,300]
> (,,) <$> ZipList[1,2] <*> ZipList[3,4] <*> ZipList[5,6]
ZipList [(1,3,5),(2,4,6)]
```

Fortgeschrittene Funktionale Programmierung mit Haskell

Applikative Funktoren & Monaden

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

19. November 2015

Liften

Wir können also Funktionen mit beliebiger Stelligkeit in einen Kontext hieven. Das Muster ist immer die gleiche mehrfache Anwendung der links-assoziativen Infix-Funktion `<*>`:

$$\texttt{f <\$> x}_1\ \texttt{<*>}\ \cdots\ \texttt{<*> x}_n$$

Dies bezeichnet man als **liften** einer Funktion.
Modul `Control.Applicative` definiert entsprechend:

```
liftA  :: Applicative f => (a -> b) -> f a -> f b
liftA  f a     = pure f <*> a

liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b   = f <$> a <*> b

liftA3 :: Applicative f =>
            (a -> b -> c -> d) -> f a -> f b -> f c -> f d
liftA3 f a b c = f <$> a <*> b <*> c
```

. . . wobei die Infix-Schreibweise inzwischen eher vorgezogen wird.

Sequence

Liften des Cons-Operators `(:)` ergibt folgende nützliche Definition aus Modul `Data.Traversable`:

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA []     = pure []
sequenceA (x:xs) = liftA2 (:) x (sequenceA xs)
                 -- = (:) <$> x <*> sequenceA xs
```

Eine Liste von Werten-im-Kontext wird zur Liste-im-Kontext:

```
> sequenceA [Just 3, Just 2, Just 1]
Just [3,2,1]
> sequenceA [Just 3, Nothing, Just 1]
Nothing
> sequenceA [(+3),(+2),(+1)] 3
[6,5,4]
> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
```

Beispiel: Applicative-Instanz für Funktionen

```
instance Applicative ((->) e) where
  pure x  = \_ -> x
  f <*> g = \e -> f e (g e)
```

Der Typ `((->) r)` hat den erwarteten Kind `* -> *`, da ein Funktionstyp ja zwei Argumente hat und hier nur ein Argument angegeben wird. Die Instanziierung der Typen ergibt also:

```
pure  :: a -> (e -> a)
<*>   :: (e -> (a -> b)) -> (e -> a) -> (e -> b)
```

Beispiel

```
>       (+) <$> (+1) <*> (*10) $ 5
56
```

Bemerkung Diese beide Funktionen sind übrigens im Lambda-Kalkül auch als die Kombinatoren $\mathbb{K}$ und $\mathbb{S}$ bekannt.

Beispiel: Applicative-Instanz für Funktionen

```
instance Applicative ((->) e) where
  pure x  = \_ -> x
  f <*> g = \e -> f e (g e)
```

Der Typ `((->) r)` hat den erwarteten Kind `* -> *`, da ein Funktionstyp ja zwei Argumente hat und hier nur ein Argument angegeben wird. Die Instanziierung der Typen ergibt also:

```
pure  :: a -> (e -> a)
<*>   :: (e -> (a -> b)) -> (e -> a) -> (e -> b)
```

Beispiel

```
> pure (+) <*> (+1) <*> (*10) $ 5
56
```

Bemerkung Diese beide Funktionen sind übrigens im Lambda-Kalkül auch als die Kombinatoren $\mathbb{K}$ und $\mathbb{S}$ bekannt.

Beispiel: Applicative-Instanz für Funktionen

```
instance Applicative ((->) e) where
  pure x  = \_ -> x
  f <*> g = \e -> f e (g e)
```

Der Typ `((->) r)` hat den erwarteten Kind `* -> *`, da ein Funktionstyp ja zwei Argumente hat und hier nur ein Argument angegeben wird. Die Instanziierung der Typen ergibt also:

```
pure  :: a -> (e -> a)
<*>   :: (e -> (a -> b)) -> (e -> a) -> (e -> b)
```

Beispiel

```
> (\x y z-> (100*(x-1))+y+z) <*> (+1) <*> (*10) $ 5
456
```

Bemerkung Diese beide Funktionen sind übrigens im Lambda-Kalkül auch als die Kombinatoren $\mathbb{K}$ und $\mathbb{S}$ bekannt.

Beispiel: Eval

Die Instanz für `(->) e` erlaubt uns z.B. folgende Vereinfachung des Codes für eine Berechnung mit Kontext:

```
data Exp v = Var v | Val Int | Neg (Exp v)
           | Add (Exp v ) (Exp v )

fetch :: Var -> Env -> Int
         -- Variable im Kontext nachschlagen
eval  :: Exp String -> Env -> Int
eval  (Var x)   env = fetch x env
eval  (Val i)   env = i
eval  (Neg p)   env = negate  $  eval  p env
eval  (Add p q) env = (+) (eval  p env)  (eval  q env)
```

Man kann argumentieren, dass es nervt, den Kontext `env` explizit zu behandeln. Dank der `(->) e`-Instanz für Applicative müssen wir das jedoch gar nicht tun!

Beispiel: Eval

Dank der `(->) e`-Instanz für Applicative können wir den Kontext `env` verstecken:

```
data Exp v = Var v | Val Int | Neg (Exp v)
           | Add (Exp v ) (Exp v )

fetch :: Var -> Env -> Int
         -- Variable im Kontext nachschlagen
eval' :: Exp String -> Env -> Int
eval' (Var x)       = fetch x
eval' (Val i)       = pure i
eval' (Neg p)       = negate <$> eval' p
eval' (Add p q)     = (+) <$> eval' p <*> eval' q
```

Zusammenfassung Applicative

- Applikative Funktoren erlauben die Behandlung beliebig stelliger Funktionen-im-Kontext
- Behandlung eines Kontexts wird bequem versteckt:
  Funktionsanwendung ohne Kontext `f $ x y`
  Funktionsanwendung mit Kontext `f <$> x <*> y`
- `f <$> x <*> y` ist äquivalent zu `pure f <*> x <*> y`
- Der Infix-Operator `<*>` ist links-assoziativ
- Applikative Funktoren wurden erst spät identifiziert:
  Functional Pearl: "Applicative Programming with Effects"
  von Conor McBride und Ross Paterson
  im Journal of Functional Programming 18:1 (2008)

Effekte abschütteln?

Beobachtung

Es gibt keinen allgemeinen Weg, einen Kontext (oder auch Effekt) abzuschütteln:

```
unpure :: Applicative f => f a -> a

unpure :: Maybe a -> a
unpure (Just x) = x
unpure Nothing  = undefined  -- Wie a erzeugen ???

unpure :: [a] -> a
unpure (x:_)    = x
unpure []       = undefined  -- Wie a erzeugen ???

unpure :: (r -> a) -> a
unpure f = undefined         -- Wie a erzeugen ???
```

Kontext wird immer berücksichtigt

```
iffy :: Applicative f => f Bool -> f a -> f a -> f a
iffy fb ft ff = cond <$> fb <*> ft <*> ff
  where cond b t f = if b then t else f
```

Dies hat als Konsequenz, dass applikative Funktoren immer alle Kontexte/Effekte berücksichtigen müssen:

```
> iffy (Just True) (Just 42) (Just 0)
Just 42
> iffy (Just True) (Just 42) Nothing
Nothing
```

Der Rückgabewerte einer Berechnung-im-Kontext kann die nachfolgenden Berechnungen-im-Kontext nicht beeinflussen!

Dies *kann* wünschenswert sein, aber nicht immer.

Monaden

Einen Ausweg bieten Monaden:
Auch hier können Kontexte/Effekte nicht abgeschüttelt werden, aber es wird die Existenz einer Operation vorausgesetzt, welche Werte zwischendurch aus ihrem Kontext/Effekt herausnehmen kann, so dass darauf reagiert werden kann:

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

```
miffy :: Monad m => m Bool -> m a -> m a -> m a
miffy mb mt mf = mb >>= condm
  where condm b = if b then mt else mf
```

```
 > miffy (Just True) (Just 42) (Nothing)
Just 42
```

Modul Control.Monad

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

  (>>)   :: m a ->       m b  -> m b
  x >> y = x >>= \_ -> y  -- Default aus (>>=) generiert

  fail :: String -> m a   -- Default
  fail msg = error msg
```

- `return` entspricht `pure`,
  also einbetten in den leeren Kontext/Effekt
- `(>>=)` gesprochen "**bind**", die Komposition
- `(>>)` ist Hintereinanderausführung,
  entspricht Komposition mit Wegwerfen des Zwischenergebnis
- `fail` erlaubt gesonderte Fehlerbehandlung

Werte des Typs `m a` nennen wir monadische **Aktion** von Typ `a`

Monaden Gesetze

Instanzen der Typklasse `Monad` *sollten* folgenden Gesetze einhalten. Wie bei den Gesetzen der Typklassen `Functor` und `Applicative` ist der Programmierer der Instanz für die Einhaltung dieser Gesetze zuständig!

1. **Links-Identität**
   `return x >>= act` macht das Gleiche wie `act x`
2. **Rechts-Identität**
   `mval >>= return` macht das Gleiche wie `mval`
3. **Assoziativität**
   `mval >>= (\x-> act1 x >>= act2)` macht das gleiche wie `(mval >>= act1) >>= act2`

Alle Instanzen, welche diesen Gesetzen genügen, sind **Monaden**!

Alle Monaden sind Applikative Funktoren

Alle Monaden definieren auch schon (applikative) Funktoren:

```
fmap f mx = mx >>= (\x -> return $ f x)

pure = return
mf <*> mx = mf >>= (\f ->
                mx >>= (\x -> return $ f x) )
```

- `Control.Monad` hat folgende redundante Definitionen:

  ```
  fmap == liftM :: Monad m =>    (a -> b) -> m a -> m b
  <*>  == ap    :: Monad m => m (a -> b) -> m a -> m b
  ```

  Grund: Früher einmal war `Applicative` noch keine Oberklasse der Typklasse `Monad` gewesen. seit GHC 7.10; 3/15
- *Umgekehrt gilt es nicht:* Nicht alle applikative Funktoren sind Monaden, z.B. `ZipList` ist keine Monade.

Fazit: Monaden sind stärker; reicht aber ein applikativer Funktor, so ist dies die bessere Wahl (wg. Verständlichkeit & Effizenz)!

Monadische Komposition

Die bind-Operation (>>=) ist bereits sehr mächtig.
Viele andere Operationen können problemlos daraus abgeleitet werden, z.B. die Komposition monadischer Funktionen:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g = \x -> f x >>= g
```

Grundidee Monaden

Monaden kann man sich weiterhin grob als Container vorstellen, welche funktionale Werte mit Kontext/Seiteneffekten verpacken.

Zwei grundlegende Operationen in einer Monade:

- Komposition zweier monadischer Aktionen, d.h. zwei Aktionen werden zu einer monadischen Aktion verschmolzen; Seiteneffekt/Kontext wird zwischen Aktionen hindurchgefädelt.
- Einbettung funktionaler Werte in die Monade mit leerem Seiteneffekt bzw. ohne Kontextänderung. `return`

Konsequenz

Monaden vereinfachen den expliziten Umgang mit Seiteneffekten in einer rein funktionalen Welt!

Beispiel Bei I/O geht es nahezu ausschließlich um Seiteneffekte. Entsprechend wird I/O in Haskell mit der IO-Monade durchgeführt.

DO-Notation für alle Monaden

```
foo = action1 w   >>= (\x ->
      action2     >>= (\y ->
      action3 x   >>
      action4 v y >>= (\z ->
      return $ bar x y z)))
```

Haskell kennt einen speziellen "syntaktischen Zucker" als vereinfachte Schreibweise, die **DO-Notation** welche für *jede Instanz* der Typklasse `Monad` unterstützt wird:

```
foo = do  x <- action1 w
          y <- action2
          action3 x
          z <- action4 v y
          return $ bar x y z
```

Es gilt wieder die Layout-Regel für die Einrückung!

DO-Notation vs. Applicative

Programme mit DO-Notation sehen sehr "imperativ" aus — und sind es oft auch *unnötigerweise*:

```
foo  :: Monad m => m (a -> b) -> m a -> m b
foo mf mx = do
  f <- mf
  x <- mx
  return  (f x)
```

Dieses häufig auftretende Muster sollte man besser so schreiben:

```
foo  :: Applicative m => m (a -> b) -> m a -> m b
foo mf mx = mf <*> mx
```

Kürzerer, lesbarer Code; und
beide Seiteneffekte können sich nicht gegenseitig beeinflussen!

⇒ Wenn Applicative ausreicht, nicht mit Monaden draufhauen!

DO-Notation vs. Applicative

Programme mit DO-Notation sehen sehr "imperativ" aus — und sind es oft auch *unnötigerweise*:

```
foo2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
foo2 f mx my = do
  x <- mx
  y <- my
  return $ f x y
```

Dieses häufig auftretende Muster sollte man besser so schreiben:

```
foo2 :: Applicative f => (a->b->c) -> f a -> f b -> f c
foo2 f x y = f <$> x <*> y
```

Kürzerer, lesbarer Code; und
beide Seiteneffekte können sich nicht gegenseitig beeinflussen!

⇒ Wenn Applicative ausreicht, nicht mit Monaden draufhauen!

Monaden Gesetze

Instanzen der Typklasse `Monad` *sollten* folgenden Gesetze einhalten. Wie bei den Gesetzen der Typklassen `Functor` und `Applicative` ist der Programmierer der Instanz für die Einhaltung dieser Gesetze zuständig!

1. **Links-Identität**
   `return x >>= act` macht das Gleiche wie `act x`
2. **Rechts-Identität**
   `mval >>= return` macht das Gleiche wie `mval`
3. **Assoziativität**
   `mval >>= (\x-> act1 x >>= act2)` macht das gleiche wie `(mval >>= act1) >>= act2`

Alle Instanzen, welche diesen Gesetzen genügen, sind **Monaden**!

Monaden Gesetze in DO-Notation

Monaden Gesetze ausgedrückt unter Verwendung der DO-Notation:

1. **Links-Identität**

```
do  y <- return x        |  do  act x
    act y
```

2. **Rechts-Identität**

```
do  y <- mval            |  do  mval
    return y
```

3. **Assoziativität**

```
do  x <- mval            |  do  y <- do
    y <- act1 x                       x <- mval
    act2 y                            act1 x
                                 act2 y
```

Alles was diesen Gesetzen genügt, ist eine **Monade**!

Monaden Gesetze in DO-Notation

Monaden Gesetze ausgedrückt unter Verwendung der DO-Notation:

1. **Links-Identität**

```
do  y <- return x        |  do  act x
    act y
```

2. **Rechts-Identität**

```
do  y <- act x           |  do  act x
    return y
```

3. **Assoziativität**

```
do  x <- act0 x          |  do  y <- do
    y <- act1 x                       x <- act0 z
    act2 y                            act1 x
                                 act2 y
```

Alles was diesen Gesetzen genügt, ist eine **Monade**!

Zusammenfassung Do-Notation

Do-Notation erlaubt es, mehrere monadische Aktionen hintereinander auszuführen. Es werden automatisch Anwendungen von bind eingefügt, um den Kontext hindurchzufädeln.

- Do-Block gilt als eine einzelne monadische Aktion
- Gesamter Do-Block hat immer den Typ der letzten Aktion
- Die Monade aller Aktionen muss die gleiche sein
- Es gilt Layout-Regel: Pro Zeile eine monadische Aktion
  Einrückung weiter rechts: Zeile geht weiter
  Einrückung weiter links: Do-Block beendet
- `let x = expression` für rein funktionale Berechnungen im Do-Block erlaubt.
  Achtung: kein `in` im Do-Block
- `let` und `<-` erlauben Pattern-Match auf linker Seite;
  Schlägt dieser Pattern-Match fehl, so wird `fail` aufgerufen.

Maybe als Monade

```
data Maybe a = Nothing | Just a
```

Diesen Datentyp können wir zur Monade machen:

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

    fail _            = Nothing
```

Modelliert Berechnungen mit Seiteneffekt "Fehler":

1. "Aktion" ist Berechnung, welche Wert liefert oder fehlschlägt.
2. Wenn Berechnung Wert liefert, dann damit weiter rechnen.
3. Wenn eine einzelne Berechnung fehlschlägt, so schlägt auch die gesamte Berechnung fehl.

Maybe als Monade

```
data Maybe a = Nothing | Just a
```

Diesen Datentyp können wir zur Monade machen:

```
instance Monad Maybe where
    return x          = Just x
    (>>=) mx f = case mx of
           Nothing  -> Nothing
           (Just x) -> f x
    fail _            = Nothing
```

Modelliert Berechnungen mit Seiteneffekt "Fehler":

1. "Aktion" ist Berechnung, welche Wert liefert oder fehlschlägt.
2. Wenn Berechnung Wert liefert, dann damit weiter rechnen.
3. Wenn eine einzelne Berechnung fehlschlägt, so schlägt auch die gesamte Berechnung fehl.

Maybe erfüllt Monaden-Gesetze

1. Links-Identität

```
return x >>= f
 = case (return x) of Nothing -> Nothing; (Just y) -> f y
 = case (Just x) of Nothing -> Nothing; (Just y) -> f y
 = f x
```

2. Rechts-Identität

```
m >>= return
 = case m of Nothing -> Nothing; (Just y) -> return y
 = case m of Nothing -> Nothing; (Just y) -> (Just y)
 = m
```

3. Assoziativität

```
m >>= (\x-> f x >>= g)
 = ...Übung...
 = (m >>= f) >>= g
```

Maybe-Monade: Beispiele (1)

```
> Just 9 >>= \x -> return (x*10)
Just 90
> Nothing >>= \x -> return (x*10)
Nothing
```

```
> :t sequence
sequence :: Monad m => [m a] -> m [a]

> sequence [Just 1, Just 2]
Just [1,2]
> sequence [Just 1, Just 2, Nothing, Just 4]
Nothing
```

Maybe-Monade: Beispiele (2)

```
maybeMult mx my = do
  x <- mx
  y <- my
  return $ x * y

> maybeMult (Just 4) (Just 5)
Just 20
> maybeMult Nothing  (Just 5)
Nothing
> maybeMult (Just 4) Nothing
Nothing
```

Das Beispiel ist vielleicht etwas unsinnig, aber in der Praxis erspart die DO-Notation für Maybe-Monaden wiederholte pattern-matches mit `(Just x)` — wenn man denn nur das Ergebnis haben will, falls *alle* Zwischenergebnisse nicht `Nothing` waren.

Maybe-Monade: Beispiele (3)

```
maybeMult mx my = (*) <$> mx <*> my
```

```
> maybeMult (Just 4) (Just 5)
Just 20
> maybeMult Nothing  (Just 5)
Nothing
> maybeMult (Just 4) Nothing
Nothing
```

Nicht vergessen:

Der Applikative-Stil noch einfacher und vielleicht auch lesbarer!

Listen als Monade

Auch Listen können wir wieder als Monaden auffassen:

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
  fail   _ = []
```

Listen-Monade *simuliert* nichtdeterministische Berechnungen:

- Anstatt eines einzelnen Wertes wird mit einer Liste von Werten simuliert gleichzeitig gerechnet.
- Schlägt eine Berechnung fehl, so wird die Menge der möglichen Ergebniswerte für diese Berechnung leer.

Simulation: Die Berechnung bleibt wie gehabt deterministisch; es werden lediglich alle auftretenden Möglichkeiten nacheinander durchprobiert!

Listen-Monade: Beispiele (1)

Unser Input ist entweder 3, 4 oder 5. Was kommt heraus, wenn wir darauf eine Funktion anwenden, welche entweder die Zahl negiert oder unverändert zurück gibt?

```
> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
```

Wenn wir keinen Input bekommen, so kann auch nichts berechnet werden:

```
> [] >>= \x -> [1..5]
[]
```

Listen-Monade: Beispiele (2)

Mehrere Alternativen müssen alle miteinander kombiniert werden:

```
> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

In DO-Notation geschrieben sieht das so aus:

```
do  n  <- [1,2]
    ch <- ['a','b']
    return (n,ch)
```

Als List-Comprehension:

```
[ (n,ch) | n <- [1,2], ch <- ['a','b'] ]
```

Im applikativen Stil:

```
(,) <$> [1,2] <*> ['a','b']
```

Listen-Monade: Beispiele (2)

List-Comprehensions sind in der Tat identisch zur DO-Notation:

```
foo1 :: [Int] -> [Int] -> [(Int,Int)]
foo1 xs ys = [(z,y)|x<-xs, x/=5, let z=x*10, y<-ys]

foo2 :: [Int] -> [Int] -> [(Int,Int)]
foo2 xs ys = do
    x <- xs
    if (x/=5) then return () else (fail "Oops!")
    let z = x*10
    y <- ys
    return (z,y)

> foo1 [4..6] [7..9]
[(40,7),(40,8),(40,9),(60,7),(60,8),(60,9)]
> foo2 [4..6] [7..9]
[(40,7),(40,8),(40,9),(60,7),(60,8),(60,9)]
```

Fortgeschrittene Funktionale Programmierung mit Haskell

Applikative Funktoren & Monaden

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

26. November 2015

MonadPlus

Das Konzept der Monade kann weiter verfeinert werden.
Listen und Maybe sind beide Instanzen der Typklasse `MonadPlus`:

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

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

Die Monade besitzt also zusätzlich die Struktur eines Monoids:

- Assoziative binäre Operation `mplus`
- Neutrales Element `mzero` zu dieser Operation

Typklasse `Monoid` erfasst nur diese Struktur ohne Monade.

Listen-Monade: Beispiele (3)

```
guard :: (MonadPlus m) => Bool -> m ()
guard True  = return ()
guard False = mzero
```

Dies erlaubt Verfeinerung des Codes für List-Comprehensions:

```
foo1 xs ys = [(z,y)|x<-xs, x/=5, let z=x*10, y<-ys]
foo3 xs ys = do
  x <- xs
  guard (x/=5)
  let z = x*10
  y <- ys
  return (z,y)
```

Schlägt ein Pattern-Match mit `<-` fehl, wird `fail` aufgerufen. Nutzen von `fail` ist nun klar: In der Listen-Monade muss z.B. nicht gleich komplette Berechnung abgebrochen werden!

Weitere Monadische Funktionen

`Control.Monad` bietet monadische Varianten für bekannte Funktionen, welche ums "fädeln" der Monade erweitert wurden:

```
replicateM :: Monad m => Int -> m a -> m [a]

mfilter    :: MonadPlus m => (a -> Bool) -> m a -> m a

foldM      :: Monad m =>
              (a -> b -> m a) -> a -> [b] -> m a

zipWithM   :: Monad m =>
              (a -> b -> m c) -> [a] -> [b] -> m [c]
```

Verschachtelte Monaden kann man hiermit auswickeln:

```
msum :: MonadPlus m =>   [m a] -> m a
join :: Monad     m => m (m a) -> m a
```

WHEN & UNLESS

Bedingte Ausführung von monadischen Aktionen erlauben uns

```
when   :: Moand m => Bool -> m () -> m ()
unless :: Monad m -> Bool -> m () -> m ()
```

Die Funktion `when` führt die übergebene Aktion nur dann aus, wenn das erste (funktionale) Argument zu `True` auswertet.

Bei der Funktion `unless` ist das umgekehrt. Sie führt die übergebene Aktion nur dann aus, wenn das erste (funktionale) Argument zu `False` auswertet.

```
when   p s = if p then s          else return ()
unless p s = if p then return () else s
```

SEQUENCE

Mehrere monadische Aktionen können wir hintereinander ausführen:

```
sequence  :: Monad m => [m a] -> m [a]
sequence_ :: [m a] -> m ()
```

BEISPIEL

```
> :t print  -- Seiteneffekt: Bildschirmausgabe
print :: Show a => a -> IO ()

> sequence $ map print [1..3]
1
2
3
[(),(),()]
```

Wenn das aufgesammelte Ergebnis nicht interessiert, z.B. weil die verwendeten Aktion wie im Beispiel immer nur `()` zurückgeben, dann können wir auch die Variante `sequence_` verwenden.

Definition Sequence

```
sequence :: Monad m => [m a] -> m [a]
sequence []  = return []
sequence (c : cs) = do
  x <- c
  xs <- sequence cs
  return (x : xs)
```

Definition Sequence

```
sequence :: Monad m => [m a] -> m [a]
sequence []  = return []
sequence (c : cs) = do
  x <- c
  xs <- sequence cs
  return (x : xs)
```

kann man im applikativen Stil auch so schreiben:

```
sequenceA :: Applicative f => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (c : cs) = (:) <$> c <*> sequenceA cs
```

. . . haben wir auch schon in Übung A5-2 implementiert!

mapM

Wir können eine Sequenz von Aktionen vorher auch noch transformieren lassen:

```
mapM  :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
```

Dabei ist `mapM f` ist tatsächlich äquivalent zu `sequence . map f`

Beispiel

```
> mapM_ print [1..5]
1
2
3
4
5
()
```

Die Variante `mapM_` verwirft lediglich das Endergebnis, d.h. nur die Seiteneffekt interessiert uns.

forM

Zum Abschluß noch ein (vielleicht) alter Bekannter:

```
forM  :: Monad m => [a] -> (a -> m b) -> m [b]
forM_ :: Monad m => [a] -> (a -> m b) -> m ()
```

Beispiel

Man kann mit `flip mapM` und anonymen Funktionen fremd-aussehende Programme schreiben:

```
import Control.Monad
main = do
    colors <- forM [1,2,3,4] (\a -> do
        putStrLn $ "Welche Farbe assoziierst Du mit "
                   ++ show a ++ "?"
        color <- getLine
        return color)
    putStrLn "Farbe der Zahlen 1, 2, 3 und 4 sind: "
    mapM putStrLn colors
```

Zusammenfassung Monaden

- Monaden sind ein *Programmierschema* für zusammengesetzte abhängige Berechnungen mit Kontext oder Seiteneffekten
- Kontext/Seiteneffekte werden durch die Monade explizit ausgedrückt und deklariert, also greifbar und sichtbar gemacht
- Monaden separieren das Fädeln des Kontexts von der eigentlich Berechnung; d.h. Hintereinanderausführung mehrerer Berechnungen mit Kontext/Seiteneffekt wird vereinfacht.
- Monadenoperation müssen drei Gesetze erfüllen: **Links-/Rechtsidentität** und **Assoziativität**.
- Monaden sind keine eingebaute Spracherweiterung; man kann Monaden auch in anderen Programmiersprachen verwenden
- GHC bietet syntaktische Unterstützung (DO-Notation)
  Monaden-Bibliotheken auch für andere Sprachen verfügbar

Übersicht häufig verwendeter Monaden

Häufige Anwendungen für Monaden sind:

- **I/O Monade** ermöglicht funktionale User-Interaktion.
- **Fehlermonade** für Berechnungen, welche fehlschlagen können
  z.B. Maybe-Monade, MonadError
- **Nichtdeterminismus** für Berechnungen, welche mit mehreren Alternativen gleichzeitig rechnen
  z.B. Listen-Monade
- **Zustandsmonade** für Berechnungen mit veränderlichen Kontext
  `Control.Monad.State`
  Spezialfälle:
  - **Lesemonade** liest Zustand nur aus, z.B. globale Parameter
    `Control.Monad.Reader`
  - **Schreibmonade** beschriebt Zustand nur, z.B. logging
    `Control.Monad.Writer`

Zustandsmonade

Die Zustandsmonade gibt einer Berechnung einen Kontext; Berechnung darf Kontext verändern, z.B. können Variablen gesetzt und gelesen werden.

- Kann imperative Programmierweise simulieren, bzw. in die funktionale Welt einbetten
- Typinformation gibt genau an, welche Information verfügbar ist, d.h. Seiteneffekte können genau eingegrenzt werden

Beispiel

```
stackOperation = do
  push 2              --     [2]
  push 3              --   [3,2]
  push 4              -- [4,3,2]
  r <- pop            --   [3,2]; r==4
  pop                 --     [2]; return 3
```

Implementation

Auch wenn es imperativ aussieht:
Hinter der Monaden-Kulisse geht alles rein funktional zu!

```
newtype State a = State (Int -> (a, Int))

instance Monad State where
  return x  = State $ \s -> (x,s)  -- :: a -> State a
  (State x) >>= f =                -- :: State a -> (a -> State b) -> State b
    State $ \s0 ->
        let (val_x, State s1) = x s0
            (State cont)      = f val_x
        in  cont s1

runState :: Int -> State a -> a
runState i (State s) = fst $ s i

getState :: State Int
getState   = State $ \s -> (s,s)

putState :: Int -> State ()
putState s = State $ \_ -> ((),s)

inc :: State ()
```

```
demo :: (Int,Int,Int)
demo = runState 0 $ do
  x <- getState
  putState $ x + 10
  y <- getState
  putState $ x + 100
  inc
  inc
  z <- getState
  return (x,y,z)
> demo
```

Implementation

Auch wenn es imperativ aussieht:
Hinter der Monaden-Kulisse geht alles rein funktional zu!

```
newtype State a = State (Int -> (a, Int))

instance Monad State where
  return x  = State $ \s -> (x,s)  --
  (State x) >>= f =                --
    State $ \s0 ->
        let (val_x, State s1) = x s0
            (State cont)      = f val_
        in  cont s1

runState :: Int -> State a -> a
runState i (State s) = fst $ s i


getState :: State Int
getState   = State $ \s -> (s,s)

putState :: Int -> State ()
putState s = State $ \_ -> ((),s)

inc :: State ()
```

Instanzdeklaration für `Functor` und `Applicative` nicht vergessen!
Können hier generisch aus der `Monad`-Instanz erstellt werden:

```
fmap f mx == mx >>=
             \x -> return $ f x
```

usw.

```
|       demo = runState 0 $ do
|         x <- getState
|         putState $ x + 10
|         y <- getState
|         putState $ x + 100
|         inc
|         inc
|         z <- getState
|         return (x,y,z)
|       > demo
```

Control.Monad.State

Da Haskell sehr modular ist, können wir uns (wie so oft) in den Bibliotheken bedienen, anstatt alles von Hand zu programmieren:

```
type State s a = ...Instanz von MonadState...

class Monad m => MonadState s m | m -> s where
  get :: m s                          -- Zustand lesen
  put :: s -> m ()                    -- Zustand schreiben
  state :: (s -> (a, s)) -> m a       -- Funktion zu Monade

runState :: State s a -> s -> (a, s)
modify   :: MonadState s m => (s -> s) -> m ()
gets     :: MonadState s m => (s -> a) -> m a
```

- `m -> s` erzwingt, dass `s` durch `m` eindeutig bestimmt ist
- `runState` wendet Zustandsberechnung auf Anfangszustand an, liefert Endzustand und Ergebnis
- `gets` erlaubt bequemes auslesen des Zustands, z.B. wenn der Zustand mehrere einzelne Werte enthält (z.B. Tupel)

Bibliotheksunterstützung

Die Bibliotheksunterstützung vereinfacht die Verwendung der Zustandsmonade sehr:

```
import Control.Monad.State

demo :: (Int,Int,Int)
demo = fst $ (flip runState) 0 $ do x <- get
                                   put $ x + 10
                                   y <- get
                                   put $ x + 100
                                   inc
                                   inc
                                   z <- get
                                   return (x,y,z)

inc :: State Int ()
inc = modify (+1)
```

Verwendung der Module `Control.Monad.Reader` und `Control.Monad.Writer` verläuft ganz ähnlich.

Reihenfolge monadischer Aktionen

Die Berechnung einer monadischen Aktion `m a`, also eines funktionalen Werts des Typs `a` im Kontext `m`, hängt natürlich immer vom Kontext ab.

Am Beispiel der Zustandsmonade sehen wir deutlich, dass der Kontext einer monadischen Aktion vom Kontext der vorangegangen Aktion abhängig ist — das war ja die Motivation von `(>>=)` gewesen.

Konsequenzen

- Kontext muss immer hindurchgefädelt werden.
- Die Reihenfolge der Abarbeitung einer Folge von monadischen Aktionen wird trotz Lazy Evaluation weitestgehend eingehalten, da ja der jeweilige Kontext bekannt sein muss!

Control.Monad.Error

Die Fehlermonade erlaubt neben dem Werfen von Ausnahmen auch das Abfangen und die Behandlung von Fehlern.

```
class Monad m => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

Eine wichtige Instanz dieser Monade ist
`MonadError IOException IO`

Control.Monad.Error

Dies funktioniert ganz ähnlich wie bei der Maybe-Monade, anstatt `Maybe` verwenden wir jedoch den Datentyp `Either String`:

```
data Either a b = Left a | Right a

instance Monad (Either a) where
  return a       = Right a
  Left  e >>= k = Left e
  Right a >>= k = k a

instance MonadError e (Either e) where
  throwError e = Left e
  Left  e `catchError` k = k e
  Right a `catchError` k = Right a
```

Nichtdeterminismus

Diese Sichtweise kann manche Berechnungen vereinfachen:

Beispiel

Die Berechnung der Potenzmenge, also aller Teilmengen einer Menge, kann man mit einem nicht-deterministischen Filter erreichen:

```
powerset :: [a] -> [[a]]
powerset xs = filterM (\x -> [True, False]) xs

> powerset [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

- Fehlermonade kann damit auch verstanden werden: Berechnung hat 0 oder 1 Ergebnis

Beispiel: Werte mit Wahrscheinlichkeiten

Wir modellieren mehrere Ergebniswerte mit Wahrscheinlichkeiten.

```
import Data.Ratio

newtype Prob a = Prob [(a,Rational)] deriving Show
getProb :: Prob a -> [(a,Rational)]
getProb (Prob l) = l

example = [("Blue",1%2),("Red",1%4),("Green",1%4)]
```

Bedeutung ist, dass Ergebnis zu $\frac{1}{2}$ =50% Blau ist, zu $\frac{1}{4}$ = 25% Grün, usw. Das die Summe 100% ergibt wird hier nicht modelliert.

Wie machen wir dies zur Monade?

Beispiel: Werte mit Wahrscheinlichkeiten

Hier ist mal sinnvoller, zuerst die `Functor`-Instanz zu definieren:

```
instance Functor Prob where
  fmap f (Prob xs) = Prob $ map (\(x,p) -> (f x,p)) xs
```

Jetzt können wir damit die `Monad`-Instanz deklarieren:

```
instance Monad Prob where
  return x = Prob [(x,1%1)]          -- 1 Antwort, 100%
  m >>= f  = vereinigen (fmap f m) --
  fail _   = Prob []

vereinigen :: Prob (Prob a) -> Prob a
vereinigen (Prob xs) = Prob $ concat $ map multAll xs
  where
    multAll (Prob inxs,p) = map (\(x,r) -> (x,p*r)) inxs
```

Join

Die Funktion `vereinigen` aus dem vorangegangen Beispiel ist generell für Monaden nützlich und wird in `Control.Monad` als `join` definiert:

```
join :: (Monad m) => m (m a) -> m a
join x =  x >>= id
```

Vorsicht: In diesem Beispiel konnten wir dies jedoch nicht nutzen, da wir `(>>=)` ja gerade über `join` definiert haben!

Es gilt generell:
`m >>= f` ist äquivalent zu `join (fmap f m)`

Hello World

Haskell Funktionen sind rein und kennen keine Seiteneffekte.
Für I/O sind Seiteneffekte essentiell. Diese werden in Haskell durch den Kontext der eingebauten `IO`-Monade modelliert:

```
main :: IO ()
main = do
  putStrLn "Wie heisst Du? "
  name <- getLine
  putStrLn ("Hallo " ++ name ++ "!")
```

- Bildschirmausgabe ist ein Seiteneffekt!
  Eingabe ist ein Seiteneffekt!
- Referenzielle Transparenz erzwingt, dass eine Funktion für gleiche Argumente das gleiche Ergebnis liefert.
  Rückgabewert von `getLine` hängt jedoch vom Kontext ab!
  Der Kontext ist hier der Benutzer bzw. die Außenwelt.

Modellierung IO als Zustandsmonade:

Wenn die Außenwelt der Kontext ist, dann können wir die fest eingebaute IO-Monade vereinfacht als Zustandsmonade auffassen:

```
type IO a = Welt -> (a,Welt)

putStr   :: String -> IO ()      -- String -> Welt -> ((),Welt)
getLine  ::           IO String  --       Welt -> (String,Welt)
```

Diese Funktionen bekommen den aktuellen Kontext und dürfen in lesen (Tastatur auslesen) und auch verändern (Bildschirm beschreiben).
Ergebnis-Welt einer IO-Funktion muss als Argument an die nächste IO-Funktion weitergereicht werden. Das Fädeln der Welt übernimmt die Monade. $\Rightarrow$ Erzwingt Einhaltung der Reihenfolge!

Hinweis: Nur Modell! Typkonstruktor IO nicht wirklich so definiert!

Komposition von IO-Aktionen

Schreiben Sie zur Übung eine `Monad`-Instanz für dieses Modell:

```
type Welt = ()                    -- Dummy ohne Bedeutung
type MyIO a = Welt -> (a,Welt)

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

`(>>)` führt einfach zwei IO-Aktionen hintereinander aus. Dazu muss aus dem Ergebnis der ersten Aktion die verändert Welt ausgepackt und als Argument für die zweite Aktion übergeben werden. Man muss die Welt also durchfädeln (engl. threading).

Typ `Welt` hätte in der Realität natürlich Felder für Bildschirm, Tastatur, Speicher, ... — für unsere Analogie hier unerheblich

Main Methode

Bisher haben wir Haskell nur im Interpreter laufen lassen. Wenn wir eine ausführbare Datei erstellen wollen, dann braucht unser Haskell Programm eine Funktion `main :: IO ()`:
Datei `HelloWorld.hs` enthält nur die eine Zeile:

```
main = putStrLn "Hello World!"
```

```
> ghc HelloWorld.hs
[1 of 1] Compiling Main    ( HelloWorld.hs, HelloWorld.o )
Linking HelloWorld ...

> ./HelloWorld
Hello World!
```

Tipp: Besteht ein Programm aus mehreren Modulen, dann sollte die Option `--make` angegeben werden, damit alle notwendigen Module ebenfalls gleich kompiliert werden.

Main Methode

Die `main` Funktion hat meist den Typ `IO ()`

Natürlich können wir auch in der Funktion `main` mehrere IO-Aktionen durchführen:

```
main = do
  putStrLn "Psst! Wie heisst Du?"
  name <- getLine
  putStrLn $ "Hey " ++ (map Data.Char.toUpper) name

gut :: String -> IO ()
gut name = putStrLn $ name ++ " gefällt mir gut!"
```

Nur `main` verfügt über die Welt. Die IO-Aktionen in `gut` sind ohne Wirkung, da `main` die Funktion `gut` nicht aufruft!

Main Methode

Wenn `main` aber `gut` aufruft, dann werden dessen Aktionen an der entsprechenden Stelle ausgeführt. Natürlich kann `gut` weitere Funktionen mit Typ `IO a` aufrufen.

```
main = do
  putStrLn "Psst! Wie heisst Du?"
  name <- getLine
  gut name
  putStrLn $ "Hey " ++ (map Data.Char.toUpper) name

gut :: String -> IO ()
gut name = putStrLn $ name ++ " gefällt mir gut!"
```

In jedem Falle gilt:

Man kann am Typ einer Funktion erkennen ob diese Seiteneffekte haben kann – und die Monade beschreibt alle möglichen Seiteneffekte!

`IO` bedeutet: Alles ist möglich!

Weltentrennung

Es empfiehlt sich daher dringend, funktionalen Code von I/O-Code so weit wie möglich zu trennen:

```
main = do
  line <- getLine
  if null line
    then putStrLn "(no input)"
    else do
      putStrLn $ reverseWords line
      main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

- DO-Ausdrücke können wie alle anderen Ausdrücke überall auftauchen, wo Ihr Typ gefragt ist.
- Auch IO-Aktionen können Rekursion nutzen

Ausgabe

- `putChar :: Char -> IO ()`
  Gibt ein einzelnes Zeichen aus.

- `putStr :: String -> IO ()`
  `putStrLn :: String -> IO () -- fügt '\n' ans Ende an`
  Gegeben einen String aus.
  Aufgrund der verzögerten Auswertung kann es sein, dass nur komplette Zeilen ausgegeben werden (Vorlesung 13).

- `print :: Show a => a -> IO ()`
  Gibt einen Wert der Typklasse `Show` aus.
  Identisch zu `putStrLn . show`

GHCi und I/O

Der Interpreter GHCi erlaubt an der Eingabeaufforderung übrigens auch beliebige IO-Aktionen.

In der Tat wird auf das Ergebnis einer jeden Eingabe ohnehin die Funktion `print` angewendet:

```
> [1..5]
[1,2,3,4,5]
it :: [Integer]
> print [1..5]
[1,2,3,4,5]
it :: ()
```

Lediglich der Ergebnistyp wird beibehalten. `print` hat den Typ `Show a => a -> IO ()`, d.h. liefert immer nur `()` zurück.

Eingabe

- `getChar :: IO Char` Liest ein einzelnes Zeichen ein.
- `getLine :: IO String`
  Liest so lange ein, bis ein Zeilchenvorschub durch drücken der Return-Taste erkannt wird.
- `getContents :: IO String`
  Liest alles ein, was der Benutzer jemals eingeben wird (oder bis ein Dateiende-Zeichen (Ctrl-D) erkannt wird)
- `interact :: (String -> String) -> IO ()`
  Verarbeitet den gesamten Input-Strom mit der gegebenen Funktion und gibt das Ergebnis zeilenweise aus.

getContents

Die Funktion `getContents :: IO String` liest die gesamte Benutzereingabe auf einmal ein. Dank Lazy Evaluation fragt GHC den Benutzer aber erst, wenn die nächste Zeile zur Bearbeitung wirklich benötigt wird:

```
import Data.Char

main = do
  input <- getContents
  let shorti   = shortLinesOnly input
      bigshort = map toUpper shorti
  putStr bigshort

shortLinesOnly :: String -> String
shortLinesOnly = unlines . shortfilter . lines
  where
    shortfilter = filter (\line -> length line < 11)
```

INTERACT

Die Funktion `interact :: (String -> String) -> IO ()` erlaubt uns, dies noch knapper auszudrücken:

```
import Data.Char

main = interact mangleinput
  where
    mangleinput = (map toUpper) . shortLinesOnly



shortLinesOnly :: String -> String
shortLinesOnly = unlines . shortfilter . lines
  where
    shortfilter = filter (\line -> length line < 11)
```

Dateizugriff

Das Modul `System.IO` bietet Varianten der IO-Funktionen für den Zugriff auf verschiedene Ein-/Ausgabegeräte an.
Die Varianten erwarten ein zusätzliche Argument des Typs `Handle`:

```
hPutStr      ::             Handle -> String -> IO ()
hPutStrLn    ::             Handle -> String -> IO ()
hPrint       :: Show a =>   Handle -> a -> IO ()
hGetLine     ::             Handle -> IO String
hGetContents ::             Handle -> IO String
```

Das Modul exportiert auch `stdin,stdout,stderr :: Handle`.
Es gilt:

```
putStr  = hPutStr stdout
getLine = hGetLine stdin
```

Datei Handles

Handles kann man auf verschiedene Arten bekommen:

```
readFile   :: FilePath -> IO String.
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
openFile   :: FilePath -> IOMode -> IO Handle
hClose   :: Handle -> IO ()

withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
```

- `type FilePath = String` $\Rightarrow$ Betriebssystem abhängig
- `writeFile` löscht Datei beim Öffnen
- `data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode deriving (Enum,Ord,Eq,Show,`
- `withFile` schließt die Datei in jedem Falle

Beispiele Dateizugriff:

Beispiel 1

```
import System.IO
import Data.Char
main = do
  contents <- readFile "whisper.txt"
  writeFile "shout.txt" (map toUpper contents)
```

Beispiele Dateizugriff:

Beispiel 1

```
import System.IO
import Data.Char
main = do
  contents <- readFile "whisper.txt"
  writeFile "shout.txt" (map toUpper contents)
```

Beispiel 2

```
main = do
  hIn  <- openFile "whisper.txt" ReadMode -- Öffnen
  hOut <- openFile "shout.txt" WriteMode
  -- Arbeiten
  input <- hGetContents hIn
  let biginput = map toUpper input
  hPutStrLn hOut biginput
  hClose hIn                              -- Schliessen
  hClose hOut
```

Beispiele Dateizugriff:

Beispiel 3

```
main = do
  withFile "something.txt" ReadMode (\handle -> do
      contents1 <- hGetContents handle
      let contents2 = foo contents1
      putStr contents2
  )
```

Dabei kann man sich `withFile` vorstellen als:

```
withFile' :: FilePath -> IOMode -> (Handle -> IO a)
                                -> IO a
withFile' path mode f = do
    handle <- openFile path mode
    result <- f handle
    hClose handle
    return result
```

Verfeinerter Zugriff

Positionieren

```
hGetPosn :: Handle -> IO HandlePosn
hSetPosn :: HandlePosn -> IO ()
hSeek :: Handle -> SeekMode -> Integer -> IO ()
```

- Nicht alle Handle-Arten unterstützen Positionierung
- Datentyp `HandlePosn` nur in Typklassen `Eq` und `Show`. Kann man sich also nur merken und wiederverwenden.

Pufferung

Man kann üblicherweise auch die Pufferung beeinflussen. `hFlush` erzwingt das leeren des Puffers.

```
hSetBuffering :: Handle -> BufferMode -> IO ()
hGetBuffering :: Handle -> IO BufferMode
hFlush :: Handle -> IO ()
```

Warnung: Vorgestellte Funktionen nur für einfache I/O-Aufgaben verwenden. Bei großen Dateien bzw. Bechmarks (!) sollte Modul `Data.Bytestring` oder `Data.Text` verwendet werden. Beide Module bieten nahezu identische Funktionen wie `Data.String`

Zusammenfassung

- Haskell erlaubt I/O im imperativen Stil;
  unter der Haube bleibt alles rein funktional dank Monaden
- IO-Aktionen verändern den Zustand der Welt, welche zwischen allen ausgeführten IO-Aktionen herumgereicht wird
- Die DO-Notation nimmt uns das explizite Fädeln der Welt ab; wird mit Aktions-Komposition zu normalen Haskell übersetzt
- IO-Aktionen sind Monaden