

FORTGESCHRITTENE FUNKTIONALE PROGRAMMIERUNG

TEIL 4: APPLIKATIVE FUNKTOREN & MONADEN

Steffen Jost

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

16. Mai 2017



1 APPLIKATIVE FUNKTOREN

- Maybe Instanz
- Listen Instanz
- Zip-Listen Instanz
- Liften
- Funktionen Instanz
- Zusammenfassung
Applicative

2 MONADEN

- Do
- Maybe Monade
- Listen Monade
- MonadPlus

- Monadische Funktionen
- Zusammenfassung
Monad

3 ANWENDUNG

- Zustand
- Monad Transformer
- ST
- Fehlermonade
- Nichtdeterminismus

4 I/O

- Main
- IO-Funktionen
- Datei Operationen
- Zusammenfassung



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.



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 $v == \text{pure id } \langle * \rangle v$

KOMPOSITION $u \langle * \rangle (v \langle * \rangle w) == \text{pure } (.) \langle * \rangle u \langle * \rangle v \langle * \rangle w$

HOMOMORPHIE $\text{pure } f \langle * \rangle \text{pure } x == \text{pure } (f x)$

INTERCHANGE $u \langle * \rangle \text{pure } y == \text{pure } (\backslash f \rightarrow f y) \langle * \rangle u$

Für die Definition $f \langle \$ \rangle x = \text{pure } f \langle * \rangle x$ 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 ??:

```
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)]
```

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 `<*>`:

$$f \langle \$ \rangle x_1 \langle * \rangle \dots \langle * \rangle x_n$$

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]]
```

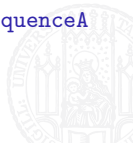
TYPKLASSE DATA.TRAVERSABLE

```
class (Functor t, Foldable t) => Traversable t where
  {-# MINIMAL traverse | sequenceA #-}
  traverse  :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  sequence  :: Monad m => t (m a) -> m (t a)
  mapM      :: Monad m => (a -> m b) -> t a -> m (t b)
```

GESETZE

- Natürliche Transformation:
`t.sequenceA == sequenceA . fmap t`
- Identität: `sequenceA . fmap Identity == Identity`
- Komposition:
`sequenceA . fmap Compose = Compose . fmap sequenceA . sequenceA`

Verallgemeinert links-nach-rechts durchlaufen von Listen



TYPKLASSE DATA.FOLDABLE

```
class Foldable t where           {-# MINIMAL foldMap | foldr #-}
  fold      :: Monoid m => t m -> m
  foldMap   :: Monoid m => (a -> m) -> t a -> m
  foldr     :: (a -> b -> b) -> b -> t a -> b
  foldl     :: (b -> a -> b) -> b -> t a -> b
  toList    :: t a -> [a]
  null      :: t a -> Bool
  length    :: t a -> Int
  elem      :: Eq a => a -> t a -> Bool
  sum, product :: Num a => t a -> a
```

Sowie Varianten `foldr'`, `foldl'`, `foldr1`, `foldl1`, etc.

GESETZE

- Identität: `fold == foldMap id`
- Funktoren-Komposition:
`foldMap f . fmap g == foldMap (f . g)`
- Gesetze für `foldMap` und `foldr/foldl` `foldr f z t == foldMap`

Verallgemeinert Zusammenfalten von Listen



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 $((\rightarrow) r)$ hat den erwarteten Kind $* \rightarrow *$, 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
```

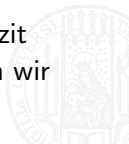
```
eval (Var x) env = fetch x env
```

```
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
- Spracherweiterung `ApplicativeDo` erlaubt Do-Notation auch für Applicative (Vgl.4.27)
- 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?

BEOBSACHTUNG

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 & Effizienz)!

MONADISCHE KOMPOSITION

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

$$\begin{aligned} (>=>) &:: \text{Monad } m \Rightarrow (a \rightarrow m \ b) \rightarrow (b \rightarrow m \ c) \rightarrow (a \rightarrow m \ c) \\ f >=> g &= \backslash x \rightarrow f \ x \ >>= \ g \end{aligned}$$


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!



SPRACHERWEITERUNG APPLICATIVE DO

Spracherweiterung `ApplicativeDo` erlaubt Do-Notation auch für nicht-monadische applikative Funktoren. GHC versucht dann, möglichst (applikative) Funktoren einzusetzen.

BEISPIEL:

```
mor :: Maybe Bool -> Maybe Bool -> Maybe Bool
mor mx my = do x <- mx
              y <- my
              return $ x || y
```

wird dann interpretiert als

```
mor mx my = (||) <$> mx <*> my
```

VORTEILE Im Gegensatz zu Monaden darf der Kompiler die Reihenfolge der Auswertung optimieren. (Applikative) Funktoren besitzen auch meist effizientere Implementierungen.

NACHTEILE Deklaration der Instanzen für `Functor`, `Applicative` und `Monad` sollte keine Do-Notation nutzen, da dies meist zu Endlosschleifen führt. Aufwendigeres Kompilieren. -foptimal-applicative-do

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
   act y
```

```
do act x
```

② Rechts-Identität

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

```
do mval
```

③ Assoziativität

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

```
do y <- do
           x <- mval
           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
   act y
```

```
do act x
```

② Rechts-Identität

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

```
do act x
```

③ Assoziativität

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

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

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



FORTGESCHRITTENE FUNKTIONALE PROGRAMMIERUNG

TEIL 4: APPLIKATIVE FUNKTOREN & MONADEN

Steffen Jost

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

23. Mai 2017



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.



LET VERSUS <-

Unterschied zwischen `let` und `<-` innerhalb eines Do-Blocks:

```
01 bar :: (Monad m) => m a -> m a
02 bar ma = do
03     x <- ma      -- x :: a
04     let y = ma  -- y :: m a
05     z <- y      -- z :: a
06     return z
```

ZEILE 04 rein funktionale Berechnung ohne Seiteneffekt

Z. 03 & 05 Seiteneffekte der monadischen Operation geschehen

BEISPIEL

Wenn wir `bar` mit einem Argument aufrufen, welches eine Bildschirmausgabe veranlasst, so findet diese nur in Zeilen 03 und 05 statt. z.B. `bar (print 42)`

HÄUFIGE MUSTER

APPLIKATIVE FUNKTOREN NICHT VERGESSEN:

<pre>do x <- mx y <- my return (f x y)</pre>	<pre>f <\$> mx <*> my</pre>
--	---

RECHTS-IDENTITÄT NICHT VERGESSEN:

<pre>do x <- mx y <- my x return y</pre>	<pre>do x <- mx my x</pre>	<pre>mx >>= my</pre>
--	---------------------------------	----------------------------

<pre>do x <- mx return x</pre>	<pre>mx</pre>
-------------------------------------	---------------

DO-Notation vor allem nützlich bei unnatürlicher Reihenfolge, z.B. Ergebnis Aktion 1 wird nicht gleich für Aktion 2 gebraucht.



MAYBE ALS MONADE

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

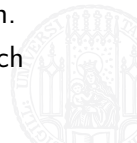
Diesen Datentyp können wir zur Monade machen:

```
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

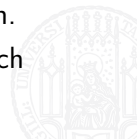
```
data Maybe a = Nothing | Just a
```

Diesen Datentyp können wir zur Monade machen:

```
instance Monad Maybe where
  return          = Just
  (>>=) 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)]
```



MONADPLUS

Das Konzept der Monade kann weiter verfeinert werden.

Listen und Maybe sind beide Instanzen der Typklasse `MonadPlus`:

```
class (Alternative m, 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.

Typklasse `Alternative` für applikative Funktoren mit Monoid-Struktur mit `empty` und `(<|>)`



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    :: Monad 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:

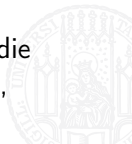
```
sequence  :: Monad m => [m a] -> m [a]
sequence_ :: [m a] -> m ()
```

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)
```

Verallgemeinert: `sequence :: (Traversable t, Monad m) =>`
`t (m a) -> m (t a)`



DEFINITION SEQUENCE

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

Verallgemeinert: `sequence :: (Traversable t, Monad m) =>`
`t (m a) -> m (t a)`

Erinnerung an applikative Version:

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



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 assoziiert 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.Trans.State`

SPEZIALFÄLLE:

- **Lesemonade** liest Zustand nur aus, z.B. globale Parameter
`Control.Monad.Trans.Reader`
- **Schreibmonade** beschreibt Zustand nur, z.B. logging
`Control.Monad.Trans.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
```



FORTGESCHRITTENE FUNKTIONALE PROGRAMMIERUNG

TEIL 4: APPLIKATIVE FUNKTOREN & MONADEN

Steffen Jost

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

30. Mai 2017



IMPLEMENTATION: SINGLE INT STATE

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, s1) = x s0
                    (State cont) = f val_x
                    in cont s1
```

```
runState :: State a -> Int -> (a,Int)
```

```
runState (State s) i = s i
```

```
put :: Int -> State ()
```

```
put s = State $ \_ -> ((),s)
```

```
get :: State Int
```

```
get = State $ \s -> (s,s)
```

```
inc :: State ()
```

```
inc = State $ \s -> ((),s+1)
```

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

```
fmap = liftM
```

```
pure = return
```

```
(<*>) = ap
```

```
| return $ show [x,y,z]
| > demo
| (" [1,21,203] ",203)
```

IMPLEMENTATION: SINGLE INT STATE

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, s1) = x s0
                    (State cont) = f val_x
                    in cont s1
```

```
runState :: State a -> Int -> (a,Int)
```

```
runState (State s) i = s i
```

```
put :: Int -> State ()
```

```
put s = State $ \_ -> ((),s)
```

```
get :: State Int
```

```
get = State $ \s -> (s,s)
```

```
inc :: State ()
```

```
inc = State $ \s -> ((),s+1)
```

```
| demo :: (String,Int)
```

```
| demo = runState (action 2) 1
```

```
| action :: Int -> State String
```

```
| action c = do x <- get
```

```
|           put $ x + 10*c
```

```
|           y <- get
```

```
|           put $ x + 100*c
```

```
|           inc
```

```
|           inc
```

```
|           z <- get
```

```
|           return $ show [x,y,z]
```

```
| > demo
```

```
| (" [1,21,203]",203)
```


IMPLEMENTATION: STATE MONADE

Verallgemeinert auf Zustand mit beliebigen Typ:

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

```
instance Monad (State s) where
```

```
  return x = State $ \s -> (x,s) -- :: a -> State s a
```

```
  (State x) >>= f = -- :: State s a -> (a -> State s b) ->
```

```
    State $ \s0 -> let (val_x, s1) = x s0 -- State s b
```

```
                    (State cont) = f val_x
```

```
                    in cont s1
```

```
runState :: State s a -> s -> (a,s)
```

```
runState (State s) i = s i
```

```
put :: s -> State s ()
```

```
put s = State $ \_ -> ((),s)
```

```
get :: State s s
```

```
get = State $ \s -> (s,s)
```

```
inc :: Num s => State s ()
```

```
inc = State $ \s -> ((),s+1)
```

```
| demo :: (String,Int)
```

```
| demo = runState (action 2) 1
```

```
| action :: Int -> State Int String
```

```
| action c = do x <- get
```

```
|           put $ x + 10*c
```

```
|           y <- get
```

```
|           put $ x + 100*c
```

```
|           inc
```

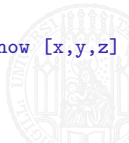
```
|           inc
```

```
|           z <- get
```

```
|           return $ show [x,y,z]
```

```
| > demo
```

```
| (" [1,21,203] ",203)
```



CONTROL.MONAD.TRANS.STATE

Da Haskell recht modular ist, können wir uns einfach in den Bibliotheken bedienen, anstatt alles von Hand zu programmieren:

```
type State s a -- Instanz der Klasse Monad
```

```
runState :: State s a -> s -> (a, s)
```

```
put      ::          s -> State s   ()
get      ::          State s       s
gets     ::          (s -> a) -> State s   a
modify   ::          (s -> s) -> State s   ()
```

- `runState` wendet Zustandsberechnung auf Anfangszustand an, liefert Endzustand und Ergebnis entspricht „unpure“
- `gets` erlaubt bequemes auslesen des Zustands, z.B. wenn der Zustand mehrere einzelne Werte enthält (z.B. Tupel)

IMPLEMENTATION: SINGLE INT STATE, VERSION 2

Bibliotheksunterstützung vereinfacht Verwendung:

```
import Control.Monad.Trans.State    -- NEU: get, put, modify,...

demo :: (String, Int)                -- unverändert
demo = runState (action 2) 1

action :: Int -> State Int String    -- unverändert
action c = do x <- get
              put $ x + 10*c
              y <- get
              put $ x + 100*c
              inc
              inc
              z <- get
              return $ show [x,y,z]

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



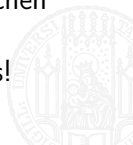
REIHENFOLGE MONADISCHER AKTIONEN

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 vorangegangenen 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!



MEHR ALS EINE MONADEN

Ein/Ausgabe (I/O) wird in Haskell mit der **IO**-Monade abgewickelt, welche man sich in etwa wie eine Zustandsmonade vorstellen kann:

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

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

Funktionen bekommen Welt als Kontext und dürfen diese lesen (z.B. Tastatur) und verändern (z.B. Bildschirm) Nur eine Analogie!

PROBLEM: Was ist, wenn man sowohl einen Zustand haben möchte und auch IO durchführen möchte?



MEHR ALS EINE MONADEN

Ein/Ausgabe (I/O) wird in Haskell mit der **IO**-Monade abgewickelt, welche man sich in etwa wie eine Zustandsmonade vorstellen kann:

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

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

Funktionen bekommen Welt als Kontext und dürfen diese lesen (z.B. Tastatur) und verändern (z.B. Bildschirm) Nur eine Analogie!

PROBLEM: Was ist, wenn man sowohl einen Zustand haben möchte und auch IO durchführen möchte?

ANTWORT: Entweder man schreibt sich eine eigene spezielle Monade dafür, oder man verwendet **Monad Transformer**. Diese Bibliotheken erlauben die Kombination von mehreren Monaden zu einem sog. Monaden-Stapel (engl. **Monad Stack**).



CONTROL.MONAD.TRANS.STATE

Die Standardbibliothek definiert die Zustandsmonade bereits über einen Transformer:

```
type State s a -- Instanz der Klasse Monad
```

```
runState :: State s a -> s -> (a, s)
```

```
put      ::          s -> State s ()  
get      ::          State s s  
gets    :: (s -> a) -> State s a  
modify  :: (s -> s) -> State s ()
```



CONTROL.MONAD.TRANS.STATE

Die Standardbibliothek definiert die Zustandsmonade bereits über einen Transformer:

```
type State s a = StateT s Identity a -- Nur Abkürzung
newtype StateT s m a                -- Instanz von Monad
```

```
runState :: State s a -> s -> (a, s)
runStateT :: StateT s m a -> s -> m (a, s)
put       :: Monad m => s -> StateT s m ()
get       :: Monad m => StateT s m s
gets     :: Monad m => (s -> a) -> StateT s m a
modify    :: Monad m => (s -> s) -> StateT s m ()
```

- `Identity` ist „leere“ Monade ($\gg=$) wird zu $\$$
- Die **innere Monade** `m` kann eine beliebige Monade sein oft `IO`



BEISPIEL: [] ALS INNERE MONADE

```
import Control.Monad.Trans.Class
import Control.Monad.Trans.State
```

```
type MyMonad = StateT Int []
demo :: [(String, Int)]
demo = runStateT (action 2) 1
```

```
action :: Int -> MyMonad String
```

```
action k = do
  c <- lift [-k,k]
  x <- get
  put $ x + 10*c
  y' <- get
  y <- lift [y',y'+1]
  put $ x + 100*c
  inc
  inc
  z <- get
  return $ show [x,y,z]
inc :: MyMonad ()
inc = modify (+1)
```

Innere Monade erreichbar mit

```
lift :: (Monad m, MonadTrans t)
      => m a -> t m a
```

Zur Erinnerung:

```
runStateT :: StateT s m a -> s
           -> m (a, s)
```

d.h. in unserem Fall

```
demo :: [(String, Int)]

> demo
[(" [1,-19,-197]",-197)
,(" [1,-18,-197]",-197)
,(" [1,21,203]",203)
,(" [1,22,203]",203)]
```



BEISPIEL: STATET BOOL [] ALS INNERE MONADE

```

type MyMonad =
    StateT Int (StateT Bool [])
demo :: [((String, Int), Bool)]
demo =
    let out= runStateT (action 2) 1
        in runStateT out True

action :: Int -> MyMonad String
action k = do
    c <- lift $ lift [-k,k]
    x <- get
    put $ x + 10*c
    y <- get
    lift $ put $ x > y
    put $ x + 100*c
    inc
    inc
    z <- get
    return $ show [x,y,z]

```

lift erreicht nun mittlere
Monad (hier StateTBool),
lift.lift die innerste der drei
Monaden (hier [])
Es gilt

```

out :: StateT Bool []
      (String, Int)

```

```

demo :: [((String, Int), Bool)]

```

```

> demo
[(" [1,-19,-197] ",-197),True)
, (" [1,21,203] ",203),False)]

```

SCHLECHTES BEISPIEL: Besser
StateT (Int,Bool) verwenden!

BEISPIEL: READERT INT [] ALS INNERE MONADE

```
type MyMonad =
  StateT Int (ReaderT Int [])

demo :: [(String, Int)]
demo =
  let out= runStateT (action 2) 1
      in  runReaderT out 3

action :: Int -> MyMonad String
action k = do
  c <- lift $ lift [-k,k]
  x <- get
  put $ x + 10*c
  y <- get
  p <- lift ask
  put $ x + 100*c*p
  inc
  inc
  z <- get
  return $ show [x,y,z]
```

Auslesen Reader-Zustands mit

```
ask :: Monad m => ReaderT r m r
```

```
demo :: [(String, Int)]
```

```
> demo
```

```
[(" [1,-19,-597] ",-597)
, (" [1,21,603] ",603)]
```



THE NEEDLE IN THE MONAD-STACK

Besprochene Monaden aus `transformer` Paket; gehört jetzt zur Standardbibliothek.

PROBLEM Tief verschachtelte Monaden-Stapel werden schnell unhandlich und sind schlecht wartbar: `lift $ lift $ lift ...`

ABHILFEN

- 1 Spezialisierte Zugriffsfunktionen verwenden, z.B.

```
myask = lift ask.
```

Oft ist IO innerste Monade, viele Bibliotheken bieten an

```
liftIO :: IO a -> m a
```

- 2 Modul `Control.Monad.Trans.RWS` bietet *eine* fertige kombinierte Monade an, in der `ask`, `tell`, `put`, `get` gleichzeitig verwendbar sind.
- 3 Bessere Bibliothek verwenden, z.B. `monads-tf` erweitert `transformer`. Mit Spracherweiterung `TypeFamilies` und überladenen Instanzen kann auf explizite `lift` verzichtet werden.



BEISPIEL: MONADS-TF

```

type MyMonad =
  StateT Int (ReaderT Int [])
demo :: [(String, Int)]
demo =
  let out= runStateT (action 2) 1
      in runReaderT out 3
action :: Int -> MyMonad String
action k = do
  c <- lift $ lift [-k,k]
  x <- get
  put $ x + 10*c
  y <- get
  p <- ask
  put $ x + 100*c*p
  inc
  inc
  z <- get
  return $ show [x,y,z]
inc :: MyMonad ()
inc = modify (+1)

```

Benötigt `monads-tf` Paket:

```

import Control.Monad.State
import Control.Monad.Reader

```

Monaden-Reihenfolge nun egal;
eigene Monaden möglichst ganz
innen oder ganz außen.

```

demo :: [(String, Int)]
> demo
[("[1,-19,-597",-597),
 ("[1,21,603]",603)]

```

IDEE: Operationen wie `ask`
werden über Typklassen
bereitgestellt; eine enthaltene
Reader-Monade wird automatisch
zur `MonadReader` Instanz.

BEISPIEL: MONADS-TF

```

type MyMonad =
  ReaderT Int (StateT Int [])
demo :: [(String, Int)]
demo =
  let out= runReaderT (action 2) 3
      in runStateT out 1
action :: Int -> MyMonad String
action k = do
  c <- lift $ lift [-k,k]
  x <- get
  put $ x + 10*c
  y <- get
  p <- ask
  put $ x + 100*c*p
  inc
  inc
  z <- get
  return $ show [x,y,z]
inc :: MyMonad ()
inc = modify (+1)

```

Benötigt `monads-tf` Paket:

```

import Control.Monad.State
import Control.Monad.Reader

```

Monaden-Reihenfolge nun egal;
eigene Monaden möglichst ganz
innen oder ganz außen.

```

demo :: [(String, Int)]
> demo
[(" [1,-19,-597]",-597),
 (" [1,21,603]",603)]

```

IDEE: Operationen wie `ask`
werden über Typklassen
bereitgestellt; eine enthaltene
Reader-Monade wird automatisch
zur `MonadReader` Instanz.

ALTERNATIVE ZUSTANDSMONADE

Module `Control.Monad.ST` und `Data.STRef` aus Standardbibliothek bieten echte Referenzen:

```
data ST s a                                -- Instanz von Monad

runST :: (forall s. ST s a) -> a

newSTRef  :: a -> ST s (STRef s a)         -- Allokation

readSTRef :: STRef s a -> ST s a          -- Lesen
writeSTRef :: STRef s a -> a -> ST s ()   -- Schreiben
modifySTRef :: STRef s a -> (a -> a) -> ST s ()
```

HINWEIS: `forall` im Typ von `runST` ist ein Trick, der verhindert das eine Referenz als Ergebnis zurückgeben wird und in einem anderen `runST` verwendet wird. Typsystem stellt sicher: jedes `runST` hat eigene, unabhängige Referenzen!



BEISPIEL: ST MONADE

```

demo :: String
demo = runST $ do
    i <- newSTRef 1
    b <- newSTRef True
    action 2 i b

action :: Int -> STRef s Int ->
        STRef s Bool -> ST s String
action c i b = do
    x <- readSTRef i
    writeSTRef i $ x + 10*c
    y <- readSTRef i
    writeSTRef b $ x > y
    writeSTRef i $ x + 100*c
    inc i
    inc i
    z <- readSTRef i
    return $ show [x,y,z]

inc :: Num a => STRef s a -> ST s ()
inc r = modifySTRef r (+1)

```

Zuerst wird je eine Referenz auf ein `Int` und ein `Bool` erstellt.

Referenzen können beliebig gelesen und geschrieben werden; sind aber niemals leer.

```

> demo
"[1,21,203]"

```

In `demo` würde `return i` am Ende zu Typfehler führen. In `action` wäre das okay.

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 vorangegangenen 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 Zeilenvorschub 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, P`
- `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. Benchmarks (!) 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