Applikative Funktoren, Monaden und Ein- und Ausgabe in Haskell

Prof. Dr. David Sabel

LFE Theoretische Informatik

Letzte Änderung der Folien: 10. Dezember 2021 ♦ Die Folien basieren zum Teil auf Material von Dr. Steffen Jost, dem an dieser Stelle für die Verwendungserlaubnis herzlich gedankt sei.

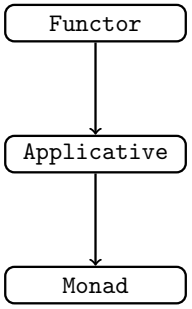
Ziele des Kapitels

- Was sind Applikative Funktoren?
- Was sind Monaden / Monadische Programmierung?
- Zustandsbasiertes Programmieren mit Monaden allgemein
- Programmierung: Ein- und Ausgabe in Haskell
- Monad-Transformer: „Vereinigung“ mehrerer Monaden
- Dabei: Anwendungen für Monaden

Monadisches Programmieren

- **Monadisches Programmieren**
  = Strukturierungsmethode, um Berechnungen zu komponieren
- Oft: um sequentiell ablaufende Programme zu implementieren
- Haskell verwendet u.a. Monaden zur Programmierung von Ein- und Ausgabe
- Begriff Monade entstammt der Kategorientheorie
  (Teilgebiet der Mathematik: Morphismen, Isomorphismen,...)
- Für die Programmierung:
  Monade ist ein Typkonstruktor + Operationen,
  wobei die sog. monadischen Gesetze gelten
- In Haskell: Umsetzung von Monaden durch die Typklasse `Monad`.
  Jeder Datentyp, der Instanz der Klasse `Monad` ist und die Gesetze erfüllt, ist eine Monade

Aktuelle Klassenstruktur



Daher: Wir beschäftigen uns erstmal mit `Applicative`

Applikative Funktoren

Funktoren, die man anwenden kann

Wiederholung: Functor

- **Functor** ≈ Datentypen über deren Inhalt man "mappen" kann:

```
class Functor (f :: * -> *) where        -- Bem.: Schreibweise mit Kind
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  (<$) = fmap . const
  {-# MINIMAL fmap #-}

-- vordefiniert:
(<$>) = fmap
```

- Mathematisch: Funktor = Strukturerhaltende Abbildung
- Gesetze: Sollten für jede Instanz von `Functor` gelten:

```
fmap id      = id
fmap (p . q) = (fmap p) . (fmap q)
```

- a `<$` o ersetzt in den Inhalt in o durch a, z.B. `10 <$ [1,2,3,5]` ergibt `[10,10,10,10]`

Applikative Funktoren: Motivation

Wiederholung: Instanz für `Maybe`:

```
instance Functor Maybe where
   fmap f Nothing  = Nothing
   fmap f (Just a) = Just (f a)
```

Beispiel: Addiere Werte innerhalb von `Maybe`, d.h. programmiere add sodass

- `(Just` a`)` ‘add‘ `(Just` b`)` = `Just` $(a+b)$
- `Nothing` herauskommt, sobald eines der Argumente `Nothing` ist

Brute-Force Methode:

```
add (Just a) (Just b) = Just (a+b)
add _        _        = Nothing
```

Entpacken und wieder zusammen packen!

Allgemeiner Ansatz:

```
add m1 m2 = (fmap (+) m1)  ???     m2
            ------------         ---------
          Maybe (Int -> Int)     Maybe Int
```

???: es fehlt Operationen zum „anwenden“

Applikative Funktoren: Motivation (2)

- Die Funktion zum Anwenden hat den Typ:

```
<*> :: Maybe (a -> b) -> Maybe a -> Maybe b
```

- Zum Vergleich: `($) :: (a -> b) -> a -> b`

  Daher ist `<*>` wie eine Applikation, aber im Maybe-Typ

- Implementierung von `<*>` für Maybe

```
Nothing  <*> _         = Nothing
(Just f) <*> something = fmap f something
```

- Damit lässt sich `add` generisch implementieren:

```
add m1 m2 = (fmap (+) m1) <*> m2
```

- Oder mit dem Synonym `<$>` für `fmap`:

```
add m1 m2 = (+) <$> m1 <*> m2
```

Die Klasse Applicative

```
class Functor f => Applicative (f :: * -> *) where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
  (*>) :: f a -> f b -> f b
  (<*) :: f a -> f b -> f a
  {-# MINIMAL pure, ((<*>) | liftA2) #-}
```

- `pure` verpackt beliebiges Objekt in den Datentyp
- `<*>` ist die sequentielle Anwendung,
- `*>` und `<*` verwerfen das linke bzw. rechte Ergebnis.
- Die Funktion `liftA2` lifted eine binäre Funktion.
  Beachte:

```
add             = liftA2 (+)
liftA2 f m1 m2  = (pure f) <*> m1 <*> m2
(<*>)           = liftA2 id
fmap f m        = (pure f) <*> m
```


Funktionen liften

Aus der Klasse `Applicative` und dem Modul `Control.Applicative`:

```
-- binäre Funktion liften:
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
-- ternäre Funktion liften:
liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
```

Z.B. dreistellige Addition in Maybe:

```
add3 :: Maybe Int -> Maybe Int -> Maybe Int -> Maybe Int
add3 = liftA3 (\x y z -> x + y + z)
```

oder direkt mit der sequentiellen Applikation `<*>` und `fmap`:

```
*> fmap (\x y z -> x+y+z)  (Just 10) <*> (Just 20) <*> (Just 30)
Just 60
*> (\x y z -> x+y+z) <$> (Just 10) <*> (Just 20) <*> (Just 30) -- alternativ
Just 60
*> pure (\x y z -> x+y+z) <*> (Just 10) <*> (Just 20) <*> (Just 30) -- alternativ
Just 60
```


Funktionen liften (2)

Allgemein: Lifte n-stellige Funktion mit einem Applikativen Funktor:

$$f \texttt{ <\$> } arg_1 \texttt{ <*> } arg_2 \texttt{ ... <*> } arg_n$$

Das Verhalten dabei ist:

Sequentielles Anwenden der Argumente auf die Funktion f, verpackt in der Struktur.

Alternativ:

$$\texttt{pure } f \texttt{ <*> } arg_1 \texttt{ <*> } arg_2 \texttt{ ... <*> } arg_n$$

Maybe-Instanz und Gesetze

Schon gesehen:

```
instance Applicative Maybe where
  pure a = Just a
  Nothing <*> _ = Nothing
  (Just f) <*> something = fmap f something
```

Aber: Die folgenden Gesetze müssen für jede Applicative-Instanz gelten:

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

Nachrechnen der Gesetze für Maybe

Identität: Zeige `pure id <*> v = v`

Es gilt: `pure id <*> v = Just id <*> v = fmap id v = v`

wobei die letzte Umformung aus dem 1. Gesetz für Funktoren folgt.

Komposition: Zeige `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`

Wir vereinfachen zunächst:

```
pure (.) <*> u <*> v <*> w = Just (.) <*> u <*> v <*> w = fmap (.) u <*> v <*> w
```

Nun unterscheide die Fälle:

- `u = Nothing`. Dann gilt

```
fmap (.) Nothing <*> v <*> w = Nothing <*> v <*> w
                             = Nothing <*> w
                             = Nothing
                             = Nothing <*> (v <*> w)¨
```


Nachrechnen der Gesetze für Maybe (2)

- `u = Just u'`. Dann gilt zunächst

```
fmap (.) (Just u') <*> v <*> w
= (Just ((.) u')) <*> v <*> w
= fmap ((.) u') v <*> w
```

Nun unterscheide erneut:

- `v = Nothing`. Dann gilt

```
fmap ((.) u') Nothing <*> w
= Nothing <*> w
= Nothing = fmap u' Nothing
          = fmap u' (Nothing <*> w)
          = (Just u') <*> (Nothing <*> w)
```

- `v = Just v'`. Dann gilt

```
fmap ((.) u') (Just v') <*> w
= Just (u' . v') <*> w
= fmap (u' . v') w =† fmap u' (fmap v' w)
                   = fmap u' (Just v' <*> w)
                   = Just u' <*> (Just v' <*> w)
```

wobei die mit † markierte Umformung aus dem 2.Gesetz für Funktoren folgt.

Nachrechnen der Gesetze für Maybe (3)

Beachte, dass wir auch die Fälle $u = \bot$ und $v = \bot$ betrachten müssten, um alle Ausdrücke abzudecken (nämlich auch die nichtterminierenden). Wir verzichten hier darauf.

Homomorphismus: Zeige `pure f <*> pure x = pure (f x)`

Es gilt

```
pure f <*> pure x = Just f <*> pure x = fmap f (pure x)= fmap f (Just x)
                                                       = Just (f x) = pure (f x).
```


Nachrechnen der Gesetze für Maybe (4)

Austausch: Zeige `u <*> pure y = pure (\x -> x y) <*> u`
Wir betrachten zwei Fälle:

- `u = Nothing`. Dann:

```
Nothing <*> pure y = Nothing = fmap (\x -> x y) Nothing
                             = Just (\x-> x y) <*> Nothing
                             = pure (\x-> x y) <*> Nothing
```

- `u = Just u'`. Dann:

```
(Just u') <*> pure y
= fmap u' (pure y)
= fmap u' (Just y)
= Just (u' y)  = Just ((\x -> x y) u')
= fmap (\x -> x y) (Just u')
= Just (\x -> x y) <*> (Just u')
= pure (\x -> x y) <*> (Just u')
```

Beispiel

- Implementiere eine Funktion

```
response :: String -> String -> Maybe String
response yearOfBirth yearToday = ...
```

  die Geburtsjahr und heutiges Jahr als Text erhält und Alter als Text berechnet.

- Wenn Texte nicht als Zahlen erkennbar, dann liefere Nothing.
- Verwende: `readMaybe :: Read a => String -> Maybe a`

Beispiel

Direkte Lösung:

```
calculateAge :: Int -> Int -> Int
calculateAge byear tyear =  tyear - byear

response :: String -> String -> Maybe String
response yearOfBirth yearToday =
 case readMaybe yearOfBirth of
   Just byear -> case readMaybe yearToday of
                   Just tyear ->  (Just $ show $ calculateAge  byear tyear)
                   Nothing -> Nothing
   Nothing -> Nothing
```

Unübersichtlich durch Einpacken / Auspacken und case-Schachtelungen

Besser mit <*>:

```
response yearOfBirth yearToday =
  show <$> ((calculateAge) <$> (readMaybe yearOfBirth) <*> (readMaybe yearToday))
```


Beispiel: Bemerkung

```
response yearOfBirth yearToday =
  show <$> ((calculateAge) <$> (readMaybe yearOfBirth) <*> (readMaybe yearToday))

calculateAge :: Int -> Int -> Int
calculateAge byear tyear =  tyear - byear
```

- Applicative bietet keine Möglichkeit, die verpackten Werte zwischendrin zu inspizieren
- Beispiel: Rufe `calculateAge` mit umgekehrten Argumenten auf,
  wenn Geburtsjahr größer als aktuelles Jahr
- Das geht nicht mit Applikativen Funktoren (aber mit Monaden!)

Listeninstanz für Applicative

```
instance Applicative [] where
   pure x          = [x]
   fs <*> xs       = [f x | f <- fs, x <- xs]
   liftA2 f xs ys = [f x y | x <- xs, y <- ys]
   xs *> ys        = [y | _ <- xs, y <- ys]
```

- `pure` verpackt Element in Liste
- `(<*>) :: [a -> b] -> [a] -> [b]`:
  Durch `fs <*> xs` werden alle möglichen `(f x)` mit `f` aus `fs` und `x` aus `xs` erzeugt.
- Auffassung: Nichtdeterministische Berechnung mit Erzeugen aller Möglichkeiten

Listeninstanz für Applicative (2)

Beispiele:

```
*Main> (+) <$> [1,2,3] <*> [4,5,6]
[5,6,7,6,7,8,7,8,9]

*Main> (++) <$> ["A","B"] <*> ["c","d","e"]
["Ac","Ad","Ae","Bc","Bd","Be"]

*Main> [(*),(+)] <*> [1,2] <*> [3,4]
[3,4,6,8,4,5,5,6]
```


Alternative Listeninstanz: Zip-Listen

Statt Kombinationen, zippe Listen, d.h.

`fs <*> xs` erzeugt alle `(f x)` mit `(f,x)` aus `zip fs xs`

Implementierung mit `newtype`:

```
newtype ZipList a = ZipList {getZipList :: [a]} deriving Show

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


Alternative Listeninstanz: Zip-Listen (2)

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)]
```


Alternative Listeninstanz: Zip-Listen (3)

- `pure x = ZipList $ repeat x` erzeugt unendliche viele Kopien von `x`
- Grund: Diese Liste kann einen Wert an jeder Position produzieren
- Durch `zipWith` werden überschüssige verworfen, denn kürzeste Liste bestimmt Länge der Ergebnisliste
- Z.B. wird das 1. Gesetz Identität `pure id <*> v` $=$ `v` falsch, wenn man definiert

  `pure x = ZipList [x]`

  Gegenbeispiel: `v = [1,2]`

Alternative Listeninstanz: Zip-Listen (4)

Simuliere `zipN` mit `ZipList`-Instanz:

```
*Main> getZipList $ (,) <$> ZipList [1..10] <*> ZipList [11..20]
[(1,11),(2,12),(3,13),(4,14),(5,15),(6,16),(7,17),(8,18),(9,19),(10,20)]

*Main> getZipList $  (,,,) <$> ZipList "Hund"
                           <*> ZipList "Maus"
                            <*> ZipList [1,2,3,4]
                             <*> ZipList [False,True,True,False]
[('H','M',1,False),('u','a',2,True),('n','u',3,True),('d','s',4,False)]
```


Applicative-Instanz für Funktionen

Funktionen können zu Instanzen von `Functor` und `Applicative` gemacht werden:

```
instance Functor ((->) e) where
  fmap :: (a -> b) -> (e -> a) -> (e -> b)
  fmap = (.)

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

Die Funktionen `pure` und `<*>` mit den obigen Typen sind übrigens im Lambda-Kalkül auch als die Kombinatoren $\mathbf{K}$ und $\mathbf{S}$ bekannt.

Applicative Instanz für Funktionen: Beispiel

Beispiel: Berechnung arithmetischer Ausdrücke mit Umgebung:

```
data Exp v = Var v | Val Int | Neg (Exp v) | Add (Exp v ) (Exp v )
type Env v = [(v,Int)]

fetch :: (Eq v) => v -> Env v -> Int  -- Variable nachschlagen
fetch x env
 | Just val <- lookup x env = val
 | otherwise                = error "Variable not found"

eval  :: Exp String -> Env String -> 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)
```

Umgebung `env` wird überall durchgeschleift. Dank der `(->) e`-Instanz für `Applicative` geht es jedoch auch ohne dies zu tun (die Umgebung `env` wird dabei größtenteils versteckt).

Applicative Instanz für Funktionen: Beispiel (Forts.)

```
import Control.Applicative
eval  :: Exp String -> Env String -> 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)

fetch :: (Eq v) => v -> Env v -> Int  -- Variable nachschlagen
fetch x = maybe (error "Variable not found") id <$> (lookup x)
```

Was hier passiert:

```
<*> :: (e -> (a -> b)) -> (e -> a) -> (e -> b)
eval  (Add p q)      =  (+) <$> (eval p)     <*>    (eval q)
                                :: Env String -> Int      :: Env String -> Int
                        :: Env String -> (Int -> Int)
                                Env String -> Int
```

Die Traversable-Klasse

Liften des Cons-Operators `(:)` für Listen ergibt:

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

Dadurch wird eine Liste von verpackten Werten zu einer verpackten Liste von Werten:

```
> sequenceA [Just 3, Just 2, Just 1]
Just [3,2,1]
(:) <$> Just 3 <*> ((:) <$> Just 2 <*> ((:) <$> Just 1 <*> pure []))
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]]
```


Die Traversable-Klasse (2)

```
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)
  -- Default-Implementations
  traverse f = sequenceA . fmap f
  sequenceA  = traverse id
  sequence   = sequenceA
  mapM       = traverse
```

Diese Typklasse steht für Typen, die linear durchlaufen werden können.
Gesetze (siehe Dokumentation von `Data.Traversable`):

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

Paare als Applikative Funktoren

Functor-Instanz (analog zu Either, erste Komponente wird ignoriert):

```
instance Functor ((,) a) where
   fmap f (x,y) = (x, f y)
```

Wie macht man einen applikativen Funktor?

- `(u,(+3)) <*> (v,5)` soll `(w,8)` ergeben.
- Wie wird aus `u` und `v` welches `w`?

Lösung: `Monoid`-Instanz der ersten Komponenten, um `u` und `v` mit `mappend` zu verknüpfen:

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

Beispiel:

```
*> (\x y z -> x + y + z) <$> (Product 3,9) <*> (Product 2,5) <*> (Product 3,10)
(Product {getProduct = 18},24)
```


Zusammenfassung: Applikative Funktoren

- Applikative Funktoren erlauben die Behandlung beliebig stelliger verpackter Funktionen
- Behandlung der Verpackung (Struktur) wird versteckt
  Funktionsanwendung ohne Verpackung: `f x1  x2 ... xn`
  Funktionsanwendung mit Verpackung: `f <$> x1 <*> x2 <*> ... <*> xn`
- Typen dazu:
  Funktionsanwendung ohne Verpackung: `($)  :: (a -> b) -> a -> b`
  Funktionsanwendung mit Verpackung:
  `(<*>) :: Applicative f => f (a -> b) -> f a -> f b`
- `f <$> x1 <*> x2 <*> ... <*> xn` ist äquivalent zu

  `pure f <*> x1 <*> x2 <*> ... <*> xn`

Monaden

Zwischenergebnisse beobachtbar machen

Applicative: Struktur kann nicht verlassen werden

Es gibt **keine** allgemeine Funktion, um Struktur zu verlassen:

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

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

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

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

Daher verbleibt man im Allgemeinen in der Struktur / Verpackung.

Typklasse: Monad

```
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  {-# MINIMAL (>>=) #-}
  m >> k      = m >>= \_ -> k
  return      = pure
```

- Instanzen müssen `>>=` definieren
- Ein Objekt vom Typ `m a` (mit `m` eine Instanz von `Monad`) heißt monadische Aktion.
- Operator `>>=` wird „bind“ ausgesprochen, verkettet zwei monadische Aktionen.
  Zweite Aktion darf auf das verpackte Ergebnis der ersten zugreifen.
- `return` verpackt Ausdruck in der Monade.
- Operation `>>` heißt „then“. Ähnlich zu `>>=`, aber:
  Zweite Aktion verwendet Ergebnis der ersten Aktion nicht.

MonadFail: Monaden mit Fehlerausgang

Unterklasse von Monad:

```
class Monad m => MonadFail (m :: * -> *) where
  fail :: String -> m a
  {-# MINIMAL fail #-}
```

Die Operation `fail` bietet einen Fehlerausgang mit Fehlerstring.

Maybe als Monade

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

instance MonadFail Maybe where
    fail _          = Nothing
```


Beispiel

Das Beispiel kennen wir bereits, zur Erinnerung:

```
response :: String -> String -> Maybe String
response yearOfBirth yearToday =
 case readMaybe yearOfBirth of
   Just byear -> case readMaybe yearToday of
                   Just tyear -> Just $ show $ calculateAge  byear tyear
                   Nothing -> Nothing
   Nothing -> Nothing

calculateAge :: Int -> Int -> Int
calculateAge byear tyear =  tyear - byear
```


Beispiel mit Monaden-Instanz

```
responseM :: String -> String -> Maybe String
responseM yearOfBirth yearToday =
 (readMaybe yearOfBirth) >>=
     (\byear -> readMaybe yearToday >>=
       \tyear -> return $ show $ calculateAge byear tyear)
```

Erweiterung: Zugriff auf Ergebnisse steuert Kontrollfluss

```
responseM' :: String -> String -> Maybe String
responseM' yearOfBirth yearToday =
 (readMaybe yearOfBirth) >>=
     (\byear -> readMaybe yearToday >>=
       \tyear -> return $ show $
                 (if byear >= tyear
                  then calculateAge byear tyear -- richtig herum
                  else calculateAge tyear byear -- vertausche
                 ))
```

Das war mit Applicative nicht möglich!

Applicative vs. Monad: Beispiel

Implementiere `applicativeIf :: Applicative f => f Bool -> f a -> f a -> f a`,

- `applicativeIf aCond aThen aElse` geht in `aThen` oder `aElse`, je nach Auswertungsergebnis von `aCond`

Versuch:

```
applicativeIf :: Applicative f => f Bool -> f a -> f a -> f a
applicativeIf aCond aThen aElse = ifte <$> aCond <*> aThen <*> aElse
   where ifte a b c = if a then b else c
```

Aufrufe:

```
<*> applicativeIf (Just True) (Just 1) (Just 2)
Just 1
*> applicativeIf (Just False) (Just 1) (Just 2)
Just 2
*> applicativeIf (Just True) (Just 1) Nothing
Nothing -- war nicht gewollt
*> applicativeIf (Just False) Nothing (Just 2)
Nothing -- war nicht gewollt
```

Problem: Applicative kann nicht auf Zwischenergebnisse zugreifen!

Applicative vs. Monad: Beispiel (2)

Lösung mit monadischer Programmierung

```
monadicIf :: Monad m => m Bool -> m b -> m b -> m b
monadicIf aCond aThen aElse = aCond >>= \r -> if r then aThen else aElse
```

Diese Implementierung verhält sich, wie es gewünscht war:

```
*> monadicIf  (Just True) (Just 1) (Just 2)
Just 1
*> monadicIf  (Just False) (Just 1) (Just 2)
Just 2
*> monadicIf  (Just True) (Just 1) Nothing
Just 1
*> monadicIf  (Just False) Nothing (Just 2)
Just 2
```


Monaden sind Applikative Funktoren

Jede Monade ist ein Applikativer Funktor, denn es gilt sowohl

```
fmap f mx   ==   mx >>= return . f
```

als auch

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

Damit kann man für jede Monaden-Instanz auch `Functor` und `Applicative`-Instanzen erstellen.

Die Umkehrung gilt nicht, denn nicht alle Applikativen Funktoren sind auch Monaden. Z.B. ist `ZipList` keine Monade.

Do-Notation

Syntaktischer Zucker: `do` *DoBlock*

wobei *DoBlock* Sequenz von monadischen Aktionen, mit Spezialsyntax:

- `x <- aktion` darf verwendet werden um auf das Ergebnis der Aktion zuzugreifen.
- `let`-Ausdrücke der Form `let x = e`
- Patterns `pat <- aktion` oder `let pat = e`:
  Verlangen Instanz der Klasse `MonadFail`,
  denn Fehlschlagen des Matches, ruft `fail`-Funktion auf

Do-Notation: Beispiel

Die Funktion `response`:

```
responseM'' :: String -> String -> Maybe String
responseM'' yearOfBirth yearToday =
 do
    byear <- readMaybe yearOfBirth
    tyear <- readMaybe yearToday
    return $ show
      (if byear >= tyear
        then calculateAge byear tyear
        else calculateAge tyear byear)
```

Do-Notation entfernen

Übersetzung in do-freien Code mit den folgenden Regeln (ohne Pattern-Match):

```
do { x <- e ; s }    = e >>= (\x -> do { s })
do { e; s }          = e >> do { s }
do { e }             = e
do { let binds; s } = let binds in do { s }
```


Beispiel zur do-Entfernung

```
responseM'' yearOfBirth yearToday =
 do
  byear <- readMaybe yearOfBirth
  tyear <- readMaybe yearToday
  return $ show
    (if byear >= tyear
      then calculateAge byear tyear
      else calculateAge tyear byear)
```

→

```
responseM'' yearOfBirth yearToday =
 (readMaybe yearOfBirth) >>= \byear ->
   do
    tyear <- readMaybe yearToday
    return $ show
      (if byear >= tyear
        then calculateAge byear tyear
        else calculateAge tyear byear)
```

→

```
responseM'' yearOfBirth yearToday =
 (readMaybe yearOfBirth) >>= \byear ->
  (readMaybe yearToday) >>= \tyear ->
    do
     return $ show
      (if byear >= tyear
        then calculateAge byear tyear
        else calculateAge tyear byear)
```

→

```
responseM'' yearOfBirth yearToday =
 (readMaybe yearOfBirth) >>= \byear ->
  (readMaybe yearToday) >>= \tyear ->
   return $ show
     (if byear >= tyear
       then calculateAge byear tyear
       else calculateAge tyear byear)
```


Applikativer Funktor statt monadischem do

Man sieht oft:

```
do
 f <- mf
 x <- mx
 return (f x)
```

- Verwendung der Monade ist hier unnötig, da

  `mf <*> mx`

  dasselbe liefert und kürzer und prägnanter ist.
- Insbesondere wird durch letztere Notation auch ausgedrückt, dass die Effekte der Ausführung von `mf` und `mx` sich nicht gegenseitig beeinflussen können!

ApplicativeDo

Mit Spracherweiterung `ApplicativeDo` kann die do-Notation auch für applikative Funktoren verwendet werden.

GHC versucht innerhalb der `do`-Blöcke – falls möglich – auf Applicative zurück zu greifen

```
Prelude> :set -XApplicativeDo
Prelude> let fun mf mx = do {f <- mf; x <- mx; return $ f x}
Prelude> :type fun
fun :: Applicative f => f (t -> b) -> f t -> f b
Prelude> :unset -XApplicativeDo
Prelude> let fun mf mx = do {f <- mf; x <- mx; return $ f x}
Prelude> :type fun
fun :: Monad m => m (t -> b) -> m t -> m b
```

ApplicativeDo (2)

Weiteres Beispiel:

```
do a <- ma          -- \_Block 1
   b <- mb          -- /
   c <- (mc a)      -- \_Block 2
   d <- (md b)      -- /
   return (c,d)
```

- Berechnungen `a <- ma` und `b <- mb` hängen **nicht** voneinander ab
- Berechnungen `c <- mc a` und `d <- md b` hängen **nicht** voneinander ab
- Block 2 **hängt** von Block 1 **ab** (da `a` und `b` verwendet werden)

Mögliche Übersetzung:

```
((,) <$> ma <*> mb) >>= \(a,b) -> (,) <$> (mc a) <*> (md b)
```

GHC übersetzt mit `ApplicativeDo`-Erweiterung (etwas effizienter) in

```
join ((\a b -> (,) <$> (mc a) <*> (md b)) <$> ma <*> mb)
```

wobei `join mma = mma >>= \ma -> ma`

Monadische Gesetze

Erstes Gesetz: *„`return` ist links-neutral bzgl. `>>=`“*

$$\texttt{return x >>= f} \quad = \quad \texttt{f x}$$

Zweites Gesetz: *„`return` ist rechts-neutral bzgl. `>>=`“*

$$\texttt{m >>= return} \quad = \quad \texttt{m}$$

Drittes Gesetz: *„eingeschränkte Assoziativität von `>>=`“*

$$\texttt{m1 >>= (\x -> m2 x >>= m3)}$$
$$=$$
$$\texttt{(m1 >>= m2) >>= m3}$$

wobei $\texttt{x} \notin FV(\texttt{m2}, \texttt{m3})$

Maybe erfüllt die monadischen Gesetze

Wir setzen voraus, dass Reduktionen die Gleichheit erhalten!

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

1. Gesetz: lässt sich direkt nachrechnen:

```
   return x >>= f
=  Just x >>= f
=  f x
```


Maybe erfüllt die monadischen Gesetze

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

2. Gesetz: Fallunterscheidung:

Fall 1: `m` wertet zu `Nothing` aus:

```
Nothing >>= return  =  Nothing
```

Fall 2: `m` wertet zu `Just a` aus:

```
Just a >>= return  =  return a  =  Just a
```

Fall 3: Die Auswertung von `m` terminiert nicht.
Dann terminiert auch die Auswertung von `m >>= return` nicht.

Maybe erfüllt die monadischen Gesetze (3)

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

3. Gesetz: `(m1 >>= m2) >>= m3 = m1 >>= (\x -> m2 x >>= m3)`

Fall 1: `m1` wertet zu `Nothing` aus

```
(Nothing >>= m2) >>= m3 = Nothing >>= m3
= Nothing = Nothing >>= (\x -> m2 x >>= m3)
```

Fall 2: `m1` wertet zu `Just e` aus

```
(Just e >>= m2) >>= m3 = m2 e >>= m3
= (\x -> m2 x >>= m3) e = Just e >>= (\x -> m2 x >>= m3)
```

Fall 3: Die Auswertung von `m1` divergiert:
dann divergieren sowohl `(m1 >>= m2) >>= m3` als auch `m1 >>= (\x -> m2 x >>= m3)`.

Gesetz für MonadFail

Für Instanzen der Klasse `MonadFail` muss das Gesetz

```
fail s >>= f  = fail s
```

erfüllt sein.

Beispiel: Maybe

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

instance MonadFail Maybe where
  fail _ = Nothing
```

Gesetz erfüllt, denn: `fail s >>= f = Nothing >>= f = Nothing = fail s`

Die Listen-Monade

Listen sind Instanzen der Klassen `Monad` und `MonadFail`.

```
instance Monad [] where
   m >>= f  = concatMap f m
   return x = [x]

instance MonadFail [] where
   fail s   = []
```

- `(>>=) :: [a] -> (a -> [b]) -> [b]` und in `m >>= f` wird `f` auf alle Elemente von `m` angewendet. Liste der Ergebnisse werden konkateniert.
- Entspricht nichtdeterministischem Ausprobieren aller Möglichkeiten
- `fail`-Operation: Kein Ergebnis

List Comprehensions

List Comprehension `[x*y | x <- [1..10], y <-[1,2]]` kann mit Listenmonade programmiert werden:

```
do
  x <- [1..10]
  y <- [1,2]
  return (x*y)
```

Beispiel mit filter: `[x*y | x <- [1..10], y <-[1,2], x*y < 15]`.

```
do
  x <- [1..10]
  y <- [1,2]
  if (x*y < 15) then return () else fail ""
  return (x*y)
```

Verfeinerung – MonadPlus

`MonadPlus`= Monaden mit Auswahloperator mit Fehlerausgang (Monoid)

```
class (Alternative m, Monad m) => MonadPlus (m :: * -> *) where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Dabei ist `Alternative` ein Monoid für einen applikativen Functor:

```
class Applicative f => Alternative (f :: * -> *) where
  empty :: f a
  (<|>) :: f a -> f a -> f a
  some :: f a -> f [a]
  many :: f a -> f [a]
```


Verfeinerung – MonadPlus (2)

Die Instanzen für Listen sind:

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


instance Alternative [] where
    empty = []
    (<|>) = (++)
```

(siehe Dokumentation zu `Control.Monad` und `Control.Applicative`)

Listen mit MonadPlus

Mithilfe von `MonadPlus` kann man definieren:

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

List Comprehensions dadurch noch eleganter:

```
do
  x <- [1..10]
  y <- [1,2]
  guard (x*y < 15)
  return (x*y)
```


Nützliche Monaden-Funktionen (1)

Siehe auch `Control.Monad`!

Nützliche Operatoren:

- `(=<<) :: Monad m => (a -> m b) -> m a -> m b`
  Entspricht `(flip (>>=))`
- `(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c`
  Links-nach-rechts mon. Komposition, `(f >=> g) x` entspricht `do {y <- f x; g y}`
- `(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c`
  Rechst-nach-links mon. Komposition, `(f <=< g) x` entspricht `do {y <- g x; f y}`

Bemerkung: Monadische Gesetze mit `>=>`:

- Left identity: `return >=> g` = `g`
- Right identity: `f >=> return` = `f`
- Associativity: `(f >=> g) >=> h` = `f >=> (g >=> h)`

Nützliche Monaden-Funktionen (2)

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

`replicateM n act` führt die Aktion `act` `n`-Mal hintereinander aus und verknüpft die Ergebnisse in einer Liste.

```
forever   :: (Applicative f) => f a -> f b
forever a = let a' = a *> a' in a'
```

`forever a` führt monadische Aktion `a` unendlich lange wiederholt aus.

```
when      :: (Applicative m) => Bool -> m () -> m ()
when p s  =  if p then s else return ()
```

`when` verhält sich wie ein imperatives `if-then`

```
unless      :: (Applicative m) => Bool -> m () -> m ()
unless p s  =  if p then return () else s
```

analog zu `when` aber Bedingung muss falsch sein

Nützliche Monaden-Funktionen (3)

```
sequence :: (Monad m) => [m a] -> m [a]
sequence []          = return []
sequence (action:as) = do r <- action
                          rs <- sequence as
                          return (r:rs)
```

`sequence` führt Liste von mon. Aktionen sequentiell aus und sammelt Ergebnisse in Liste.

Beachte: In `Control.Monad` ist `sequence` allgemeiner definiert mit dem Typ

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

```
sequence_ :: (Monad m) => [m a] -> m ()
sequence_ []          = return ()
sequence_ (action:as) = do action
                           sequence_ as
```

wie `sequence`, aber Ergebnisse werden verworfen.

Nützliche Monaden-Funktionen (4)

```
mapM            :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as       =  sequence (map f as)
```

`map`-Ersatz für das monadische Programmieren.

```
mapM_           :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as      =  sequence_ (map f as)
```

wie `mapM`, aber Ergebnis wird verworfen.
Beispiel:

```
*Main> mapM_ print [1..100]
1
2
3
4
5
...
```


Nützliche Monaden-Funktionen (5)

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

Ersatz für for-Schleifen.

Beispiel:

```
import Control.Monad
main = do namen <- forM [1,2,3,4] (\i ->
                       do
                          putStrLn $ "Gib den " ++ show i ++ ". Namen ein!"
                          getLine
                        )
          putStrLn "Die Namen sind"
          forM_ [1,2,3,4] (\i -> putStrLn $ show i ++ "." ++ (namen!!(i-1)))
```

Nützliche Monaden-Funktionen (6)

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

Verallgemeinerung der `filter`-Funktion.

```
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

Liften der `filter`-Funktion in die Monade

Beispiel: Berechnung der Potenzmenge, also aller Teilmengen einer Menge:

```
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],[]]
```


Nützliche Monaden-Funktionen (7)

```
foldM :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m b
```

analog zu `fold`

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

analog zu `zipWith`

```
msum :: (Foldable t, MonadPlus m) => t (m a) -> m a
msum = foldr mplus mzero
```

Verallgemeinerung von `concat`

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

Die Funktion `join` wickelt verschachtelte Monaden aus.

Zustandsmonade

Zustandsbasiertes Programmieren mit Monaden

Die Zustandsmonde

Zustandsbasiertes Programmieren:

- Verwende Monade zur Kapselung des Zustands
- Der wesentliche Trick:
  Datentyp kapselt die Effekte, nicht den Zustand
- Daher: Datentyp ist ein Zustandsveränderer
- Manche Literatur spricht von `StateTransformer`, wir verwenden `State`

Zustandsveränderer:

- Funktion von Zustand nach (Ergebnis,Zustand)
- In Haskell:

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

  ist polymorph über Zustand `s` und Rückgabe `a`

Running Example: Taschenrechner

- Einfacher Taschenrechner
- beherrscht nur die Grundrechenarten
- Wertet bei jeder Operation direkt aus
- Kann keine Punkt-vor-Strichrechnung
- Zustand: (noch anzuwendende Funktion, Wert)
- Alles Double Werte

Typsynonyme:

```
type InternalCalcState = (Double -> Double, Double)
type CalcState a = State InternalCalcState a
```

Zur Erinnerung:

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

D.h. `CalcState a = State (InternalCalcState -> (a,InternalCalcState)`

Running Example: Taschenrechner (2)

Vorgehen beim Entwurf

- Primitive Zustandsveränderer für Operationen wie Zifferneingabe, Addition usw.
- Ablauf des Tachenrechners: Sequenz der primitiven Operationen
- Sequentielle Verkettung wird durch Monade und >>= usw. implementiert

Generischer:

- Zunächst primitive Operationen allgemein für `State s a`
- Primitive Operationen für den Taschenrechner können diese dann verwenden.

Primitive Operationen für State s a

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

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

- `put` schreibt inneren Zustand
- `get` liest den inneren Zustand aus.

Monaden-Instanz für State s

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

Die Instanzen für `Functor` und `Applicative`:

```
instance Functor (State s) where
  fmap f mx  = mx >>= return . f

instance Applicative (State s) where
  pure       = return
  mf <*> mx  = mf >>= (\f -> mx >>= return . f)
```

Funktionen modify und runState

Inneren Zustand verändern:

```
modify :: (s -> s) -> State s ()
modify f = do
            a <- get   -- aktuellen Zustand lesen
            put (f a)  -- veränderten Zustand schreiben
```

Beachte: Bei der Programmierung sieht man die Zustände nicht explizit!
(es sei denn man fordert sie explizit z.B. mit `get` an).

Zustandsveränderer wirklich laufen lassen:

```
runState :: State s a -> s -> (a,s)
runState (State x) i = x i
```

d.h. Zustandsveränderer auf ein initialen Zustand anwenden

Beispiel

- Innerer Zustand: Eine `Int`-Zahl
- `Int`-Zähler wird mehrfach erhöht:

```
run = runState prg 0
 where prg = do  modify (+1)
                 modify (+1)
                 modify (+1)
                 modify (+1)
                 get
```

- Ausführung:

```
*> run
(4,4)
```


Running Example: Taschenrechner (3)

Abarbeitung von 30+50= und innere Zustände:

Zustand	Resteingabe
`(\x -> x, 0.0)`	30+50=

Zustand	Resteingabe
`(\x -> x, 0.0)`	30+50=
`(\x -> x, 3.0)`	0+50=

`(\x -> x, 0.0)`	30+50=
`(\x -> x, 3.0)`	0+50=
`(\x -> x, 30.0)`	+50=

Running-Example: Taschenrechner (4)

Startzustand:

```
start = (id, 0.0)
```

Nächste Aufgabe:

Implementiere `CalcState`-Aktionen passend zu den Funktionen des Taschenrechners

Wir können `get`, `put`, `modify` verwenden

Running-Example: Taschenrechner (5)

- `CalcState`-Aktionen die nur den inneren Zustand ändern, geben keinen Wert zurück
- Modellierung: Nulltupel `()` als Rückgabewert

Zustandsveränderung für beliebigen binären Operator:

```
oper :: (Double -> Double -> Double) -> CalcState ()
oper op = modify (\ (fn,num) -> (op (fn num), 0))
```

Z.B. `oper (+)` usw.

clear: Löschen der letzten Eingabe:

```
clear :: CalcState ()
clear =  modify (\ (fn,num) -> if num == 0 then (id,0.0) else (fn,0.0))
```


Running-Example: Taschenrechner (6)

total: Ergebnis berechnen:

```
total :: CalcState ()
total = do (fn,num) <- get
           put (id, fn num)
```

digit: Eine Ziffer verarbeiten:

```
digit :: Int -> CalcState ()
digit i = do (fn,num) <- get
             put (fn,num*10 + (fromIntegral i) ) -- Ziffern verschieben
                                                 -- und Ziffer hinzu
```

readResult: Ergebnis auslesen:

```
readResult:: CalcState Double
readResult = do (fn,num) <- get
                return (fn num)
```


Running-Example: Taschenrechner (7)

calcStep: Ein Zeichen der Eingabe verarbeiten:

```
calcStep :: Char -> CalcState ()
calcStep x
 | isDigit x = digit (fromIntegral $ digitToInt x)

calcStep '+' = oper (+)
calcStep '-' = oper (-)
calcStep '*' = oper (*)
calcStep '/' = oper (/)
calcStep '=' = total
calcStep 'c' = clear
calcStep _   = return () -- nichts machen
```


Running-Example: Taschenrechner (8)

calc: Hauptfunktion, monadische Abarbeitung:

```
calc xs = do
            mapM_ calcStep xs
            readResult
```

Ausführen des Taschenrechners:

```
runCalc :: CalcState Double -> (Double,InternalCalcState)
runCalc act = runState act start

mainCalc xs = fst $ runCalc (calc xs)
```

Running-Example: Taschenrechner (9)

Beispiele:

```
*Main> mainCalc "1+2*3"
9.0
*Main> mainCalc "1+2*3="
9.0
*Main> mainCalc "1+2*3c*3"
0.0
*Main> mainCalc "1+2*3c5"
15.0
*Main> mainCalc "1+2*3c5====="
15.0
```

Es fehlt noch:

> Sofortiges Ausdrucken nach Eingabe
> $\Rightarrow$ Interaktion und I/O (später)

State-Monade

- Typ `State` ist in `Control.Monad.Trans.State` vordefiniert
- Interna sind anders (sehen wir noch)
- Es gibt noch weitere Monaden:
  - `Reader r a` in der Bibliothek `Control.Monad.Trans.Reader`, wenn interner Zustand nur gelesen wird
  - `Writer w a` in `Control.Monad.Trans.Writer`, wenn interner Zustand nur geschrieben wird

Beispiele zu Reader

Primitive Operation: `ask :: Reader r`.

Beispiel

```
*> :m + Control.Monad.Trans.Reader
*> runReader (do {a <- ask; b <-ask; return (a,b)}) 10
(10,10)
```


Beispiele zu Reader (2)

Größeres Beispiel:

```
import Control.Monad.Trans.Reader

data Exp v = Var v | Val Int | Neg (Exp v) | Add (Exp v ) (Exp v )
type Env v = [(v,Int)]

fetch :: String -> Reader (Env String) Int
fetch x = do env  <- ask
             case lookup x env of
               Nothing ->  error "Variable not found"
               Just val -> return val
...
```

Beispiele zu Reader (3)

```
...
eval :: Exp String -> Reader (Env String) Int
eval  (Var x)  = fetch x
eval  (Val i)  = return i
eval  (Neg p)  = do  r <- eval p
                     return (negate r)
eval  (Add p q) = do e1 <- eval p
                     e2 <- eval q
                     return (e1+e2)
evaluate e env = runReader (eval e) env
```

Ein Beispielaufruf ist:

```
*Main> evaluate (Add (Val 1) (Var "x")) [("x",5)]
6
```


I/O in Haskell

Problematik

Echte Seiteneffekte sind in Haskell nicht erlaubt.

Warum?

Annahme: `getZahl::Int` wäre eine „Funktion“, die eine Zahl von der Standardeingabe liest

Referentielle Transparenz

Gleiche Funktion auf gleiche Argumente liefert stets das gleiche Resultat

Referentielle Transparenz ist verletzt, da `getZahl` je nach Ablauf unterschiedliche Werte liefert.

Problematik (2)

Gelten (bei Seiteneffekten) noch mathematische Gleichheiten wie

$$e + e = 2 * e \ ?$$

Nein: für $e =$ `getZahl` z.B.

```
getZahl*getZahl ---> 1*getZahl ---> 1*3 ---> 3
```

aber:

```
2*getZahl ---> gerade Zahl
```

Problematik (3)

Weiteres Problem: Betrachte die Auswertung von

```
length [getZahl,getZahl]
```

Wie oft wird nach einer Zahl gefragt?

```
      length (getZahl:(getZahl:[]))
--->  1 + length (getZahl:[])
--->  1 + (1 + (length []))
--->  1 + (1 + 0)
--->  1 + 1
--->  2
```

⇒ Keine Fragen gestellt!, da `length` die Auswertung der Listen-Elemente nicht braucht
⇒ Festlegung auf eine genaue Auswertungsreihenfolge nötig.
(Nachteil: verhindert Optimierungen + Parallelisierung)

Monadisches I/O

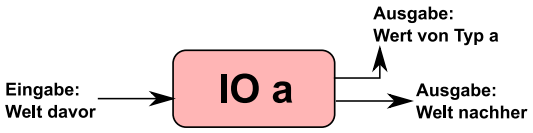
Kapselung des I/O

- Datentyp `IO a`
- Wert des Typs IO a ist eine I/O-Aktion, die beim Ausführen Ein-/Ausgabe durchführt und anschließend einen Wert vom Typ `a` liefert.
- `IO` ist Instanz von Monad
- Analog zu State, wobei der Zustand die gesamte Welt ist
- Programmiere in Haskell I/O-Aktionen, Ausführung quasi außerhalb von Haskell

Monadisches I/O (2)

Vorstellung:

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

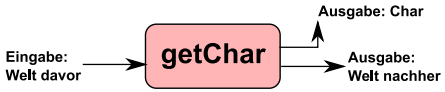


Monadisches I/O (3)

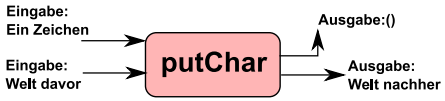
- Werte vom Typ `IO a` sind Werte, d.h. sie können nicht weiter ausgewertet werden
- Sie können allerdings ausgeführt werden (als Aktion)
- diese operieren auf einer Welt als Argument
- Ein Wert vom Typ `IO a` kann nicht zerlegt werden durch pattern match, da der Datenkonstruktor versteckt ist.

Primitive I/O-Operationen

```
getChar :: IO Char
```

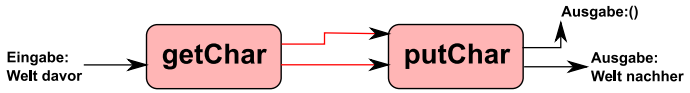



```
putChar :: Char -> IO ()
```



I/O Aktionen programmieren

- Man braucht Operationen, um I/O Operationen miteinander zu kombinieren!
- Z.B. erst ein Zeichen lesen (`getChar`), danach dieses Zeichen ausgeben (`putChar`)

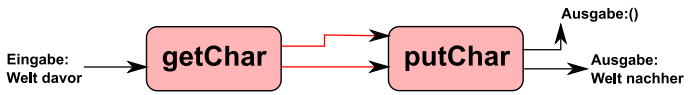


I/O Aktionen komponieren

Die gesuchte Verknüpfung bietet der >>= Operator

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

```
echo :: IO ()
echo = getChar >>= putChar
```



Alternativ mit do:

```
echo = do
         c <- getChar
         putChar c
```


I/O Aktionen komponieren (3)

Beispiel mit >>:

```
(>>) :: IO a -> IO b -> IO b
```

```
echoDup :: IO ()
echoDup = getChar >>= (\x -> putChar x >> putChar x)
```

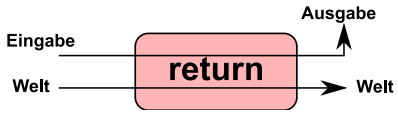
Alternativ mit do:

```
echoDup = do
            x <- getChar
            putChar x
            putChar x
```

I/O Aktionen komponieren (4)

I/O-Aktionen zusammenbauen, die zwei Zeichen liest und als Paar zurück liefert

```
getTwoChars :: IO (Char,Char)
getTwoChars = do
                x <- getChar
                y <- getChar
                return (x,y)
```



Beispiel

Eine Zeile einlesen

```
getLine :: IO [Char]
getLine = do c <- getChar;
             if c == '\n' then
               return []
             else
               do
                cs <- getLine
                return (c:cs)
```


Implementierung der IO-Monade

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

instance Monad IO where
 (IO m) >>= k = IO (\s -> case m s of
                  (a',s') -> case (k a') of
                        (IO k') -> k' s'
 return x = IO (\ s -> (x, s))
```

Interessanter Punkt:

Implementierung ist nur richtig bei call-by-need Auswertung

Implementierung der IO-Monade (2)

Beispiel von Simon Peyton Jones:

```
getChar >>= \c -> (putChar c >> putChar c)
```

wird übersetzt in (Vereinfachung: ohne IO-Konstruktor)

```
\w -> case getChar w of
       (c,w1) -> case putChar c w1 of
                   (_,w2) -> putChar c w2
```

Wenn beliebiges Kopieren korrekt wäre (a la call-by-name):

```
\w -> case getChar w of
       (c,w1) -> case putChar c w1 of
                   (_,w2) -> putChar (fst (getChar w)) w2
```

Nun 2 Probleme:

- `getChar w` wird zwei Mal aufgerufen
- Der Weltzustand `w` wurde verdoppelt

Deshalb:

- Implementierung nur korrekt, wenn nicht bel. kopiert wird.
- GHC extra getrimmt keine solche Transformation durchzuführen

Monadische Gesetze und IO

Oft liest man:

IO ist eine Monade.
D.h. die monadischen Gesetze sind erfüllt.

Aber:

```
(return True) >>= (\x -> undefined) ≠ (\x -> undefined) True
```

```
seq ((return True) >>= (\x -> undefined)) False ----> False
seq ((\x -> undefined) True) False              ----> undefined
```

Wenn man den Gleichheitstest nur auf Werte (ohne Kontext wie (seq [·] False)) beschränkt, dann gelten die Gesetze vermutlich.

Wenn man seq auf nicht-monadische Ausdrücke beschränkt, dann gelten die Gesetze.

GHC: eher pragmatische Sichtweise

Monadisches I/O: Anmerkungen

- Beachte: Es gibt keinen Weg aus der Monade heraus
- Aus I/O-Aktionen können nur I/O-Aktionen zusammengesetzt werden
- Keine Funktion vom Typ IO a -> a!
- Das ist nur die halbe Wahrheit!
- Wenn obiges gilt, funktioniert I/O sequentiell
- Aber: Man möchte auch "lazy I/O"
- Modell passt dann eigentlich nicht mehr

Beispiel: readFile

readFile: Liest den Dateiinhalt aus
– explizit mit Handles (= erweiterte Dateizeiger)

```
-- openFile :: FilePath -> IOMode -> IO Handle
-- hGetChar :: Handle -> IO Char

readFile :: FilePath -> IO String
readFile path =
 do
  handle <- openFile path ReadMode
  inhalt <- leseHandleAus handle
  return inhalt
```


Beispiel: readFile (2)

Handle auslesen: Erster Versuch

```
leseHandleAus handle =
 do
  ende <- hIsEOF handle
  if ende then hClose handle >> return []
   else
    do
     c  <- hGetChar handle
     cs <- leseHandleAus handle
     return (c:cs)
```

Ineffizient: Komplette Datei wird gelesen, bevor etwas zurück gegeben wird.

```
*Main> readFile "LargeFile" >>= print . head
'1'
7.09 secs, 263542820 bytes
```

unsafeInterleaveIO

```
unsafeInterleaveIO :: IO a -> IO a
```

- bricht strenge Sequentialisierung auf
- gibt sofort etwas zurück ohne die Aktion auszuführen
- Aktion wird "by-need" ausgeführt: erst wenn die Ausgabe vom Typ a in `IO a` benötigt wird.
- nicht vereinbar mit "Welt"-Modell!

Handle auslesen: verzögert

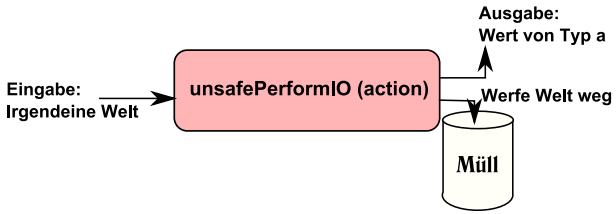
```
leseHandleAus handle =
 do
  ende <- hIsEOF handle
  if ende then hClose handle >> return []
   else
    do
     c  <- hGetChar handle
     cs <- unsafeInterleaveIO (leseHandleAus handle)
     return (c:cs)
```

Test:

```
*Main> readFile1 "LargeFile" >>= print . head}
'1'
(0.00 secs, 0 bytes)
```


UnsafePerformIO

- `unsafePerformIO :: IO a -> a`
- unsauberer Sprung aus der Monade



Nicht im Haskell-Standard enthalten

Implementierung von unsafeInterleaveIO

```
unsafeInterleaveIO :: IO a -> IO a
unsafeInterleaveIO a = return (unsafePerformIO a)
```

- Führt Aktion direkt mit neuer Welt aus
- Neue Welt wird verworfen
- Das Ganze wird mit `return` wieder in die IO-Monade verpackt

Veränderliche Speicherplätze

Nur IO-monadisch verwendbar;
Mit polymorphem Typ des Inhalts:

```
data IORef a -- Abstrakter Typ

newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

Imperativ (z.B. C): In Haskell mit IORefs

```
int x := 0          do
x := x+1              x <- newIORef 0
                      y <- readIORef x
                      writeIORef x (y+1)
```


Integration von IO in den Taschenrechner

Der bisherige Taschenrechner hat kein I/O durchgeführt.

Wünschenswert:

- Eingegebene Zeichen werden sofort verarbeitet
- Ausgabe erscheint sofort

```
main = do c <- getChar
          if c /= '\n' then do calcStep c
                               main
           else return ()
```

- Funktioniert nicht!
- `getChar` in `IO`-Monade
- `calcStep c` in `State`-Monade

Monad-Transformer

Monad-Transformer

- **Lösung:** Verknüpfe zwei Monaden zu einer neuen
- Genauer: Erweitere die State-Monade, so dass "Platz" für eine weitere Monade ist.
- Andere Sicht: Neue Monade = IO-Monade erweitert mit Zustandsmonade

```
-- alt:
newtype State state a =
  ST (state -> (a,state))

-- neu:
newtype StateT state monad a = StateT (state -> monad (a,state))
```

- Die gekapselte Funktion ist jetzt eine monadische Aktion.
- Monaden, die um eine Monade erweitert sind, nennt man Monad-Transformer

StateT

Monaden-Instanz für `StateT`:
(Applicative und Functor-Instanz lassen wir aus)

```
instance Monad m => Monad (StateT s m) where
    return x = StateT $ \s -> return (x, s)
    (StateT x) >>= f  = StateT $ \s ->  do
                                        (a,s') <- x s
                                        case (f a) of
                                           (StateT y) -> (y s')
```


put, get und runStateT für StateT

```
put :: Monad m => s -> StateT s m ()
put x = StateT $ \_ -> return ((),x)

get :: Monad m => StateT s m s
get = StateT $ \s -> return (s,s)

modify :: Monad m => (s -> s) -> StateT s m ()
modify f = do
           a <- get
           put (f a)

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


State und StateT in der Bibliothek

In `Control.Monad.Trans.State`:

```
type State s a = StateT s Identity a
```

D.h. State-Transformer, mit `Identity` als innere Monade!

Identity-Monade

Sie tut im Grunde nichts:

```
newtype Identity a = Identity a

instance Monad Identity where
 return x = Identity x
 (Identity m) >>= f  = (f m)

instance Applicative Identity where
 pure = return
 (Identity f) <*> (Identity x) = Identity (f x)

instance Functor Identity where
 fmap f (Identity x) = Identity (f x)
```

runState

Funktion `runState` kann dann mit den neuen Typen definiert werden als

```
runState :: State s a -> s -> (a,s)
runState f s = case runStateT f s of
                 Identity r -> r
```


Taschenrechner mit IO (1)

Typsynonym:

```
type CalcStateIO a = StateT InternalCalcState IO a
```

- Primitive Aktionen, die kein IO verwenden, können wie vorher programmiert werden
- Nur der Typ ändert sich: `CalcStateIO a` statt `CalcState a`

Aktionen der inneren Monade ausführen: Liften

Allgemeine Funktion zum „liften“ durch Typklasse `MonadTrans` (im Modul `Control.Monad.Trans.Class`) spezifiziert:

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

Für `StateT`:

```
instance MonadTrans (StateT s) where
  lift m = StateT $ \s -> do
                          a <- m
                          return (a,s)
```

- Aktion führt zunächst die Operation in der Monade `m` aus
- verpackt dann das erhaltene Ergebnis in die `StateT`-Monade.

Taschenrechner mit IO (2)

Neuimplementierung von `total` und `clear`:

```
total' :: CalcStateIO ()
total' =  do (fn,num) <- get
             lift $ putStr $ show (fn num)
             put (id,fn num)

clear' :: CalcStateIO ()
clear' = do (fn,num) <- get
            if num == 0.0 then do
               lift $ putStr ("\r"  ++ (replicate 100 ' ') ++ "\r")
               put start
             else do
              let l = length (show num)
              lift $ putStr $ (replicate l '\b')
                              ++ (replicate l ' ' )
                              ++ (replicate l '\b')
              put (fn,0.0)
```

Taschenrechner mit IO (3)

Im Grunde wie vorher:

```
calcStep' :: Char -> CalcStateIO ()
calcStep' x
 | isDigit x = digit (fromIntegral $ digitToInt x)

calcStep' '+' = oper (+)
calcStep' '-' = oper (-)
calcStep' '*' = oper (*)
calcStep' '/' = oper (/)
calcStep' '=' = total'
calcStep' 'c' = clear'
calcStep' _   = return ()
```


Taschenrechner mit IO (4)

Für die Hauptschleife:

```
calc' :: CalcStateIO ()
calc' = do
          c <- lift $ getChar
          if c /= '\n' then do
            calcStep' c
            calc'
           else return ()
```


Taschenrechner mit IO (5)

Hauptprogramm:

```
main = do
          hSetBuffering stdin NoBuffering  -- neu
          hSetBuffering stdout NoBuffering -- neu
          runStateT calc' start
```


Zusammenfassung Monaden-Transformer

- Monad-Transformer dienen dazu mehrere Monaden zu vereinen
- `lift`-Funktionen, um bestehende Funktionen in die neuer Monade zu übernehmen
- Beachte: Man kann auch eigene Monade erstellen, anstatt Monad-Transformer zu verwenden
- Ausnahme: IO, da man nicht an die Interna kommt

Anwendungen

Mehrere Monaden verketten

- Taschenrechner: Zwei Monaden vereint
- Das geht auch mit vielen Monaden
- Man erhält Stack von Monaden

Beispiel:

- Arithmetische Ausdrücke auswerten mit Umgebung (Reader-Monade)
- Dabei Loggen, welche lookups durchgeführt wurden (Writer-Monade)
- Dabei Statusmeldungen ausdrucken (IO-Monade)

Monaden-Stack: Beispiel

```
fetch :: String -> ReaderT (Env String) (WriterT [String] IO) Int
fetch x = do
          env  <- ask
          case lookup x env of
           Nothing ->  do
                       lift $ tell [("not found " ++ x)]
                       return 0
           Just val -> do lift $ tell [("found " ++ x)]
                       return val
```


Monaden-Stack: Beispiel (2)

```
eval :: Exp String -> ReaderT (Env String) (WriterT [String] IO) Int
eval  (Var x)  = fetch x
eval  (Val i)  = return i
eval  (Neg p)  = do
                 r <- eval p
                 lift $ lift $ putStrLn "computed a negation"
                 return (negate r)
eval  (Add p q) = do
                 e1 <- eval p
                 e2 <- eval q
                 lift $ lift $ putStrLn "computed a sum"
                 return (e1+e2)
```

Monaden-Stack: Beispiel (3)

Ein Aufruf ist

```
*Main> runWriterT (runReaderT  (eval (Add (Var "x") (Neg (Var "y"))))  [("x",1)])
computed a negation
computed a sum
(1,["found x","not found y"])
```


Monaden-Stack: Nachteil

- Anzahl an `lift`-Operationen unübersichtlich und schwierig
- Abhilfe 1: `liftIO :: MonadIO m => IO a -> m a` aus `Control.Monad.Trans`:
- Abhilfe 2 :`monad-tf`-Paket
  Verwendet Typ-Familien, kann die passende `ask` oder `tell`-Operation ermitteln, ohne explizite `lift`s

ST-Monade

Alternative zur Zustandsmonade: In `Control.Monad.ST` und `Data.STRef`

```
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 ()
```

- `forall` im Typ von `runST`: Trick, der verhindert, dass eine Referenz als Ergebnis zurückgeben wird und in einem anderen `runST` verwendet werden kann.
- Das Typsystem stellt sicher, dass jedes `runST` eigene, unabhängige Referenzen hat.

ST-Monade: Beispiel

```
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)
```

Ausführung ergibt: `"[1,21,203]"`

forall-Trick

Falsch (compiliert nicht):

```
fehlerdemo = let x = runST $ do x <- newSTRef 42
                                return x
             in      runST $ do y <- readSTRef x
                                return y
```

- Wegen `runST :: (forall s. ST s a)->a` kann der Typ der Referenz nicht in einem anderen `runST` verwendet werden.
- Sichergestellt, dass verschiedene Berechnungen mit Zustand voneinander unabhängig sein müssen.

Fehlermonade

In `Control.Monad.Trans.Except` wird die Fehlermonade `Except` (bzw. `ExceptT`) definiert.

- Werfen und Fangen von Fehlern

```
type Except e a = ExceptT e Identity a

runExceptT :: ExceptT e m a -> m (Either e a)

throwE :: Monad m => e -> ExceptT e m a
catchE :: Monad m => ExceptT e1 m a
                     -> (e1 -> ExceptT e2 m a)
                     -> ExceptT e2 m a
```

In `ExceptT e m a` ist `e` der Typ der Ausnahmen, `m` die innere Monade und `a` der Typ des funktionalen Ergebnis der Berechnung.

Fehlermonade (2)

Analog zu `Either e`-Monade:

```
data Either a b = Left a | Right a
instance Monad (Either a) where
      return x       = Right x
      Left  e >>= _  = Left e
      Right x >>= mf = mf x

instance Monad m => Monad (ExceptT e m) where
      return = ExceptT . return . Right
      mx >>= mf = ExceptT $ do
        x <- runExceptT mx
        case x of
          Left  e -> return $ Left e
          Right y -> runExceptT $ mf y
```


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

ex1= Prob[("Blue",1%2),("Red",1%4),("Green",1%4)]
```

Modelliert, dass das Ergebnis zu $\frac{1}{2}$ =50% Blau ist, zu $\frac{1}{4}$ = 25% Grün, usw.
Wie machen wir dies zur Monade?

Beispiel: Werte mit Wahrscheinlichkeiten (2)

Einfach:

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

Applicative-Instanz: generisch

```
instance Applicative Prob where
  pure = return
  mf <*> mx = mf >>= (mx >>=).(return.)
```


Beispiel: Werte mit Wahrscheinlichkeiten (3)

Idee für >>=:

- `m >>= f` ist immer gleich zu `join (fmap f m)`
- Wie implementiert man `join` direkt?
- Liste von Elementen mit Wahrscheinlichkeiten, wobei Elemente selbst solche Listen
- Wie man diese Liste glättet ist mathematisch klar: Multipliziere die die innere und äußere Wahrscheinlichkeit jeweils.

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


Beispiel

```
ex1= Prob[("Blue",1%2),("Red",1%4),("Green",1%4)]
ex2= Prob[(("Bright "++),1%5),(("Dark "++),2%5),(id,2%5)]

*> ex2 <*> ex1
  Prob[("Bright Blue",1%10), ("Bright Red",1%20)
      ,("Bright Green",1%20), ("Dark Blue",1%5)
      ,("Dark Red",1%10), ("Dark Green",1%10)
      ,("Blue",1%5), ("Red",1%10), ("Green",1%10)]
```

D.h. wenn wir zufällig eine Farbe wählen mit den gegebenen Wahrscheinlichkeiten und dann zufällig zu 20% die Farbe aufhellen oder zu 40% abdunkeln, dann ist das Ergebnis z.B. mit 5% Wahrscheinlichkeit hell rot.

Zusammenfassung

- Monaden zur Komposition von Berechnungen
- Zustandsmonade zur Zustandsbasierten Programmierung
- IO in Haskell verwendet interne IO-Monade (Zustand = Welt, wird wegkompiliert)
- Verketten von Monaden mit Monad-Transformern
- Gut einen gewissen Satz an Monaden zu kennen