Praktische Informatik 3: Funktionale Programmierung

Vorlesung 11 / 16. Januar 2024

Monaden

Thomas Barkowsky

Wintersemester 2023/24

Universität
Bremen

Übersicht

- Datentypen und Funktionen
- Rekursion und Listen
- Funktionen höherer Ordnung
- Algebraische Datentypen
- Rekursive und zyklische Datenstrukturen
- Abstrakte Datentypen
- I/O, Aktionen und Zustände
- Testen und Qualitätssicherung
- Monaden
- Domänenspezifische Sprachen
- Funktionale Programmierung in der Praxis

heute in dieser Vorlesung...

- Noch einmal: *bind* - Operator und `do` - Notation
- Monaden mit Beispielen
- Zustandsabhängige Funktionen
- Generische Funktionen für Monaden
- Gesetzmäßigkeiten für Monaden

bind-Operator und `do`-Notation

Beispiel: Fehlerbehandlung

- Datentyp für Ausdrücke aus `Int` und Division und

```
data Expr = Val Int | Div Expr Expr
```

  - eine Evaluationsfunktion:

```
eval :: Expr -> Int
eval (Val n) = n
eval (Div x y) = div (eval x) (eval y)
```

  - Beispiel:

```
> eval (Div (Val 8) (Val 2))
4
> eval (Div (Val 8) (Val 0))
*** Exception: divide by zero
```

Beispiel: Fehlerbehandlung (2)

- Sichere Division mit `Maybe`:

```
safediv :: Int -> Int -> Maybe Int
safediv a b = if b == 0 then Nothing else Just (div a b)
```

- Damit `eval`:

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = case eval x of
                   Nothing -> Nothing
                   Just n  -> case eval y of
                                Nothing -> Nothing
                                Just m  -> safediv n m
```

```
> eval (Div (Val 8) (Val 0))
Nothing
```

Beispiel: Fehlerbehandlung (3)

- Jetzt funktioniert `eval`, aber Definition ist umständlich/redundant:

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = case eval x of
                   Nothing -> Nothing
                   Just n  -> case eval y of
                                Nothing -> Nothing
                                Just m  -> safediv n m
```

- Was passiert hier? (zweimal!)
  - ein `Maybe a` wird ausgewertet und an eine Funktion übergeben, die daraus ein `Maybe b` erzeugt:
  - `... :: Maybe a -> (a -> Maybe b) -> Maybe b`
  - Kennen wir das nicht irgendwoher???

Der *bind* – Operator (revisited)

- Vgl.: *bind* für `IO`:

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

- Hier (*bind* für `Maybe`):

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
mx >>= f = case mx of
             Nothing -> Nothing
             Just x -> f x
```

  - `>>=` bekommt ein Argument vom Typ `a`, das evtl. `Nothing` ist und eine Funktion vom Typ `a -> b`, deren Resultat evtl. `Nothing` ist
  - `Nothing` wird propagiert, sonst Funktionsanwendung

Der *bind* – Operator (revisited)

- *bind* für `Maybe`:

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
mx >>= f = case mx of
             Nothing -> Nothing
             Just x -> f x
```

- Damit `eval`:

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = eval x >>= \n -> eval y >>= \m -> safediv n m
```

  - `>>=` ist linksassoziativ!

Auswertungsbeispiel

```
eval (Div (Val 4) (Val 2))
===>
eval (Val 4) >>= \n -> eval (Val 2) >>= \m -> safediv n m
===>
   (Just 4)  >>= \n ->   (Just 2)   >>= \m -> safediv n m
===>
    (\n -> (Just 2) >>= \m -> safediv n m) 4
===>
    (\n ->   ((\m -> safediv n m) 2)     ) 4
```

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = eval x >>= \n -> eval y >>= \m -> safediv n m
```

bind-Operator und `do`-Notation (revisited)

- Typische Struktur mit *bind*-Operator:

```
m1 >>= \x1 ->
m2 >>= \x2 ->
...
mn >>= \xn ->
f x1 x2 ... xn
```

und

vereinfacht in `do`-Notation (*syntactic sugar*):

```
do x1 <- m1
   x2 <- m2
   ...
   xn <- mn
   f x1 x2 ... xn
```

- Auswertung der `m1` ... `mn`, dann Anwendung der Funktion `f`
- `>>=` ist linksassoziativ
- Für `Maybe`: automatische Propagation von `Nothing`

`eval` mit `do`-Notation

```
eval :: Expr -> Maybe Int
eval (Val n)   = Just n
eval (Div x y) = do n <- eval x
                    m <- eval y
                    safediv n m
```

- `do`-Notation wie bekannt aus I/O:
  - "Abseitsregel": Alignierung der Anweisungen
  - "`xi <-`" entfällt, wenn Ergebnis von `mi` nicht benötigt wird

- Jetzt aber: Monaden!

Monaden

Die Typkonstruktorklasse `Monad`

- `do`-Notation nicht nur für `IO` und `Maybe`
- Verallgemeinerung: Klasse `Monad`:

```
class Monad m where
(>>=)  :: m a -> (a -> m b) -> m b
(>>)   :: m a -> m b -> m b
return :: a -> m a
fail   :: String -> m a

m >> k =  m >>= \_ -> k
fail s =  error s
```

Instanzen der Klasse `Monad`

- In *standard prelude* sind Listen, `IO` und `Maybe` Instanzen von `Monad`
  - aber: `IO` nicht innerhalb von Haskell definiert, sondern fest eingebaut (wg. Systemabhängigkeit)
- Nicht vergessen: die Kontrollstrukuren `sequence`, `sequence_`, `mapM` und `mapM_` (siehe Vorlesung 9)

- `IO` und `Maybe` kennen wir jetzt schon als Monaden
  - wie sieht das mit Listen aus?

Listen als Instanz von `Monad`

```
instance Monad [] where
m >>= k  = concat (map k m)
return x = [x]
fail s   = []
```

(aus *standard prelude)*

- ... sieht trivial aus: Funktion `k` wird auf alle Elemente aus `m` angewandt, alle Ergebnisse in einer Liste zurückgegeben
- nützlich für Beschreibung von Ausdrücken, die mehrere verschiedene Ergebnisse liefern können ("Nichtdeterminismus")
- Beispiele...

Listen als Monaden

```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = do x <- xs
                 y <- ys
                 return (x,y)

sums :: [Int] -> [Int] -> [Int]
sums xs ys = do x <- xs
                y <- ys
                return (x + y)
```

```
> pairs [1,2] [3,4]
[(1,3),(1,4),(2,3),(2,4)]
```

```
> sums [1,2] [3,4]
[4,5,5,6]
```

- Wichtig zum Verständnis: klarmachen, was *bind* bei Listen tut!
- Vgl.: *List comprehension*! (*syntactic sugar* für *bind* über Listen)

Auswertung am Beispiel von `pairs`

```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = do x <- xs
                 y <- ys
                 return (x,y)
```

- Syntactic sugar der do-Notation auflösen:

```
pairs xs ys = xs >>= \x -> ys >>= \y -> return(x,y)
```

Auswertung am Beispiel von `pairs`

```
pairs [1,2] [3,4]
 [1,2]  >>=  \x ->       [3,4]  >>=  \y -> return(x,y)
concat (map (\x ->       [3,4]  >>=  \y -> return(x,y))     [1,2])
concat (map (\x -> (concat (map(\y -> return(x,y)) [3,4]))) [1,2])
concat (map (\x -> (concat  [return (x,3), return (x,4)])  [1,2])
concat (map (\x -> (concat [[(x,3)],[x,4]])                [1,2])
concat (map                (\x -> [(x,3),(x,4)])           [1,2])
concat [[(1,3),(1,4)], [(2,3),(2,4)]]
[(1,3),(1,4), (2,3),(2,4)]
```

```
pairs xs ys = xs >>= \x -> ys >>= \y -> return(x,y)
```

```
instance Monad [] where
m >>= k  = concat (map k m)
return x = [x]
```

Zustandsabhängige Funktionen

Monaden für Zustände

- Funktionen, die mit veränderlichen Zuständen umgehen
- Vereinfachende Annahme (o.B.d.A.): Zustände als `Int` Werte:

```
type State = Int
```

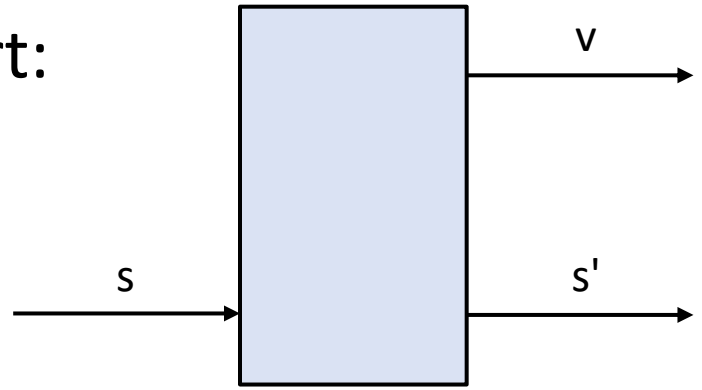
- Funktion zur Veränderung eines Zustands (*state transformer*):

```
type ST = State -> State
```

- besser: *state transformer* mit Rückgabewert:

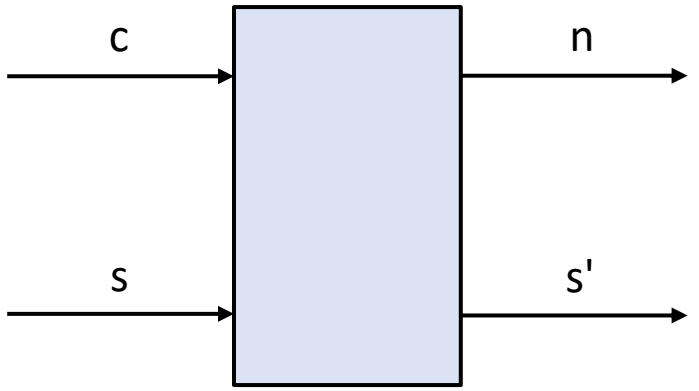
```
type ST a = State -> (a, State)
```

  - z.B. durchlaufende Buchungsnummer und Kontostand
  - Wie kommt der Rückgabewert zustande? ...



Monaden für Zustände (2)

- noch besser: *state transformer* zusätzlich mit Eingabewert:
  - z.B. Eingabe `Char`, Rückgabe `Int`
  - bereits durch *Currying* möglich:
    `Char -> State -> (Int, State)`
    realisiert durch `Char -> ST Int`



- Bei `ST` handelt es sich um eine explizite Repräsentation von Seiteneffekten (Zustandsänderungen): Monade!
- Um `ST` als Instanz einer Typklasse definieren zu können, müssen wir `ST` nicht durch `type`, sondern `data` oder `newtype` definieren,
- daher...

Monaden für Zustände (3)

- `ST` mit `newtype`, Dummy-Konstruktor `S` :

```
newtype ST a = S (State -> (a, State))
```

- nützlich: Hilfsfunktion, zur Entfernung des Dummy-Konstruktors:

```
app :: ST a -> State -> (a, State)
app (S st) = st
```

  - `app` bekommt einen *state transformer* und liefert eine Funktion `State -> (a, State)` zurück.

- `ST` als Monade:

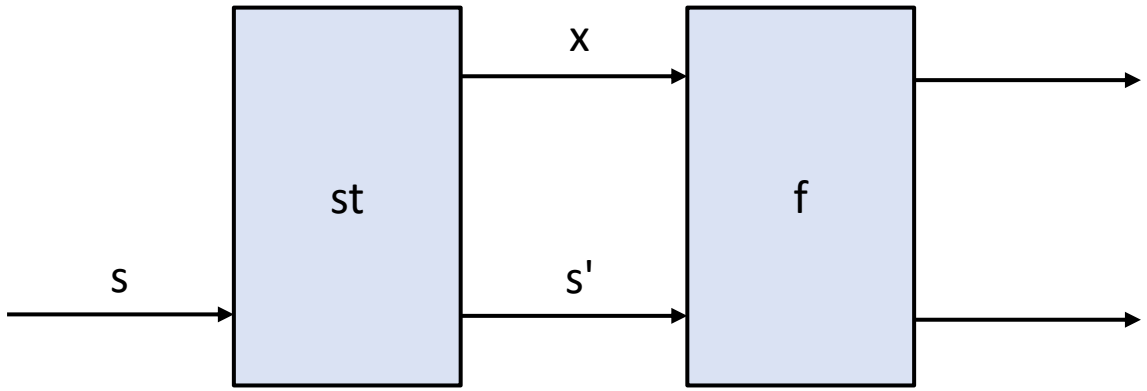
```
instance Monad ST where
   st >>= f =
      S (\s -> let (x,s') = app st s in app (f x) s')
```

Monaden für Zustände (4)

- `ST` als Monade:

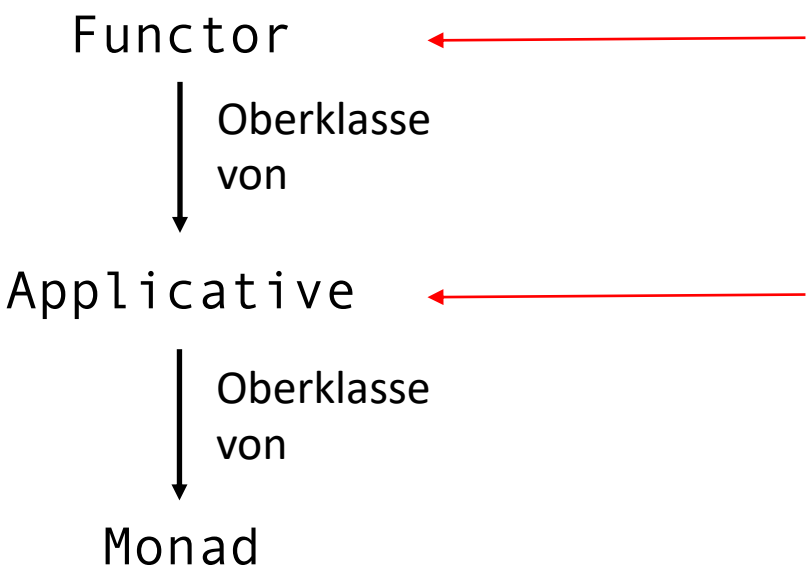
```
instance Monad ST where
   st >>= f = S (\s -> let (x,s') = app st s in app (f x) s')
```

  - `st >>= f` wendet *state transformer* `st` auf einen Anfangszustand `s` an, wendet `f` auf das Ergebnis `x` an und erzeugt so einen neuen *state transformer*:
  - damit verbindet der *bind*-Operator für die `ST` Monade die Sequenzierung von *state transformers* mit der Weiterverarbeitung der Ergebniswerte



Was noch fehlt…

- Typklassen-Hierarchie seit GHC 7.1 (2015):
  - nicht in *Haskell 2010 Language Report*



bekannt aus Vorlesung 7 (`fmap`)

zwei Funktionen:

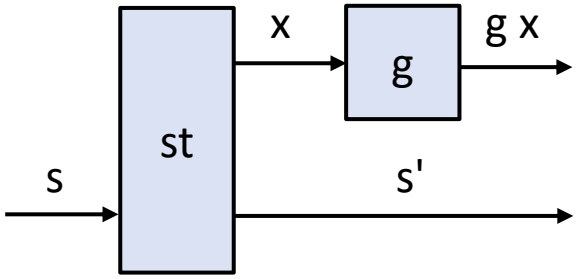
```
class Functor f => Applicative f where
   pure  :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

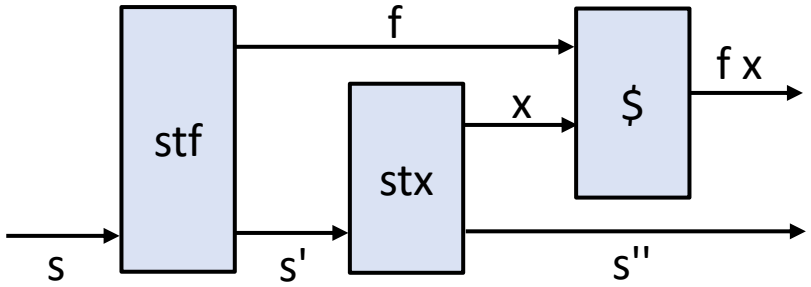
Was noch fehlt... (2)

- Fehlende Definitionen:

```
instance Functor ST where
   fmap g st = S (\s ->
      let (x, s') = app st s in (g x, s'))

instance Applicative ST where
   pure x = S (\s -> (x,s))
   stf <*> stx = S (\s ->
      let (f,s') = app stf s
          (x,s'')= app stx s' in (f x, s''))
```





Beispiel: Blätter in Bäumen nummerieren

- ... mit zustandsabhängiger Funktion
  - gegeben:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

  - Beispielbaum:

```
tree = Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')
```

  - Aufgabe: Umbenennen aller Blätter mit je einer neuen Zahl
  - konventionelle Lösung mit mitgeführtem `Int`-Argument:

```
rlabel :: Tree a -> Int -> (Tree Int, Int)
rlabel (Leaf _) n   = (Leaf n, n+1)
rlabel (Node l r) n = (Node l' r', n'') where
                                  (l', n')  = rlabel l n
                                  (r', n'') = rlabel r n'
```

Beispiel: Blätter in Bäumen nummerieren (2)

```
tree = Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')
```

```
> rlabel tree 5
(Node (Node (Leaf 5) (Leaf 6)) (Leaf 7),8)
```

- Lösung geht, ist aber nicht schön wg. zusätzlichem Parameter
- besser mit *state transformer*: `Tree a -> ST (Tree Int)` mit nächstem `Int`-Label als Zustand
- *state transformer*, der aktuellen Zustand und Folgezustand zurückgibt:

```
fresh :: ST Int
fresh = S (\n -> (n, n+1))
```

Beispiel: Blätter in Bäumen nummerieren (3)

- Monadische Lösung:

```
mlabel :: Tree a -> ST (Tree Int)
mlabel (Leaf _) = do n <- fresh
                     return (Leaf n)
mlabel (Node l r) = do l' <- mlabel l
                       r' <- mlabel r
                       return (Node l' r')
```

```
tree = Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')
```

```
> app (mlabel tree) 5
(Node (Node (Leaf 5) (Leaf 6)) (Leaf 7),8)
```

Generische Funktionen für Monaden

Vorteil der Verwendung von Monaden

- Generische Funktionen für **alle** Monaden
- vgl. `fmap` für Funktoren
- siehe `Control.Monad`

- Beispiel: `mapM` ...

Beispiel: `mapM`

- Monadische Version von `map` (*standard prelude*):

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []     = return []
mapM f (x:xs) = do y  <- f x
                   ys <- mapM f xs
                   return (y:ys)
```

  - übergebene Funktion und `mapM` selbst geben monadische Typen zurück
- Beispiel für Verwendung von `mapM`: ...

Beispiel: `mapM` (2)

- Konvertierung von Zahlzeichen (`Char`) in Zahlen, wenn möglich:

```
conv :: Char -> Maybe Int
conv c | isDigit c = Just (digitToInt c)
       | otherwise = Nothing
```

  - `isDigit`, `digitToInt` aus `Data.Char`

- Anwendung von `mapM` mit `conv`:
  - `mapM conv` nur erfolgreich, wenn jedes Zeichen im String ein Zahlzeichen ist

```
> mapM conv "123"
Just [1,2,3]
> mapM conv "i23"
Nothing
```

noch ein Beispiel: `filterM`

- monadische Version von `filter` (`Control.Monad`)

```
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []     = return []
filterM p (x:xs) = do b  <- p x
                      ys <- filterM p xs
                      return (if b then x:ys else ys)
```

- Beispiel: Potenzmenge (alle Möglichkeiten, Elemente einer Menge ein- oder auszuschließen):

```
> filterM (\x -> [True, False]) [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

Gesetzmäßigkeiten für Monaden

Gesetzmäßigkeiten für Monaden

- Folgende Gesetzmäßigkeiten für Monaden müssen gelten:
  - vgl. Funktoren (Vorlesung 7): Identität und Komposition
  - `return` eines Wertes und Verknüpfung mit einer monadischen Funktion ist identisch mit der Anwendung der Funktion auf den Wert:
    `return x >>= f  = f x`
  - Ergebnis einer monadischen Operation verknüpft mit `return` ist identisch mit diesem selbst:
    `m >>= return  = m`
  - Assoziativität von >>=:
    `(m >>= f) >>= g  = m >>= (\x -> (f x >>= g))`

Zusammenfassung

- Monaden als Muster für Berechnungen mit Seiteneffekten
  - I/O
  - Fehlerbehandlung
  - Zustände
  - Nichtdeterminismus (Listen)

- `>>=` (bzw. `do` - Notation) und `return`

- Verallgemeinerte Funktionen für Monaden

nächstes Mal...

- Anwendung von Monaden: Parsing
- Domänenspezifische Sprachen mit Haskell
- Praktisches Anwendungsbeispiel