Monaden für alle

Franz Pletz
<fpletz@muc.ccc.org>

Chaos Computer Club München

13. Juni 2010, GPN10

Wieso, weshalb, warum?

- einige von euch haben sich sicher schon mal Haskell angeschaut und sind an Monaden gefailed (inkl. mir ;-))
- man versteht nicht sofort den Sinn
- Monaden sind aber ein wichtiger Aspekt warum Haskell pur funktional ist und sein kann
- Beispiel IO: Ein-/Ausgabe wird in einen Container bzw. State gewrappt und jeder Operation/Funktion übergeben, die ihn manipuliert und weiterreicht
- damit kann man auch in einer lazy Sprache wie Haskell eine Order of Evaluation erzwingen

Wie kann man Monaden verstehen?

- dafür habe ich viele Tutorials und Bücher gelesen damit ichs selbst verstehe
- aus einigen Quellen habe ich Ideen zusammengetragen
- Ziel: jeder, der etwas programmieren und logisch denken kann soll es ohne viel Mathematik verstehen
- herausgekommen ist dieser Vortrag ;-)

Fahrplan

map

- `map` wendet eine Funktion auf Elemente einer Liste an und liefert eine Liste mit den Ergebnissen zurück
- formal (rekursiv):

```
map :: (a -> b) -> [a] => [b]
map _ []     = []
map f (x:xs) = (f x) : (map f xs)
```

- oder mit List Comprehension:

```
map f xs = [f x | x <- xs]
```

Funktoren

- Typklasse für Container-Typen, die mappable sind, also ein `map` Sinn macht
- Beispiele: Bäume, Maybe, Listen
- Funktoren müssen `fmap` implementieren

```
fmap :: (a -> b) -> f a -> f b
```

Funktor-Instanzen, `fmap`

- Beispiel `Maybe`

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

- Beispiel `IO`

```
instance Functor IO where
    fmap f action = do
        result <- action
        return (f result)
```

Benutzen von Funktoren

```
-- get value for key in list of tuples
lookup :: (Eq a) => a -> [(a, b)] -> Maybe b

-- lookup in a dictionary
let l = [(1, "fnord"), (2, "foo"), (3, "bar")] in
    fmap reverse $ flip lookup l 2
-- = Just "oof"
```

Benutzen von Funktoren (2)

```
-- without fmap
main = do line <- getLine
          let line' = reverse line
          putStrLn line'

-- with fmap
main = do line <- fmap reverse getLine
          putStrLn line
```

Funktionskomposition als Funktor

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
    -- or, easier
    fmap = (.)

-- inferred type of fmap
fmap :: (a -> b) -> f a -> f b
fmap :: (a -> b) -> ((->) r a) -> ((->) r b)
fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

Funktionskomposition als Funktor (2)

```
f :: (Num a) => a -> a
f = fmap (*3) (+100)

-- example
f 1 == 303

-- alternative definitions
f = (*3) `fmap` (+100)
f = (*3) . (+100)
```

Gesetze von Funktoren

1. `fmap id = id`
   - Identität
   - "Wenn wir mit der `id` Funktion über einen Funktor mappen, erhalten wir denselben Funktor wieder"
2. `fmap (f . g) = fmap f . fmap g`
   - Distributivität bzgl. Funktionskomposition
   - "Wenn wir zwei Funktionen kompositieren und diese Funktion über einen Funktor mappen, erhalten wir das selbe wie wenn wir erst die eine Funktion und dann die andere über den Funktor mappen"

Fahrplan

Definition

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

- `pure` legt einen Wert in den Container-Typ
- der `<*>` Operator verlangt eine Funktion und einen Wert in einem Container-Typ und übergibt den Wert als Argument der Funktion und liefert das Ergebnis wieder im Container-Typ zurück

Maybe als Applicative

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

-- examples
pure (+3) <*> Just 9     -- Just 12
pure (+3) <*> Nothing    -- Nothing
Nothing <*> Just "woot" -- Nothing
pure (++"!") <*> Just "hello"   -- Just "hello!"
pure (+) <*> Just 3 <*> Just 9  -- Just 12
```

Applicative Style

```
-- applicative style
pure f <*> x <*> y <*> ...

-- trivial
pure f <*> x == fmap f x

-- defined in Control.Applicative
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x

-- application of f with arguments x, y, z
f <$> x <*> y <*> z
-- if x, y, z weren't in containers
f x y z
```

Listen als Applicative

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

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

[(*0),(+100),(^2)] <*> [1,2,3]
-- [0,0,0,101,102,103,1,4,9]

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

IO als Applicative

```
instance Applicative IO where
    pure = return
    a <*> b = do
        f <- a
        x <- b
        return (f x)

-- example
myAction = do
    a <- getLine
    b <- getLine
    return $ a ++ b

-- applicative style
myAction = (++) <$> getLine <*> getLine
```

Funktionen als Applicative

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

-- examples
pure 3 "blah"   -- 3
(+) <$> (+3) <*> (*100) $ 5      -- 508
(\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
-- [8.0,10.0,2.5]
```

Applicative Lifts

```
-- from Control.Applicative
liftA2 :: (Applicative f) => (a -> b -> c) ->
                        f a -> f b -> f c
liftA2 f a b = f <$> a <*> b

-- examples
liftA2 (:) (Just 3) (Just [4])   -- Just [3,4]
(:) <$> Just 3 <*> Just [4]      -- Just [3,4]
```

Spass mit Lifting

```
-- takes list of Applicatives and creates an
-- Applicative with a list of the values
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs

-- shorter ;)
sequenceA = foldr (liftA2 (:)) (pure [])

-- example
sequenceA [Just 1, Just 2]        -- Just [1,2]
```

Applicative Functor Gesetze

1. alle Gesetze von Funktoren
2. `pure f <*> x = fmap f x`
3. `pure id <*> v = v`
4. `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
5. `pure f <*> pure x = pure (f x)`
6. `u <*> pure y = pure ($ y) <*> u`

Fahrplan

Multiplikations- und Listenkonkatenationsoperatoren

- sowohl `*` als auch `++`
  - nehmen 2 Parameter
  - Parameter und Rückgabewerte haben den gleichen Typ
  - haben einen Wert für welchen sich der andere Parameter nicht ändert

```
4 * 1                   -- 4
1 * 9                   -- 9
[1,2,3] ++ []           -- [1,2,3]
[] ++ [0.5, 2.5]        -- [0.5,2.5]
```

  - die Reihenfolge der Auswertung ist egal (Assoziativität)

```
(3 * 2) * (8 * 5)           -- 240
3 * (2 * (8 * 5))           -- 240
"la" ++ ("di" ++ "da")      -- "ladida"
("la" ++ "di") ++ "da"      -- "ladida"
```

Fahrplan

Meet the Monoids!

- Definition:

```
-- defined in Data.Monoid
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

- ein Monoid ist wenn man für einen Typ eine assoziative binäre Funktion (Operator) und einen Wert, der als neutrales Element für diese Funktion agiert

Monoid Gesetze

1. `mempty `mappend` x = x`
   - neutrales Element (links)
2. `x `mappend` mempty = x`
   - neutrales Element (rechts)
3. `(x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)`
   - Assoziativität

Listen als Monoide

```
instance Monoid [a] where
    mempty = []
    mappend = (++)

-- examples
[1,2,3] `mappend` [4,5,6]    -- [1,2,3,4,5,6]
mconcat [[1,2],[3,6],[9]]    -- [1,2,3,6,9]
```

Produkt als Monoid

```
-- helper declaration
newtype Product a =  Product { getProduct :: a }
    deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Product a) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)

-- examples
getProduct $ Product 3 `mappend` Product 9
-- 27
getProduct $ Product 3 `mappend` mempty
-- 3
getProduct . mconcat . map Product $ [3,4,2]
-- 24
```

`Any` und `All` als Monoide

```
instance Monoid Any where
        mempty = Any False
        Any x `mappend` Any y = Any (x || y)

instance Monoid All where
        mempty = All True
        All x `mappend` All y = All (x && y)

-- examples
getAll . mconcat . map All $ [True, True, False]
-- False
getAny . mconcat . map Any $ [False, False, True]
-- True
```

Maybe als Monoid

```
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    Nothing `mappend` m = m
    m `mappend` Nothing = m
    Just m1 `mappend` Just m2 =
                Just (m1 `mappend` m2)

-- examples
Nothing `mappend` Just "foo"      -- Just "foo"
Just "bar" `mappend` Nothing      -- Just "bar"
Just "foo" `mappend` Just "bar"   -- Just "foobar"
```

Fahrplan

Monade als Applikativer Funktor

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

-- defined in Control.Monad
class Monad m where
    return :: a -> M a
    (>>=)  :: M a -> (a -> M b) -> M b

    (>>)   :: m a -> m b -> m b
    m >> n = m >>= \_ -> n
    fail   :: String -> m a
```

Monade als Applikativer Funktor (2)

- man kann aus jeder Instanz von `Monad` eine Instanz von `Applicative` machen:

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

ap f a = do
    f' <- f
    a' <- a
    return (f' a')
```

Fahrplan

Gesetze von Monaden

1. `m >>= return = m`
   - Rechtseinheit
2. `return x >>= f = f x`
   - Linkseinheit
3. `(m >>= f) >>= g = m >>= (\ x -> f x >>= g)`
   - Assoziativität

do-Blöcke

```
do { x }                          --> x
do { let { y = v }; x }           --> let y = v in do { x }
do { v <- y; x }                  --> y >>= \v -> do { x }
do { y; x }                       --> y >>= \_ -> do { x }

-- example: read line from stdin, reverse, print
main = do
    x <- getLine
    x' <- reverse x
    putStrLn x'
-- points-free style
main = getLine >>= (\x' -> (\x -> putStrLn x)
                                  (reverse x'))
```

Maybe als Monade

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

-- examples
(//) :: Maybe a -> Maybe a -> Maybe a
    _ // Nothing = Nothing
    Nothing // _  = Nothing
    _ // Just 0 = Nothing
    Just x // Just y = Just (x / y)
```

Maybe als Monade (2)

```
-- examples (cont.)
test1 x y = let
    one <- return 1
    jx <- return x
    jy <- return y
    in one // (jx // jy)

test2 = do
    x <- Just 1 // Just 2       -- Just 0.5
    y <- Just 1 // Just 0       -- Nothing
    return $ x // y             -- Nothing
```

Fahrplan

Monaden und Funktoren

```
fmap   :: (a -> b) -> M a -> M b     -- functor

return :: a -> M a
join   :: M (M a) -> M a

-- with this, definition of bind
m >>= g = join (fmap g m)

-- or, definition of fmap and join with bind
fmap f x = x >>= (return . f)
join x   = x >>= id
```

Fahrplan

Assoziativität von »=

- die Assoziativität sorgt dafür, dass »= sich nur um die Reihenfolge der Berechnungen kümmert, jedoch nicht um ihre Verschachtelung
- Definition des Äquivalents der Funktionskomposition für Monaden:

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

- analog zur Assoziativität von Monoiden (wenn man vom Lambda-Ausdruck absieht)!

Fahrplan

Was gibts noch?

- `MonadPlus`, was mehrere Berechnungen repräsentiert und äquivalent zu Monoiden ist

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

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

instance MonadPlus Maybe where
    mzero                     = Nothing
    Nothing `mplus` Nothing = Nothing
    Just x  `mplus` Nothing = Just x
    Nothing `mplus` Just x  = Just x
    Just x  `mplus` Just y  = Just x
```

Was haben wir heute gelernt?

- einige sehr erweiterte funktionale Techniken
- Funktore, Applikative Funktoren und Monoide sind mit Monaden sehr nah verwandt und können teilweise untereinander umgewandelt werden
- mit diesem Wissen könnt ihr weiter an Monaden forschen damit ihr sie und weitere Zusammenhänge zu anderen Typen finden

Vielen Dank fürs Wachbleiben! ;)