Kategorien und funktionales Programmieren

Johannes Waldmann
Institut für Informatik
Universität Leipzig

Kategorien

mathematische Objekte,
mit Struktur

Beziehungen zwischen Objekten,
die Struktur erhalten

Beispiele:

Gruppen mit Homomorphismen,
topologische Räume mit stetigen Abbildungen,
Halbordungen mit monotonen Funktionen

Definition:

Kategorie C
hat Objekte ObC
und Pfeile (Morphismen) Mor$C$.

Jeder Pfeil $f \in$ Mor$C$
hat Quelle (Domain) dom$f \in$ ObC
und Ziel (Co-Domain) cod$f \in$ ObC.

Es gibt assoziative Operation
$\circ$: Mor $\times$ Mor $\rightarrow$ Mor
wenn $f : A \rightarrow B, g : B \rightarrow C$
dann $(g \circ f) : A \rightarrow C$.

Zu jedem $A \in$ ObC
gibt es genau ein id$_A : A \rightarrow A$
mit $f \circ$ id $= f =$ id $\circ f$

Rechnen in Kategorien

innere Struktur der Objekte nicht zugänglich.

Struktur erscheint nur
durch Vernüpfen mit anderen Objekten.

Beispiel: Produkt

für Mengen: $A \times B = \{(a,b) | a \in A, b \in B\}$,
wobei $(a,b) := \{a, \{a,b\}\}$.

allgemein: $(\times, \mathsf{prj}_1, \mathsf{prj}_2)$:

$A \times B$ ist C mit:

$$\forall f : D \to A, g : D \to B\ \exists h : D \to C :$$
$$f = \mathsf{prj}_1 \circ h, g = \mathsf{prj}_2 \circ h$$

Dualität

alle Pfeile rumdrehen, Aussagen bleiben gültig

Beispiel:

Produkt: $(\times, \mathsf{prj}_1, \mathsf{prj}_2)$:

$A \times B$ ist C mit:

$$\forall f : D \to A, g : D \to B\ \exists h : D \to C :\\ f = \mathsf{prj}_1 \circ h, g = \mathsf{prj}_2 \circ h$$

Summe: $(+, \mathsf{inj}_1, \mathsf{inj}_2)$:

$A + B$ ist C mit:

$$\forall f : A \to D, g : B \to D\ \exists h : C \to D :\\ f = h \circ \mathsf{inj}_1, g = h \circ \mathsf{inj}_2$$

Kategorien in der Informatik

1. Semantik für Lambda-Kalkül

Pfeil $A \to B$ muß selbst Objekt sein
(cartesisch abgeschlossene Kategorien),
Fixpunkte von Pfeilen,
stetige Funktionen auf $\omega-$CPOs

2. Daten und Funktionen darauf

Rechenregeln für Programme
(Korrektheit, Optimieren)

Funktoren, natürliche Transformationen,
Haskell: Typ- und Konstruktor-Klassen,
3. funktionale Verpackung des Imperativen

Berechnungen mit Resultat und Wirkung,
Monaden, Kleisli-Kategorien,
Haskell: Klassen, I/O

Datentypen und Funktoren

Objekte: Datentypen,
Pfeile: Funktionen,

polymorphe Typen `List a`, `Tree a`,
polymorphe Funktionen

```
map :: (a -> b) -> (List a -> List b)
```

Funktoren

Kategorie mit allen Kategorien als Objekten.
Was sind die Pfeile? Funktoren.

Definition:
F ist Funktor von Kategorie C nach Kat. D:
$F_{\mathsf{Ob}} : \mathsf{Ob}C \to \mathsf{Ob}D$, $F_{\mathsf{Mor}} : \mathsf{Mor}C \to \mathsf{Mor}D$:
$F\,\mathsf{id} = \mathsf{id}, \;\; Fg \circ Fh = F(g \circ h)$

`List` und `Tree` sind (Endo-)Funktoren.

Konstruktorklassen

```
data List a = Nil | Cons a (List a)
mapList :: (a -> b) -> (List a -> List b)
mapList Nil = Nil
mapList (Cons x xs) = Cons (f x) (mapList f xs)

data Tree a = Leaf | Node a (Tree a) (Tree a)
mapTree :: (a -> b) -> (Tree a -> Tree b)
mapTree f Leaf = Leaf
mapTree f (Node x l r) =
    Node (f x) (mapTree f l) (mapTree f r)
```

```
class Functor f where
    map :: (a -> b) -> (f a -> f b)
instance Functor List where map = mapList
instance Functor Tree where map = mapTree
```

Typen haben Sorte `*`,
Konstruktoren haben Sorte `* -> *`

Typklassen

```
class Show a where show :: a -> String
instance Show Ratio
   where show (x :% y) = ...
```

Implementieren:

jede Klasse bekommt Wörterbuch-Typ,
jede Instanz Wörterbuch

```
f :: Show a => a -> ...
f x y ... = ... show x ...

f' :: ShowDict -> a -> ...
f' dict x y ... = ... dict.show x ...

class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)       -- Default
class Eq a => Ord a where       -- Vererbung
    (<=) :: a -> a -> Bool
instance Eq a => Eq (List a)    -- Vererbung
```

Katamorphismen

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

cataTree :: (TreeF b a -> a) -> (Tree b -> a)
cataTree f Leaf = f LeafF
cataTree f (Node x l r) =
     f (NodeF (f x) (cataTree f l) (cataTree f r))

size = cataTree (\ LeafF -> 1, \ NodeF x l r -> l + r)

Tree a = fix (TreeF a)
    where   TreeF a t = LeafF | NodeF a  t  t
                      =   1   +      (a, t, t)

out :: fix f -> f (fix f) -- Isomorphie

instance Functor (TreeF a) where
instance Functor (TreeF a) where
    map f LeafF = LeafF
    map f (NodeF x l r) = NodeF x (map f l) (map f r)

cata :: Functor f => (f a -> a) -> (fix f -> a)
cata = f . map (cata f) . out

flatten :: Tree a -> [a]
flatten = cata flattenF  where
  flattenF :: TreeF a [a] -> [a]
  flattenF LeafF         = []
  flattenF (NodeF x l r) = l ++ [x] ++ r
```

Dual: Anamorphismen

```
anaTree :: (a -> TreeF b a a) -> (a -> Tree b)
anaTree f x = case f x of
    LeafF y -> Leaf y
    NodeF x l r -> Node x (anaTree f l) (anaTree f r)

in :: f (fix f) -> fix f -- Isomorphie

ana :: Functor f => (a -> f a) -> (a -> fix f)
ana = in . map (ana f) . f

build :: Ord a => [a] -> Tree a
build = ana buildF  where
  buildF :: [a] -> TreeF a [a] [a]
  buildF []       = LeafF
  buildF (x : xs) =
    NodeF x [ y | y <- xs, y <= x ]
            [ y | y <- xs, y > x  ]  )
```

Vorsicht: `build` ist nicht frei

FP with Bananas, Lenses, Envelopes, and Barbed Wire

(Meijer, Fokkinga, Paterson, FCPA 1991)
Squiggol (Bird, Meertens, 1983, 1989, 1992)

`cata phi` $= (\!|\phi|\!)$, `ana psi` $= [\!(\psi)\!]$

Identität: `cata in = id, ana out = id`

Fusion:

`h . f = g . map h` $\Rightarrow$ `h . cata f = cata g`
`f . h = map h . g` $\Rightarrow$ `ana f . h = ana g`

Acid Rain, Deforestation (Wadler, 1990):

```
hylo :: (Functor f, Functor g) =>
    (f a -> a) -> (b -> f b) -> (b -> a)
```

`hylo f g` $= [\![f, g]\!] = (\!|f|\!) \circ [\!(g)\!]$

`sort = flatten . build = hylo (flatF, buildF)`

Polymorphe Funktionen und natürliche Transformationen

Kategorie mit Funktoren von C nach D
Was sind Pfeile? Natürliche Transformationen.

Definition: F und G Funktoren von C nach D.
h ist n. T. von F nach G, wenn
für alle $A \in \mathsf{Ob}C : h_A : F_{\mathsf{Ob}}A \to G_{\mathsf{Ob}}A$
für jedes
$f : A \to B : \quad G_{\mathsf{Mor}}f \circ h_A = h_B \circ F_{\mathsf{Mor}}f$

Beispiel: `flatten :: Tree a -> List a`
(Inorder-Liste der Knotenmarkierungen)
ist n. T. von `Tree` nach `List`.

`reverse :: List a -> List a`
ist n. T. von `List` nach `List`.

Jede polymorphe Funktion ist n. T.

`map f . reverse = reverse . map f`

(Theorems for Free!, Wadler, FPLCA 1989; Freyd, Reynolds, 198*)

Spezialfall: Listen

```
instance Functor [] where
    map [] = []; map f (x : xs) = f x : map f xs

foldr             :: (a -> b -> b) -> b -> [a] -> b
foldr f z []       = z
foldr f z (x:xs)   = f x (foldr f z xs)
```

```
Identität: foldr (:) [] = id
```

```
Fusion:  h (f a b) = g a (h b)
  => h . foldr f b = foldr g (h b)
```

```
Acid rain: X, Y sind feste Typen
g :: (X -> b -> b) -> b -> (X -> b)
  =>   foldr f b . g (:) [] = g f b
```

```
length   = foldr (\ n _ -> n + 1) 0
xs ++ ys = foldr (:) ys xs
concat   = foldr (++) []

foldl             :: (a -> b -> a) -> a -> [b] -> a
foldl f z []       = z
foldl f z (x:xs)   = foldl f (f z x) xs

reverse  = foldl (flip (:)) []
```

Zusammenfassung Teil 1

funktionales Programmieren	Kategorien-Theorie
Typ	Objekt
Funktion	Pfeil
Typkonstruktor	Funktor
polymorphe Funktion	natürliche Transf.

Rechenregeln für Programme
Polymorphie $\rightarrow$ theorems for free

Implementierung: Konstruktorklassen

rekursive Typen als Fixpunkte von (Bi-)Funktoren

passende Funktionen sind Kata-, Ana-, Hylomorphismen

weitere Transformationsregeln (fusion, acid rain)

Berechnungen

Berechnung = Resultat und Wirkung

keine Wirkung und deterministisch: `Id r = r`

nichtdet: `List r`, `Tree r`

det und Zustand: `s -> (r, s)`

nichtdet und Zustand: `s -> [(r, s)]`

Berechnungen lassen sich verknüpfen
(nacheinander ausführen),
das ist assoziative Operation mit Identität,
also *Monoid* (Halbgruppe mit Eins).

In Kategorie der Datentypen: *Monade*
(Kleisli 1965; Moggi 1989; Wadler 199*)

Monaden

Funktor `m`: `r` $\rightarrow$ Berechnung mit Resultat `r`

```
class Functor m => Monad m where
    return ::       r   -> m r -- Einselement
    join   :: m (m r) -> m r -- Verknüpfung
```

Definition: `(m, return, join)` ist Monade,
wenn `(m, map)` Funktor,
`return` und `join` natürliche Transformationen,

```
join .     return = id
join . map return = id
join . join = join . map join
```

Beispiel:

```
instance Monad [] where
    return x = [x]; join = concat
join [[1,2,3],[],[4,5]] = [1,2,3,4,5]
```

Beispiele für Monaden

```
data Id a = Id a
instance Functor Id where
    map f (Id x) = Id (f x)
instance Monad Id where
    return x = Id x; join (Id (Id x)) = Id x

data Maybe a = Just a | Nothing
instance Functor Maybe where
    map f (Just x) = Just (f x)
    map Nothing = Nothing
instance Monad Maybe where
    return x = Just x
    join (Just x) = x; join Nothing = Nothing

data Tree a = Leaf a | Node (Tree a) (Tree a)
instance Functor Tree where map f x = ...
instance Monad Tree where
    return x = Leaf x
    join (Leaf t) = t
    join (Node l r) = Node (join l) (join r)
```

State transformer monad

```
join :: m (m a) -> m a
(>>=) :: m a -> (a -> m b) -> m b
    c >>= f = join (map f c)
    join x = x >>= id

data ST s a = ST (s -> (a, s))
instance Functor (ST s) where
    map f (ST m) = ST ( \ s ->
        let (a, s') = m s in (f a, s') )
instance Monad (ST s) where
    return x = \ s -> (x, s)
    ST m >>= f = ST ( \ s ->
        let (a, s') = m s; ST m' = f a
        in  m' s' )
put :: s -> ST s ()
    put s = ST ( \ _ -> ((), s) )
get :: ST s
    get   = ST ( \ s -> (s, s) )
run :: ST s a -> s -> a
    run (ST m) s = let (a, s') = m s in a
```

Anwendung State Transformer

```
click :: ST Int Int
click = get >>= \ s ->
        put (s + 1) >>= \ () ->
        return s

label :: Tree a -> Tree Int
label t = run (labelm t) 1

labelm :: Tree a -> ST Int (Tree Int)
labelm (Leaf x) =
    click >>= \ s ->
    return (Leaf s)
labelm (Node l r) =
    labelm l >>= \ l' ->
    labelm r >>= \ r' ->
    return (Node l' r')
```

do-Notation

```
labelm (Node l r) =
    do  l' <- labelm l
        r' <- labelm r
        return (Node l' r')

click =
    do  s <- get
        put (s + 1)
        return s
```

comprehensions

`[ (x, y)`
`| x <- xs`
`, y <- ys`
`]`
	`do`
	` x <- xs`
	` y <- ys`
	` return (x, y)`
		`xs >>= \ x ->`
		`ys >>= \ y ->`
		`return (x, y)`

Monadische IO

```
import System
import IO

wc :: [Char] -> IO Int
wc name =
    do  h <- openFile name ReadMode
        c <- hGetContents h
        let n = length (lines c)
        putStrLn $ show n ++ " " ++ name
        hClose h
        return n

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

main :: IO ()
main =
    do  argv <- getArgs
        ns <- accumulate [ wc arg | arg <- argv ]
        putStrLn $ show (sum ns) ++ " " ++ "total"
```

Monaden mit Null

```
class Monad m => MonadZero m where
    zero :: m a
instance MonadZero [] where zero = []
instance MonadZero Maybe where zero = Nothing
```

Filter

```
[ (x, y)
| x <- xs
, y <- ys
, x /= y
]
```

```
do
 x <- xs
 y <- ys
 if x /= y
 return (x, y)
```

```
xs >>= \ x ->
ys >>= \ y ->
if x /= y then
 return (x, y)
else
 zero
```

refutable patterns

```
C y z <- xs
...
```

```
xs >>= \ x ->
   case x of
      C y z -> ...
      _     -> zero
```

Kleisli-Kategorien

m eine Monade in Kategorie C.

Kleisli-Kategorie von m:
Objekte: Ob C,   Pfeile: Mor$(a, m\ b)$

Operation:

```
(@@)::(a -> m b) -> (b -> m c) -> (a -> m c)
f @@ g = \ x -> f x >>= g
f @@ g = join . map g . f

return @@ f = f = f @@ return
f @@ (g @@ h) = (f @@ g) @@ h

zero :: m b; const zero :: a -> m b
const zero @@ f = const zero
f @@ const zero = const zero
```

Monaden mit Plus

```
class MonadZero m => MonadPlus m
    where (++) :: m a -> m a -> m a
zero ++ x = x = x ++ zero
x ++ (y ++ z) = (x ++ y) ++ z

instance MonadPlus [] where (++) = append

instance MonadPlus Maybe where
    Nothing ++ y = y
    Just x  ++ y = Just x
```

Distributivität

```
(@@)::m b -> m b -> m b
(@+)::(a -> m b) -> (a -> m b) -> (a -> m b)
f @+ g = \ x -> f x ++ g x

(f @+ g) @@ h =  (f @@ h) @+ (g @@ h)
f @@ (g @+ h) =? (f @@ g) @+ (f @@ h)
```

Parametrischer State Transformer

```
data STP p s a = STP (s -> p (a, s))

instance Monad p => Functor (STP p s) where
    map f (STP m) = STP ( \ s ->
        do (a, s') <- m s; return (f a, s') )

instance Monad p => Monad (STP p s) where
    return x = STP ( \ s -> return (x, s) )
    STP m >>= f = STP ( \ s ->
        do (a, s') <- m s; let STP m' = f a
           m' s' )

put :: Monad p => s -> STP p s ()
put s = STP ( \ _ -> return ((), s) )

get :: Monad p => STP p s s
get   = STP ( \ s -> return (s, s) )

instance MonadZero p => MonadZero (STP p s) where
    zero = STP ( \ s -> zero )
instance MonadPlus p => MonadPlus (STP p s) where
    STP x ++ STP y = STP ( \ s -> x s ++ y s )
```

Jones, Duponcheel: Composing Monads, 1993

Parser-Monade

```
type Parser c a = STP [] [c] a

parses :: Parser c a -> [c] -> [(a, [c])]
parses (STP f) xs = f xs

parse :: Parser c a -> [c] -> a
parse p xs =
    case parses p xs of
        [(a, [])] -> a
        []        -> error "no parse"
        [(a, _)]  -> error "incomplete parse"
        _         -> error "ambiguous parse"

item :: Parser c c
item =
    do  xs <- get
        if (not (null xs))
        put (tail xs)
        return (head xs)

eof :: Parser c ()
eof =
    do  xs <- get
        if (null xs)
        return ()
```

Parser

```
many, many1 :: Parser c a -> Parser c [a]
many  p = many1 p ++ return []
many1 p = do  x <- p; xs <- many p; return (x : xs)

sat :: (c -> Bool) -> Parser c c
sat p = do  c <- item; if p c; return c

digit :: Parser Char Char
digit = sat ( ‘elem‘ "0123456789" )

nat :: Parser Char Int
nat = do  ds <- many1 digit
          return (read ds)

op :: Parser Char String
op =  many1 (sat ( ‘elem‘ "+-*/^<>="))

data Expr = Op String Expr Expr | Nat Int

atom = do n <- nat; return (Nat n)
    ++ do ’(’ <- item; e <- expr; ’)’ <- item; return e

expr = do  a <- atom; o <- op; e <- expr
           return (Op o a e)
    ++ atom
```

Partridge, Wright: Predictive Parser Combinators, in: JFP 6(2), 1996

Zusammenfassung Teil 2

Berechnung mit Resultat und Wirkung,
Hintereinanderausführung $\rightarrow$ Monade,
assoziative Operation

Implementierung: Konstruktor-Klassen

Instanz-Deklaration versteckt Details,
Programme werden abstrakter,
übersichtlicher,

und flexibler: nachträglich Monade ändern
(Espinosa: Semantic Lego, 1994)

typische Anwendung: Parser

lineare Zustands-Monaden für Verbindung mit
Außenwelt: IO, Arrays, C-Funktionen