

Categories in programming

Benedikt Ahrens

June 11, 2009

Contents

- 1 Notions of category theory
- 2 What is functional programming?
- 3 Haskell as a category
- 4 Inductive and Coinductive Types

Some category theory

Definition: Category \mathcal{C}

consists of

- collection of objects \mathcal{O}
- collection of arrows \mathcal{A} with maps
 - source : $\mathcal{A} \rightarrow \mathcal{O}$
 - target : $\mathcal{A} \rightarrow \mathcal{O}$

For $f \in \mathcal{A}$, source $f = a$, target $f = b$, we write $f : a \rightarrow b$,
 $\text{Hom}(a, b) := \{f : a \rightarrow b\}$

Structure in a category

- *composition* of **composable** arrows
- for each object an *identity* arrow, left and right neutral under composition
- associativity of composition

Examples of categories

Category **Set**

- objects: a class of sets
- arrows $\text{Hom}(X, Y)$: the total functions from X to Y

Vect_K - the cat of vector spaces over fixed K

- objects: vector spaces over K
- arrows: K -linear maps

monoid M

- one object
- arrows: the elements of M
- composition of arrows $==$ multiplication in M

Functors

Definition

Let \mathcal{C} and \mathcal{D} be categories. A functor $F : \mathcal{C} \rightarrow \mathcal{D}$ consists of

- $F_{\mathcal{O}} : \mathcal{O}_{\mathcal{C}} \rightarrow \mathcal{O}_{\mathcal{D}}$ a map for the objects
- $F_{\mathcal{A}} : \mathcal{A}_{\mathcal{C}} \rightarrow \mathcal{A}_{\mathcal{D}}$ a map for the arrows such that
-

$$\begin{array}{ccc} c & & F_{\mathcal{O}}c \\ \downarrow f & & \downarrow F_{\mathcal{A}}f \\ c' & & F_{\mathcal{O}}c' \end{array}$$

- $F \text{id}_c = \text{id}_{F_c}$ for any object c of \mathcal{C}
- $F(g \circ_{\mathcal{C}} f) = Fg \circ_{\mathcal{D}} Ff$

Examples of functors

Example: Maybe : **Set** \rightarrow **Set**

- $X \mapsto \text{Maybe } X = 1 + X = \{*_X\} \cup X$ (add one element)
- For $f : X \rightarrow Y$ we define

$$\text{Maybe } f(x) = \begin{cases} f(x) & \text{if } x \in X \\ *_Y & \text{if } x = *_X \end{cases}$$

Power set functor $\mathcal{P} : \mathbf{Set} \rightarrow \mathbf{Set}$

- $\mathcal{P}X =$ power set of X
- for $f : X \rightarrow Y$ define

$$\begin{aligned} \mathcal{P}f : \mathcal{P}X &\rightarrow \mathcal{P}Y \\ X' &\mapsto f(X') \subset Y \end{aligned}$$

Composition of functors

Let $F : \mathcal{C} \rightarrow \mathcal{D}$, $G : \mathcal{D} \rightarrow \mathcal{E}$ be functors. Then we can define $G \circ F : \mathcal{C} \rightarrow \mathcal{E}$

- $(G \circ F)_{\mathcal{O}} := G_{\mathcal{O}} \circ F_{\mathcal{O}}$
- $(G \circ F)_{\mathcal{A}} := G_{\mathcal{A}} \circ F_{\mathcal{A}}$

$G \circ F$ is a functor.

Example:

Since $\mathcal{P} : \mathbf{Set} \rightarrow \mathbf{Set}$, we can compose \mathcal{P} with itself:

$$\mathcal{P} \circ \mathcal{P} : \mathbf{Set} \rightarrow \mathbf{Set}$$

Examples of natural transformations

For the functor $\mathcal{P} : \mathbf{Set} \rightarrow \mathbf{Set}$ we can define, for every set X , a map

$$\begin{aligned} \eta_X : X &\rightarrow \mathcal{P}X & \eta : \text{Id}_{\mathbf{Set}} &\rightarrow \mathcal{P} \\ x &\mapsto \{x\} \end{aligned}$$

$$\begin{aligned} \mu_X : \mathcal{P}\mathcal{P}X &\rightarrow \mathcal{P}X & \mu : \mathcal{P}\mathcal{P} &\rightarrow \mathcal{P} \\ \{A_i\} &\mapsto \bigcup_i A_i \end{aligned}$$

η and μ fulfill some compatibility conditions which make them **natural transformations**.

Definition of Monad

A **monad** is a triple (F, η, μ) where

- $F : \mathcal{C} \rightarrow \mathcal{C}$ is a functor
- $\eta : \text{Id}_{\mathcal{C}} \rightarrow F$ is a nat. transformation
- $\mu : F^2 \rightarrow F$ is a nat. transformation

such that for any object c of \mathcal{C} the following commute:

$$\begin{array}{ccc}
 Fc & \xrightarrow{F\eta_c} & FFc & \xleftarrow{\eta_{Fc}} & Fc \\
 & \searrow & \downarrow \mu_c & & \swarrow \\
 & & Fc & &
 \end{array}$$

$$\begin{array}{ccc}
 FFFc & \xrightarrow{\mu_{Fc}} & FFc \\
 F\mu_c \downarrow & & \downarrow \mu_c \\
 FFc & \xrightarrow{\mu_c} & Fc
 \end{array}$$

Some words about functional programming

Traditional programming languages

$x = x + 1;$

- what is x mathematically ? it is not a (constant) function !
- fundamental notion: **state** $\Gamma \in \{\text{variables}\} \rightarrow \mathbb{N} \cup \mathbb{R} \cup \dots$
- every statement is a transformation of environments $\Gamma \mapsto \Gamma'$

This makes it difficult to reason about a program, i.e. prove its correctness.

functional programming languages

In a pure functional programming language, every function is also a function in the mathematical sense:

- no pointers
- no variables - but constant functions
- no loops - recursion as magic bullet

We can prove properties of programs by the usual mathematical means!

Partial functions are allowed !

Implementation of factorial

factorial in maths

$$f(n) = \begin{cases} 1 & \text{if } n = 0, \\ n * f(n - 1) & \text{otherwise} \end{cases}$$

factorial in Haskell (FP)

```
fac n | n == 0    = 1
      | otherwise = n * fac (n - 1)
```

The category 'Haskell'

category \mathcal{H}

- objects: Haskell data types
- morphisms: functions
- composition and identity: ...

functors $\mathcal{H} \rightarrow \mathcal{H}$ (and bi-, tri-, ...)

"parametrized data types"

- Lists: [Bool], [Integer]
- **data Maybe** $x = \mathbf{Nothing} \mid \mathbf{Just} \ x$

Our functors are not complete yet...

fmap - the arrow function for functors

Implementation of the Functor structure by a **typeclass**:

```
class Functor f where
```

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

```
instance Functor Maybe where
```

```
  fmap f Nothing = Nothing
```

```
  fmap f (Just x) = Just (f x)
```

The function fmap is overloaded, i.e. fmap is the arrow function for every instance of Functor.

Attention: Compatibility rules for fmap have to be checked by hand!

An alternative definition of 'Monad'

Let (F, η, μ) be a monad. For types a, b define the **bind** operator:

$$\begin{aligned}(\gg=) &: (a \rightarrow Fb) \rightarrow Fa \rightarrow Fb \\ (\gg=)(f) &:= \mu_b \circ Ff\end{aligned}$$

We will prefer uncurried infix notation:

$$\begin{aligned}(\gg=) &: Fa \times (a \rightarrow Fb) \rightarrow Fb \\ x \gg= f &:= (\mu_b \circ Ff)(x)\end{aligned}$$

The three monad laws

Let $f : a \rightarrow Fb$, $x : a$, $y : Fa$ and $g : b \rightarrow Fc$. Then

$$\eta_a(x) \gg= f == f(x)$$

$$y \gg= \eta_a == y$$

$$(x \gg= f) \gg= g == x \gg= (\lambda z \rightarrow f(z) \gg= g)$$

Theorem

$\mathcal{C} = (\mathcal{O}, \mathcal{A})$ a category, $F_{\mathcal{O}} : \mathcal{O} \rightarrow \mathcal{O}$ an object mapping and

$$\eta : \mathcal{O} \rightarrow \mathcal{A}, \quad \eta : a \mapsto (\eta_a : a \rightarrow F_{\mathcal{O}}a)$$

$$\gg=: F_{\mathcal{O}}a \times (a \rightarrow F_{\mathcal{O}}b) \rightarrow F_{\mathcal{O}}b.$$

Then $(F_{\mathcal{O}}, \eta, \gg=)$ is a monad iff the three monad laws are fulfilled.

Proving the alternative monad definition adequate

Attention: η a priori not a nat. transformation (consequence!)
In an environment as in theorem and $f : a \rightarrow b$, define

$$\begin{aligned} F_{\mathcal{A}}f : F_{\mathcal{O}}a &\rightarrow F_{\mathcal{O}}b \\ x &\mapsto x \gg= \eta_b \circ f \end{aligned}$$

$$\begin{aligned} \mu_a : F_{\mathcal{O}}F_{\mathcal{O}}a &\rightarrow F_{\mathcal{O}}a \\ x &\mapsto x \gg= \text{id}_{F_{\mathcal{O}}a} \end{aligned}$$

Then prove the functor and monad rules for $(F = (F_{\mathcal{O}}, F_{\mathcal{A}}), \eta, \mu)$.

Example: lists

- For every type a we can define the type $[a]$ of lists with entries in a .
- Functoriality: for $f : a \rightarrow b$ we define $\text{map } f : [a] \rightarrow [b]$ to be the map which applies f to every element in the list.

`square` $x = x * x$

map `square` $[1, 2, 3] = [1, 4, 9]$

- Monad: $\eta_a : a \rightarrow [a], x \mapsto [x]$ (a list of one element)
 $\mu_a : [[a]] \rightarrow [a], \mu_a = \text{concat}$:

concat $[[1], [2, 5]] = [1, 2, 5]$

List is a 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
```

```
instance Monad [ ] where
```

```
  xs >>= f  = concat (map f xs)
```

```
  return x  = [x]
```

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

```
Prelude List Monad> join [[2],[1]]
```

```
[2,1]
```

The IO monad

'read user input' shall be a function. But how ?

```
data IO a =  
    ST((String , String) -> ((String , String) , a) )
```

```
getChar :: ST ((String , String) ->  
                ((String , String) , Char) )
```

```
getChar = ST (\(char:chars , t) ->  
                ((chars , t) , char)
```

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

```
putChar c = ST((is , os) -> ((is , os++[c]) , ()))
```

```
main = getChar >>= putChar
```

About inductive and coinductive types

Inductive Types

Definition

data Nat = Zero | Succ Nat

Nat is the smallest set which is stable under the constructors.

Peano:

- $\text{Succ} : \text{Nat} \rightarrow \text{Nat}$ is an injective mapping
- $\forall n : \text{Nat}, \text{Zero} \neq \text{Succ}(n)$

Recursion principle

$g : \text{Nat} \rightarrow a$ can be defined by $g = (x : a, h : a \rightarrow a)$ s.t.

$$\begin{aligned}g(\text{Zero}) &== b \\g(\text{Succ}(n)) &== h(g(n))\end{aligned}$$

same for lists

Definition

```
data List a = Nil | Cons a (List a)
```

Recursion principle

$g : \text{List } a \rightarrow b$ can be defined by $g = (x : b, h : a \times b \rightarrow b)$ s.t.

$$\begin{aligned}g(\text{Nil}) &== x \\g(\text{Cons}(x, xs)) &== h(x, (g(xs)))\end{aligned}$$

Not that different from the principle for Nat, is it?

Inductive types categorically

Definition

\mathcal{C} a category, $F : \mathcal{C} \rightarrow \mathcal{C}$ a functor. An F -algebra is a tuple $(c, \varphi : Fc \rightarrow c)$.

Definition

A morphism of F -algebras $(f, \varphi, \psi) : (c, \varphi) \rightarrow (d, \psi)$ is a commutative diagram

$$\begin{array}{ccc} Fc & \xrightarrow{\varphi} & c \\ Ff \downarrow & & \downarrow f \\ Fd & \xrightarrow{\psi} & d \end{array}$$

The category of F -algebras

Theorem

The F -algebras of \mathcal{C} form a category $F - \text{Alg}$ when identity and composition are defined as

$$\begin{aligned} \text{id}_\varphi &= (\text{id}_\varphi, \varphi, \varphi) \\ (g, \varphi_2, \varphi_3) \circ (f, \varphi_1, \varphi_2) &= (g \circ f, \varphi_1, \varphi_3) \end{aligned}$$

Identity and composition diagrams for F -Alg

Identity diagram:

$$\begin{array}{ccc} Fc & \xrightarrow{\varphi} & c \\ F \text{ id} = \text{id} \downarrow & & \downarrow \text{id} \\ Fc & \xrightarrow{\varphi} & c \end{array}$$

Composition:

$$\begin{array}{ccc} Fc & \xrightarrow{\varphi_1} & c \\ Ff \downarrow & & \downarrow f \\ Fd & \xrightarrow{\varphi_2} & d \\ Fg \downarrow & & \downarrow g \\ Fe & \xrightarrow{\varphi_3} & e \end{array}$$

Two commutative diagrams glued together give a commutative diagram!

Initial F -algebras

An **initial F -algebra** $(\mu F, \text{in})$ is an initial object in the category $F\text{-Alg}$, i. e. for all (c, φ) there is precisely one arrow f of \mathcal{C} such that the following commutes:

$$\begin{array}{ccc}
 F\mu F & \xrightarrow{\text{in}} & \mu F \\
 Ff \downarrow & & \downarrow f = \langle \varphi \rangle \\
 Fc & \xrightarrow{\varphi} & c
 \end{array}$$

Definition (fold/foldr/cata)

$$\begin{aligned}
 \text{fold} : (Fc \rightarrow c) &\rightarrow \mu F \rightarrow c \\
 \varphi &\mapsto \langle \varphi \rangle
 \end{aligned}$$

About catamorphisms

Some equalities:

$$\begin{aligned}\varphi \circ Ff &== (\llbracket \varphi \rrbracket) \circ \text{in} \\ \text{id} &== (\llbracket \text{id} \rrbracket)\end{aligned}$$

$$\begin{array}{ccc} Fc & \xrightarrow{\varphi} & c \\ Fg \downarrow & & \downarrow g \\ Fd & \xrightarrow{\psi} & d \end{array} \quad \Rightarrow \quad (\llbracket \psi \rrbracket) == g \circ (\llbracket \varphi \rrbracket)$$

Initial algebra is an isomorphism

Theorem (Lambek)

The initial F -algebra $\text{in}_F : F\mu F \rightarrow \mu F$ is an isomorphism in \mathcal{C} with inverse

$$\text{out}_F = \text{in}_F^{-1} = \langle F \text{in} \rangle$$

Remark

There is an explicit description for the initial algebra:

$$\mu F = \text{forall } x. (F x \rightarrow x) \rightarrow x$$

Example: Naturals

Consider $\text{Maybe} : \mathbf{Set} \rightarrow \mathbf{Set}$, $\text{Maybe } X = 1 + X$. The corresponding data type definition is

```
data Maybe x = Nothing | Just x
```

The initial algebra μ_{Maybe} is $\text{in}_{\text{Nat}} : 1 + \text{Nat} \rightarrow \text{Nat}$. We have

$$\begin{array}{ccc}
 1 + \text{Nat} & \xrightarrow{\text{in}=[\text{zero},\text{succ}]} & \text{Nat} \\
 \downarrow 1+f & & \downarrow f=[\varphi] \\
 1 + X & \xrightarrow{\varphi=[b,h]} & X
 \end{array}$$

$$f \circ \text{zero} = b$$

$$f \circ \text{succ} = h \circ f$$

Defining functions from Naturals as folds

```
addN :: Nat -> Nat -> Nat
addN x y = fold phi x where
  phi Z      = y
  phi (S n) = succN n
```

```
mulN :: Nat -> Nat -> Nat
mulN x y = fold phi x where
  phi Z      = y
  phi (S n) = addN y n
```

Other functions (like factorial) need more complex recursion principles (or dirty tricks like tupling).

Example: polymorphic lists

A polymorphic inductive type is the initial algebra of a bifunctor with one fixed argument: Consider

$L : \mathbf{Set} \times \mathbf{Set} \rightarrow \mathbf{Set}$, $L(A, X) = 1 + A \times X$.

```
data L a x = N | C a x
```

```
instance Functor (L a) where
```

```
  fmap f N = N
```

```
  fmap f (C a xs) = C a (f xs)
```

The initial algebra $\mu(LA)$ is $\text{in}_{(\text{List } A)} : 1 + A \times \text{List } A \rightarrow \text{List } A$.

Principle of recursion coming with Lists

$$\begin{array}{ccc} 1 + A \times \text{List } A & \xrightarrow{\text{in}=[\text{Nil}, \text{Cons}]} & \text{List } A \\ \text{id} + (\text{id} \times f) \downarrow & & \downarrow f = (\varphi) \\ 1 + A \times X & \xrightarrow{\varphi=[b, h]} & X \end{array}$$

$f \circ \text{Nil} = b$
 $f \circ \text{Cons} = h \circ (\text{id} \times f)$

Defining functions from lists as folds

```
lengthL :: List a -> Nat
```

```
lengthL = cata phi where
```

```
  phi N      = zeroN
```

```
  phi (C _ n) = succN n
```

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

```
mapList f = cata phi where
```

```
  phi N      = nilL
```

```
  phi (C a bs) = consL (f a) bs
```

Coinductive types

A coinductive type is a **terminal object** $(\nu F, \text{out})$ in a category with objects $\varphi : c \rightarrow Fc$.

$$\begin{array}{ccc} c & \xrightarrow{\varphi} & Fc \\ \downarrow f = [\varphi] & & \downarrow Ff \\ \nu F & \xrightarrow{\text{out}} & F\nu F \end{array}$$

There is a similar recursion principle which allows to define maps **towards** the coinductive type.

Definition (unfold/ana)

$$\begin{aligned} \text{unfold} : (c \rightarrow Fc) &\rightarrow c \rightarrow \nu F \\ \varphi &\mapsto [\varphi] \end{aligned}$$

Final remarks

- A cpo is a set s.t. every directed subset has a supremum and which has a least element.
- Every set S can be turned into a cpo by adding a least element \perp and introducing a flat order with $\perp \leq s$ and $s \leq s$ for every $s \in S$ and no other order relations (Wikipedia).
- The Haskell category is not like **Set**, but like **CPO**, the CCC of cpos (not necessarily $f(\perp) = \perp$)

```
Prelude List Monad> :t undefined  
undefined :: a
```

- Here the initial and terminal F -algebras coincide.

References



Vene, Varmo. *Categorical programming with inductive and coinductive types*, Dissertationes mathematicae universitatis tartuensis 23 (2002)



Klinger, Stefan. *The Haskell Programmers Guide to the IO Monad – Dont Panic.*, Technical report (2005)