

Category Theory and Haskell

Parallel Universes

Siva Kalyan and T. Mark Ellison

April 19, 2017

Australian National University

Table of contents

1. Categories
2. Functors
3. Natural transformations
4. Cartesian-Closed Categories
5. Applicatives
6. Adjoint Functors
7. Monads
8. Further Reading

Haskell and Category Theory

Haskell	Category Theory
Category	Category
Type	Object
Function	Morphism
<u>Hask</u>	<u>Set</u>
...	Terminal Objects
Value	Global Element
Tuple	Product
Currying, Function Application	Cartesian Closure
Type Constructor, Functor	Functor
...	Natural Transformation
Applicative	...
...	Adjoint Functor Pair
Monad	Monad

Categories

Categories

A category $\underline{\mathbf{C}}$ consists of

1. a class $\text{Obj}(\underline{\mathbf{C}})$ of *objects*, and
2. for each pair of objects $A, B \in \text{Obj}(\underline{\mathbf{C}})$, a set $\text{Hom}_{\underline{\mathbf{C}}}(A, B)$ of *arrows* (or *morphisms*) from A to B , known as a *hom-set*.

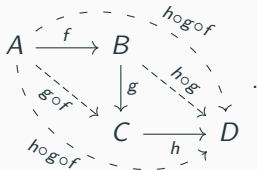
$$\begin{array}{ccc} & \text{Hom}_{\underline{\mathbf{C}}}(A, B) & \\ A & \begin{array}{c} \rightrightarrows \\ \longrightarrow \end{array} & B \end{array}$$

Many familiar parts of Haskell form a category **Hask**: objects are *types* (**Int**, **Char**, etc.), and arrows are *functions* between types (e.g. **ord** :: **Int** \rightarrow **Char**).

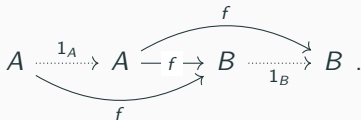
Category Laws

In a category $\underline{\mathbf{C}}$:

1. Given arrows $f: A \rightarrow B$ and $g: B \rightarrow C$ in $\underline{\mathbf{C}}$, the *composition* $g \circ f: A \rightarrow C$ ($= g.f$) is also in $\underline{\mathbf{C}}$.
2. Given arrows $f: A \rightarrow B$, $g: B \rightarrow C$ and $h: C \rightarrow D$,
 $(h \circ g) \circ f = h \circ (g \circ f) = h \circ g \circ f$:



3. Every object $A \in \text{Obj}(\underline{\mathbf{C}})$ is associated with an *identity arrow* $1_A: A \rightarrow A$ ($= \text{id}$). Given any arrow $f: A \rightarrow B$, we have



Examples

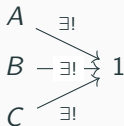
	<u>Set</u>	<u>Hask</u>	<u>POrd</u>	<u>Cat</u>
Objects	sets	types	items	small cats
Morphisms	functions	functions	$a \leq b$	functors
Composition	$f \circ g$	$f.g$	transitivity	$F \circ G$
Identity	1_A	id	$a = a$	$1_{\underline{C}}$

Not everything in Haskell can be in Hask if we want it to be a category. Every type in the language contains a **Bottom** (\perp) or **undefined** value, but these 'values' cause mayhem with the category laws (in particular the **Identity** constraint). So when we talk about Hask we'll be talking about vanilla Hask without these abnormal values. Haskell wiki page on Hask

Category Theory: Terminal Objects

A *terminal object* is a type 1 (a.k.a. T) in $\text{Obj}(\underline{\mathbf{C}})$, such that there is only a single mapping from any other type A onto that type:

$$\forall A \in \text{Obj}(\underline{\mathbf{C}}), |\text{Hom}_{\underline{\mathbf{C}}}(A, 1)| = 1.$$



In **Hask**:

```
1  () -- the type corresponding to 1, containing only itself
2  terminalMap :: t -> ()
3  terminalMap _ = ()
```


Global Elements

A *global element* of an object A in category $\underline{\mathbf{C}}$ with terminal object 1 is an arrow $a : 1 \rightarrow A$.

$$1 \xrightarrow{a} A$$

In **Hask**, if we have a value v in some type a , we can upgrade it to the global element by use of **const** v .

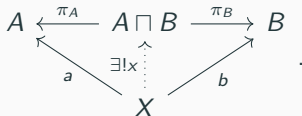
```
1  const :: a -> b -> a  -- but for our purposes, choose b = ()  
2  const v = \ _ -> v
```

Examples

	<u>Set</u>	<u>Hask</u>	<u>POrd</u>	<u>Cat</u>
Objects	sets	types	items	small cats
Morphisms	functions	functions	$a \leq b$	functors
Composition	$f \circ g$	$f.g$	transitivity	$F \circ G$
Identity	1_A	id	$a = a$	$1_{\underline{c}}$
Terminal obj.	$\{*\}$	$()$	upper bound	$\underline{1}$

Products

Given objects A, B in $\underline{\mathbf{C}}$ there may be a (*pairwise*) *product* $A \sqcap B \in \text{Obj}(\underline{\mathbf{C}})$ and *projection arrows* $\pi_A: A \sqcap B \rightarrow A$ and $\pi_B: A \sqcap B \rightarrow B$ such that for any object X in the same category and arrows $a: X \rightarrow A$ and $b: X \rightarrow B$ there is a *unique* arrow $x: X \rightarrow A \sqcap B$ such that $a = \pi_A \circ x$ and $b = \pi_B \circ x$:



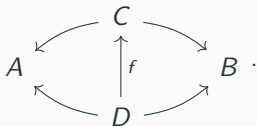
In other words: Given a particular way of mapping X to A and to B , there's only *one* way of mapping X to $A \sqcap B$ such that everything's consistent.

Products

Alternatively, the triplet $\langle A \sqcap B, \pi_A, \pi_B \rangle$ is a *terminal object* in the category whose objects are diagrams of the form

$$A \longleftarrow C \longrightarrow B ,$$

and whose arrows are (commutative) diagrams of the form



Products in Haskell

```
1  (a,b) -- the type containing pairs from types a and b ( $A \times B$ )
2  fst :: (a,b) -> a -- the projection function  $\pi_A$ 
3  fst (x,y) = x
4  snd :: (a,b) -> b -- the projection function  $\pi_B$ 
5  snd (x,y) = y
6  factorThroughProd :: (c -> a) -> (c -> b) -> (c -> (a,b))
7  factorThroughProd f g = \ x -> (f x,g x)
```

It should be obvious that

fst.(factorThroughProd f g) = f, and

snd.(factorThroughProd f g) = g.

Examples

	<u>Set</u>	<u>Hask</u>	<u>POrd</u>	<u>Cat</u>
Objects	sets	types	items	small cats
Morphisms	functions	functions	$a \leq b$	functors
Composition	$f \circ g$	$f.g$	transitivity	$F \circ G$
Identity	1_A	id	$a = a$	$1_{\underline{C}}$
Terminal obj.	$\{*\}$	$()$	upper bound	$\underline{1}$
Product	$A \times B$	(a,b)	$\min(a,b)$	$\underline{C} \times \underline{D}$

Exponential Objects

Given objects A and B in $\underline{\mathbf{C}}$, an *exponential object* B^A (also written $[A \rightarrow B]$) is an object with an arrow eval_B^A such that for any C and any arrow $f: C \sqcap A \rightarrow B$,

$$\begin{array}{ccc} C \sqcap A & & \\ \downarrow \exists! & \searrow f & \\ B^A \sqcap A & \xrightarrow{\text{eval}_B^A} & B \end{array} .$$

Alternatively, the pair $\langle B^A, \text{eval}_B^A \rangle$ constitutes a terminal object in the category whose objects are diagrams of the form

$$C \sqcap A \longrightarrow B ,$$

and whose arrows are commutative diagrams of the form

$$\begin{array}{ccc} D \sqcap A & & \\ \downarrow & \searrow & \\ C \sqcap A & \searrow & B \end{array} .$$

Exponential Objects in Haskell

In **Hask**, the exponential object of two types **a** and **b** is the *function type* $(a \rightarrow b)$ (it's akin to the *hom-set* of **a** and **b**). Let's see how this satisfies the above definition.

```
1  eval :: ((a -> b),a) -> b
2  eval (f,x) = f x
3  factoredArrow :: ((c,a) -> b) -> ((c,a) -> ((a -> b),a))
4  factoredArrow f = \ (y,x) -> ((\ x' -> f(y,x')),x)
```

(Spot the currying!)

It can be proven that $\text{eval} \cdot (\text{factoredArrow } f) = f$ — and that **factoredArrow** is the *only* arrow for which this is true.

Functors

Functors

A *functor* is a mapping $F: \underline{\mathbf{C}} \rightarrow \underline{\mathbf{D}}$ that takes objects in $\underline{\mathbf{C}}$ to objects in $\underline{\mathbf{D}}$ and arrows in $\underline{\mathbf{C}}$ to arrows in $\underline{\mathbf{D}}$, in such a way that

1. for any $A \in \text{Obj}(\underline{\mathbf{C}})$, $F(1_A) = 1_{F(A)}$:

$$\begin{array}{ccc} A & \xrightarrow{1_A} & A \\ \downarrow & & \downarrow \\ F(A) & \xrightarrow{1_{F(A)}} & F(A) \end{array} ;$$

2. for any $f: A \rightarrow B$ and $g: B \rightarrow C$ in $\underline{\mathbf{C}}$, $F(g \circ f) = F(g) \circ F(f)$:

$$\begin{array}{ccccc} & & B & & \\ & f \nearrow & \downarrow & \nwarrow g & \\ A & \xrightarrow{\quad\quad\quad} & & \xrightarrow{\quad\quad\quad} & C \\ & \downarrow & & \downarrow g \circ f & \\ & F(B) & & & \\ \downarrow & \nearrow F(f) & & \nwarrow F(g) & \\ F(A) & \xrightarrow{\quad\quad\quad} & & \xrightarrow{\quad\quad\quad} & F(C) \\ & F(g) \circ F(f) & & & \end{array} .$$

Functors in Haskell

In Haskell, functors are *type constructors*: they take a type (a) and produce another type ($F\ a$); and via `fmap`, they take an arrow between two types ($a \rightarrow b$) and produce an arrow between the images of those two types ($F\ a \rightarrow F\ b$).

E.g. the list constructor:

```
1 data [] a = [] | a : [a] -- "[]" is the type constructor for lists
2 fmap f [] = [] -- mapping f over an empty list does nothing
3 fmap f (x : xs) = (f x) : (fmap f xs)
4 -- to turn f into a list function, apply f to the head of the list ,
5 -- apply the list version of f to the tail of the list , and construct
```

You can verify the functor laws in Hask:

`fmap id (x : xs) = (id x) : (fmap id xs) = id (x : xs)`, and that
`fmap f (fmap g (x : xs)) = fmap f ((g x) : (fmap g xs))`
`= (f g x) : (fmap f (fmap g xs)) = fmap f g (x : xs)`.

Examples

	<u>Set</u>	<u>Hask</u>	<u>POrd</u>	<u>Cat</u>
Objects	sets	types	items	small cats
Morphisms	functions	functions	$a \leq b$	functors
Composition	$f \circ g$	$f.g$	transitivity	$F \circ G$
Identity	1_A	id	$a = a$	$1_{\underline{C}}$
Terminal obj.	$\{*\}$	$()$	upper bound	$\underline{1}$
Product	$A \times B$	(a,b)	$\min(a, b)$	$\underline{C} \times \underline{D}$
Endofunctors	functors	type const.	OPTs	nat. trans.

Natural transformations

Natural Transformations

A *natural transformation* α is a mapping between two functors $F: \underline{\mathbf{C}} \rightarrow \underline{\mathbf{D}}$ and $G: \underline{\mathbf{C}} \rightarrow \underline{\mathbf{D}}$. It consists of a family of arrows in $\underline{\mathbf{D}}$ (the *components* of α) which map each object $F(A)$ in the image of F to the corresponding object $G(A)$ in the image of G . Crucially, the following diagram always commutes:

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ \alpha_A \downarrow & & \downarrow \alpha_B \\ G(A) & \xrightarrow{G(f)} & G(B) \end{array} \quad .$$

In other words, you can “jump across” from F to G at any time; it doesn’t matter when.

Natural Transformations in Haskell

A natural transformation in Haskell is given by a function between two type constructors. It allows us to “unwrap” one constructor and “repackage” the type with the other. If `mu` is a natural transformation, the following is necessarily true (cf. the commutative diagram):

```
1  mu :: Functor f, Functor g => f a -> g a
2  mu . (fmap k) = fmap (mu . k)
```

According to Milewski, *all* functions with the above type signature are natural transformations.

In the special case where `f` is the identity functor `id`:

```
1  nu :: Functor g => a -> g a
2  nu . k = fmap (nu . k)
```

Cartesian-Closed Categories

Cartesian-Closed Categories (CCC)

There is a terminal object 1 .

There are binary products \sqcap .

There is a two-argument functor taking $A \sqcap B$ onto B^A , obeying the following rules:

$$A \cong 1 \sqcap A \cong A^1$$

$$\mathrm{Hom}_{\underline{\mathbf{C}}}(A \sqcap B, C) \cong \mathrm{Hom}_{\underline{\mathbf{C}}}(A, C^B) \quad (4.1)$$

The latter relation is called the *Howard-Curry isomorphism*, or *currying*.

Cartesian-Closed Categories

Set the singleton set, pairs, sets of functions

Hask $()$, (a, b) , $a \rightarrow b$

There are more examples, but they're pretty complicated.

Applicatives

Applicative Functors

An *applicative* functor F is a functor $\underline{\mathbf{C}} \rightarrow \underline{\mathbf{D}}$ such that:

1. $\underline{\mathbf{C}}$ is CCC,
2. $im(F)$ is CCC,
3. F preserves terminal objects, i.e. $F(1_{\underline{\mathbf{C}}}) = 1_{\underline{\mathbf{D}}}$,
4. F preserves products, i.e. $F(A \sqcap B) = F(A) \sqcap F(B)$, and
5. F preserves the power functor, i.e. $(FB)^{FA} = F(B^A)$.

Applicative Functors

```
1 class Functor f => Applicative f where  
2   -- | Lift a value.  
3   pure :: a -> f a  
4   -- | Sequential application.  
5   (<*>) :: f (a -> b) -> f a -> f b
```

`pure` applies the functor to global elements (arrows from 1 to a).

$$\phi : A^1 \rightarrow F(A^1)$$

$$\phi \circ \epsilon_a \mapsto \epsilon'_a$$

`<*>` takes the image of an arrow and of a global element and constructs the image of the composition (also a global element).

$$\psi : F(B^A) \times F(A^1) \rightarrow F(B^1)$$

$$\psi \circ (\epsilon_f, \epsilon_{a'}) \circ \delta \mapsto \epsilon'_b$$

The Laws of Applicatives

From Wikibooks: *Haskell* chapter on **Applicative Functors**:

1. **Identity** `pure id <*> v = v`
2. **Homomorphism** `pure f <*> pure x = pure (f x)`
3. **Interchange** `u <*> pure y = pure ($ y) <*> u`
4. **Composition** `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`

Applicatives

The Identity Law

pure **id** <*> v = v

$$\lambda \circ (\phi \circ H(1_A; 1, A^A), H(v; 1, FA)) \circ \delta = H(v; 1, FA)$$

$$\begin{array}{ccc} 1 & \xrightarrow{\phi} & F1 \\ \epsilon_{1_A}, \epsilon_v \downarrow & & \downarrow F_{\epsilon_{1_A}, F_{\epsilon_v}} \\ A^A \times (FA) & \xrightarrow{\phi, 1} & F(A^A) \times (FA) \\ & & \downarrow = \\ & & (FA)^{FA} \times (FA) \\ & & \downarrow \lambda_{FA, FA} \\ & & FA \end{array}$$

Applicative Functors Example

```
1 instance Applicative Maybe where
2   pure x = Just x
3   Nothing <*> _ = Nothing
4   (Just f) <*> something = fmap f something
```


Applicative Functors Example

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

This definition of applicative `[]` only holds for lists of the same length.

Adjoint Functors

Adjoint Functors

A 4-tuple $(F, G, \varepsilon, \eta)$ is an *adjunction* between two categories $\underline{\mathbf{C}}$ and $\underline{\mathbf{D}}$ when:

1. F is a functor from $\underline{\mathbf{C}} \rightarrow \underline{\mathbf{D}}$
2. G is a functor from $\underline{\mathbf{D}} \rightarrow \underline{\mathbf{C}}$
3. ε is a natural transformation from $F \circ G \rightarrow 1_{\underline{\mathbf{C}}}$ (the *counit* of adjunction)
4. η is a natural transformation from $1_{\underline{\mathbf{D}}} \rightarrow G \circ F$ (the *unit* of adjunction)
5. $(\varepsilon F) \circ (F\eta) = 1_F$
6. $(G\varepsilon) \circ (\eta G) = 1_G$

Another way of talking about adjoints is that (F, G) composed with Hom form an adjoint pair when there is a natural isomorphism

$$\text{Hom}(F-, -) \cong_{\Phi} \text{Hom}(-, G-)$$

Examples

<u>C</u>	<u>D</u>	<i>F</i>	<i>G</i>
<u>Set</u>	<u>Set²</u>	Δ	\sqcap
<u>Set²</u>	<u>Set</u>	\sqcup	Δ
<u>Set</u>	<u>Grp</u>	Free	Forgetful
<u>Set</u>	<u>Top</u>	Discrete	Forgetful
<u>Top</u>	<u>Set</u>	Forgetful	Trivial

Adjoint Functors

Although Haskell doesn't talk about adjoints as much as Monads, one adjunction-pair is fundamental to FP: the Currying adjunction.

$$\text{Hom}_{\underline{\mathbf{C}}}(X \times Y, Z) \cong \text{Hom}_{\underline{\mathbf{C}}}(X, Z^Y)$$

Think of product-with- Y and to-the-power-of- Y as functors, then product is the left-adjoint of power.

Monads

Monads

A *monad* is a triple (T, η, μ) where

1. $T : \underline{\mathbf{C}} \rightarrow \underline{\mathbf{C}}$ is an endofunctor,
2. $\eta : 1_{\underline{\mathbf{C}}} \rightarrow T$ is the counit of adjunction
3. $\mu : T \circ T \rightarrow T$ is the unit of adjunction

$$\begin{array}{ccc} T^3 & \xrightarrow{T\mu} & T^2 \\ \mu T \downarrow & & \downarrow \mu \\ T^2 & \xrightarrow{\mu} & T \end{array}$$

M2

$$\begin{array}{ccc} T & \xrightarrow{\eta T} & T^2 \\ T\eta \downarrow & \searrow & \downarrow \mu \\ T^2 & \xrightarrow{\mu} & T \end{array}$$

Wikipedia, `Monad_(category_theory)`

Monads

Earlier definition of monad.

```
1  class Functor m => Monad m where
2      return      :: a -> m a
3      join        :: m (m a) -> m a
```

King & Wadler 1993

This is how it is defined now.

```
1  class Applicative m => Monad m where
2      (>>=)       :: forall a b. m a -> (a -> m b) -> m b
3      return      :: a -> m a
```

GHC.Base base-4.9.1.0

```
1  (>>=) m g = join (fmap g m)
```


Monad Laws

(omitting two that guarantee functorhood of `f`)

```
1  fmap (f . return) = return . f  -- return is a nat trans
2  fmap (f . join) = join . $ fmap (fmap f)  -- join is a nat trans
3  join . (fmap join) = join . join  -- diagram M1
4  join . return = id  -- bottom-left of M2
5  join . (fmap return) = id  -- top-right of M2
```

King & Wadler 1993

- Arrows
- Comonads
- Lens
- Kleisli Arrows

Further Reading

Further Reading



S. Diel.

Blog.

Personal webpage.



D. Elkins.

Calculating monads with category theory.

The Monad.Reader, 13:73–91, 2009-03-12.



B. Milewski.

Category theory for programmers.

Personal webpage.