Category Theory and Haskell

Parallel Universes

Siva Kalyan and T. Mark Ellison

April 19, 2017

Australian National University

Table of contents

Haskell and Category Theory

Haskell	Category Theory
Category	**Category**
Type	**Object**
Function	**Morphism**
Hask	**Set**
...	**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 $\mathrm{Obj}(\underline{\mathbf{C}})$ of *objects*, and
2. for each pair of objects $A, B \in \mathrm{Obj}(\underline{\mathbf{C}})$, a set $\mathrm{Hom}_{\underline{\mathbf{C}}}(A, B)$ of *arrows* (or *morphisms*) from A to B, known as a *hom-set*.

$$A \overset{\mathrm{Hom}_{\underline{\mathbf{C}}}(A,B)}{\Rrightarrow} B$$

Many familiar parts of Haskell form a category $\underline{\mathbf{Hask}}$: objects are *types* (**Int**, **Char**, etc.), and arrows are *functions* between types (e.g. `ord :: Int -> Char`).

Category Laws

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

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

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[dr, dashed, "g \circ f"'] \arrow[drr, dashed, bend left=60, "h \circ g \circ f"] \arrow[drr, dashed, bend right=60, "h \circ g \circ f"'] & B \arrow[d, "g"] \arrow[dr, dashed, "h \circ g"] & \\ & C \arrow[r, "h"'] & D \end{tikzcd} .$$

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

$$\begin{tikzcd} A \arrow[r, dotted, "1_A"] \arrow[rr, bend right, "f"'] & A \arrow[r, "f"] \arrow[rr, bend left, "f"] & B \arrow[r, dotted, "1_B"'] & B \end{tikzcd} .$$

Examples

	Set	**Hask**	**POrd**	**Cat**
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**	$a = a$	$1_{\underline{\mathbf{C}}}$

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 $\mathrm{Obj}(\underline{\mathbf{C}})$, such that there is only a single mapping from any other type A onto that type:

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

$$\begin{tikzcd}
A \arrow[dr, "\exists!"] & \\
B \arrow[r, dashed, "\exists!" description] & 1 \\
C \arrow[ur, "\exists!"'] &
\end{tikzcd}$$

In **Hask**:

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

Global Elements

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

$$1 \xrightarrow{a} A$$

In $\underline{\mathbf{Hask}}$, if we have a value `v` in some type `a`, we can upgrade it to the global element by use of `const v`.

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

Examples

	$\underline{\mathbf{Set}}$	$\underline{\mathbf{Hask}}$	$\underline{\mathbf{POrd}}$	$\underline{\mathbf{Cat}}$
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**	$a = a$	$1_{\underline{\mathbf{C}}}$
Terminal obj.	$\{*\}$	`()`	upper bound	$\underline{\mathbf{1}}$

Products

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

$$\begin{tikzcd}
A & A \sqcap B \arrow[l, "\pi_A"'] \arrow[r, "\pi_B"] & B \\
& X \arrow[ul, "a"] \arrow[u, dotted, "\exists! x"] \arrow[ur, "b"'] &
\end{tikzcd}.$$

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

$$\begin{tikzcd} & C \arrow[dl] \arrow[dr] & \\ A & & B \\ & D \arrow[ul] \arrow[ur] \arrow[uu, "f"'] & \end{tikzcd} .$$

Products in Haskell

```haskell
(a,b) -- the type containing pairs from types a and b (A ⊓ B)
fst :: (a,b) -> a -- the projection function π_A
fst (x,y) = x
snd :: (a,b) -> b -- the projection function π_B
snd (x,y) = y
factorThroughProd :: (c -> a) -> (c -> b) -> (c -> (a,b))
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

	$\underline{\mathbf{Set}}$	$\underline{\mathbf{Hask}}$	$\underline{\mathbf{POrd}}$	$\underline{\mathbf{Cat}}$
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**	$a = a$	$1_{\underline{\mathbf{C}}}$
Terminal obj.	$\{*\}$	`()`	upper bound	$\underline{\mathbf{1}}$
Product	$A \times B$	`(a,b)`	$\min(a, b)$	$\underline{\mathbf{C}} \times \underline{\mathbf{D}}$

Exponential Objects

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

$$\begin{tikzcd}
C \sqcap A \arrow[d, dotted, "\exists!"'] \arrow[dr, "f"] & \\
B^A \sqcap A \arrow[r, "\mathrm{eval}_B^A"'] & B
\end{tikzcd} .$$

Alternatively, the pair $\langle B^A, \mathrm{eval}_B^A \rangle$ 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

$$\begin{tikzcd}
D \sqcap A \arrow[d] \arrow[dr, bend left] & \\
& B \\
C \sqcap A \arrow[ur, bend right] &
\end{tikzcd} .$$

Exponential Objects in Haskell

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

```haskell
eval :: ((a -> b),a) -> b
eval (f,x) = f x
factoredArrow :: ((c,a) -> b) -> ((c,a) -> ((a -> b),a))
factoredArrow f = \ (y,x) -> ((\ x' -> f(y,x')),x)
```

(Spot the currying!)

It can be proven that `eval . (factoredArrow f) = f` — and that `factoredArrow` is the *only* arrow for which this is true.

Functors

Functors

A *functor* is a mapping $F\colon \underline{\mathbf{C}} \to \underline{\mathbf{D}}$ 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 $A \in \mathrm{Obj}(\underline{\mathbf{C}})$, $F(1_A) = 1_{F(A)}$:

$$\begin{tikzcd}
A \arrow[r, "1_A"] \arrow[d, mapsto] & A \arrow[d, mapsto] \\
F(A) \arrow[r, "1_{F(A)}"'] & F(A)
\end{tikzcd} ;$$

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

$$\begin{tikzcd}
& B \arrow[dr, "g"] \arrow[dd, mapsto] & \\
A \arrow[ur, "f"] \arrow[rr, "g \circ f"'] \arrow[dd, mapsto] & & C \arrow[dd, mapsto] \\
& F(B) \arrow[dr, "F(g)"] & \\
F(A) \arrow[ur, "F(f)"] \arrow[rr, "F(g) \circ F(f)"'] & & F(C)
\end{tikzcd} .$$

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 -> b`) and produce an arrow between the images of those two types (`F a -> F b`).

E.g. the list constructor:

```haskell
data [] a = [] | a : [a] -- "[]" is the type constructor for lists
fmap f [] = [] -- mapping f over an empty list does nothing
fmap f (x : xs) = (f x) : (fmap f xs)
-- to turn f into a list function, apply f to the head of the list,
-- 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

	Set	**Hask**	**POrd**	**Cat**
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**	$a = a$	$1_{\underline{\mathbf{C}}}$
Terminal obj.	$\{*\}$	`()`	upper bound	$\underline{\mathbf{1}}$
Product	$A \times B$	`(a,b)`	$\min(a, b)$	$\underline{\mathbf{C}} \times \underline{\mathbf{D}}$
Endofunctors	functors	type const.	OPTs	nat. trans.

Natural transformations

Natural Transformations

A *natural transformation* α is a mapping between two functors $F\colon \underline{\mathbf{C}} \to \underline{\mathbf{D}}$ and $G\colon \underline{\mathbf{C}} \to \underline{\mathbf{D}}$. 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:

$$\begin{tikzcd} F(A) \arrow[r, "F(f)"] \arrow[d, "\alpha_A"'] & F(B) \arrow[d, "\alpha_B"] \\ G(A) \arrow[r, "G(f)"'] & G(B) \end{tikzcd} .$$

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):

```
mu :: Functor f, Functor g => f a -> g a
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**:

```
nu :: Functor g => a -> g a
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:

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

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

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

Cartesian-Closed Categories

<u>**Set**</u> the singleton set, pairs, sets of functions

<u>**Hask**</u> `()`, `(a,b)`, `a -> b`

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

Applicatives

Applicative Functors

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

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

Applicative Functors

```haskell
class Functor f => Applicative f where
    -- | Lift a value.
    pure :: a -> f a
    -- | Sequential application.
    (<*>) :: f (a -> b) -> f a -> f b
```

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

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

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

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

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

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

Applicatives

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

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

$$\begin{tikzcd}
1 \arrow[r, "\phi"] \arrow[d, "\epsilon_{1_A}, \epsilon_v"'] & F1 \arrow[d, "F\epsilon_{1_A}, F\epsilon_v"] \\
A^A \times (FA) \arrow[r, "\phi, 1"] & F(A^A) \times (FA) \arrow[d, "="] \\
& (FA)^{FA} \times (FA) \arrow[d, "\lambda_{FA,FA}"] \\
& FA
\end{tikzcd}$$

Applicatives

Applicative Functors Example

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

Applicatives

Applicative Functors Example

```
instance Applicative [] where
    pure x = [ x ]
    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 $\underline{\mathbf{C}} \to \underline{\mathbf{D}}$
2. G is a functor from $\underline{\mathbf{D}} \to \underline{\mathbf{C}}$
3. ε is a natural transformation from $F \circ G \to 1_{\underline{\mathbf{C}}}$ (the *counit* of adjunction)
4. η is a natural transformation from $1_{\underline{\mathbf{D}}} \to G \circ F$ (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

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

Examples

$\underline{\mathbf{C}}$	$\underline{\mathbf{D}}$	F	G
$\underline{\mathbf{Set}}$	$\underline{\mathbf{Set}^2}$	Δ	$\sqcap$
$\underline{\mathbf{Set}^2}$	$\underline{\mathbf{Set}}$	$\sqcup$	Δ
$\underline{\mathbf{Set}}$	$\underline{\mathbf{Grp}}$	Free	Forgetful
$\underline{\mathbf{Set}}$	$\underline{\mathbf{Top}}$	Discrete	Forgetful
$\underline{\mathbf{Top}}$	$\underline{\mathbf{Set}}$	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.

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

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

M1
$$\begin{tikzcd} T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\ T^2 \arrow[r, "\mu"'] & T \end{tikzcd}$$

M2
$$\begin{tikzcd} T \arrow[r, "\eta T"] \arrow[d, "T\eta"'] \arrow[dr, equal] & T^2 \arrow[d, "\mu"] \\ T^2 \arrow[r, "\mu"'] & T \end{tikzcd}$$

Wikipedia, Monad_(category_theory)

Monads

Earlier definition of monad.

```haskell
class Functor m => Monad m where
  return      :: a -> m a
  join        :: m (m a) -> m a
```

King & Wadler 1993

This is how it is defined now.

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

GHC.Base base-4.9.1.0

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

Monad Laws

(omitting two that guarantee functorhood of f)

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

King & Wadler 1993

Other Connections

- 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.