Category Theory for Functional Programmers

Functors, Natural Transformations, Adjunctions

Nick Hu

University of Oxford

nick.hu@cs.ox.ac.uk

ABSTRACT

Category Theory is a very abstract theory of mathematical structure, which has strong links with functional programming. In this essay, I explore basic category theory constructions using insights from functional programming to provide examples and motivation. The categorical triad of functoriality, naturality, and universality (adjunction) is ubiquitous in mathematical spheres, and I aim to cover it in a brief yet elucidating manner, unburdened from complex heavy rigour present in almost any graduate text on category theory.

INTRODUCTION

Categories are the essence of composition, category theory yields a ubiquitous language interesting to logicians, philosophers, computer scientists, mathematicians, and physicists alike. In fact, Baez (2010) goes on to suggest that often different concepts in wildly different disciplines are actually merely different exposition for 'the same' phenomena.

For computer science especially, category theory 'gives a precise handle on important notions such as compositionality, abstraction, representation-independence, genericity and more' (Abramsky and Tzevelekos 2010).

More fundamental still, is that category theory attempts to address and challenge the age-old problem of what formal 'equality' means and what is actually desirable — notions of equality from set theory are fraying at the edges in modern mathematics, as seen in the new and fast-moving field of Homotopy Type Theory.

Certain categories, which will be touched on later, suffice to model computation and logic; a prime example of this is the relationship between the simply-typed λ-calculus, intuitionistic logic, and Cartesian Closed Categories, as stipulated by the Curry-Howard-Lambek correspondence.

Much of this essay is inspired by Bartosz Milewski's excellent book on Category Theory for Programmers (2017), and that is where the title comes from — a lot of this work can be regarded as my personal distillation and insights from reading it, presented as briefly as I could justify.

Throughout this essay, I will make assertions in **bold** — the reader may treat these assertions as exercises to justify.

GROUNDWORK

A category $\mathcal{C}$ is defined by a collection of *objects*, $\mathrm{ob}(\mathcal{C})$, and a collection of morphisms, $\hom(\mathcal{C})$, which are maps between objects. A morphism f has a *domain* $\mathrm{dom}(f)$ and a *codomain* $\mathrm{cod}(f)$, which are objects, and $f\colon X \to Y$ or equivalently $X \xrightarrow{f} Y$ is written to denote that f is a morphism with domain X and codomain Y.

Categories must also satisfy the following conditions:

- for each object X, there exists a morphism $\mathrm{id_X}\colon X \to X$;
- for any two morphisms $f\colon X \to Y$, $g\colon Y \to Z$, there exists a composite morphism $g \circ f\colon X \to Z$.

Composition must be associative, and identity morphisms must act as a unit to composition, as shown in the following commutative diagrams:

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle \mathrm{id_X}}\downarrow & \nearrow{\scriptstyle f} & \\ X & & \end{array} \qquad \begin{array}{ccc} X & \xrightarrow{f} & Y \\ & {\scriptstyle f}\searrow & \downarrow{\scriptstyle \mathrm{id_Y}} \\ & & Y \end{array}$$

In a category, identity morphisms are necessarily unique.

The category which we are primarily interested in is **Hask**, which has Haskell types as its objects and Haskell functions as its morphisms.

To see that this forms a category, recall the polymorphic function:

```
id :: a -> a
id x = x
```

then if any type `A` is an object of **Hask**, its identity morphism is given by the monomorphic specialisation of `id` to `id :: A -> A`.

Morphisms compose via ordinary Haskell function composition (post monomorphic specialisation):

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

For **Hask** to really be a category, 'fast and loose reasoning' will be used to conveniently pretend that Haskell functions are total, and ignore concerns arising from $\perp$ (Danielsson et al. 2006).

These functions are sufficient to establish Hask as a category.

In category theory, we generally don't care about what the objects are, but instead how they interact with each other (via morphisms).

Other examples of a category include:

- **Set**, with objects as sets and morphisms as set-functions between them;
- **Pos**, with objects as partially ordered sets and morphisms as monotone maps;
- the category of a specific preordered set $(X, \prec)$, with objects $x \in X$ and morphisms $x \to y \iff x \prec y$;[1]
- the category of a specific monoid $(M, \cdot, e)$, with a single object and morphisms $m \in M$. The identity morphism is given by e, and composition is given by $\cdot$. If instead we have a group, then all of the morphisms are *isomorphisms*;
- **Cat**, with objects as categories[2] and morphisms as *functors*;
- $\mathcal{C}^{\mathrm{op}}$, the *dual category* to $\mathcal{C}$ with the direction of its morphisms reversed — $f\colon X \to Y \in \hom(\mathcal{C}) \iff f\colon Y \to X \in \hom(\mathcal{C}^{\mathrm{op}})$. If $\mathcal{C} = \mathcal{C}^{\mathrm{op}}$ then $\mathcal{C}$ is *self-dual*;
- **Rel**, with objects as sets and relations as morphisms; **Rel** is self-dual because relations are symmetric whereas functions are not.

The category **Set** is very important, and new concepts will frequently be introduced via **Set**.

Isomorphism

In a category $\mathcal{C}$, if there is a morphism $f\colon X \to Y$, and a morphism $g\colon Y \to X$ such that the following diagram commutes:

$$\mathrm{id_X} \circlearrowright X \underset{g}{\overset{f}{\rightleftarrows}} Y \circlearrowleft \mathrm{id_Y}$$

then X and Y are *isomorphic*, and f and g are *isomorphisms*.

The problem of equality is a far-ranging one, but the usual definition of extensional equality (or even intensional equality) is often too strict. In category theory, isomorphism is commonly regarded as a sufficient notion of 'sameness'; often, category theorists come up with *universal constructions* establishing some property that is 'unique up to unique isomorphism' (canonical isomorphism).

An analogy from programming is the abstraction barrier: in general, we don't really care how an interface is implemented given that it fully satisfies the specification. A stack can be implemented using a linked list, a heap, or an array, and correct implementations will behave indistinguishably in the abstract — in the concrete, implementations will behave differently, but beyond the concern of rationally reasoning about correctness. Another way of viewing the isomorphisms is as translations between different representations of the same data — a linked list can be traversed and the contents dumped into an array, and similarly a linked list can be built from the contents of an array.

[1] Identity morphisms exist due to the reflexivity of the ordering relation, and composite morphisms from transitivity.

[2] Which are *small* — $\mathcal{C}$ is small when its collections of objects and morphisms are set-sized. **Cat** itself is large, but otherwise we will only consider small categories.

LIMITS AND COLIMITS

In a category $\mathcal{C}$, a terminal object 1 is an object where every other object X in $\mathcal{C}$ has a unique morphism ι_X targeting it (i.e. $\forall X \in \mathrm{ob}(\mathcal{C}) \cdot \exists! \, \iota_X \cdot X \xrightarrow{\iota_X} 1$). *Dually*, an initial object 0 is an object where every other object X in $\mathcal{C}$ has a unique morphism $!_X$ originating from it. **Initial and terminal objects are unique up to unique isomorphism.**

In **Set**, $\emptyset$ is initial, and the singleton set $\{*\}$ is terminal. To see why, consider the number of functions $f_X\colon \emptyset \to X$; there is only one such function: the empty function. For any set X, there is only one function $f_X\colon X \to \{*\}$: the constant function. This can be checked by considering that the number of functions from two sets X and Y is $|Y|^{|X|}$ — for the initial object we have $|X|^0 = 1$, and for the terminal object we have $1^{|X|} = 1$.

In **Hask**, the initial object is a type `Void` which has no constructors. The unique morphism associated to it is:

```
absurd :: Void -> a
```

As `Void` has no inhabitants, there is no meaningful definition for this function, just as the empty set-function has no meaningful map. The unit type, `()`, is terminal, and the unique morphism associated to it is:

```
unit :: a -> ()
unit _ = ()
```

or equivalently `unit = const ()`.

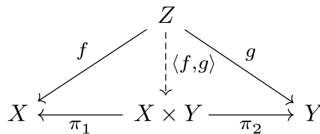
A useful, but naïve, intuition for the relationship between **Hask** and **Set** is to treat each type as a set with cardinality equal to the number of inhabitants of that type. `Void` has no inhabitants, and thus corresponds to the empty set. Similarly, unit has a single inhabitant `()` — it is unimportant what that is, because all singleton sets are trivially canonically isomorphic.

A product of two sets X and Y is traditionally given by $\{(x, y) \mid x \in X, y \in Y\}$ — the Cartesian product construction. The product can alternatively be defined to be $\{(y, x) | x \in X, y \in Y\}$, or even $\{(x, \emptyset, y) \mid x \in X, y \in Y\}$. As long as our theorems about products of sets are aware of which representation is chosen, they still hold, and indeed these representations are isomorphic to each other.

Categorically, this *implementation detail* can be abstracted away, and instead a product is defined in terms of an abstract interface: a product of two objects X and Y is an object $X \times Y$ along with projection maps $\pi_1\colon X \times Y \to X$ and $\pi_2\colon X \times Y \to Y$.

However, this definition is insufficient — the Cartesian set product $X \times Y \times \{0, 1\}$ as a categorical product of objects X and Y in **Set**, with projection maps $\pi_1(x, y, 0) \mapsto x$, $\pi_1(x, y, 1) \mapsto x$ (and similarly for π_2) satisfies this definition, but this 'product' contains an extra bit of information. We want our notions of product to be isomorphic to each other, but there is no canonical mapping between the Cartesian set products $X \times Y \to X \times Y \times \{0, 1\}$. But the reverse map exists canonically by simply forgetting the extra bit of information; in general, this corresponds to the existence of a morphism in **Set** from 'products' with too much information, to the traditional Cartesian product.

Hence, for a satisfying definition of product, we require the product object $X \times Y$ be the 'most extreme' object in some sense (hence limit); given a third object Z and morphisms $f : Z \to X$ and $g : Z \to Y$, we require there to exist a unique morphism $\langle f, g \rangle : Z \to X \times Y$ making the following diagram commute:[3]



Z corresponds to our object with necessary projection maps; a 'product' which might have too much information — if it truly is a product, then by definition there would exist a unique morphism $X \times Y \to Z$ **establishing a unique isomorphism between** Z **and** $X \times Y$. Another way of thinking about this is that f and g factor through $\langle f, g \rangle$.

In doing this, a more universal notion of product independent from set membership $\in$ is given, and it doesn't matter which one is used as products are *canonically isomorphic* (unique up to unique isomorphism).

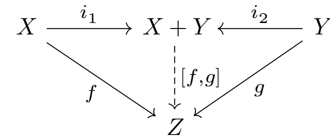
Cartesian set products are an instance of categorical product in **Set**, with the obvious projection maps. In **Hask**, the product of types `A` and `B` is the tuple type `(A, B)` with projection maps `fst :: (a, b) -> a` and `snd :: (a, b) -> b`.

Some other examples of products:

- in the category of a specific partially ordered set, the product of two objects is its meet/infimum/greatest lower bound;
- in the category of logic predicates, the product of two objects is its logical conjunction.

Dually, there is a notion of coproduct: a coproduct of two objects X and Y is an object $X + Y$ along with injection maps $i_1 : X \to X + Y$ and $i_2 : Y \to X + Y$ such that the following diagram commutes:

[3] A dashed arrow to is used to indicate uniqueness.



In **Set**, the coproduct of X and Y is given by the disjoint union $X \uplus Y$. In **Hask**, coproducts of types are sum types tagged by constructors; the canonical example:

```
data Either a b = Left a | Right b
```

is the coproduct of polymorphic types `a` and `b`, with constructors `Left :: a -> Sum a b` and `Right :: b -> Sum a b` providing the injection maps.

The existence of binary (co) products and a (initial) terminal object implies the existence of arbitrary finite (co) products.

Algebra of types

These types are called 'algebraic data types' because **Hask** is a *monoidal* category with respect to product, in the sense that `a` is isomorphic to `(a, ())` (the product of `a` and the terminal object `()`). As witness to the isomorphism:

```
rho :: (a, ()) -> a
rho = fst

rho_inv :: a -> (a, ())
rho_inv x = (x, ())
```

Hask is also monoidal with respect to coproduct (sum), with `a` being isomorphic to `Either a Void` (the coproduct of `a` and the initial object `Void`). As `Void` is uninhabited, only terms with the `Left` constructor can be created, and so the isomorphism is just to deconstruct or apply it.

Using the categorical syntax, we have established:

if a is an object in **Hask**, then,

- $a \times 1 \simeq a$ — monoidal with respect to product, terminal object as identity;
- $a + 0 \simeq a$ — monoidal with respect to coproduct, initial object as identity;
- $a \times 0 \simeq 0$ — `(a, Void)` also has no inhabitants.

Stretching this analogy, it can be shown that products distribute over sums: $a \times (b + c) \simeq a \times b + a \times c$, or in Haskell by showing that `(a, Either b c)` is isomorphic to `Either (a, b) (a, c)`. Such an isomorphism is witnessed by the following:

```
distribute :: (a, Either b c) -> Either (a, b) (a, c)
distribute (x, Left y) = Left (x, y)
distribute (x, Right z) = Right (x, z)

undistribute :: Either (a, b) (a, c) -> (a, Either b c)
undistribute (Left (x, y)) = (x, Left y)
undistribute (Right (x, z)) = (x, Right z)
```

This means that **Hask** has a semiring structure, and the naturals are further interpreted as the number of inhabitants of a type. For instance, the type `data Bool = True | False` can be regarded as syntax sugar for `data Bool = Either (True (), False ())` where `True` and `False` are tags distinguishing distinct singleton types; rewriting this categorically, we get $\text{Bool} = 1 + 1 = 2$.

A more interesting example is the cons list:

```
data List a = Nil | Cons a (List a)
-- desugaring Nil and Cons into tags, uncurrying Cons
data List a = Either (Nil (), Cons (a, (List a)))
```

Categorically, writing x for `List a`:

$$
\begin{aligned}
x &= 1 + a \times x \\
&= 1 + a \times (1 + a \times x) && = 1 + a + a^2 \times x \\
&= 1 + a \times (1 + a \times (1 + a \times x)) && = 1 + a + a^2 + a^3 \times x \\
&= \ldots
\end{aligned}
$$

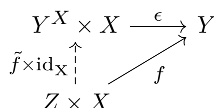
which precisely describes a list: it is a singleton (empty list), or an `a` in a singleton list, or a 2-tuple `(a, a)` corresponding to the 2-length list, etc.

EXPONENTIALS

In **Set**, the collection of morphisms from two sets X and Y itself forms a set, which is called a *hom-set*, denoted by $\mathbf{Set}(X, Y)$ or Y^X. As the objects of **Set** are all sets, Y^X is also an object of **Set**, so it is described as *internal*. **Hask** also has internal hom-sets in the form of function types.

To describe this abstractly in the language of category theory, this notion needs to be related to objects and morphisms. Let X, Y, and Y^X be objects in a category $\mathcal{C}$ with binary products such that there exists an *evaluation morphism* $\epsilon\colon Y^X \times X \to Y$. Y^X is an *exponential* object if for any other Z and $f\colon Z \times X \to Y$, there is a unique morphism $\tilde{f}\colon Z \to Y^X$ that factors f through ϵ: $f = \epsilon \circ (\tilde{f} \times \mathrm{id_X})$.[4]

Equivalently, the following diagram commutes:



Looking at $\tilde{f}$, we see that it takes an object and gives us an exponential object, corresponding to a function type in **Hask**; this allows us to interpret $\tilde{f}$ as the curried version of f, and our factorisation recovering f from $\tilde{f}$ gives uncurrying.

The notation for the exponential objects is no coincidence, and the normal algebraic exponentiation identities just fall out (with equality swapped for isomorphism):

- $X^0 \simeq 1$,
- $1^X \simeq 1$,
- $X^1 \simeq X$,
- $X^{Y+Z} \simeq X^Y \times X^Z$,
- $(X^Y)^Z \simeq X^{Y \times Z}$,
- $(X \times Y)^Z \simeq X^Z \times Y^Z$.

These identities can be interpreted in Hask by polymorphic invertible functions which serve as witness to the isomorphism, in the same fashion as the previous section on algebraic data types.

Curry-Howard-Lambek

If a category has a terminal object, binary products, and exponential objects, then it is Cartesian Closed. A Cartesian Closed Category (CCC) provides sufficient structure to model the simply-typed λ-calculus, which is the core of functional programming. It also models intuitionistic logic; consult the following translation table (using **Hask** as an approximation for simply-typed λ-calculus):

Table 1: Curry-Howard-Lambek correspondence

CCC $\mathcal{C}$	Intuitionistic Logic	**Hask**
0	$\bot$	`Void`
1	$\top$	`()`
$X \times Y$	$x \wedge y$	`(a, b)`
$X + Y$	$x \vee y$	`Either a b`
Y^X	$x \to y$	`a -> b`

Now, our evaluation morphism $\epsilon\colon Y^X \times X \to Y$ is interpreted in **Hask** as a function `eval :: ((a -> b), a) -> a` which corresponds to function application, and in logic as $((x \to y) \wedge x) \to y$ — modus ponens. The Curry-Howard part of the correspondence goes to say that a proof of a theorem is equivalent to a typed expression with a type corresponding to the formula. To prove such a theorem, it suffices to show that `eval`'s type is inhabited:

```
eval :: ((a -> b), a) -> b
eval (f, x) = f x
```

and therefore modus ponens is a valid theorem in intuitionistic logic.

In the same vein, **the `absurd` function is an interpretation of ex falso quodlibet ($\bot \to x$).**

In fact, the Curry-Howard-Lambek correspondence states that simply-typed λ-calculus, intuitionistic logic, and CCCs are isomorphic.

FUNCTORIALITY

A *functor* is a structure-preserving map between categories, which preserves composition and identity morphisms; given

[4] The product of morphisms will be justified later, in the discussion of the *product functor*, but for now $\tilde{f} \times \mathrm{id_X} = \langle \tilde{f}, \mathrm{id_X} \rangle$.

categories $\mathcal{C}$ and $\mathcal{D}$, and a functor $F\colon \mathcal{C} \to \mathcal{D}$ between them, we have:

- $F(f\colon X \to Y) = F(f)\colon F(X) \to F(Y)$;
- for every object X in $\mathcal{C}$, $F(\mathrm{id_X}) = \mathrm{id_{F(X)}}$ in $\mathcal{D}$;
- $F(g \circ_{\mathcal{C}} f) = F(g) \circ_{\mathcal{D}} F(f)$ whenever $g \circ f$ is defined in $\mathcal{C}$.[5]

Some examples of functors:

- the *forgetful* functor from **Mon** to **Set**, which maps each monoid to its underlying set (monoid homomorphisms are weakened to functions);
- the selection functor mapping the trivial singleton category to any non-empty category $\mathcal{C}$;
- the constant functor $\Delta_X\colon \mathcal{C} \to \mathcal{D}$ mapping every object of $\mathcal{C}$ to X in $\mathcal{D}$, and every morphism of $\mathcal{C}$ to $\mathrm{id_X}$,
- the diagonal functor $\Delta\colon \mathcal{C} \to \mathcal{C} \times \mathcal{C}$ mapping objects X in $\mathcal{C}$ to $X \times X$ in the product category of $\mathcal{C}$ with itself, $\mathcal{C} \times \mathcal{C}$,
- the product functor $- \times -\colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, assuming $\mathcal{C}$ has binary products, mapping every object (X, Y) in $\mathcal{C} \times \mathcal{C}$ to $X \times Y$ in $\mathcal{C}$, and every morphism $(f, g)\colon (X, Y) \to (X', Y')$ to $f \times g\colon X \times Y \to X' \times Y'$. $f \times g$ is given by $\langle f \circ \pi_1, g \circ \pi_2 \rangle$.

In **Hask**, functors[6] are given by the `Functor` typeclass:

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

The type constructor gives the object mapping, and `fmap` gives the morphism mapping.

One intuition for Haskell functors is containers; for example the Haskell list `[]` is a functor with `fmap` given by the standard `map` function on lists. The option type `Maybe` is also *functorial*:

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

and so is `Either a`.

Contravariance

The functors presented so far are *covariant functors*. A functor $F\colon \mathcal{C} \to \mathcal{D}$ is *contravariant* if it maps morphisms $f\colon X \to Y$ in $\mathcal{C}$ to $F(f)\colon F(Y) \to F(X)$ in D. This is the same as a covariant functor from $\mathcal{C}^{\mathrm{op}}$ to $\mathcal{D}$. In **Hask**, this corresponds to a different typeclass:

```
class Contravariant f where
  contramap :: (b -> a) -> (f a -> f b)
```

Hom-functors

For objects A and B in a category $\mathcal{C}$, the *covariant hom-functor* $\mathcal{C}(A, -) : \mathcal{C} \to \mathbf{Set}$ is defined. $\mathcal{C}(A, -)$ maps each object X in $\mathcal{C}$ to the hom-set $\mathcal{C}(A, X)$, and each morphism $f\colon X \to Y$ to $\mathcal{C}(A, f)\colon \mathcal{C}(A, X) \to \mathcal{C}(A, Y)$ with mapping $g \mapsto f \circ g$ for each function g in the hom-set $\mathcal{C}(A, X)$.

In **Hask**, this is known as the `Reader` functor:

```
type Reader a b = a -> b
instance Functor (Reader a) where
  fmap = (.)
```

Similarly, the *contravariant hom-functor* $\mathcal{C}(-, B) : \mathcal{C} \to \mathbf{Set}$ is also defined, but the morphism map is instead postcomposition:

```
type Coreader a b = b -> a
instance Contravariant (Coreader a) where
  contramap = flip (.)
```

Functors which are *naturally isomorphic* to a hom-functor are called *representable*.

In **Hask**, this can be encoded as follows:

```
class Representable f where
  type Rep f :: Type
  tabulate :: (Rep f -> x) -> f x
  index :: f x -> (Rep f -> x)
```

We can think of the representable functor[7] as a statically-sized data structure which can be encoded as data in memory `f x`, or isomorphically as a function from the indexing type to the data type `Rep f -> x`, along with methods to transform between the two encodings; i.e. the following diagram commutes:

$$\texttt{f x} \underset{\texttt{tabulate}}{\overset{\texttt{index}}{\rightleftarrows}} \texttt{Rep f -> x}$$

A good example of this is a statically-sized vector of size n — the `Rep` type is the finitely bounded naturals $[0, n)$, which give the indices of the vector. Then the `Rep f -> x` encoding is interpreted as a sparse vector, and `f x` as a dense one. The representable functor formalises the idea that these two encodings are 'the same data' (isomorphism), and gives a method to convert between the two, but in a computational context one may be preferable over the other.

The pair type `(a, b)` is `Representable` in the same fashion.

NATURALITY

It is only natural to wonder if there is a notion of mapping between functors; let F and G be two functors from $\mathcal{C}$ to $\mathcal{D}$, then a *natural transformation* α from F and G, denoted $\alpha\colon F \Rightarrow G$ is a family of morphisms $\alpha_X\colon F(X) \to G(X)$ indexed by objects X in $\mathcal{C}$. Each particular morphism is called the component of α at X. The functors of the natural transformation must map morphisms identically, and the natural transformation must make the following diagram

[5] Sometimes $\circ_{\mathcal{C}}$ will be used to denote the composition in category $\mathcal{C}$ where ambiguity may arise.

[6] More specifically, endofunctors — functors from a category back into itself; in this case, **Hask**.

[7] Also called a Naperian functor, where `Rep f` is like a logarithm of `f` (Gibbons 2017).

(*naturality square*) commute for each morphism $f\colon X \to Y$ in $\mathcal{C}$:

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \alpha_Y} \\ G(X) & \xrightarrow[G(f)]{} & G(Y) \end{array}$$

If every component of a natural transformation is an isomorphism, then it is a *natural isomorphism*, and this is our best notion for two functors being 'the same'.

Thinking about the components, natural transformation can be considered as a mapping of objects to morphisms, such that the resultant morphism has a commuting naturality square. A lot of commuting diagrams that are required in order to fulfil a particular property is really a naturality square in disguise, given the correct choice of functors.

In **Hask**, a natural transformation looks like:

```
alpha :: (Functor f, Functor g) => F a -> G a
```

The *genericity* provided by the parametric polymorphic type `a` suffices to define the whole natural transformation, and this is a good intuition to understand them. In fact, due to *parametricity*, in **Hask** the naturality condition:

```
(fmap f :: G a -> G b) . (alpha :: F a -> G a)
= (alpha :: F b -> G b) . (fmap f :: F a -> F b)
```

arises automatically for a function `f :: a -> b`, as a result of *free theorems* (Wadler 1989). Furthermore, all the types can be inferred.

If a functor is a generalisation of a container, then a natural transformation is the generalisation of repacking data from different containers. Naturality is the independence between transforming the data with `fmap` and repacking — these two operations commute.

Another interesting property is that because all standard algebraic data types are functorial, polymorphic functions between them are actually natural transformations (in the opposite category to **Hask**, as function types are contravariant).

Functor categories

A *functor category* of $\mathcal{C}$ and $\mathcal{D}$, denoted by $[\mathcal{C}, \mathcal{D}]$, is a category in which the objects are functors $\mathcal{C} \to \mathcal{D}$, and morphisms are natural transformations between them. Compositions of morphisms are given by the *vertical composition* of natural transformations, as shown in the following commuting diagram, given another functor $H\colon \mathcal{C} \to \mathcal{D}$ and natural transformation $\beta\colon G \Rightarrow H$:

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \alpha_Y} \\ G(X) & \xrightarrow{G(f)} & G(Y) \\ {\scriptstyle \beta_X}\downarrow & & \downarrow{\scriptstyle \beta_Y} \\ H(X) & \xrightarrow{H(f)} & H(Y) \end{array}$$

Cat is actually Cartesian closed, with the functor category $[\mathcal{C}, \mathcal{D}]$ as the exponential object $\mathcal{D}^{\mathcal{C}}$.

ADJUNCTION

Two categories $\mathcal{C}$ and $\mathcal{D}$ are *equivalent* if there are a pair of functors $L\colon \mathcal{C} \to \mathcal{D}$ and $R\colon \mathcal{D} \to \mathcal{C}$ such that the composition in either direction is naturally isomorphic to the identity functor of the respective category.

Adjunction is a slightly weaker notion than this, and it appears everywhere in mathematics and computer science. More specifically, it is weaker in the sense that the composite functor need not be naturally isomorphic to the identity functor, but these directions of natural transformations should exist:

$$\eta\colon I_{\mathcal{C}} \Rightarrow R \circ L,$$
$$\epsilon\colon L \circ R \Rightarrow I_{\mathcal{D}}.$$

η and ϵ are called the *unit* and *counit* of the adjunction respectively, and we use the notation $L \dashv R$ to denote that L is the left-adjoint functor to R, and that R is the right-adjoint functor to L.

Adjunctions are subject to the following triangle identities (in the functor category):

$$\begin{array}{ccc} L & \xrightarrow{L\circ\eta} & L \circ R \circ L \\ & \searrow\!\!\!= & \downarrow{\scriptstyle \epsilon\circ L} \\ & & L \end{array} \qquad \begin{array}{ccc} R \circ L \circ R & \xleftarrow{\eta\circ R} & R \\ {\scriptstyle R\circ\epsilon}\downarrow & =\!\!\!\swarrow & \\ R & & \end{array}$$

where the double line is the identity natural transformation.

In **Hask**, the unit is called:

```
return :: a -> m a
```

and the counit is:

```
extract :: m a -> a
```

which are exactly the lifting/unlifting operations of the `Monad` and `Comonad` typeclasses — fundamentally, every monad and comonad arises from adjunction.

An alternative but equivalent definition of adjunction can be given in terms of hom-functors; given X, an object of $\mathcal{C}$, and Y, an object of $\mathcal{D}$, an adjunction consists of functors L and R along with a natural isomorphism

$$\mathcal{C}(X, R(Y)) \underset{\phi_{X,Y}}{\overset{\psi_{X,Y}}{\rightleftarrows}} \mathcal{D}(L(X), Y)$$

From this definition, unit can be derived:

$$\begin{aligned} \forall Y \in \text{ob}\,(\mathcal{D}) \cdot \mathcal{D}(L(X), Y) &\simeq \mathcal{C}(X, R(Y)) \\ \implies \mathcal{D}(L(X), L(X)) &\simeq \mathcal{C}(X, (R \circ L)(X)) \\ &= \mathcal{C}(I_X(X), (R \circ L)(X)) \end{aligned}$$

which precisely gives the natural transformation η at each component X by taking id_X along $\phi_{X,Y}$.

The counit can be derived similarly, and the hom-functor definition can be derived from unit and counit.

Many universal constructions, including product and exponential, arise from adjunction.

A *free construction*, given functorially, is a nice notion of 'canonical' ways to create structure; for instance, given any set X, a free monoid consisting of elements of that set under concatenation can be generated, introducing a special null-character ϵ as an identity (very similar to Kleene star construction). Free functors are left adjoint to a forgetful functor, which 'forgets' some structure. The free monoid functor from **Set** to **Mon** is left adjoint to the forgetful functor from **Mon** to **Set** which maps each monoid to its underlying set.

Following this, one intuition for adjunctions is as a way to create some sort of 'canonical approximation'.

Some more examples of free/forgetful adjunctions:

- the category of integers $\mathbb{Z}$ with morphisms given by the ordering $\leq$ has a injection functor $i\colon \mathbb{Z} \to \mathbb{R}$ – the floor and ceiling functors back into $\mathbb{Z}$ are right and left adjoints to i;
- the category **Top**, of topological spaces and continuous maps, has a forgetful functor into **Set** which takes each topological space to its underlying set — the left adjoint takes a set to its 'finest' topology: the discrete topology whereby every element is a singleton open set; the right adjoint takes a set to its 'coarsest' topology, the trivial topology whereby only the whole set is open.

FURTHER TOPICS

- **Cat** as a 2-category, and higher category theory.
- More limit constructions: equalisers, pullbacks.
- Limits and colimits in terms of universal *cone/cocones*.
- Properties of functors: *full*, *faithful*, injective on objects and essentially surjective. Notions of embedding and equivalence described only with functors.
- Yoneda lemma — all **Set**-valued functors arise via natural transformation of hom-functors.
- Universal constructions via adjunction.
- Right adjoint functors preserve limits, and left adjoint functors preserve colimits.
- Monads, Kleisli categories, and Eilenberg-Moore algebras.

REFERENCES

Abramsky, Samson, and Nikos Tzevelekos. 2010. "Introduction to Categories and Categorical Logic." In *New Structures for Physics*, 3–94. Springer.

Baez, John, and Mike Stay. 2010. "Physics, Topology, Logic and Computation: A Rosetta Stone." In *New Structures for Physics*, 95–172. Springer.

Danielsson, Nils Anders, John Hughes, Patrik Jansson, and Jeremy Gibbons. 2006. "Fast and Loose Reasoning Is Morally Correct." *SIGPLAN Not.* 41 (1). New York, NY, USA: ACM: 206–17. doi:10.1145/1111320.1111056.

Gibbons, Jeremy. 2017. "APLicative Programming with Naperian Functors." In *European Symposium on Programming*, edited by Hongseok Yang, 10201:568–83. LNCS. doi:10.1007/978-3-662-54434-1_21.

Milewski, Bartosz. 2017. "Category Theory for Programmers." https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/.

Wadler, Philip. 1989. "Theorems for Free!" In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, 347–59. FPCA '89. New York, NY, USA: ACM. doi:10.1145/99370.99404.