Category theory was initially introduced in the 1940s as a way to define naturality. Before looking into that, however, we define more examples of functors.

0.1 More functors

0.1.1 Example 1: an arbitrary functor

We can define functor F : **Set** $\to$ **Set** by the following mappings:

- objects: $X \mapsto X^2 + 7$, where $7 = \{0, 1, 2, 3, 4, 5, 6\}$ is the set of seven objects.
- morphisms: $g = Ff$ in $\begin{array}{ccc} X & \xrightarrow{F} & X^2+7 \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ Y & & Y^2+7 \end{array}$ e.g. $\begin{array}{ccc} Towns & \xrightarrow{F} & Towns^2+7 \\ {\scriptstyle population}\downarrow & & \downarrow{\scriptstyle F\ population} \\ \mathbb{N} & & \mathbb{N}^2+7 \end{array}$

Given the morphism $f \in$ **Set** we can define the mapping as follows:

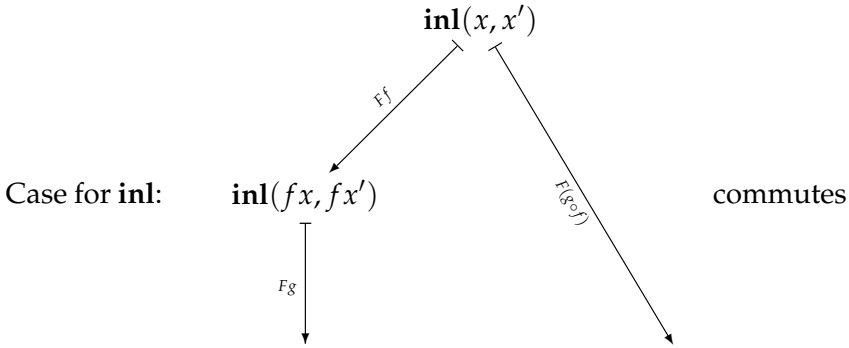
$$F\,f = \begin{array}{c} X^2+7 \\ \downarrow{\scriptstyle F\,f} \\ Y^2+7 \end{array} = \begin{array}{c} \mathbf{inl}(x, x') \\ \downarrow{\scriptstyle Ff} \\ \mathbf{inl}(f\ x, f\ x') \end{array} + \begin{array}{c} \mathbf{inr}\ j \\ \downarrow{\scriptstyle Ff} \\ \mathbf{inl}\ j \end{array}$$

Hence, composition is as follows:

Given $\begin{array}{c} X \\ \downarrow{\scriptstyle f} \\ Y \\ \downarrow{\scriptstyle g} \\ Z \end{array}$ we know $\begin{array}{ccc} & & X^2+7 \\ & \swarrow{\scriptstyle Ff} & \downarrow{\scriptstyle F(f;g)} \\ Y^2+7 & & \\ & \searrow{\scriptstyle Fg} & \\ & & Z^2+7 \end{array}$ must commute. i.e. $(Ff);(Fg) = F(f;g)$

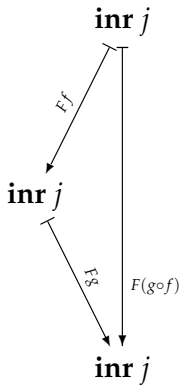
To check composition works, we check the each sum case (**inl** and **inr**):

- Case for **inl**:



commutes because $g(fx) = (g \circ f)\ x$

- Case for **inr**:



commutes

0.1.2 Example 2: Powerset

$F : X \mapsto \mathcal{P}_X$ (powerset of X)

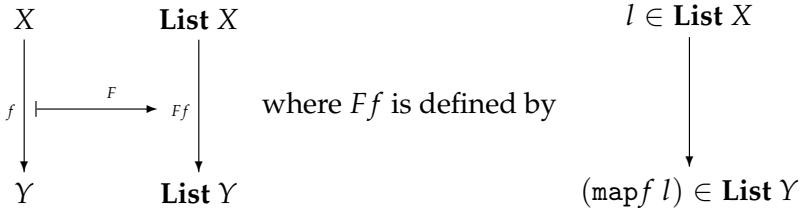
Given $\begin{array}{c} X \\ \downarrow f \\ Y \end{array}$ the mapped morphism would be $\begin{array}{c} \mathcal{P}_X \\ \downarrow Ff \\ \mathcal{P}_Y \end{array}$ defined by $\begin{array}{c} u \subseteq X \\ \downarrow Ff \\ \{f\ x \mid x \in u\} \end{array}$

Objects in the powerset are subsets of X. Thus, u is any subset of X and is mapped to any subset of Y.

0.1.3 Example 3: List

F : **Set** $\rightarrow$ **Set**

- Objects: $X \overset{F}{\longmapsto}$ **List** X
- Morphisms:



where Ff is defined by

0.1.4 Example 4: Opposite to Powerset

$F : \mathbf{Set}^{op} \to \mathbf{Set}$

- Objects: $X \overset{F}{\longmapsto} \mathcal{P}X$
- Morphisms: $\begin{array}{ccc} X & & \mathcal{P}X \\ {\scriptstyle f}\downarrow & \overset{F}{\longmapsto} & \downarrow{\scriptstyle Ff} \\ Y & & \mathcal{P}_Y \end{array}$ where Ff is defined by $\begin{array}{c} \{x \in X \mid f\,x \in u\} \\ \uparrow \\ u \in \mathcal{P}_Y \end{array}$

Note that F is a **contravariant** functor. This definition will be coming up soon.

0.1.5 Example 5 : Functor $C^2 \to C$

$F : \mathbf{Set}^2 \to \mathbf{Set}$

- Objects: $(X, X') \overset{F}{\longmapsto} X + X'$
- Morphisms: $\begin{array}{ccc} (X, X') & & X + X' \\ {\scriptstyle f}\downarrow\,\downarrow{\scriptstyle f'} & \overset{F}{\longmapsto} & \downarrow{\scriptstyle F(f,f')} \\ (Y, Y') & & Y + Y' \end{array}$ where $F(f, f')$ is defined by $\begin{array}{ccc} \mathbf{inl}\ x \in X & & \mathbf{inr}\ x' \in X' \\ \downarrow & + & \downarrow \\ \mathbf{inl}\ fx & & \mathbf{inr}\ fx' \end{array}$

In general, if C is a category with binary coproducts, we can define a functor $+ : C^2 \to C$.

0.2 Natural transformations

In the diagram $\begin{array}{ccc} C & \xrightarrow{F} & D \\ & \Downarrow{\scriptstyle \alpha} & \\ C & \xrightarrow[G]{} & D \end{array}$ α is a natural transformation from F to G.

$\forall x \in C\,.\,\alpha : F \Rightarrow G$ provides a D morphism $FX \xrightarrow{\alpha_X} GX$ such that for any C morphism $X \xrightarrow{f} Y$ the following square commutes:

$$\begin{array}{ccc} FX & \xrightarrow{Ff} & FY \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \alpha_Y} \\ GX & \xrightarrow[Gf]{} & GY \end{array}$$

0.2.1 Example

Given the **List** functor defined before (the one that maps lists), we can show that:

$reverse_X$: **List** $X \to$ **List** X is a natural transformation.

To do this, we must show commutativity.

Given $\begin{array}{c} \textbf{List } X \\ \downarrow{\scriptstyle rev_X} \\ \textbf{List } X \end{array}$ we must show that $\begin{array}{ccc} \textbf{List } X & \xrightarrow{Listf} & \textbf{List } Y \\ \downarrow{\scriptstyle rev_X} & & \downarrow{\scriptstyle rev_Y} \\ \textbf{List } X & \xrightarrow[Listf]{} & \textbf{List } Y \end{array}$ commutes.

*note: $List\ f = map\ f$, which is how the *List* functor would be implemented in some programming languages (like Haskell).

The following diagram depicts this precise property:

$$\begin{array}{ccc} [x_0, \ldots, x_{n-1}] & \xmapsto{map\ f} & [fx_0, \ldots, fx_{n-1}] \\ \downarrow{\scriptstyle rev_X} & & \downarrow{\scriptstyle rev_Y} \\ [x_{n-1}, \ldots, x_0] & \xrightarrow[map\ f]{} & [fx_{n-1}, \ldots, fx_0] \end{array}$$

0.2.2 Non-example

The transformation (rev') which reverses a list given no duplicates is not natural. We can prove this by defining a case where the diagram does not commute. This consists of a function and a list onto which we map said function.

Given set $S = a, b, c$, we define function $f : S \to S$ to be:

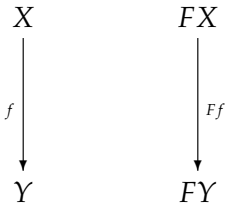
$f(a) = a$
$f(b) = a$
$f(c) = c$

$$\begin{array}{ccccc} [a,b,c] & \xmapsto{map\ f} & [a,a,c] & \xmapsto{rev'_Y} & [a,a,c] \\ \downarrow{\scriptstyle rev'_X} & & & & \neq \\ [c,b,a] & & \xrightarrow[map\ f]{} & & [c,a,a] \end{array}$$

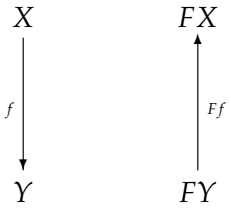
Does not commute because $[a, a, c] \neq [c, a, a]$.

0.3 Covariance and contravariance

A **covariant** functor maps to morphisms going in the same direction of the original mapped morphism.



A **contravariant** functor maps to morphisms going in the opposite direction of the original mapped morphism.



In other words, covariance preserves the ordering of types while contravariance reverses this order.

Sometimes, we refer to structures with only covariant (positive) or contravariant (negative) occurences as **functorial**, which means there is an obvious way to extend the structure to make a functor.

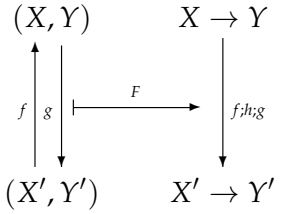
Informally, **non-functorial** structures have both positive and negative occurences. This means there is no obvious way of extending it to become a functor. However, we don't know otherwise, i.e. there may be a way.

0.3.1 Example: arrow (function types)

The left side of an arrow is contravariant. This can be explained in category theory as the following functor:

$F : \mathbf{Set}^{op} \times \mathbf{Set} \to \mathbf{Set}$

- Objects: $(X, Y) \overset{F}{\longmapsto} X \to Y$
- Morphisms:



where $h : X \to Y$

Note that having $\mathbf{Set}^{op}$ is the only way the function type can be defined since there would be no way to create the functor otherwise. This is analogous to $C^{op} \times C \to \mathbf{Set}$ seen before.

The implications of this is that in functions with subtypes (generally speaking, Object Oriented languages), we need to deal with left side contravariance of arrows.

e.g. Given `String` $\leq$ `Object`

- `String -> String` can be safely used wherever you expect `String -> Object`
- `Object -> String` can be safely used wherever you expect `String -> String`

As `fold` is a higher-order function, and takes one as an argument, it will have both possitive and negative occurences. Hence, it has no simple functor.

0.3.2 Example: more on programming languages

Programming languages that support subtyping must deal with variance. Variance refers to how subtyping between complex types relates to subtyping between their components.

For example, in C#:

- `Enumerable<Cat>` is a subtype of `Enumerable<Animal>`, i.e. `Enumerable<T>` is **covariant** on `T`.
- `Action<Animal>` is a substype of `Action<Cat>`, i.e. `Action<T>` is **contravariant** on `T`

This problem can be observed in Java and C# arrays. To avoid errors when creating an `Object` array, should we treat this as:

- **covariant**: `String[]` is a `Object[]`

  However, you should always be able to put `Integer` in `Object[]`, so treating `String[]` as a `Object[]` results in errors when writing.
- **contravariant**: `String[]` is a `Object[]`

  However, you should expect to only read `String` from `String[]`, so treating `Object[]` as a `String[]` results in errors when reading since `Object[]` could contain `Integer`.
- **invariant**: `String[]` and `Object[]` are different

  However, making arrays invariant rules out useful polymorphic programs.

To make arrays polymorphic, the designers of Java and C# made all arrays covariant. This, however causes runtime errors when writing into arrays, which the designers deal with using exceptions. e.g. `ArrayStoreException` in Java. This is known as "ad-hoc

polymorphism". i.e. Polymorphism where we get different behaviour depending on the type.

This was until the addition of generics, which provided a method of creating polymorphic functions without relying on covariance, which allows type checking for errors at compile time rather than runtime. This is known as "parametric polymorphism". i.e. Polymorphism where we get the same behaviour regardless the type–as in functional programs.

0.4 Exercises

0.4.1 A transformation that doubles entries

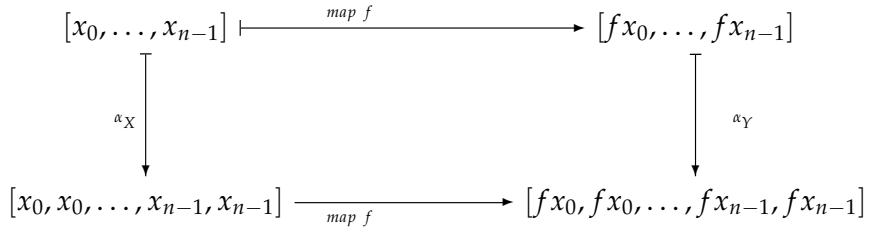
Given a transformation that doubles entries on a list:

$\alpha_x : \text{List } \mathbf{X} \to \text{List } \mathbf{X} = [x_0, \ldots, x_{n-1}] \mapsto [x_0, x_0, \ldots, x_{n-1}, x_{n-1}]$

Prove it is natural.

Given $f : X \to Y$

We can show α is natural through the following square commutes:



As this is natural, behaviour does not depend on the type of the argument. Thus, this is parametrically polymorphic on $\mathbf{X}$.

0.4.2 A transformation that only reverses lists of natural numbers

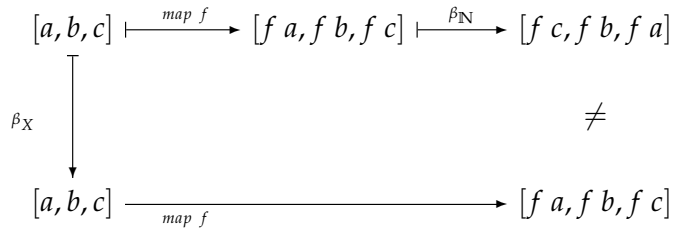
Given a transformation that reverses the list only if $\mathbf{X} = \mathbb{N}$:

$$\beta_x : \text{List } \mathbf{X} \to \text{List } \mathbf{X} = \begin{cases} [x_0, \ldots, x_{n-1}] \mapsto [x_0, \ldots, x_{n-1}] & \mathbf{X} \neq \mathbb{N} \\ [x_0, \ldots, x_{n-1}] \mapsto [x_{n-1}, \ldots, x_0] & \mathbf{X} = \mathbb{N} \end{cases}$$

Prove it not is natural.

Given $f : X \to \mathbb{N}$ where $X = a, b, c$

We can show α is natural through the following square commutes:



The diagram above does not commute, thus, is not natural.

Since this is not natural, and behaviour depends on the type of the argument, in this case whether they are natural numbers, this is ad-hoc polymorphism.

0.4.3 A transformation that only reverses lists of countably infinite types

Given a transformation that reverses the list only if **X** is countably infinite:

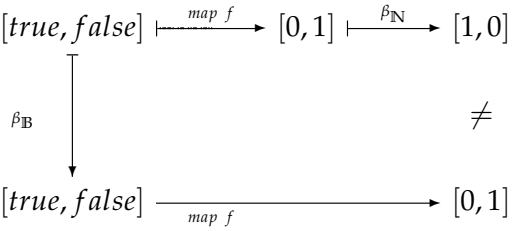
$$\gamma_x : \text{List } \mathbf{X} \to \text{List } \mathbf{X} = \begin{cases} [x_0, \ldots, x_{n-1}] \mapsto [x_0, \ldots, x_{n-1}] & \mathbf{X} \text{ is not countably infinite} \\ [x_0, \ldots, x_{n-1}] \mapsto [x_{n-1}, \ldots, x_0] & \mathbf{X} \text{ is countably infinite} \end{cases}$$

Prove it not is natural.

Given $f : \mathbb{B} \to \mathbb{N}$:

$$f(b) = \begin{cases} 0 & b = true \\ 1 & b = false \end{cases}$$

We can show α is natural through the following square commutes:



The diagram above does not commute, thus, is not natural. In fact, this example would also work for the previous exercise.