Function Types and Natural Transformations

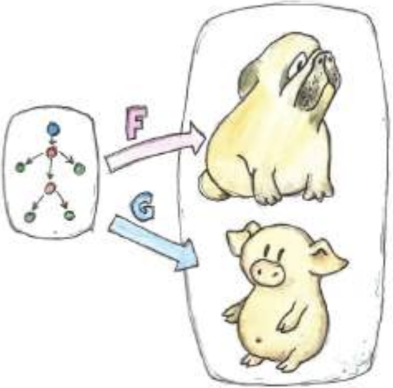
Jakub Sordyl, Szymon Wojtulewicz

04 November 2025

Recap: Functors

Functors

Structure-preserving mappings between categories.



Natural transformations

Functors

Structure-preserving mappings between categories.

Natural transformations

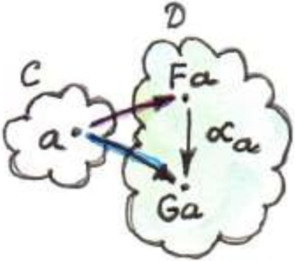
Functors

Structure-preserving mappings between categories.

Natural transformations

Mappings between functors that are *natural*/behave *naturally*.

Mapping objects: Components



F, G - functors between $\mathbf{C}$ and $\mathbf{D}$
Object a from $\mathbf{C}$ is mapped to Fa and Ga respectively

Let's say that α is the natural transformation then α_a is a selected morphism in $\mathbf{D}$ and is called the *component* of α at a.

Mapping morphisms

Morphism f between a and b in $\mathbf{C}$ is mapped to two morphisms in $\mathbf{D}$:
$Ff :: Fa \to Fb$
$Gf :: Ga \to Gb$

Mapping morphisms

Morphism f between a and b in $\mathbf{C}$ is mapped to two morphisms in $\mathbf{D}$:
$Ff :: Fa \to Fb$
$Gf :: Ga \to Gb$

Natural transformation α provides two additional morphisms:
$\alpha_a :: Fa \to Ga$
$\alpha_b :: Fa \to Gb$

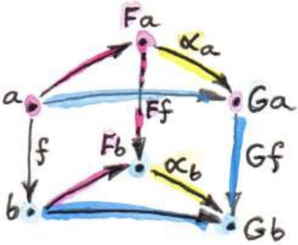
Mapping morphisms

Morphism f between a and b in $\mathbf{C}$ is mapped to two morphisms in $\mathbf{D}$:
$Ff :: Fa \to Fb$
$Gf :: Ga \to Gb$

Natural transformation α provides two additional morphisms:
$\alpha_a :: Fa \to Ga$
$\alpha_b :: Fa \to Gb$

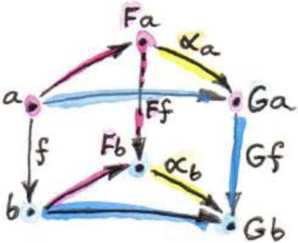
What are the restrictions on the choice of α_a and α_b ?

Naturality condition



What are the restrictions on the choice of α_a and α_b ?

Naturality condition

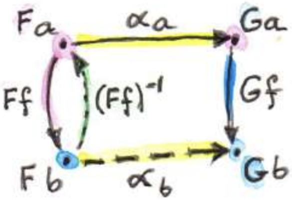


What are the restrictions on the choice of α_a and α_b ?

Naturality condition

$Gf \circ \alpha_a = \alpha_b \circ Ff$

Naturality condition - invertible morphism

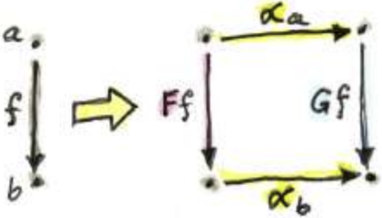


Naturality condition

$Gf \circ \alpha_a = \alpha_b \circ Ff$

$\alpha_b = Gf \circ \alpha_a \circ (Ff)^{-1}$

Commuting squares



Polymorphic functions

```haskell
alpha_a :: F a -> G a

alpha :: forall a . F a -> G a

alpha :: F a -> G a
```

```cpp
template<class A>
G<A> alpha(F<A>);
```

Polymorphic functions and naturality condition

Naturality condition

$Gf \circ \alpha_a = \alpha_b \circ Ff$

```
fmap f . alpha = alpha . fmap f
```

Example natural transformation - safeHead

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

Example natural transformation - safeHead

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

```haskell
fmap f [] = []
fmap f (x:xs) = f x : fmap f xs
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

Example natural transformation - safeHead

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

```
fmap f [] = []
fmap f (x:xs) = f x : fmap f xs
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

```
fmap f . safeHead = safeHead . fmap
```

Example natural transformation - safeHead

```haskell
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

```haskell
fmap f [] = []
fmap f (x:xs) = f x : fmap f xs
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

```haskell
fmap f . safeHead = safeHead . fmap
```

```haskell
fmap f (safeHead []) = fmap f Nothing = Nothing
safeHead (fmap f []) = safeHead [] = Nothing
```

Example natural transformation - safeHead

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

```
fmap f [] = []
fmap f (x:xs) = f x : fmap f xs
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

```
fmap f . safeHead = safeHead . fmap
```

```
fmap f (safeHead []) = fmap f Nothing = Nothing
safeHead (fmap f []) = safeHead [] = Nothing
```

```
fmap f (safeHead (x:xs)) = fmap f (Just x) = Just (f x)
safeHead (fmap f (x:xs)) = safeHead (f x : fmap f xs)
    = Just (f x)
```

Example natural transformation - safeHead

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

```
fmap f [] = []
fmap f (x:xs) = f x : fmap f xs
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

```
fmap f . safeHead = safeHead . fmap
```

```
fmap f (safeHead []) = fmap f Nothing = Nothing
safeHead (fmap f []) = safeHead [] = Nothing
```

```
fmap f (safeHead (x:xs)) = fmap f (Just x) = Just (f x)
safeHead (fmap f (x:xs)) = safeHead (f x : fmap f xs)
    = Just (f x)
```

```
maybeToList :: Maybe a -> [a]
```

Example natural transformation - length

```
length :: [a] -> Const Int a
length [] = Const 0
length (_:xs) = Const (1 + unConst (length xs))
```

Example natural transformation - `length`

```
length :: [a] -> Const Int a
length [] = Const 0
length (_:xs) = Const (1 + unConst (length xs))
```

```
length (_:xs) = fmap (+1) (length xs)
```

The Reader type

```
newtype Reader e a = Reader (e -> a)
instance Functor (Reader e) where
    fmap f (Reader g) = Reader (\x -> f (g x))
```

The Reader type

```
newtype Reader e a = Reader (e -> a)
instance Functor (Reader e) where
    fmap f (Reader g) = Reader (\x -> f (g x))
```

```
alpha :: Reader () a -> Maybe a
dumb (Reader _) = Nothing
obvious (Reader g) = Just (g ())
```

The Reader type

```haskell
newtype Reader e a = Reader (e -> a)
instance Functor (Reader e) where
    fmap f (Reader g) = Reader (\x -> f (g x))
```

```haskell
alpha :: Reader () a -> Maybe a
dumb (Reader _) = Nothing
obvious (Reader g) = Just (g ())
```


```haskell
alpha :: Reader () a -> Maybe a
dumb (Reader _) = Nothing
obvious (Reader g) = Just (g ())
```

Contravariance

```haskell
newtype Op r a = Op (a -> r)

instance Contravariant (Op r) where
    contramap f (Op g) = Op (g . f)
```

Contravariance

```haskell
newtype Op r a = Op (a -> r)

instance Contravariant (Op r) where
    contramap f (Op g) = Op (g . f)
```

```haskell
predToStr (Op f) = Op (\x -> if f x then "T" else "F")
```

Contravariance

```haskell
newtype Op r a = Op (a -> r)

instance Contravariant (Op r) where
    contramap f (Op g) = Op (g . f)
```

```haskell
predToStr (Op f) = Op (\x -> if f x then "T" else "F")
```

```haskell
contramap f . predToStr = predToStr . contramap f
```

Functor Category

There is one category of functors for each pair of categories, $\mathbf{C}$ and $\mathbf{D}$. Objects are functors from $\mathbf{C}$ to $\mathbf{D}$, and morphism are natural transformations

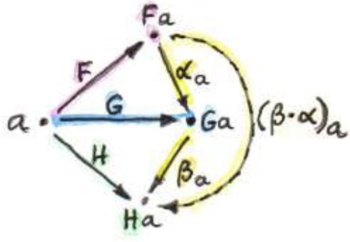
Functor Category

There is one category of functors for each pair of categories, $\mathbf{C}$ and $\mathbf{D}$. Objects are functors from $\mathbf{C}$ to $\mathbf{D}$, and morphism are natural transformations

Composition of natural transformations

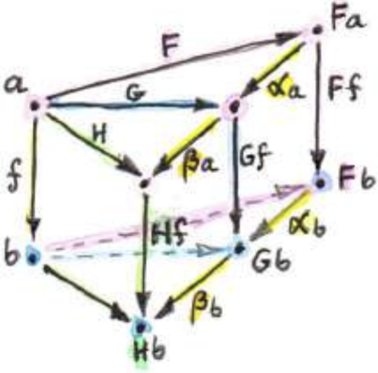
Let α be a natural transformation from functor F to functor G.
Let β be a natural transformation from functor G to functor H.
Components of α and β at a:
$\alpha_a :: Fa \to Ga$
$\beta_a :: Ga \to Ha$

Functor Category



$$(\beta \cdot \alpha)_a = \alpha_a \cdot \beta_a$$

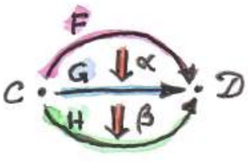
Functor Category



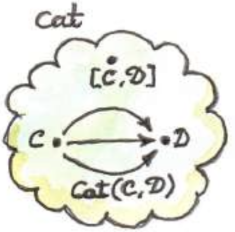
$$Hf \circ (\beta \cdot \alpha)_a = (\beta \cdot \alpha)_a \circ Ff$$

$$\mathbf{id}_{Fa} :: fa \to Fa$$

Vertical composition



2-Categories



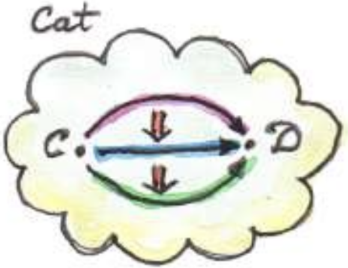
2-category - a category with 2-morphisms

2-morphisms are morphisms between morphisms

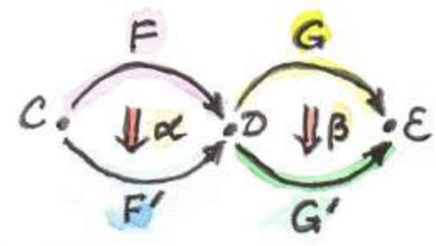
Cat as a 2-category

- Objects: (Small) categories
- 1-morphisms: Functors between categories
- 2-morphisms: Natural transformations between functors

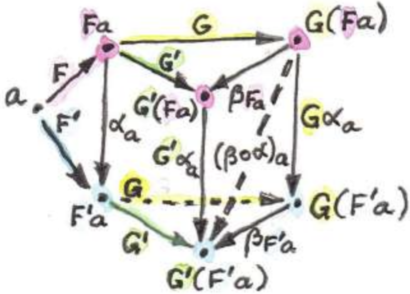
2-Categories



2-Categories



2-Categories



2-Categories