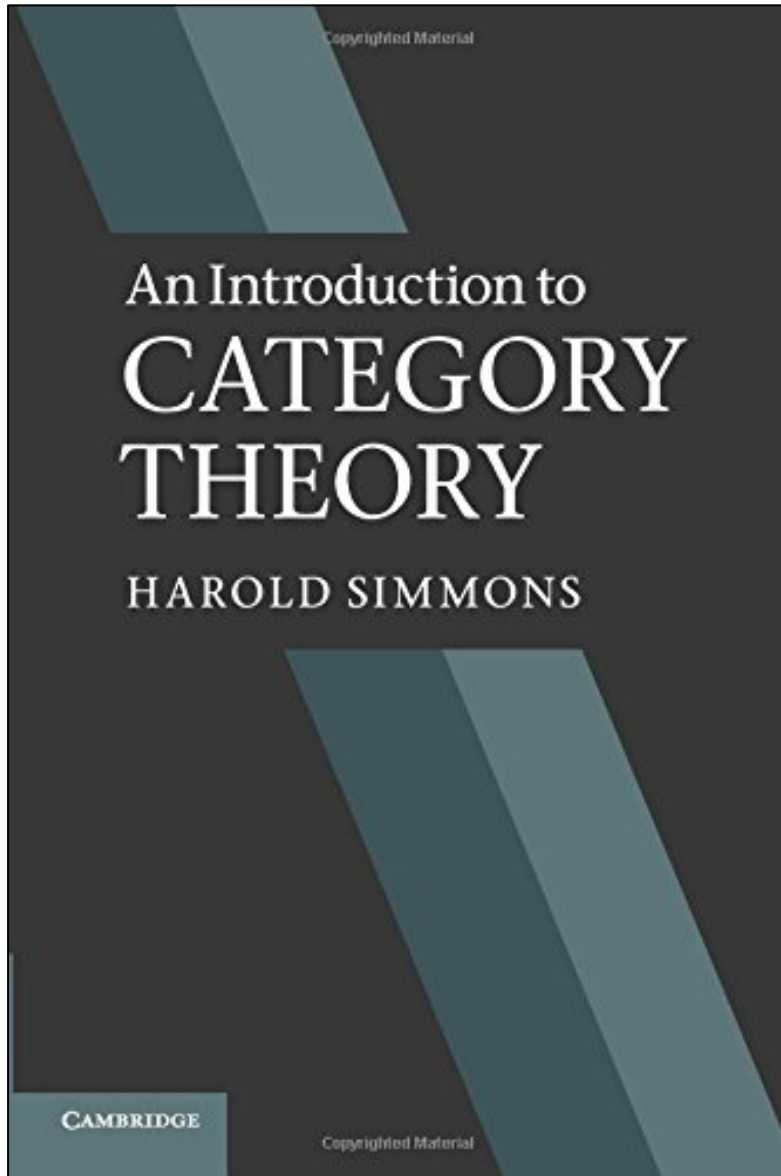


A summary of **Natural Transformations** based on two pages of the book on the left and on Bartosz Milewski's great lecture on the subject



Doesn't mention programming


## Category Theory 9.1: **Natural transformations**



**Bartosz Milewski**

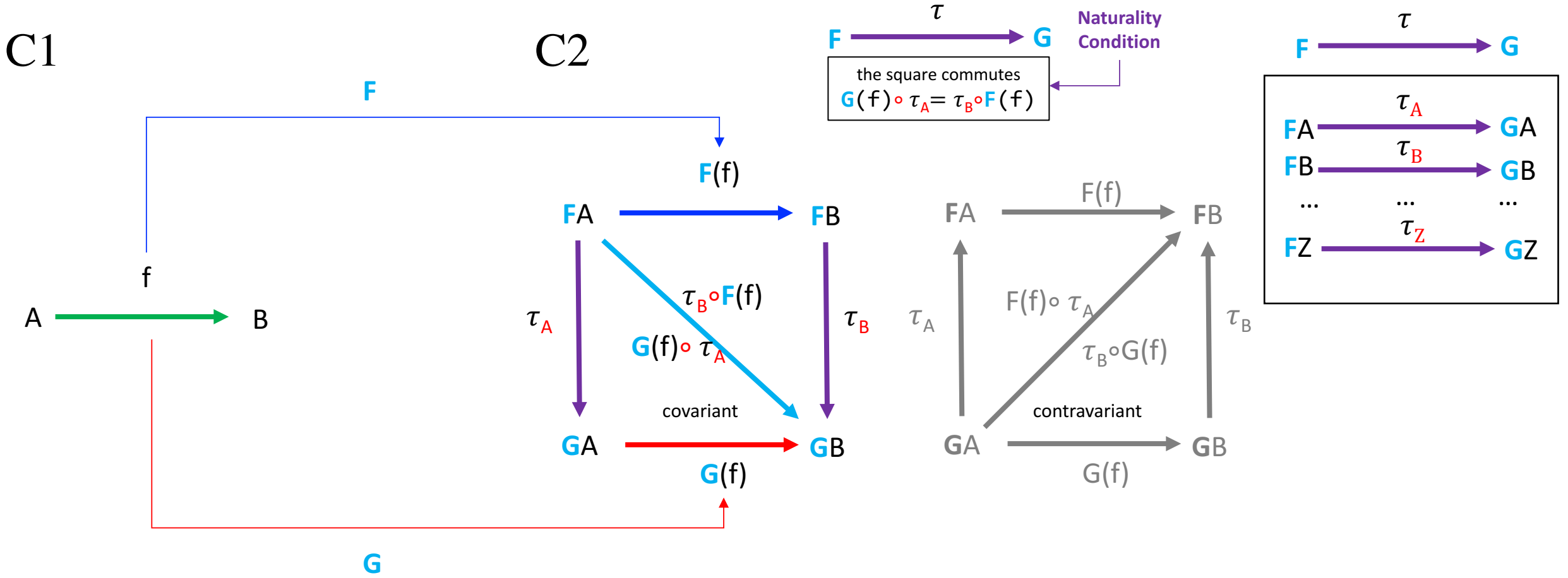
<https://twitter.com/BartoszMilewski>

Relates the subject to programming and shows examples

 **YouTube**<sup>GB</sup> <https://youtu.be/2LJC-XD5Ffo>

# Natural Transformation

C1 and C2 are categories and  $\circ$  denotes their composition operations.  
**F** and **G** are functors from C1 to C2 which map each C1 object to a C2 object and map each C1 arrow to a C2 arrow  
 A natural transformation  $\tau$  from **F** to **G** (either both covariant or both contravariant) is  
 a family of arrows  $\tau_X: \mathbf{F}X \rightarrow \mathbf{G}X$  of C2 indexed by the object X of C1 such that for each arrow  $f: A \rightarrow B$  of C1, the appropriate square in C2 commutes (depending on the variance)





So that's what a **natural transformation** is in Category Theory. But now you are asking me the question **what does it have to do with programming?** We already know what a **functor** is. We mostly talk about **endofunctors**. So we know what an endofunctor is.

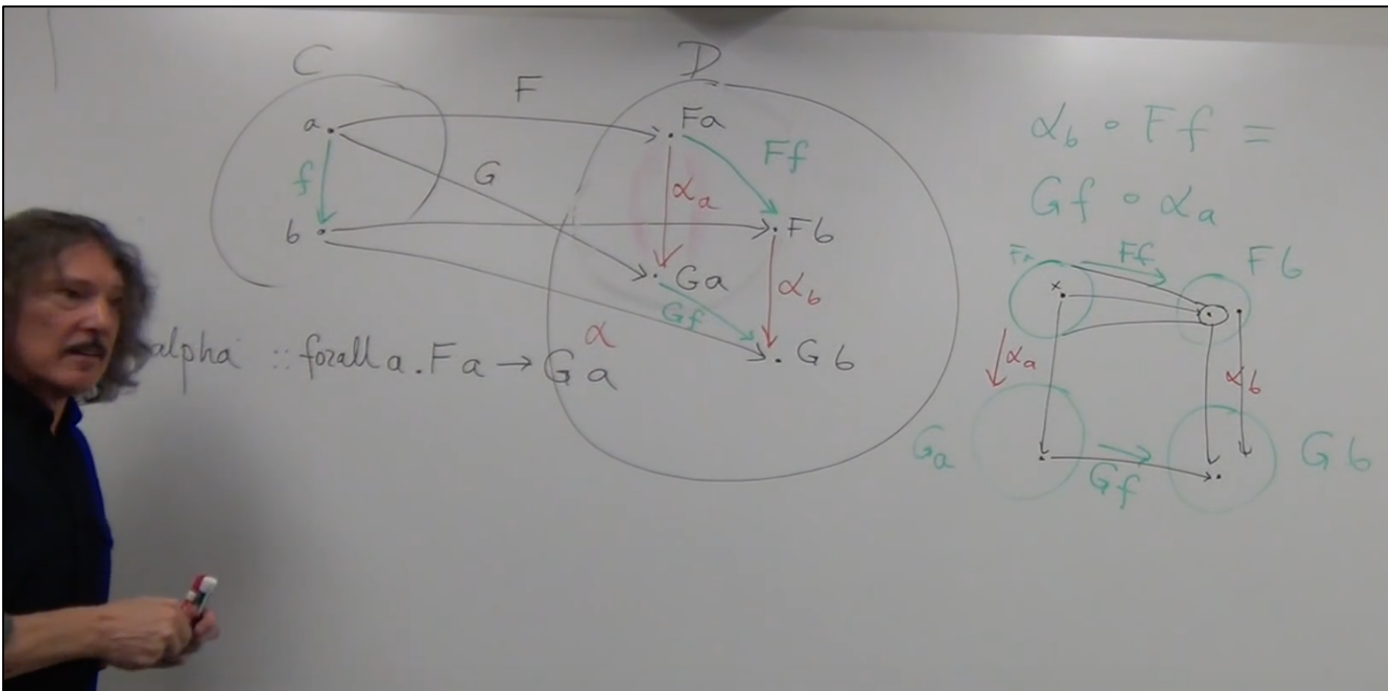
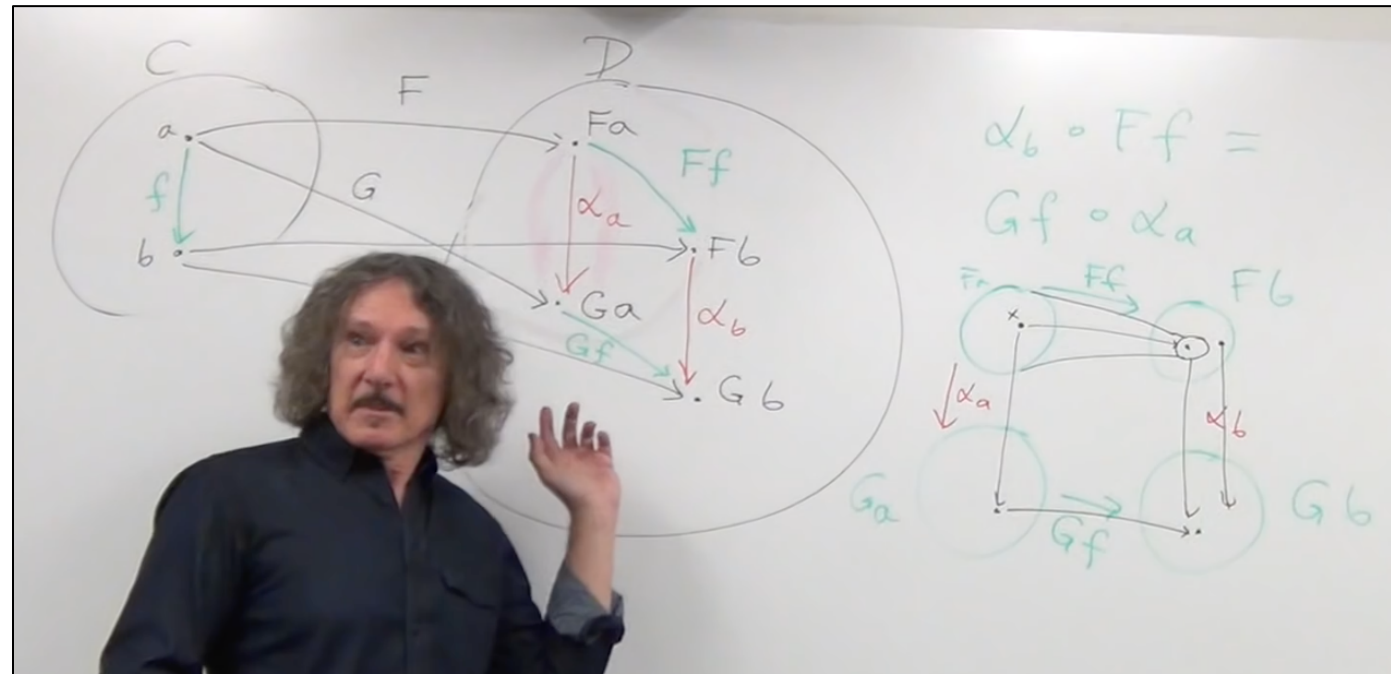
So a natural transformation would be a family of **morphisms** between two endofunctors. Morphisms here are functions. So it is a **family of functions**. A family of functions that is parametrized by a type is called a **polymorphic function**. So a **natural transformation is a polymorphic function**.

**Bartosz Milewski**

<https://twitter.com/BartoszMilewski>

Category Theory 9.1: Natural transformations

<https://youtu.be/2LJC-XD5Ffo>



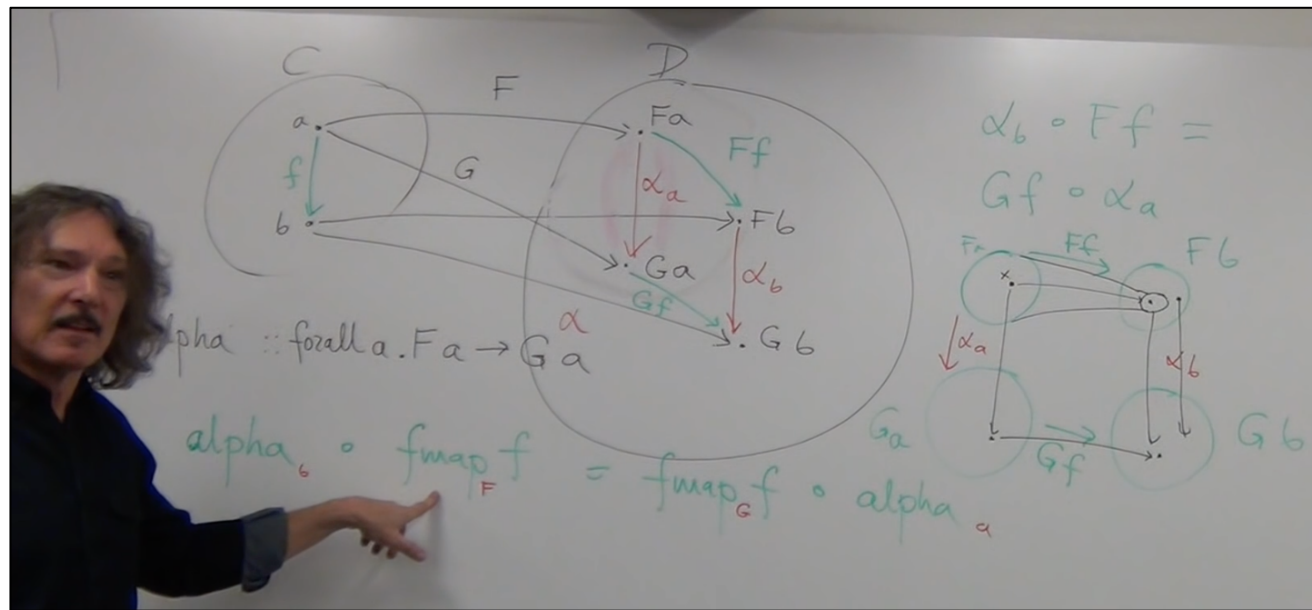
So suppose that we have two endofunctors **F** and **G**. So a natural transformation will go from **Fa** to **Ga**. So if we define (natural transformation) **alpha** it would be a function that goes from functor **Fa** to functor **Ga**.

**alpha** :: **Fa** -> **Ga**

So it is a function from **Fa** to **Ga**. If **a** is a lowercase letter for a type that means alpha is polymorphic in that type, but in **Haskell** we can actually say 'for all',

**alpha** :: forall a. **Fa** -> **Ga**

It is not mandatory, we can write a polymorphic function without **forall**, but if we want to stress the fact that this is defined for all types **a** we can use the 'explicit forall' extension.



**Bartosz Milewski**  
<https://twitter.com/BartoszMilewski>  
 Category Theory 9.1: Natural transformations  
<https://youtu.be/2LJC-XD5Ffo>

There is a subtle difference between this definition and our categorical definition.

The subtle difference is that in this form, in Haskell, we are assuming **parametric polymorphism**, meaning if we want to define this function we'll have to use one single formula for all **a**.

We cannot say do this thing for integers and a different thing for booleans, we cannot do that when we use parametric polymorphism.

We could use ad hoc polymorphism, but then we would have to go to type classes, but in this form, this means parametric polymorphism: one single formula for all. And this is much stronger than the categorical definition, because we haven't yet talked about **the naturality condition** ( $\alpha_a \circ Ff = Gf \circ \alpha_b$ ), the naturality square.

What would that mean. It would mean that...what is **Ff**? That's the **lifting** of a function in **Haskell**. That would be a lifting of the function **f** using the functor, capital **F**.

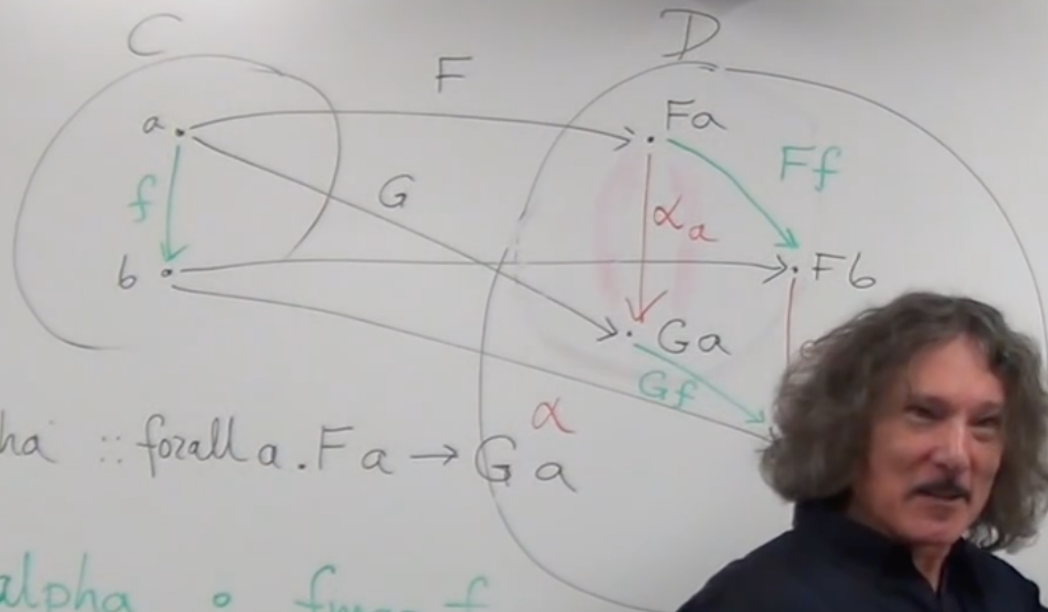
Lifting of a function is done through **fmap**. So the formula ( $\alpha_a \circ Ff = Gf \circ \alpha_b$ ) translates into:

$$\alpha_b \circ \text{fmap } f = \text{fmap } f \circ \alpha_a$$

So this is the naturality condition written in **Haskell**, and in **Haskell** I don't have to specify that the first is **alpha b** and the second is **alpha a**. I mean I could do this for explanation. These two **fmap**s are different **fmap**s. The first one is **fmap** for the functor **F**, which could be completely different from the second which is **fmap** for the functor **G**.

$$\alpha_b \circ \text{fmap}_F f = \text{fmap}_G f \circ \alpha_a$$

And instead of talking of this, I'll give you an example in a moment. But what I want to say is that **because of parametric polymorphism, this is automatic. This is a theorem for free. I don't have to check it. I never have to check the naturality condition. If I defined a function of type  $Fa \rightarrow Ga$ , that is parametrically polymorphic, it is automatically a natural transformation.**



$\alpha_i :: \text{forall } a. Fa \rightarrow Ga$

$\alpha_b \circ \text{fmap}_F f =$

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

safeHead . fmap f (x:xs)  
 Just (fx) : fmap f xs

fmap f safeHead (x:xs)  
 Just x  
 → Just (fx)

**Bartosz Milewski**  
<https://twitter.com/BartoszMilewski>

Category Theory 9.1:  
 Natural transformations  
<https://youtu.be/2LJC-XD5Ffo>

So let's pick two functors. Let's pick the **List** functor and the **Maybe** functor...and let's talk about **safeHead**. Now **head** is a function that takes a list and returns the first element of the list. And **it's a bad function because it is not total: if the list is empty it just blows up. But we can define a safeHead.**

...  
 I like this example because it shows you that **category theory can be used in programming in a very practical way!** If you look at this, **it is actually an optimisation. If the compiler knows about the naturality condition, it can do a clever thing. Applying an fmap to a list is expensive, so being able to do the naturality thing and applying safeHead first and then fmap is cheaper.** Of course not in **Haskell**, because **Haskell** is lazy. But in many cases these kinds of transformations that have a basis in category theory can actually be used to optimise code.

# Concrete Scala Example: **safeHead** - natural transformation $\tau$ from **List** functor to **Option** functor

```

val length: String => Int = s => s.length

// a natural transformation
def safeHead[A]: List[A] => Option[A] = {
  case head::_ => Some(head)
  case Nil => None
}
    
```

natural transformation  $\tau$  from **List** to **Option**



**F**[A] is type A lifted into context **F**  
 $f_{\uparrow F}$  is function f lifted into context **F**

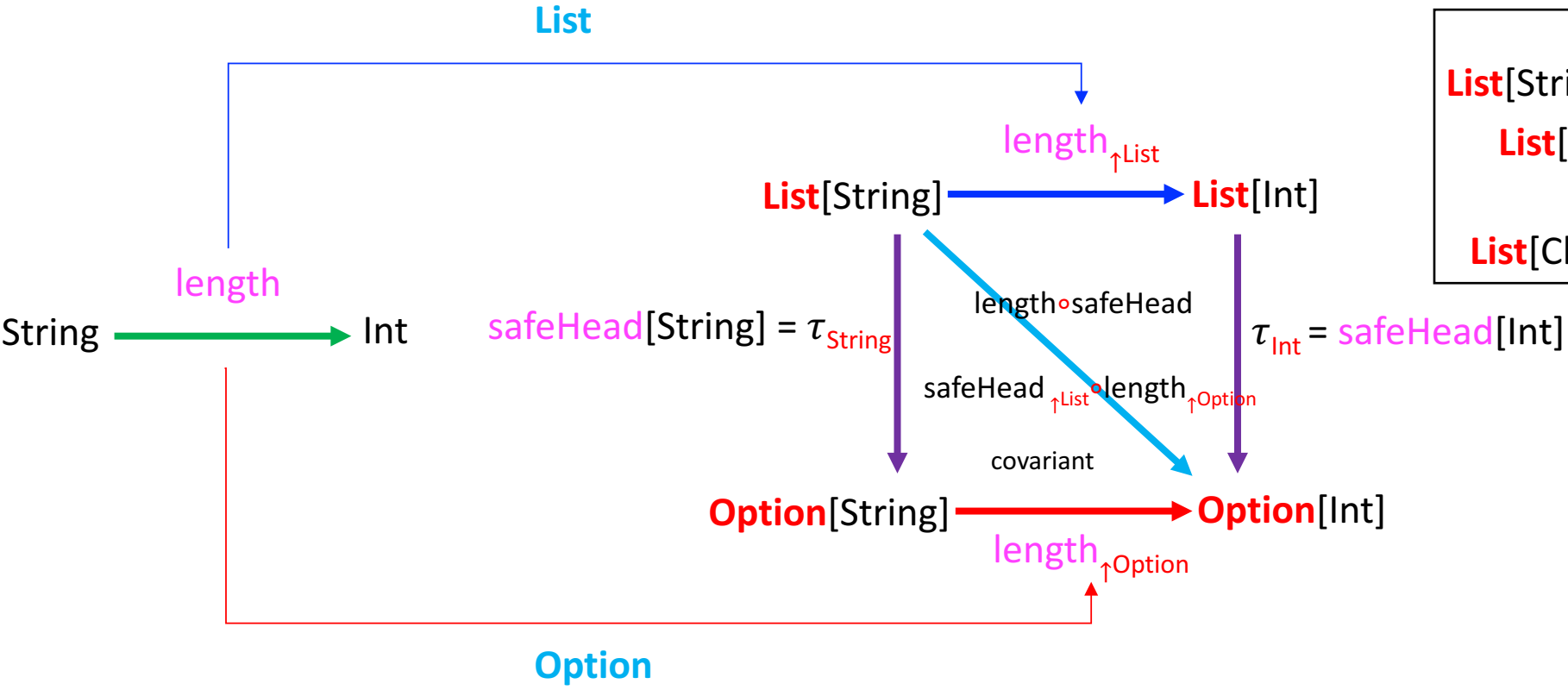
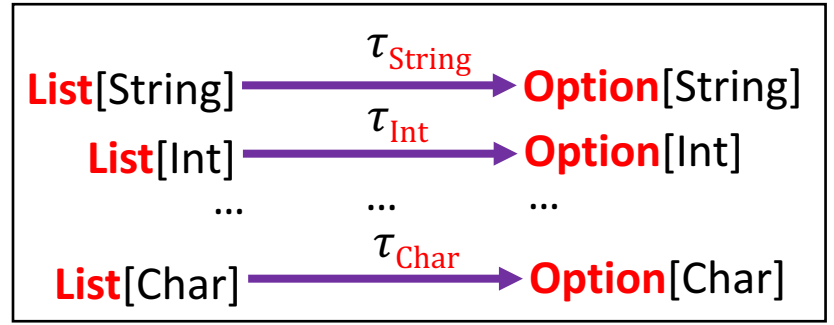
**map** lifts f into **F**  
 $f_{\uparrow F}$  is **map** f

the square commutes

$$\begin{aligned}
 \text{length}_{\uparrow \text{List}} \circ \text{safeHead} &= \text{safeHead} \circ \text{length}_{\uparrow \text{Option}} \\
 (\text{map}_{\text{List}} \text{ length}) \circ \text{safeHead} &= \text{safeHead} \circ (\text{map}_{\text{Option}} \text{ length})
 \end{aligned}$$

C1 = C2 = **Scala** types and functions

Naturality Condition



# Concrete Scala Example: `safeHead` - natural transformation $\tau$ from `List` functor to `Option` functor

```
trait Functor[F[_]] {  
  def map[A, B](f: A => B): F[A] => F[B]  
}
```

```
val listF = new Functor[List] {  
  def map[A, B](f: A => B): List[A] => List[B] = {  
    case head::tail => f(head)::map(f)(tail)  
    case Nil => Nil  
  }  
}
```

```
val length: String => Int = s => s.length
```

```
def safeHead[A]: List[A] => Option[A] = {  
  case head::_ => Some(head)  
  case Nil => None  
}
```

```
val mapAndThenTransform: List[String] => Option[Int] = safeHead compose (listF map length)
```

```
val transformAndThenMap: List[String] => Option[Int] = (optionF map length) compose safeHead
```

```
assert(mapAndThenTransform(List("abc", "d", "ef")) == transformAndThenMap(List("abc", "d", "ef")))
```

```
assert(mapAndThenTransform(List("abc", "d", "ef")) == Some(3))
```

```
assert(transformAndThenMap(List("abc", "d", "ef")) == Some(3))
```

```
assert(mapAndThenTransform(List()) == transformAndThenMap(List()))
```

```
assert(mapAndThenTransform(List()) == None)
```

```
assert(transformAndThenMap(List()) == None)
```

```
val optionF = new Functor[Option] {  
  def map[A, B](f: A => B): Option[A] => Option[B] = {  
    case Some(a) => Some(f(a))  
    case None => None  
  }  
}
```



Naturality  
Condition

the square commutes

$$\text{length}_{\uparrow\text{List}} \circ \text{safeHead} = \text{safeHead} \circ \text{length}_{\uparrow\text{Option}}$$

$$(\text{map}_{\text{List}} \text{length}) \circ \text{safeHead} = \text{safeHead} \circ (\text{map}_{\text{Option}} \text{length})$$