Bartosz Milewski's Programming Cafe

Category Theory, Haskell, Concurrency, C++

April 7, 2015

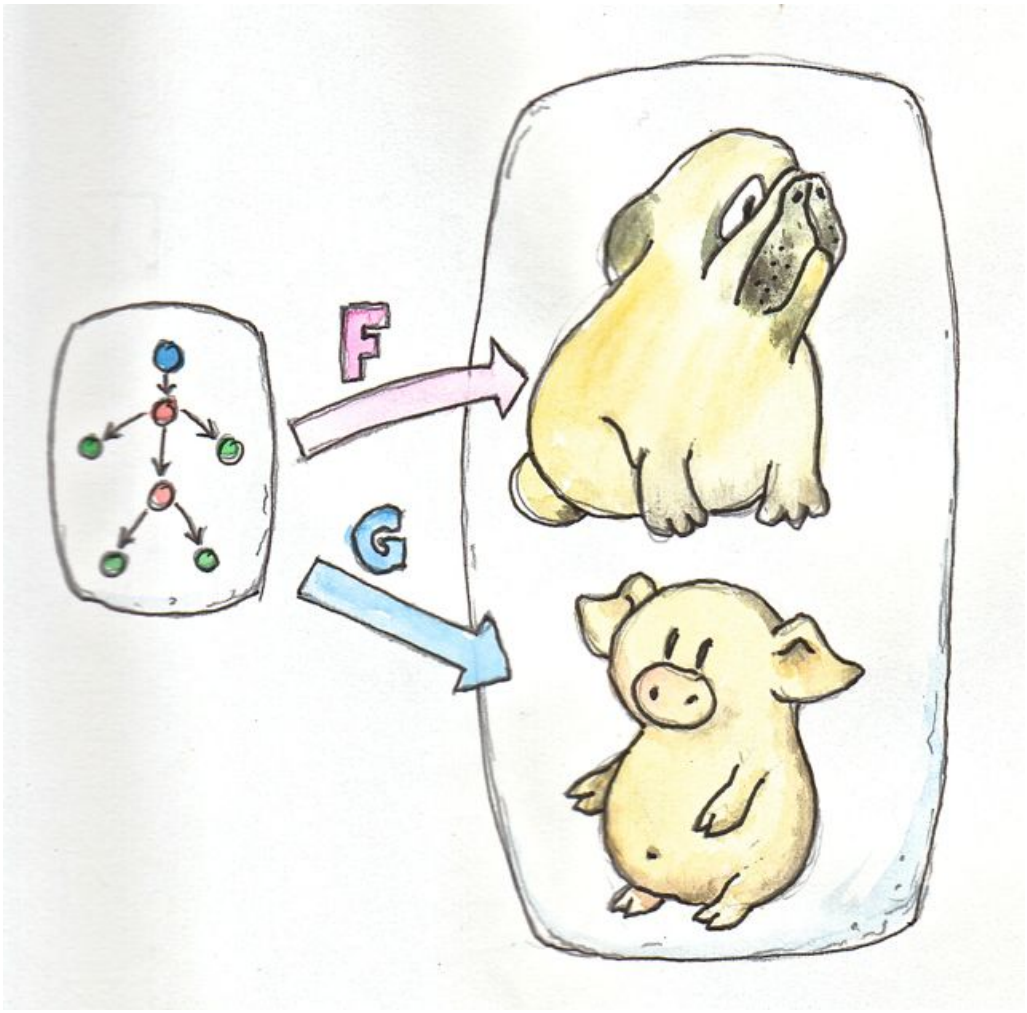
Natural Transformations

Posted by Bartosz Milewski under Category Theory, Functional Programming, Haskell, Programming [59] Comments

i
27 Votes

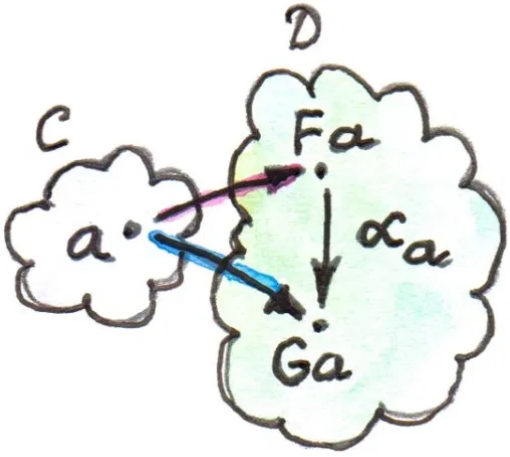
This is part 10 of Categories for Programmers. Previously: Function Types. See the Table of Contents.

We talked about functors as mappings between categories that preserve their structure. A functor "embeds" one category in another. It may collapse multiple things into one, but it never breaks connections. One way of thinking about it is that with a functor we are modeling one category inside another. The source category serves as a model, a blueprint, for some structure that's part of the target category.



There may be many ways of embedding one category in another. Sometimes they are equivalent, sometimes very different. One may collapse the whole source category into one object, another may map every object to a different object and every morphism to a different morphism. The same blueprint may be realized in many different ways. Natural transformations help us compare these realizations. They are mappings of functors — special mappings that preserve their functorial nature.

Consider two functors `F` and `G` between categories *C* and *D*. If you focus on just one object `a` in *C*, it is mapped to two objects: `F a` and `G a`. A mapping of functors should therefore map `F a` to `G a`.



Notice that `F a` and `G a` are objects in the same category *D*. Mappings between objects in the same category should not go against the grain of the category. We don't want to make artificial connections between objects. So it's *natural* to use existing connections, namely morphisms. A natural transformation is a selection of morphisms: for every object `a`, it picks one morphism from `F a` to `G a`. If we call the natural transformation α, this morphism is called the *component* of α at `a`, or α_a.

```
αa :: F a -> G a
```

Keep in mind that `a` is an object in *C* while α_a is a morphism in *D*.

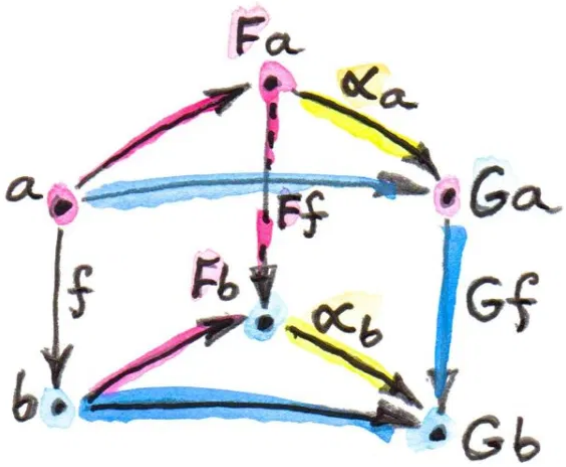
If, for some `a`, there is no morphism between `F a` and `G a` in *D*, there can be no natural transformation between `F` and `G`.

Of course that's only half of the story, because functors not only map objects, they map morphisms as well. So what does a natural transformation do with those mappings? It turns out that the mapping of morphisms is fixed — under any natural transformation between F and G, `F f` must be transformed into `G f`. What's more, the mapping of morphisms by the two functors drastically restricts the choices we have in defining a natural transformation that's compatible with it. Consider a morphism `f` between two objects `a` and `b` in *C*. It's mapped to two morphisms, `F f` and `G f` in *D*:

```
F f :: F a -> F b
G f :: G a -> G b
```

The natural transformation α provides two additional morphisms that complete the diagram in *D*:

```
αa :: F a -> G a
αb :: F b -> G b
```

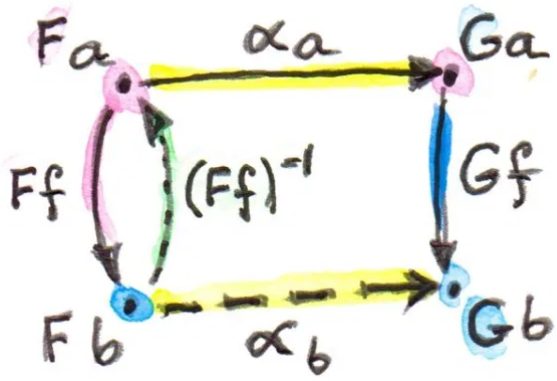


Now we have two ways of getting from `F a` to `G b`. To make sure that they are equal, we must impose the *naturality condition* that holds for any `f`:

```
G f ∘ αa = αb ∘ F f
```

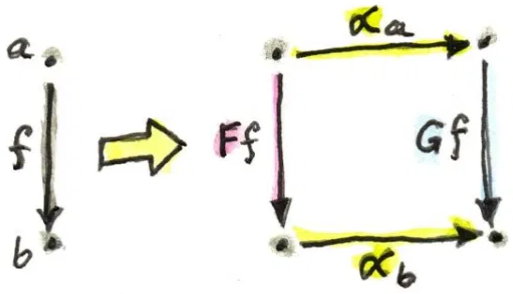
The naturality condition is a pretty stringent requirement. For instance, if the morphism `F f` is invertible, naturality determines α_b in terms of α_a. It *transports* α_a along `f`:

$$\alpha_b = (G\ f) \circ \alpha_a \circ (F\ f)^{-1}$$



If there is more than one invertible morphism between two objects, all these transports have to agree. In general, though, morphisms are not invertible; but you can see that the existence of natural transformations between two functors is far from guaranteed. So the scarcity or the abundance of functors that are related by natural transformations may tell you a lot about the structure of categories between which they operate. We'll see some examples of that when we talk about limits and the Yoneda lemma.

Looking at a natural transformation component-wise, one may say that it maps objects to morphisms. Because of the naturality condition, one may also say that it maps morphisms to commuting squares — there is one commuting naturality square in *D* for every morphism in *C*.



This property of natural transformations comes in very handy in a lot of categorical constructions, which often include commuting diagrams. With a judicious choice of functors, a lot of these commutativity conditions may be transformed into naturality conditions. We'll see examples of that when we get to limits, colimits, and adjunctions.

Finally, natural transformations may be used to define isomorphisms of functors. Saying that two functors are naturally isomorphic is almost like saying they are the same. *Natural isomorphism* is defined as a natural transformation whose components are all isomorphisms (invertible morphisms).

Polymorphic Functions

We talked about the role of functors (or, more specifically, endofunctors) in programming. They correspond to type constructors that map types to types. They also map functions to functions, and this mapping is implemented by a higher order function `fmap` (or `transform`, `then`, and the like in C++).

To construct a natural transformation we start with an object, here a type, `a`. One functor, `F`, maps it to the type `F a`. Another functor, `G`, maps it to `G a`. The component of a natural transformation `alpha` at `a` is a function from `F a` to `G a`. In pseudo-Haskell:

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

A natural transformation is a polymorphic function that is defined for all types `a`:

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

The `forall a` is optional in Haskell (and in fact requires turning on the language extension `ExplicitForAll`). Normally, you would write it like this:

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

Keep in mind that it's really a family of functions parameterized by `a`. This is another example of the terseness of the Haskell syntax. A similar construct in C++ would be slightly more verbose:

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

There is a more profound difference between Haskell's polymorphic functions and C++ generic functions, and it's reflected in the way these functions are implemented and type-checked. In Haskell, a polymorphic function must be defined uniformly for all types. One formula must work across all types. This is called *parametric polymorphism*.

C++, on the other hand, supports by default *ad hoc polymorphism*, which means that a template doesn't have to be well-defined for all types. Whether a template will work for a given type is decided at instantiation time, where a concrete type is substituted for the type parameter. Type checking is deferred, which unfortunately often leads to incomprehensible error messages.

In C++, there is also a mechanism for function overloading and template specialization, which allows different definitions of the same function for different types. In Haskell this functionality is provided by type classes and type families.

Haskell's parametric polymorphism has an unexpected consequence: any polymorphic function of the type:

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

where `F` and `G` are functors, automatically satisfies the naturality condition. Here it is in categorical notation (`f` is a function `f::a->b`):

```
G f ∘ α_a = α_b ∘ F f
```

In Haskell, the action of a functor `G` on a morphism `f` is implemented using `fmap`. I'll first write it in pseudo-Haskell, with explicit type annotations:

```
fmap_G f . alpha_a = alpha_b . fmap_F f
```

Because of type inference, these annotations are not necessary, and the following equation holds:

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

This is still not real Haskell — function equality is not expressible in code — but it's an identity that can be used by the programmer in equational reasoning; or by the compiler, to implement optimizations.

The reason why the naturality condition is automatic in Haskell has to do with "theorems for free." Parametric polymorphism, which is used to define natural transformations in Haskell, imposes very strong limitations on the implementation — one formula for all types. These limitations translate into equational theorems about such functions. In the case of functions that transform functors, free theorems are the naturality conditions. [You may read more about free theorems in my blog Parametricity: Money for Nothing and Theorems for Free.]

One way of thinking about functors in Haskell that I mentioned earlier is to consider them generalized containers. We can continue this analogy and consider natural transformations to be recipes for repackaging the contents of one container into another container. We are not touching the items themselves: we don't modify them, and we don't create new ones. We are just copying (some of) them, sometimes multiple times, into a new container.

The naturality condition becomes the statement that it doesn't matter whether we modify the items first, through the application of `fmap`, and repackage later; or repackage first, and then modify the items in the new container, with its own implementation of `fmap`. These two actions, repackaging and `fmap`ping, are orthogonal. "One moves the eggs, the other boils them."

Let's see a few examples of natural transformations in Haskell. The first is between the list functor, and the `Maybe` functor. It returns the head of the list, but only if the list is non-empty:

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

It's a function polymorphic in `a`. It works for any type `a`, with no limitations, so it is an example of parametric polymorphism. Therefore it is a natural transformation between the two functors. But just to convince ourselves, let's verify the naturality condition.

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

We have two cases to consider; an empty list:

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

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

and a non-empty list:

```
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)
```

I used the implementation of `fmap` for lists:

```
fmap f [] = []
fmap f (x:xs) = f x : fmap f xs
```

and for `Maybe`:

```
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

An interesting case is when one of the functors is the trivial `Const` functor. A natural transformation from or to a `Const` functor looks just like a function that's either polymorphic in its return type or in its argument type.

For instance, `length` can be thought of as a natural transformation from the list functor to the `Const Int` functor:

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

Here, `unConst` is used to peel off the `Const` constructor:

```
unConst :: Const c a -> c
unConst (Const x) = x
```

Of course, in practice `length` is defined as:

```
length :: [a] -> Int
```

which effectively hides the fact that it's a natural transformation.

Finding a parametrically polymorphic function *from* a `Const` functor is a little harder, since it would require the creation of a value from nothing. The best we can do is:

```
scam :: Const Int a -> Maybe a
scam (Const x) = Nothing
```

Another common functor that we've seen already, and which will play an important role in the Yoneda lemma, is the `Reader` functor. I will rewrite its definition as a `newtype`:

```
newtype Reader e a = Reader (e -> a)
```

It is parameterized by two types, but is (covariantly) functorial only in the second one:

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

For every type `e`, you can define a family of natural transformations from `Reader e` to any other functor `f`. We'll see later that the members of this family are always in one to one correspondence with the elements of `f e` (the Yoneda lemma).

For instance, consider the somewhat trivial unit type `()` with one element `()`. The functor `Reader ()` takes any type `a` and maps it into a function type `()->a`. These are just all the functions that pick a single element from the set `a`. There are as many of these as there are elements in `a`. Now let's consider natural transformations from this functor to the `Maybe` functor:

```
alpha :: Reader () a -> Maybe a
```

There are only two of these, `dumb` and `obvious`:

```
dumb (Reader _) = Nothing
```

and

```
obvious (Reader g) = Just (g ())
```

(The only thing you can do with `g` is to apply it to the unit value `()`.)

And, indeed, as predicted by the Yoneda lemma, these correspond to the two elements of the `Maybe ()` type, which are `Nothing` and `Just ()`. We'll come back to the Yoneda lemma later — this was just a little teaser.

Beyond Naturality

A parametrically polymorphic function between two functors (including the edge case of the `Const` functor) is always a natural transformation. Since all standard algebraic data types are functors, any polymorphic function between such types is a natural transformation.

We also have function types at our disposal, and those are functorial in their return type. We can use them to build functors (like the `Reader` functor) and define natural transformations that are higher-order functions.

However, function types are not covariant in the argument type. They are *contravariant*. Of course contravariant functors are equivalent to covariant functors from the opposite category. Polymorphic functions between two contravariant functors are still natural transformations in the categorical sense, except that they work on functors from the opposite category to Haskell types.

You might remember the example of a contravariant functor we've looked at before:

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

This functor is contravariant in `a`:

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

We can write a polymorphic function from, say, `Op Bool` to `Op String`:

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

But since the two functors are not covariant, this is not a natural transformation in **Hask**. However, because they are both contravariant, they satisfy the "opposite" naturality condition:

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

Notice that the function `f` must go in the opposite direction than what you'd use with `fmap`, because of the signature of `contramap`:

```
contramap :: (b -> a) -> (Op Bool a -> Op Bool b)
```

Are there any type constructors that are not functors, whether covariant or contravariant? Here's one example:

```
a -> a
```

This is not a functor because the same type `a` is used both in the negative (contravariant) and positive (covariant) position. You can't implement `fmap` or `contramap` for this type. Therefore a function of the signature:

```
(a -> a) -> f a
```

where `f` is an arbitrary functor, cannot be a natural transformation. Interestingly, there is a generalization of natural transformations, called dinatural transformations, that deals with such cases. We'll get to them when we discuss ends.

Functor Category

Now that we have mappings between functors — natural transformations — it's only natural to ask the question whether functors form a category. And indeed they do! There is one category of functors for each pair of categories, C and D. Objects in this category are functors from C to D, and morphisms are natural transformations between those functors.

We have to define composition of two natural transformations, but that's quite easy. The components of natural transformations are morphisms, and we know how to compose morphisms.

Indeed, let's take a natural transformation α from functor F to G. Its component at object `a` is some morphism:

α_a `:: F a -> G a`

We'd like to compose α with β, which is a natural transformation from functor G to H. The component of β at `a` is a morphism:

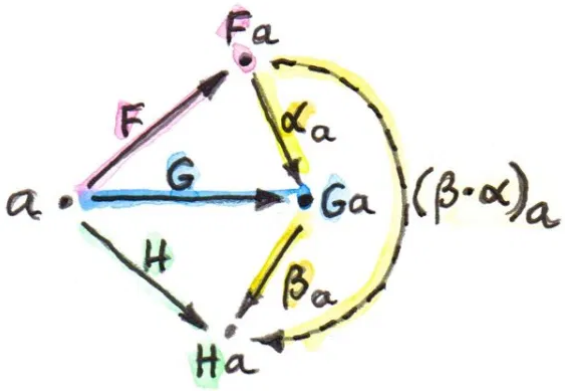
β_a `:: G a -> H a`

These morphisms are composable and their composition is another morphism:

```
βa ∘ αa :: F a -> H a
```

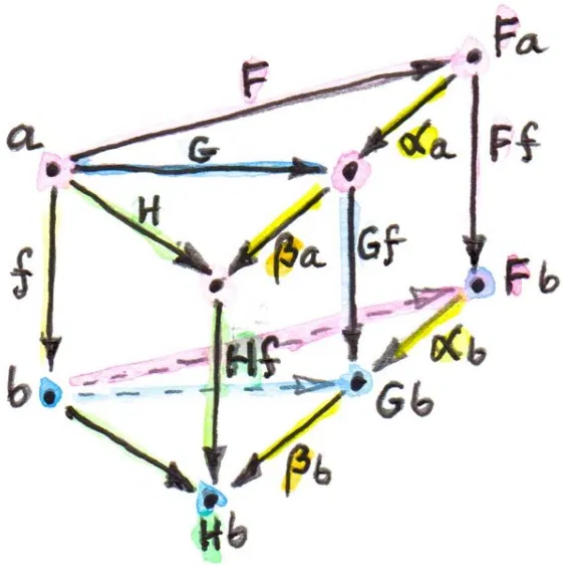
We will use this morphism as the component of the natural transformation $\beta \cdot \alpha$ — the composition of two natural transformations β after α:

```
(β · α)a = βa ∘ αa
```



One (long) look at a diagram convinces us that the result of this composition is indeed a natural transformation from F to H:

```
H f ∘ (β · α)a = (β · α)b ∘ F f
```



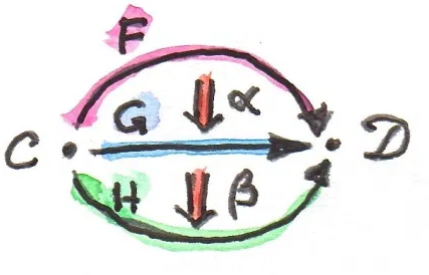
Composition of natural transformations is associative, because their components, which are regular morphisms, are associative with respect to their composition.

Finally, for each functor F there is an identity natural transformation 1_F whose components are the identity morphisms:

```
idF a :: F a -> F a
```

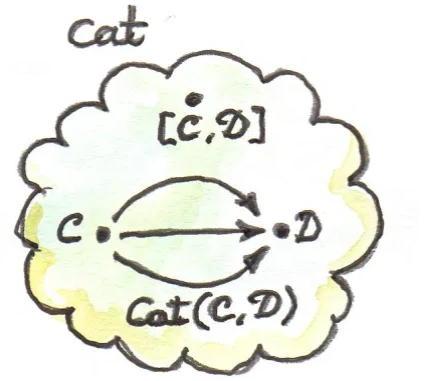
So, indeed, functors form a category.

A word about notation. Following Saunders Mac Lane I use the dot for the kind of natural transformation composition I have just described. The problem is that there are two ways of composing natural transformations. This one is called the vertical composition, because the functors are usually stacked up vertically in the diagrams that describe it. Vertical composition is important in defining the functor category. I'll explain horizontal composition shortly.



The functor category between categories C and D is written as `Fun(C, D)`, or `[C, D]`, or sometimes as D^C. This last notation suggests that a functor category itself might be considered a function object (an exponential) in some other category. Is this indeed the case?

Let's have a look at the hierarchy of abstractions that we've been building so far. We started with a category, which is a collection of objects and morphisms. Categories themselves (or, strictly speaking *small* categories, whose objects form sets) are themselves objects in a higher-level category **Cat**. Morphisms in that category are functors. A Hom-set in **Cat** is a set of functors. For instance Cat(C, D) is a set of functors between two categories C and D.



A functor category [C, D] is also a set of functors between two categories (plus natural transformations as morphisms). Its objects are the same as the members of Cat(C, D). Moreover, a functor category, being a category, must itself be an object of **Cat** (it so happens that the functor category between two small categories is itself small). We have a relationship between a Hom-set in a category and an object in the same category. The situation is exactly like the exponential object that we've seen in the last section. Let's see how we can construct the latter in **Cat**.

As you may remember, in order to construct an exponential, we need to first define a product. In **Cat**, this turns out to be relatively easy, because small categories are *sets* of objects, and we know how to define cartesian products of sets. So an object in a product category C × D is just a pair of objects, `(c, d)`, one from C and one from D. Similarly, a morphism between two such pairs, `(c, d)` and `(c', d')`, is a pair of morphisms, `(f, g)`, where `f :: c -> c'` and `g :: d -> d'`. These pairs of morphisms compose component-wise, and there is always an identity pair that is just a pair of identity

morphisms. To make the long story short, **Cat** is a full-blown cartesian closed category in which there is an exponential object D^C for any pair of categories. And by "object" in **Cat** I mean a category, so D^C is a category, which we can identify with the functor category between C and D.

2-Categories

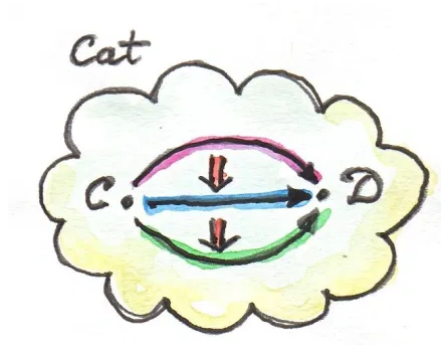
With that out of the way, let's have a closer look at **Cat**. By definition, any Hom-set in **Cat** is a set of functors. But, as we have seen, functors between two objects have a richer structure than just a set. They form a category, with natural transformations acting as morphisms. Since functors are considered morphisms in **Cat**, natural transformations are morphisms between morphisms.

This richer structure is an example of a 2-category, a generalization of a category where, besides objects and morphisms (which might be called 1-morphisms in this context), there are also 2-morphisms, which are morphisms between morphisms.

In the case of **Cat** seen as a 2-category we have:

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

Instead of a Hom-set between two categories C and D, we have a Hom-category — the functor category D^C. We have regular functor composition: a functor F from D^C composes with a functor G from E^D to give $G \circ F$ from E^C. But we also have composition inside each Hom-category — vertical composition of natural transformations, or 2-morphisms, between functors.



With two kinds of composition in a 2-category, the question arises: How do they interact with each other?

Let's pick two functors, or 1-morphisms, in **Cat**:

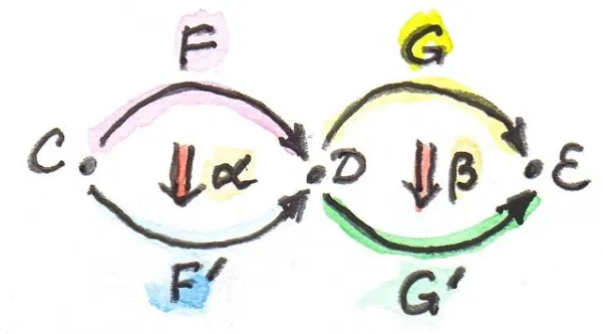
```
F :: C -> D
G :: D -> E
```

and their composition:

```
G ∘ F :: C -> E
```

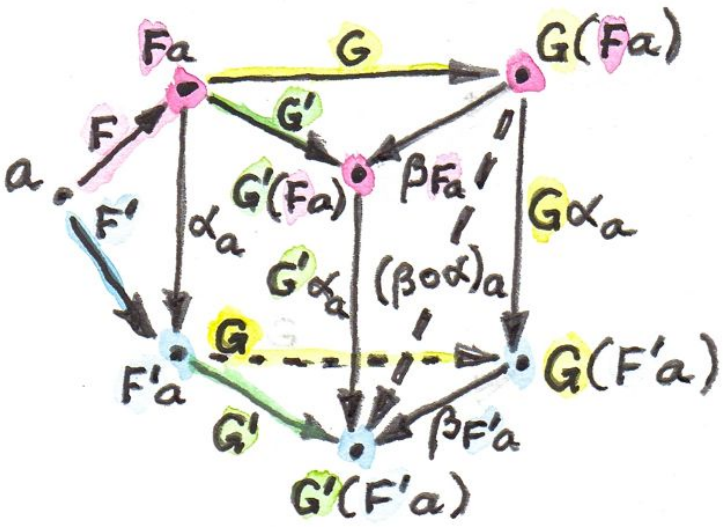
Suppose we have two natural transformations, α and β, that act, respectively, on functors F and G:

```
α :: F -> F'
β :: G -> G'
```



Notice that we cannot apply vertical composition to this pair, because the target of α is different from the source of β. In fact they are members of two different functor categories: D^C and E^D. We can, however, apply composition to the functors F′ and G′, because the target of F′ is the source of G′ — it's the category D. What's the relation between the functors G′ ∘ F′ and G ∘ F?

Having α and β at our disposal, can we define a natural transformation from G ∘ F to G′ ∘ F′? Let me sketch the construction.



As usual, we start with an object `a` in C. Its image splits into two objects in D: `F a` and `F'a`. There is also a morphism, a component of α, connecting these two objects:

```
αa :: F a -> F'a
```

When going from D to E, these two objects split further into four objects:

```
G (F a), G'(F a), G (F'a), G'(F'a)
```

We also have four morphisms forming a square. Two of these morphisms are the components of the natural transformation β:

```
βF a :: G (F a) -> G'(F a)
βF'a :: G (F'a) -> G'(F'a)
```

The other two are the images of α_a under the two functors (functors map morphisms):

```
G αa :: G (F a) -> G (F'a)
G'αa :: G'(F a) -> G'(F'a)
```

That's a lot of morphisms. Our goal is to find a morphism that goes from `G (F a)` to `G'(F'a)`, a candidate for the component of a natural transformation connecting the two functors G ◦ F and G′◦ F′. In fact there's not one but two paths we can take from `G (F a)` to `G'(F'a)`:

```
G'αa ∘ βF a
βF'a ∘ G αa
```

Luckily for us, they are equal, because the square we have formed turns out to be the naturality square for β.

We have just defined a component of a natural transformation from G ◦ F to G′◦ F′. The proof of naturality for this transformation is pretty straightforward, provided you have enough patience.

We call this natural transformation the *horizontal composition* of α and β:

```
β ∘ α :: G ∘ F -> G'∘ F'
```

Again, following Mac Lane I use the small circle for horizontal composition, although you may also encounter star in its place.

Here's a categorical rule of thumb: Every time you have composition, you should look for a category. We have vertical composition of natural transformations, and it's part of the functor category. But what about the horizontal composition? What category does that live in?

The way to figure this out is to look at **Cat** sideways. Look at natural transformations not as arrows between functors but as arrows between categories. A natural transformation sits between two categories, the ones that are connected by the functors it transforms. We can think of it as connecting these two categories.

Let's focus on two objects of **Cat** — categories C and D. There is a set of natural transformations that go between functors that connect C to D. These natural transformations are our new arrows from C to D. By the same token, there are natural transformations going between functors that connect D to E, which we can treat as new arrows going from D to E. Horizontal composition is the composition of these arrows.

We also have an identity arrow going from C to C. It's the identity natural transformation that maps the identity functor on C to itself. Notice that the identity for horizontal composition is also the identity for vertical composition, but not vice versa.

Finally, the two compositions satisfy the interchange law:

```
(β' · α') ∘ (β · α) = (β' ∘ β) · (α' ∘ α)
```

I will quote Saunders Mac Lane here: The reader may enjoy writing down the evident diagrams needed to prove this fact.

There is one more piece of notation that might come in handy in the future. In this new sideways interpretation of **Cat** there are two ways of getting from object to object: using a functor or using a natural transformation. We can, however, re-interpret the functor arrow as a special kind of natural transformation: the identity natural transformation acting on this functor. So you'll often see this notation:

```
F ∘ α
```

where F is a functor from D to E, and α is a natural transformation between two functors going from C to D. Since you can't compose a functor with a natural transformation, this is interpreted as a horizontal composition of the identity natural transformation 1_F after α.

Similarly:

```
α ∘ F
```

is a horizontal composition of α after 1_F.

Conclusion

This concludes the first part of the book. We've learned the basic vocabulary of category theory. You may think of objects and categories as nouns; and morphisms, functors, and natural transformations as verbs. Morphisms connect objects, functors connect categories, natural transformations connect functors.

But we've also seen that, what appears as an action at one level of abstraction, becomes an object at the next level. A set of morphisms turns into a function object. As an object, it can be a source or a target of another morphism. That's the idea behind higher order functions.

A functor maps objects to objects, so we can use it as a type constructor, or a parametric type. A functor also maps morphisms, so it is a higher order function — `fmap`. There are some simple functors, like `Const`, product, and coproduct, that can be used to generate a large variety of algebraic data types. Function types are also functorial, both covariant and contravariant, and can be used to extend algebraic data types.

Functors may be looked upon as objects in the functor category. As such, they become sources and targets of morphisms: natural transformations. A natural transformation is a special type of polymorphic function.

Challenges

1. Define a natural transformation from the `Maybe` functor to the list functor. Prove the naturality condition for it.
2. Define at least two different natural transformations between `Reader ()` and the list functor. How many different lists of `()` are there?
3. Continue the previous exercise with `Reader Bool` and `Maybe`.
4. Show that horizontal composition of natural transformation satisfies the naturality condition (hint: use components). It's a good exercise in diagram chasing.
5. Write a short essay about how you may enjoy writing down the evident diagrams needed to prove the interchange law.
6. Create a few test cases for the opposite naturality condition of transformations between different `Op` functors. Here's one choice:

```
op :: Op Bool Int
op = Op (\x -> x > 0)
```

and

```
f :: String -> Int
f x = read x
```

Next: Declarative Programming.

Acknowledgments

I'd like to thank Gershom Bazerman for checking my math and logic, and André van Meulebrouck, who has been volunteering his editing help.

59 Responses to "Natural Transformations"

1. benjaminy Says:

   April 11, 2015 at 1:13 am
   Context: I've been skimming each of your CT posts so far, but not doing the exercises. I'm a computer scientist who has been happily (off and on) programming in a basic to intermediate functional style for about a decade and a half.

I've been mildly curious about CT for years, but I'm still struggling to see how framing programming constructs in CT jargon helps engineers write substantially better programs. To my eye the book you're working on here very much follows in the tradition of other writings on CT, by which I mean that you describe the interesting (and perhaps beautiful) connections between programming concepts like functions, algebraic data types, … and CT concepts. But I'm not seeing how understanding those connections helps one write better programs. I'm very curious about this because I know there are very smart people who are convinced that there's software engineering gold to be found in CT. Can anyone help the scales fall from my eyes?

I was hoping to find some examples in these (draft) pages that looked like: here's a challenge that comes up in programming. Here's a good solution without CT. Here's what we can do when we apply CT thinking. So much better! Hooray! I am just not seeing it?

2. Bartosz Milewski Says:

April 12, 2015 at 1:39 pm

If you are looking for the "killer app" of category theory — it's the monad. This is a concept straight from category theory, and it has already revolutionized functional programming and is quickly invading imperative languages as well. It provides control over side effects through the type system. Programming languages and methodologies can no longer afford to ignore side effects since the advent of concurrent and parallel programming. Hidden side effects have devastating effect on concurrency.

I'm talking from personal experience. I've been a C++ programmer for many years, and what finally pushed me into functional programming and CT was the frustration with concurrency bugs and the inability to take advantage of multicore hardware. I'm still watching the evolution of C++ and witnessing the struggles within the committee. One bad decision after another is made because of the fear of the monad. C++ futures are a perfect case study of how a design of a library can be botched for lack of understanding of CT.

Other important influences from CT came through algebraic data types, initial and final algebras, catamorphisms, and lenses (see my post about the Yoneda lemma and lenses). Eric Niebler's work on C++ ranges was influenced by our exchanges about F-algebras and monads.

3. benjaminy Says:

April 12, 2015 at 4:40 pm

Monadic patterns are certainly an interesting example. For what it's worth, I've managed to convince myself that the main idea for parallelism in mainstream programming should not be threads with carefully controlled/managed memory effects, but rather processes. Starting from isolated memory parallelism and refining from there just seems so much more sane to me. And it's far more compatible with conventional software engineering patterns/practices than getting the whole world to switch to a functional style.

Your other examples:

ADTs are pretty obviously great. However, I wasn't aware of either a historical connection (was CT the inspiration for early ADT work?) or any reason why programmers should think in terms of CT to use ADTs.
I plead ignorance WRT how initial and final algebras and catamorphisms provide advantages over

more prosaic ways of thinking about iteration/folding/…
Lenses are also interesting, but again I'm not clear on exactly how thinking about them in CT terms makes them easier/better/whatever.

From the examples I know it seems like CT can be useful for programming language designers and designers of libraries that are so core that it's a bit hard to distinguish them from the language itself. The extreme abstraction of CT makes sense to me here because languages have to be adaptable to so many different applications. However, the vast majority of programmers will never design a programming language or core ecosystem library in their lives. I'm still failing to see what the relevance of CT is to these programmers.

4. Bartosz Milewski Says:

April 12, 2015 at 6:45 pm
If we divide the world into language designers, core library designers, utility library designers, and application programmers, then there indeed is a gradation of how much mileage can be gotten from knowing category theory at each level. You can be an excellent application programmer and know nothing about CT, or lambda calculus — or, on the other end of the spectrum, processor architecture. We know that.

5. owen Says:

May 3, 2015 at 4:20 pm
This is so confusing I think it will only work if it is its own language and syntax.

6. Steve Says:

May 4, 2015 at 10:33 pm
I read all the chapters and as a result, developed some intuitions how CT related to functional programming. I understand CT is object and arrow and that I should be able to think of them in various order of abstraction. What is confusion to me is when to call something a category and when I can not call something a category. I suppose when a type has a function that transforms its type to another type, it is a category? If this is the case, then I must have encountered many many categories. It seems silly to call a type with morphism to another type a category as this definition fit so commonly in programs. Would you elaborate when one should call something a category?

7. Bartosz Milewski Says:

May 4, 2015 at 10:52 pm
You can call something a category if you can identify objects and morphisms. The only requirement is that morphisms compose, the composition is associative, and that there are identity morphisms. There's nothing more to it. In programming we identify types as objects and functions as morphisms, but there are many other categories. I gave several examples in the early chapters.

8. greg nwosu (@buddistfist) Says:

July 22, 2015 at 9:58 pm
whats the difference between
alpha :: forall a . F a -> G a
and

alpha :: F a -> G a
a is a type variable so i dont see the semantic difference , can you explain?

9. Bartosz Milewski Says:

July 23, 2015 at 12:30 pm
@buddistfist: No difference in Haskell. Any top level type variable is automatically universally quantified. I make it explicit for pedagogical reasons.

10. Paul Harrison Says:

January 7, 2016 at 12:44 am
I'm stuck on something.

You say "under any natural transformation between F and G, F f must be transformed into G f". However, I can not see how to obtain an equation for G f in terms of F f. We only have G f . alpha a = alpha b . F f. This seems weaker, and I don't have an intuitive grasp of it.

11. Bartosz Milewski Says:

January 10, 2016 at 3:48 pm
What I meant (and maybe I should have been more explicit about it) is that you don't have to define the action of natural transformations on morphisms, because you really don't have a choice. Both F f and G f are given — they are part of the definition of functors F and G. Naturality condition for `f::a->b` narrows down your choices for α_a and α_b.

12. musa al-hassy Says:

February 12, 2016 at 5:33 pm
Please elaborate on "you really don't have a choice" 🙂

13. Bartosz Milewski Says:

February 12, 2016 at 10:57 pm
Look closely at the naturality diagram. It contains all the objects and morphisms that are related to f. Two of the morphisms define components of the natural transformation, so they are not good candidates for the image of f. Two other two are images of f under F and G. One involves only F, the other only G. The only thing that's not taken is the composite morphism from F a to G b. But this morphism is unique only if the two compositions are equivalent. And that's exactly the naturality condition. So, if you want, you can call this morphism the image of f under natural transformation. But it's existence is the result of the naturality condition.

14. Dipucian (@Dipucian) Says:

March 31, 2016 at 7:06 pm
I'm confused that you said: "In fact they are members of two different functor categories: F'^F and G'^G.", shouldn't that be categories D^C and E^D?

15. Bartosz Milewski Says:

April 1, 2016 at 10:59 am
@Dipucian : Thanks for paying attention. Fixed!

16. greenbergjosh Says:

June 15, 2016 at 1:48 pm
I have been viewing Catsters videos on youtube as well as reading your blog. I run into difficulty with some notation when I arrive at monads (their Monads 1 video). I see you are currently writing that section, so I have chosen to post this here since it is really a question about composition of natural transformations and functors.

In particular, there are diagrams for the unit triangles for eta (the unit function in the monad).

Let T be the functor.
Let η be the unit function
Let η_X be the x-component of η.

The unit triangle can be labeled (leaving out the mu part)
Tx — $T\eta_X$ –> T^2x
And in the opposite direction
Tx – η_{Tx} –> T^2x

I know that an eta-component is a function from an object (in one category) to a morphism (in another category, though the two categories are the same in the monad).

So, if eta is a function from an object to a morphism.
And, T is a functor, which maps objects to objects and morphisms to morphisms.
What does it mean $T\eta_X$?

If I am starting from Tx, as in the unit triangle, then I am starting from an object in the destination category, namely, T x. If I apply η_X, the x-component of eta, to that object (to T x), I must get back a morphism, since eta maps objects to morphisms. So, if I then apply T to that morphism, I get back a morphism. T^2x, however, is not a morphism, it is an object.

This makes no sense to me. Honestly, the greatest thing in the world would be an internal diagram at the level of me, a dummy. I really like the catster selection of using letters and lists, but I would love to see an internal diagram that actually shows what is getting mapped to what.

I believe the set would contain all letters as well as all lists of letters. But, I will leave that as part of my question, because I am really not certain.

Many thanks and kind regards.

17. Bartosz Milewski Says:

June 16, 2016 at 6:54 am
η is a natural transformation from the identity functor to the functor T. Its component at an object x is a morphism from x to Tx. (Tx or T x is the action of T on the object x.)

$T\eta_X$ is the action of the functor T on the morphism η_X. It is therefore a morphism from Tx to T(Tx).

η_{TX} is the component of η at the object Tx. It is therefore a morphism from Tx to T(Tx).

T^2x is notation for T(Tx).

18. greenbergjosh Says:

June 16, 2016 at 10:39 am

Thank you for that clarification. It definitely helped me see how the diagram works. However, while I can now see that the composition makes sense, I still can't piece together the building from one level of abstraction to the next. I read Lawvere, and a few others, but the map from something concrete to the abstractions is never clear.

The category used for the monad of monoids in the Catster video is the category of sets. I believe that means the objects are sets and the morphisms are functions between sets (with composition and identity, and composition being associative).

I would like to picture in my mind one of the objects in the category being the set of symbols and another being the set of lists of symbols. Then, the identity morphism is clearly just each object in the set of symbols mapped to itself, and likewise for the set of lists of symbols. I can also picture a morphism from the set of symbols to the set of lists of symbols. Something like lifting, or unit, that maps 3->[3]. However, I am not certain if I am doing this at the right level of abstraction.

My next step would be to define a functor. In this case, the functor would have to go from the category of sets and morphisms between them, to itself. So, I have to map the set of symbols to some other set. If I map it to itself, and likewise for every other set, then I have the identity functor, assuming I do the same for every morphism. So, what is a functor other than the identity functor?

I start by thinking about an endomorphism in the original category, like successor, so that 3->4, 4->5, etc. Now, when I think about my functor, I can picture the set of symbols object mapping to the set of lists of symbols object. And, the endomorphism 3->4, 4->5, mapping to the endomorphism [3]->[4], [4]->[5]. That gives me the nice commutative diagram for the functor. I can travel from 3 to 4 first then over to [4], or I can travel from 3 to [3], and then to [4]. So, now I have a reasonable functor. If I wanted to use something other than an endomorphism, I guess I could create sets of different types of boxes. So, x->[x] in the source category, and (x)->{x} in the destination category, and then the functor would map x to (x) and [x] to {x}, and the morphisms, accordingly to commute.

Here is where I get confused. I think of the functor as mapping objects to objects. It mapped the set of symbols to the set of lists of symbols. It mapped x to [x]. But, it did not map each element of x to each element of [x]. That is, it did not map 3 to [3]. To me, the object, set of symbols, has internal structure, namely the symbols. I know we want to abstract past that, but I also know we want to retain certain properties. So, when we say that the functor maps an object (a set in this case) to another object (also a set in this case), what does that mean with respect to the internal structure of those sets? I see how a morphism can map 3->[3] in one category, and another morphism can map (3)->{3} in the other category. Now, when I put a functor between them, I need the functor to map 3 to (3) and [3] to {3}. How does it do that? Let me try to say it a different way. I know that 3 is a symbol in the set of symbols. If I have a morphism from 3 to [3], I understand that. If I have a functor, it does not act on the internal structure of the set of symbols; it acts on the set as a whole (the object). So, it does not

make sense
for me to say F 3 (where F is a functor). It does make sense for me to say F SetOfSymbols. In fact, we have said F SetOfSymbols -> ListOfSymbols.

So, if I start with 3, I cannot apply F, its at the wrong level of abstraction. But, I would like to start with 3, and then build up all of the concepts from there. I know there is an object that contains 3 and I know I can apply F to that object, but how does that help? I want to start from 3.

Is the functor making any statement about 3 mapping to (3), or [3] mapping to {3}? If not, then it is only making a statement about structure. And, maybe that is exactly the point. I'm just not sure about that. Since the functor also insists on retaining the identity morphism and the associativity of morphisms, it is stating that there is a shared pattern between the two categories. However, am I right in assuming that I can't really get anywhere starting from 3? Instead, I can only say that if I start from 3 in one category, and at the same time start from (3) in another category (even if it is the same category as in the case of the monad) then I can follow an analogous path, in lock step, and end up at an analogous place. But, I worry, that there is nothing to stop me from starting at 3 in one category and ending at [3], while at the same time starting at (3) in the other category but ending at {4}. After all, the Functor does not map 3 to (3), it just says there is an analogous path between objects in two different categories. So, what stops me from ending up at the right object, but at the wrong element of that object?

Wait, I think it is the fact that the functor has to map the identity morphisms in one category to the identity morphisms in the other category coupled with the associativity requirement. So, those two requirements on the functor guarantee that if we start at 3 in one category and (3) in the other category, then we will end at [3] and {3} respectively. It does this without the functor ever acting directly on 3.

So, the functor does not work on 3, but it works on, say Int32, and says that if we have a morphism from Int32 to Float32 then we also have a analogous morphism from [Int32]->[Float32]. And, further that the functor must guarantee that if the first morphism takes us from 3 to 3.0, then when we apply the functor to that morphism it will give us a function that takes us from [3] to [3.0].

But, nothing says how to write that functor. It's not as if we can derive it out of thin air. We have only stated what properties it must have, and what it's existence implies, not how to make one.

I realize this is a somewhat tedious way to ask the question, but I believe it is a stream on consciousness that might be shared by many learners. In short my questions are:

Is it correct that F never acts on 3 and is at a completely different level of abstraction?
Is it correct that existence of F is a statement about analogous structure of two categories?
Is it correct that it is the identity and associativity rules of F that guarantee that if 3 takes us to [3], then (3) takes us to {3}?
Is it correct that none of this helps with the construction of F?

19. Bartosz Milewski Says:

June 17, 2016 at 12:50 am
Your intuitions are mostly right. However, a functor does not map individual elements of sets.

Abstraction is about letting go. Don't think of individual elements. A functor is just a mapping of objects and morphisms. Morphisms are just arrows between objects.

20. Joshua Greenberg Says:

June 17, 2016 at 6:13 am
All due respect, I got the part that a functor does not map individual objects – that is exactly that F never acts on 3, in my examples. I would really like an answer to my question, but I realize I was too verbose. I will be extremely brief and specific here.

At this link
http://www.euclideanspace.com/maths/discrete/category/principles/functor/index.htm

there is an example of a functor – example 2.

The functor is called List. In the first picture, List is mapping objects in C to objects in List(C).

In the second picture, List is mapping a morphism in C to a morphism in List(C). But, wait, the morphism is between elements of int. This is not right, it should be between objects of C not elements of int, but everyone does it this way.

A functor maps objects and morphisms from one category to another. When I select the granularity of my objects to be (int, bool, word, etc), then I should select my granularity to be morphisms between those objects (not elements of those objects, not their internal structure). Otherwise, I have not "let go".

But, they show List mapping a morphism from an element of int to another element of int. It's at the wrong granularity. What am I missing?

In your Haskell examples, it's the same thing. You define functor at the level of type, but then use, x, an element of the internal structure of the type, in the definition (instance) of the functor. It's mixing granularities. If the functor "let's go" and doesn't know about internal structure, then why does internal structure appear in the definition of the functor?

21. Bartosz Milewski Says:

June 17, 2016 at 10:19 am
There is this one special category called **Set** that is derived from set theory. In it objects are sets and morphisms are functions. Because we can look at those morphisms as functions, we can split them into components, that is values at particular points — the x's. In effect we are switching between categorical and set-theoretical views. Think of set theory as the assembly language of category theory. I don't know how else to explain it. I mention this "split personality" character of **Set** in the section dedicated to free/forgetful adjunctions.

22. jonszingale Says:

August 10, 2016 at 3:04 pm
I love the exposition and the water colors.

23. edmundsecho Says:

February 2, 2017 at 4:32 pm
The hackage documentation for Control.Monad.Trans.Class (https://hackage.haskell.org/package/transformers-0.5.2.0/docs/Control-Monad-Trans-Class.html#v:lift), describes `lift` as a natural transformation. So if `t:: M a -> N a` and `mapSateT t :: StateT s M a -> StateT s N a` , then `lift` is defined such that `mapStateT t . lift = lift . t` . However, when lift is used in code, it's used to lift kleisli arrows from the base monad (e.g., lift . putStrLn). Kleisli arrows are `a -> m b` . Structure changes but not in a way that is naturally clear for `StateT s M a` . Kleisli arrows also transform the "payload" `a -> b` which is not the case for `t` . The question: Can you explain how `lift` is a natural transformation when it is composed with kleisli arrows? Also, what might be a concrete example of `M a -> N a` described in the Trans document? Thanks in advance for your thoughts on this. – E

24. Bartosz Milewski Says:

February 2, 2017 at 11:27 pm
The documentation talks about the category of monads. Monads are (endo-) functors, so it's like a restricted functor category. In this category, objects are monads and morphisms are natural transformations between them.

For instance, given two monads M and N, a morphism between them is a natural transformation:

```
t :: forall a. M a -> N a
```

Examples of M and N? How about the list monad and the Maybe monad. A good example for `t` would be `safeHead`.

An (endo-) functor in the category of monads will act on monads and morphism between monads. For instance, `StateT s` (for a given s) takes any monad `M` and produces another monad `StateT s M`. That's the action of a functor on objects (monads, in this case). It's action on morphisms (natural transformations, in this case) is given by `mapStateT`. For instance, given the transformation `t` above, we get:

```
mapStateT t :: forall a. StateT s M a -> StateT s N a
```

It takes one natural transformation, `t`, and produces another. So `StateT s` is indeed a functor in the category of monads.

So that's a functor in the category of monads. What's a natural transformation in this category? A natural transformation is a family of morphisms indexed by an object. So it would be a family of natural transformations between monads indexed by a monad. Let's look at `lift`:

```
lift :: m a -> t m a
```

It's an `a`-component of a natural transformation between two functors (really monads), `m` and `t m`. We can look at it as the `m`-component of a natural transformation between two functors-on-monads. We know that `t` is such a functor. We don't see the corresponding functor on the left hand side, but that just means that it's the identity functor or, in this case, the identity monad transformer. It takes a

monad and returns it back. If we call it IdT, we can write `lift` as a monad-natural-transformation between `IdT` and `t`, taken at point `m`, which gives a regular natural transformation taken at point `a`. It could be written as:

```
forall m. forall a. IdT m a -> t m a
```

When you use a natural transformation in Haskell, you always use a component of it, and you never specify at which object you're taking it. The compiler figures it out. With `lift`, it figures out both the monad `m` and the object `a`. So if you write:

```
lift . putStrLn
```

`m` is taken to be `IO`, and `a` is taken to be `()`. Once `m` and `a` are fixed, it's just a regular function that can be composed with another function.

25. edmundsecho Says:

February 3, 2017 at 8:02 am
This is great.

I understand how examples of `M` and `N` might be `[]` and `Maybe` where `t` might be `safehead`. However, this is not what `lift` ends up lifting. As you ended, `lift` maps (not the function, rather as in: it *is* a map) `IO ()` to `StateT s IO ()`. This map is a natural transformation which in this case is more specifically an endofunctor in the category of monads. To link to what you drew in this blog episode:

Is `lift` an example of α`() :: IO () -> StateT IO ()` where `b :: ()`?

This matters to me because I need to describe α`a` . I'm ok conceptually assuming `a :: String`, but then the meaning of the transformation `M -> N` is lost where it should be `M -> M`.

PS: Perhaps you explained this with

```
forall m. forall a. IdT m a -> t m a
```

But I'm not sure we need to include the concept of Identity, because `m` and `StateT m` are both monads, each a single construct (the latter is a composite, but a single layer nonetheless).

Thank you advance for helping me to continue to peel this onion.

26. Bartosz Milewski Says:

February 3, 2017 at 10:55 am
A natural transformation is a family of morphisms indexed by an object. Here, morphisms themselves are natural transformations. So after indexing the outer natural transformation `lift` with the monad `m`, you still have to index the result by the type `a`. Then you get a regular function. So strictly speaking, one should write:

$(lift_m)_a$

27. edmundsecho Says:

February 3, 2017 at 2:15 pm
This is helpful. I have to spend more time to internalize the first two sentences of your response but I think I'm starting to catch-up.

I think it's safe to write:

```
" lift"[]" .  safeHead"a" :: [a] -> StateT s Maybe a "
```

… to capture the type of the composition, which is as you have said, what's all about (which is a kleisli). If I analogize what I learned with `safeHead` to `lift . putStrLn` , for it to meet the criteria for endofunctors in the category of monads, we have to find a way to construct a monad from `String` , even say an `IO String` , *before* returning `IO ()` . Or we could include in our category of monads an object that captures what is not a monad, but can become one with application of a functor (say `Identity`).

All this to say, do you think the hackage documentation's use of the categorical terms is capturing the story, especially since we more often lift kleisli arrows that take us from non-monads to the base monad? How would you have described it?

28. edmundsecho Says:

February 6, 2017 at 9:07 am
I figured it out. I was missing the now clear distinction between `lift` and `mapXXXT`. The first generates kleisli arrows that type with the option to change both the structure and payload (`a -> m b`). Lifting `safeHead` exploits the option to change structure, not payload. This in contrast to what happens when we lift `putStrLn`. This is all done with composition `(.)`.

mapXXXT: We can generate a function that operates in the new context using partial application. The rules are more rigid here. e.g., `SateT s M a -> StateT s N a` implies we can only change the base monad. All else must remain constant. One of the "so whats" for me: getting IO functions to fit here is generally more difficult: changing to or from IO, in a polymorphic way without fixing the data type is not obvious (and may be what makes IO special).

Thank you for your help and patience.

29. hololeap Says:

November 1, 2017 at 4:03 pm
I really have to ask, is exercise #5 meant to be a joke?

30. Robert Peszek Says:

December 3, 2017 at 6:08 pm
I would like to share this observation about natural transformations and non-Hask categories.

Standard Category (Control.Category) typeclass is kind polymorphic allowing for creation of exotic categories using DataKinds and GADTs.
One way to do that is use GADT to simply enumerate all possible morphisms. This way I can write code representing a simple category based on some directed graph.

`type f ~> g = forall x. f x -> g x` is also kind-polymorphic (`PolyKinds`) and infers `f:: k -> *, g:: k -> *`. This is all great, except in this general case I lose naturality free theorem. For small `k`-s, naturality condition becomes a proof obligation.

https://github.com/rpeszek/notes-milewski-ctfp-hs/wiki/N_P1Ch10b_NTsNonHask

I still find this approach very useful as it allows me to play programmatically with various examples of categories.
That begs the question: maybe there is a range of categories where free naturality theorem is true, not just Hask? I do not know. And this feels like a hard question too.

31. Bartosz Milewski Says:

December 4, 2017 at 1:57 am
Free theorems are the result of parametricity, which is a property of the language rather than a category. In simple words, parametricity means that we define a polymorphic function using a single formula for all types. As soon as you allow pattern-matching on types, you lose parametricity.

32. Robert Peszek Says:

December 4, 2017 at 8:17 am
Bartosz,
Hmm, I did not think about it this way. Thank you for your answer.
These functions are polymorphic on kind `MyCat.Object`
`forall (x ::`MyCat.Object`) . f x -> g x`
There is no pattern matching on these types. I have not implemented singletons for them
and even if I did, singletons are not part of these functions.
However, since the function space is so limited (enumerated in GADT), being able to pattern match on morphisms effectively does the same thing. I can recover type information from that pattern match! I will include your explanation in my note.

33. Lifu Huang Says:

December 16, 2017 at 6:04 pm
Hi Bartosz, you mentioned the identity for horizontal composition is the identity natural transformation that maps the identity functor on C to itself. I can understand the identity is a identity natural transformation for endofunctors on C, but I cannot figure out why it must also be an identity functor, can you show me how I can prove it? Thanks

34. Bartosz Milewski Says:

December 17, 2017 at 4:06 am
Here's how I would go about proving it: First, you can easily see that the identity natural transformation on identity functors is indeed the identity for horizontal composition. Then use the uniqueness of identity. Suppose that you have two such identities: compose them horizontally. What do you get?

35. Jeff Brown Says:

April 7, 2018 at 12:57 pm

This book is fantastic.

Why is the last sentence in the following quote true? “[In Cat] an object in a product category C × D is just a pair of objects, (c, d), one from C and one from D. Similarly, a morphism between two such pairs, (c, d) and (c′, d′), is a pair of morphisms, (f, g), where f :: c -> c′ and g :: d -> d′.”

I’m guessing it has something to do with the fact that morphisms in Cat are functors, but I’m not seeing it.

36. Bartosz Milewski Says:

April 8, 2018 at 2:28 am
This is the definition of a product category. It specifies object and morphisms — the composition is obvious and you can easily check unit and associativity laws.

37. 自然变换 - 奇奇问答 Says:

June 12, 2018 at 5:24 am
[…] 原文：https://bartoszmilewski.com/2… […]

38. Bogdan Popescu Says:

October 7, 2018 at 12:09 am
Hello! I have a question if someone could help me…
In Functoriality chapter I encountered the following definition that seemed very clear
type Op r a = a -> r

In this post I see this
newtype Op r a = Op (a -> r)

They don’t seem to be the same. Op is a new type definition, but is Op () something Haskell specific? Sorry, I’m very new to the language. Thank you

39. Bartosz Milewski Says:

October 7, 2018 at 9:30 am
These two definitions are mathematically equivalent, but they interact differently with the Haskell type checker. The first one is just a synonym for the function type (a->r), so a function that accepts (Op r a), will also gladly accept a function (a->r). If you use the second definition, it won’t! It will consider these two types different and you’ll have to explicitly convert one to another. I had to switch definitions because Haskell will complain if I use type synonyms in class instantiations.

40. Alby Says:

January 2, 2019 at 9:59 pm
I am trying to draw the naturality diagram (third) figure in the blog for the example where F and G are List and Maybe functors and the morphism is safeHead. I am running into some confusion …

If I understand correctly safeHead is really the morphism in the ‘target’ category (Fa -> Ga).

Now, identifying the object a, in the 'source' category with '(x:xs)' seems to be meaningless. Any suggestion? Is it even possible to do draw such parallel?

Thanks a lot in advance!!!

41. Alex Says:

February 4, 2019 at 4:06 pm
Hi Bartosz, thank you for the very helpful series and lectures.

I understand from your word on notation that you are using · (dot) for vertical composition and ◦ (small circle) for horizontal composition. I find the use of that notation intuitive in interchange law

$(\beta' \cdot \alpha') \circ (\beta \cdot \alpha) = (\beta' \circ \beta) \cdot (\alpha' \circ \alpha)$

but I'm puzzled by the use of ◦ in this equation:

$H\ f \circ (\beta \cdot \alpha)a = (\beta \cdot \alpha)b \circ F\ f$

Is ◦ somehow meant to denote horizontal composition in this case too?

Thanks for your help.

42. Bartosz Milewski Says:

February 4, 2019 at 4:45 pm
No, it doesn't. Unfortunately, the circle is overloaded and you have to analyze the context. When acting on natural transformations, it means horizontal composition. When acting on morphisms, it means morphism composition. Since components of natural transformations are regular morphisms, they are composed using a circle. I know it's confusing, but I had to be consistent with mathematical literature.

43. Alex Says:

February 5, 2019 at 2:06 pm
For the statement of interchange law

$(\beta' \cdot \alpha') \circ (\beta \cdot \alpha) = (\beta' \circ \beta) \cdot (\alpha' \circ \alpha)$

are the definitions for α and β meant to be the same as those from the running example earlier in the "2-Categories" section?

I thought $\beta \cdot \alpha$ meant "the vertical composition of β after α." But I'm wondering how that could be since "we cannot apply vertical composition to this pair, because the target of α is different from the source of β."

Thanks for your help, Bartosz 🙂

44. Bartosz Milewski Says:

February 5, 2019 at 5:31 pm
The target of alpha is the same functor as the source of beta

45. Al Says:

    April 24, 2019 at 10:34 am
    What is the object and what is the morphism when you say “Looking at a natural transformation component-wise, one may say that it maps objects to morphisms.” ? Would you please explain it using the picture bellow aforementioned sentence in your blog? Thanks a lot. 🙂

46. Al Says:

    April 24, 2019 at 11:17 am
    Regarding natural transformation being a polymorphic function, would you please verify if I got it right or not as follow:

    The functors will act sort of like an interface or mediator so that all natural transformation needs to know is about functors(interface) and not the original objects(types) and so that is why natural transformation can act on variety of types because the duty of dealing with each type is delegated to functors and all natural transformation needs to know is how to deal with those two functors.
    For example if we have functors F and G as follow:

    F :: Int -> String
    G :: Int -> Bool

    All natural transformation alpha needs to know is how to deal with String and Bool and not Int, hence it can also work with two functors that transform List to String and Bool respectively:

    List -> String
    Bool -> String.

    Am I correct?

    Thanks you in advance for your time and help 🙂

47. Bartosz Milewski Says:

    April 24, 2019 at 11:24 am
    Look at the second picture in this blog (one with the clouds). For every object a you have a morphism α_a.

48. Bartosz Milewski Says:

    May 6, 2019 at 5:22 pm
    @Al: What you are calling functors F and G looks more like functions.

49. Bartosz Milewski Says:

    June 14, 2019 at 10:40 am
    Note about notation: The use of a circle for horizontal composition of natural transformations makes sense if you consider that it’s a natural transformation between composed functors–the functors being composed using the circle.

    $\beta \circ \alpha$ :: G ∘ F -> G′ ∘ F′

50. 
František Řezáč (@calaverainfo) Says:

    August 6, 2019 at 1:49 am
    The case of a Const functor left me with a question what is the relation between natural transformation and well known fold functions. Can I say that any use of fold (or reduce) *is* a natural transformation?

51. 
Bartosz Milewski Says:

    August 6, 2019 at 6:31 am
    Good question, but I don't think so. Fold is a catamorphism, and it's defined for an F-algebra f b -> b. An algebra is specifically *not* a polymorphic function. In particular, you can see that the type parameters in fold occur both in covariant and contravariant positions.

52. 
Siyuan Chen (@chansey97) Says:

    August 21, 2019 at 11:45 am
    At #13, you mentioned that "But this morphism is unique only if the two compositions are equivalent. And that's exactly the naturality condition."

    Could you explain the keyword "unique" further ?

    IMO, given two functor F and G, there may be a lot of natural transformations between them.

    For example:
    In Challenges 2, I defined 3 natural transformations for (Bool -> a) -> Maybe a

    ```
    nat1:: (Bool -> a) -> Maybe a
    nat1 _ = Nothing

    nat2:: (Bool -> a) -> Maybe a
    nat2 g = Just (g True)

    nat3:: (Bool -> a) -> Maybe a
    nat3 g = Just (g False)
    ```

    Thanks a lot.

53. 
Bartosz Milewski Says:

    August 21, 2019 at 3:33 pm
    There is a whole set of natural transformations between any two functors (it could be empty, though).

    The diagonal morphism in the naturality square is only unique if the square commutes. Otherwise the two compositions would be different.

54. 
Siyuan Chen (@chansey97) Says:

    August 24, 2019 at 10:17 am

Thank you for your explanation.
But I still have some confusion about natural transformation, so I reorganize the language and elaborate as follows:

Consider that we have a category C with only two objects a and b, but there are 3 different morphisms f1 f2 f3 between a and b.
We have another category D, this D can be an arbitrary category (maybe a Set).
We also have two functors F and G between categories C and D:
F map objects a and b in C to Fa and Fb in D
F map morphisms f1 f2 f3 in C to Ff1 Ff2 Ff3 in D
G map objects a and b in C to Ga and Gb in D
G map morphisms f1 f2 f3 in C to Gf1 Gf2 Gf3 in D

Note that since D is an arbitrary category (e.g. a Set), there may be many morphisms between Fa -> Ga and many morphisms between Gb -> Gb

My question is:
What is the mean about "pick one natural transformation α :: F->G" ?
There are two options for me:
(1) pick two morphisms (called components) αa :: Fa->Ga and αb :: Fb->Gb in D, such that
$\exists f \in \{f1,f2,f3\}$ satisfy $\alpha b \cdot Ff = Gf \cdot \alpha a$
(2) pick two morphisms (called components) αa :: Fa->Ga and αb :: Fb->Gb in D, such that
$\forall f \in \{f1,f2,f3\}$ satisfy $\alpha b \cdot Ff = Gf \cdot \alpha a$

Which statement is correct?
Thanks.

55. Bartosz Milewski Says:

September 10, 2019 at 4:41 pm
Maybe it would be clearer if I said "consider any morphism f". The second option is correct.

56. Tom Says:

September 29, 2019 at 2:56 am
I'm really enjoying your book! It was just the content I wanted and I really wish more maths books were written in this style. Teaching content and teaching mathematical rigor should be separated imo.

Maybe you should mention that parametric polymorphism implies naturality is nontrival. I got mislead they way you've written it

57. Functors and Vectors - Hey There Buddo! Says:

October 14, 2019 at 1:10 pm
[…] mappings correspond in categorical terminology to natural transformations between the functors f and g. There is a category where objects are Functors and morphisms are […]

58. Joe Square Says:

May 10, 2020 at 11:48 pm
I really love the illustrations. You draw them yourself?

59. Bartosz Milewski Says:

May 11, 2020 at 7:37 am
Yes