Bartosz Milewski's Programming Cafe

Category Theory, Haskell, Concurrency, C++

November 17, 2015

It's All About Morphisms

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

i
19 Votes

This is part 17 of Categories for Programmers. Previously: Yoneda Embedding. See the Table of Contents.

If I haven't convinced you yet that category theory is all about morphisms then I haven't done my job properly. Since the next topic is adjunctions, which are defined in terms of isomorphisms of hom-sets, it makes sense to review our intuitions about the building blocks of hom-sets. Also, you'll see that adjunctions provide a more general language to describe a lot of constructions we've studied before, so it might help to review them too.

Functors

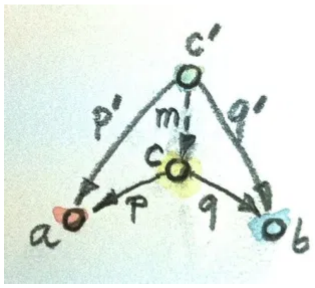
To begin with, you should really think of functors as mappings of morphisms — the view that's emphasized in the Haskell definition of the `Functor` typeclass, which revolves around `fmap`. Of course, functors also map objects — the endpoints of morphisms — otherwise we wouldn't be able to talk about preserving composition. Objects tell us which pairs of morphisms are composable. The target of one morphism must be equal to the source of the other — if they are to be composed. So if we want the composition of morphisms to be mapped to the composition of *lifted* morphisms, the mapping of their endpoints is pretty much determined.

Commuting Diagrams

A lot of properties of morphisms are expressed in terms of commuting diagrams. If a particular morphism can be described as a composition of other morphisms in more than one way, then we have a commuting diagram.

In particular, commuting diagrams form the basis of almost all universal constructions (with the notable exceptions of the initial and terminal objects). We’ve seen this in the definitions of products, coproducts, various other (co-)limits, exponential objects, free monoids, etc.

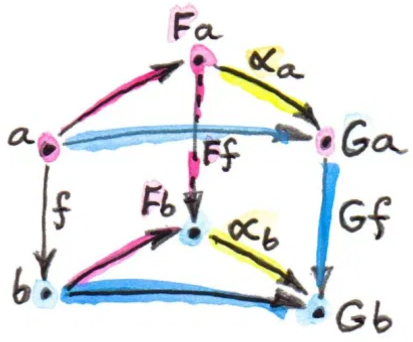
The product is a simple example of a universal construction. We pick two objects `a` and `b` and see if there exists an object `c`, together with a pair of morphisms `p` and `q`, that has the universal property of being their product.



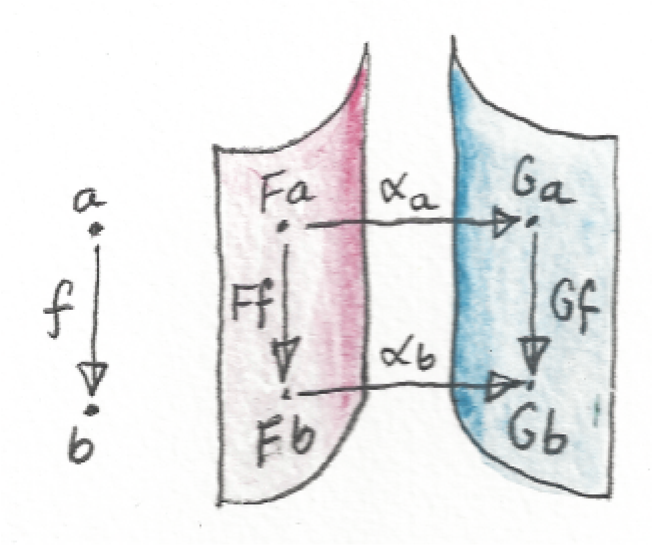
A product is a special case of a limit. A limit is defined in terms of cones. A general cone is built from commuting diagrams. Commutativity of those diagrams may be replaced with a suitable naturality condition for the mapping of functors. This way commutativity is reduced to the role of the assembly language for the higher level language of natural transformations.

Natural Transformations

In general, natural transformations are very convenient whenever we need a mapping from morphisms to commuting squares. Two opposing sides of a naturality square are the mappings of some morphism `f` under two functors `F` and `G`. The other sides are the components of the natural transformation (which are also morphisms).



Naturality means that when you move to the “neighboring” component (by neighboring I mean connected by a morphism), you’re not going against the structure of either the category or the functors. It doesn’t matter whether you first use a component of the natural transformation to bridge the gap between objects, and then jump to its neighbor using the functor; or the other way around. The two directions are orthogonal. A natural transformation moves you left and right, and the functors move you up and down or back and forth — so to speak. You can visualize the *image* of a functor as a sheet in the target category. A natural transformation maps one such sheet corresponding to F, to another, corresponding to G.



We’ve seen examples of this orthogonality in Haskell. There the action of a functor modifies the content of a container without changing its shape, while a natural transformation repackages the untouched contents into a different container. The order of these operations doesn’t matter.

We’ve seen the cones in the definition of a limit replaced by natural transformations. Naturality ensures that the sides of every cone commute. Still, a limit is defined in terms of mappings *between* cones. These mappings must also satisfy commutativity conditions. (For instance, the triangles in the definition of the product must commute.)

These conditions, too, may be replaced by naturality. You may recall that the *universal* cone, or the limit, is defined as a natural transformation between the (contravariant) hom-functor:

```
F :: c -> C(c, Lim D)
```

and the (also contravariant) functor that maps objects in *C* to cones, which themselves are natural transformations:

```
G :: c -> Nat(Δc, D)
```

Here, Δ_c is the constant functor, and `D` is the functor that defines the diagram in *C*. Both functors `F` and `G` have well defined actions on morphisms in *C*. It so happens that this particular natural transformation between `F` and `G` is an *isomorphism*.

Natural Isomorphisms

A natural isomorphism — which is a natural transformation whose every component is reversible — is category theory’s way of saying that “two things are the same.” A component of such a transformation must be an isomorphism between objects — a morphism that has the inverse. If you visualize functor images as sheets, a natural isomorphism is a one-to-one invertible mapping between those sheets.

Hom-Sets

But what are morphisms? They do have more structure than objects: unlike objects, morphisms have two ends. But if you fix the source and the target objects, the morphisms between the two form a boring set (at least for locally small categories). We can give elements of this set names like `f` or `g`, to distinguish one from another — but what is it, really, that makes them different?

The essential difference between morphisms in a given hom-set lies in the way they compose with other morphisms (from abutting hom-sets). If there is a morphism `h` whose composition (either pre- or post-) with `f` is different than that with `g`, for instance:

```
h ∘ f ≠ h ∘ g
```

then we can directly “observe” the difference between `f` and `g`. But even if the difference is not directly observable, we might use functors to zoom in on the hom-set. A functor `F` may map the two morphisms to distinct morphisms:

```
F f ≠ F g
```

in a richer category, where the abutting hom-sets provide more resolution, e.g.,

```
h' ∘ F f ≠ h' ∘ F g
```

where `h'` is not in the image of `F`.

Hom-Set Isomorphisms

A lot of categorical constructions rely on isomorphisms between hom-sets. But since hom-sets are just sets, a plain isomorphism between them doesn’t tell you much. For finite sets, an isomorphism just says that they have the same number of elements. If the sets are infinite, their cardinality must be the same. But any meaningful isomorphism of hom-sets must take into account composition. And composition involves more than one hom-set. We need to define isomorphisms that span whole collections of hom-sets, and we need to impose some compatibility conditions that interoperate with composition. And a *natural* isomorphism fits the bill exactly.

But what’s a natural isomorphism of hom-sets? Naturality is a property of mappings between functors, not sets. So we are really talking about a natural isomorphism between hom-set-valued functors. These functors are more than just set-valued functors. Their action on morphisms is induced by the appropriate hom-functors. Morphisms are canonically mapped by hom-functors using either pre- or post-composition (depending on the covariance of the functor).

The Yoneda embedding is one example of such an isomorphism. It maps hom-sets in *C* to hom-sets in the functor category; and it’s natural. One functor in the Yoneda embedding is the hom-functor in *C* and the other maps objects to sets of natural transformations between hom-sets.

The definition of a limit is also a natural isomorphism between hom-sets (the second one, again, in the functor category):

$$C(c, \text{Lim } D) \simeq \text{Nat}(\Delta_c, D)$$

It turns out that our construction of an exponential object, or that of a free monoid, can also be rewritten as a natural isomorphism between hom-sets.

This is no coincidence — we’ll see next that these are just different examples of adjunctions, which are defined as natural isomorphisms of hom-sets.

Asymmetry of Hom-Sets

There is one more observation that will help us understand adjunctions. Hom-sets are, in general, not symmetric. A hom-set `C(a, b)` is often very different from the hom-set `C(b, a)`. The ultimate demonstration of this asymmetry is a partial order viewed as a category. In a partial order, a morphism from `a` to `b` exists if and only if `a` is less than or equal to `b`. If `a` and `b` are different, then there can be no morphism going the other way, from `b` to `a`. So if the hom-set `C(a, b)` is non-empty, which in this case means it’s a singleton set, then `C(b, a)` must be empty, unless `a = b`. The arrows in this category have a definite flow in one direction.

A preorder, which is based on a relation that’s not necessarily antisymmetric, is also “mostly” directional, except for occasional cycles. It’s convenient to think of an arbitrary category as a generalization of a preoder.

A preorder is a thin category — all hom-sets are either singletons or empty. We can visualize a general category as a "thick" preorder.

Challenges

1. Consider some degenerate cases of a naturality condition and draw the appropriate diagrams. For instance, what happens if either functor `F` or `G` map both objects `a` and `b` (the ends of `f :: a -> b`) to the same object, e.g., `F a = F b` or `G a = G b`? (Notice that you get a cone or a co-cone this way.) Then consider cases where either `F a = G a` or `F b = G b`. Finally, what if you start with a morphism that loops on itself — `f :: a -> a`?

Next: Adjunctions.

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 throughout this series of posts.

Follow @BartoszMilewski

6 Responses to "It's All About Morphisms"

1. Jesus Prieto Colomina Says:

   December 16, 2015 at 2:53 pm
   Thanks a lot for the book! It has been my entrypoint to Haskell

2. Jonathan Says:

   February 3, 2016 at 6:01 am
   I really like the idea of bringing functional programming concepts closer to the average imperative programmer who's normally running away as soon as she hears the word "functor". Explaining the fundamentals in a way that they may be applied in every day life (or at least get you thinking) and keeping it simple.

   That said, I feel you're drifting off more and more into the realm of theory and I struggle to keep the connection to my humble every day programming. The promised C++ examples got scarcer and scarcer and I had to more or less force myself through the last few chapters.

Maybe my expectation were a bit too high on that, but I sincerely hope you can get back to where you started off: simple, more practical examples.

3. Bartosz Milewski Says:

February 6, 2016 at 1:09 pm
@Jonathan: Congratulations! You’ve been able to read through 17 installments of this series, and even if you had to force yourself through the last few chapters, that’s a tremendous accomplishment.

As the topics are getting more and more abstract, I’m having problems translating them to C++ and had to lean more on Haskell. One problem is that C++ doesn’t have enough abstracting power. The other is that the more abstract you go, the worse the C++ syntax becomes. I have already had to tap into template-template parameters and anonymous lambdas. C++ type inference is pitifully inadequate, which forces the client to specify template parameters explicitly. At this point, it really becomes worthwhile to learn some Haskell. In fact it was those problems with C++ template syntax that forced me to learn Haskell.

A lot of higher order abstractions in Haskell translate into patterns in C++. It’s good to learn to recognize these patterns in C++, but if you want a compact description of such patterns, you express them in Haskell.

The next installment of the series will be about adjunctions, which unify a lot of constructs that I presented before. This will probably be the most abstract part of the series. But adjunctions tie nicely to monads and comonads that will come next. And there will be a lot of examples of monads both in Haskell and C++ (where they are more of a pattern). Next are F-algebras, which have important applications in C++. With Kan extensions we’ll again ratchet up the level of abstraction. Kan extensions can be expressed in Haskell, and they play an important role in constructing adjoint functors. Yoneda, adjunctions, and Kan extensions provide the abstract underpinning of the Haskell lens library.

So the series will become incrementally more abstract and the readership will slowly shift from programmers to computer scientists, and maybe even to math students. I’ll do my best to present the material in the most approachable way possible, but it will get harder.

If you were able to follow the series so far, you have learned some of the most advanced theory that’s been only available to professional mathematicians, some computer scientists and a few Haskell programmers.

4. Jonathan Says:

February 8, 2016 at 10:46 am
@Bartosz: Thanks for the clarification.
I know, this is indeed a really tough task and I’m grateful that you’re attempting to take it on!
It might be useful to have a few practical examples that relate to the theoretical concepts. Like, what different types can be described as a monad/functor and what are the implications of morphisms for those types (independent of languages)? Even though, of course, those examples couldn’t cover the breadth of abstraction the theory offers, they might give an idea of what the theory is aiming at.

I just can’t help but being a bit disappointed that the chapters seem to move away from your initial idea (according to the preface and the introduction) to write this book for programmers instead of scientists and the next installments seem to continue that trend.
Still, I think it’s a good step in the right direction.

I’ll certainly keep reading and I can’t wait for the next installment.

Thanks

5. Pyry Kontio Says:

   February 14, 2016 at 3:06 pm
   Thanks for this series! I had a lot of trouble following the construction of limits earlier, because I didn’t know what “commuting diagrams” meant! Did I mistakenly skip something? Also, it might be helpful to provide some connection or intuition about the notion of commutative property in the context that most of us know it: normal, boring algebra.

   Btw. I have to ask for clearing this definition up:

   > *A lot of properties of morphisms are expressed in terms of commuting diagrams. If a particular morphism can be described as a composition of other morphisms in more than one way, then we have a commuting diagram.*

   Does the words “more than one way” mean that there has to be at least 2 different ways BESIDES the morphism itself? Or is is one additional way + the morphism itself enough for commutitavity?

6. Bartosz Milewski Says:

   February 14, 2016 at 4:03 pm
   A commuting diagram is a diagram where there are two paths (following the arrows in the diagram) that lead from one object to another and we require that both paths be equivalent. For instance, in a triangular diagram — say the left triangle of the definition of the product — we have two paths: one is p’ (single arrow), and the other is p after m. Because we are in a category, p after m corresponds to a single arrow $p \circ m$ that is the composition of p and m. Commutativity is the requirement that these two morphism: p’ and $p \circ m$ are identical.

   In a square diagram you want to go from one corner object to another. For instance, in the naturality diagram above, you want to get from F a to G b. You can either do that by composing G f after α_a or α_b after F f. Commutativity means that both of these composite morphism are identical. These morphisms lay on the diagonal of the square.

   Notice that, in principle, these two composite morphisms could be different from each other.

This site uses Akismet to reduce spam. Learn how your comment data is processed.

Blog at WordPress.com.