Haskell for all

Saturday, September 15, 2012

The functor design pattern

This post builds on my previous post on the category design pattern and this time I will discuss the functor design pattern. If you are an intermediate Haskell programmer and you think you already understand functors, then think again, because I promise you this post will turn most of your preconceptions about functors on their head and show you how functors are much more powerful and generally applicable than you might realize.

Mixing features

So let's pretend that my previous post on categories inflamed your interest in compositional programming. The first thing you might ask is "Which category do I use?". This seems like a perfectly reasonable question, since you'd like to pick a category that:

- attracts a lot of mindshare,
- contains a large library of reusable components,
- boasts many features, and
- is simple to use.

Unfortunately, reality says that we seldom get all of the above qualities and they often conflict with one another. For example, let's compare two categories I previously discussed:

- Ordinary functions of type `a -> b` that you compose using: `(.)`
- Monadic functions of type `a -> m b` that you compose using: `(<=<)`

Ordinary functions are simpler to read, write, use, and you can reason about their behavior more easily. However, monadic functions boast more useful features, some of which are indispensable (such as side effects when we use the `IO` monad). We really need some way to mix these two categories together to get the best of both worlds.

Fortunately, programmers solve compatibility problems like this all the time. We often have tools written in different languages or different frameworks and if we want to mix features from multiple frameworks we write code to bridge between them. So let's solve our category mixing problem by writing an adapter layer between the two categories. We write some function that transforms all the components in one category into components in the other category, so that we can then freely mix components from the two categories.

Typically, one category will be more featureful than the other, so the transformation is unidirectional. Using the above example, monadic functions are strictly more featureful and powerful than ordinary functions. Fortunately, we can promote all ordinary functions to monadic functions using the following `map` function:

About Me

Gabriel Gonzalez

View my complete profile

```
-- This "map"s an ordinary function to a monadic function
map :: (Monad m) => (a -> b) -> (a -> m b)
map f = return . f
```

... but we cannot write the reverse function and automatically map every monadic function to an ordinary function.

We use `map` to combine a pure function `f` and a monadic function `g`. To do this, we promote `f` using `map` and then combine both of them using Kleisli composition:

```
f     ::              a ->   b
map f :: (Monad m) => a -> m b

g     :: (Monad m) => b -> m c


g <=< map f :: (Monad m) => a -> m c
```

Perfect! Now we can reuse all of our ordinary functions within this Kleisli category and not have to rewrite anything!

However, there's still a problem. Monad binds are not free and sometimes they get in the way of compiler optimization, so you can imagine that it would be wasteful if we lifted two pure functions in a row:

```
f     ::              a ->   b
map f :: (Monad m) => a -> m b

g     ::              b ->   c
map g :: (Monad m) => b -> m c

h     :: (Monad m) => c -> m d

-- Wasteful!
h <=< map g <=< map f :: (Monad m) => a -> m d
```

However, we're smart and we know that we can just optimize those two ordinary functions by using ordinary function composition first before lifting them with `map`:

```
-- Smarter!
h <=< map (g . f)
```

In other words, we assumed that the following transformation should be safe:

```
map g <=< map f = map (g . f)
```

Similarly, we expect that if we lift an identity function into a chain of Kleisli compositions:

```
g <=< map id <=< f
```

... then it should have no effect. Well, we can easily prove that because:

```
map id = return . id = return
```

.. and `return` is the identity of Kleisli composition, therefore:

```
f :: (Monad m) => a -> m b
g :: (Monad m) => b -> m c

map id :: (Monad m) => b -> m b
```

```
g <=< map id <=< f
= g <=< return <=< f
= g <=< f  :: (Monad m) => a -> m c
```

Well, we just unwittingly defined our first functor! But where is the functor?

Functors

A functor transforms one category into another category. In the previous section we transformed the category of Haskell functions into the category of monadic functions and that transformation is our functor.

I will notationally distinguish between the two categories in question so I can be crystal clear about the mathematical definition of a functor. I will denote our "source" category's identity as `idA` and its composition as `(._A)`, and these must obey the category laws:

```
-- Yes, "._A" is ugly, I know
idA ._A f = f                       -- Left identity

f ._A id = f                        -- Right identity

(f ._A g) ._A h = f ._A (g ._A h)  -- Associativity
```

Similarly, I denote the "destination" category's identity as `idB` and its composition as `(._B)`, which also must obey the category laws:

```
idB ._B f = f                       -- Left identity

f ._B idB = f                       -- Right identity

(f ._B g) ._B h = f ._B (g ._B h)  -- Associativity
```

Then a functor uses a function that we will call `map` to convert every component in the source category into a component in the destination category.

We expect this `map` function to satisfy two rules:

Rule #1: `map` must transform the composition operator in the source category to the composition operator in the destination category:

```
map (f ._A g) = map f ._B map g
```

This is the "composition law".

Rule #2: `map` must transform the identity in the source category to the identity in the destination category:

```
map idA = idB
```

This is the "identity law".

Together these two rules are the "functor laws" (technically, the covariant functor laws).

In the last section, our source category "A" was the category of ordinary functions:

```
idA   = id
(._A) = (.)
```

... and our destination category "B" was the Kleisli category:

```
idB   = return
(._B) = (<=<)
```

... and our `map` function obeyed the functor laws:

```
map id = return
map (f . g) = map f <=< map g
```

In other words, functors serve as adapters between categories that promote code written for the source category to be automatically compatible with the destination category. Functors arise every time we write compatibility layers and adapters between different pieces of software.

Functors hidden everywhere

I'll provide a few more examples of functors to tickle people's brains and show how functors arise all the time in your code without you even realizing it. For example, consider the `length` function:

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

We can treat list concatenation as a category, where:

```
(.) = (++)
id  = []

[] ++ x = x                    -- Left  identity
x ++ [] = x                    -- Right identity
(x ++ y) ++ z = x ++ (y ++ z)  -- Associativity
```

Similarly, we can treat addition as a category, where:

```
(.) = (+)
id  = 0

0 + x = x                  -- Left  identity
x + 0 = x                  -- Right identity
(x + y) + z = x + (y + z)  -- Associativity
```

Then `length` is a functor from the category of list concatentation to the category of integer addition:

```
-- Composition law
length (xs ++ ys) = length xs + length ys

-- Identity law
length [] = 0
```

Or consider the `pipe` function from `Control.Pipe`:

```
pipe :: (Monad m) => (a -> b) -> Pipe a b m r

-- Composition law
pipe (f . g) = pipe f <+< pipe g

-- Identity law
pipe id = idP
```

Also, `concat` defines a functor from one list concatenation to another:

```
-- Composition
concat (x ++ y) = concat x ++ concat y

-- Identity
```

```
concat [] = []
```

So don't assume that the `Functor` class is in any way representative of the full breadth of functors.

The `Functor` class

So far I've deliberately picked examples that do not fit within the mold of Haskell's `Functor` class to open people's minds about functors. A lot of new Haskell programmers mistakenly believe that functors only encompass "container-ish" things and I hope the previous examples dispel that notion.

However, the `Functor` class still behaves the same way as the functors I've already discussed. The only restriction is that the `Functor` class only encompass the narrow case where the source and target categories are both categories of ordinary functions:

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)

fmap (f . g) = fmap f . fmap g  -- Composition law

fmap id = id                    -- Identity law
```

Haskell `Functors` recapitulate the themes of compatibility between categories and component reuse. For example, we might have several ordinary functions lying around in our toolbox:

```
f :: a -> b
g :: b -> c
```

.. but we need to manipulate lists using functions of type:

```
h :: [a] -> [c]
```

Rather than rewrite all our old functions to work on lists, we can instead automatically promote all of them to work on lists using the `map` function from the Prelude:

```
map :: (a -> b) -> ([a] -> [b])

map f :: [a] -> [b]
map g :: [b] -> [c]

h = map f . map g :: [a] -> [c]
```

We know that we can combine two passes over a list into a single pass:

```
h = map (f . g) :: [a] -> [c]
```

.. and doing nothing to each element does nothing to the list:

```
map id = id
```

Once again, we've just stated the functor laws:

```
map (f . g) = map f . map g  -- Composition law

map id = id                  -- Identity law
```

Notice that functors free us from having to write code that targets some monolithic category. Instead, we write all our code using whatever category we deem most appropriate and then promote it as necessary to whatever other categories might need the code we just wrote. This lets us work within

focused and specialized categories suitable for their respective tasks rather than waste our time arguing over what category to standardize on.

Another benefit of functors is that they make our code automatically future-proof. We write our components using whatever category we have at our disposal and then as new categories arise we just define new functors to promote our existing code to work within those new categories.

Monad morphisms

Compatibility issues arise all the time between various Haskell frameworks. For example, let's assume I have a sizeable code-base written using the `iteratee` library, but then I find a really useful library on hackage using `enumerator`. I would rather not rewrite the `enumerator`-based library to use `iteratee` so I instead choose to write an adapter function that allows me to mix the two. I have to define some function, `morph`, that transforms `Iteratees` from the `iteratee` library into `Iteratees` from the `enumerator` library:

```
import qualified Data.Enumerator as E
import qualified Data.Iteratee.Base as I

morph :: I.Iteratee a m b -> E.Iteratee a m b
```

However, I might suspect that the `iteratee` library has a faster `Monad` instance since it uses continuation-passing style (disclaimer: I have no idea if this is true, it's just a hypothetical example). This means that I would like to be able to factor code to use the `iteratee` library's monad whenever possible:

```
f :: a -> I.Iteratee s m b

g :: b -> I.Iteratee s m c

h :: c -> E.Iteratee s m d

-- Hypothetically slower, since it uses E.Iteratee's bind
code1 :: a -> E.Iteratee s m d
code1 a = do b <- morph $ f a
             c <- morph $ g b
             h c

-- Hypothetically faster, since it uses I.Iteratee's bind
code2 :: a -> E.Iteratee s m d
code2 a = do c <- morph $ do b <- f a
                             g b
             h c
```

I would also expect that if I do nothing using `enumerator`, that it's equivalent to doing nothing using `iteratee`:

```
morph $ return x
= return x
```

Interestingly, we encounter a pattern when we write the above functions using a point-free style:

```
code1 = h <=< (morph . g) <=< (morph . f)

code2 = h <=< (morph . (g <=< f))
```

```
morph . return = return
```

This pattern seems so familiar...

```
map :: (a -> I.Iteratee s m b) -> (a -> E.Iteratee s m b)
map = (morph .)

map (f <=< g) = map f <=< map g  -- Composition law

map return = return              -- Identity law
```

Oh, I've accidentally defined a functor! This time both the source and destination categories are Kleisli categories and the functor preserves both the composition and identity correctly.

Category theorists have a very specific name for the above pattern: a monad morphism. Specifically, a monad morphism is any function:

```
morph :: (Monad m, Monad n) => forall r . m r -> n r
```

... such that `map = (morph .)` defines a functor between two Kleisli categories:

```
map :: (Monad m, Monad n) => (a -> m b) -> (a -> n b)
map = (morph .)
```

Also, intermediate Haskell programmers will recognize a subtle variation on this pattern:

```
lift :: (Monad m, MonadTrans t) => m r -> t m r

(lift .) :: (Monad m, MonadTrans t) => (a -> m b) -> (a -> t m b)

-- Identity law
(lift .) return = return

-- Composition law
(lift .) (f >=> g) = (lift .) f >=> (lift .) g
```

These are just the monad transformer laws! However, they are usually written in this form:

```
lift $ return x = return x

lift $ do y <- f x
          g y
= do y <- lift $ f x
     lift $ g y
```

In other words, monad transformers are a special subset of monad morphisms and the monad transformer laws are just the functor laws in disguise!

Now, every time you use a monad transformer you can appreciate that you are using a functor as an adapter layer between two categories: the base monad's Kleisli category and the transformed monad's Kleisli category.

Conclusion

The functor design pattern embodies a philosophy of programming that emphasizes:

- compatibility over standardization,

- specialization over monolithic frameworks, and
- short-term completion over future-proofing.

However, the above tenets, while popular, haven't completely taken hold because we associate:

- compatibility with cruft,
- specialization with fragmentation, and
- short-term completion with lack of foresight.

In a future post I will discuss how the functor laws mitigate these problems and allow you to layer on as many abstractions over as many tools as you wish without the abstractions collapsing under their own weight.

Functors don't even necessarily need to be within a single programming language. A programmer could even use the category design pattern in completely separate programming languages and then use the functor design pattern to bridge components written in one language to another. Please don't limit your imagination to just the examples I gave!

However, the functor design pattern doesn't work at all if you aren't using categories in the first place. This is why you should structure your tools using the compositional category design pattern so that you can take advantage of functors to easily mix your tools together. This is true whether you are programming in one language or several languages. As long as each tool forms its own category and you obey the functor laws when switching between them, you can be very confident that all your tools will mix correctly.

Posted by Gabriel Gonzalez at 10:19 AM

25 comments:

id September 16, 2012 at 3:07 AM

I'm a relative newcomer to Haskell, and your blog has been a huge boon for learning to think more functionally. Great work, and thanks!

Reply

Replies

Twisol September 16, 2012 at 3:10 AM

Eh, I guess your blog didn't like the OpenID auth I used, but that's my comment above.

Reply

Twisol September 16, 2012 at 3:09 AM

This comment has been removed by the author.

Reply

Dan Fornika September 16, 2012 at 10:52 AM

How have you learned such a deep understanding of Haskell's type system and category theory? Do you study this at school? Do you read a particular academic journal, or have a good textbook to suggest?

I've been making my way through Real World Haskell, and it's great but this sort of post is just on a whole other level.

Reply

Replies

Gabriel Gonzalez September 16, 2012 at 12:03 PM

I think the way I learn most of these things is a combination of programming in Haskell, some self-taught category theory, and looking for patterns in my own programs and libraries.

I don't read a lot of computer science papers. Usually I only read a paper if it comes up in a Google search or if it appears on /r/haskell. Most of the category theory I learn is either (a) from Wikipedia, (b) from asking more experienced Haskell programmers (either on #haskell or /r/haskell), or (c) from Steve Awodey's Category Theory textbook, in roughly that order.

Unfortunately, most literature on category theory uses math examples to motivate each topic, which would really frustrate me, so I would read about it and I wouldn't "get" the significance of it to programming at all. Like, I'd be able to define it and point to Haskell code examples of it, but I wouldn't understand the purpose or motivation behind it or why I should structure my code that way as opposed to some other abstraction or design pattern.

For example, I took a scientific writing mini-course and they would always stress that you should begin with the motivation and purpose before you introduce anything else, and I found that profoundly missing from every category theory topic, at least with respect to programming. So I decided I would just try to figure it out on my own instead of waiting for the category theory tutorial that would never come. Mathematicians seemed to be generally excited about category theory and how generally applicable it was, so I just sort of kept trying, assuming that they were really onto something. It took me a while before I could finally figure out a way to articulate their excitement in a programmer's voice.

Dan Fornika September 17, 2012 at 7:34 AM

Wow, that's really pretty inspiring that you're mostly self-taught. I've seen you around on /r/haskell and I appreciate the effort you put into discussing, debating and explaining things there. Thanks and keep it up!

Reply

gergo-erdi September 16, 2012 at 7:07 PM

There's a typo in the type of (fst .):

(fst .) :: ((a, c) -> (b, c)) -> (a -> b)

this type is uninhabited (or would be if not for bottoms), snce you have nowhere to pull from the c to plug in, and the b in the result can depend arbitrarily on it.

I think you meant

(fst .) :: (a -> (b, c)) -> (a -> b)

which is, incidentally, the principal type of (fst .) anyway.

Reply

Replies

Gabriel Gonzalez September 24, 2012 at 5:14 PM

Yes, this was a mistake on my part. At the time I wrote it I had in mind the following composition:

```
(>->) :: (a -> (b, b)) -> (b -> (c, c)) -> (a -> (c, c))
(f >-> g) a = (g $ fst $ f a, g $ snd $ f a))

id2 :: a -> (a, a)
id2 a = (a, a)

id2 >-> f = f
f >-> id2 = f
(f >-> g) >-> h = f >-> (g >-> h)
```

And then you get the correct functor laws:

```
(fst .) id2 = id
(fst .) (f >-> g) = (fst . f) >>> (fst . g)
```

... but somehow the wires got crossed when I wrote it and I wrote something completely wrong. I'll just remove the example.

Reply

Bryan Richter October 23, 2012 at 12:27 PM

Hi Gabriel, thanks for this. I want to point out a typo in, and ask a question about, building the bridge between enumerator and iteratee.

Typo: the 'f' at the end of the chain should be 'h' for both code1 and code2, e.g. "code1 = f <=< ..." should be "code1 = h <=< ..."

Question: You motivate a preference for I.(>>=) over E.(>>=), but it's not clear where that topic leads. By showing that the composition law holds, are you simply showing that the code is provably correct regardless of which bind you choose? For some value of "provably". :)

Reply

Replies

Gabriel Gonzalez October 23, 2012 at 12:31 PM

Yes, the code should run identically no matter which bind you choose. In practice, there may be performance differences.

This definition of equality ignores observable differences in performance, but you could imagine some more sophisticated language where performance differences had some in-language representation and equality also meant "equally performant".

Also, I will fix the typo. Thanks for bringing that up.

Reply

Giorgio Valoti October 25, 2012 at 9:54 AM

Hi Gabriel, if I understand correctly, you seem to imply that a functor is more or less a function which respects a given set of properties regarding composition and identity of the source and destination categories. However, when I look at this example: http://www.haskell.org/haskellwiki

/Category_theory
/Natural_transformation#Vertical_arrows:_sides_of_objects it seems to view the functor more as a container.

They both make sense but I don't know how to reconcile them.

Reply

Replies

Gabriel Gonzalez October 25, 2012 at 10:50 AM

A functor is actually TWO things:

1) A way to map objects, and
2) way to map morphisms.

This post only discussed how a functor maps morphisms because I wanted to give a treatment of objects in a later post.

So your "container-ish" intuition is more or less right in the sense that a functor transforms an object. You might imagine that if the functor is "F", and the object is "X", then you can wrap the object "X" in "F" to generate a new object: "F(X)".

However, since I gave a "morphism-only" treatment of functors, it didn't really make clear what the objects were and what "F" is wrapping, so I'll try to address this using the concrete example I gave of "length".

It's type signature is:

length :: [a] -> Int

"length" transforms morphisms, not objects, meaning that the list (i.e. "[a]") is a morphism in some source category and the Int is a morphism in some destination category.

In this particular case, the source category is a Monoid (where mappend = (++) and mempty = []) and the destination category is a Monoid (where mappend = (+) and mempty = 0). Remember that the lists are morphisms, not the objects, and the integers are morphisms, not the objects. That's why they don't have the container-like shape you are expecting.

But there are objects there, even if they don't show up in length's type signature. To see why, you have to think of a Monoid as a category with one object. In fact, the nature of the object does not matter at all. It could be anything. The only thing that matters is just that there is only one such object.

So let's call that single object "X". That means that lists are morphisms from "X" to "X". We can prove this by wrapping Haskell lists in a GADT that permits a suitable "Category" instance:

```
-- Notice that we never actually use 'x'
data List e a b where
List :: [e] -> List e x x

instance Category (List e) where
id = List []
List xs . List ys = List (xs ++ ys)
```

Since there is only one object in this category, every morphism has the same source and destination object, so all compositions type-check. Monoids are basically categories where we just want to ensure that every composition type-checks.

We can similarly define a category for Ints:

```
data IntCat a b where
IntCat :: Int -> IntCat x x

instance Category IntCat where
id = IntCat 0
IntCat x . IntCat y = IntCat (x + y)
```

So using these two GADTs I've made the objects explicit so we can talk about them now.

This means our "length" function should really be thought of as:

```
length :: List e a b -> Int (f a) (f b)
```

... where f is the container-like thing you were looking for. Notice the similarity between our upgraded "length" type signature and "fmap"'s type signature:

```
length :: List e a b -> Int (f a) (f b)
fmap :: (a -> b) -> (f a -> f b)
```

Now we can see that we are actually transforming objects under the hood in a "container-ish" way, even if the original type of "length" didn't really make that explicit.

The only difference is which source and destination category we use. fmap uses the category of Haskell functions for both its source and destination category. length uses the list monoid as its source category and the Int monoid as its destination category.

I'm sorry if that's long-winded and perhaps confusing, but if you need any more clarification I'd be happy to help.

Giorgio Valoti October 29, 2012 at 4:05 AM

Yeah, I'm confused :)

So, lists and integer are both morphisms in the Monoid category, because the source and destination objects of these morphisms coincide. More concretely, in the category IntCat 3, 3 is the both the source and the destination. Correct?

Gabriel Gonzalez November 1, 2012 at 11:32 AM

Actually, 3 is the morphism and the source/destination are just some anonymous object which could be anything. That's the confusing thing about monoids, which is that the monoid elements (i.e. the 3) are the morphisms, not the object. Mappending monoid elements (i.e. adding integers) is analogous to composing morphisms and the monoid laws correspond to the category laws.

Monoids are an example of a category where the morphisms are more interesting than the object.

To make the analogy perhaps clearer, you can remember that for any Haskell `Category` instance you can simply pick one object in that `Category`, any object, and all functions from that object to itself form a monoid:

```
instance (Category c) => Monoid (c a a) where
mempty = id
mappend = (.)
```

For example, if the category is the category of Haskell functions, then you just pick some type `a` and all function of type (a -> a) form a monoid. For example, let's say that `a = String`, then, you can take all functions of type `String -> String` form a monoid:

```
f :: String -> String
g :: String -> String

mappend f g = f . g :: String -> String

mempty :: String -> String
mempty = id
```

Reply

Riccardo Pelizzi September 3, 2013 at 5:47 PM

I'm still struggling to understand the purpose of `forall`, and maybe this is a good chance to get some insight: what is the purpose of `forall` in

```
morph :: (Monad m, Monad n) => forall r . m r -> n r
```

The function would be already polymorphic without the quantifier. How does the quantification change the type, in concrete terms?

Reply

Replies

Gabriel Gonzalez September 3, 2013 at 6:16 PM

It doesn't change the type in that specific case. The reason it is there is that monad morphisms do need the forall notation when they are the argument of a function and I just kept it for consistency.

Reply

thaumkid March 29, 2014 at 2:32 PM

I'm not very good at category theory, so take this for what it's worth. Delete if not helpful.

From a mathematics view, it seems that categories and monoids are being confused with each other. Under "Functors Hidden Everywhere", the post claims lists with append and numbers with addition are categories. I would claim they are objects in the category of monoids. You have an underlying set (lists of Characters, or individual Integers), an operation on members of the set (append, or add), and an identity (empty list, or zero) and out comes a monoid.

To treat lists as a functor, which you can do, you have to think what categories they are moving between. One possible list functor (there are many), moves from the category of sets to the category of monoids (also called the free monoid functor). Take your underlying alphabet (the set of all characters, or the set of all objects of some kind), and lift it into the kleene closure (all finite lists of characters from the alphabet), with the added append operation and empty list. This is essentially what Haskell lists do and why they are functors. All lists of a given kind (say, lists with lower case letters as elements) comprise a single object in the monoid category.

Integers with addition can be thought of as an object in a category, like the category of abelian groups, or the category of monoids.

Basically, these ideas in category theory are one step more abstract than the way they are presented and used here. A category does not have an identity object, but each object in a category has an identity arrow. Composition is the stringing of arrows together, not an arbitrary operation between objects that you get to define.

Reply

Replies

rob April 5, 2014 at 11:02 AM

You can construct a category from any monoid. That category has just one object, and the monoid elements become the arrows; the neutral element becomes the identity arrow. Under this translation, monoid homomorphisms become functors.

It's unclear to me that it's useful to think of these as functors instead of monoid homomorphisms, though...

Gabriel Gonzalez April 6, 2014 at 9:52 AM

Why I should think of them as monoid homomorphisms when functors describe them just fine. What is the advantage of introducing an additional concept?

Reply

jartur May 20, 2014 at 10:36 PM

What I don't understand is when I write

instance Functor Maybe where
fmap = ...

what is exactly a functor here? Is fmap a functor? Is Maybe a functor? Seems to me that Maybe is not a functor but why then we say that it is by defining an instance of Functor?

Reply

Replies

Gabriel Gonzalez May 30, 2014 at 8:00 AM

Sorry for the late reply. So to be totally accurate a functor is a combination of two things: a way to transform morphisms and way to transform objects. Using Haskell's `Functor` class as an example, `fmap` would be the way to transform morphisms (which are in this case functions) and the type constructor (like `Maybe`) would be the way to transform the objects (which in this case are types).

In the monad morphism examples, this time the source and destination categories are kleisli categories (i.e. morphisms of the shape `a -> m b`) and the transformation between morphism is `(morph .)`. The objects are Haskell types and the monad morphism just leaves them unchanged (i.e. the identity transformation).

The other examples are trickier because the source and destination categories in those examples are monoids. You can think of a monoid as a category with a single anonymous object, and the elements of the monoid are morphisms that begin and end on that object.

Using the `length` example, the morphisms of the source category are lists and the morphisms of the destination category are integers, and both categories have a single nameless object which is not included in the type. The `length` function is the transformation between the morphisms and the single anonymous object is left unchanged (i.e. the identity object).

So going back to your original question, the answer is that the combination of `Maybe` and `fmap` (specialized to the `Maybe` type) are the true functor.

jartur May 31, 2014 at 1:30 AM

Thank you. I don't understand everything but this explanation fills in some of the gaps of my understanding.

Gabriel Gonzalez May 31, 2014 at 8:27 PM

You're welcome!

Reply

S Pradeep Kumar December 6, 2014 at 2:59 AM

This post just blew my mind!

I guess your point is that Haskell awesomeness is part of a much bigger scheme.

"Why Functional Programming Matters" by John Hughes was my first aha moment regarding FP - composing using higher-order functions and lazy evaluation as glue.

This is my second aha moment - that we can use functors, monads, and other math stuff as glue. We can get powerful and generally applicable code by gluing together seemingly unrelated areas (e.g., using the same function on numbers, lists, Maybes, trees, IOs, and whatnot thanks to fmap).

I keep hearing about Category Theory but never really got _why_ it matters (amid all the vague-sounding mathematical talk). This is the first time I've heard anyone claiming direct concrete benefits to us programmers.

That's really inspiring! Now that I know it can make a huge difference, I'll dive into the whole math intuition behind Monad/Functors/Categories, etc. Thank you so much.

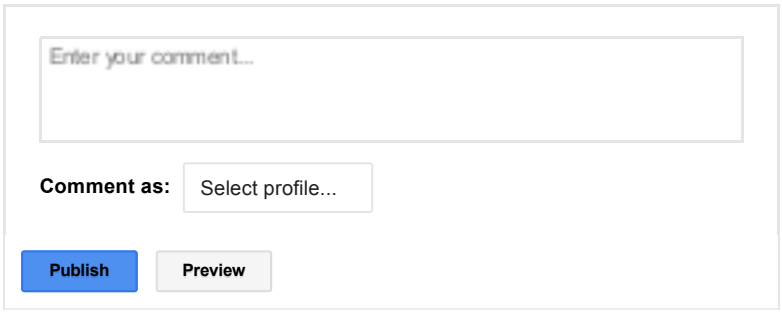
Reply

Replies

Gabriel Gonzalez December 31, 2014 at 2:52 PM

Yeah, Haskell is the first language that let me see the connections between mathematics and programming. It really appealed to me because I had always loved mathematics because of all the patterns and commonalities, so I loved it even more when I could find patterns and commonalities in my code that were also mathematical.

Reply



Newer Post Home Older Post

Blog Archive