Bartosz Milewski's Programming Cafe

Category Theory, Haskell, Concurrency, C++

May 16, 2018

Free Monoidal Functors, Categorically!

Posted by Bartosz Milewski under Category Theory
[14] Comments

i
4 Votes

Functors from a monoidal category C to Set form a monoidal category with Day convolution as product. A monoid in this category is a lax monoidal functor. We define an initial algebra using a higher order functor and show that it corresponds to a free lax monoidal functor.

Recently I've been obsessing over monoidal functors. I have already written two blog posts, one about free monoidal functors and one about free monoidal profunctors. I followed some ideas from category theory but, being a programmer, I leaned more towards writing code than being preoccupied with mathematical rigor. That left me longing for more elegant proofs of the kind I've seen in mathematical literature.

I believe that there isn't that much difference between programming and math. There is a whole spectrum of abstractions ranging from assembly language, weakly typed languages, strongly typed languages, functional programming, set theory, type theory, category theory, and homotopy type theory. Each language comes with its own bag of tricks. Even within one language one starts with some relatively low level encodings and, with experience, progresses towards higher abstractions. I've seen it in Haskell, where I started by hand coding recursive functions, only to realize that I can be more productive using bulk operations on types, then building recursive data structures and applying recursive schemes, eventually diving into categories of functors and profunctors.

I've been collecting my own bag of mathematical tricks, mostly by reading papers and, more recently, talking to mathematicians. I've found that mathematicians are happy to share their knowledge even with outsiders like me. So when I got stuck trying to clean up my monoidal functor code, I reached out to Emily Riehl, who forwarded my query to Alexander Campbell from the Centre for Australian Category Theory. Alex's answer was a very elegant proof of what I was clumsily trying to show in my previous posts. In this blog post I will explain his approach. I should also mention that most of the results presented in this post have already been covered in a comprehensive paper by Rivas and Jaskelioff, Notions of Computation as Monoids.

Lax Monoidal Functors

To properly state the problem, I'll have to start with a lot of preliminaries. This will require some prior knowledge of category theory, all within the scope of my blog/book.

We start with a monoidal category C, that is a category in which you can "multiply" objects using some kind of a tensor product $\otimes$. For any pair of objects a and b there is an object $a \otimes b$; and this mapping is functorial in both arguments (that is, you can also "multiply" morphisms). A monoidal category will also have a special object I that is the unit of multiplication. In general, the unit and associativity laws are satisfied up to isomorphism:

$$\lambda : I \otimes a \cong a$$

$$\rho : a \otimes I \cong a$$

$$\alpha : (a \otimes b) \otimes c \cong a \otimes (b \otimes c)$$

These isomorphisms are called, respectively, the left and right unitors, and the associator.

The most familiar example of a monoidal category is the category of types and functions, in which the tensor product is the cartesian product (pair type) and the unit is the unit type `()`.

Let's now consider functors from C to the category of sets, Set. These functors also form a category called $[C, Set]$, in which morphisms between any two functors are natural transformations.

In Haskell, a natural transformation is approximated by a polymorphic function:

```
type f ~> g = forall x. f x -> g x
```

The category Set is monoidal, with cartesian product $\times$ serving as a tensor product, and the singleton set 1 as the unit.

We are interested in functors in $[C, Set]$ that preserve the monoidal structure. Such a functor should map the tensor product in C to the cartesian product in Set and the unit I to the singleton set 1. Accordingly, a strong monoidal functor F comes with two isomorphisms:

$$Fa \times Fb \cong F(a \otimes b)$$

$$1 \cong FI$$

We are interested in a weaker version of a monoidal functor called *lax monoidal* functor, which is equipped with a one-way natural transformation:

$$\mu : Fa \times Fb \to F(a \otimes b)$$

and a one-way morphism:

$$\eta : 1 \to FI$$

A lax monoidal functor must also preserve unit and associativity laws.

$$\begin{tikzcd}
(Fa \times Fb) \times Fc \arrow[rr, leftrightarrow, "\alpha"] \arrow[d, "\mu \times id"] & & Fa \times (Fb \times Fc) \arrow[d, "id \times \mu"] \\
F(a \otimes b) \times Fc \arrow[d, "\mu"] & & Fa \times F(b \otimes c) \arrow[d, "\mu"] \\
F((a \otimes b) \otimes c) \arrow[rr, leftrightarrow, "F\alpha"] & & F(a \otimes (b \otimes c))
\end{tikzcd}$$

Associativity law: α is the associator in the appropriate category (top arrow, in Set; bottom arrow, in C).

$$\begin{tikzcd}
Fa \times 1 \arrow[r, "id \times \eta"] \arrow[d, "\rho"'] & Fa \times FI \arrow[d, "\mu"] \\
Fa & F(a \otimes I) \arrow[l, "F\rho"']
\end{tikzcd}$$

$$\begin{tikzcd}
1 \times Fa \arrow[r, "\eta \times id"] \arrow[d, "\lambda"'] & FI \times Fa \arrow[d, "\mu"] \\
Fa & F(I \otimes a) \arrow[l, "F\lambda"']
\end{tikzcd}$$

In Haskell, a lax monoidal functor can be defined as:

```
class Monoidal f where
  eta :: () -> f ()
  mu  :: (f a, f b) -> f (a, b)
```

It's also known as the *applicative* functor.

Day Convolution and Monoidal Functors

It turns out that our category of functors $[C, Set]$ is also equipped with monoidal structure. Two functors F and G can be "multiplied" using Day convolution:

$$(F \star G)c = \int^{ab} C(a \otimes b, c) \times Fa \times Gb$$

Here, $C(a \otimes b, c)$ is the hom-set, or the set of morphisms from $a \otimes b$ to c. The integral sign stands for a coend, which can be interpreted as a generalization of an (infinite) coproduct (modulo some identifications). An element of this coend can be constructed by injecting a triple consisting of a morphism from $C(a \otimes b, c)$, an element of the set Fa, and an element of the set Gb, for some a and b.

In Haskell, a coend corresponds to an existential type, so the Day convolution can be defined as:

```
data Day f g c where
  Day :: ((a, b) -> c, f a, g b) -> Day f g c
```

(The actual definition uses currying.)

The unit with respect to Day convolution is the hom-functor:

$C(I, -)$

which assigns to every object c the set of morphisms $C(I, c)$ and acts on morphisms by post-composition.

The proof that this is the unit is instructive, as it uses the standard trick: the co-Yoneda lemma. In the coend form, the co-Yoneda lemma reads, for a covariant functor F:

$\int^x C(x, a) \times Fx \cong Fa$

and for a contravariant functor H:

$\int^x C(a, x) \times Hx \cong Ha$

(The mnemonics is that the integration variable must appear twice, once in the negative, and once in the positive position. An argument to a contravariant functor is in a negative position.)

Indeed, substituting $C(I, -)$ for the first functor in Day convolution produces:

$(C(I, -) \star G)c = \int^{ab} C(a \otimes b, c) \times C(I, a) \times Gb$

which can be "integrated" over a using the Yoneda lemma to yield:

$\int^b C(I \otimes b, c) \times Gb$

and, since I is the unit of the tensor product, this can be further "integrated" over b to give Gc. The right unit law is analogous.

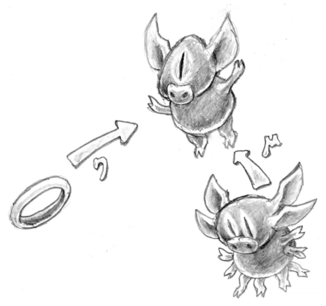
To summarize, we are dealing with three monoidal categories: C with the tensor product $\otimes$ and unit I, Set with the cartesian product and singleton 1, and a functor category $[C, Set]$ with Day convolution and unit $C(I, -)$.

A Monoid in [C, Set]

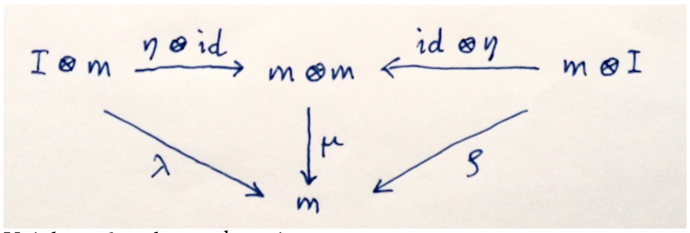
A monoidal category can be used to define monoids. A monoid is an object m equipped with two morphisms — unit and multiplication:

$\eta : I \to m$

$\mu : m \otimes m \to m$

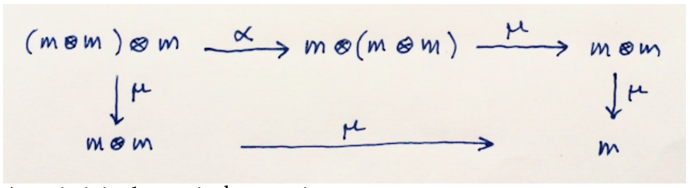


These morphisms must satisfy unit and associativity conditions, which are best illustrated using commuting diagrams.



$$\begin{tikzcd} I \otimes m \arrow[r, "\eta \otimes id"] \arrow[dr, "\lambda"'] & m \otimes m \arrow[d, "\mu"] & m \otimes I \arrow[l, "id \otimes \eta"'] \arrow[dl, "\varrho"] \\ & m & \end{tikzcd}$$

Unit laws. λ and ϱ are the unitors.



$$\begin{tikzcd} (m \otimes m) \otimes m \arrow[r, "\alpha"] \arrow[d, "\mu"] & m \otimes (m \otimes m) \arrow[r, "\mu"] & m \otimes m \arrow[d, "\mu"] \\ m \otimes m \arrow[rr, "\mu"] & & m \end{tikzcd}$$

Associativity law: α is the associator.

This definition of a monoid can be translated directly to Haskell:

```
class Monoid m where
  eta :: () -> m
  mu  :: (m, m) -> m
```

It so happens that a lax monoidal functor is exactly a monoid in our functor category $[C, Set]$. Since objects in this category are functors, a monoid is a functor F equipped with two natural transformations:

$\eta : C(I, -) \to F$

$\mu : F \star F \to F$

At first sight, these don't look like the morphisms in the definition of a lax monoidal functor. We need some new tricks to show the equivalence.

Let's start with the unit. The first trick is to consider not one natural transformation but the whole hom-set:

$$[C, Set](C(I, -), F)$$

The set of natural transformations can be represented as an end (which, incidentally, corresponds to the `forall` quantifier in the Haskell definition of natural transformations):

$$\int_c Set(C(I, c), Fc)$$

The next trick is to use the Yoneda lemma which, in the end form reads:

$$\int_c Set(C(a, c), Fc) \cong Fa$$

In more familiar terms, this formula asserts that the set of natural transformations from the hom-functor $C(a, -)$ to F is isomorphic to Fa.

There is also a version of the Yoneda lemma for contravariant functors:

$$\int_c Set(C(c, a), Hc) \cong Ha$$

The application of Yoneda to our formula produces FI, which is in one-to-one correspondence with morphisms $1 \to FI$.

We can use the same trick of bundling up natural transformations that define multiplication μ.

$$[C, Set](F \star F, F)$$

and representing this set as an end over the hom-functor:

$$\int_c Set((F \star F)c, Fc)$$

Expanding the definition of Day convolution, we get:

$$\int_c Set(\int^{ab} C(a \otimes b, c) \times Fa \times Fb, Fc)$$

The next trick is to pull the coend out of the hom-set. This trick relies on the co-continuity of the hom-functor in the first argument: a hom-functor from a colimit is isomorphic to a limit of hom-functors. In programmer-speak: a function from a sum type is equivalent to a product of functions (we call it case analysis). A coend is a generalized colimit, so when we pull it out of a hom-functor, it turns into a limit, or an end. Here's the general formula, in which pxy is an arbitrary profunctor:

$$Set(\int^x pxx, y) \cong \int_x Set(pxx, y)$$

Let's apply it to our formula:

$$\int_c \int_{ab} Set(C(a \otimes b, c) \times Fa \times Fb, Fc)$$

We can combine the ends under one integral sign (it's allowed by the Fubini theorem) and move to the next trick: hom-set adjunction:

$$Set(a \times b, c) \cong Set(a, b \to c)$$

In programming this is known as currying. This adjunction exists because Set is a cartesian closed category. We'll use this adjunction to move $Fa \times Fb$ to the right:

$$\int_{abc} Set(C(a \otimes b, c), (Fa \times Fb) \to Fc)$$

Using the Yoneda lemma we can "perform the integration" over c to get:

$$\int_{ab} (Fa \times Fb) \to F(a \otimes b))$$

This is exactly the set of natural transformations used in the definition of a lax monoidal functor. We have established one-to-one correspondence between monoidal multiplication and lax monoidal mapping.

Of course, a complete proof would require translating monoid laws to their lax monoidal counterparts. You can find more details in Rivas and Jaskelioff, Notions of Computation as Monoids.

We'll use the fact that a monoid in the category $[C, Set]$ is a lax monoidal functor later.

Alternative Derivation

Incidentally, there are shorter derivations of these formulas that use the trick borrowed from the proof of the Yoneda lemma, namely, evaluating things at the identity morphism. (Whenever mathematicians speak of Yoneda-like arguments, this is what they mean.)

Starting from $F \star F \to F$ and plugging in the Day convolution formula, we get:

$$\int^{a'b'} C(a' \otimes b', c) \times Fa' \times Fb' \to Fc$$

There is a component of this natural transformation at $(a \otimes b)$ that is the morphism:

$$\int^{a' b'} C(a' \otimes b', a \otimes b) \times Fa' \times Fb' \to F(a \otimes b)$$

This morphism must be defined for all possible values of the coend. In particular, it must be defined for the triple $(id_{a \otimes b}, Fa, Fb)$, giving us the μ we seek.

There is also an alternative derivation for the unit: Take the component of the natural transformation η at I:

$$\eta_I : C(I, I) \to LI$$

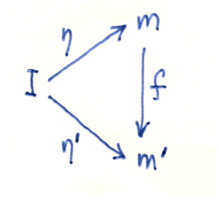
$C(I, I)$ is guaranteed to contain at least one element, the identity morphism id_I. We can use $\eta_I \, id_I$ as the (only) value of the lax monoidal constraint at the singleton 1.

Free Monoid

Given a monoidal category C, we might be able to define a whole lot of monoids in it. These monoids form a category $Mon(C)$. Morphisms in this category correspond to those morphisms in C that preserve monoidal structure.

Consider, for instance, two monoids m and m'. A monoid morphism is a morphism $f : m \to m'$ in C such that the unit of m' is related to the unit of m:

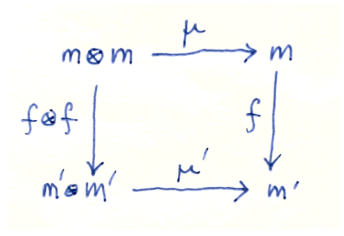
$$\eta' = f \circ \eta$$


$$\begin{tikzcd} & m \arrow[d, "f"] \\ I \arrow[ur, "\eta"] \arrow[r, "\eta'"'] & m' \end{tikzcd}$$

and similarly for multiplication:

$$\mu' \circ (f \otimes f) = f \circ \mu$$

Remember, we assumed that the tensor product is functorial in both arguments, so it can be used to lift a pair of morphisms.


$$\begin{tikzcd} m \otimes m \arrow[r, "\mu"] \arrow[d, "f \otimes f"'] & m \arrow[d, "f"] \\ m' \otimes m' \arrow[r, "\mu'"] & m' \end{tikzcd}$$

There is an obvious forgetful functor U from $Mon(C)$ to C which, for every monoid, picks its underlying object in C and maps every monoid morphism to its underlying morphism in C.

The left adjoint to this functor, if it exists, will map an object a in C to a free monoid La.

The intuition is that a free monoid La is a list of a.

In Haskell, a list is defined recursively:

```
data List a = Nil | Cons a (List a)
```

Such a recursive definition can be formalized as a fixed point of a functor. For a list of `a`, this functor is:

```
data ListF a x = NilF | ConsF a x
```

Notice the peculiar structure of this functor. It's a sum type: The first part is a singleton, which is isomorphic to the unit type `()`. The second part is a product of `a` and `x`. Since the unit type is the unit of the product in our monoidal category of types, we can rewrite this functor symbolically as:

$$\Phi a x = I + a \otimes x$$

It turns out that this formula works in any monoidal category that has finite coproducts (sums) that are preserved by the tensor product. The fixed point of this functor is the free functor that generates free monoids.

I'll define what is meant by the fixed point and prove that it defines a monoid. The proof that it's the result of a free/forgetful adjunction is a bit involved, so I'll leave it for a future blog post.

Algebras

Let's consider algebras for the functor F. Such an algebra is defined as an object x called the carrier, and a morphism:

$f : Fx \to x$

called the *structure map* or the evaluator.

In Haskell, an algebra is defined as:

```
type Algebra f x = f x -> x
```

There may be a lot of algebras for a given functor. In fact there is a whole category of them. We define an *algebra morphism* between two algebras $(x, f : Fx \to x)$ and $(x', f' : Fx' \to x')$ as a morphism $\nu : x \to x'$ which commutes with the two structure maps:

$\nu \circ f = f' \circ F\nu$

$$\begin{tikzcd} Fx \arrow[r, "F\nu"] \arrow[d, "f"'] & Fx' \arrow[d, "f'"] \\ x \arrow[r, "\nu"] & x' \end{tikzcd}$$

The initial object in the category of algebras is called the initial algebra, or the fixed point of the functor that generates these algebras. As the initial object, it has a unique algebra morphism to any other algebra. This unique morphism is called a *catamorphism*.

In Haskell, the fixed point of a functor `f` is defined recursively:

```
newtype Fix f = In { out :: f (Fix f) }
```

with, for instance:

```
type List a = Fix (ListF a)
```

A catamorphism is defined as:

```
cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

A list catamorphism is called `foldr`.

We want to show that the initial algebra La of the functor:

$\Phi a x = I + a \otimes x$

is a free monoid. Let's see under what conditions it is a monoid.

Initial Algebra is a Monoid

In this section I will show you how to concatenate lists the hard way.

We know that function type $b \to c$ (a.k.a., the exponential c^b) is the right adjoint to the product:

$Set(a \times b, c) \cong Set(a, b \to c)$

The function type is also called the internal hom.

In a monoidal category it's sometimes possible to define an internal hom-object, denoted $[b, c]$, as the right adjoint to the tensor product:

$curry : C(a \otimes b, c) \cong C(a, [b, c])$

If this adjoint exists, the category is called *closed monoidal*.

In a closed monoidal category, the initial algebra La of the functor $\Phi a x = I + a \otimes x$ is a monoid. (In particular, a Haskell list of `a`, which is a fixed point of `ListF a`, is a monoid.)

To show that, we have to construct two morphisms corresponding to unit and multiplication (in Haskell, empty list and concatenation):

$\eta : I \to La$

$\mu : La \otimes La \to La$

What we know is that La is a carrier of the initial algebra for Φa, so it is equipped with the structure map:

$$I + a \otimes La \to La$$

which is equivalent to a pair of morphisms:

$$\alpha : I \to La$$

$$\beta : a \otimes La \to La$$

Notice that, in Haskell, these correspond the two list constructors: `Nil` and `Cons` or, in terms of the fixed point:

```
nil :: () -> List a
nil () = In NilF

cons :: a -> List a -> List a
cons a as = In (ConsF a as)
```

We can immediately use α to implement η.

The second one, β, one can be rewritten using the hom adjuncion as:

$$\bar{\beta} = curry\, \beta$$

$$\bar{\beta} : a \to [La, La]$$

Notice that, if we could prove that $[La, La]$ is a carrier for the same algebra generated by Φa, we would know that there is a unique catamorphism from the initial algebra La:

$$\kappa_{[La, La]} : La \to [La, La]$$

which, by the hom adjunction, would give us the desired multiplication:

$$\mu : La \otimes La \to La$$

Let's establish some useful lemmas first.

Lemma 1: For any object x in a closed monoidal category, $[x, x]$ is a monoid.

This is a generalization of the idea that endomorphisms form a monoid, in which identity morphism is the unit and composition is multiplication. Here, the internal hom-object $[x, x]$ generalizes the set of endomorphisms.

Proof: The unit:

$$\eta : I \to [x, x]$$

follows, through adjunction, from the unit law in the monoidal category:

$$\lambda : I \otimes x \to x$$

(In Haskell, this is a fancy way of writing `mempty = id`.)

Multiplication takes the form:

$$\mu : [x, x] \otimes [x, x] \to [x, x]$$

which is reminiscent of composition of edomorphisms. In Haskell we would say:

```
mappend = (.)
```

By adjunction, we get:

$$curry^{-1}\, \mu : [x, x] \otimes [x, x] \otimes x \to x$$

We have at our disposal the counit $eval$ of the adjunction:

$$eval : [x, x] \otimes x \cong x$$

We can apply it twice to get:

$$\mu = curry(eval \circ (id \otimes eval))$$

In Haskell, we could express this as:

```
mu :: ((x -> x), (x -> x)) -> (x -> x)
mu (f, g) = \x -> f (g x)
```

Here, the counit of the adjunction turns into simple function application.

$\square$

Lemma 2: For every morphism $f : a \to m$, where m is a monoid, we can construct an algebra of the functor Φa with m as its carrier.

Proof: Since m is a monoid, we have two morphisms:

$$\eta : I \to m$$

$$\mu : m \otimes m \to m$$

To show that m is a carrier of our algebra, we need two morphisms:

$$\alpha : I \to m$$

$$\beta : a \otimes m \to m$$

The first one is the same as η , the second can be implemented as:

$$\beta = \mu \circ (f \otimes id)$$

In Haskell, we would do case analysis:

```
mapAlg :: Monoid m => ListF a m -> m
mapAlg NilF = mempty
mapAlg (ConsF a m) = f a `mappend` m
```

□

We can now build a larger proof. By lemma 1, $[La, La]$ is a monoid with:

$$\mu = curry(eval \circ (id \otimes eval))$$

We also have a morphism $\bar{\beta} : a \to [La, La]$ so, by lemma 2, $[La, La]$ is also a carrier for the algebra:

$$\alpha = \eta$$

$$\beta = \mu \circ (\bar{\beta} \otimes id)$$

It follows that there is a unique catamorphism $\kappa_{[La,La]}$ from the initial algebra La to it, and we know how to use it to implement monoidal multiplication for La . Therefore, La is a monoid.

Translating this to Haskell, $\bar{\beta}$ is the curried form of `Cons` and what we have shown is that concatenation (multiplication of lists) can be implemented as a catamorphism:

```
concat :: List a -> List a -> List a
conc x y = cata alg x y
  where alg NilF        = id
        alg (ConsF a t) = (cons a) . t
```

The type:

```
List a -> (List a -> List a)
```

(parentheses added for emphasis) corresponds to $La \to [La, La]$.

It's interesting that concatenation can be described in terms of the monoid of list endomorphisms. Think of turning an element `a` of the list into a transformation, which prepends this element to its argument (that's what $\bar{\beta}$ does). These transformations form a monoid. We have an algebra that turns the unit I into an identity transformation on lists, and a pair $a \otimes t$ (where t is a list transformation) into the composite $\bar{\beta} a \circ t$. The catamorphism for this algebra takes a list La and turns it into one composite list transformation. We then apply this transformation to another list and get the final result: the concatenation of two lists. □

Incidentally, lemma 2 also works in reverse: If a monoid m is a carrier of the algebra of Φa , then there is a morphism $f : a \to m$. This morphism can be thought of as inserting generators represented by a into the monoid m .

Proof: if m is both a monoid and a carrier for the algebra Φa , we can construct the morphism $a \to m$ by first applying the identity law to go from a to $a \otimes I$, then apply $id_a \otimes \eta$ to get $a \otimes m$. This can be right-injected into the coproduct $I + a \otimes m$ and then evaluated down to m using the structure map for the algebra on m .

$$a \to a \otimes I \to a \otimes m \to I + a \otimes m \to m$$

$$\begin{tikzcd} a \arrow[r, "\rho"] & a \otimes I \arrow[r, "id \otimes \eta"] & a \otimes m \arrow[r, "inr"] & I + a \otimes m \arrow[r, "{[\alpha, \beta]}"] & m \end{tikzcd}$$

In Haskell, this corresponds to a construction and evaluation of:

```
ConsF a mempty
```

□

Free Monoidal Functor

Let's go back to our functor category. We started with a monoidal category C and considered a functor category $[C, Set]$. We have shown that $[C, Set]$ is itself a monoidal category with Day convolution as tensor product and the hom functor $C(I, -)$ as unit. A monoid is this category is a lax monoidal functor.

The next step is to build a free monoid in $[C, Set]$, which would give us a free lax monoidal functor. We have just seen such a construction in an arbitrary closed monoidal category. We just have to translate it to $[C, Set]$. We do this by replacing objects with functors and morphisms with natural transformations.

Our construction relied on defining an initial algebra for the functor:

$$I + a \otimes b$$

Straightforward translation of this formula to the functor category $[C, Set]$ produces a higher order endofunctor:

$$A_F G = C(I, -) + F \star G$$

It defines, for any functor F, a mapping from a functor G to a functor $A_F G$. (It also maps natural transformations.)

We can now use A_F to define (higher-order) algebras. An algebra consists of a carrier — here, a functor T — and a structure map — here, a natural transformation:

$$A_F T \to T$$

The initial algebra for this higher-order endofunctor defines a monoid, and therefore a lax monoidal functor. We have shown it for an arbitrary *closed* monoidal category. So the only question is whether our functor category with Day convolution is closed.

We want to define the internal hom-object in $[C, Set]$ that satisfies the adjunction:

$$[C, Set](F \star G, H) \cong [C, Set](F, [G, H])$$

We start with the set of natural transformations — the hom-set in $[C, Set]$:

$$[C, Set](F \star G, H)$$

We rewrite it as an end over c, and use the formula for Day convolution:

$$\int_c Set(\int^{ab} C(a \otimes b, c) \times Fa \times Gb, Hc)$$

We use the co-continuity trick to pull the coend out of the hom-set and turn it into an end:

$$\int_{cab} Set(C(a \otimes b, c) \times Fa \times Gb, Hc)$$

Keeping in mind that our goal is to end up with Fa on the left, we use the regular hom-set adjunction to shuffle the other two terms to the right:

$$\int_{cab} Set(Fa, C(a \otimes b, c) \times Gb \to Hc)$$

The hom-functor is continuous in the second argument, so we can sneak the end over bc under it:

$$\int_a Set(Fa, \int_{bc} C(a \otimes b, c) \times Gb \to Hc)$$

We end up with a set of natural transformations from the functor F to the functor we will call:

$$[G, H] = \int_{bc} (C(- \otimes b, c) \times Gb \to Hc)$$

We therefore identify this functor as the right adjoint (internal hom-object) for Day convolution. We can further simplify it by using the hom-set adjunction:

$$\int_{bc} (C(- \otimes b, c) \to (Gb \to Hc))$$

and applying the Yoneda lemma to get:

$$[G, H] = \int_b (Gb \to H(- \otimes b))$$

In Haskell, we would write it as:

```
newtype DayHom f g a = DayHom (forall b . f b -> g (a, b))
```

Since Day convolution has a right adjoint, we conclude that the fixed point of our higher order functor defines a free lax monoidal functor. We can write it in a recursive form as:

$$Free_F = C(I, -) + F \star Free_F$$

or, in Haskell:

```
data FreeMonR f t =
      Done t
    | More (Day f (FreeMonR f) t)
```

Free Monad

This blog post wouldn't be complete without mentioning that the same construction works for monads. Famously, a monad is a monoid in the category of endofunctors. Endofunctors form a monoidal category with functor composition as tensor product and the identity functor as unit. The fact that we can construct a free monad using the formula:

$$FreeM_F = Id + F \circ FreeM_F$$

is due to the observation that functor composition has a right adjoint, which is the right Kan extension. Unfortunately, due to size issues, this Kan extension doesn't always exist. I'll quote Alex Campbell here: "By making suitable size restrictions, we can give conditions for free monads to exist: for example, free monads exist for accessible endofunctors on locally presentable categories; a special case is that free monads exist for finitary endofunctors on Set, where *finitary* means the endofunctor preserves filtered colimits (more generally, an endofunctor is accessible if it preserves κ-filtered colimits for some regular cardinal number κ)."

Conclusion

As we acquire experience in programming, we learn more tricks of trade. A seasoned programmer knows how to read a file, parse its contents, or sort an array. In category theory we use a different bag of tricks. We bunch morphisms into hom-sets, move ends and coends, use Yoneda to "integrate," use adjunctions to shuffle things around, and use initial algebras to define recursive types.

Results derived in category theory can be translated to definitions of functions or data structures in programming languages. A lax monoidal functor becomes an `Applicative`. Free monoidal functor becomes:

```
data FreeMonR f t =
      Done t
    | More (Day f (FreeMonR f) t)
```

What's more, since the derivation made very few assumptions about the category C (other than that it's monoidal), this result can be immediately applied to profunctors (replacing C with $C^{op} \times C$) to produce:

```
data FreeMon p s t where
     DoneFM :: t -> FreeMon p s t
     MoreFM :: p a b -> FreeMon p c d ->
                        (b -> d -> t) ->
                        (s -> (a, c)) -> FreeMon p s t
```

Replacing Day convolution with endofunctor composition gives us a free monad:

```
data FreeMonadF f g a =
    DoneFM a
  | MoreFM (Compose f g a)
```

Category theory is also the source of laws (commuting diagrams) that can be used in equational reasoning to verify the correctness of programming constructs.

Writing this post has been a great learning experience. Every time I got stuck, I would ask Alex for help, and he would immediately come up with yet another algebra and yet another catamorphism. This was so different from the approach I would normally take, which would be to get bogged down in inductive proofs over recursive data structures.

14 Responses to "Free Monoidal Functors, Categorically!"

1. Juan Manuel (@babui_) Says:

   May 18, 2018 at 12:34 am
   In the associativity law diagram:

   ```
   Associativity law: \alpha is the associator in the appropriate category (top arrow, in Set; bottom arrow, in C).
   ```

   But in the diagram \alpha is associating Fa x (Fb x Fc) with (Fa x Fb) x Fc so it is the associativity of the cartesian product of Sets, isn't it? So the top arrow associator (which is in Set) cannot be \alpha.

   Well, or I can be completely wrong 😀

   Juan Manuel

2. Bartosz Milewski Says:

   May 18, 2018 at 10:26 am
   What I meant was that these are two different alphas. In programming, we are used to overloading names.

3. Vlad Patryshev Says:

   May 18, 2018 at 11:52 am
   Thank you! While reading it, I figured you can as well draw images on my presheaf:

4. vpatryshev Says:

   May 18, 2018 at 3:07 pm
   Sorry, not sure if my previous comment got through.
   I see you draw diagrams with your hand; but you can use my presheaf.com, like this:

5. vpatryshev Says:

   May 18, 2018 at 3:09 pm
   (wordpress problems) I mean, http://presheaf.com/?d=d5z5zq1731w475o691s194s6s4t4s

6. Bartosz Milewski Says:

   May 18, 2018 at 5:21 pm
   @Vlad: For some reason, whatever you're trying to embed, doesn't pass through wordpress. I had a look at your website and the documentation. It is a little intimidating. It would be nice to have a wysiwyg application.

7. edmundsecho Says:

   May 22, 2018 at 10:02 pm
   Hi Bartosz, I like the conclusion. How the same construction can produce a monad helped to complete the picture. I'm a little unclear of the significance of Lax (Applicative) being a one-way arrow because it only needs to consider
   the monoidal structure (I'm assuming you're referring to the Functor layer in `m a` or `f a`). The reason I'm unclear is because I'm having a tough time imagining any meaningful monoidal computation where we don't loose information once we combine the two Functor values to produce the single result value (3 + 1 -> 4, but I can't map a 1:1 solution of 1 and 3 from 4). So I get that we don't have an isomorphism because we cannot define a valid reverse function. Can I say the same thing for monoidal computation embedded in the monad? I'm getting the impression from the blog the answer is, no. That would mean that every `m b` in the range of the `k :: a -> m b` has the information required to reproduce the `m a` value in the domain. There seems to be something missing in the constraints for that to be true. I assume that has to do with what Alex described.

   But more concretely, take for example a `k` that generates `Nothing` in one of two situations, in one of 2 values of `a` (say 0 and 1 where `a :: Int`). If the value of `m b` ends up being `Nothing` I can't say which of the two possible `a` values, 0 or 1, produced the `Nothing` result. Does what Alex describe explain/constrain for this possibility? Furthermore, I can't even determine if the `Nothing` was produced by the `m` value in the `m a` or was it from the return value of `k` in the `bind` operation. Categorically the monad could be behaving like Applicative in that it is propagating `Nothing`, not actually running `k` because there is no `a`. Alternatively, it could behave like a monad where the `Nothing` was the result of the new, dominant `m` value generated by `k`, distinctively monadic. But even then, it's not isomorphic in the example I gave. Applicative generates the same monoidal result except will apply the `f a` computations regardless, and where `f` cannot change value. But similarly, once I perform the monoidal join of the multiple `f` values there is no way to reproduce, or map back to the inputs.

   There is only one scenario I can imagine where the monad is isomorphic: If the *interaction* or the product of `m b` is unique for every combination of `m` and `a`. That is a pretty high bar to have to meet. In other words, rare. In statistics there is a precise understanding of the term *interaction*. If present, the model cannot predict the *change* in the value of `y` without knowing the *absolute* values of both factors where an interaction was demonstrated.

   For instance is we wanted to know, on average, did the medication improve the health of the patients? If you can answer "yes", and be correct the same "number of times" whether you are describing old versus young, big versus small, male versus female, married versus single etc., then there is no interaction. On the other hand, if you have to answer the question with "it depends on what patient group", then you have an interaction; e.g., patients with long curly hair had improved health, but those with short blonde hair did not experience a change in the course of the disease compared to controls, then we have an interaction `treatment success * hair color` that explains the range of responses to treatment. When there is an interaction, the only way to predict the average treatment response (the delta) is if we know both which patients were treated And their hair style; only with that additional information will we know the effectiveness of the treatment. That's bad news for marketing being able to tell a simple story, but good news for evidence of an isomorphism; I can tell you who was treated and their hairstyle based on the response to treatment.

   In statistics a group of individuals is a single value in our set; a collection of people becomes a distinct data point, a distinct group when the average of that group is statistically different than the average of some other collection of people. In the interaction scenario I described, I need 2 arrows to represent the two functions that once composed, encode the findings (one to represent the treatment effect, the other to encode the interaction). What is special about an interaction, is that if we only applied one of the functions, we end up in a place that does not actually exist, was never demonstrated statistically. If on the other hand, on average everyone responded the same, positive way to the medication, I would only need 1 arrow to connect the untreated and treated patients; the transformation encoded in the body of the arrow is the treatment effect. But if all I knew is that the patient responded well to the medication, I cannot tell you anything about the person's hair; that encoding is lost where the interaction does not exist.

   All this to say, `k :: a -> m b` is required in our programming designs when we cannot express, when there is literally no way of representing `b` without `m` to represent anything that is useful. If this is the case, there is an interaction between `m` and `b` that is encoded, informed in the body of `k` given the values of `m` and `a` in `m a`. The body := the biology of the medication that for some crazy reason cares about hair style, the `m` and `a` encode whether the person was treated, and the style of the patient's hair.

   I have not read a better explanation of the relationships you described in your blog post. This said, as close as you have gotten, the Categorical explanations of Applicative compared to Monad for instance, should somehow be able to capture this concept of interaction; how dependent is our abstraction on the simultaneous expression of `m` and `a`. FP and Haskell in particular, delivers on the capacity of computing with limited reliance on state. Managing state is expensive but mentally easy.

One last area where Category seems to fall short: the type signatures are sometimes a by-product of a technicality as opposed to reflecting a Categorical intent (e.g., `fmap` for `F f`, `return` as a natural transformation). It just so happens that with `bind` we see how we cannot start computing `k` until we have completed `m a` (often, but by no means a requirement, the result of the previous `k0`; it could have been produce using unit/return); as such, one could argue, *by happenstance*, this monadic feature is represented in our Categorical drawings. But, this often sited monadic quality, is similar in many regards to `f a` given how we rarely operate on `f a` but rather require that we first extract `a` from `f a` before delegating the application of a lifted function to fmap (we are not required to delegation application in Applicative to `fmap`, it just clarifies the "one in the same" tasks shared between `fmap` and `(<*>)`.

When I first heard you lecture on YouTube about five years ago now, I was truly impressed with your imagination and capacity to tie together what seemed like disparate concepts. Category still has gaps for how we need this tool in computing. Are there new rules and encodings that you can conceive of and can start using? I'm talking about Categorical tools not bound by the limits of the original intent to solving problems in Physics and Calculus? A Categorical approach that captures what we experience and know to be foundational in how we imagine, conceive and deploy our ideas and intellect in Haskell? Can we build a Categorical approach that will facilitate getting us to a place where for instance, Ed Kmett got to when he conceived and implemented the elegance and power of Lenses? If it's possible, Bartosz, you are on your way to doing it. Keep pushing!

- E

8. xieyuheng Says:

   June 18, 2018 at 9:14 am
   is there a categorical construction to generalize arrow composition,
   by allowing domain and codomain to be refined (or changed) by the
   composition ?

   this construction would be useful for
   forming theoretical background of dependent type system.

   for example, compose two functions
   f : (A x -> B y)
   g : (B n -> C z)
   will give us a function of type (A n -> C z)

   another example would be the following generalized composition in cartesian closed category :

```
 f   : (t1, t2) -> (t3, t4)
g   : (t, t3, t4) -> (t6, t7)
f;g : (t, t1, t2) -> (t6, t7)
```

   and

```
 f   : (t1, t2) -> (t, t3, t4)
g   : (t3, t4) -> (t6, t7)
f;g : (t, t1, t2) -> (t, t6, t7)
```

   xieyuheng

9. Bartosz Milewski Says:

   June 18, 2018 at 12:57 pm
   The standard categorical interpretation of dependent type systems is done using fibrations. There is also a generalization of a category that involves morphisms that have multiple sources (multiple targets are easy to encode using products). It's called an operad.

10. Resumen de lecturas compartidas durante mayo de 2018 | Vestigium Says:

    July 10, 2018 at 12:12 am
    […] Free monoidal functors, categorically! ~ Bartosz Milewski (@BartoszMilewski) #haskell #CategoryTheory […]

11. xieyuheng Says:

    July 17, 2018 at 1:35 am
    Thank you professor, Thanks for your reply.

    I apologize to post offtopic comments here,
    because I read your book and wish to learn more from you,
    but I do not know how to contact you otherwise.

    I am trying to provide a categorical model of dependent type systems,
    by using what I learned from "What is Unification?" — a paper by Joseph A. Goguen,
    i.e. to use equalizer to define unification,
    and when `B` is unified with `C` in `(A -> B) and (C -> D)`,
    using pullback and pushout to apply the equalizer to `A` and `D`.
    Also I have a project proposal here :: https://xieyuheng.github.io/cicada

12. Bartosz Milewski Says:

    July 17, 2018 at 11:16 am

Have you looked at the Idris language? Also, Agda and Coq are two very mathematical languages worth studying.

13. xieyuheng Says:

    July 17, 2018 at 11:46 am
    Yes, I have.

    In Idris and Agda, to model category theory,
    record-type is needed (for its first-class-ness),
    but record-type in them are not inheritable, and not easy to use.

    I tried myself
    https://github.com/xieyuheng/pure/blob/master/category.agda
    https://github.com/xieyuheng/idris-category/blob/master/Category.idr

    And I read a lot (all that i can find) about how other people formalize category theory in Idris, Agda and Coq.
    I am not satisfied with those languages, for very concrete reasons.

14. xieyuheng Says:

    July 17, 2018 at 1:30 pm
    I am using partly inhabited typed record as type,
    and fully inhabited typed record inhabits such type.

    Let's call it *fulfilling type system*.

    This allows types to be used in a more flexible way,
    and makes sub-type relation easily expressed.