# Bartosz Milewski's Programming Cafe

*Category Theory, Haskell, Concurrency, C++*

**June 15, 2016**

## Free/Forgetful Adjunctions

Posted by Bartosz Milewski under [Programming](Programming)
[[17] Comments](17)

i
17 Votes

*This is part 19 of Categories for Programmers. Previously: [Adjunctions](Adjunctions). See the [Table of Contents](Table of Contents).*

## Free Monoid from Adjunction

Free constructions are a powerful application of adjunctions. A *free functor* is defined as the left adjoint to a *forgetful functor*. A forgetful functor is usually a pretty simple functor that forgets some structure. For instance, lots of interesting categories are built on top of sets. But categorical objects, which abstract those sets, have no internal structure — they have no elements. Still, those objects often carry the memory of sets, in the sense that there is a mapping — a functor — from a given category $C$ to **Set**. A set corresponding to some object in $C$ is called its *underlying set*.

Monoids are such objects that have underlying sets — sets of elements. There is a forgetful functor $U$ from the category of monoids **Mon** to the category of sets, which maps monoids to their underlying sets. It also maps monoid morphisms (homomorphisms) to functions between sets.

I like to think of **Mon** as having split personality. On the one hand, it's a bunch of sets with multiplication and unit elements. On the other hand, it's a category with featureless objects whose only structure is encoded in morphisms that go between them. Every set-function that preserves multiplication and unit gives rise to a morphism in **Mon**.

Things to keep in mind:

- There may be many monoids that map to the same set, and
- There are fewer (or at most as many as) monoid morphisms than there are functions between their underlying sets.
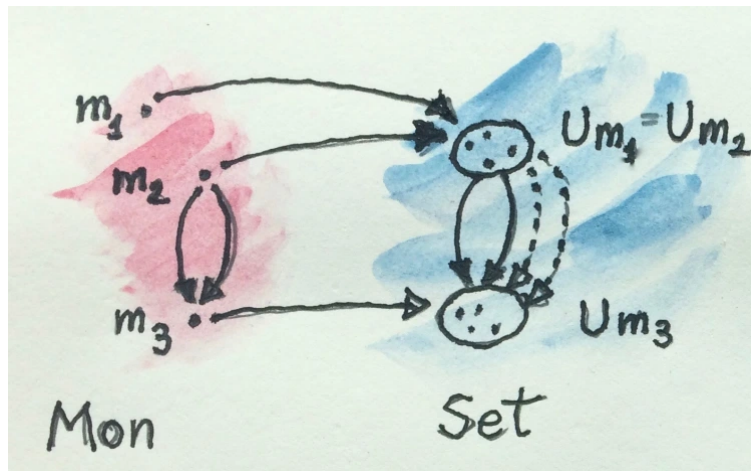


Monoids $m_1$ and $m_2$ have the same underlying set. There are more functions between the underlying sets of $m_2$ and $m_3$ than there are morphisms between them.

The functor `F` that's the left adjoint to the forgetful functor `U` is the free functor that builds free monoids from their generator sets. The adjunction follows from the _free monoid universal construction_ we've discussed before.

In terms of hom-sets, we can write this adjunction as:

```
Mon(F x, m)  ≅ Set(x, U m)
```

This (natural in `x` and `m`) isomorphism tells us that:

- For every monoid homomorphism between the free monoid `F x` generated by `x` and an arbitrary monoid `m` there is a unique function that embeds the set of generators `x` in the underlying set of `m`. It's a function in `Set(x, U m)`.
- For every function that embeds `x` in the underlying set of some `m` there is a unique monoid morphism between the free monoid generated by `x` and the monoid `m`. (This is the morphism we called `h` in our universal construction.)
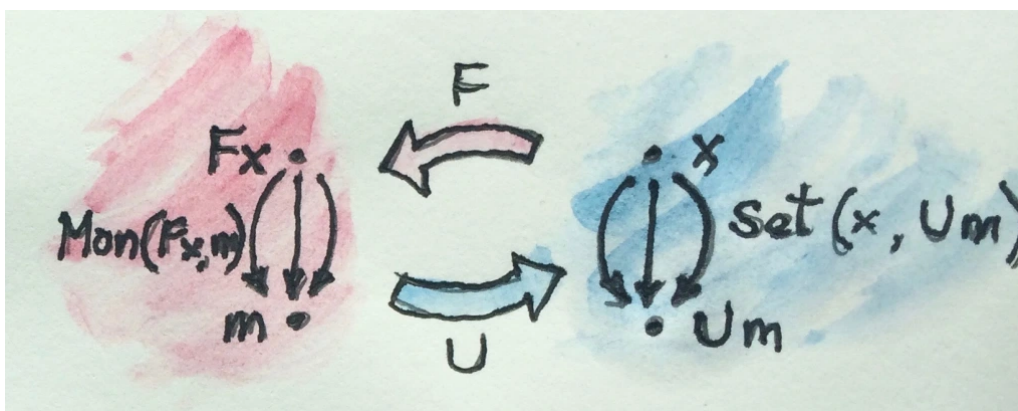
The intuition is that `F x` is the "maximum" monoid that can be built on the basis of `x`. If we could look inside monoids, we would see that any morphism that belongs to `Mon(F x, m)` *embeds* this free monoid in some other monoid `m`. It does it by possibly identifying some elements. In particular, it embeds the generators of `F x` (i.e., the elements of `x`) in `m`. The adjunction shows that the embedding of `x`, which is given by a function from `Set(x, U m)` on the right, uniquely determines the embedding of monoids on the left, and vice versa.

In Haskell, the list data structure is a free monoid (with some caveats: see [Dan Doel's blog post](#)). A list type `[a]` is a free monoid with the type `a` representing the set of generators. For instance, the type `[Char]` contains the unit element — the empty list `[]` — and the singletons like `['a']`, `['b']` — the generators of the free monoid. The rest is generated by applying the "product." Here, the product of two lists simply appends one to another. Appending is associative and unital (that is, there is a neutral element — here, the empty list). A free monoid generated by `Char` is nothing but the set of all strings of characters from `Char`. It's called `String` in Haskell:

```
type String = [Char]
```

(`type` defines a type synonym — a different name for an existing type).

Another interesting example is a free monoid built from just one generator. It's the type of the list of units, `[()]`. Its elements are `[]`, `[()]`, `[(), ()]`, etc. Every such list can be described by one natural number — its length. There is no more information encoded in the list of units. Appending two such lists produces a new list whose length is the sum of the lengths of its constituents. It's easy to see that the type `[()]` is isomorphic to the additive monoid of natural numbers (with zero). Here are the two functions that are the inverse of each other, witnessing this isomorphism:

```
toNat :: [()] -> Int
toNat = length

toLst :: Int -> [()]
toLst n = replicate n ()
```

For simplicity I used the type `Int` rather than `Natural`, but the idea is the same. The function `replicate` creates a list of length `n` pre-filled with a given value — here, the unit.

# Some Intuitions

What follows are some hand-waving arguments. Those kind of arguments are far from rigorous, but they help in forming intuitions.

To get some intuition about the free/forgetful adjunctions it helps to keep in mind that functors and functions are lossy in nature. Functors may collapse multiple objects and morphisms, functions may bunch together multiple elements of a set. Also, their image may cover only part of their codomain.

An "average" hom-set in **Set** will contain a whole spectrum of functions starting with the ones that are least lossy (e.g., injections or, possibly, isomorphisms) and ending with constant functions that collapse the whole domain to a single element (if there is one).

I tend to think of morphisms in an arbitrary category as being lossy too. It's just a mental model, but it's a useful one, especially when thinking of adjunctions — in particular those in which one of the categories is **Set**.

Formally, we can only speak of morphisms that are invertible (isomorphisms) or non-invertible. It's that latter kind that may be though of as lossy. There is also a notion of mono- and epi- morphisms that generalize the idea of injective (non-collapsing) and surjective (covering the whole codomain) functions, but it's possible to have a morphism that is both mono and epi, and which is still non-invertible.

In the Free ⊣ Forgetful adjunction, we have the more constrained category $C$ on the left, and a less constrained category $D$ on the right. Morphisms in $C$ are "fewer" because they have to preserve some additional structure. In the case of **Mon**, they have to preserve multiplication and unit. Morphisms in $D$ don't have to preserve as much structure, so there are "more" of them.

When we apply a forgetful functor `U` to an object `c` in $C$, we think of it as revealing the "internal structure" of `c`. In fact, if $D$ is **Set** we think of `U` as *defining* the internal structure of `c` — its underlying set. (In an arbitrary category, we can't talk about the internals of an object other than through its connections to other objects, but here we are just hand-waving.)

If we map two objects `c'` and `c` using `U`, we expect that, in general, the mapping of the hom-set `C(c', c)` will cover only a subset of `D(U c', U c)`. That's because morphisms in `C(c', c)` have to preserve the additional structure, whereas the ones in `D(U c', U c)` don't.
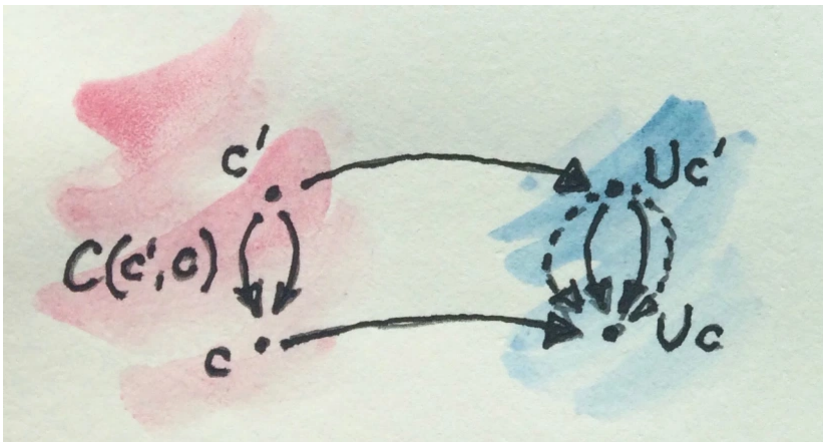


But since an adjunction is defined as an *isomporphism* of particular hom-sets, we have to be very picky with our selection of `c'`. In the adjunction, `c'` is picked not from just anywhere in $C$, but from the (presumably smaller) image of the free functor `F`:

`C(F d, c) ≅ D(d, U c)`

The image of `F` must therefore consist of objects that have lots of morphisms going to an arbitrary `c`. In fact, there has to be as many structure-preserving morphisms from `F d` to `c` as there are non-structure preserving morphisms from `d` to `U c`. It means that the image of `F` must consist of essentially structure-free objects (so that there is no structure to preserve by morphisms). Such "structure-free" objects are called free objects.
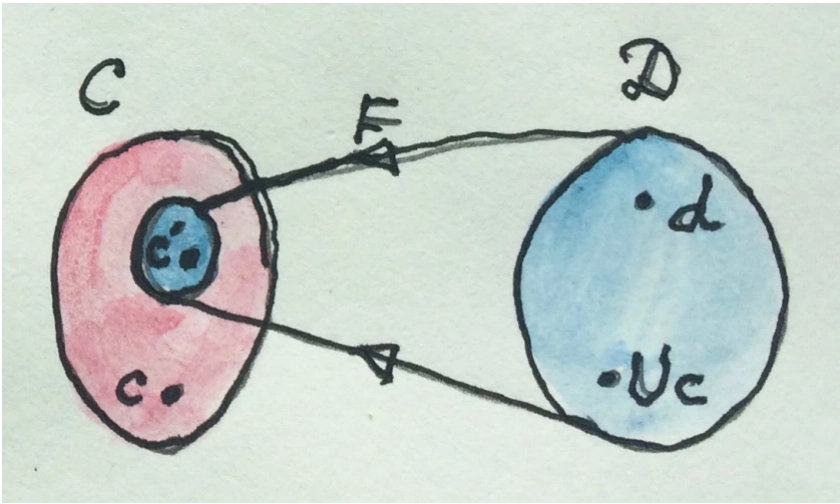
In the monoid example, a free monoid has no structure other than what's generated by unit and associativity laws. Other than that, all multiplications produce brand new elements.

In a free monoid, 2*3 is not 6 — it's a new element [2, 3]. Since there is no identification of [2, 3] and 6, a morphism from this free monoid to any other monoid `m` is allowed to map them separately. But it's also okay for it to map both [2, 3] and 6 (their product) to the same element of `m`. Or to identify [2, 3] and 5 (their sum) in an additive monoid, and so on. Different identifications give you different monoids.

This leads to another interesting intuition: Free monoids, instead of performing the monoidal operation, accumulate the arguments that were passed to it. Instead of multiplying 2 and 3 they remember 2 and 3 in a list. The advantage of this scheme is that we don't have to specify what monoidal operation we will use. We can keep accumulating arguments, and only at the end apply an operator to the result. And it's then that we can chose what operator to apply. We can add the numbers, or multiply them, or perform addition modulo 2, and so on. A free monoid separates the creation of an expression from its evaluation. We'll see this idea again when we talk about algebras.

This intuition generalizes to other, more elaborate free constructions. For instance, we can accumulate whole expression trees before evaluating them. The advantage of this approach is that we can transform such trees to make the evaluation faster or less memory consuming. This is, for instance, done in implementing matrix calculus, where eager evaluation would lead to lots of allocations of temporary arrays to store intermediate results.

# Challenges

1. Consider a free monoid built from a singleton set as its generator. Show that there is a one-to-one correspondence between morphisms from this free monoid to any monoid `m`, and functions from the singleton set to the underlying set of `m`.

Next: <u>Monads: Programmer's Definition</u>.

# 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

# 17 Responses to "Free/Forgetful Adjunctions"

1. The Week in C++ 2016.24: June 11th-18th | Utah C++ Users Group Says:

   June 22, 2016 at 10:16 am
   […] Free/Forgetful Adjunctions by Bartosz Milewski […]

2. Bogdan Popescu Says:

   December 10, 2018 at 5:43 am
   I have a question. For the generator set x = {2, 3}.
   We have F x monoid build using concatenation and that has the underlying set
   {1, 2, 3, (2, 2), (2, 3), (3, 2), etc. }

   From this F x we have a homomorphism to the multiplication monoid m
   h(2) = 2, h(3) = 3, h(2, 3) = 6, h(3, 2) = 6, etc.
   that has the underlying set
   Um = { 1, 2, 3, 4, 6, 9, etc.}

   There has to be as many structure preserving morphisms from F x to m as there are non-structure preserving morphisms from x to Um.

   But from x = {2, 3} we can have a lot of mappings to Um.
   One is 2 -> 2 and 3 -> 3. Another could be 2 -> 4 and 3 -> 6.

   Does this mean that for this second x to Um mapping, the homomorphism from F x to m would be something like this?
   h'(2) = 4, h'(3) = 6, h'(2, 3) = 24, h'(3, 3) = 36, etc.
   (elements 24, 36, etc. are also part of the same Um set)

3. Bogdan Popescu Says:

   December 11, 2018 at 2:04 am
   So I guess my question is, if we clearly have multiple functions (morphisms in Set) from x to Um and the two home sets are isomorphic, how can I find the morphisms from F x to m? In the example above, besides the obvious "evaluation" morphism, that transforms the concatenation in

multiplication.

4. <u>Bartosz Milewski</u> Says:

<u>December 11, 2018 at 10:30 am</u>

A morphism from F x to m must preserve monoidal structure, therefore it must transform concatenation to multiplication.

5. Bogdan Popescu Says:

<u>December 11, 2018 at 11:56 am</u>

So in order to have multiple homomorphisms between F x and m (as many as functions from x to Um), we need to vary something. I was wondering if using the generic h(a concat b) = h(a) * h(b), we will obtain multiple homomorphisms by making h(i) = f(i) where "i in x" for the various possible versions of f : x -> Um (e.g. f(2) = 2, f(3) = 3, and f'(2) = 4, f'(3) = 6 etc.)
Is this correct?

6. <u>Bartosz Milewski</u> Says:

<u>December 12, 2018 at 9:42 pm</u>

I wish you called the elements of x {a, b}. Once you fix the mapping of a and b, the rest just follows.

7. Fabricio Olivetti de França Says:

<u>February 16, 2019 at 1:58 am</u>

I have a question, if it wasn't for the Num constraint, could we say that [a] and Sum a are adjuncts?
So `Sum a` forgets the structure of `[a]` (the free monoid):

```
1  instance Adjunction [] Sum where
2    unit x           = Sum [x]
3    counit           = getSum . mconcat
4    leftAdjunct f   = fmap f . unit
5    rightAdjunct f = counit . fmap f
```

8. <u>Fabricio O de França (@fabriciolivetti)</u> Says:

<u>February 16, 2019 at 6:25 am</u>

Thinking about the left and right adjoints of Free monoids, would be correct to think that `[a]` and `Sum a` are free and forgetful monoids, respectively (if it wasn't for the Num restriction to type parameter `a`):

```
1  instance Adjunction [] Sum where
2    unit x           = Sum [x]
3    counit           = getSum . mconcat
4    leftAdjunct f   = fmap f . unit
5    rightAdjunct f = counit . fmap f
```

The correct right adjunction for the Free Monoid would be outside Haskell category? Something like

```
1   instance Adjunction [] Mon where
2     unit x            = Mon [x]
3     counit            = getMon . mconcat
4     leftAdjunct f   = fmap f . unit
5     rightAdjunct f = counit . fmap f
```

Is my intuition correct?

9. <u>Bartosz Milewski</u> Says:

<u>February 16, 2019 at 10:58 am</u>
But without the Num constraint this doesn't work.

10. Fabricio Olivetti de França Says:

<u>February 18, 2019 at 2:25 am</u>
Indeed it doesn't, I was trying to get a concrete example to better understand the concept. Regarding Free Monoids, there's no way to create an instance of Adjunction in Haskell, right?

11. <u>uhotspot</u> Says:

<u>October 20, 2019 at 4:33 pm</u>
Would you please provide a simple example for following two statements?

There may be many monoids that map to the same set, and
There are fewer (or at most as many as) monoid morphisms than there are functions between their underlying sets.

12. <u>Bartosz Milewski</u> Says:

<u>October 20, 2019 at 5:46 pm</u>
The standard example is the set of natural numbers underlying two different monoids, one with addition and zero, and another with multiplication and one. A monoid morphism between these two would have to map zero to one (an example would be $n \to 2^n$). But there are many functions from natural numbers to natural numbers that don't map zero to one, much less sum to product.

13. <u>uhotspot</u> Says:

<u>October 21, 2019 at 10:49 am</u>
But how would n -> 2^n map sum to product? Maybe I haven't got the essence of category of monoids! I would really appreciate if you clarify it for me.

14. <u>Bartosz Milewski</u> Says:

<u>October 21, 2019 at 11:54 am</u>
2^(n + m) = 2^n * 2^m

15. sahinka Says:

<u>May 3, 2020 at 12:05 pm</u>

"An "average" hom-set in Set will contain a whole spectrum of functions starting with the ones that are least lossy (e.g., injections or, possibly, isomorphisms) and ending with constant functions that collapse the whole domain to a single element (*if there is one*)."

What do you mean by "if there is one"? Constant functions always exist.

16. <u>Bartosz Milewski</u> Says:

   <u>May 3, 2020 at 6:54 pm</u>
   Not if the target is empty

17. sahinka Says:

   <u>May 3, 2020 at 7:32 pm</u>
   Can I suggest rephrasing the sentence? It sounds like constant functions sometimes exist but the injective functions always exist.

This site uses Akismet to reduce spam. <u>Learn how your comment data is processed</u>.


<u>Blog at WordPress.com.</u>