Bartosz Milewski's Programming Cafe

Category Theory, Haskell, Concurrency, C++

April 22, 2020

Terminal Coalgebra as Directed Limit

Posted by Bartosz Milewski under Programming
1 Comment

i
3 Votes

Previously, we talked about the construction of initial algebras. The dual construction is that of terminal coalgebras. Just like an algebra can be used to fold a recursive data structure into a single value, a coalgebra can do the reverse: it lets us build a recursive data structure from a single seed.

Here's a simple example. We define a tree that stores values in its nodes

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

We can build such a tree from a single list as our seed. We can choose the algorithm in such a way that the tree is ordered in a particular way

```
split :: Ord a => [a] -> Tree a
split [] = Leaf
split (a : as) = Node a (split l) (split r)
  where (l, r) = partition (<a) as
```

A traversal of this tree will produce a sorted list. We'll get back to this example after working out some theory behind it.

The functor

The tree in our example can be derived from the functor

```
data TreeF a x = LeafF | NodeF a x x

instance Functor (TreeF a) where
  fmap _ LeafF = LeafF
  fmap f (NodeF a x y) = NodeF a (f x) (f y)
```

Let's simplify the problem and forget about the payload of the tree. We're interested in the functor

```
data F x = LeafF | NodeF x x
```

Remember that, in the construction of the initial algebra, we were applying consecutive powers of the functor to the initial object. The dual construction of the terminal coalgebra involves applying powers of the functor to the terminal object: the unit type `()` in Haskell, or the singleton set 1 in Set.

Let's build a few such trees. Here are a some terms generated by the single power of `F`

```
w1, u1 :: F ()
w1 = LeafF
u1 = NodeF () ()
```

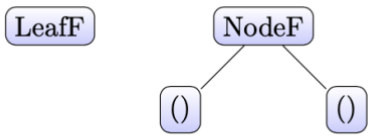


FIGURE 1. Trees generated by `F ()`

And here are some generated by the square of `F` acting on `()`

```
w2, u2, t2, s2 :: F (F ())

w2 = LeafF
u2 = NodeF w1 w1
t2 = NodeF w1 u1
s2 = NodeF u1 u1
```

Or, graphically

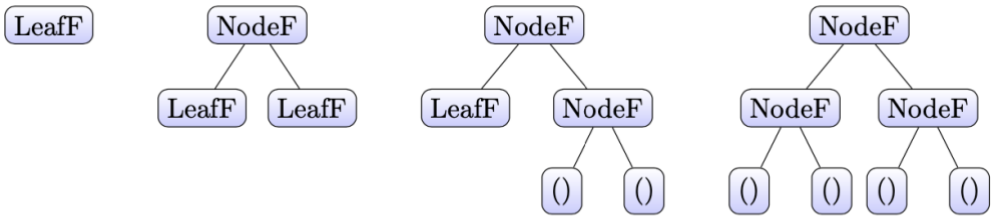


FIGURE 2. Examples of trees generated by $F^2 1$ or `F (F ())`

Notice that we are getting two kinds of trees, ones that have units `()` in their leaves and ones that don't. Units may appear only at the $(n+1)$-st layer (root being the first layer) of F^n.

We are also getting some duplication between different powers of F. For instance, we get a single `LeafF` at the F level and another one at the F^2 level (in fact, at every consecutive level after that as well). The node with two `LeafF` leaves appears at every level starting with F^2, and so on. The trees without unit leaves are the ones we are familiar with—they are the finite trees. The ones with unit leaves are new and, as we will see, they will contribute infinite trees to the terminal coalgebra . We'll construct the terminal coalgebra as a limit of an ω-chain.

Terminal coalgebra as a limit

As was the case with initial algebras, we'll construct a chain of powers of F, except that we'll start with the terminal rather than the initial object, and we'll use a different morphism to link them together. By definition, there is only one morphism from any object to the terminal object. In category theory, we'll call this morphism $¡ : a \to 1$ (upside-down exclamation mark) and implement it in Haskell as a polymorphic function

```
unit :: a -> ()
unit _ = ()
```

First, we'll use $¡$ to connect $F1$ to 1, then lift $¡$ to connect $F^2 1$ to $F1$, and so on, using $F^n ¡$ to transform $F^{n+1}1$ to $F^n 1$.

$$1 \xleftarrow[¡]{} F1 \xleftarrow[F¡]{} F^2 1 \xleftarrow[F^2¡]{} F^3 1 \leftarrow\text{-----} \ldots$$

Let's see how it works in Haskell. Applying `unit` directly to `F ()` turns it into `()`.

Values of the type `F (F ())` are mapped to values of the type `F ()`

```
w2' = fmap unit w2
> LeafF
u2' = fmap unit u2
> NodeF () ()
t2' = fmap unit t2
> NodeF () ()
s2' = fmap unit s2
> NodeF () ()
```

and so on.

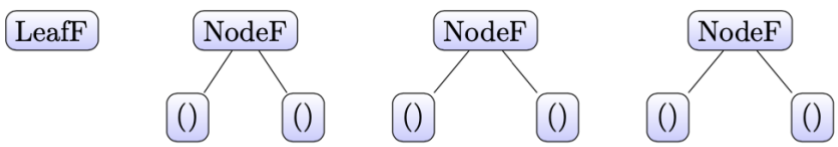


FIGURE 3. Examples of `fmap unit` acting on trees from the previous figure

The following pattern emerges. $F^n 1$ contains trees that end with either leaves (at any level) or values of the unit type (only at the lowest, $(n+1)$-st level). The lifted morphism F^{n-1}¡ (the $(n-1)$ st power of `fmap` acting on `unit`) operates strictly on the n th level of a tree. It turns leaves and two-unit-nodes into single units `()`.

Alternatively, we can look at the preimages of these mappings—conceptually reversing the arrows. Observe that all trees at the F^2 level can be seen as generated from the trees at the F level by replacing every unit `()` with either a leaf `LeafF` or a node `NodeF ()()`.

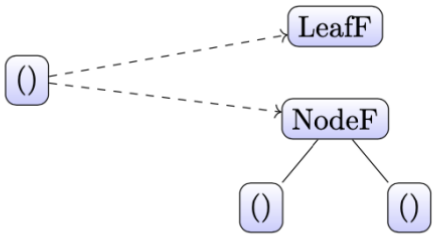


FIGURE 4. Universal seed becomes either a leaf or a two-seed-node

It's as if a unit were a *universal seed* that can either sprout a leaf or a two-unit-node. We'll see later that this process of growing recursive data structures from seeds corresponds to anamorphisms. Here, the terminal object plays the role of a universal seed that may give rise to two parallel universes. These correspond to the inverse image (a so-called fiber) of the lifted `unit`.

Now that we have an ω-chain, we can define its limit. It's easier to understand a limit in the category of sets. A limit in Set is a set of cones whose apex is the singleton set.

The simplest example of a limit is a product of sets. In that case, a cone with a singleton at the apex corresponds to a selection of elements, one per set. This agrees with our understanding of a product as a set of tuples.

A limit of a directed *finite* chain is just the starting set of the chain (the rightmost object in our pictures). This is because all projections, except for the rightmost one, are determined by commuting triangles. In the example below, π_b is determined by π_a:

$$\pi_b = f_1 \circ \pi_a$$

and so on. Here, every cone from 1 is fully determined by a function $1 \to a$, and the set of such functions is isomorphic to a.

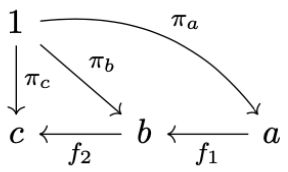
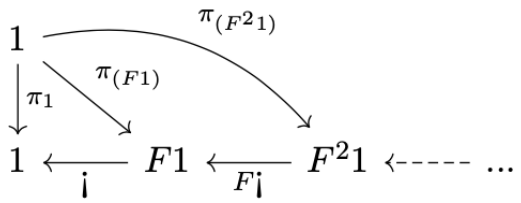


FIGURE 5. The limit of this chain is isomorphic to its starting set a

Things are more interesting when the chain is infinite, and there is no rightmost object—as is the case of our ω-chain. It turns out that the limit of such a chain is the terminal coalgebra for the functor F.



In this case, the interpretation where we look at preimages of the morphisms in the chain is very helpful. We can view a particular power of F acting on 1 as a set of trees generated by expanding the universal seeds embedded in the trees from the lower power of F. Those trees that had no seeds, only `LeafF` leaves, are just propagated without change. So the limit will definitely contain all these trees. But it will also contain infinite trees. These correspond to cones that select ever growing trees in which there are always some seeds that are replaced with double-seed-nodes rather than `LeafF` leaves.

Compare this with the initial algebra construction which only generated finite trees. The terminal coalgebra for the functor `TreeF` is larger than the initial algebra for the same functor.

We have also seen a functor whose initial algebra was an empty set

```
data StreamF a x = ConsF a x
```

This functor has a well-defined non-empty terminal coalgebra. The n-th power of `(StreamF a)` acting on `()` consists of lists of `a`s

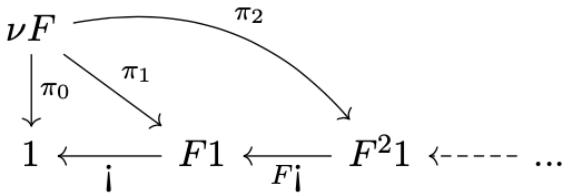
```
ConsF a1 (ConsF a2 (... (ConsF an ())...))
```

The lifting of `unit` acting on such a list replaces the final `(ConsF a ())` with `()` thus shortening the list by one item. Its "inverse" replaces the seed `()` with any value of type `a` (so it's a *multi-valued* inverse, since there are, in general, many values of `a`). The limit is isomorphic to an infinite stream of `a`. In Haskell it can be written as a recursive data structure

```
data Stream a = ConsF a (Stream a)
```

Anamorphism

The limit of a diagram is defined as a universal cone. In our case this would be the cone consisting of the object we'll call νF, with a set of projections π_n



such that any other cone factors through νF. We want to show that νF (if it exists) is a terminal coalgebra.

First, we have to show that νF is indeed a coalgebra, that is, there exists a morphism

$$k \colon \nu F \to F(\nu F)$$

We can apply F to the whole diagram. If F preserves ω-limits, then we get a universal cone with the apex $F(\nu F)$ and the ω-chain with $F1$ on the left. But our original object νF forms a cone over the same chain (ignoring the projection π_0). Therefore there must be a unique mapping k from it to $F(\nu F)$.

The coalgebra $(\nu F, k)$ is terminal if there is a unique morphism from any other coalgebra to it. Consider, for instance, a coalgebra $(A, \kappa \colon A \to FA)$. With this coalgebra, we can construct an ω-chain

$$A \xrightarrow{\kappa} FA \xrightarrow{F\kappa} F^2A \xrightarrow{F^2\kappa} F^3A \dashrightarrow \ldots$$

We can connect the two omega chains using the terminal morphism from A to 1 and all its liftings

$$\begin{array}{ccccccccc} A & \xrightarrow{\kappa} & FA & \xrightarrow{F\kappa} & F^2A & \xrightarrow{F^2\kappa} & F^3A & \dashrightarrow & \ldots \\ \downarrow{\scriptstyle !} & & \downarrow{\scriptstyle F!} & & \downarrow{\scriptstyle F^2!} & & \downarrow{\scriptstyle F^3!} & & \\ 1 & \xleftarrow[!]{} & F1 & \xleftarrow[F!]{} & F^21 & \xleftarrow[F^2!]{} & F^31 & \dashleftarrow & \ldots \end{array}$$

Notice that all squares in this diagram commute. The leftmost one commutes because 1 is the terminal object, therefore the mapping from A to it is unique, so the composite $! \circ F! \circ \kappa$ must be the same as $!$. A is therefore an apex of a cone over our original ω-chain. By universality, there must be a unique morphism from A to the limit of this ω-chain, νF. This morphism is in fact a coalgebra morphism and is called the *anamorphism*.

Adjunction

The constructions of initial algebras and terminal coalgebras can be compactly described using adjunctions.

There is an obvious forgetful functor U from the category of F-algebras C^F to C. This functor just picks the carrier and forgets the structure map. Under certain conditions, the left adjoint free functor Φ exists

$$C^F(\Phi x, a) \cong C(x, Ua)$$

This adjunction can be evaluated at the initial object (the empty set in Set).

$$C^F(\Phi 0, a) \cong C(0, Ua)$$

This shows that there is a unique algebra morphism—the catamorphism— from $\Phi 0$ to any algebra a. This is because the hom-set $C(0, Ua)$ is a singleton for every a. Therefore $\Phi 0$ is the initial algebra νF.

Conversely, there is a cofree functor Ψ

$$C_F(c, \Psi x) \cong C(Uc, x)$$

It can be evaluated at a terminal object

$$C_F(c, \Psi 1) \cong C(Uc, 1)$$

showing that there is a unique coalgebra morphism—the anamorphism—from any coalgebra c to $\Psi 1$. This shows that $\Psi 1$ is the terminal coalgebra νF.

Fixed point

Lambek's lemma works for both, initial algebras and terminal coalgebras. It states that their structure maps are isomorphisms, therefore their carriers are fixed points of the functor F

$$\mu F \cong F(\mu F)$$

$$\nu F \cong F(\nu F)$$

The difference is that μF is the least fixed point, and νF is the greatest fixed point of F. They are, in principle, different. And yet, in a programming language like Haskell, we only have one recursively defined data structure defining the fixed point

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

So which one is it?

We can define both the catamorphisms from-, and anamorphisms to-, the fixed point:

```
type Algebra f a = f a -> a

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

type Coalgebra f a = a -> f a

ana :: Functor f => Coalgebra f a -> a -> Fix f
ana coa = Fix . fmap (ana coa) . coa
```

so it seems like `Fix f` is both initial as the carrier of an algebra and terminal as the carrier of a coalgebra. But we know that there are elements of νF that are not in μF —namely infinite trees and infinite streams—so the two fixed points are not isomorphic and cannot be both described by the same `Fix f`.

However, they are not unrelated. Because of the Lambek's lemma, the initial algebra $(\mu F, j)$ gives rise to a coalgebra $(\mu F, j^{-1})$, and the terminal coalgebra $(\nu F, k)$ generates an algebra $(\nu F, k^{-1})$.

Because of universality, there must be a (unique) algebra morphism from the initial algebra $(\mu F, j)$ to $(\nu F, k^{-1})$, and a unique coalgebra morphism from $(\mu F, j^{-1})$ to the terminal coalgebra $(\nu F, k)$. It turns out that these two are given by the same morphism $f : \mu F \to \nu F$ between the carriers. This morphism satisfies the equation

$$k \circ f \circ j = F f$$

which makes it both an algebra and a coalgebra morphism

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{Ff} & F(\nu F) \\ \downarrow{\scriptstyle j} & & \uparrow{\scriptstyle k} \\ \mu F & \xrightarrow{f} & \nu F \end{array}$$

Furthermore, it can be shown that, in Set, f is injective: it embeds μF in νF. This corresponds to our observation that νF contains μF plus some infinite data structures.

The question is, can `Fix f` describe infinite data? The answer depends on the nature of the programming language: infinite data structures can only exist in a lazy language. Since Haskell is lazy, `Fix f` corresponds to the *greatest* fixed point. The least fixed point forms a subset of `Fix f` (in fact, one can define a metric in which it's a dense subset).

This is particularly obvious in the case of a functor that has no terminating leaves, like the stream functor.

```
data StreamF a x = StreamF a x
  deriving Functor
```

We've seen before that the initial algebra for `StreamF a` is empty, essentially because its action on `Void` is uninhabited. It does, however have a terminal coalgebra. And, in Haskell, the fixed point of the stream functor indeed generates infinite streams

```
type Stream a = Fix (StreamF a)
```

How do we know that? Because we can construct an infinite stream using an anamorphism. Notice that, unlike in the case of a catamorphism, the recursion in an anamorphism doesn't have to be well founded and, indeed, in the case of a stream, it never terminates. This is why this won't work in an eager language. But it works in Haskell. Here's a coalgebra whose carrier is `Int`

```
coaInt :: Coalgebra (StreamF Int) Int
coaInt n = StreamF n (n + 1)
```

It generates an infinite stream of natural numbers

```
ints = ana coaInt 0
```

Of course, in Haskell, the work is not done until we demand some values. Here's the function that extracts the head of the stream:

```
hd :: Stream a -> a
hd (Fix (StreamF x _)) = x
```

And here's one that advances the stream

```
tl :: Stream a -> Stream a
tl (Fix (StreamF _ s)) = s
```

This is all true in Set, but Haskell is not Set. I had a discussion with Edward Kmett and he pointed out that Haskell's fixed point data type can be considered the initial algebra as well. Suppose that you have an infinite data structure, like the stream we were just discussing. If you apply a catamorphism for an arbitrary algebra to it, it will most likely never terminate (try summing up an infinite stream of integers). In Haskell, however, this is interpreted as the catamorphism returning the bottom $\bot$, which is a perfectly legitimate value. And once you start including bottoms in your reasoning, all bets are off. In particular `Void` is no longer uninhabited—it contains $\bot$—and the colimit construction of the initial algebra is no longer valid. It's possible that some of these results can be generalized using domain theory and enriched categories, but I'm not aware of any such work.

Bibliography

1. Adamek, Introduction to coalgebra
2. Michael Barr, Terminal coalgebras for endofunctors on sets

April 9, 2020

Initial Algebra as Directed Colimit

Posted by Bartosz Milewski under Programming
[11] Comments

i
2 Votes

There is a bit of folklore about algebras in Haskell, which says that both the initial algebra and the terminal coalgebra for a given functor are defined by the same fixed point formula. This works for most common cases, but is not true in general. What is definitely true is that they are both fixed points–this result is called the Lambek's lemma–but there may be many fixed points. The initial algebra is the *least fixed point*, and the terminal coalgebra is the *greatest fixed point*.

In this series of blog posts I will explore the ways one can construct these (co-)algebras using category theory and illustrate it with Haskell examples.

In this first installment, I'll go over the construction of the initial algebra.

A functor

Let's start with a simple functor that generates binary trees. Normally, we would store some additional data in a tree (meaning, the functor would take another argument), either in nodes or in leaves, but here we're just interested in pure shapes.

```
data F a = Leaf
         | Node a a
  deriving Show
```

Categorically, this functor can be written as a *coproduct* (sum) of the terminal object 1 (singleton) and the *product* of a with itself, here written simply as a^2

$$Fa = 1 + a^2$$

The lifting of functions is given by this implementation of `fmap`

```
instance Functor F where
  fmap _ Leaf       = Leaf
  fmap f (Node x y) = Node (f x) (f y)
```

We can use this functor to build arbitrary level trees. Let's consider, for instance, terms of type `F Int`. We can either build a `Leaf`, or a `Node` with two numbers in it

```
x1, y1 :: F Int
x1 = Leaf
y1 = Node 1 2
```

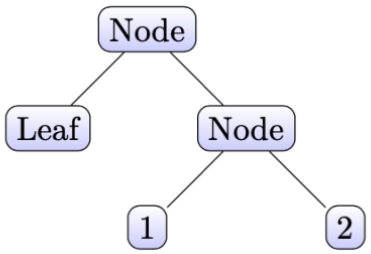
With those, we can build next-level values of the type F^2a or, in our case, `F (F Int)`

```
x2, y2 :: F (F Int)
x2 = Leaf
y2 = Node x1 y1
```

We can display `y2` directly using `show`

```
> Node Leaf (Node 1 2)
```

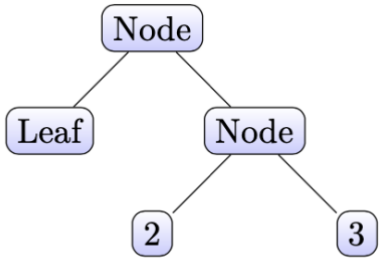
or draw the corresponding tree



Since F is an endofunctor, so is F^2. Lifting a function $f : a \to b$ to F^2 can be implemented by applying `fmap` twice. Here's the action of the function `(+1)` on our test values

```
fmap (fmap (+1)) x2
> Leaf
fmap (fmap (+1)) y2
> Node Leaf (Node 2 3)
```

or, graphically,



You can see that `Leaf`s at any level remain untouched; only the contents of bottom `Node`s in the tree are transformed.

The colimit construction

The carrier of the initial algebra can be constructed as a colimit of an infinite sequence. This sequence is constructed by applying powers of F to the initial object which we'll denote as 0. We'll first see how this works in our example.

The initial object in Haskell is defined as a type with no data constructor (we are ignoring the question of non-termination in Haskell).

```
data Void
  deriving Show
```

In Set, this is just an empty set.

The `Show` instance for `Void` requires the pragma

```
{-# language EmptyDataDeriving #-}
```

Even though there are no values of the type `Void`, we can still construct a value of the type `F Void`

```
z1 :: F Void
z1 = Leaf
```

This degenerate version of a tree can be drawn as

This illustrates a very important property of our F: Its action on an empty set does not produce an empty set. This is what allows us to generate a non-trivial sequence of powers of F starting with the empty set.

Not every functor has this property. For instance, the construction of the initial algebra for the functor

```
data StreamF a x = ConsF a x
```

will produce an uninhabited type (empty set). Notice that this is different from its terminal coalgebra, which is the infinite stream

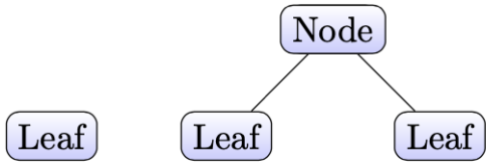
```
data Stream a = Cons a (Stream a)
```

This is an example of a functor whose initial algebra is not the same as the terminal coalgebra.

Double application of our `F` to `Void` produces, again, a `Leaf`, as well as a `Node` that contains two `Leaf`s.

```
z2, v2 :: F (F Void)
z2 = Leaf

v2 = Node z1 z1
> Node Leaf Leaf
```

Graphically,



In general, powers of F acting on `Void` generate trees which terminate with `Leaf`s, but there is no possibility of having terminal `Node`s). Higher and higher powers of F acting on `Void` will eventually produce any tree we can think of. But for any given power, there will exist even larger trees that are not generated by it.

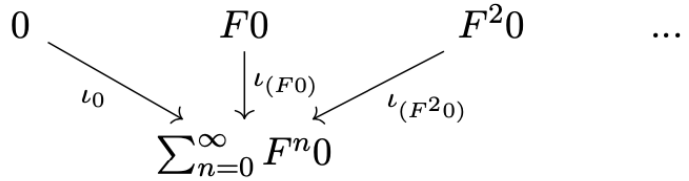
In order to get all the trees, we could try to take a sum (a coproduct) of infinitely many powers. Something like this

$$\sum_{n=0}^{\infty} F^n 0$$

The problem is that we'd also get a lot of duplication. For instance, we saw that `z1` was the same tree as `z2`. In general, a single `Leaf` is produced at all non-zero powers of F acting on `Void`. Similarly, all powers of F greater than one produce a single node with two leaves, and so on. Once a particular tree is produced at some power of F, all higher powers of F will also produce it.

We have to have a way of identifying multiply generated trees. This is why we need a *colimit* rather than a simple coproduct.

As a reminder, a coproduct is defined as a universal cocone. Here, the base of the cocone is the set of all powers of F acting on 0 (Haskell `Void`).



In a more general colimit, the objects in the base of the cocone may be connected by morphisms.

Coming from the initial object, there can be only one morphism. We'll call this morphism $!$ or, in Haskell, `absurd`

```
absurd :: Void -> a
absurd a = case a of {}
```

This definition requires another pragma

```
{-# language EmptyCase #-}
```

We can construct a morphism from $F0$ to $F^2 0$ as a lifting of $!$, $F!$. In Haskell, the lifting of `absurd` doesn't change the shape of trees. Here it is acting on a leaf

```
z1' :: F (F Void)
z1' = fmap absurd z1
> Leaf
```

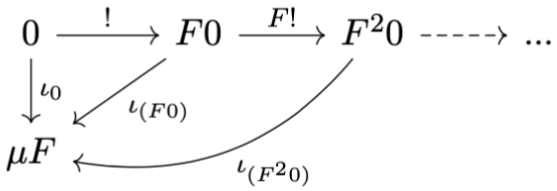
We can continue this process of lifting `absurd` to higher and higher powers of F

```
z2', v2' :: F (F (F Void))

z2' = fmap (fmap absurd) z2
> Leaf

v2' = fmap (fmap absurd) v2
> Node Leaf Leaf
```

We can construct an infinite chain (this kind of directed chain indexed by natural numbers is called an ω-chain)

$$0 \xrightarrow{!} F0 \xrightarrow{F!} F^2 0 \dashrightarrow \dots$$

We can use this chain as the base of our cocone. The colimit of this chain is defined as the universal cocone. We will call the apex of this cocone μF



In Set these constructions have simple interpretations. A coproduct is a discriminated union. A colimit is a discriminated union in which we identify all those injections that are connected by morphisms in the base of the cocone. For instance

$$\iota_0 = \iota_{(F0)} \circ !$$
$$\iota_{(F0)} = \iota_{(F^2 0)} \circ F!$$

and so on.

Here we use the lifted `absurd` (or `!` in the picture above) as the morphisms that connect the powers of F acting of `Void` (or 0 in the picture).

These are exactly the identifications that we were looking for. For instance, $F!$ maps the leaf generated by $F0$ to the leaf which is the element of $F^2 0$. Or, translating it to Haskell, `(fmap absurd)` maps the leaf generated by `F Void` to the leaf generated by `F (F Void)`, and so on.

All trees generated by the n'th power of F are injected into the $n+1$'st power of F by `absurd` lifted by the n th power of F.

The colimit is formed by equivalence classes with respect to these identifications. In particular, there is a class for a degenerate tree consisting of a single leaf whose representative can be taken from `F Void`, or from `F (F Void)`, or from `F (F (F Void))` and so on.

Initiality

The colimit μF is exactly the initial algebra for the functor F. This follows from the universal property of the colimit. First we will show that for any algebra $(A, \alpha \colon FA \to A)$ there is a unique morphism from μF to A. Indeed, we can build a cocone with A at its apex and the injections given by

$!$

$\alpha \circ F!$

$\alpha \circ F\alpha \circ F^2!$

and so on…

$$\begin{array}{ccccccc} 0 & \xrightarrow{!} & F0 & \xrightarrow{F!} & F^2 0 & \dashrightarrow & \dots \\ \downarrow{\scriptstyle !} & & \downarrow{\scriptstyle F!} & & \downarrow{\scriptstyle F^2!} & & \\ A & \xleftarrow[\alpha]{} & FA & \xleftarrow[F\alpha]{} & F^2 A & \dashleftarrow & \dots \end{array}$$

Since the colimit μF is defined by the universal cocone, there is a unique morphism from it to A. It can be shown that this morphism is in fact an algebra morphism. This morphism is called a *catamorphism*.

Fixed Point

Lambek's lemma states that the initial algebra is a fixed point of the functor that defines it

$F(\mu F) \cong \mu F$

This can also be seen directly, by applying the functor to every object and morphism in the ω-chain that defines the colimit. We get a new chain that starts at $F0$

$$F0 \xrightarrow{F!} F^2 0 \xrightarrow{F^2!} F^3 0 \dashrightarrow \dots$$

But the colimit of this chain is the same as the colimit μF of the original chain. This is becuase we can always add back the initial object to the chain, and define its injection ι_0 as the composite

$\iota_0 = \iota_{(F0)} \circ !$

On the other hand, if we apply F to the whole universal cocone, we'll get a new cocone with the apex $F(\mu F)$. In principle, this cocone doesn't have to be universal, so we cannot be sure that $F(\mu F)$ is a colimit. If it is, we say that F *preserves* the particular type of colimit—here, the ω-colimit.

Remember: the image of a cocone under a functor is always a cocone (this follows from functoriality). Preservation of colimits is an additional requirement that the image of a *universal* cocone be *universal*.

The result is that, if F preserves ω-colimits, then the initial algebra μF is a fixed point of F

$$F(\mu F) \cong \mu F$$

because both sides can be obtained as a colimit of the same ω-chain.

Bibliography

1. Adamek, Milius, Moss, Initial Algebras, Terminal Coalgebras, and the Theory of Fixed Points of Functors

February 24, 2020

Math is your insurance policy

Posted by Bartosz Milewski under Programming
[15] Comments

i
36 Votes

We live in interesting times. For instance, we are witnessing several extinction events all at once. One of them is the massive extinction of species. The other is the extinction of jobs. Both are caused by advances in technology. As programmers, we might consider ourselves immune to the latter–after all, somebody will have to program these self-driving trucks that eliminate the need for drivers, or the diagnostic tools that eliminate the need for doctors. Eventually, though, even programming jobs will be automated. I can imagine the last programmer putting finishing touches on the program that will make his or her job redundant.

But before we get there, let's consider which programming tasks are the first to go, and which have the biggest chance to persist for the longest time. Experience tells us that it's the boring menial jobs that get automated first. So any time you get bored with your work, take note: you are probably doing something that a computer could do better.

One such task is the implementation of user interfaces. All this code that's behind various buttons, input fields, sliders, etc., is pretty much standard. Granted, you have to put a lot of effort to make the code portable to a myriad of platforms: various desktops, web browsers, phones, watches, fridges, etc. But that's exactly the kind of expertise that is easily codified. If you find yourself doing copy and paste programming, watch out: your buddy computer can do it too. The work on generating UI has already started, see for instance, pix2code.

The *design* of user interfaces, as opposed to their implementation, will be more resistant to automation. Not only because it involves creativity, but also because it deals with human issues. Good design must serve the human in front of it. I'm not saying that modeling a human user is impossible, but it's definitely harder. Of course, in many standard tasks, a drastically simplified human model will work just fine.

So I'm sorry to say that, but those programmers who specialize in HTML and JavaScript will have to retrain themselves.

The next job on the chopping block, in my opinion, is that of a human optimizer. In fact the only reason it hasn't been eliminated yet is economical. It's still cheaper to hire people to optimize code than it is to invest in the necessary infrastructure. You might think that programmers are expensive–the salaries of programmers are quite respectable in comparison to other industries. But if this were true, a lot more effort would go into improving programmers' productivity, in particular in creating better tools. This is not happening. But as demand for software is growing, and the AI is getting cheaper, at some point the economic balance will change. It will be advantageous to use AI to optimize code.

I'm sorry to say that, but C and C++ programmers will have to go. These are the languages whose only raison d'être is to squeeze maximum performance from hardware. We'll probably always be interested in performance, but there are other ways of improving it. We are familiar with optimizing compilers that virtually eliminated the job of an assembly language programmer. They use optimizers that are based on algorithmic principles–that is methods which are understandable to humans. But there is a whole new generation of AI waiting in the aisles, which can be trained to optimize code written in higher level languages. Imagine a system, which would take this definition of quicksort written in Haskell:

```
qsort [] = []
qsort (p:xs) = qsort lesser ++ [p] ++ qsort greater
    where (lesser, greater) = partition (< p) xs
```

and produce code that would run as fast as its hand-coded C counterpart. Even if you don't know Haskell, I can explain this code to you in just a few sentences. The first line says that sorting an empty list produces an empty list. The second line defines the action of quicksort on a list that consists of a head `p`–that will be our pivot–and the tail `xs`. The result is the concatenation (the symbol `++`) of three lists. The first one is the result of (recursively) sorting the list `lesser`, the second is the singleton list containing the pivot, and the third is the result of sorting the list `greater`. Finally, the pair of lists `(lesser, greater)` is produced by partitioning `xs` using the predicate `(< p)`, which reads "less than p." You can't get any simpler than that.

Of course the transformation required for optimizing this algorithm is highly nontrivial. Depending on the rest of the program, the AI might decide to change the representation of data from a list to a vector, replace copying by destructive swapping, put some effort in selecting a better pivot, use a different algorithm for sorting very short lists, and so on. This is what a human optimizer would do. But how much harder is this task than, say, playing a game of go against a grandmaster?

I am immensely impressed with the progress companies like Google or IBM made in playing go, chess, and Jeopardy, but I keep asking myself, why don't they invest all this effort in programming technology? I can't help but see parallels with Ancient Greece. The Ancient Greeks made tremendous breakthroughs in philosophy and mathematics–just think about Plato, Socrates, Euclid, or Pythagoras–but they had no technology to speak of. Hero of Alexandria invented a steam engine, but it was never put to work. It was only used as a parlor trick. There are many explanations of this phenomenon, but one that strikes close to home is that the Greeks didn't need technology because they had access to cheap labor through slavery. I'm not implying that programmers are treated like slaves–far from it–but they seem to be considered cheap labor. In fact it's so cheap to produce software that most of it is given away for free, or for the price of users' attention in ad-supported software. A lot of software is just bait that's supposed to entice the user to buy something more valuable, like beer.

It's gradually becoming clear that programming jobs are diverging. This is not yet reflected in salaries, but as the job market matures, some programming jobs will be eliminated, others will increase in demand. The one area where humans are still indispensable is in specifying what has to be done. The AI will eventually be able to implement any reasonable program, as long as it gets a precise enough specification. So the programmers of the future will stop telling the computer how to perform a given task; rather they will specify what to do. In other words, declarative programming will overtake imperative programming. But I don't think that explaining to the AI what it's supposed to do will be easy. The AI will continue to be rather dumb, at least in the foreseeable future. It's been noted that software that can beat the best go players in the world would be at a complete loss trying to prepare a dinner or clean the dishes. It's able to play go because it's reasonably easy to codify the task of playing go– the legal moves and the goal of the game. Humans are extremely bad at expressing their wishes, as illustrated by the following story:

> *A poor starving peasant couple are granted three wishes and the woman, just taking the first thing that comes to her mind, wishes for one sausage, which she receives immediately. Her husband, pointing out that she could have wished for immense wealth or food to last them a lifetime, becomes angry with her for making such a stupid wish and, not thinking, wishes the sausage were stuck on her nose. Sure enough, the sausage is stuck in the middle of her face, and then they have to use the third wish to make it go away, upon which it disappears completely.*

As long as the dumb AI is unable to guess our wishes, there will be a need to specify them using a precise language. We already have such language, it's called math. The advantage of math is that it was invented for humans, not for machines. It solves the basic problem of formalizing our thought process, so it can be reliably transmitted and verified. The definition of quicksort in Haskell is very mathematical. It can be easily verified using induction, because it's recursive, and it operates on a recursive data structure: a list. The first line of code establishes the base case: an empty list is trivially sorted. Then we perform the induction step. We assume that we know how to sort all proper sublists of our list. We create two such sublists by partitioning the tail around the pivot. We sort the sublists, and then construct the final sorted list by inserting the pivot between them. As mathematical proofs go, this one is not particularly hard. In fact, in a typical mathematical text, it would be considered so trivial as to be left as an exercise for the reader.

Still, this kind of mathematical thinking seems to be alien to most people, including a lot of programmers. So why am I proposing it as the "programming language" of the future? Math is hard, but let's consider the alternatives. Every programming language is a compromise between the human and the computer. There are languages that are "close to the metal," like assembly or C, and there are languages that try to imitate natural language, like Cobol or SQL. But even in low level languages we try to use meaningful names for variables and functions in an attempt to make code more readable. In fact, there are programs that purposefully obfuscate source code by removing the formatting and replacing names with gibberish. The result is unreadable to most humans, but makes no difference to computers. Mathematical language doesn't have to be machine readable. It's a language that was created by the people, for the people. The reason why we find mathematical texts harder to read than, say, C++ code is because mathematicians work at a much higher abstraction level. If we tried to express the same ideas in C++, we would very quickly get completely lost.

Let me give you a small example. In mathematics, a monad is defined as a monoid in the category of endofunctors. That's a very succinct definition. In order to understand it, you have to internalize a whole tower of abstractions, one built on top of another. When we implement monads in Haskell, we don't use that definition. We pick a particular very simple category and implement only one aspect of the definition (we don't implement monadic laws). In C++, we don't even do that. If there are any monads in C++, they are implemented ad hoc, and not as a general concept (an example is the future monad which, to this day, is incomplete).

There is also some deeper math in the quicksort example. It's a recursive function and recursion is related to algebras and fixed points. A more elaborate version of quicksort decomposes it into its more fundamental components. The recursion is captured in a combination of unfolding and folding that is called a hylomorphism. The unfolding is described by a coalgebra, while folding is driven by an algebra.

```
data TreeF a r = Leaf | Node a r r
  deriving Functor

split :: Ord a => Coalgebra (TreeF a) [a]
split [] = Leaf
split (a: as) = Node a l r
  where (l, r) = partition (< a) as

join :: Algebra (TreeF a) [a]
join Leaf = []
join (Node a l r) = l ++ [a] ++ r

qsort :: Ord a => [a] -> [a]
qsort = hylo join split
```

You might think that this representation is an overkill. You may even use it in a conversation to impress your friends: "Quicksort is just a hylomorphism, what is the problem?" So how is it better than the original three-liner?

```
qsort [] = []
qsort (p:xs) = qsort lesser ++ [p] ++ qsort greater
    where (lesser, greater) = partition (< p) xs
```

The main difference is that the flow of control in this new implementation is driven by a data structure generated by the functor `TreeF`. This functor describes a binary tree whose every node has a value of type `a` and two children. We use those children in the unfolding process to store lists of elements, lesser ones on the left, greater (or equal) on the right. Then, in the folding process, these children are replenished again–this time with sorted lists. This may seem like an insignificant change, but it uses a different processing ability of our brains. The recursive function tells us a linear, one-dimensional, story. It appeals to our story-telling ability. The functor-driven approach appeals to our visual cortex. There is an up and down, and left and right in the tree. Not only that, we can think of the algorithm in terms of movement, or animation. We are first "growing" the tree from the seed and then "traversing" it to gather the fruit from the branches. These are some powerful metaphors.

If this kind of visualization works for us, it might as well work for the AI that will try to optimize our programs. It may also be able to access a knowledge base of similar algorithms based on recursion schemes and category theory.

I'm often asked by programmers: How is learning category theory going to help me in my everyday programming? The implication being that it's not worth learning math if it can't be immediately applied to your current job. This makes sense if you are trying to locally optimize your life. You are close to the local minimum of your utility function and you want to get even closer to it. But the utility function is not constant–it evolves in time. Local minima disappear. Category theory is the insurance policy against the drying out of your current watering hole.

November 6, 2019

Fixed Points and Diagonal Arguments

Posted by Bartosz Milewski under Programming
[5] Comments

i
11 Votes

What does Gödel's incompletness theorem, Russell's paradox, Turing's halting problem, and Cantor's diagonal argument have to do with the fact that negation has no fixed point? The surprising answer is that they are all related through
Lawvere's fixed point theorem.

Before we dive deep into category theory, let's unwrap this statement from the point of view of a (Haskell) programmer. Let's start with some basics. Negation is a function that flips between `True` and `False`:

```
not :: Bool -> Bool
not True  = False
not False = True
```

A fixed point is a value that doesn't change under the action of a function. Obviously, negation has no fixed point. There are other functions with the same signature that have fixed points. For instance, the constant function:

```
true True  = True
true False = True
```

has `True` as a fixed point.

All the theorems I listed in the preamble (and a few more) are based on a simple but extremely powerful proof technique invented by Georg Cantor called the diagonal argument. I'll illustrate this technique first by showing that the set of binary sequences is not countable.

Cantor's job interview question

A binary sequence is an infinite stream of zeros and ones, which we can represent as Booleans. Here's the definition a sequence (it's just like a list, but without the nil constructor), together with two helper functions:

```
data Seq a = Cons a (Seq a)
  deriving Functor

head :: Seq a -> a
head (Cons a as) = a

tail :: Seq a -> Seq a
tail (Cons a as) = as
```

And here's the definition of a binary sequence:

```
type BinSeq = Seq Bool
```

If such sequences were countable, it would mean that you could organize them all into one big (infinite) list. In other words we could implement a sequence generator that generates every possible binary sequence:

```
allBinSeq :: Seq BinSeq
```

Suppose that you gave this problem as a job interview question, and the job candidate came up with an implementation. How would you test it? I'm assuming that you have at your disposal a procedure that can (presumably in infinite time) search and compare infinite sequences. You could throw at it some obvious sequences, like all `True`, all `False`, alternating `True` and `False`, and a few others that came to your mind.

What Cantor did is to use the candidate's own contraption to produce a counter-example. First, he extracted the diagonal from the sequence of sequences. This is the code he wrote:

```
diag :: Seq (Seq a) -> Seq a
diag seqs = Cons (head (head seqs)) (diag (trim seqs))

trim :: Seq (Seq a) -> Seq (Seq a)
trim seqs = fmap tail (tail seqs)
```

You can visualize the sequence of sequences as a two-dimensional table that extends to infinity towards the right and towards the bottom.

```
T F F T ...
T F F T ...
F F T T ...
F F F T ...
...
```

Its diagonal is the sequence that starts with the fist element of the first sequence, followed by the second element of the second sequence, third element of the third sequence, and so on. In our case, it would be a sequence `T F T T ...`.

It's possible that the sequence, `diag allBinSeq` has already been listed in `allBinSeq`. But Cantor came up with a devilish trick: he negated the whole diagonal sequence:

```
tricky = fmap not (diag allBinSeq)
```

and ran his test on the candidate's solution. In our case, we would get `F T F F ...` The tricky sequence was obviously not equal to the first sequence because it differed from it in the first position. It was not the second, because it differed (at least) in the second position. Not the third either, because it was different in the third position. You get the gist…

Power sets are big

In reality, Cantor did not screen job applicants and he didn't even program in Haskell. He used his argument to prove that real numbers cannot be enumerated.

But first let's see how one could use Cantor's diagonal argument to show that the set of subsets of natural numbers is not enumerable. Any subset of natural numbers can be represented by a sequence of Booleans, where `True` means a given number is in the subset, and `False` that it isn't. Alternatively, you can think of such a sequence as a function:

```
type Chi = Nat -> Bool
```

called a *characteristic* function. In fact characteristic functions can be used to define subsets of any set:

```
type Chi a = a -> Bool
```

In particular, we could have defined binary sequences as characteristic functions on naturals:

```
type BinSeq = Chi Nat
```

As programmers we often use this kind of equivalence between functions and data, especially in lazy languages.

The set of all subsets of a given set is called a *power set*. We have already shown that the power set of natural numbers is not enumerable, that is, there is no function:

```
enumerate :: Nat -> Chi Nat
```

that would cover all characteristic functions. A function that covers its codomain is called surjective. So there is no surjection from natural numbers to all sequences of natural numbers.

In fact Cantor was able to prove a more general theorem: for any set, the power set is always larger than the original set. Let's reformulate this. There is no surjection from the set A to the set of characteristic functions $A \to 2$ (where 2 stands for the two-element set of Booleans).

To prove this, let's be contrarian and assume that there is a surjection:

```
enumP :: A -> Chi A
```

Since we are going to use the diagonal argument, it makes sense to uncurry this function, so it looks more like a table:

```
g :: (A, A) -> Bool
g = uncurry enumP
```

Diagonal entries in the table are indexed using the following function:

```
delta :: a -> (a, a)
delta x = (x, x)
```

We can now define our custom characteristic function by picking diagonal elements and negating them, as we did in the case of natural numbers:

```
tricky :: Chi A
tricky = not . g . delta
```

If `enumP` is indeed a surjection, then it must produce our function `tricky` for some value of `x :: A`. In other words, there exists an `x` such that `tricky` is equal to `enumP x`.
This is an equality of functions, so let's evaluate both functions at `x` (which will evaluate `g` at the diagonal).

```
tricky x == (enumP x) x
```

The left hand side is equal to:

```
tricky x = {- definition of tricky -}
not (g (delta x)) = {- definition of g -}
not (uncurry enumP (delta x)) = {- uncurry and delta -}
not ((enumP x) x)
```

So our assumption that there exists a surjection `A -> Chi A` led to a contradition!

```
not ((enumP x) x) == (enumP x) x
```

Real numbers are uncountable

You can kind of see how the diagonal argument could be used to show that real numbers are uncountable. Let's just concentrate on reals that are greater than zero but less than one. Those numbers can be represented as sequences of decimal digits (the digits following the decimal point). If these reals were countable, we could list them one after another, just like we attempted to list all streams of Booleans. We would get some kind of a table infinitely extending to the right and towards the bottom. There is one small complication though. Some numbers have two equivalent decimal representations. For instance 0.48 is the same as $0.47999...$, with infinitely many nines. So lets remove all rows from our table that end in an infinity of nines. We get something like this:

```
3 5 0 5 ...
9 9 0 8 ...
4 0 2 3 ...
0 0 9 9 ...
...
```

We can now apply our diagonal argument by picking one digit from every row. These digits form our diagonal number. In our example, it would be `3 9 2 9`.

In the previous proof, we negated each element of the sequence to get a new sequence. Here we have to come up with some mapping that changes each digit to something else. For instance, we could add one to it, modulo nine. Except that, again, we don't want to produce nines, because we could end up with a number that ends in an infinity of nines. But something like this will work just fine:

```
h n = if n == 8
      then 3
      else (n + 1) `mod` 9
```

The important part is that our function `h` replaces every digit with a different digit. In other words, `h` *doesn't have a fixed point*.

Lawvere's fixed point theorem

And this is what Lawvere realized: The diagonal argument establishes the relationship between the existence of a surjection on the one hand, and the existence of a no-fix-point mapping on the other hand. So far it's been easy to find a no-fix-point functions. But let's reverse the argument: If there is a surjection $A \to (A \to Y)$ then every function $Y \to Y$ must have a fixed point. That's because, if we could find a no-fixed-point function, we could use the diagonal argument to show that there is no surjection.

But wait a moment. Except for the trivial case of a function on a one-element set, it's always possible to find a function that has no fixed point. Just return something else than the argument you're called with. This is definitely true in Set, but when you go to other categories, you can't just construct morphisms willy-nilly. Even in categories where morphisms are functions, you might have constraints, such as continuity or smoothness. For instance, every continuous function from a real segment to itself has a fixed point (Brouwer's theorem).

As usual, translating from the language of sets and functions to the language of category theory takes some work. The tricky part is to generalize the definition of a fixed point and surjection.

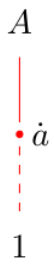
Points and string diagrams

First, to define a fixed point, we need to be able to define a point. This is normally done by defining a morphism from the terminal object 1, such a morphism is called a *global element*. In Set, the terminal object is a singleton set, and a morphism from a singleton set just picks an element of a set.

Since things will soon get complicated, I'd like to introduce string diagrams to better visualise things in a cartesian category. In string diagrams lines correspond to objects and dots correspond to morphisms. You read such diagrams bottom up. So a morphism

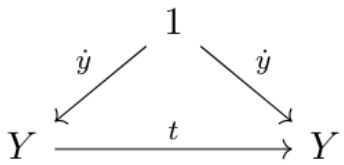
$$\dot{a} \colon 1 \to A$$

can be drawn as:

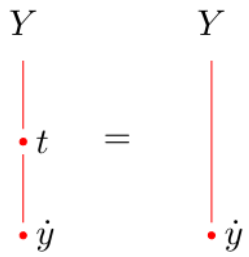


I will use dotted letters to denote "points" or morphisms originating in the unit. It is also customary to omit the unit from the picture. It turns out that everything works just fine with implicit units.

A fixed point of a morphism $t\colon Y \to Y$ is a global element $\dot{y}\colon 1 \to Y$ such that $t \circ \dot{y} = \dot{y}$. Here's the traditional diagam:

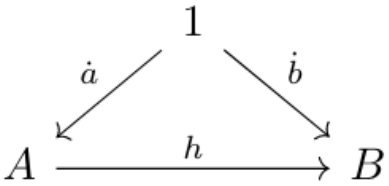


And here's the corresponding string diagram that encodes the commuting condition.

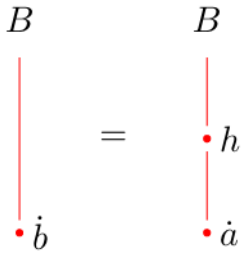


In string diagrams, morphisms are composed by stringing them along lines in the bottom up direction.

Surjections can be generalized in many ways. The one that works here is called *surjection on points*. A morphism $h\colon A \to B$ is surjective on points when for every point $\dot{b}$ of B (that is a global element $\dot{b}\colon 1 \to B$) there is a point $\dot{a}$ of A (the domain of h) that is mapped to $\dot{b}$. In other words $h \circ \dot{a} = \dot{b}$



Or string-diagrammatically, for every $\dot{b}$ there exists an $\dot{a}$ such that:



Currying and uncurrying

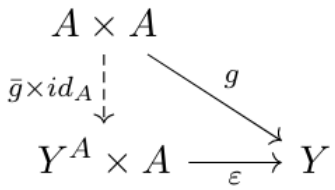
To formulate Lawvere's theorem, we'll replace B with the exponential object Y^A, that is an object that represents the set of morphisms from A to Y. Conceptually, those morphism will correspond to rows in our table (or characteristic functions, when Y is 2). The mapping:

$$\bar{g} \colon A \to Y^A$$

generates these rows. I will use barred symbols, like $\bar{g}$ for curried morphisms, that is morphisms to exponentials. The object A serves as the source of indexes for enumerating the rows of the table (just like the natural numbers in the previous example). The same object also provides indexing within each row.

This is best seen after uncurrying $\bar{g}$ (we assume that we are in a cartesian closed category). The uncurried morphism, $g \colon A \times A \to Y$ uses a product $A \times A$ to index simultaneously into rows and columns of the table, just like pairs of natural numbers we used in the previous example.

The currying relationship between these two is given by the universal construction:

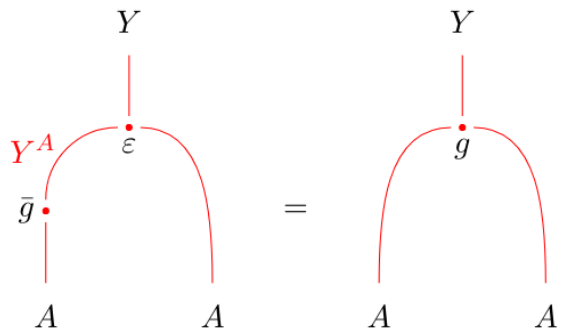


with the following commuting condition:

$$g = \varepsilon \circ (\bar{g} \times id_A)$$

Here, ε is the evaluation natural transformation (the couinit of the currying adjunction, or the dollar sign operator in Haskell).

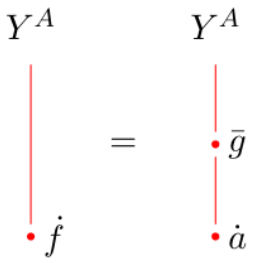
This commuting condition can be visualized as a string diagram. Notice that products of objects correspond to parallel lines going up. Morphisms that operate on products, like ε or g, are drawn as dots that merge such lines.



We are interested in mappings that are point-surjective. In this case, we would like to demand that for every point $\dot{f} \colon 1 \to Y^A$ there is a point $\dot{a} \colon 1 \to A$ such that:

$$\dot{f} = \bar{g} \circ \dot{a}$$

or, diagrammatically, for every $\dot{f}$ there exists an $\dot{a}$ such that:

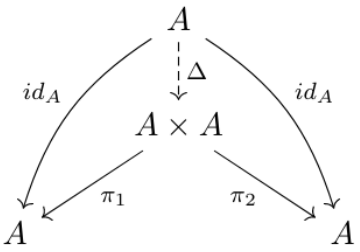


Conceptually, $\dot{f}$ is a point in Y^A, which represents some arbitrary function $A \to Y$. Surjectivity of $\bar{g}$ means that we can always find this function in our table by indexing into it with some $\dot{a}$.

This is a very general way of expressing what, in Haskell, would amount to: Every function `f :: A -> Y` can be obtained by partially applying our `g_bar :: X -> A -> Y` to some `x :: X`.

The diagonal

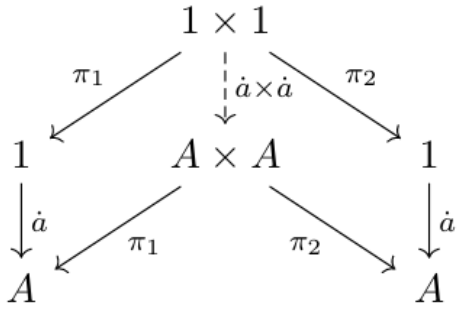
The way to index the diagonal of our table is to use the diagonal morphism $\Delta \colon A \to A \times A$. In a cartesian category, such a morphism always exists. It can be defined using the universal property of the product:

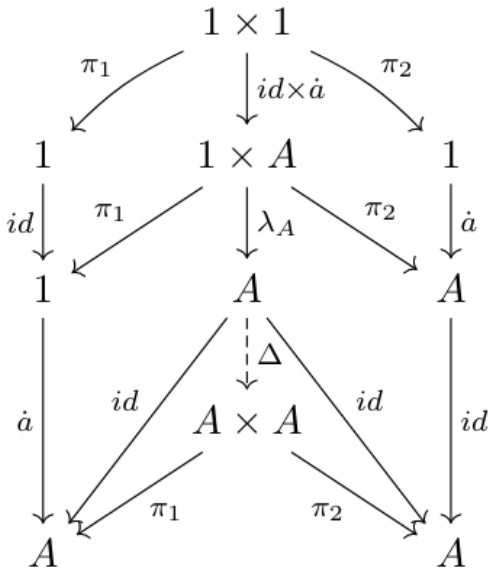


By combining this diagram with the diagram that defines the lifting of a pair of points $\dot{a}$ we arrive at a useful identity:

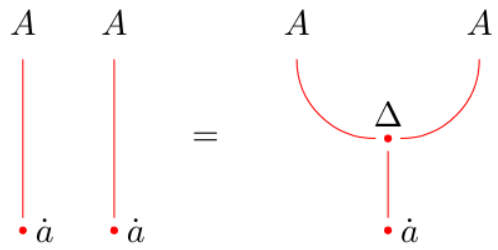
$$\dot{a} \times \dot{a} = \Delta_A \circ \lambda_A \circ (id_A \times \dot{a})$$

where λ_A is the left unitor, which asserts the isomorphism $1 \times A \to A$





Here's the string diagram representation of this identity:



In string diagrams we ignore unitors (as well as associators). Now you see why I like string diagrams. They make things much simpler.

Lawvere's fixed point theorem

Theorem. *(Lawvere) In a cartesian closed category, if there exists a point-surjective morphism* $\bar{g}\colon A \to Y^A$ *then every endomorphism of* Y *must have a fixed point.*

Note: Lawvere actually used a weaker definition of point surjectivity.

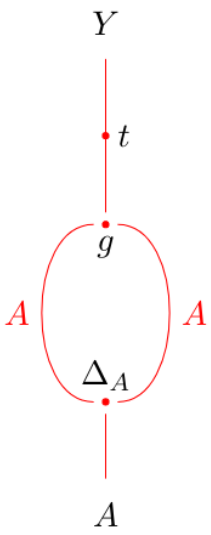
The proof is just a generalization of the diagonal argument.

Suppose that there is an endomorphims $t\colon Y \to Y$ that has no fixed point. This generalizes the negation of the original example. We'll create a special morphism by combining the diagonal entries of our table, and then "negating" them with t.

The table is described by (uncurried) g; and we access the diagonal through Δ_A. So the tricky morphism $A \to Y$ is just:

$$f = t \circ g \circ \Delta_A$$

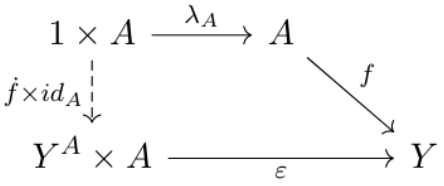
or, in diagramatic form:



$$f\colon A \xrightarrow{\Delta_A} A \times A \xrightarrow{g} Y \xrightarrow{t} Y$$

Since we were guaranteed that our table g is an exhaustive listing of all morphisms $A \to Y$, our new morphism f must be somewhere there. But in order to search the table, we have to first convert f to a point in the exponential object Y^A.

There is a one-to-one correspondence between points $\dot{f}\colon 1 \to Y^A$ and morphisms $f\colon A \to Y$ given by the universal property of the exponential (noting that $1 \times A$ is isomorphic to A through the left unitor, $\lambda_A\colon 1 \times A \to A$).



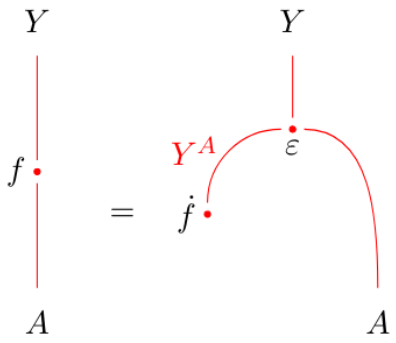
In other words, $\dot{f}$ is the curried form of $f \circ \lambda_A$, and we have the following commuting codition:

$$f \circ \lambda_A = \varepsilon \circ (\dot{f} \times id_A)$$

Since λ is an isomorphism, we can invert it, and get the following formula for f in terms of $\dot{f}$:

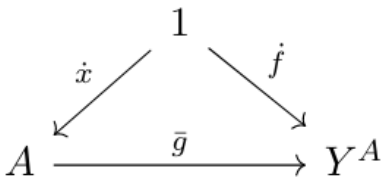
$$f = \varepsilon \circ (\dot{f} \times id_A) \circ \lambda_A^{-1}$$

In the corresponding string diagram we omit the unitor altogether.



Now we can use our assumption that $\bar{g}$ is point surjective to deduce that there must be a point $\dot{x} \colon 1 \to A$ that will produce $\dot{f}$, in other words:

$$\dot{f} = \bar{g} \circ \dot{x}$$



So $\dot{x}$ picks the row in which we find our tricky morphism. What we are interested in is "evaluating" this morphism at $\dot{x}$. That will be our paradoxical diagonal entry. By construction, it should be equal to the corresponding point of f, because this row is point-by-point equal to f; after all, we have just found it by searching for f! On the other hand, it should be different, because we've build f by "negating" diagonal entries of our table using t. Something has to give and, since we insist on surjectivity, we conclude that t is not doing its job of "negating." It must have a fixed point at $\dot{x}$.

Let's work out the details.

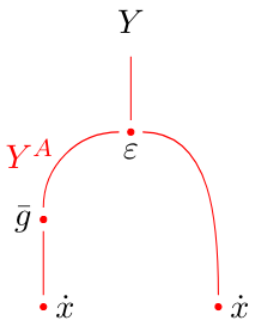
First, let's apply the function we've found in the table at row $\dot{x}$ to $\dot{x}$ itself. Except that what we've found is a point in Y^A. Fortunately we have figured out earlier how to get f from $\dot{f}$. We apply the result to $\dot{x}$:

$$f \circ \dot{x} = \varepsilon \circ (\dot{f} \times id_A) \circ \lambda_A^{-1} \circ \dot{x}$$

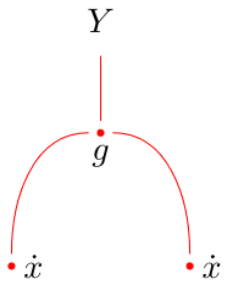
Plugging into it the entry $\dot{f}$ that we have found in the table, we get:

$$f \circ \dot{x} = \varepsilon \circ ((\bar{g} \circ \dot{x}) \times id_A) \circ \lambda_A^{-1} \circ \dot{x}$$

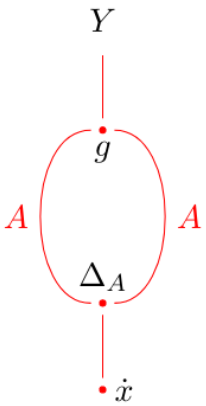
Here's the corresponding string diagram:



We can now uncurry $\bar{g}$

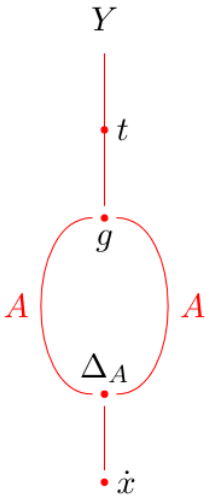


And replace a pair of $\dot{x}$ with a Δ:



Compare this with the defining equation for f, as applied to $\dot{x}$:

$$f \circ \dot{x} = t \circ g \circ \Delta_A \circ \dot{x}$$



In other words, the morphism $1 \to Y$:

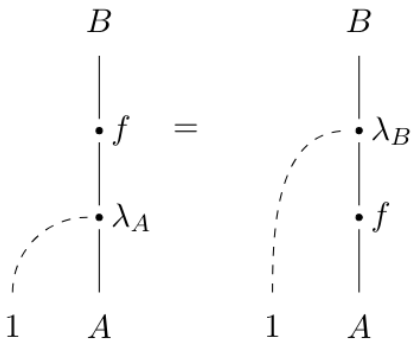
$$g \circ \Delta_A \circ \dot{x}$$

is a fixed point of t . This contradicts our assumption that t had no fixed point.

Conclusion

When I started writing this blog post I though it would be a quick and easy job. After all, the proof of Lawvere's theorem takes just a few lines both in the original paper and in all other sources I looked at. But then the details started piling up. The hardest part was convincing myself that it's okay to disregard all the unitors. It's not an easy thing for a programmer, because without unitors the formulas don't "type check." The left hand side may be a morphism from $A \times I$ and the right hand side may start at A . A compiler would reject such code. I tried to do due diligence as much as I could, but at some point I decided to just go with string diagrams. In the process I got into some interesting discussions and even posted a question on Math Overflow. Hopefully it will be answered by the time you read this post.

One of the things I wasn't sure about was if it was okay to slide unitors around, as in this diagram:



It turns out this is just naturality condition for λ, as John Baez was kind to remind me on Twitter (great place to do category theory!).

Acknowledgments

I'm grateful to Derek Elkins for reading the draft of this post.

Literature

1. F. William Lawvere, Diagonal arguments and cartesian closed categories
2. Noson S. Yanofsky, A Universal Approach to Self-Referential Paradoxes, Incompleteness and Fixed Points
3. Qiaochu Yuan, Cartesian closed categories and the Lawvere fixed point theorem

October 9, 2019

Fibrations, Cleavages, and Lenses

Posted by Bartosz Milewski under Programming
[2] Comments

i
11 Votes

In category theory, as in life, you spend half of your time trying to forget things, and half of the time trying to recover them. A morphism, the basic building block of every category, is like a defective isomorphism. It maps the initial state to the final state, but it provides no guarantees that you can recover the original. But it seems like this lossiness is what makes morphisms useful.

There are people who can memorize mathematical formulas perfectly but have no idea what they mean. And there are those who get just the gist of it, but can derive the rest when needed. Somehow *understanding* is related to lossy compression.

We can't recover lost information. Once it's gone, it's gone. All we can do is to try to figure out what the original might have been like. In fact, knowing how the information was lost, we might be able to generate all possible inputs that could have led to a given output. It's the closest we can get to inverting the uninvertible. This is the main idea behind fibrations.

Let me illustrate this concept with an example. Consider the function `isEven`:

```
isEven :: Integer -> Bool
isEven n = n `mod` 2 == 0
```

This function is definitely not invertible. If I only told you that the output was `True`, you couldn't tell me what the input was. You could, however, give me the set of all inputs that could have produced this output: it's the set of even numbers. We often call this set, which is the inverse image of `True`, a *fiber* over `True`. Similarly, the fiber over `False` is the set of odd numbers. In this case we only have only two fibers and they happen to be isomorphic.

Here's a more interesting example. Consider a set of all lists of integers and a function that returns the length of a list: a natural number:

```
length :: [Integer] -> Nat
```

This function is not invertible, but it defines fibers over natural numbers. The fiber over zero is a one-element set that contains only the empty list. The fiber over 1 is the set of lists of length one (which is isomorphic to the set of integers). The fiber over 2 is a set of 2-element lists, or pairs of integers, and so on. You may recognize these fibers as length-indexed lists, or vectors. You can find them, for instance, in the Haskell Vec library or as `Vect` in Idris. These are not your typical data types, though. They are examples of *dependent types*–types that depend on values (here, natural numbers).

Notice that the name "length-indexed lists" suggests a slightly different interpretation of these types. You may think of them as families of types parameterized by natural numbers. This would suggest a mapping from elements of a set (natural numbers) to types. These two views are equivalent, but in category theory we try to avoid, if possible, talking about sets (and set elements in particular). The interpretation of dependent types as fibrations is more general, so let's dig into fibrations.

As a generalization of functions like `isEven` or `length`, we'll consider a morphism $\pi : e \to b$, and call it a projection, since it projects each fiber down to one element. Our goal, though, is to define a fiber as the pre-image of an element in b. But what's an element? The closest we can get to defining an element in category theory is to consider a morphism x from the terminal object 1. Such morphism is called a *global element* and, in Set it really picks a single element from a (non-empty) set. Now we have two morphisms converging on b: π and x. Conceptually, a fiber is a subobject e_x of e, which means that there must be a morphism s that embeds e_x in e. Moreover, when this embedding is followed by the projection π, it must produce the same element as x. The best such object is given by a universal construction which, in this case, is a pullback.

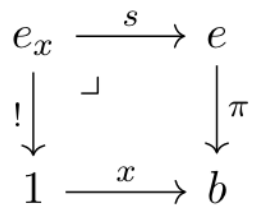


Fig. 1.

(The exclamation mark stands for the unique morphism to the terminal object.) Incidentally, this is why a pullback is sometimes called a *fiber product*.

If fibers over all elements x are isomorphic, the pair (e, π) is, quite fittingly, called a *fiber bundle*. The object b, from which the fibers sprout, is called the base. (Notice that lenght-indexed lists don't form a bundle.)

Anything you can do with functions, you can do with functors, only better. So we can have a category E, another category B, and a functor $p \colon E \to B$. Since a functor acts as a function on objects (modulo size issues), we can define a fiber as a set of objects in E that are mapped to a single object in B. The big question is, what do we do with morphisms? We have potentially lots of morphisms in E that go between any two fibers, and which get projected down to a single hom-set in B. If we want to invert p, we have to design a procedure for lifting morphisms from B to E.

Here's the idea: We would like each fiber to form a subcategory of E, and we'd like to pick morphisms between fibers in such a way that p^{-1} becomes a functor from B to Cat. In other words, we want p^{-1} to map objects of B to categories (the fibers), and morphisms of B to functors between those categories. If this is too much to ask (which it often is), we'll settle for p^{-1} to be a *pseudo-functor*, which is a functor that preserves unit and composition only up to isomorphism. In fact the original construction (attributed to Grothendieck) produces a *contravariant* pseudo-functor. In this post I'll describe the *covariant* version of this construction, which is called *opfibration*, and which is easier to explain.

The starting point of Grothendieck fibration is the recipe for lifting morphisms from the base category B to the total category E. There is a universal construction for doing that. The resulting morphisms are called *opcartesian*.

Let's start with a morphism $f \colon a \to b$ in the base category and pick an arbitrary object s (source) in the fiber over a (hence $a = ps$). This will be the source of our opcartesian morphism. We have a lot of choices for the target. Strictly speaking, the target should be one of the objects *over* b, and that's what we are aiming for. However, a universal construction should look at a much larger pool of candidates, some of them with targets in other fibers. This pool enlargement helps narrow down the final choice with greater accuracy. (Remember, universal constructions are unique only up to isomorphism.)

The opcartesian morphism over f, with the specified source s, is a morphism $g \colon s \to t$, such that $pg = f$.

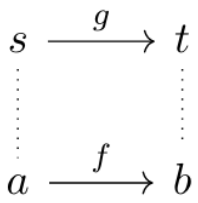


Fig. 2.

It must satisfy a universal property that I'm about to describe.

First, we pick an arbitrary object x and a morphism $h\colon s \to x$. This is supposed to be the competition for g. When projected down to B, it becomes $ph\colon a \to px$.

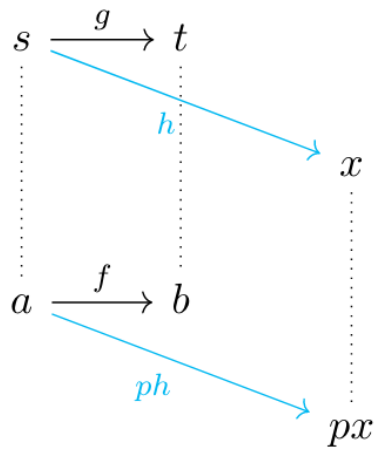


Fig. 3.

We are interested in the case when ph factorizes through f, that is, there is a morphism $u\colon b \to px$ such that $ph = u \circ f$. Whenever such factorization is possible in B, we demand that there be a unique lifting of it to E.

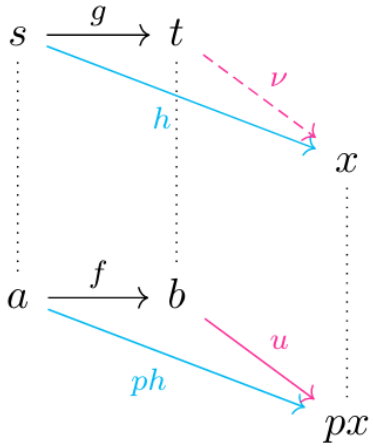


Fig. 4.

In other words, there exists a unique $\nu\colon t \to x$ such that $h = \nu \circ g$ and $p\nu = u$.

If you find this definition a little confusing, you're in good company. So let's try a slightly different imagery that has more to do with the original ideas from algebraic topology. Think of objects as shapes. A morphism $f: a \to b$ is a proof that b is a proper subset of a, or that a *contains* b. A functor between two categories of shapes must map shapes to shapes in a way that preserves inclusion. It may map many shapes to one, so imagine that the shapes in the category E are three-dimensional, and their projections using the functor p are their flat shadows. Functoriality means that, if s contains t, then its shadow $a = ps$ contains $b = pt$. Next, we introduce a new object x that is contained inside s, and the proof of that is $h: s \to x$. It follows from functoriality that a contains the shadow of x.

Now suppose that this shadow falls inside the smaller b (with the proof $u: b \to px$).

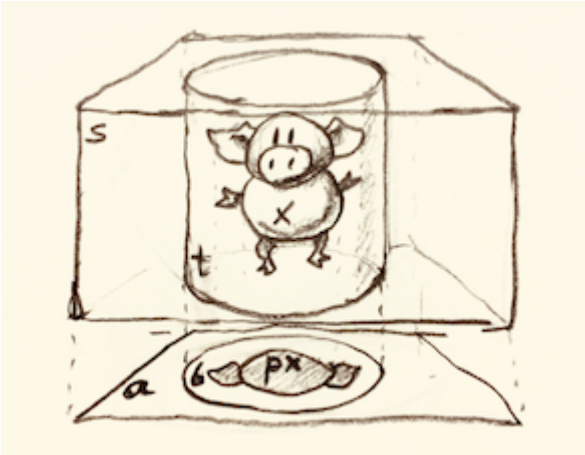


Fig. 5.

Normally, this would not imply that x is inside t. It's possible that (parts of) x are sticking out below or above t. Our universal condition demands that this cannot happen. There can be no room above or below t — it's a cylinder carved into s. Universality guarantees that we get the absolutely optimal shape.

Now that we know what an opcartesian morphism is, we might ask the question, does it always exist? Given an arbitrary morphism $f: a \to b$ in B and an object s over a, can we always find an opcartesian morphism $g: s \to t$ such that t is over b? If we can, then we call the pair $(E, p: E \to B)$ an *opfibration*.

Here's an interesting observation. You might wonder whether the definition of an opcartesian morphism isn't overly complicated. Wouldn't it be enough to restrict the pool of possible candidates to those morphisms whose target, the x in our picture, was over b, the target of f? This was, in fact, the original idea in the Grothendieck construction. The problem was that, with such definition, there was no guarantee that a composition of two opcartesian morphisms would be again opcartesian. The current definition makes that automatic.

Given an opfibration, we now face the opposite problem: there may be too many opcartesian morphisms. Remember, we wanted to (a) make fibers into subcategories of E and (b) use opcartesian morphisms to define functors between them. The first part is relatively easy: a fiber E_a has, as objects, those objects of E whose projection is a. We select as morphisms in E_a those morphisms that project down to identity, id_a (notice that we ignore other endomorphism $a \to a$). These are called *vertical* morphisms. But to define functors between fibers we need to map each object of one fiber to exactly one object in the other fiber (and the same for vertical morphisms). Think of this as *transporting* objects between fibers. In a fibered category, we could use opcartesian morphisms for transport. Any time two

objects are connected in the base by a morphism, we have a bunch of opcartesian morphisms over it starting from every single object in the source fiber. We could use them to try to define a functor between fibers.

But in general we have more than one opcartesian morphism between a source object in one fiber and candidate target objects in the other fiber. But we can design a procedure to pick one (if you're into set theory, you'll notice that we have to use the Axiom of Choice). Such choice is called an *opcleavage*, and the resulting construction is called *cloven opfibration*.

Formally, an opcleavage is described by a function $\kappa(f, s)$

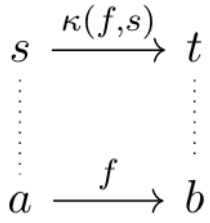


Fig. 6.

It takes a morphism $f : a \to b$ and an object s (such that $ps = a$), and produces an object t (such that $pt = b$), which is the target of some opcartesian morphism $s \to t$. This is exactly the morphism selected by opcleavage.

The universal construction of opcartesian morphisms can then be used to define the mapping of vertical morphisms thus completing the definition of a functor between fibers.

A geometric intuition is that an opcleavage provides a way of transporting objects in the horizontal direction. Vertical morphisms transport objects vertically, and the functors defined by the opcleavage transport them horizontally, in such a way that their shadows follow the arrows in the base. The origin of this intuition goes back to differential geometry, where one is able to define *continuous* paths in the base manifold and use them to transport objects, such as vectors, between fibers. Category theory lets us abstract away continuity (and differentiability) from this picture. You might also see transport used in homotopy type theory, with paths standing for equality proofs.

Now, remember what I said about the composition of opcartesian morphisms resulting in an opcartesian morphism? Unfortunately, once we start picking individual morphisms to construct an opcleavage, this compositionality might be lost. The composition of any two morphisms from the selected pool is still opcartesian, but it's not necessarily part of the opcleveage. This is why we might have to relax compositionality and embrace pseudofunctors.

But sometimes an opcleavage preserves compositionality. We call this situation *split opfibration*. It must satisfy these two conditions:

$$\kappa(id_a, s) = id_s$$

$$\kappa(f', s') \circ \kappa(f, s) = \kappa(f' \circ f, s)$$

for any $f : a \to b$ and $f' : b \to c$.

A split opfibration defines a functor $B \to Cat$, which maps objects from the base category to fibers seen as categories; and morphisms from the base category to functors between those fibers. So defined functor may be interpreted as an attempt at inverting the original projection $p\colon E \to B$.

If the splitting conditions are satisfied only up to isomorphism, we get a pseudofunctor $B \to Cat$. This makes things more complicated but also more interesting. It means that horizontal transport depends on the path. In particular, transport along a closed path–a chain of morphisms in the base that compose to identity–may produce an object that's different from (albeit isomorphic to) the starting object. In differential geometry we would say that the space has non-zero curvature.

Interestingly, this procedure of generating split opfibrations has its inverse. Given a functor $B \to Cat$ it's possible to reconstruct the total category E and a projection $p\colon E \to B$. This is called the Grothendieck construction.

Since our new slogan is "lenses are everywhere," it should come as no big surprise that a split opfibration may be seen as a type of a lens. The projection corresponds to *view* or *get*. It extracts a, the focus of the lens, out of s. The opcleavage part of opfibration, $\kappa(f, s)$ corresponds to *put* or, more precisely to *over*. It takes a morphism that modifies the focus from a to b and it takes the object s, and produces the new object t. In programming, *get* and *put* are just functions between sets, here they are object mappings of two functors, but the similarity is hard to ignore.

Acknowledgment

I'm grateful to Bryce Clarke for reading the draft and helpful comments.

Papers to read

1. Johnson, Rosebrugh, and Wood, Lenses, fibrations and universal translations
2. Johnson and Rosebrugh, Delta lenses and opfibrations

September 20, 2019

The Power of Adjunctions

Posted by Bartosz Milewski under Category Theory, Monads, Programming
[15] Comments

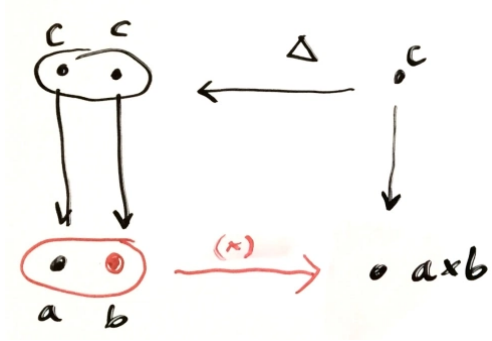
i
6 Votes

In my previous blog post, Programming with Universal Constructions, I mentioned in passing that one-to-one mappings between sets of morphisms are often a manifestation of adjunctions between functors. In fact, an adjunction just extends a universal construction over the whole category (or two categories, in general). It combines the mapping-in with the mapping-out conditions. One functor (traditionally called the left adjoint) prepares the input for mapping out, and the other (the right adjoint) prepares the output for mapping in. The trick is to find a pair of functors that complement each other: there are as many mapping-outs from one functor as there are mapping-ins to the other functor.

To gain some insight, let's dig deeper into the examples from my previous post.

The defining property of a *product* was the universal mapping-in condition. For every object c equipped with a pair of morphisms going to, respectively, a and b, there was a unique morphism h mapping c to the product $a \times b$. The commuting condition ensured that the correspondence went both ways, that is, given an h, the two morphisms were uniquely determined.

A pair of morphisms can be seen as a single morphism in a product category $C \times C$. So, really, we have an isomorphism between hom-sets in two categories, one in $C \times C$ and the other in C. We can also define two functors going between these categories. An arbitrary object c in C is mapped by the diagonal functor Δ to a pair $\langle c, c \rangle$ in $C \times C$. That's our left functor. It prepares the source for mapping out. The right functor maps an arbitrary pair $\langle a, b \rangle$ to the product $a \times b$ in C. That's our target for mapping in.



The adjunction is the (natural) isomorphism of the two hom-sets:

$$(C \times C)(\Delta c, \langle a, b \rangle) \cong C(c, a \times b)$$

Let's develop this intuition. As usual in category theory, an object is defined by its morphisms. A product is defined by the mapping-in property, the totality of morphisms incoming from all other objects. Hence we are interested in the hom-set between an arbitrary object c and our product $a \times b$. This is the right hand side of the picture. On the left, we are considering the mapping-out morphism from the object $\langle c, c \rangle$, which is the result of applying the functor Δ to c. Thus an adjunction relates objects that are defined by their mapping-in property and objects defined by their mapping-out property.

Another way of looking at the pair of adjoint functors is to see them as being approximately the inverse of each other. Of course, they can't, because the two categories in question are not isomorphic. Intuitively, $C \times C$ is "much bigger" than C. The functor that assigns the product $a \times b$ to every pair $\langle a, b \rangle$ cannot be injective. It must map many different pairs to the same (up to isomorphism) product. In the process, it "forgets" some of the information, just like the number 12 forgets whether it was obtained by multiplying 2 and 6 or 3 and 4. Common examples of this forgetfulness are isomorphisms such as

$$a \times b \cong b \times a$$

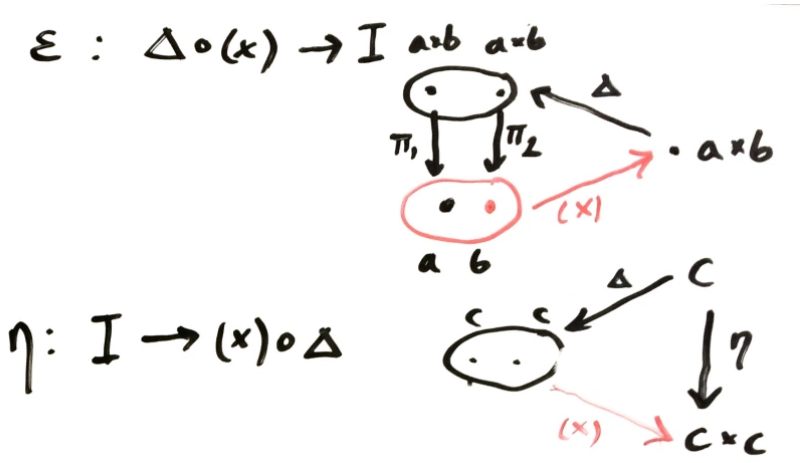
or

$$(a \times b) \times c \cong a \times (b \times c)$$

Since the product functor loses some information, its left adjoint must somehow compensate for it, essentially by making stuff up. Because the adjunction is a natural transformation, it must do it uniformly across the whole category. Given a generic object c, the only way it can produce a pair of objects is to duplicate c. Hence the diagonal functor Δ. You might say that Δ "freely" generates a pair. In almost every adjunction you can observe this interplay of "freeness" and "forgetfulness." I'm using these therm loosely, but I can be excused, because there is no formal definition of forgetful (and therefore free or cofree) functors.

Left adjoints often create free stuff. The mnemonic is that "the left" is "liberal." Right adjoints, on the other hand, are "conservative." They only use as much data as is strictly necessary and not an iota more (they also preserve limits, which the left adjoints are free to ignore). This is all relative and, as we'll see later, the same functor may be the left adjoint to one functor and the right adjoint to another.

Because of this lossiness, a round trip using both functors doesn't produce an identity. It is however "related" to the identity functor. The combination left-after-right produces an object that can be mapped back *to* the original object. Conversely, right-after-left has a mapping *from* the identity functor. These two give rise to natural transformations that are called, respectively, the counit ε and the unit η.



Here, the combination diagonal functor *after* the product functor takes a pair $\langle a, b \rangle$ to the pair $\langle a \times b, a \times b \rangle$. The counit ε then maps it back to $\langle a, b \rangle$ using a pair of projections $\langle \pi_1, \pi_2 \rangle$ (which is a single morphism in $C \times C$). It's easy to see that the family of such morphisms defines a natural transformation.

If we think for a moment in terms of set elements, then for every element of the target object, the counit *extracts* a pair of elements of the source object (the objects here are pairs of sets). Note that this mapping is not injective and, therefore, not invertible.

The other composition–the product functor after the diagonal functor–maps an object c to $c \times c$. The component of the unit natural transformation, $\eta_c \colon c \to c \times c$, is implemented using the universal property of the product. Indeed, such a morphism is uniquely determined by a pair of identity morphsims $\langle id_c, id_c \rangle$. Again, when we vary c, these morphisms combine to form a natural transformation.

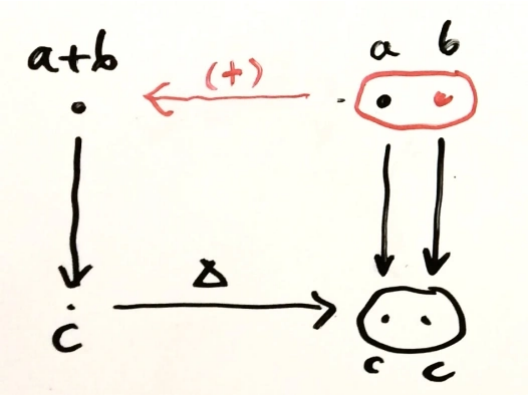
Thinking in terms of set elements, the unit *inserts* an element of the set c in the target set. And again, this is not an injective map, so it cannot be inverted.

Although in an arbitrary category we cannot talk about elements, a lot of intuitions from Set carry over to a more general setting. In a category with a terminal object, for instance, we can talk about *global elements* as mappings from the terminal object. In the absence of the terminal object, we may use other objects to define *generalized elements*. This is all in the true spirit of category theory, which defines all properties of objects in terms of morphisms.

Every construction in category theory has its dual, and the product is no exception.

A *coproduct* is defined by a mapping out property. For every pair of morphisms from, respectively, a and b to the common target c there is a unique mapping out from the coproduct $a + b$ to c. In programming, this is called *case analysis*: a function from a sum type is implemented using two functions corresponding to two cases. Conversely, given a mapping out of a coproduct, the two functions are uniquely determined due to the commuting conditions (this was all discussed in the previous post).

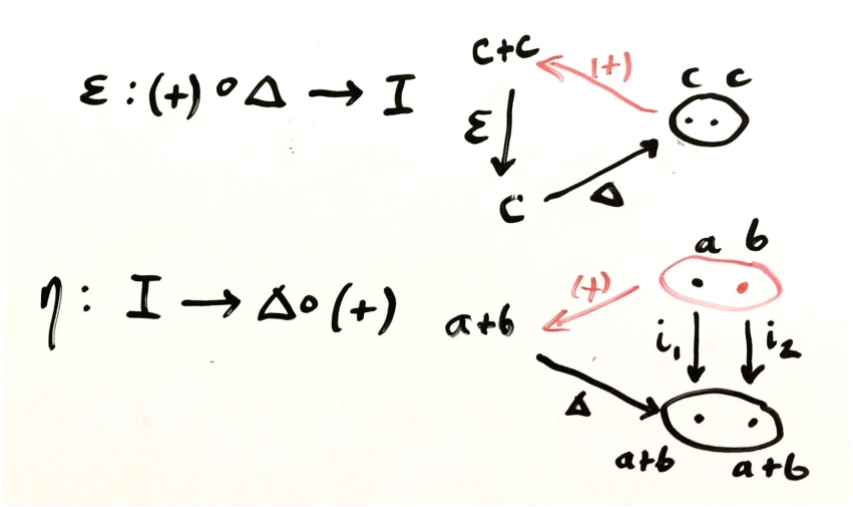
As before, this one-to-one correspondence can be neatly encapsulated as an adjunction. This time, however, the coproduct functor is the left adjoint of the diagonal functor.



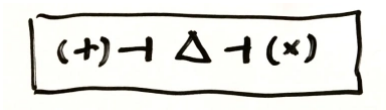
The coproduct is still the "forgetful" part of the duo, but now the diagonal functor plays the role of the *cofree* funtor, relative to the coproduct. Again, I'm using these terms loosely.

The counit now works in the category C and it "extracts a value" from the symmetric coproduct of c with c. It does it by "pattern matching" and applying the identity morphism.

The unit is more interesting. It's built from two injections, or two constructors, as we call them in programming.



I find it fascinating that the simple diagonal functor can be used to define both products and coproducts. Moreover, using terse categorical notation, this whole blog post up to this point can be summarized by a single formula.

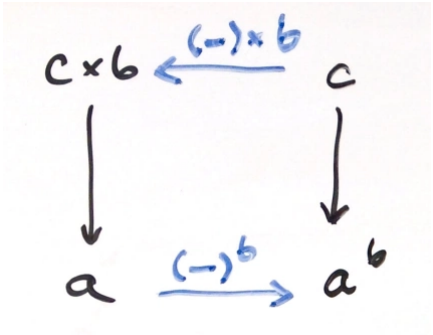


That's the power of adjunctions.

There is one more very important adjunction that every programmer should know: the exponential, or the currying adjunction. The exponential, a.k.a. the function type, is the right adjoint to the product functor. What's the product functor? Product is a bifunctor, or a functor from $C \times C$ to C. But if you fix one of the arguments, it just becomes a regular functor. We're interested in the functor $(-) \times b$ or, more explicitly:

$$(-) \times b : a \to a \times b$$

It's a functor that multiplies its argument by some fixed object b. We are using this functor to define the exponential. The exponential a^b is defined by the mapping-in property. The mappings out of the product $c \times b$ to a are in one to one correspondence with morphisms from an arbitrary object c to the exponential a^b.

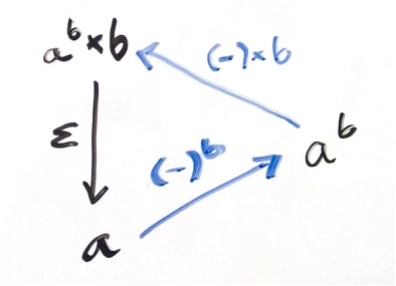


$$C(c \times b, a) \cong C(c, a^b)$$

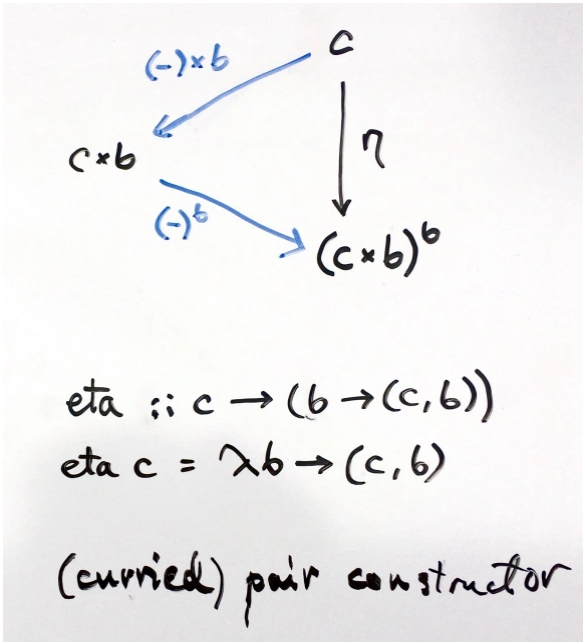
The exponential a^b is an object representing the set of morphisms from b to a, and the two directions of the isomorphism above are called *curry* and *uncurry*.

This is exactly the meaning of the universal property of the exponential I discussed in my previous post.

The counit for this adjunction extracts a value from the product of the function type (the exponential) and its argument. It's just the evaluation morphism: it applies a function to its argument.



The unit injects a value of type c into a function type $b \to c \times b$. The unit is just the curried version of the product constructor.



I want you to look closely at this formula through your programming glasses. The target of the unit is the type:

```
b -> (c, b)
```

You may recognize it as the state monad, with `b` representing the state. The unit is nothing else but the natural transformation whose component we call `return`. Coincidence? Not really! Look at the component of the counit:

```
(b -> a, b) -> a
```

It's the `extract` of the `Store` comonad.

It turns out that every adjunction gives rise to both a monad and a comonad. Not only that, every monad and every comonad give rise to adjunctions.

It seems like, in category theory, if you dig deep enough, everything is related to everything in some meaningful way. And every time you revisit a topic, you discover new insights. That's what makes category theory so exciting.

July 4, 2019

Filtered Colimits

Posted by Bartosz Milewski under Programming
1 Comment

i
10 Votes

Previously we were exploring universal constructions for products, coproducts, and exponentials. In particular, we were able to prove the distributive law:

$$(a + b) \times c \cong a \times c + b \times c$$

The power of this law is that it relates the mapping-in universal construction (product on the left) with the mapping-out one (coproduct on the right). If you take into account that products and coproducts are just special cases of limits and colimits, you may ask a more general question: under what conditions limits commute with colimits. In a cartesian closed category a product of sums is not equal to the sum of products:

$$(a + b) \times (c + d) \ncong a \times c + b \times d$$

So, in general, products don't commute with coproducts. But if you replace coproducts with a special kind of colimits, then it can be shown that:
Theorem.
In Set*, filtered colimits commute with finite limits.*

In this post I'll try to explain these terms and provide some intuition why it works and how filtered colimits are related to the more traditional notion of limits that we know from calculus.

Limits

Let's start with limits. They are like products, except that, instead of just two objects at the bottom, you have any number of objects plus a bunch of morphisms between them. That's called a diagram. Then you have an apex with arrows going down to all the objects in the diagram; and you get what is called a cone. If you have morphisms in your diagram, they form triangles. These triangles must commute. For instance, in Fig 1, we have:

$$g \circ \pi_1 = \pi_3$$

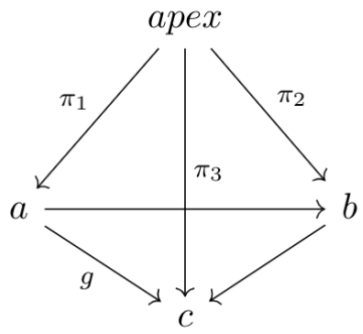


Fig. 1. A cone

This means that not all projections are independent–that you may obtain one projection from another by post-composing it with a morphism from the diagram. In Fig 1, for instance, you may extract a value of c either directly using π_3 or by applying g to the result of π_1.

A limit is defined as the universal cone with the apex Lim. It means that, if you have any other cone with some apex c, built over the same diagram, there is a unique morphism h from c to Lim that makes all the triangles commute. For instance, in Fig 2, one of the commuting conditions is:

$$\pi_1 \circ h = f_1$$

and so on. We've seen similar commuting conditions in the definition of the product.

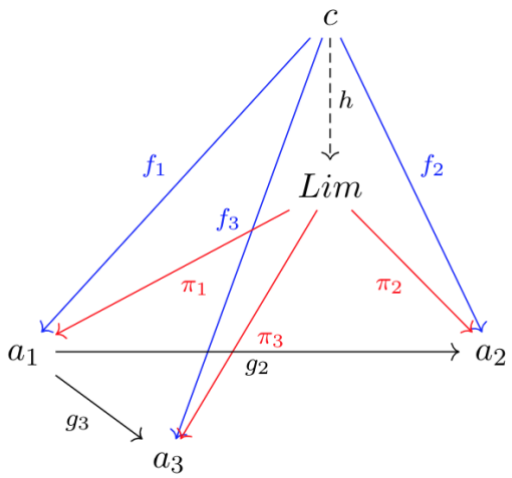


Fig. 2. A universal cone

If you think of Lim in this example as a data structure, you would implement it as a product of a_1, a_2, and a_3, together with two functions:

$$g_3 : a_1 \to a_3$$

$g_2 : a_1 \to a_2$

But because of the commuting conditions, the three values stored in Lim cannot be independent. If you pick a value for a_1, then the values for a_2 and a_3 are uniquely determined.

A limit, just like a product, is defined by a mapping-in property. If you want to define a morphism from some c to Lim, you need to provide three morphisms f_1, f_2, and f_3. However, unlike in the case of a product, these morphisms must satisfy some commuting conditions. Here, f_3 must be equal to $g_3 \circ f_1$ and $f_2 = g_2 \circ f_1$. So, really, you only need to define f_1, and that uniquely determines h. This is why the cones in Fig 2 can be simplified, as shown in Fig 3.

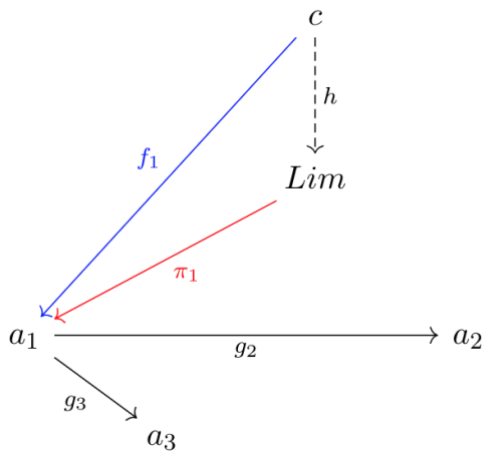


Fig. 3. A simplified universal cone

Notice that the diagram essentially forms a subcategory inside the category C, even if we don't explicitly draw all the identity morphisms or all the compositions. This is because triangles built by composing commuting triangles are again commuting. It therefore makes sense to define a diagram as a functor F from an (often much smaller) index category J to C. In our case it would be a category with just three objects, j_1, j_2, j_3, and two non-identity morphisms. (The diagram category for the product is even simpler: just two objects, no non-trivial morphisms.)

The properties of the diagram category determine the nature of cones and the nature of the limits. For instance, functors from a finite category will produce *finite limits*.

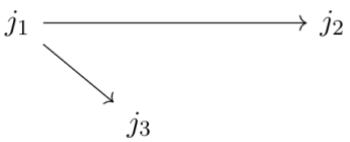


Fig. 4. Diagram category J

The diagram category J in our example has a very peculiar property: it has a cone for every pair of objects (it's a cone inside J, not to be confused with the cone in C). For instance, the pair j_2, j_3 is part of the cone with the apex j_1. This is also the apex for the (somewhat degenerate) cone based on j_1 and j_2 (with or without the connecting morphism). A category in which there is a cone for every finite subdiagram is called *cofiltered*. Limits defined by functors from cofiltered categories are called *cofiltered limits*.

The intuition is that cofiltered categories exhibit some kind of ordering. You may think of j_1 as a lower bound of j_2 and j_3. Following these bounds, you might eventually get to some kind of roots–here it's the object j_1–and these roots will dictate the behavior of cones and the behavior of limits. Things get really interesting when the diagram category is infinite, because then there is no guarantee that you'll ever reach a root. There is, for instance, no smallest (negative) integer, even though integers are ordered. You can begin to see parallels with traditional limits, like:

$$\lim_{j \to -\infty} a_j$$

That's where these ideas originally came from.

Limits in the category of sets have a particularly simple interpretation. In Set, we can use functions from the terminal object–the singleton set–to pick individual set elements.

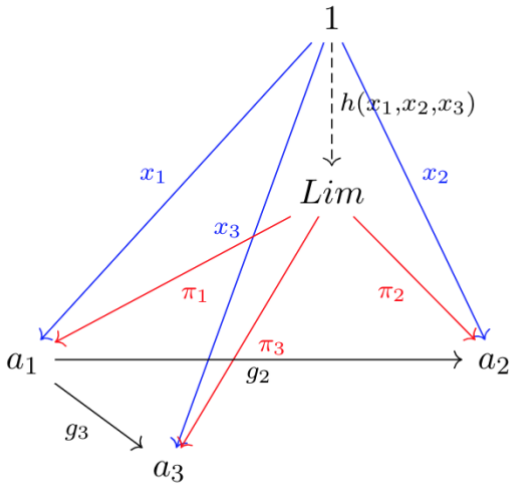


Fig. 5. Elements of the limit

For every selection in Fig 5. of x_1, x_2, x_3 there is a unique $h(x_1, x_2, x_3)$ that picks an element in Lim. But a selection of x_1, x_2, x_3 is nothing but a cone with the apex 1. So there is a one-to-one correspondence between elements of Lim and such cones. In other words, Lim is a *set of apex-1 cones*.

Colimits

Colimits are dual to limits–you get them by inverting all the arrows. So, instead of projections, you get injections, and the universal condition defines a *mapping out* of a colimit (see Fig 6).

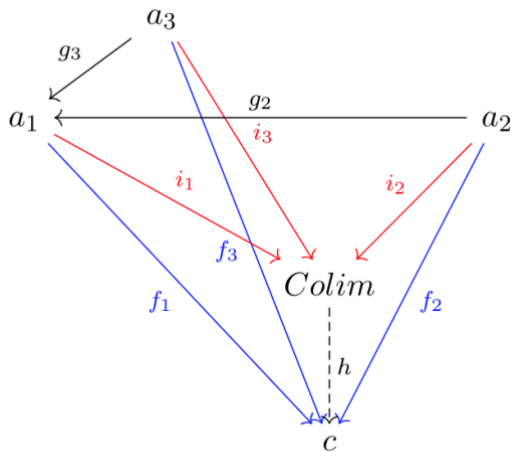


Fig. 6. A universal cocone

If you look at the colimit as a data structure, it is similar to a coproduct, except that not all the injections are independent. In the example in Fig 6, i_3 and i_2 are determined by pre-composing i_1 with g_3 and g_2, respectively. It's not clear how to implement a colimit in Haskell, so here's a pseudo-Haskell attempt using imaginary dependent-type syntax:

```
data Colim a1 a2 a3 (g2 :: a2 -> a1) (g3 :: a3 -> a1) =
         = A1 a1 | A2 a2 | A3 a3
```

To deconstruct this colimit, you only need to provide one function $f_1 : a_1 \to c$.

```
h :: (a1 -> c) -> Colim a1 a2 a3 g2 g3 -> c
h f1 (A1 a1)    = f1 a1
h f1 (A2 a2) = f1 (g2 a2)
h f1 (A3 a3) = f1 (g3 a3)
```

Granted, in a lazy language like Haskell, this would be an overkill way to store essentially just one value.

A colimit in the category of sets simplifies to a disjoint union of sets, in which some elements are identified. Suppose that the colimit $Colim_J F$ is defined by some diagram category J and a functor $F : J \to Set$. Each object j in J produces a set Fj.

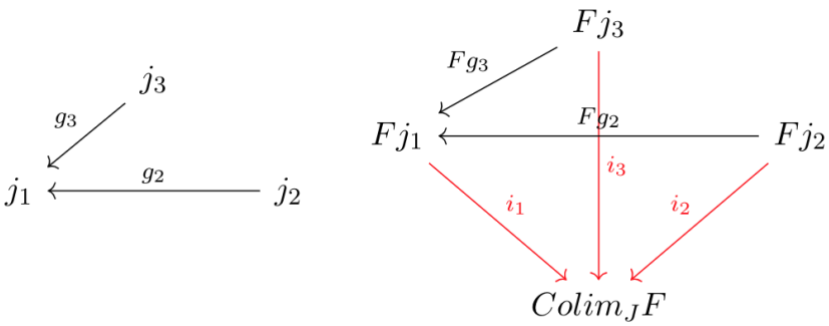


Fig. 7. Colimit in Set. On the left, the diagram category J.

The disjoint union of all these sets is a set whose elements are the pairs (x, j) where $x \in Fj$. (Notice that the sets may overlap, but each element from the overlap will be counted as many times as the number of sets it belongs to.) Coproduct injections are then functions that take an element $x \in Fj$ and map it into an element $(x, j) \in Colim_J F$. But that doesn't take into account the presence of morphisms in the diagram. These morphisms are mapped to functions between corresponding sets. For instance, in Fig 7, we can take an element $x \in Fj_2$. It is injected, using i_2, as an element $(x, j_2) \in Colim_J F$. But there is another path from Fj_2 that uses Fg_2 followed by i_1. That produces $((Fg_2)x, j_1)$. If the triangle is to commute, these two must be equal. So in the actual colimit, they must be identified. In general, any two elements of the disjoint union that satisfy this relation:

$$(x, j) \rightsquigarrow (x', j') \text{ if } \exists_{g:j \to j'} (Fg)x = x'$$

must be identified. This is not an equivalence relation, but it can be extended to one (by first symmetrizing it, and then making it transitive again). A colimit is then a quotient of the disjoint union by this equivalence.

As before, I chose this example to illustrate a special type of a diagram. This is a diagram that can be obtained using a functor from a *filtered* category. A filtered category has this property that for any finite subdiagram, there is a cocone under it. Here, for instance, the subdiagram formed by j_2 and j_3 has a cocone with the apex j_1. Again, you may think of j_1 as a kind of upper bound of j_2 and j_3. If the filtered category is finite, following upper bounds will eventually lead you to some roots. And in Set, the equivalence relation will allow you shift all the elements down to those roots. But in an infinite case (think natural numbers) there may be no largest element–no root. And that brings filtered colimits closer to the intuition we have for limits in calculus. In fact, all the interesting filtered colimits are based on infinite diagrams.

Commuting Limits and Colimits

What does it mean for a limit to commute with a colimit? A single colimit is generated by a functor from some index category $I \to C$. What we need is a bunch of such colimits so that we can take a limit over those. Therefore we need a bunch of functors $I \to C$. Moreover, those colimits have to form a diagram. So we need another index category J to parameterize those functors. Altogether, we need a functor of two arguments:

$$F : I \times J \to C$$

It follows that, for any given j in J we have a functor $F(-, j) : I \to C$. We can take a colimit of that. Then we gather those colimits into a diagram whose shape is defined by J, and then take its limit. We get:

$$Lim_J(Colim_I F)$$

Alternatively, when we fix some i in I, we get a functor $F(i, -) : J \to C$. We can take a limit of that. Then we can gather all those limits and form a diagram whose shape is defined by I. Finally we can take a colimit of that:

$$Colim_I(Lim_J F)$$

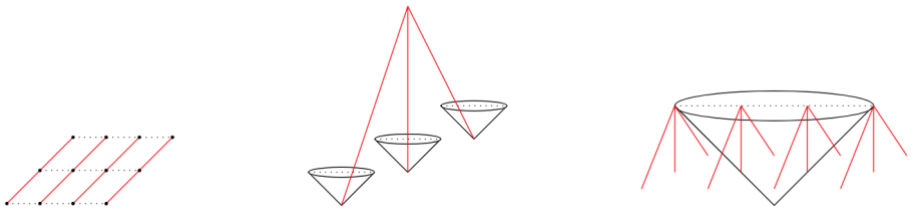


Fig. 8. Commuting limits (red diagram of shape J) and colimits (black diagram of shape I)

It's not difficult to construct the mapping:

$$Colim_I(Lim_J F) \to Lim_J(Colim_I F)$$

using the universal property, since the colimit has the mapping-out property. It's the other way around that's tricky. But it always works in the special case when I is filtered, J is finite, and C is Set.

Here's the sketch of this amazing proof, which you can find in Saunders Mac Lane's *Categories for the Working Mathematician*.

Since the target of the functor is Set, it might help to visualize its image as a rectangular array of sets. A fixed j picks up a row of such sets, whereas a fixed i picks up a column. Because we are dealing with sets, we can try to define the mapping:

$$Lim_J(Colim_I F) \to Colim_I(Lim_J F)$$

pointwise. Let's pick an element of the limit on the left. As we've established earlier, a limit in Set is a set of apex-1 cones. So let's pick one such cone. It's just a selection of elements from a bunch of colimits.

As we've seen before, a colimit in Set is a discriminated union with some identifications. So our apex-1 cone will pick a set of representatives, one per colimit, say (x_n, i_n). Any time there is a morphism $g : i_n \to i'_n$, we can replace one representative with another $(g(x_n), i'_n)$. The intuition is that we can slide the representatives horizontally within each row along morphisms.

If I is a filtered category, then for any finite number of objects i_n, we can always find a common root (it will be the apex i of a cocone formed by i_n in I). So we can slide all the representatives to a single column. In other words, our cone can be brought to a set of representatives (y_n, i), with a common i.

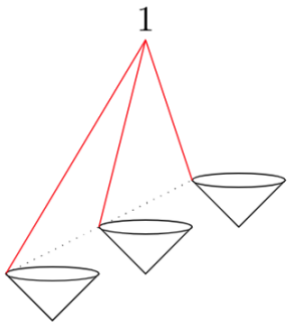


Fig. 9. A single cone after shifting representatives from all colimits to a common column

But that's just a cone over J. It's an element of $Lim_J F$. And we can inject it into a colimit over I to get an element of $Colim_I(Lim_J F)$. We have thus defined our mapping.

Conclusion

If you didn't get the proof the first time, don't get discouraged. Take a break, sleep over it, and then read it slowly again. Make sure you have internalized all the definitions. Draw your own pictures. The two major tricks are: (1) visualizing an element of a limit as a cone originating from the singleton set, and (2) the idea of sliding the elements of multiple colimits to a common column.

The importance of this theorem is that it tells you when and how you can define mappings out of limits. For instance, how to define functions from a product or from an end.

Acknowledgment

I'm grateful to Derek Elkins for correcting mistakes in the original version of this post.