Chapter 12

Algebras

The essence of algebra is the formal manipulation of expressions. But what is an expression, and how do we manipulate it?

The first things to observe about algebraic expressions like $2(x + y)$ or $ax^2 + bx + c$ is that there are infinitely many of them. There is a finite number of rules for making them, but these rules can be used in infinitely many combinations. This suggests that the rules are used *recursively*.

In programming, expressions are virtually synonymous to (parsing) trees. Consider this simple example of an arithmetic expression:

```
data Expr = Val Int | Plus Expr Expr
```

It's a recipe for building trees. We start with little trees using the `Val` constructor. We then plant these seedlings into nodes, and so on.

```
e2 = Val 2
e3 = Val 3
e5 = Plus e2 e3
e7 = Plus e5 e2
```

Such recursive definitions work perfectly well in a programming language. The problem is that every new recursive data structure would require its own library of functions that operate on it.

From type-theory point of view, we've been able to define recursive types, such as natural numbers or lists, by providing, in each case, specific introduction and elimination rules. What we need is something more general, a procedure for generating arbitrary recursive types from simpler pluggable components.

There are two orthogonal concerns when it comes to recursive data structures. One is the machinery of recursion. The other is the pluggable components to be used by recursion.

We know how to work recursion: We assume that we know how to construct small trees. Then we use the recursive step to plant those trees into nodes to make bigger trees.

Category theory tells us how to formalize this imprecise description.

12.1 Algebras from Endofunctors

The idea of planting smaller trees into nodes requires that we formalize what it means to have a data structure with holes—a "container for stuff." This is exactly what functors are for. Because we want to use these functors recursively, they have to be *endo*-functors.

For instance, the endofunctor from our earlier example would be defined by the following data structure, where `x` marks the spots:

```
data ExprF x = ValF Int | PlusF x x
```

Information about all possible shapes of expressions is abstracted into a single functor.

The other important piece of information is the recipe for evaluating expressions. This, too, can be encoded using the same endofunctor.

Thinking recursively, let's assume that we know how to evaluate all subtrees of a larger expression. Then the remaining step is to plug these results into the top level node and evaluate it.

For instance, suppose that the `x`'s in the functor were replaced by integers—the results of evaluation of the subtrees. It's pretty obvious what we should do in the last step. If the top of the tree is a leaf `ValF` (which means there were no subtrees to evaluate) we'll just return the integer stored in it. If it's a `PlusF` node, we'll add the two integers in it. This recipe can be encoded as:

```
eval :: ExprF Int -> Int
eval (ValF n)    = n
eval (PlusF m n) = m + n
```

We have made some seemingly obvious assumptions based on our experience. For instance, since the node was called `PlusF` we assumed that we should add the two numbers. But multiplication or subtraction would work equally well.

Since the leaf `ValF` contained an integer, we assumed that the expression should evaluate to an integer. But there is an equally plausible evaluator that pretty-prints the expression by converting it to a string, and using concatenation instead of addition:

```
pretty :: ExprF String -> String
pretty (ValF n)    = show n
pretty (PlusF s t) = s ++ " + " ++ t
```

In fact there are infinitely many evaluators, some sensible, others less so, but we shouldn't be judgmental. Any choice of the target type and any choice of the evaluator should be equally valid. This leads to the following definition:

An *algebra* for an endofunctor F is a pair (A, α). The object A is called the *carrier* of the algebra, and the evaluator $\alpha\colon FA \to A$ is called the *structure map*.

In Haskell, given the functor `f` we define:

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

Notice that the evaluator is *not* a polymorphic function. It's a specific choice of a function for a specific type `a`. There may be many choices of the carrier types and there be many different evaluators for a given type. They all define separate algebras.

We have previously defined two algebras for `ExprF`. This one has `Int` as a carrier:

```
eval :: Algebra ExprF Int
eval (ValF n)    = n
eval (PlusF m n) = m + n
```

and this one has `String` as a carrier:

```
pretty :: Algebra ExprF String
pretty (ValF n)    = show n
pretty (PlusF s t) = s ++ " + " ++ t
```

12.2 Category of Algebras

Algebras for a given endofunctor F form a category. An arrow in that category is an algebra morphism, which is a structure-preserving arrow between their carrier objects.

Preserving structure in this case means that the arrow must commute with the two structure maps. This is where functoriality comes into play. To switch from one structure map to another, we have to be able to lift an arrow that goes between their carriers.

Given an endofunctor F, an *algebra morphism* between two algebras (A, α) and (B, β) is an arrow $f : A \to B$ that makes this diagram commute:

$$\begin{array}{ccc} FA & \xrightarrow{Ff} & FB \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} \\ A & \xrightarrow{f} & B \end{array}$$

In other words, the following equation must hold:

$$f \circ \alpha = \beta \circ Ff$$

The composition of two algebra morphisms is again an algebra morphism, which can be seen by pasting together two such diagrams. The identity arrow is also an algebra morphism, because

$$id_A \circ \alpha = \alpha \circ F(id_A)$$

(a functor maps identity to identity).

The commuting condition in the definition of an algebra morphism is very restrictive. Consider for instance a function that maps an integer to a string. In Haskell there is a `show` function (actually, a method of the `Show` class) that does it. It is *not* an algebra morphism from `eval` to `pretty`.

Exercise 12.2.1. *Show that `show` is not an algebra morphism. Hint: Consider what happens to a `PlusF` node.*

Initial algebra

The initial object in the category of algebras is called the *initial algebra* and, as we'll see, it plays a very important role.

By definition, the initial algebra (I, i) has a unique algebra morphism f from it to any other algebra (A, α). Diagrammatically:

$$\begin{array}{ccc} FI & \xrightarrow{Ff} & FA \\ \downarrow{\scriptstyle i} & & \downarrow{\scriptstyle \alpha} \\ I & \dashrightarrow^{f} & A \end{array}$$

This unique morphism is called a *catamorphism* for the algebra (A, α).

Exercise 12.2.2. *Let's define two algebras for the following functor:*

```
data FloatF x = Num Float | Op x x
```

The first algebra:

```
addAlg :: Algebra FloatF Float
addAlg (Num x) = log x
addAlg (Op x y) = x + y
```

The second algebra:

```
mulAlg :: Algebra FloatF Float
mulAlg (Num x) = x
mulAlg (Op x y) = x * y
```

Make a convincing argument that `log` *(logarithm) is an algebra morphism between these two. (*`Float` *is a built-in floating-point number type.)*

12.3 Lambek's Lemma and Fixed Points

Lambek's lemma says that the structure map i of the initial algebra is an isomorphism.

The reason for it is the self-similarity of algebras. You can lift any algebra (A, α) using F, and the result $(FA, F\alpha)$ is also an algebra with the structure map $F\alpha \colon F(FA) \to FA$.

In particular, if you lift the initial algebra (I, i), you get an algebra with the carrier FI and the structure map $Fi \colon F(FI) \to FI$. It follows then that there must be a unique algebra morphism from the initial algebra to it:

$$\begin{array}{ccc} FI & \xrightarrow{Fh} & F(FI) \\ \downarrow{\scriptstyle i} & & \downarrow{\scriptstyle Fi} \\ I & \dashrightarrow^{h} & FI \end{array}$$

This h is the inverse of i. To see that, let's consider the composition $i \circ h$. It is the arrow at the bottom of the following diagram

$$\begin{array}{ccccc} FI & \xrightarrow{Fh} & F(FI) & \xrightarrow{Fi} & FI \\ \downarrow{\scriptstyle i} & & \downarrow{\scriptstyle Fi} & & \downarrow{\scriptstyle i} \\ I & \overset{h}{\dashrightarrow} & FI & \xrightarrow{\quad i \quad} & I \end{array}$$

This is a pasting of the original diagram with a trivially commuting diagram. Therefore the whole rectangle commutes. We can interpret this as $i \circ h$ being an algebra morphism from (I, i) to itself. But there already is such an algebra morphism—the identity. So, by uniqueness of the mapping out from the initial algebra, these two must be equal:

$$i \circ h = id_I$$

Knowing that, we can now go back to the previous diagram, which states that:

$$h \circ i = Fi \circ Fh$$

Since F is a functor, it maps composition to composition and identity to identity. Therefore the right hand side is equal to:

$$F(i \circ h) = F(id_I) = id_{FI}$$

We have thus shown that h is the inverse of i, which means that i is an isomorphism. In other words:

$$FI \cong I$$

We interpret this identity as stating that I is a fixed point of F (up to isomorphism). The action of F on I "doesn't change it."

There may be many fixed points, but this one is the *least fixed point* because there is an algebra morphism from it to any other fixed point. The least point of an endofunctor F is denoted μF, so we have:

$$I = \mu F$$

Fixed point in Haskell

Let's consider how the definition of the fixed point works with our original example:

```
data ExprF x = ValF Int | PlusF x x
```

Its fixed point is a data structure defined by the property that `ExprF` acting on it reproduces it. If we call this data structure `Expr`, the fixed point equation becomes (in pseudo-Haskell):

```
Expr = ExprF Expr
```

Expanding `ExprF` we get:

```
Expr = ValF Int | PlusF Expr Expr
```

Compare this with the recursive definition (actual Haskell):

```
data Expr = Val Int | Plus Expr Expr
```

We get a recursive data structure as a solution to the fixed-point equation.

In Haskell, we can define a fixed point data structure for any functor (or even just a type constructor). As we'll see later, this doesn't always give us the carrier of the initial algebra. It only works for those functors that have the "leaf" component.

Let's call `Fix f` the fixed point of a functor `f`. Symbolically, the fixed-point equation can be written as:

$$f(\mathrm{Fix} f) \cong \mathrm{Fix} f$$

or, in code,

```
data Fix f where
  In :: f (Fix f) -> Fix f
```

The data constructor `In` is exactly the structure map of the initial algebra whose carrier is `Fix f`. Its inverse is:

```
out :: Fix f -> f (Fix f)
out (In x) = x
```

The Haskell standard library contains a more idiomatic definition:

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

To create terms of the type `Fix f` we often use "smart constructors." For instance, with the `ExprF` functor, we would define:

```
val :: Int -> Fix ExprF
val n = In (ValF n)

plus :: Fix ExprF -> Fix ExprF -> Fix ExprF
plus e1 e2 = In (PlusF e1 e2)
```

and use it to generate expression trees like this one:

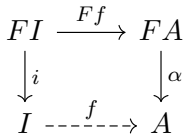
```
e9 :: Fix ExprF
e9 = plus (plus (val 2) (val 3)) (val 4)
```

12.4 Catamorphisms

Our goal, as programmers, is to be able to perform a computation over a recursive data structure—to "fold" it. We now have all the ingredients.

The data structure is defined as a fixed point of a functor. An algebra for this functor defines the operation we want to perform. We've seen the fixed point and the

algebra combined in the following diagram:



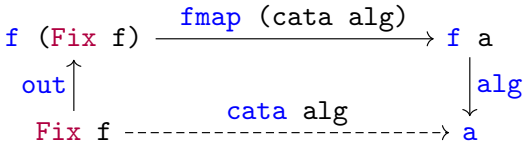
that defines the catamorphism f for the algebra (A, α).

The final piece of information is the Lambek's lemma, which tells us that i could be inverted because it's an isomorphism. It means that we can read this diagram as:

$$f = \alpha \circ Ff \circ i^{-1}$$

and interpret it as a recursive definition of f.

Let's redraw this diagram using Haskell notation. The catamorphism depends on the algebra so, for the algebra with the carrier `a` and the evaluator `alg`, we'll have the catamorphism `cata alg`.



By simply following the arrows, we get this recursive definition:

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

Here's what's happening: `cata alg` is applied to some `Fix f`. Every `Fix f` is obtained by applying `In` to a functorful of `Fix f`. The function `out` "strips" this data constructor.

We can evaluate the functorful of `Fix f` by `fmap`'ing `cata alg` over it. This is a recursive application. The idea is that the trees inside the functor are smaller than the original tree, so the recursion eventually terminates. It terminates when it hits the leaves.

After this step, we are left with a functorful of values, and we apply the evaluator `alg` to it, to get the final result.

The power of this approach is that all the recursion is encapsulated in one data type and one library function: We have the definition of `Fix` and the catamorphism. The client of the library has only to provide the *non-recursive* pieces: the functor and the algebra. These are much easier to deal with.

Examples

We can immediately apply this construction to our earlier examples. You can check that:

```
cata eval e9
```

evaluates to 9 and

```
cata pretty e9
```

evaluates to the string "2 + 3 + 4".

Sometimes we want to display the tree on multiple lines with indentation. This requires passing a depth counter to recursive calls. There is a clever trick that uses a function type as a carrier:

```
pretty' :: Algebra ExprF (Int -> String)
pretty' (ValF n) i = indent i ++ show n
pretty' (PlusF f g) i = f (i + 1) ++ "\n" ++
                        indent i ++ "+" ++ "\n" ++
                        g (i + 1)
```

The auxiliary function `indent` replicates the space character:

```
indent n = replicate (n * 2) ' '
```

The result of:

```
cata pretty' e9 0
```

when printed, looks like this:

```
    2
  +
    3
+
  4
```

Let's try defining algebras for other familiar functors. The fixed point of the `Maybe` functor:

```
data Maybe x = Nothing | Just x
```

after some renaming, is equivalent to the type of natural numbers

```
data Nat = Z | S Nat
```

An algebra for this functor consists of a choice of the carrier `a` and an evaluator:

```
alg :: Maybe a -> a
```

The mapping out of `Maybe` is determined by two things: the value corresponding to `Nothing` and a function `a->a` corresponding to `Just`. In our discussion of the type of natural numbers we called these `init` and `step`. We can now see that the elimination rule for `Nat` is the catamorphism for this algebra.

The list type that we've seen previously is equivalent to a fixed point of the following functor:

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

An algebra for this functor is a mapping out

```
alg :: ListF a c -> c
alg NifF = init
alg (ConsF a c) = step (a, c)
```

which is determined by the value `init` and the function `step`:

```
init :: c
step :: (a, c) -> c
```

A catamorphism for such an algebra is the list recursor:

```
recList :: c -> ((a, c) -> c) -> (List a -> c)
```

where `(List a)` is can be identified with the fixed point `Fix (ListF a)`.

We've seen before a recursive function that reverses a list. It was implemented by appending elements to the end of a list, which is very inefficient. It's easy to rewrite this function using a catamorphism, but the problem remains.

Prepending elements, on the other hand, is cheap. A better algorithm would traverse the list, accumulating elements in a last-in-first-out stack, and then pop them one-by-one and prepend them to a new list.

The stack regimen can be implemented by using composition of closures: each closure is a function that remembers its environment. Here's the algebra whose carrier is a function type:

```
revAlg :: Algebra (ListF a) ([a]->[a])
revAlg NilF = id
revAlg (ConsF a f) = \as -> f (a : as)
```

At each step, this algebra creates a function that will apply the previous function `f` to the result of prepending the current element `a` to its argument. It's this element that's remembered by the closure. The catamorphism for this algebra accumulates a stackful of such closures. To reverse a list, we apply the result of the catamorphism for this algebra to an empty list:

```
reverse :: Fix (ListF a) -> [a]
reverse as = (cata revAlg as) []
```

This trick is at the core of the fold-left funcion, `foldl`. Care should be taken when using it, because of the danger of stack overflow.

Lists are so common that their eliminators (called "folds") are included in the standard library. But there are infinitely many possible recursive data structures, each generated by its own functor, and we can use the same catamorphism on all of them.

12.5 Initial Algebra from Universality

Another way of looking at the initial algebra, at least in **Set**, is to view it as a collection of catamorphisms that, as a whole, hint at the existence of an underlying object. Instead of seeing μF as a set of trees, we can look at it as a set of functions from algebras to their carriers.

In a way, this is just another manifestation of the Yoneda lemma: every data structure can be described either by mappings in or mappings out. The mappings in, in this case, are the constructors of the recursive data structure. The mappings out are all the catamorphisms that can be applied to it.

First, let's make the polymorphism in the definition of `cata` explicit:

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

and then flip the arguments. We get:

```
cata' :: Functor f => Fix f -> forall a. Algebra f a -> a
cata' (In x) = \alg -> alg (fmap (flip cata' alg) x)
```

The function `flip` reverses the order of arguments to a function:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f b a = f a b
```

This gives us a mapping from `Fix f` to a set of polymorphic functions.

Conversely, given a polymorphic function of the type:

```
forall a. Algebra f a -> a
```

we can reconstruct `Fix f`:

```
uncata :: Functor f => (forall a. Algebra f a -> a) -> Fix f
uncata alga = alga In
```

In fact, these two functions, `cata'` and `uncata`, are the inverse of each other, establishing the isomorphism between `Fix f` and the type of polymorphic functions:

```
data Mu f = Mu (forall a. Algebra f a -> a)
```

Folding over `Mu f` is easy, since `Mu` carries in itself its own set of catamorphisms:

```
cataMu :: Algebra f a -> (Mu f -> a)
cataMu alg (Mu h) = h alg
```

You might be wondering how one can construct terms of the type `Mu f` for, let's say lists. It can be done using recursion:

```
fromList :: forall a. [a] -> Mu (ListF a)
fromList as = Mu h
  where h :: forall x. Algebra (ListF a) x -> x
        h alg = go as
          where
            go [] = alg NilF
            go (n: ns) = alg (ConsF n (go ns))
```

To compile this code you have to use the language pragma:

```
{-# language ScopedTypeVariables #-}
```

which puts the type variable `a` in the scope of the `where` clause.

Exercise 12.5.1. *Write a test that takes a list of integers, converts it to the `Mu` form, and calculates the sum using `cataMu`.*

12.6 Initial Algebra as a Colimit

In general, there is no guarantee that the initial object in the category of algebras exists. But if it exists, Lambek's lemma tells us that it's a fixed point of the endofunctor for these algebras. The construction of this fixed point is a little mysterious, since it involves tying the recursive knot.

Loosely speaking, the fixed point is reached after we apply the functor infinitely many times. Then, applying it once more wouldn't change anything. This idea can be made precise, if we take it one step at a time. For simplicity, let's consider algebras in the category of sets, which has all the nice properties.

We've seen, in our examples, that building instances of recursive data structures always starts with the leaves. The leaves are the parts in the definition of the functor that don't depend on the type parameter: the `NilF` of the list, the `ValF` of the tree, the `Nothing` of the `Maybe`, etc.

We can tease them out if we apply our functor F to the initial object—the empty set 0. Since the empty set has no elements, the instances of the type $F0$ are leaves only.

Indeed, the only inhabitant of the type `Maybe Void` is constructed using `Nothing`. The only inhabitants of the type `ExprF Void` are `ValF n`, where `n` is an `Int`.

In other words, $F0$ is the "type of leaves" for the functor F. Leaves are trees of depth one. For the `Maybe` functor there's only one—the type of leaves for this functor is a singleton:

```
m1 :: Maybe Void
m1 = Nothing
```

In the second iteration, we apply F to the leaves from the previous step and get trees of depth at most two. Their type is $F(F0)$.

For instance, these are all the terms of the type `Maybe(Maybe Void)`:

```
m2, m2' :: Maybe (Maybe Void)
m2 = Nothing
m2' = Just Nothing
```

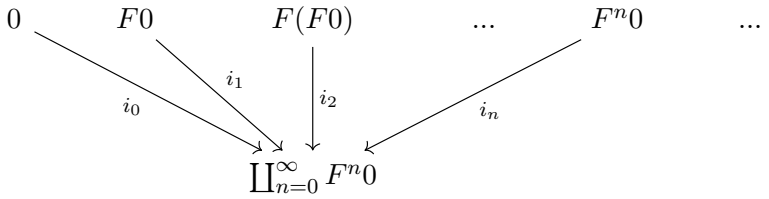
We can continue this process, adding deeper and deeper trees at each step. In the n-th iteration, the type $F^n 0$ (n-fold application of f to the initial object) describes all trees of depth up to n. However, for every n, there are still infinitely many trees of depth greater than n that are not covered.

If we knew how to define $F^\infty 0$, it would cover all possible trees. But the next best thing we could try is to gather all these partial trees into one infinite sum type. Just like we have defined sums of two types, we can define sums of many types, including infinitely many.

An infinite sum (a coproduct):

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

is just like a finite sum, except that it has infinitely many constructors i_n:



It has the universal mapping-out property, just like the sum of two types, only with infinitely many cases. (Obviously, we can't express it in Haskell.)

To construct a tree of depth n, we would first select it from $F^n 0$ and use the n-th constructor i_n to inject it into the sum.

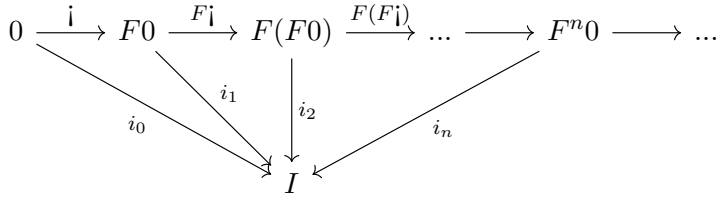
There is only one problem: the same tree shape can also be constructed using any of the $F^m 0$, with $m > n$.

Indeed, we've seen the leaf `Nothing` appear in `Maybe Void` and `Maybe(Maybe Void)`. In fact it shows up in any nonzero power of `Maybe` acting on `Void`.

Similarly, `Just Nothing` shows up in all powers starting with two.

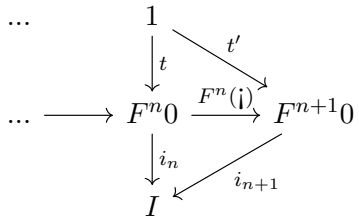
`Just(Just(Nothing))` shows up in all powers starting with three, and so on...

But there is a way to get rid of all these duplicates. The trick is to replace the sum by the colimit. Instead of a diagram consisting of discrete objects, we can construct a chain. Let's call this chain Γ, and its colimit $I = Colim\,\Gamma$.



It's almost the same as the sum, but with additional arrows at the base of the cocone. These arrows are the liftings of the unique arrow ¡ that goes from the initial object to $F0$ (we called it `absurd` in Haskell). The effect of these arrows is to collapse the set of infinitely many copies of the same tree down to just one representative.

To see that, consider for instance a tree of depth n. It can be first found as an element of $F^n 0$, that is to say, as an arrow $t\colon 1 \to F^n 0$. It is injected into the colimit I as the composite $i_n \circ t$.



The same shape of a tree is also found in $F^{n+1}0$, as the composite $t' = F^n(¡) \circ t$. It is injected into the colimit as the composite $i_{n+1} \circ t' = i_{n+1} \circ F^n(¡) \circ t$.

This time, however, we have the commuting triangle as the face of the cocone:

$$i_{n+1} \circ F^n(¡) = i_n$$

which means that

$$i_{n+1} \circ t' = i_{n+1} \circ F^n(¡) \circ t = i_n \circ t$$

The two copies of the tree have been identified in the colimit. You can convince yourself that this procedure removes all duplicates.

We can prove directly that $I = Colim\,\Gamma$ is the initial algebra. There is however one assumptions that we have to make: the functor F must preserve the colimit. The colimit of $F\Gamma$ must be equal to FI.

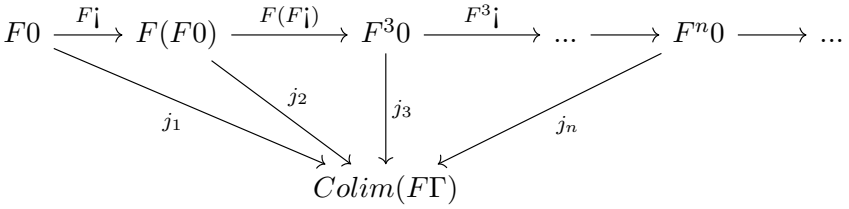
$$\mathrm{Colim}(F\Gamma) \cong FI$$

Fortunately, this assumption holds in **Set**.

Here's the sketch of the proof: First we'll construct an arrow $I \to FI$ and then an arrow in the opposite direction. We'll skip the proof that they are the inverse of each other.

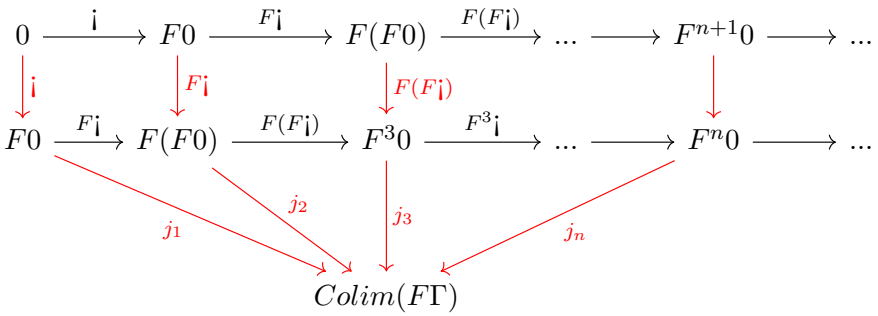
We start with the universality of the colimit. If we can construct a cocone from the chain Γ to $Colim(F\Gamma)$ then, by universality, there must be an arrow from I to $ColimF\Gamma$. And the latter, by our assumption, is isomorphic to FI. So we'll have a mapping $I \to FI$.

To construct this cocone, notice that $Colim(F\Gamma)$ is, by definition, the apex of a cocone $F\Gamma$.



The diagram $F\Gamma$ is the same as Γ, except that it's missing the naked initial object at the start of the chain.

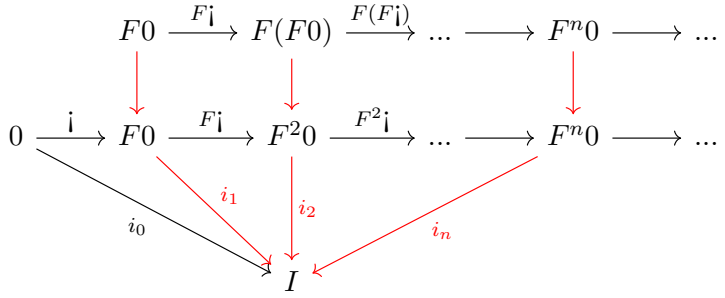
The spokes of the cocone we are looking for are marked in red in the diagram below:



And since $I = Colim\Gamma$ is the apex of the universal cocone based on Γ, there must be a unique mapping out of it to $Colim(F\Gamma)$, which is FI:

$$I \to FI$$

Next, notice that the chain $F\Gamma$ is a sub-chain of Γ, so it can be embedded in it. It means that we can construct a cocone from $F\Gamma$ to the apex I by going through (a sub-chain of) Γ.

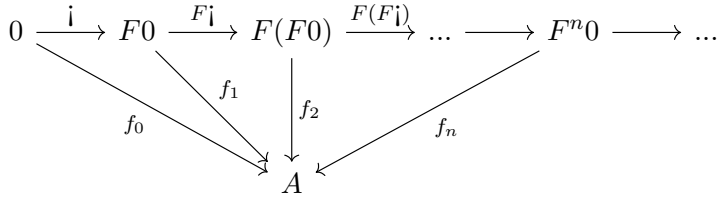


From the universality of $Colim(F\Gamma)$ it follows that there is a mapping out

$$Colim(F\Gamma) \cong FI \to I$$

This shows that I is a carrier of an algebra. In fact, it can be shown that the two mappings are the inverse of each other, as we would expect from the Lambek's lemma.

To show that this is indeed the initial algebra, we have to construct a mapping out of it to an arbitrary algebra $(A, \alpha\colon FA \to A)$. Again, we can use universality, if we can construct a cocone from Γ to A.



The zero'th spoke of this cocone goes from 0 to A, so it's just $f_0 = ¡$.
The first one, $F0 \to A$, is $f_1 = \alpha \circ Ff_0$, because $Ff_0\colon F0 \to FA$.
The third one, $F(F0) \to A$ is $f_2 = \alpha \circ Ff_1$. And so on...

The unique mapping from I to A is then our catamorphism. With some more diagram chasing, it can be shown that it's indeed an algebra morphism.

Notice that this construction only works if we can "prime" the process by creating the leaves of the functor. If, on the other hand, $F0 \cong 0$, then there are no leaves, and all further iterations will just reproduce the 0.