Recursion Schemes for Mathematicians

An introduction to induction in programming

10 September 2020

Recursion schemes are a neat construction that allows one to better structure and reason about recursive functions. Instead of viewing recursion as functions that call themselves, these schemes allow us to take a higher-order point of view. Most of recursive definitions may be abstracted as some operator that takes a function which knows how to deal with individual parts of a structure and turns it into a function defined for the entire structure. The canonical example for this perspective is turning the binary function $+$, which adds two numbers, into a new function that sums all elements of a list with arbitrary length[1]. Another use case is, when writing an evaluator for an expression tree, to only tell your program how to deal with each node and let a recursion scheme turn it into a full evaluator.

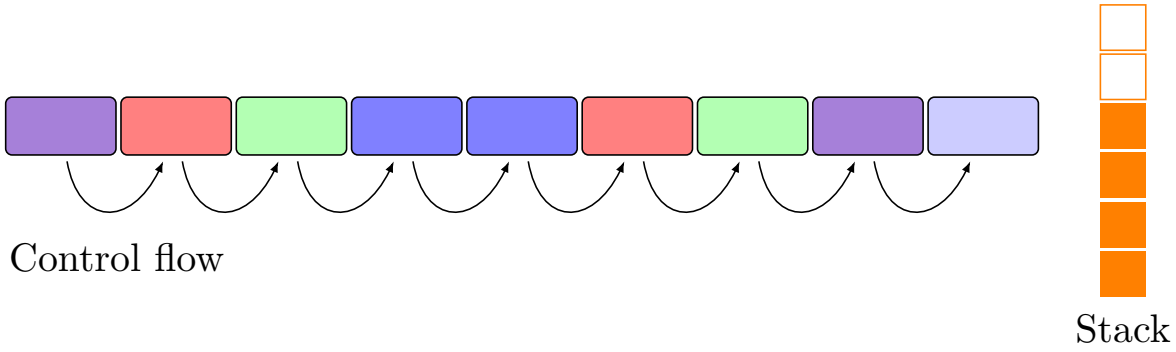
In this post, we will take a look at the three simplest varieties of recursion schemes: catamorphims, anamorphisms, and hylomorphisms; all of them deeply linked with structural induction. As we will see, they respectively encapsulate the notions of folding a data structure, constructing a data structure and using a data structure as an intermediate step.

The first time I heard about recursion schemes was after stumbling with the paper Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire by Erik Meijer, Maarten Fokkinga, and Ross Paterson. It is a real gem of functional wisdom but the authors use a notation called Squiggol, which I had a really hard time trying to grasp. Fortunately, I later found Patrick Thomson's excellent series of posts, which explain recursion schemes using Haskell code. After absorbing this content, I tried to explain the idea to some friends who don't know Haskell but know some category theory. This post is an enriched version of that presentation with more cohesion and much less of my bad drawings.

In the spirit of the original presentation, I've decided to make this post completely language-agnostic. I also tried to keep any type theory or lambda calculus to a minimum in order to make it easier to grasp. My hope is that anyone with some experience in programming and mathematics will be able to read this, instead of confining this post only to people that know a certain programming language. Of course, in doing that I'm risking that no one besides me will actually understand this post. But, very well, I will do my best.

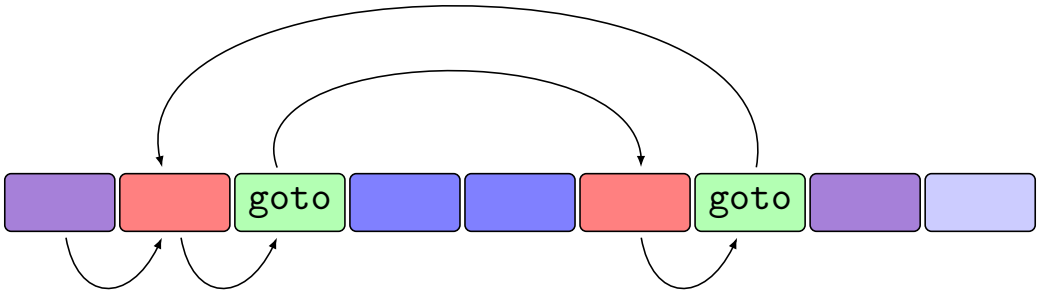
A not so historical introduction

We begin our journey in the prehistory of programming; that time when computers where still mostly made of coconuts and mammoth fur. At the time, the way to program was quite unstructured. The code of a program was essentially given by a sequence of commands whereas the data was simply piled up in a stack.



But the program does not have to execute the instructions in order. Oh no, it would be rather boring and inexpressive... In this programming paradigm, there is a special command called `goto` which allows one to jump to any labeled

part of the program. In fact, this is the only command responsible for the program's control flow.



The `goto` is a pretty powerful construction, but it also has its caveats. So much for its use to be considered harmful. Although any kind of program flow can be constructed using only `goto`, it may become too confuse. The `goto` statement makes it too easy to write spaghetti code, which is practically impossible to debug.

After prehistory, we arrive into the societal period of imperative programming. In here, the programming languages become *structured*. Data is no long viewed as simply a stack of memory but classified into types. There are some primitive types as well as structures to combine them into more complex types. In a language like C, for example, there are `struct` types, `union` types, and array types.

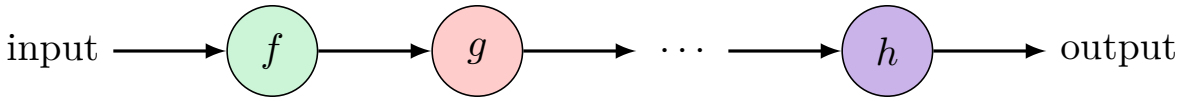
The changes also arrived to the control flow and an effort was made to tame the once wild `goto`. The programmers from that time analysed its most common use cases and created new statements fulfilling each of these. You are probably acquainted to them as `for` loops, `while` loops, `switch/case` statements, function and subroutine declarations and `if/else` statements; just to name a few.

Both in regards to data and control flow, the movement we encounter in here consists of substituting general purpose structures that concretely represent the instructions we give to the processor by specific structures having more abstract roles in the code. In terms of computational expressiveness, nothing changes. What the processor sees is the same in both unstructured and structured programming. The benefit lies in the programmer's expressiveness. With more specific statements, it becomes easier to write larger, more complex programs as well as properly debug them. As an example, we may notice that it is possible to guarantee that a `for` loop always terminates if it is iterating over a

block of code that is known to terminate. No strange thing can happen. On the other side, the general character of the `goto` allows us to simulate the behavior of a `for` but there is no guarantee that a program with a `goto` statement must terminate.

Until now, we were looking at a programming paradigm called *imperative programming*, but the world of programming languages is not so uniform. Other paradigms exist. And the one that mostly fits what we will see today is called *functional programming*.

While the imperative paradigm views the code as a sequence of commands the programmer gives the computer to execute (hence the name, *imperative*), the functional paradigm views the code as a composition of functions in the mathematical sense. A function takes a value as input, does some processing to it, calling other functions for this, and returns another value as output.



If every program only consisted of applying a finite amount of previously defined functions, the language's expressiveness would be rather limited. To overcome this, we need some form of control flow, which is achieved via recursion.

A *recursive function* is a function that, in order to process its input into the output, may call itself in an intermediate step. Probably the most famous recursive function is the factorial, defined as

$$\mathrm{fat}(n) = \begin{cases} 1, & n = 0 \\ n \cdot \mathrm{fat}(n-1), & \text{otherwise.} \end{cases}$$

The expressiveness gained from recursion is essentially the same as the one gained from `goto` in imperative languages. That is, the control flow given by function composition together with recursion allows one to do anything imaginable with the programming language. However, all this expressiveness comes with its caveats. It is too easy to write a functional spaghetti code if recursion is used indiscriminately. Because of all its power, recursive code lacks in safety. It would even be fair to say that, like `goto`, it is too unstructured.

The idea of structured control flow really caught on in the imperative world. One hardly sees a wild `goto` in the middle of a modern piece of code. In fact, many languages don't even allow it. In the functional world, on the other side,

the trendy for taming recursion never really caught on. Despite many functional languages organizing their data using types, control is still done using crude recursion.

Recursion schemes are ways to organize and structure different kinds of recursive functions. Instead of writing a recursive function in terms of itself, we define higher order functions that receive an ordinary function as argument, do some recursive magic on it, and return a recursive analogue of that function as output. It is similar to how a `while` loop takes a boolean statement and a block of code, and turns them into a repeating block of code. If it seems too confusing, just keep on. What I mean in here will become clearer after we construct catamorphisms.

Before we end this motivation and proceed to the actual construction of recursion schemes, there is an intuition that I believe useful to have in mind. In imperative programming, there is a close relationship between how we structure data and how we structure the control flow. Working with arrays almost asks the programmer to design programs with `for` loops. Similarly, it is natural to deal with `union` types using `switch` statements (also called `case` or `cond` in some languages). Recursion schemes will arise as an analogous to this idea in the functional programming setting. So far so good for motivation, let's dive into some math.

Algebraic Data Types

The simplest form to write a type is by enumerating its elements such as

$$\texttt{Bool} \coloneqq \mathrm{true} \mid \mathrm{false}\,.$$

This way, we define the type of boolean values. That is, a type with exactly two terms called true and false. In general any finite type can be written just by enumerating its elements. As another example, there is the type of musical notes

$$\texttt{Notes} \coloneqq \texttt{do} \mid \texttt{re} \mid \texttt{mi} \mid \texttt{fa} \mid \texttt{sol} \mid \texttt{la} \mid \texttt{si}.$$

Nevertheless, in practice we also want to deal with types that are more complex than finite collections. The solution to this is assuming that the programming language comes with some built-in types such as integers, characters, and

floating-point numbers, together with structures that allow us to compose these types.

One common compound type consists of a structure capable of storing data of more than one type at the same time. As an example, let's say we are in a spacial war against many alien species. Your job is to catalogue how many battleships each army has. One way to store this data is with a type containing a string for the species which the army belongs together with a natural number for how many battleships they have. We will write this type as

$$\mathtt{Army} := \mathrm{ships}\ \mathtt{String} \times \mathbb{N}.$$

Here, $\mathrm{ships} \colon \mathtt{String} \times \mathbb{N} \to \mathtt{Army}$ is called a *constructor* for the type $\mathtt{Army}$. Constructors are (possibly multivariate) functions that receive terms as arguments and return a term of a compound type.

After you finish your catalogue, orders arrive for you to develop a new laser cannon to be used in the war. This weapon should scan the sky to find the enemy armies' positions (modeled in a type $\mathtt{Pos}$ and shoot them. But beware! There are also allied bases around, encapsulated in the type $\mathtt{Base}$. Since friendly fire is really far from the ideal, our target type should have two different constructors, one for allies and another for enemies:

$$\begin{aligned}\mathtt{Target} := {} & \mathrm{ally}\ \mathtt{Position} \times \mathtt{Base} \\ & \mid \mathrm{enemy}\ \mathtt{Position} \times \mathtt{Army}.\end{aligned}$$

In here we extended our notation for enumerating types to also accept constructors. Different constructors always produce different terms of the compound type, thus, we may view the functions $\mathrm{ally} \colon \mathtt{Position} \times \mathtt{Base} \to \mathtt{Target}$ and $\mathrm{enemy} \colon \mathtt{Position} \times \mathtt{Army} \to \mathtt{Target}$ as tags representing from which type our $\mathtt{Target}$ was constructed. This works as if we are storing an element of type $\mathtt{Army}$ together with a tag enemy on the type $\mathtt{Target}$. So the definition of target is saying that its terms are of the form $\mathrm{ally}(p, x)$ or $\mathrm{enemy}(p, x)$, just like we previously enumerated the terms of finite types.

This point of view allows us to define functions on compound types by enumerating what it does on the different constructors, a method called *pattern matching*. For example, the function $\mathrm{not} \colon \mathtt{Bool} \to \mathtt{Bool}$ is defined as

$$\begin{aligned}\mathrm{not}\,\mathrm{true} &= \mathrm{false} \\ \mathrm{not}\,\mathrm{false} &= \mathrm{true}\,.\end{aligned}$$

While our ally-aware cannon shooting function may be defined as something like

$$\begin{aligned}\mathrm{shoot}(\mathrm{enemy}(p, y)) &= \texttt{laser\_blast}(p)\\ \mathrm{shoot}(\mathrm{ally}(p, x)) &= \texttt{wave\_your\_hand}(p).\end{aligned}$$

Taking advantage of the type system via pattern matching is a nice way to make it obvious that our code does what it should. In this case, the types don't allow us to obliterate our friends.

Although these compound types are nice to organize our data, the true power of types comes from taking advantage of two other constructions: *function types* and *fixed point types*. The function type between two types A and B, represents all the functions receiving an input of type A and returning an argument of type B. To mimic the usual notation for type signatures, this function type is denoted as $A \to B$. A type system with function types allows us to define higher-order functions. For example, given a function $\phi\colon A \to B$, the composition operator K_ϕ defined as

$$K_\phi f = f \circ \phi$$

has type signature $K_\phi\colon (B \to C) \to (A \to C)$. As you can see, it takes a function and turns it into another one. This was a simple example but be sure that many more will soon come, since higher-order functions are at the heart of recursion schemes.

Now we arrive at the last kind of compound type we are going to discuss today: fixed point types. These are the star of today's show, so pay attention. We may define a compound data type that is parameterized by some variable (a function between types, if you prefer) such as

$$\texttt{maybe}A \coloneqq \mathrm{nothing} \mid \mathrm{just}\ A.$$

Given a type A, a term of type $\texttt{maybe}A$ is either $\mathrm{nothing}$ or the constructor just applied to a term of type A. Thus, **maybe** receives a type and augments it with a special point[2].

Given a type operator F, a fixed point of F is a type X satisfying

$$X \simeq FX.$$

In here, we use the isomorphism sign $\simeq$ instead of equality because there is some boilerplate involved regarding tags. Although this is a simple construction, the concept is extremely powerful, being directly connected to recursion.

As an example let's explore the fixed point of **maybe**. It is a type N satisfying

$$N \simeq \mathbf{maybe}N = \mathrm{nothing} \mid \mathrm{just}\, N.$$

This means that any term of N is either $\mathrm{nothing}$ or just a term of N. Since we know $\mathrm{nothing}$, we can construct a new term $\mathrm{just}(\mathrm{nothing})$, then $\mathrm{just}(\mathrm{just}(\mathrm{nothing}))$, and proceed successively in this way. Thus for any natural number n, applying just to $\mathrm{nothing}$ n times defines a unique term of N. Moreover, the definition of N says that all of its terms are of this form, meaning that N is isomorphic to the natural numbers[3].

If you find this last result strange, remember that the natural numbers are inductively defined as being either zero or the successor of another natural number. Using our notation for types, this is equivalent to saying

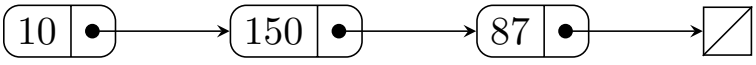
$$\mathbb{N} = \mathrm{zero} \mid \mathrm{succ}\, \mathbb{N}.$$

If we alter the tag names, this tells us that $\mathbb{N}$ is a fixed point of **maybe** as expected.

An example with lists

The natural numbers are surely the most famous example of an inductive type. Nevertheless, almost no one thing of them like that while programming. Treating 3 as $\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{zero})))$ would be cumbersome, to say the least. Any language comes with a (possibly signed) integer type already bundled with the usual arithmetic operations defined for it. Thus, let's discuss a little bit about another inductive type that is also pretty famous but has more of a inductive data structure flavour to it: the *linked list*.

Let A be your favorite type. Intuitively, a list over A is a finite-length sequence of elements of A. The simplest case possible is an empty list. To represent a non-empty list, we notice that any non-empty list with n elements may be "factorized" into its first element and another list with the remaining $n-1$ elements.



Thus, by defining a type operator P as

$$PX \coloneqq \mathrm{nil} \mid \mathrm{cons}\ A \times X,$$

the previous discussion shows that the type of lists over A, hereby denoted $L(A)$, is a fixed point of P,

$$L(A) \simeq \mathrm{nil} \mid \mathrm{cons}\ A \times L(A).$$

Here, the constructor nil takes the role of the empty list while the constructor cons represents a pair containing an element of A and another list.

This inductive definition of lists allows us to recursively define operations over it. For example, let's construct a function that sums all the elements in a list of integers. Its signature is $\mathrm{sum}\colon L(\mathbb{Z}) \to \mathbb{Z}$. If the list is empty, the sum returns zero. Otherwise, it takes the first element and adds it to the sum of what remains,

$$\begin{aligned} \mathrm{sum}(\mathrm{nil}) &= 0, \\ \mathrm{sum}(\mathrm{cons}(x, l)) &= x + \mathrm{sum}(l). \end{aligned}$$

This last definition conceals an extremely useful pattern. It is the programming analogue of a proof by induction over the list. The empty list is the base case, which we set to zero. On a list of length $n > 0$, we assume that we've already solved the computation for the sublist of length $n - 1$ and them add it to the remaining element. Just like an inductive proof, see?

As it stands, you would be right to guess that constructing functions in this way is so pervasive in programming as proofs by induction are in mathematics. A simple variation of sum would be a function that multiplies a list of real numbers, $\mathrm{prod}\colon L(\mathbb{R}) \to \mathbb{R}$. To construct it, all we need to do is take the definition of sum, replace 0 by 1, replace addition by multiplication, and it is all done!

$$\begin{aligned} \mathrm{prod}(\mathrm{nil}) &= 1, \\ \mathrm{prod}(\mathrm{cons}(x, l)) &= x \cdot \mathrm{prod}(l). \end{aligned}$$

As expected, this pattern appears every time we want to somehow combine the elements of a list into a single value. It is so common that people have abstracted it on a function called reduce[4]. Its type signature is

$$\mathrm{reduce}\colon B \times (A \times B \to B) \to (L(A) \to B).$$

Pretty scary, right? Let's break it down to see how it is just good ol' induction. When reducing a list of type $L(A)$ into a value of type B, there are two situations we may encounter. In the base case the list is empty, and we must specify

a value of type B to be returned. In the other steps, we consider that we already know the solution for the sublist of length $n-1$, which yields a value of type B, and then need a rule to combine this value of the remaining term of type A into another term of type B. Thus, $\mathbf{reduce}$ is an instance of what we call a *higher-order function*. It is a machine that takes an initial value and a binary function and outputs another function, now defined on lists. We call it higher-order because it doesn't work with ordinary data, no, it transforms simple functions into recursive ones! Considering what it does, its definition is actually rather simple. The result $h = \mathbf{reduce}(v, g)$ is the function that does

$$\begin{aligned} h(\mathbf{nil}) &= v, \\ h(\mathbf{cons}(x, l)) &= g(x, h(l)). \end{aligned}$$

Take a moment to absorb this definition, there really is a lot encapsulated on this. First substitute v by 0 and g by $+$ to see that it becomes $\mathbf{sum}$. Then, substitute v by 1 and g by $\cdot$ to see that it becomes $\mathbf{prod}$. Finally, congratulate yourself because you just understood your first recursion scheme!

If the way $\mathbf{reduce}$ was introduced made you think that it is only used to collapse a list into a primitive type, you should know that it is much more ubiquitous than that. Two of the most used list-manipulating functions in functional programming are $\mathbf{map}$ and $\mathbf{filter}$. And, guess what, both are implementable in terms of $\mathbf{reduce}$. The first, $\mathbf{map}$, takes a function $f\colon A \to B$ and turns it into another function that applies f elementwisely to a list. Its type signature for $\mathbf{map}$ is therefore

$$\mathbf{map}\colon (A \to B) \to (L(A) \to L(B)).$$

On the empty list, $\mathbf{map}(f)$ does nothing since it has no elements. On a $\mathbf{cons}$ node, it should apply f to the element stored on it and then proceed to apply f to the list's tail. Thus, the definition of $\mathbf{map}$ in terms of $\mathbf{reduce}$ is

$$\begin{aligned} g(x, l) &= \mathbf{cons}(f(x), l), \\ \mathbf{map}(f) &= \mathbf{reduce}(\mathbf{nil}, g). \end{aligned}$$

If it is hard to wrap your head around this last definition, try doing it step by step with a simple function such as x^2 and a small list such as $(1, 2, 3, 4)$. I promise you it will be enlightening.

The filter function takes a predicate, represented as $p\colon A \to \mathtt{Bool}$, and outputs a function that filters a list. That is removes all elements of the list for which p is false. Its type signature is thus

$$\mathrm{filter}\colon (A \to \mathtt{Bool}) \to (L(A) \to L(A)).$$

Since the empty list has no elements, there is nothing to do in this case. In a $\mathrm{cons}(x, l)$, we should return it exactly if $p(x)$ is true and return just l otherwise. We can assembly this in terms of reduce as

$$\begin{aligned} h(x, l) &= \begin{cases} \mathrm{cons}(x, l), & \text{if } p(x) = \mathrm{true} \\ l, & \text{if } p(x) = \mathrm{false} \end{cases} \\ \mathrm{reduce}(f) &= \mathrm{reduce}(\mathrm{nil}, h). \end{aligned}$$

I hope you liked these examples because the next step in our journey is generalizing reduce to any inductive data type! To achieve this, we must pass through the land of category theory.

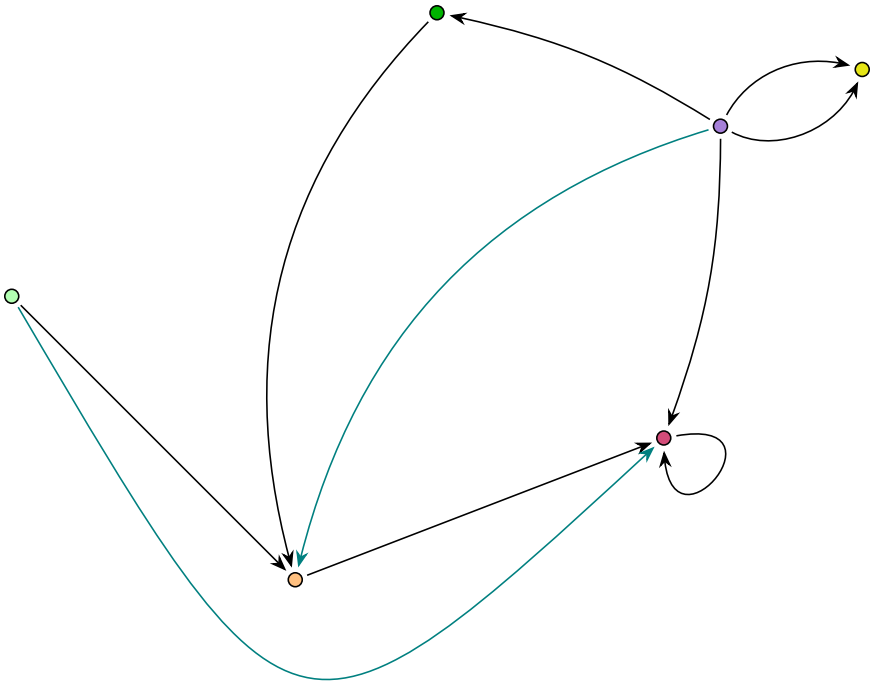
A walk through the land of categories

While the previous sections dealt more with programming, this one leans more to the mathematical side. But be assured that the fruits of all this abstraction will pay well.

A nice thing about types and functions between them is that they form a *category*.[5] In fact, with all the structure we defined, they have the much more powerful structure of a Cartesian closed category. But we aren't going to use this last part today. If you happen to not know what a category is, I'm afraid I can't properly introduce them in here.[6] But I will go through the basics for the sake of completeness.

A category $\mathcal{C}$ may be imagined as a special kind of graph. There is a collection of vertices, called its *objects* and between each pair of vertices there is a collection of *directed edges*, called *arrows* or *morphisms*. The A and B are objects, the notation for an arrow r from A to B is the same as for functions, $r\colon A \to B$. Soon we will see why. What differentiates a category from a simple directed graph is that we have a notion of *composition* between the arrows. That is, if

$f\colon A \to B$ and $g\colon B \to C$, there always exists another arrow $g \circ f\colon A \to C$. You may think of arrows as representing paths and $g \circ f$ as the arrow path going through f and then through g.



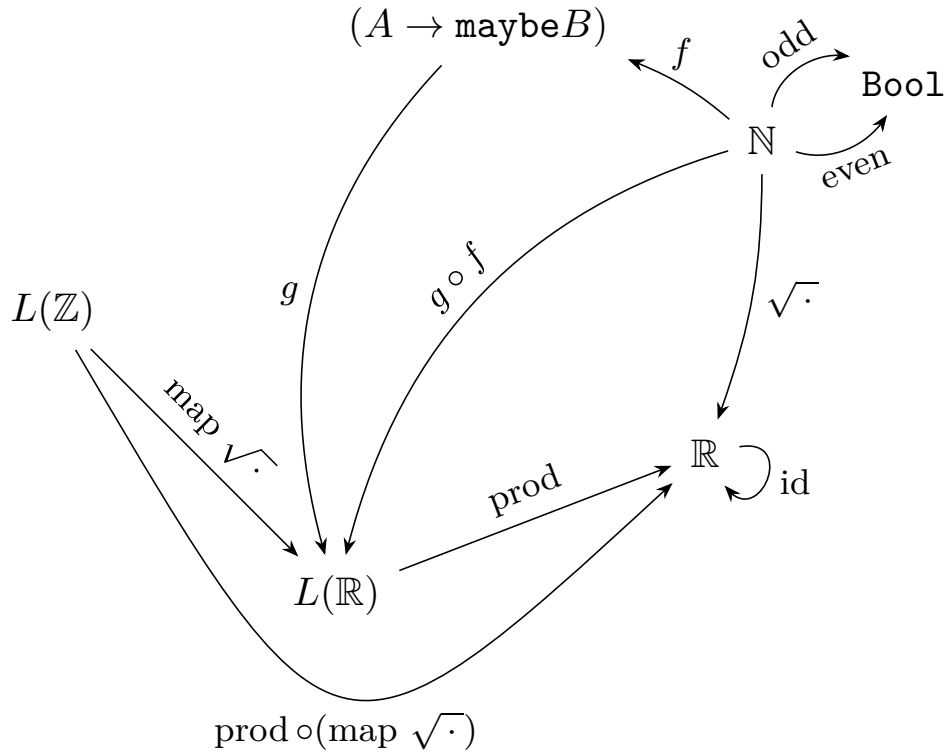
To call $\mathcal{C}$ a category, composition must also satisfy two laws. First, it must be associative. That is, for any composable triple of arrows,

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

Second, for each object X, a special arrow $\mathrm{id}_X\colon X \to X$ must exist, called the *identity* for X, with the special property that for any $f\colon A \to B$,

$$f \circ \mathrm{id}_A = \mathrm{id}_B \circ f = f.$$

Now, back to types. Probably the most striking feature of functional programming is that it is stateless. That is, we don't think of functions as a series of procedures mutating some registers, we think of them as mathematical functions and as such, we must have that $f(x)$ returns the exact same value no matter when we execute it. For us, the main consequence of this is that function composition becomes associative. Thus, there is a category **`Types`** whose objects are types and arrows are functions between types. Composition is the usual for functions and the identities are the identity functions $\mathrm{id}_A(x) = x$.



Category theory is not only concerned with categories but also with transformations between them. As in most of mathematics, we study the transformations that preserve the structure of composition and identity, called *functors*. More precisely, a functor from $\mathcal{C}$ to $\mathcal{D}$ is comprised of a function taking objects of $\mathcal{C}$ to objects of $\mathcal{D}$ and a function taking arrows of $\mathcal{C}$ to arrows of $\mathcal{D}$ (normally both denoted by F) such that composition and identities are preserved,

$$\begin{aligned} F(g \circ f) &= F(g) \circ F(f) \\ F(\text{id}) &= \text{id}\,. \end{aligned}$$

As an example, the type operator **maybe** is a functor from **Types** to itself when coupled with the operation on arrows

$$\begin{aligned} \texttt{maybe}(f)(\text{nothing}) &= \text{nothing} \\ \texttt{maybe}(f)(\text{just}(x)) &= \text{just}(f(x)). \end{aligned}$$

More generally, if a type operator G is defined using only products, coproducts, other functors, and function types with the variable appearing only on the right-hand side there is a canonical way to turn it into a functor from **Types** to itself. Some compilers, such as Haskell's GHC can even do that automatically for us. Instead of going through the constructions gory details, I think doing an example is better to elucidate how the method. Say we have a type operator

$$
\begin{aligned}
GX = &\operatorname{pig} A \\
&\mid \operatorname{yak} A \times X \times (\mathtt{maybe} X) \\
&\mid \operatorname{cow} B \times (C \to X).
\end{aligned}
$$

To turn it into a functor, we define an action on functions as

$$
\begin{aligned}
(G\,f)(\operatorname{pig}(a)) &= \operatorname{pig}(a) \\
(G\,f)(\operatorname{yak}(a, x, m)) &= \operatorname{yak}(a, f(x), \mathtt{maybe}(f)(m)) \\
(G\,f)(\operatorname{cow}(b, g)) &= \operatorname{cow}(c, f \circ g)
\end{aligned}
$$

Essentially, what $G\,f$ does is to unwrap each constructor and pass f to each of its arguments according to one of the following rules:

1. If the term if of type X, apply f to it.
2. If the term is of type $(A \to X)$, compose f with it.
3. If the term is of type FX for some functor F, apply $F\,f$ to it.
4. Otherwise, just return the term itself.

Functors and their fixed points

Now we know that both the natural numbers and lists aren't just the fixed points of ordinary type operators but of functors from **`Types`** to itself. But is there anything special in this? As you may expect, the answer is *yes*. If a type is the fixed point of a functor, then we can define recursion schemes for it that work very much like structural induction. Since this construction works on any category, it won't bite to do that in full generality. We begin by fixing a category $\mathcal{C}$ and a functor F from $\mathcal{C}$ to itself. From F, we will construct an auxiliary category, called the category of *F-algebras*.

A F-algebra is an object X of $\mathcal{C}$ together with an arrow $f\colon FX \to X$. Morphisms between F-algebras $f\colon FX \to X$ and $g\colon FY \to Y$ are given by an arrow $h\colon X \to Y$ such that the following diagram commutes

$$
\begin{array}{ccc}
FX & \xrightarrow{Fh} & FY \\
{\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\
X & \xrightarrow[h]{} & Y
\end{array}
$$

You can check that this definition turns F-algebras into a category where the identity for $f\colon FX \to X$ is given by id_X itself. On this category, we are interested in a special kind of object called an *initial F-algebra*. That is a F-algebra $\mathbf{in}$ with the special property that for any other F-algebra f there is a *unique* morphism going from $\mathbf{in}$ to f. Ok, time for some action. We haven't proved any theorem in this post yet.

> **Theorem.**
> Any initial F-algebra $\mathbf{in}\colon FI \to I$ is an isomorphism in $\mathcal{C}$.

Before we prove this theorem, notice that it implies that I is a fixed point of F! It's not every fixed point that defines an initial algebra, but the just the ones that are "smallest" in a certain sense.[7] Nevertheless, finite length inductive data structures are generally initial.

Proof. First, notice that if $\mathbf{in}\colon FI \to I$ is a F-algebra, so is $F\,\mathbf{in}\colon F(FI) \to FI$. Since $\mathbf{in}$ is initial, there is a unique arrow $g\colon I \to FI$ making the following diagram commute

$$\begin{array}{ccc} FI & \xrightarrow{Fg} & F(FI) \\ {\scriptstyle f}\big\downarrow & & \big\downarrow{\scriptstyle F\,\mathbf{in}} \\ I & \xrightarrow[g]{} & Y \end{array}$$

Since $\mathbf{in}$ itself may be viewed as a morphism between the F-algebras $F\,\mathbf{in}$ and $\mathbf{in}$, we get that their composition is a morphism from $\mathbf{in}$ to itself represented by the following diagram, where all paths still commute

$$\begin{array}{ccccc} FI & \xrightarrow{Fg} & F(FI) & \xrightarrow{F\,\mathbf{in}} & FI \\ {\scriptstyle \mathbf{in}}\big\downarrow & & \big\downarrow{\scriptstyle F\,\mathbf{in}} & & \big\downarrow{\scriptstyle \mathbf{in}} \\ I & \xrightarrow[g]{} & FI & \xrightarrow[\mathbf{in}]{} & I \end{array}$$

Since $\mathbf{in}$ is initial, the unique arrows going from it to itself is the identity. Thus, we must have $g \circ \mathbf{in} = \mathrm{id}$. Moreover, from the definition of functor and the previous diagram,

$$\mathbf{in} \circ g = (Fg) \circ (F\,\mathbf{in}) = F(g \circ \mathbf{in}) = F(\mathrm{id}) = \mathrm{id}\,.$$

Therefore, g is an inverse to $\mathbf{in}$, concluding the proof. □

Using catastrophes in your favor

We finally have the necessary tools to define our first recursion scheme in all its glory and generality. Consider, just as in the previous section, a functor F from $\mathbf{Types}$ to itself and call its initial algebra $\mathbf{in} : FA \to A$. Given a F-algebra $f : FX \to X$, its *catamorphism*, which we will denote by $\mathbf{cata}\, f$, is the unique arrow from A to X given by the initially of $\mathbf{in}$.

Before proceeding, I must allow myself a little rant: sometimes mathematicians are just terrible name-givers. Like, what in hell is a catamorphism? What kind of intuition should it elicit? Well... as a matter of fact the name makes a lot of sense if you happen to be speaking in ancient Greek. Since, despite the arcane name, the catamorphism is a super cool concept, I think it deserves that we do a little etymological break to explain where its name comes from.

The word catamorphism has two parts, *cata-* + *morphism*. The later comes from the Greek 'μορφή' and means "form" or "shape". Its use is common in category theory and stems from the fact that usually, the arrows in a category are structure (or shape) preserving functions. For example, in the category of F-algebras, the morphisms are functions that commute with the algebras, thus "preserving" its application. The prefix "cata-" comes from the Greek 'κατά' and means something like a "downward motion". It is the same prefix as in "cataclysm" or "catastrophe" and this is the intuition we shall keep in mind. If you think of an inductive data structure as a great tower piercing the sky, the catamorphism is a divine smite that, in a catastrophic manner, collapses the tower into a single value.[8]

Ok, back to mathematics.

As you may have imagined, the catamorphism is a generalization of the $\mathbf{reduce}$ operator. The difference is that while $\mathbf{reduce}$ only works for lists, $\mathbf{cata}$ can collapse the initial algebra of any functor. But, in all this generality, how do we actually compute a catamorphism? Let's start by taking a look at its type signature:

$$\mathbf{cata} : (FX \to X) \to (A \to X).$$

It is a higher-order operator that takes a function from FX to X and turns it into another function, now from the fixed point A to X. Intuitively, what it encapsulates is that if we know how to collapse one step of a data structure, then we can use structural induction to collapse the entire structure.

To properly calculate $\mathbf{cata}$, let's take a look at the commutative diagram defining it,

$$\begin{array}{ccc} FA & \xrightarrow{F(\operatorname{cata} f)} & FX \\ {\scriptstyle \text{in}^{-1}}\uparrow & & \downarrow{\scriptstyle f} \\ A, & \xrightarrow[\operatorname{cata} f]{} & X \end{array}$$

In this diagram, there are two ways to go from A to X. From its commutativity, we know that they are equal. Hence, $\mathbf{cata}$ must satisfy

$$\mathbf{cata}\, f = f \circ F(\mathbf{cata}\, f) \circ \mathbf{in}^{-1}\,.$$

So, that's a pretty concise formula. Let's unpack it with some examples before trying to give a full explanation.

The first inductive data type we encountered where the natural numbers, defined as the least fixed point of

$$FX = \mathbf{zero} \mid \mathbf{succ}\ X.$$

For it, the initial algebra and its inverse are defined as

$$\begin{aligned} \mathbf{in}(\mathbf{zero}) &= 0, & \mathbf{in}^{-1}(0) &= \mathbf{zero}, \\ \mathbf{in}(\mathbf{succ}\, n) &= n + 1, & \mathbf{in}^{-1}(n) &= \mathbf{succ}(n-1). \end{aligned}$$

If these definitions seem too obvious, remember that the $\mathbb{N}$ are the natural numbers as we think of them everyday while $\mathbf{zero}$ and $\mathbf{succ}$ are the formal constructors of the functor F. Keeping track of this boilerplate is essential.

As our first example, let's construct the function $\mathbf{exp} \colon \mathbb{N} \to \mathbb{R}$, which takes a natural n to the real number e^n, as a catamorphism. Exponentiation is recursively defined by the equations

$$\begin{aligned} e^0 &= 1 \\ e^{n+1} &= e \cdot e^n. \end{aligned}$$

In terms of an algebra $f : F\mathbb{R} \to \mathbb{R}$, this means that on the constructor $\mathbf{zero}$ we must return 1 while in each $\mathbf{succ}$ step, we must return e times the already accumulated value. Then, $\mathbf{exp} = \mathbf{cata}\, f$, where

$$\begin{aligned} f(\mathbf{zero}) &= 1 \\ f(\mathbf{succ}\, x) &= e \cdot x. \end{aligned}$$

Let's use the catamorphism formula to unwrap the calculations of $\mathbf{exp}$. First, in^{-1} writes a number as the $\mathbf{succ}$ of its predecessor. Then, the way we defined the functor action of a data type means that $F(\mathbf{cata}\, f)$ unwraps $\mathbf{succ}\, n$ to apply $\mathbf{cata}\, f$ to the natural number n stored inside it and then reapplies the constructor, leaving us with $\mathbf{succ}((\mathbf{cata}\, f)(n))$. This process recursively continues until we encounter a $\mathbf{zero}$. But this time $F(\mathbf{cata}\, f)$ *does not* reapplies $\mathbf{cata}\, f$; instead, it returns $\mathbf{zero}$ as is. At this point, our call stack is consists of the function f applied exactly n times to the constructor $\mathbf{succ}$ applied exactly n times to $\mathbf{zero}$. For example, let's take a look at the traceback of calling $\mathbf{exp}(2)$:

$$\begin{aligned} \mathbf{exp}(2) &= (f \circ F(\mathbf{cata}\, f) \circ \mathbf{in}^{-1})(2) \\ &= f(F(\mathbf{cata}\, f)(\mathbf{in}^{-1}(2))) \\ &= f(F(\mathbf{cata}\, f)(\mathbf{succ}\, 1)) \\ &= f(\mathbf{succ}(\mathbf{cata}\, f(1))) \\ &= f(\mathbf{succ}(f(F(\mathbf{cata}\, f)(\mathbf{in}^{-1}(1))))) \\ &= f(\mathbf{succ}(f(F(\mathbf{cata}\, f)(\mathbf{succ}\, 0)))) \\ &= f(\mathbf{succ}(f(\mathbf{succ}(\mathbf{cata}\, f(0))))) \\ &= f(\mathbf{succ}(f(\mathbf{succ}(f(F(\mathbf{cata}\, f)(\mathbf{in}^{-1}(0))))))) \\ &= f(\mathbf{succ}(f(\mathbf{succ}(f(F(\mathbf{cata}\, f)(\mathbf{zero})))))) \\ &= f(\mathbf{succ}(f(\mathbf{succ}(f(\mathbf{zero}))))) \\ &= f(\mathbf{succ}(f(\mathbf{succ}\, 1))) \\ &= f(\mathbf{succ}(e \cdot 1)) \\ &= e \cdot (e \cdot 1). \end{aligned}$$

Ok, these were a lot of parentheses. However, if you actually followed that mess above, the catamorphism's pattern should be clear by now.

As a final example of catamorphism, let's write a little calculator that supports addition, multiplication and exponentiation by a natural. A calculator should take an arithmetic expression and return a real number. We define an expres-

sion recursively. It may be a real number, the sum of two other expressions, the multiplication of two expressions, or an expression raised to a natural exponent. This is represented by a type $\mathbf{Expr}$ which is the least fixed point of the functor

$$\begin{aligned} F\,X = &\operatorname{const}\,\mathbb{R} \\ &\mid \operatorname{add}\,X \times X \\ &\mid \operatorname{mult}\,X \times X \\ &\mid \operatorname{pow}\,X \times \mathbb{N}. \end{aligned}$$

To evaluate an expression, we need an appropriate F-algebra $f\colon F\mathbb{R} \to \mathbb{R}$. As with natural numbers, the idea in here is to treat the constructor $\mathbf{const}$ where X don't appear as a base case and to think of the others constructors as storing an already solved problem on the X. With this in mind, the evaluator F-algebra is

$$\begin{aligned} f(\operatorname{const}(a)) &= a \\ f(\operatorname{add}(x, y)) &= x + y \\ f(\operatorname{mult}(x, y)) &= x \cdot y \\ f(\operatorname{pow}(x, n)) &= x^n. \end{aligned}$$

And the evaluator $\operatorname{eval}\colon \mathbf{Expr} \to \mathbb{R}$ is just $\operatorname{cata} f$.

Another application of a catamorphism is if instead of evaluating the expression, we want to print it as a string. Let's say we have a method str that converts numbers to strings and an operator $\diamond$ that concatenates two strings. Erring on the side of too many parentheses, a candidate F-algebra is

$$\begin{aligned} g(\operatorname{const}(a)) &= \operatorname{str}(a) \\ g(\operatorname{add}(x, y)) &= x \diamond \texttt{"+"} \diamond y \\ g(\operatorname{mult}(x, y)) &= \texttt{"("} \diamond x \diamond \texttt{")*("} \diamond y \diamond \texttt{")"} \\ g(\operatorname{pow}(x, n)) &= \texttt{"("} \diamond x \diamond \texttt{")\^{}"} \diamond \operatorname{str}(n). \end{aligned}$$

As you probably already expect, the function $\operatorname{show}\colon \mathbf{Expr} \to \mathbf{String}$ that converts an expression to a string is just $\operatorname{cata} g$.

Least fixed points are recursive data structures. The catamorphism abstracts the process of transforming these structures into another type via structural induction. All recursion occurs inside the formula for cata. If you just use it as a black box that turns functions $FX \to X$ into functions $A \to X$ (where A is the

fixed point), you will never actually see the recursion's autoreference. The operator $\mathbf{cata}$ abstracts recursion just like a `for` loop abstracts the `goto` away. It is still there but following a fixed structure.

Since $\mathbf{cata}$ is a more rigid structure than general recursion, it is easier to reason about it. For example, one of the strongest properties of catamorphisms is that if the recursive data structure is finite and f is total function, then $\mathbf{cata}\ f$ is also guaranteed to eventually stop; As a lot of applications use finite data, this facilitates a lot the act of debugging a program.

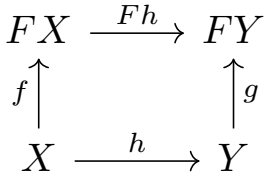
Taking your functions to the gym

Collapsing a data structure into a value has many use cases but is not the end of the story. Sometimes we want to do the exact opposite: take a value as a seed to construct a list or another structure from it. This notion is dual to the catamorphism and, of course, there is also a recursion scheme to encapsulate it: the *anamorphism*.[9]

Again we have a pretty arcane name in our hands. Let's take a look at its etymology in order to clarify things. The prefix *ana-* comes from the Greek 'ἀνα' and generally means "upward motion" but is also used to mean "strengthening" or "spreading all over". It is the same prefix of *analogy* and of *anabolic steroids*. In both these words, the prefix means that we are building something up.

In order to describe the catamorphism in all its generality and yet with a concise formula, we had to dive into some categorical abstract nonsense. As you might expect, the same applies to the anamorphism. Fortunately, there is a tool in category theory called duality. If we simply reverse the arrows in a theorem, its conclusion still holds but also with some arrows reversed.

Given a category $\mathcal{C}$ and a functor $F : \mathcal{C} \to \mathcal{C}$, we define a F-coalgebra to an object X of $\mathcal{C}$ together with an arrow $f : X \to FX$. The F-coalgebras form a category where the morphisms between $f : X \to FX$ and $g : Y \to FY$ are arrows $h : X \to Y$ such that the diagram below commutes.

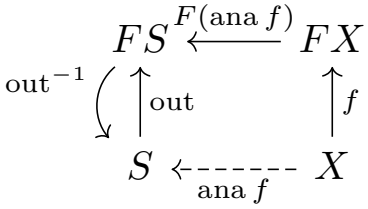


The dual notion to an initial object $F-$algebra is a *terminal* F-coalgebra. Namely, an algebra $\mathbf{out}\colon S \to FS$ such that there is a unique morphism from any other F-coalgebra to $\mathbf{out}$. As you probably already expect, terminal coalgebras are always isomorphisms. The proof is essentially the same as the one for initial F-algebras. The only difference being some arrows reversed.

> **Theorem.**
> Any terminal F-coalgebra $\mathbf{out}\colon C \to FS$ is an isomorphism.

The most direct consequence of this theorem is that S must be a fixed point of F. Since there is an arrow from every other coalgebra to S, it is called the *greatest fixed point* of F. For this presentation, there is no need to discriminate between least and greatest fixed points. Specially because there are languages such as Haskell where they are the same. Just keep in mind that when working with coalgebras, there is no guarantee that functions must eventually stop. We are no longer on the land of finite data structures but in the much greater land of possibly infinite data structures.

Given a F-coalgebra $f\colon X \to FX$, its anamorphism $\mathbf{ana}\, f$ is defined as the unique arrow from f to the terminal object



Thus, the type signature of $\mathbf{ana}$ viewed as a higher-order operator is

$$\mathbf{ana}\colon (X \to FX) \to (X \to S).$$

It turns a coalgebra, which we may view as one construction step, into a function that constructs an entire inductive data structure. If we pay attention to the commutative diagram above, we can see that there is a concise recursive formula defining $\mathbf{ana}$,

$$\operatorname{ana} f = \operatorname{out}^{-1} \circ F(\operatorname{ana} f) \circ f.$$

This is the same formula we had for cata, but the order of composition reversed! This reversion, however, means that we are building structures instead of collapsing them.

Let's do some examples with lists. Recall that the type $L(A)$ of lists with elements of type A is a fixed point of the functor

$$PX = \operatorname{nil} \mid \operatorname{cons}\ A \times X.$$

Using the usual notation $[a, b, c, \ldots]$ to represent lists, the F-coalgebra out is defined as

$$\begin{aligned} \operatorname{out}([\,]) &= \operatorname{nil} \\ \operatorname{out}([a, b, c, \ldots]) &= \operatorname{cons}(a, [b, c, \ldots]). \end{aligned}$$

One of the simplest list anamorphisms is a function that receives a natural number n and returns a decreasing list from n to 1. It is defined as $\operatorname{ana} g$, where g is the coalgebra

$$g(n) = \begin{cases} \operatorname{nil}, & n < 1 \\ \operatorname{cons}(n, n-1), & \text{otherwise.} \end{cases}$$

In practice, it is most common to use an increasing list, however. The induction on this one is a little trickier but nothing an anamorphism can't handle. To make it more fun, let's construct a function $\operatorname{range} \colon \mathbb{Z} \times \mathbb{Z} \to L(\mathbb{Z})$ taking a pair of integers to the closed integer interval between them. We want $\operatorname{range}(-2, 1) = [-2, -1, 0, 1]$ and $\operatorname{range}(7, 5) = [\,]$, for example. To achieve this, we will need a family of coalgebras

$$g_b(n) = \begin{cases} \operatorname{nil}, & n > b \\ \operatorname{cons}(n, n+1), & \text{otherwise.} \end{cases}$$

Then, our range function is the anamorphism

$$\operatorname{range}(a, b) = (\operatorname{ana} g_b)(a).$$

If you ever dealt with plotting libraries, you are probably familiar with a `linspace` method. It receives two real arguments a, b, and a natural argument n and returns a list of n points uniformly distributed between a and b. The construction of such a function

$$\mathrm{linspace}\colon \mathbb{R} \times \mathbb{R} \times \mathbb{N} \to L(\mathbb{R})$$

is a simple variation on the $\mathbf{range}$ we defined earlier. We start with a family of coalgebras

$$g(b,n)(x) = \begin{cases} \mathrm{nil}, & x > b \\ \mathrm{cons}(x, x + \frac{b-a}{n}), & \text{otherwise}, \end{cases}$$

and define it to be the anamorphism

$$\mathrm{linspace}(a,b,n) = (\mathrm{ana}\, g(b,n))(a).$$

To end this section, let's turn to an example from number theory. We will construct the sieve of Eratosthenes.[10] This algorithm receives a natural number n and returns a list of all primes below n. The idea is to start with a list of numbers between 2 and n, and to recursively refine it eliminating all multiples of a given prime. Let $\mathrm{test}(p)$ be the predicate that tests if a number is not divisible by p. You can implement it by testing if the remainder of the input by p equals zero, for example. This refinement process is encapsulated by the function $\mathbf{sieve} = \mathbf{ana}\, g$ where g is the coalgebra

$$\begin{aligned} \mathrm{g}([\,]) &= \mathrm{nil} \\ \mathrm{g}([p, x, \ldots]) &= \mathrm{cons}(p, (\mathrm{filter}(\mathrm{test}(p))([x, \ldots]))). \end{aligned}$$

And our prime-listing function is the composition

$$\mathrm{era}(n) = \mathrm{sieve}(\mathrm{range}(2,n)).$$

This process works because after each filtering, the list's first element cannot be divisible by any number below it. Thus it must be prime. Notice that this function eventually stop because as soon as we reach an empty list, the algorithm returns a nil.

Finally, to show the power of anamorphisms coupled with lazy evaluation, let's use $\mathbf{sieve}$ to compute a list of all primes. We begin by writing a list l of all natural numbers starting at 2,

$$\begin{aligned} h(x) &= \mathrm{cons}(x, x+1) \\ l &= (\mathrm{ana}\, h)(2). \end{aligned}$$

This list must be infinite because, since h never returns $\mathbf{nil}$, the anamorphism recurses forever. Unsurprisingly, the list of all primes is defined as $\mathbf{primes} = \mathbf{sieve}(l)$. At first, this may not seem very useful since any computer would take infinite time to calculate $\mathbf{primes}$. But, after some thought, you will notice that the way $\mathbf{sieve}$ calculates its output is rather special. After it produces an element of the list, it never touches that element again! Thus, although it's impossible to properly calculate $\mathbf{primes}$, we have that for any natural N, the first N elements of $\mathbf{primes}$ are calculated in a finite amount of time. So you can think of $\mathbf{primes}$ as an iterator that generates $\mathbf{era}(n)$ in finite time.

This last property is common to all anamorphisms. Although their output can be a possible infinite data structure (even if the input is finite), it is produced in an extremely structured manner. When a value is computed, we go to the next and it is never touched again. Therefore, we can still use an anamorphism to calculate finite data so long as we properly say when we wish to stop. In fact, if we take a look at programs such as games or operational systems, this is exactly the behaviour we want from them. Imagine if any OS terminated after a finite amount of steps... No, what we want is for the OS to run indefinitely producing some well-defined actions in finite time.

Building up to a collapse

By now, we've seem two kinds of recursion schemes. Anamorphisms start with a seed value and end with a data structure constructed from it, while catamorphisms start with a data structure and collapse it to end with result. Both of these are powerful tools but their type signatures are too constrained. They must explicitly return or receive a data structure. What if we want to write a recursive function from a primitive value to another primitive value? Our next (and last for today) recursion scheme addresses exactly this.

Our strategy will be to use a inductive type as our middle man. The *hylomorphism* takes a value, applies an anamorphism to turn it into a data structure and then applies a catamorphism to collapse it into another value.[11] This means that we don't need to go through yet another category to define it. No, given a functor F, the type signature for hylo is

$$\mathrm{hylo}\colon (FB \to B) \times (A \to FA) \to (A \to B)$$

and the definition is simply[12]

$$\mathbf{hylo}\ f\ g = \mathbf{cata}\ f \circ \mathbf{ana}\ g.$$

The etymology for hylomorphism is a little harder to motivate but let's try anyway. The prefix *hylo-* comes from the Greek 'ὕλη' and means "wood" or "matter". And as we've previously seem, the term morphism comes from the Greek word for form. The name hylomorphism is a pun on an Aristotelian theory of the same name which says that the being is composed from matter and form. Since I never really understood Aristotle's writings and was unable to find another word starting with *hylo-* outside the context of philosophy, I will just try to improvise an intuition. Let's think of algebras and coalgebras as ways to give form or create matter and a $\mathbf{hylo}$ combines them into a single being. It's kinda lame but was the best I could think of.

Although the first recursive function introduced in this post was the factorial (way back in the motivation) we haven't rewritten it as a recursion scheme yet. It's time to fix that. If you open the definition of $\mathbf{fat}$, you will see that $\mathbf{fat}(n)$ stand for the product of the first n natural numbers,

$$\mathbf{fat}(n) = \prod_{k=1}^{n} k.$$

Thus, an algorithm to compute the factorial is to first construct a decreasing list of integers from n to 1 and then collapse it by multiplying all its elements. We've already constructed both these functions but let's rewrite their definitions for the sake of completeness. We start with the coalgebra and algebra,

$$\begin{aligned} g(n) &= \begin{cases} \mathbf{nil}, & n < 1 \\ \mathbf{cons}(n, n-1), & \text{otherwise}, \end{cases} \\ f(\mathbf{nil}) &= 1, \\ f(\mathbf{cons}(x, y)) &= x \cdot y, \end{aligned}$$

and finally define the factorial as $\mathbf{fat} = \mathbf{hylo}\ f\ g$.

Time for a more complex example, let's see how to use hylomorphisms to better shape the call procedure tree and get faster algorithms. Recall how we defined $\mathbf{exp}$ as a catamorphism. For what we knew at the time, it was fine but, at its $O(n)$ complexity, it's just too slow. With a hylomorphism, we can do better than that. The trick is noticing that for even values, the exponential satisfies $e^{2n} =$

$(e^{n/2})^2$, which gives us a much better recursive relation. Thus, instead of multiplying e n times, we can construct a much smaller call tree and collapse it to get the desired result. We define a type **`CallTree`** as the fixed point of the functor

$$TX = \mathrm{leaf}\,\mathbb{R} \mid \mathrm{square}\,X \mid \mathrm{mult}\,\mathbb{R} \times X.$$

This type encapsulates the call tree. As base cases, n equal to zero or one means that we store e^n on a leaf node. If n is even, we construct a square node and pass $n/2$ to it, meaning that the value of this node will be squared on the algebra. Finally, if n is odd, we store e and $n-1$ on this node, which will be multiplied when folding with the algebra. The coalgebra that construct this structure is

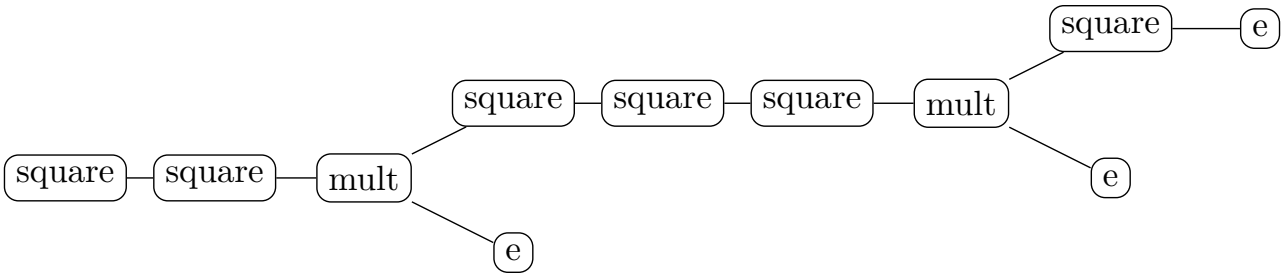
$$g(n) = \begin{cases} \mathrm{leaf}(1), & n = 0 \\ \mathrm{leaf}(e), & n = 1 \\ \mathrm{square}(\frac{n}{2}), & n \text{ is even} \\ \mathrm{mult}(e, n-1), & n \text{ is odd.} \end{cases}$$

To collapse the tree, we will use the algebra

$$\begin{aligned} f(\mathrm{leaf}\ x) &= x \\ f(\mathrm{square}(x)) &= x^2 \\ f(\mathrm{mult}(x, y)) &= x \cdot y. \end{aligned}$$

Finally, we define the exponential as the hylomorphism $\exp = \mathrm{hylo}\ f\ g$.

When we analyzed the traceback for the exponential as a catamorphism, we noticed that it did a multiplication for each $\mathbf{succ}$ that appeared until we reached $\mathbf{zero}$. Since a natural number n is the n-th successor of zero, this amounts to n multiplications. Viewing a natural number in such a unary representation has its theoretical advantages but is too slow and cumbersome in practice. With our new definition using a hylomorphism, we let the binary representation of n aid us in designing a more efficient call tree. Since the number of multiplications necessary is proportional to $\log_2 n$, we get an algorithm with complexity $O(\log n)$. This is much better! If you're not impressed, take a look at the call tree generated for $n = 100$ and see how it requires much less than 100 computations! Better yet, to compute e^{200}, this tree would have to be augmented by only one node.



Hylomorphisms really shine when an algorithm implements an intermediate data structure that is used but not returned. A special class of such are the divide-and-conquer algorithms, whose structure lend them to have elegant implementations as hylomorphisms. There are many examples such as merge sort, quick sort, convex hull and many others that can be written as an anamorphism that divides the input in a structured manner followed by a catamorphism that conquers the output from it. Please try to write any of these as a $\mathbf{hylo}$! If you pick your favorite algorithm, there is a great chance it can be written in a clear manner as a hylomorphism.

To end this section (and this post), we will do a last derivation showing that hylomorphisms do not need to be defined in the way they were. No, they accept a recursive definition in terms of themselves as any good recursion scheme should.

> **Theorem.**
> Given a F-algebra f and a F-coalgebra g, their hylomorphism satisfies
>
> $$\mathrm{hylo}\ f\ g = f \circ F(\mathrm{hylo}\ f\ g) \circ g.$$

Proof. Since the least and greatest fixed points of F are equal, we can assume that $\mathbf{in}$ and $\mathbf{out}$ are inverses to each other. By pasting together the diagrams defining the anamorphism and the catamorphism we get

$$\begin{array}{ccccc} FX & \xrightarrow{F(\mathrm{ana}\ g)} & FS & \xrightarrow{F(\mathrm{cata}\ f)} & FY \\ {\scriptstyle g}\uparrow & & {\scriptstyle \mathrm{in}}\downarrow \uparrow {\scriptstyle \mathrm{out}} & & \downarrow{\scriptstyle f} \\ X & \xrightarrow[\mathrm{ana}\ g]{} & S & \xrightarrow[\mathrm{cata}\ f]{} & Y \end{array}$$

Going from X to Y through the bottom path amounts for our definition of hylomorphism. Since the diagram is commutative, it must be equivalent to trace the top path. This, together with the functoriality of F, implies

$$\begin{aligned}\mathrm{hylo}\ f\ g &= f \circ F(\mathrm{cata}\ f) \circ F(\mathrm{ana}\ g) \circ g \\ &= f \circ F(\mathrm{cata}\ f \circ \mathrm{ana}\ g) \circ g \\ &= f \circ F(\mathrm{hylo}\ f\ g) \circ g.\end{aligned}$$

Thus we conclude the proof. □

Although our definition of a $\mathbf{hylo}$ is good to reason about, this formula has two advantages over it. The first one is practical: because there is less boilerplate happening, we only have half the function calls happening. As you can see in the proof's second line, our definition requires $\mathbf{hylo}$ to first recurse through $\mathbf{ana}$, stacking a pile of calls to $\mathbf{in}$, to immediately thereafter recurse through $\mathbf{cata}$, canceling each $\mathbf{in}$ with a $\mathbf{out}$. This does not change the algorithm's complexity but still hinders its efficiency. In very large programs that run for weeks, even the complexity constant hurts. There is a practical difference between a program that runs in a week and a program that runs in two.

The second reason is more theoretical: This last formula has no reference to fixed points. There is no explicit mention of any data structure whatsoever. All we have to do is give $\mathbf{hylo}$ one rule saying how to construct each step and one rule saying how to destruct each step and the call stack becomes our intermediate data structure! This is much cleaner and spare us from thinking about least and greatest fixed points. For short, when thinking about hylomorphisms, use the definition as an anamorphism followed by a catamorphism but when actually implementing one, use this formula instead.

Summary

This was a long journey but we finally got to the end. Throughout it we met with our three heroes: catamorphism, anamorphism, and hylomorphism, and I truly hope that the ideas they encapsulate will appear on your future programs.

Although these three are powerful tools, they're not the only recursion schemes available. Oh no, there is an entire zoo of them and, as you may expect, every single one has an arcane ancient Greek prefix. Each recursion scheme encapsulates a common pattern of recursion and comes in two flavours: construction and destruction of an inductive type. The ones we've seem today are the schemes for structural recursion but if instead we wanted primitive recursion, we would use a para- or apomorphism. It is even possible to combine different

schemes to end up with monsters such as the infamous zygohistomorphic prepromorphisms. Moreover, as one can expect of anything related to category theory, all of them are special cases of a certain categorical construction.

Another topic I haven't given the deserved attention are the fusion laws for a recursion scheme. The recursive formulas we derived for each scheme may be seem as certain algebraic equations that they must obey. Besides these, there are some other laws related to composition of schemes or other contexts. These give us some of the guarantees of structured programming we talked earlier. If a compiler knows that the code only use certain recursion schemes, it can use the fusion laws to substitute your code for something semantically equivalent but much more efficient. You get the type safe, organized code and the machine gets the faster, optimized one. Win-win.

Well, that's all for today. Good recursion for y'all!

---

1. This sum example is simple to code with a loop or linear recursion. Recursion schemes really shine when manipulating more complex data structures, such as trees with varying arity.↩
2. This is also called `Option` in some languages and is specially useful to safely model partial functions and computations that may fail.↩
3. Choosing the letter N was no accident.↩
4. Also known as `accumulate` or `foldr` in some languages. The 'r' means that this function folds a list from the right.↩
5. This is not entirely true. Questions such as non-termination or laziness/strictness may break this structure in some languages, see this link for a discussion concerning Haskell. Nevertheless, types are similar enough to a category to be worth it to think of them as forming one.↩
6. Take a look on chapter 1 of Emily Riehl's book if you want to learn this (and much more) properly. Trust me, it's a great book for the mathematically inclined.↩
7. The technical term is *least fixed point*.↩
8. Fortunately, as functional languages have no side-effects, applying a catamorphism doesn't make everyone start talking in a different language.↩
9. Also called `unfold` in the context of lists.↩

10. This example is adapted from an anamorphism in the lecture notes Programming with Categories by Brendan Fong, Bartosz Milewski and David I. Spivak.↩

11. As it stands, hylomorphisms are the theoretical analogous of me playing any construction game as a kid.↩

12. To write that, we must assume that the least and the greatest fixed point of F coincide. Although this assumption may sound strange, it always holds for languages such as Haskell.↩

Made with make and pandoc
Source @ Github