Are bananas a good source of recursion in your diet?

Sankalp Gambhir

March 26, 2025

Reference

Main reference:

Erik Meijer, Maarten Fokkinga, and Ross Paterson. **“Functional programming with bananas, lenses, envelopes and barbed wire”.** In: *Functional Programming Languages and Computer Architecture*. Ed. by John Hughes. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 124–144. ISBN: 978-3-540-47599-6

But also:

nLab authors. **recursion scheme.** `https://ncatlab.org/nlab/show/recursion+scheme`. Revision 5. Mar. 2025

Wikipedia contributors. **F-algebra — Wikipedia, The Free Encyclopedia.** [Online; accessed 25-March-2025]. 2024. URL: `https://en.wikipedia.org/w/index.php?title=F-algebra&oldid=1265837291`

Bartosz Milewski. **Category theory for programmers.** Bartosz Milewski, 2019

Back to basics

```
int fac(int target) {
  int i = 1;
  int n = 1;

  start:
    if (n > target) goto end;
    i *= n++;
    goto start;
  end:

  return i;
}

```

Abstracting out of shame

```
int facw(int target) {
  int i = 1;
  int n = 1;

  while (n <= target) i *= n++;

  return i;
}

```

Shamelessly recursive

```
def facr(target: Int): Int =
  if target <= 0 then
    1
  else
    target * facr(target - 1)
```

Shamelessly recursive

```
def facr(target: Int): Int =
  if target <= 0 then
    1
  else
    target * facr(target - 1)

def facf(target: Int): Int =
  (1 to target).fold(1)(_ * _)
```

What can we do with our abstractions?

- `goto` can express any computable function

What can we do with our abstractions?

- `goto` can express any computable function
- So can `while` loops

What can we do with our abstractions?

- `goto` can express any computable function
- So can `while` loops
- What about recursion v our abstractions? Is there a subset that allows expressing a reasonably large set of functions? All the recursively computable functions?

Catamorphism

"Cata" - down/downwards

Catamorphism

"Cata" - down/downwards

```
def cataList[A, B](b: B)(*: (A, B) => B)(l: List[A]): B =
  l match {
    case Nil => b
    case h :: t => h * cataList(b)(f)(t)
  }

```

Catamorphism

"Cata" - down/downwards

```scala
def cataList[A, B](b: B)(*: (A, B) => B)(l: List[A]): B =
  l match {
    case Nil => b
    case h :: t => h * cataList(b)(f)(t)
  }

```

Given a `b : B`, a function `*: (A, B) => B`, $(\!|$`b, *`$|\!)$ is a catamorphism over lists of type `A`. The final type is `List[A] => B`.

Anamorphism

"Ana" - up/upwards

Anamorphism

"Ana" - up/upwards

```
def cataList[A, B](b: B)(*: (A, B) => B)(l: List[A]): B =
  l match {
    case Nil => b
    case h :: t => h * cataList(b)(f)(t)
  }

```

Anamorphism

"Ana" - up/upwards

```
def cataList[A, B](b: B)(*: (A, B) => B)(l: List[A]): B =
  l match {
    case Nil => b
    case h :: t => h * cataList(b)(f)(t)
  }

```

- stopping point / base case

Anamorphism

"Ana" - up/upwards

```
def cataList[A, B](b: B)(*: (A, B) => B)(l: List[A]): B =
  l match {
    case Nil => b
    case h :: t => h * cataList(b)(f)(t)
  }

```

- stopping point / base case
- inductive case - a way to split a `B` into an `h` and a `t`.

Anamorphism

“Ana” - up/upwards

```
def cataList[A, B](b: B)(*: (A, B) => B)(l: List[A]): B =
  l match {
    case Nil => b
    case h :: t => h * cataList(b)(f)(t)
  }

```

- stopping point / base case
- inductive case - a way to split a `B` into an `h` and a `t`.

Anamorphism

"Ana" - up/upwards

```scala
def cataList[A, B](b: B)(*: (A, B) => B)(l: List[A]): B =
  l match {
    case Nil => b
    case h :: t => h * cataList(b)(f)(t)
  }

```

- stopping point / base case
- inductive case - a way to split a `B` into an `h` and a `t`. Sufficient to have an `h` and a `b1`.

Anamorphism

"Ana" - up/upwards

Anamorphism

"Ana" - up/upwards

```
def anaList[A, B](p: B => Bool)(g: B => (A, B))(b: B): List[A] =
  if p(b) then
    Nil
  else
    val (h, b1) = g(b)
    h :: anaList(p)(f)(b1)

```

or an *'unfold'*. Written as concave lenses $[\!(g, p)\!]$.

Hylomorphism

Look back at `facf`

```
def facf(target: Int): Int =
  (1 to target).fold(1)(_ * _)
```

Hylomorphism

Look back at `facf`

```
def facf(target: Int): Int =
  (1 to target).fold(1)(_ * _)
```

We produce a list (ana) and then immediately consume it (cata).

Hylomorphism

Look back at `facf`

```
def facf(target: Int): Int =
  (1 to target).fold(1)(_ * _)
```

We produce a list (ana) and then immediately consume it (cata). The composition of an anamorphism (`B => List[A]`) and a catamorphism (`List[A] => C`) is a *hylomorphism*.

Hylomorphism

Look back at `facf`

```
def facf(target: Int): Int =
  (1 to target).fold(1)(_ * _)
```

We produce a list (ana) and then immediately consume it (cata). The composition of an anamorphism (`B => List[A]`) and a catamorphism (`List[A] => C`) is a *hylomorphism*.

'Hylo' - 'dust/matter' (wood?). Originating from philosophical idea that form and matter are one. Written $[\![(b, *), (g, p)]\!]$.

The list was not necessary in the computation, we just used it as a concrete data representation of our computation plan.

Hylomorphisms, in principle

The list was not necessary in the computation, we just used it as a concrete data representation of our computation plan. In the end, the computation was on a completely different data type (nat)!

Hylomorphisms, in principle

The list was not necessary in the computation, we just used it as a concrete data representation of our computation plan. In the end, the computation was on a completely different data type (nat)!

Hylomorphisms are more generally any functions whose call trees are “list-like”.

Hylomorphisms, in principle

The list was not necessary in the computation, we just used it as a concrete data representation of our computation plan. In the end, the computation was on a completely different data type (nat)!

Hylomorphisms are more generally any functions whose call trees are "list-like". (I think these are exactly linear recursive functions?)

Why are we reiterating this with different names and pretty frames?

Why are we reiterating this with different names and pretty frames?

Why?

Why are we reiterating this with different names and pretty frames?

Why? Why these abstractions?

Downward into catamorphisms

Downward into catamorphisms

A catamorphism (a.k.a. a fold) is the unique F-algebra homomorphism from an initial algebra for the functor F.

Downward into catamorphisms

A catamorphism (a.k.a. a fold) is the unique F-algebra homomorphism from an initial algebra for the functor F.

Or:

Given an endofunctor F such that the category of F-algebras has an initial object $(\mu F, in)$, the catamorphism for an F-algebra (A, φ) is the unique homomorphism from the initial F-algebra $(\mu F, in)$ to (A, φ). The unique morphism between the carriers is also denoted cata $\varphi : \mu F \to A$.

Downward into catamorphisms

A catamorphism (a.k.a. a fold) is the unique F-algebra homomorphism from an initial algebra for the functor F.

Or:

Given an endofunctor F such that the category of F-algebras has an initial object $(\mu F, in)$, the catamorphism for an F-algebra (A, φ) is the unique homomorphism from the initial F-algebra $(\mu F, in)$ to (A, φ). The unique morphism between the carriers is also denoted cata $\varphi : \mu F \rightarrow A$.

Or in code:

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

```

Downward into catamorphisms

A catamorphism (a.k.a. a fold) is the unique F-algebra homomorphism from an initial algebra for the functor F.

Or:

Given an endofunctor F such that the category of F-algebras has an initial object $(\mu F, in)$, the catamorphism for an F-algebra (A, φ) is the unique homomorphism from the initial F-algebra $(\mu F, in)$ to (A, φ). The unique morphism between the carriers is also denoted cata $\varphi : \mu F \to A$.

Or in code:

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

```

This one line describes *all* folds. Not over lists, not over trees, but *all possible* folds.

Just Enough Category Theory to be Dangerous

A *category* is a collection of objects (points) and a collection of morphisms (arrows) between them.

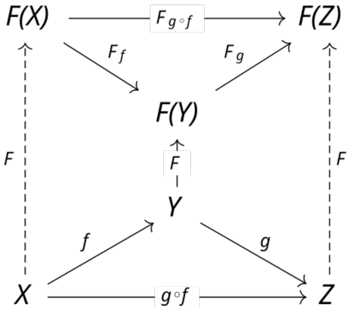
- For every object A, there is an identity morphism $id_A : A \to A$
- If $f : A \to B$ and $g : B \to C$, there must be a morphism $h : A \to C$ such that $g \circ f = h$.
- Composition is associative.

$$
\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, "g \circ f"'] \arrow[loop left, "1_A"] & B \arrow[d, "g"] \arrow[loop right, "1_B"] \\
 & C \arrow[loop right, "1_C"]
\end{tikzcd}
$$

Just Enough Category Theory to be Dangerous

A *functor* is a mapping between categories that preserves the structure of the category.

- For every object A, there is a corresponding object $F(A)$.
- For every morphism $f : A \to B$, there is a morphism $F(f) : F(A) \to F(B)$.
- Identity is preserved: $F(id_A) = id_{F(A)}$.
- Composition is preserved: $F(g \circ f) = F(g) \circ F(f)$.



We are mostly interested in *endofunctors* on types, i.e. functors that map some category containing types from our program to itself, mapping some types to other types, and correspondingly their functions.

We are mostly interested in *endofunctors* on types, i.e. functors that map some category containing types from our program to itself, mapping some types to other types, and correspondingly their functions.

For example, `List[_]` is a functor (`Type =>> Type`) that maps any type `A` to `List[A]`, and a function `A => B` to a function `List[A] => List[B]`.

Missing *initial*, *F-algebra*, and related terms. Hopefully understood better through code.

Implementation

Algebra

For an endofunctor F on a category C, an F-algebra is a pair (A, α) where A is an object in C and $\alpha : F(A) \to A$ is a morphism in C.

Algebra

For an endofunctor F on a category C, an F-algebra is a pair (A, α) where A is an object in C and $\alpha : F(A) \to A$ is a morphism in C.

So (A, α) can be an algebra if and only if there is a way to reduce values of the "next iteration" of F back into A, i.e., A is closed under $F(\cdot)$ (by α).

Algebra

For an endofunctor F on a category C, an F-algebra is a pair (A, α) where A is an object in C and $\alpha : F(A) \to A$ is a morphism in C.

So (A, α) can be an algebra if and only if there is a way to reduce values of the "next iteration" of F back into A, i.e., A is closed under $F(\cdot)$ (by α).

We can check that these algebras form a category with algebra homomorphisms as their morphisms. This category has an *initial object*, which is a bit like the bottom element in a lattice, it has an arrow to every other object in the category.

So we are saying, an initial algebra can be 'embedded' (via a homomorphism) into all other algebras. It is a minimal F-algebra.

In particular, we used the fact that our language, the meta-logic, has a way to provide us least fixpoint semantics (via recursive type definitions), so the result is in fact the least fixpoint of the functor.

Why am I looking at recursion schemes?

- Well, it's interesting. What constructs do we finally really need to eliminate arbitrary recursion?
- What kind of analysis can we do on programs when we know they are composed of only these schemes?
- Existing work on decidable verification of programs with catamorphisms as theory extensions. Led to the questions "why catamorphisms?" and "what more is out there?"
- Once we (I) understand catamorphisms, the next thing to understand is why they lead to decidable theories.
- How do structured recursion schemes relate to other decidable fragments, e.g. sufficiently surjective functions, which (I think) subsume catamorphisms?

All good questions to hopefully answer in the next few lifetimes.

Thank you!