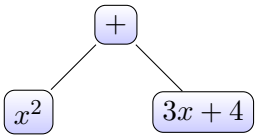
1. Expressions

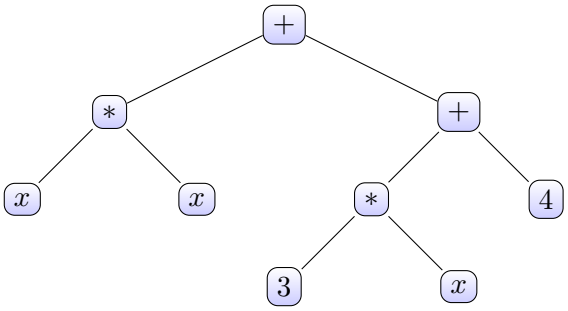
When we see an algebraic expression like

$$x^2 + 2x + 3$$

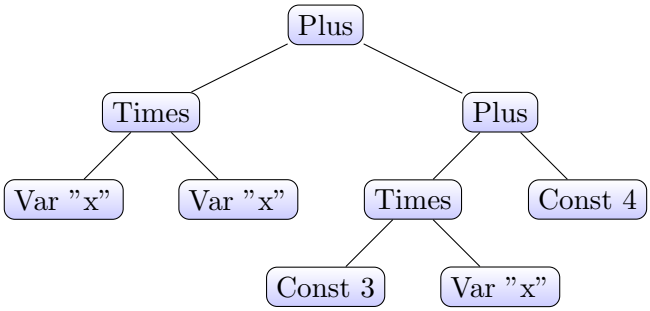
we parse it as a tree. We start with



We then recursive expand the nodes



A parser of algebraic expressions would parse it into a tree data structure



2. Examples

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

$$Fa = 1 + a^2$$

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

Let's consider, for instance, terms of type `F Int`. It's either a `Leaf` or a `Node` with two numbers in it

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

Here are some values of the type `F (F Int)`

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

We can display `y2`

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

Lifting a function $f\colon a \to b$ to F^2 can be described as applying `fmap` twice. Here's the action of `(+1)` on our test values

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

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

2.1. **Initial algebra as a colimit.** Now let's apply powers of F to the initial object, which we define as

```
data Void
  deriving Show
```

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

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

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

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

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

Au contraire, the same construction for a functor that generates infinite streams

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

will produce an uninhabited initial algebra–which is different from its terminal coalgebra, that is

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

Double application of our `F` to `Void` produces a new `Leaf` and a `Node` that contains two `Leaf`s. Notice the duplication: a `Leaf` is produced at both (in fact, at all non-zero) powers of F acting on `Void`

```
z2, v2 :: F (F Void)
z2 = Leaf
v2 = Node z1 z1
> Node Leaf Leaf
```

Powers of F acting on `Void` generate trees which terminate with `Leaf`s but no terminal `Node`s. Since we are starting with `Void`, the only function that we can apply or lift is

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

This definition requires another pragma

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

Lifting `absurd` doesn't change the shapes of these trees.

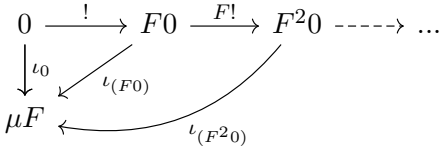
```
z1' :: F (F Void)
z1' = fmap absurd z1
> Leaf
```

```
z2', v2' :: F (F (F Void))
z2' = fmap (fmap absurd) z2
> Leaf
v2' = fmap (fmap absurd) v2
> Node Leaf Leaf
```

Higher and higher powers of F acting on `Void` will eventually produce any tree. But for any given power, there will exist even larger trees that are not generated by it.

On the other hand, if we were to take a sum (a coproduct) of all (infinitely many) powers, we'll get all the trees, but also a lot of duplication. For instance, `z1` is the same tree as `z2`, and so on. We have to have a way of identifying multiply generated trees. This is what a *colimit* of a chain of powers will accomplish.

A colimit in *Set* is a sum, or a discriminated union, in which we identify all the injections that are connected by morphisms in the cocone base.



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

In particular, `fmap absurd` maps the leaf generated by `F Void` to the leaf generated by `F (F Void)`, and so on. All trees generated by the n'th power of F are injected into the $n+1$'st power of F by `absurd` lifted by the nth power of F. The colimit is formed by equivalence classes with respect to these identifications. In particular, there is a class for a tree consisting of a single leaf whose representative can be taken from `F Void` or from `F (F Void)` and so on.

2.2. **Terminal coalgebra as a limit.** Things are different with powers of F acting on the terminal object: singleton or unit type. We get trees with both terminal `Leaf`s and terminal `Node`s that contain units

```
w1, u1 :: F ()
w1 = Leaf
u1 = Node () ()
```

The square of F produces, among others,

```
w2, u2, t2, s2 :: F (F ())
w2 = Leaf
> Leaf
u2 = Node w1 w1
> Node Leaf Leaf
t2 = Node w1 u1
> Node Leaf (Node () ())
s2 = Node u1 u1
> Node (Node () ()) (Node () ())
```

This time we are interested in mapping into the terminal object

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

We are going to lift `unit` to transform $F^{n+1}1$ to F^n1.

Applying `unit` directly to `F ()` turns it into `()`.

Values of the type F^21 are mapped to values of the type $F1$

```
w2' = fmap unit w2
> Leaf
u2' = fmap unit u2
> Node () ()
t2' = fmap unit t2
> Node () ()
s2' = fmap unit s2
> Node () ()
```

and so on.

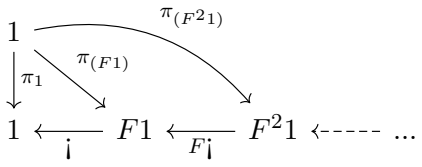
The following pattern emerges. F^n1 contains trees that end with either leaves or the values of the unit type. The latter play the role of potential seeds. Each seed can give rise to any value of the type `F ()`, be it a leaf or a node. Normally, we use a particular coalgebra to generate these values from a carrier, which serves as the seed. Here, we are modeling a situation where a singleton seed generates the whole set of all possible values. It's a "universal" seed.

Except that we are reversing the process: by lifting `unit` we transform F^{n+1} to F^n. Instead of expanding the seeds, we are shrinking single-level trees, or buds, back to their seeds. For instance, take

```
t2 :: F (F ())
t2 = Node w1 u1
> Node Leaf (Node () ())
```

which has two buds, `Leaf` and `Node () ()`. `fmap unit` shrinks it down to `Node () ()`. We can thing of the "inverse" of the lifted `unit` to generate all possible trees of the next level from the universal seeds embedded in the previous level.

Since a terminal coalgebra is dual to intial algebra, we can generate it as a limit of a sequence that contains powers of F acting on `()` connected by lifted `unit`s. A limit in Set is equivalent to a set of cones whose tip is the singleton set. The base of such cone is a sequence of $F^n 1$, starting with 1. The arrows in the base are lifted `unit`s (or ¡ in the picture below) pointing from $F^{n+1}1$ to $F^n 1$.



This chain is infinite, so it's not immediately obvious what part of the cone is relevant. But imagine that the chain terminates on the right. Then the rightmost projection uniquely determines all the others (all the triangles commute).

Notice also that if we select a leaf-terminated tree, such as `Node Leaf Leaf`, than all the values to the right of it (with higher powers of F) in the sequence must all be the same. For instance, the third power of F

```
w3 :: F (F (F ()))
w3 = Node Leaf Leaf
```

is mapped to the second power of F that gives the same tree

```
w3' :: F (F ())
w3' = fmap (fmap unit) w3
> Node Leaf Leaf
```

This way, all finite trees are generated from cones that eventually reach a stable point on the right. However, there are cones that never stabilize, and those correspond to infinite trees.

The `StreamF` functor, in particular, doesn't produce leaves, so all its cones are infinite, corresponding to infinite `Stream`s.

3. Adjunction

There is an obvious forgetful functor U from the category of F-algebras to Set. Under certain conditions, the left adjoint free functor Φ exists

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

This adjunction can be evaluated at the initial object—the empty set.

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

This shows that there is a unique algebra morphism—the catamorphism— from $\Phi 0$ to any algebra a.

Conversely, there is a cofree functor Ψ

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

It can be evaluated at a terminal object

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

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

4. Fixed point

In Haskell, the least fixed point and the greatest fixed point, are often given by the same formula

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

We can still define them separately by directly encoding their universal property.

5. Initial algebra and catamorphism

The initial algebra can be defined by its mapping out property.

```
newtype Mu f = Mu (forall a. (f a -> a) -> a)
```

Notice that this definition requires the following language pragma

```
{-# language RankNTypes #-}
```

This definition works because for every least fixed point one can define a catamorphism, which can be rewritten as

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

(`flip` is a function that reverses the order of arguments of its (function) argument.) What the definition of `Mu` is saying is that it's an object that, for all algebras, has a mapping out to a catamorphism.

It's easy to define a catamorphism in terms of `Mu`, since `Mu` *is* a catamorphism

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

The challenge is to construct terms of type `Mu f`. For instance, let's convert a list of `a` to a term of type `Mu (ListF a)`

```
mkList :: forall a. [a] -> Mu (ListF a)
mkList as = Mu cata
  where cata :: forall x. (ListF a x -> x) -> x
        cata unf = go as
          where
```

```
        go [] = unf NilF
        go (n: ns) = unf (ConsF n (go ns))
```

Notice that we use the type `a` defined in the type signature of `mkList` to define the type signature of the helper function `cata`. For the compiler to identify the two, we have to use the pragma

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

You can now verify that

```
cataMu sumAlg (mkList [1..10])
```

produces the correct result for the following algebra

```
sumAlg :: Algebra (ListF Int) Int
sumAlg NilF = 0
sumAlg (ConsF a x) = a + x
```

6. Terminal coalgebra and anamorphism

The terminal coalgebra, on the other hand, is defined by its mapping in property. This requires a definition in terms of existential types. If Haskell had an existential quantifier, we could write the following definition for the terminal coalgebra

```
data Nu f =  Nu (exists a. (a -> f a, a))
```

Existential types can be encoded in Haskell using the so called Generalized Algebraic Data Types or GADTs

```
data Nu f where
  Nu :: (a -> f a) -> a -> Nu f
```

The use of GADTs requires the language pragma

```
{-# language GADTs #-}
```

The argument is that, for every greatest fixed point one can define an anamorphism

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

We can uncurry it

```
ana :: Functor f => forall a. (a -> f a, a) -> Fix f
ana (coa, x) = Fix (fmap (curry ana coa) (coa x))
```

A universally quantified mapping out

```
forall a. ((a -> f a, a) -> Fix f)
```

is equivalent to a mapping out of an existential type (in pseudo-Haskell)

```
(exists a. (a -> f a, a)) -> Fix f
```

which is the type signature of the constructor of `Nu f`.

The intuition is that, if you want to implement a function from an existential type—a type which hides some other type `a` to which you have no access—your function has to be prepared to handle any `a`. In other words, it has to be polymorphic in `a`.

Since in an existential type we have no access to the hidden type, it has to provide both the "producer" and the "consumer" for this type. Here we are given a value of type `a` on the produces side, and the function `a -> f a` as the consumer. All we can do is to apply this function to `a` and obtain the term of the type `f a`. Since `f` is a functor, we can lift our function and apply it again, to get something of the type `f (f a)`. Continuing this process, we can obtain arbitrary powers of `f` acting on `a`. We get a recursive data type.

An anamorphism in terms of `Nu` is given by

```
anaNu :: Functor f => Coalgebra f a -> a -> Nu f
anaNu coa a = Nu coa a
```

Notice however that we cannot directly pass the result of `anaNu` to `cataMu` because we are no longer guaranteed that the initial algebra is the same as the terminal coalgebra for a given functor.

7. End/Coend formulation

Let's rewrite `Mu` using GADTs

```
data Mu f where
  Mu :: (forall a. (f a -> a) -> a) -> Mu f
```

We use a natural transformation to construct a `Mu`. Categorically, we can write it as an end

$$\mu f = \int_a a^{C(fa,a)}$$

It's an end over the profunctor

$$pab = b^{C(fb,a)}$$

Where the power is defined as

$$C(x, a^s) \cong Set(s, C(x, a))$$

Projection from the end is a catamorphism

$$\pi_a \colon \mu f \to a^{C(fa,a)}$$

It's a morphism from the hom-set

$$C(\mu f, a^{C(fa,a)})$$

or an element of

$$Set(C(fa, a), C(\mu f, a))$$

Similarly, we can rewrite `Nu`

```
data Nu f where
  Nu :: (a -> f a) -> a -> Nu f
```

as a coend

$$\nu f = \int^a C(a, fa) \cdot a$$

over the profunctor

$$pab = C(a, fb) \cdot b$$

where the copower is defined as

$$C(s \cdot a, x) \cong Set(s, C(a, x))$$

Injection into the coend is an anamorphism

$$\iota_a \colon C(a, fa) \cdot a \to \nu f$$

It's a morphism from the hom-set

$$C(C(a, fa) \cdot a, \nu f)$$

or an element of

$$Set(C(a, fa), C(a, \nu f))$$

Because of Lambek's lemma, an initial algebra is also a coalgebra, and a terminal coalgebra is also an algebra. Universality, therefore, tells us that there is a unique algebra morphism (as well as a unique coalgebra morphism)

$$\phi \colon \mu f \to \nu f$$

This is a canonical embedding, but not necessarily an isomorphism.

The two profunctors in the definition of `Mu` and `Nu` can be written as

```
data M f a b = M ((f b -> a) -> b)
```

```
instance Functor f => Profunctor (M f) where
  dimap g g' (M h) = M (\j -> g'( h (g . j . fmap g')))
```

```
data N f a b = N (a -> f b) b
```

```
instance Functor f => Profunctor (N f) where
  dimap g g' (N h b) = N (fmap g' . h . g) (g' b)
```

8. Hylomorphism

If the mapping from from the terminal coalgebra ν to the initial algebra μ exists, it is an element of the following hom-set

$$C\Big(\int^a C(a, fa) \cdot a, \int_b b^{C(fb,b)}\Big)$$

By co-continuity of the hom-functor, this is isomorphic to

$$\int_a C\Big(C(a, fa) \cdot a, \int_b b^{C(fb,b)}\Big)$$

Using continuity we get

$$\int_{a,b} C\Big(C(a, fa) \cdot a, b^{C(fb,b)}\Big)$$

Using the definition of the copower

$$C(s \cdot a, x) \cong Set(s, C(a, x))$$

we get

$$\int_{a,b} Set\Big(C(a, fa), C(a, b^{C(fb,b)})\Big)$$

And using the definition of the power

$$C(x, a^s) \cong Set(s, C(x, a))$$

we get

$$\int_{a,b} Set\Big(C(a, fa), Set\big(C(fb, b), C(a, b)\big)\Big)$$

Finally, applying the currying adjunction we get

$$\int_{a,b} Set\Big(C(a, fa) \times C(fb, b), C(a, b)\Big)$$

in which you may recognize a hylomorphism

```
hylo :: Functor f => Coalgebra f a -> Algebra f b -> a -> b
hylo coa alg = alg . fmap (hylo coa alg) . coa
```

9. Bibliography

- Michael Barr, Terminal coalgebras for endofunctors on sets

10. Random things

10.1. **Initial algebra structure map.**

$$j \colon f(\mu f) \to \mu f$$

$$C\Big(f(\mu f), \int_b b^{C(fb,b)}\Big)$$

is a member of

$$\int_b C\Big(f(\mu f), b^{C(fb,b)}\Big)$$

Using the definition of the power

$$C(x, a^s) \cong Set(s, C(x, a))$$

we get

$$\int_b Set\Big(C(fb, b), C\big(f(\mu f), b\big)\Big)$$

Using Yoneda lemma we replace b with $f(\mu f)$

$$C(f(f(\mu f)), f(\mu f))$$

10.2. **Terminal coalgebra structure map.**

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

is a member of

$$C\Big(\int^a C(a, fa)\cdot a, f(\nu f)\Big)$$

$$\int_a C\Big(C(a, fa)\cdot a, f(\nu f)\Big)$$

Using the definition of the copower

$$C(s\cdot a, x) \cong Set(s, C(a, x))$$

we get

$$\int_a Set\Big(C(a, fa), C\big(a, f(\nu f)\big)\Big)$$

Yoneda lemma

$$C(f(\nu f), f(f(\nu f)))$$

10.3. **Kan extensions.**

$$\mu f = \int_a a^{C(fa,a)}$$

$$\nu f = \int^a C(a, fa)\cdot a$$

These formulas are reminiscent of Kan extensions. For comparison, the right Kan extension of g along f is given by

$$(Ran_f g)c = \int_a (ga)^{C(c,fa)}$$

The left Kan extension is

$$(Lan_f g)c = \int^a C(fa, c)\cdot ga$$

If f has left and right adjoints, they are given by

$$Ran_f Id \dashv f \dashv Lan_f Id$$

In particular, using the adjunction

$$(Lan_f Id)c = \int^a C(a, (Lan_f Id)c)\cdot a$$

This shows that $(Lan_f Id)c$ is a fixed point of the functor

$$\Phi(x) = \int^a C(a, x)\cdot a$$

10.4. **Ends as limits.** Twisted arrow category on $Tw(\mathbf{C})$ has, as objects, morphisms in $\mathbf{C}$ (or, strictly speaking, triples $(a, b, f\colon a \to b)$). A morphism from $f\colon a \to b$ to $g\colon a' \to b'$ is a pair of morphisms

$$(h\colon a' \to a, h'\colon b \to b')$$

For every profunctor $p\colon C^{op} \times C \to \mathbf{Set}$ define a functor $\bar{p}\colon Tw(\mathbf{C}) \to Set$. On objects

$$\bar{p}(a, b, f) = pab$$

and on morphisms, it's just profunctor lifting.

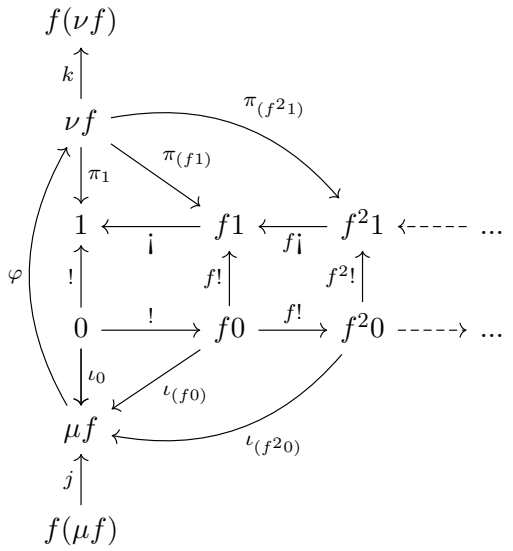
It can be shown that the end is just a limit over the twisted arrow category

$$\int_c pcc \cong \lim_{Tw(C)} \bar{p}$$

Similarly, the coend is a colimit over $Tw(C^{op})^{op}$

$$\int^c pcc \cong \operatorname*{colim}_{Tw(C^{op})^{op}} \bar{p}$$

10.5. **Iterative solution.** Terminal coalgebra is a limit, and initial algebra is a colimit of these two chains



11. Algebra morphisms

$$\begin{array}{ccc} F[A] & \xrightarrow{F(f)} & F[B] \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle b} \\ A & \xrightarrow[f]{} & B \end{array}$$

Composition

$$\begin{array}{ccccc} F[A] & \xrightarrow{F(f)} & F[B] & \xrightarrow{F(g)} & F[C] \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle b} & & \downarrow{\scriptstyle c} \\ A & \xrightarrow[f]{} & B & \xrightarrow[g]{} & C \end{array}$$

Identity

$$\begin{array}{ccc} F[A] & \xrightarrow{id} & F[A] \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle a} \\ A & \xrightarrow[id]{} & A \end{array}$$

Initial algebra

$$\begin{array}{ccc} F[I] & \xrightarrow{F(m)} & F[A] \\ {\scriptstyle j}\downarrow & & \downarrow{\scriptstyle a} \\ I & \xrightarrow[m]{} & A \end{array}$$

$$\begin{array}{ccc} F[Fix[F]] & \xrightarrow{F(m)} & F[A] \\ {\scriptstyle in}\downarrow\uparrow{\scriptstyle out} & & \downarrow{\scriptstyle alg} \\ Fix[F] & \xrightarrow[m]{} & A \end{array}$$

$$\begin{array}{ccc} FI & \xrightarrow{Fm} & FA \\ {\scriptstyle j}\downarrow & & \downarrow{\scriptstyle a} \\ I & \xrightarrow[m]{} & A \end{array}$$

$$\begin{array}{ccc} FI & \xrightarrow{Fm} & F(FI) \\ {\scriptstyle j}\downarrow & & \downarrow{\scriptstyle Fj} \\ I & \xrightarrow[m]{} & FI \end{array}$$

$$\begin{array}{ccc} F(FI) & \xrightarrow{Fj} & FI \\ {\scriptstyle Fj}\downarrow & & \downarrow{\scriptstyle j} \\ FI & \xrightarrow[j]{} & I \end{array}$$

$$\begin{array}{ccccc} FI & \xrightarrow{Fm} & F(FI) & \xrightarrow{Fj} & FI \\ {\scriptstyle j}\downarrow & & \downarrow{\scriptstyle Fj} & & \downarrow{\scriptstyle j} \\ I & \xrightarrow[m]{} & FI & \xrightarrow[j]{} & I \end{array}$$