In Haskell, the least fixed point and the greatest fixed point, are 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.

1. 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

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

2. 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

```haskell
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

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

The use of GADTs requires the language pragma

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

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

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

We can uncurry it

```haskell
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

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

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

```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.

3. 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)
```

4. 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
```

5. Bibliography

- Michael Barr, Terminal coalgebras for endofunctors on sets

6. Random things

6.1. 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 $f(\mu f)$

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

6.2. 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

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

6.3. **Kan extensions.**

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

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

These formulas are reminescent 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

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

6.4. **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 $Tw(C^{op})^{op}$

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

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

$$
\begin{tikzcd}
f(\nu f) & & & \\
\nu f \arrow[u, "k"] \arrow[d, "\pi_1"] \arrow[dr, "\pi_{f1}"] \arrow[drr, bend left, "\pi_{f^2 1}"] & & & \\
1 & f1 \arrow[l, "!"] & f^2 1 \arrow[l, "f!"] & \ldots \arrow[l, dashed] \\
0 \arrow[u, "!"] \arrow[r, "!"] \arrow[d, "\iota_0"] & f0 \arrow[u, "f!"] \arrow[r, "f!"] \arrow[dl, "\iota_{f0}"] & f^2 0 \arrow[u, "f^2!"] \arrow[r, dashed] \arrow[dll, bend left, "\iota_{f^2 0}"] & \ldots \\
\mu f \arrow[uuu, bend left, "\phi"] & & & \\
f(\mu f) \arrow[u, "j"] & & &
\end{tikzcd}
$$