In Haskell, least fixed point and greatest fixed point, when uniquely defined, coincide. We can still define them separately by directly encoding their universal property.

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 myAlg (mkList [1..10])
```

produces the correct result for the following algebra

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

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.

1. End/Coend formulation

Mu can be rewritten as

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

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

End over profunctor

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

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

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

Coend over profunctor

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

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) $\mu f \to \nu f$, but not necessarily the other way around.

These formulas are reminescent of Kan extensions. For comparison

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

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

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

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

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)
```

2. 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 $Tw(C^{op})^{op}$

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

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

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

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

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