26

Ends and Coends

There are many intuitions that we may attach to morphisms in a category, but we can all agree that if there is a morphism from the object *a* to the object *b* than the two objects are in some way "related." A morphism is, in a sense, the proof of this relation. This is clearly visible in any poset category, where a morphism *is* a relation. In general, there may be many "proofs" of the same relation between two objects. These proofs form a set that we call the hom-set. When we vary the objects, we get a mapping from pairs of objects to sets of "proofs." This mapping is functorial — contravariant in the first argument and covariant in the second. We can look at it as establishing a global relationship between objects in the category. This relationship is described by the hom-functor:

$$\mathbf{C}(-, =) :: \mathbf{C}^{op} \times \mathbf{C} \to \mathbf{Set}$$

In general, any functor like this may be interpreted as establishing a relation between objects in a category. A relation may also involve two different categories **C** and **D**. A functor, which describes such a relation, has the following signature and is called a profunctor:

$$p :: \mathbf{D}^{op} \times \mathbf{C} \to \mathbf{Set}$$

Mathematicians say that it's a profunctor from **C** to **D** (notice the inversion), and use a slashed arrow as a symbol for it:

$$\mathbf{C} \nrightarrow \mathbf{D}$$

You may think of a profunctor as a *proof-relevant relation* between objects of **C** and objects of **D**, where the elements of the set symbolize proofs of the relation. Whenever $p\ a\ b$ is empty, there is no relation between a and b. Keep in mind that relations don't have to be symmetric.

Another useful intuition is the generalization of the idea that an endofunctor is a container. A profunctor value of the type $p\ a\ b$ could then be considered a container of bs that are keyed by elements of type a. In particular, an element of the hom-profunctor is a function from a to b.

In Haskell, a profunctor is defined as a two-argument type constructor `p` equipped with the method called `dimap`, which lifts a pair of functions, the first going in the "wrong" direction:

```haskell
class Profunctor p where
    dimap :: (c -> a) -> (b -> d) -> p a b -> p c d
```

The functoriality of the profunctor tells us that if we have a proof that `a` is related to `b`, then we get the proof that `c` is related to `d`, as long as there is a morphism from `c` to `a` and another from `b` to `d`. Or, we can think of the first function as translating new keys to the old keys, and the second function as modifying the contents of the container.

For profunctors acting within one category, we can extract quite a lot of information from diagonal elements of the type $p\ a\ a$. We can prove that b is related to c as long as we have a pair of morphisms $b \to a$ and $a \to c$. Even better, we can use a single morphism to reach off-diagonal values. For instance, if we have a morphism $f :: a \to b$, we can lift the pair $\langle f, \mathbf{id}_b \rangle$ to go from $p\ b\ b$ to $p\ a\ b$:

```
dimap f id (p b b) :: p a b
```

Or we can lift the pair $\langle \mathbf{id}_a, f \rangle$ to go from $p\ a\ a$ to $p\ a\ b$:

```
dimap id f (p a a) :: p a b
```

26.1 Dinatural Transformations

Since profunctors are functors, we can define natural transformations between them in the standard way. In many cases, though, it's enough to define the mapping between diagonal elements of two profunctors. Such a transformation is called a dinatural transformation, provided it satisfies the commuting conditions that reflect the two ways we can connect diagonal elements to non-diagonal ones. A dinatural transformation between two profunctors p and q, which are members of the functor category $[\mathbf{C}^{op} \times \mathbf{C}, \mathbf{Set}]$, is a family of morphisms:

$$\alpha_a :: p\ a\ a \to q\ a\ a$$

for which the following diagram commutes, for any $f :: a \to b$:

$$\begin{tikzcd}
& p\,b\,a \arrow[dl, "p\,f\,id_a"'] \arrow[dr, "p\,id_b\,f"] & \\
p\,a\,a \arrow[d, "\alpha_a"'] & & p\,b\,b \arrow[d, "\alpha_b"'] \\
q\,a\,a \arrow[dr, "q\,id_a\,f"'] & & q\,b\,b \arrow[dl, "q\,f\,id_b"] \\
& q\,a\,b &
\end{tikzcd}$$

Notice that this is strictly weaker than the naturality condition. If α were a natural transformation in $[\mathbf{C}^{op} \times \mathbf{C}, \mathbf{Set}]$, the above diagram could be constructed from two naturality squares and one functoriality condition (profunctor q preserving composition):

$$\begin{tikzcd}
& p\,b\,a \arrow[dl, "p\,f\,id_a"'] \arrow[dd, "\alpha_{ba}"] \arrow[dr, "p\,id_b\,f"] & \\
p\,a\,a \arrow[dd, "\alpha_{aa}"'] & & p\,b\,b \arrow[dd, "\alpha_{bb}"] \\
& q\,b\,a \arrow[dl, "q\,f\,id_a"] \arrow[dr, "q\,id_b\,f"'] & \\
q\,a\,a \arrow[dr, "q\,id_a\,f"'] & & q\,b\,b \arrow[dl, "q\,f\,id_b"] \\
& q\,a\,b &
\end{tikzcd}$$

Notice that a component of a natural transformation α in $[\mathbf{C}^{op} \times \mathbf{C}, \mathbf{Set}]$ is indexed by a pair of objects α_{ab}. A dinatural transformation, on the other hand, is indexed by one object, since it only maps diagonal elements of the respective profunctors.

26.2 Ends

We are now ready to advance from “algebra” to what could be considered the “calculus” of category theory. The calculus of ends (and coends) borrows ideas and even some notation from traditional calculus. In particular, the coend may be understood as an infinite sum or an integral, whereas the end is similar to an infinite product. There is even something that resembles the Dirac delta function.

An end is a generalization of a limit, with the functor replaced by a profunctor. Instead of a cone, we have a wedge. The base of a wedge is formed by diagonal elements of a profunctor p. The apex of the wedge is an object (here, a set, since we are considering **Set**-valued profunctors), and the sides are a family of functions mapping the apex to the sets in the base. You may think of this family as one polymorphic function — a function that’s polymorphic in its return type:

$$\alpha :: \forall a\ .\ apex \to p\ a\ a$$

Unlike in cones, within a wedge we don’t have any functions that would connect vertices of the base. However, as we’ve seen earlier, given any morphism $f :: a \to b$ in **C**, we can connect both $p\ a\ a$ and $p\ b\ b$ to the common set $p\ a\ b$. We therefore insist that the following diagram commute:

$$\begin{tikzcd}
 & apex \arrow[dl, "\alpha_a"'] \arrow[dr, "\alpha_b"] & \\
p\,a\,a \arrow[dr, "p\,id_a\,f"'] & & p\,b\,b \arrow[dl, "p\,f\,id_b"] \\
 & p\,a\,b &
\end{tikzcd}$$

This is called the *wedge condition*. It can be written as:

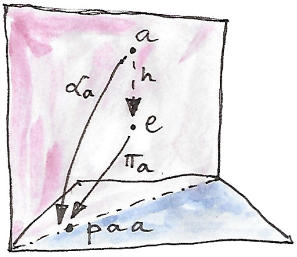
$$p\ \mathbf{id}_a\ f \circ \alpha_a = p\ f\ \mathbf{id}_b \circ \alpha_b$$

Or, using Haskell notation:

```
dimap id f . alpha = dimap f id . alpha
```

We can now proceed with the universal construction and define the end of p as the universal wedge — a set e together with a family of functions π such that for any other wedge with the apex a and a family α there is a unique function $h :: a \to e$ that makes all triangles commute:

$$\pi_a \circ h = \alpha_a$$



The symbol for the end is the integral sign, with the "integration variable" in the subscript position:

$$\int_c p\ c\ c$$

Components of π are called projection maps for the end:

$$\pi_a :: \int_c p\ c\ c \to p\ a\ a$$

Note that if C is a discrete category (no morphisms other than the identities) the end is just a global product of all diagonal entries of p across the whole category C. Later I'll show you that, in the more general case, there is a relationship between the end and this product through an equalizer.

In Haskell, the end formula translates directly to the universal quantifier:

```
forall a. p a a
```

Strictly speaking, this is just a product of all diagonal elements of p, but the wedge condition is satisfied automatically due to parametricity[1]. For any function $f :: a \to b$, the wedge condition reads:

```
dimap f id . pi = dimap id f . pi
```

or, with type annotations:

$$\texttt{dimap f id}_b\ \texttt{. pi}_b\ \texttt{= dimap id}_a\ \texttt{f . pi}_a$$

where both sides of the equation have the type:

```
Profunctor p => (forall c. p c c) -> p a b
```

and `pi` is the polymorphic projection:

```
pi :: Profunctor p => forall c. (forall a. p a a) -> p c c
pi e = e
```

[1] https://bartoszmilewski.com/2017/04/11/profunctor-parametricity/

Here, type inference automatically picks the right component of **e**.

Just as we were able to express the whole set of commutation conditions for a cone as one natural transformation, likewise we can group all the wedge conditions into one dinatural transformation. For that we need the generalization of the constant functor Δ_c to a constant profunctor that maps all pairs of objects to a single object c, and all pairs of morphisms to the identity morphism for this object. A wedge is a dinatural transformation from that functor to the profunctor p. Indeed, the dinaturality hexagon shrinks down to the wedge diamond when we realize that Δ_c lifts all morphisms to one identity function.

Ends can also be defined for target categories other than **Set**, but here we'll only consider **Set**-valued profunctors and their ends.

26.3 Ends as Equalizers

The commutation condition in the definition of the end can be written using an equalizer. First, let's define two functions (I'm using Haskell notation, because mathematical notation seems to be less user-friendly in this case). These functions correspond to the two converging branches of the wedge condition:

```haskell
lambda :: Profunctor p => p a a -> (a -> b) -> p a b
lambda paa f = dimap id f paa

rho :: Profunctor p => p b b -> (a -> b) -> p a b
rho pbb f = dimap f id pbb
```

Both functions map diagonal elements of the profunctor **p** to polymorphic functions of the type:

```
type ProdP p = forall a b. (a -> b) -> p a b
```

These functions have different types. However, we can unify their types, if we form one big product type, gathering together all diagonal elements of `p`:

```
newtype DiaProd p = DiaProd (forall a. p a a)
```

The functions `lambda` and `rho` induce two mappings from this product type:

```
lambdaP :: Profunctor p => DiaProd p -> ProdP p
lambdaP (DiaProd paa) = lambda paa

rhoP :: Profunctor p => DiaProd p -> ProdP p
rhoP (DiaProd pbb) = rho pbb
```

The end of `p` is the equalizer of these two functions. Remember that the equalizer picks the largest subset on which two functions are equal. In this case it picks the subset of the product of all diagonal elements for which the wedge diagrams commute.

26.4 Natural Transformations as Ends

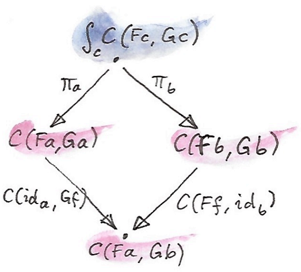
The most important example of an end is the set of natural transformations. A natural transformation between two functors F and G is a family of morphisms picked from hom-sets of the form $\mathbf{C}(F\ a, G\ a)$. If it weren't for the naturality condition, the set of natural transformations would be just the product of all these hom-sets. In fact, in Haskell, it is:

```
forall a. f a -> g a
```

The reason it works in Haskell is because naturality follows from parametricity. Outside of Haskell, though, not all diagonal sections across such hom-sets will yield natural transformations. But notice that the mapping:

$$\langle a, b \rangle \to \mathbf{C}(F\ a, G\ b)$$

is a profunctor, so it makes sense to study its end. This is the wedge condition:


$$\begin{tikzcd}
 & \int_c C(Fc, Gc) \arrow[dl, "\pi_a"'] \arrow[dr, "\pi_b"] & \\
C(Fa, Ga) \arrow[dr, "C(id_a, Gf)"'] & & C(Fb, Gb) \arrow[dl, "C(Ff, id_b)"] \\
 & C(Fa, Gb) &
\end{tikzcd}$$

Let's just pick one element from the set $\int_c \mathbf{C}(F\ c, G\ c)$. The two projections will map this element to two components of a particular transformation, let's call them:

$$\tau_a :: F\ a \to G\ a$$
$$\tau_b :: F\ b \to G\ b$$

In the left branch, we lift a pair of morphisms $\langle \mathbf{id}_a, G\ f \rangle$ using the hom-functor. You may recall that such lifting is implemented as simultaneous pre- and post-composition. When acting on τ_a the lifted pair gives us:

$$G\ f \circ \tau_a \circ \mathbf{id}_a$$

The other branch of the diagram gives us:

$$\mathbf{id}_b \circ \tau_b \circ F\ f$$

Their equality, demanded by the wedge condition, is nothing but the naturality condition for τ.

26.5 Coends

As expected, the dual to an end is called a coend. It is constructed from a dual to a wedge called a cowedge (pronounced co-wedge, not cow-edge).

An edgy cow?

The symbol for a coend is the integral sign with the "integration variable" in the superscript position:

$$\int^c p\ c\ c$$

Just like the end is related to a product, the coend is related to a coproduct, or a sum (in this respect, it resembles an integral, which is a limit of a sum). Rather than having projections, we have injections going from the diagonal elements of the profunctor down to the coend. If it weren't for the cowedge conditions, we could say that the coend of the profunctor p is either $p\ a\ a$, or $p\ b\ b$, or $p\ c\ c$, and so on. Or we could say that there exists such an a for which the coend is just the set $p\ a\ a$. The universal quantifier that we used in the definition of the end turns into an existential quantifier for the coend.

This is why, in pseudo-Haskell, we would define the coend as:

```
exists a. p a a
```

The standard way of encoding existential quantifiers in Haskell is to use universally quantified data constructors. We can thus define:

```haskell
data Coend p = forall a. Coend (p a a)
```

The logic behind this is that it should be possible to construct a coend using a value of any of the family of types $p\ a\ a$, no matter what a we chose.

Just like an end can be defined using an equalizer, a coend can be described using a *coequalizer*. All the cowedge conditions can be summarized by taking one gigantic coproduct of $p\ a\ b$ for all possible functions $b \to a$. In Haskell, that would be expressed as an existential type:

```haskell
data SumP p = forall a b. SumP (b -> a) (p a b)
```

There are two ways of evaluating this sum type, by lifting the function using `dimap` and applying it to the profunctor p:

```
lambda, rho :: Profunctor p => SumP p -> DiagSum p
lambda (SumP f pab) = DiagSum (dimap f id pab)
rho    (SumP f pab) = DiagSum (dimap id f pab)
```

where `DiagSum` is the sum of diagonal elements of p:

```
data DiagSum p = forall a. DiagSum (p a a)
```

The coequalizer of these two functions is the coend. A coequalizer is obtained from `DiagSum p` by identifying values that are obtained by applying `lambda` or `rho` to the same argument. Here, the argument is a pair consisting of a function $b \to a$ and an element of $p\ a\ b$. The application of `lambda` and `rho` produces two potentially different values of the type `DiagSum p`. In the coend, these two values are identified, making the cowedge condition automatically satisfied.

The process of identification of related elements in a set is formally known as taking a quotient. To define a quotient we need an *equivalence relation* $\sim$, a relation that is reflexive, symmetric, and transitive:

$$\begin{aligned}
& a \sim a \\
& \text{if } a \sim b \text{ then } b \sim a \\
& \text{if } a \sim b \text{ and } b \sim c \text{ then } a \sim c
\end{aligned}$$

Such a relation splits the set into equivalence classes. Each class consists of elements that are related to each other. We form a quotient set by picking one representative from each class. A classic example is the definition of rational numbers as pairs of whole numbers with the following equivalence relation:

$$(a, b) \sim (c, d) \text{ iff } a * d = b * c$$

It's easy to check that this is an equivalence relation. A pair (a, b) is interpreted as a fraction $\frac{a}{b}$, and fractions whose numerator and denominator have a common divisor are identified. A rational number is an equivalence class of such fractions.

You might recall from our earlier discussion of limits and colimits that the hom-functor is continuous, that is, it preserves limits. Dually, the contravariant hom-functor turns colimits into limits. These properties can be generalized to ends and coends, which are a generalization of limits and colimits, respectively. In particular, we get a very useful identity for converting coends to ends:

$$\mathbf{Set}(\int^x p\ x\ x, c) \cong \int_x \mathbf{Set}(p\ x\ x, c)$$

Let's have a look at it in pseudo-Haskell:

```
(exists x. p x x) -> c ≅ forall x. p x x -> c
```

It tells us that a function that takes an existential type is equivalent to a polymorphic function. This makes perfect sense, because such a function must be prepared to handle any one of the types that may be encoded in the existential type. It's the same principle that tells us that a function that accepts a sum type must be implemented as a case statement, with a tuple of handlers, one for every type present in the sum. Here, the sum type is replaced by a coend, and a family of handlers becomes an end, or a polymorphic function.

26.6 Ninja Yoneda Lemma

The set of natural transformations that appears in the Yoneda lemma may be encoded using an end, resulting in the following formulation:

$$\int_z \mathbf{Set}(\mathbf{C}(a, z), F\ z) \cong F\ a$$

There is also a dual formula:

$$\int^z \mathbf{C}(z, a) \times F\ z \cong F\ a$$

This identity is strongly reminiscent of the formula for the Dirac delta function (a function $\delta(a-z)$, or rather a distribution, that has an infinite peak at $a = z$). Here, the hom-functor plays the role of the delta function.

Together these two identities are sometimes called the Ninja Yoneda lemma.

To prove the second formula, we will use the consequence of the Yoneda embedding, which states that two objects are isomorphic if and only if their hom-functors are isomorphic. In other words $a \cong b$ if and only if there is a natural transformation of the type:

$$[\mathbf{C}, \mathbf{Set}](\mathbf{C}(a, -), \mathbf{C}(b, =))$$

that is an isomorphism.

We start by inserting the left-hand side of the identity we want to prove inside a hom-functor that's going to some arbitrary object c:

$$\mathbf{Set}(\int^z \mathbf{C}(z, a) \times F\ z, c)$$

Using the continuity argument, we can replace the coend with the end:

$$\int_z \mathbf{Set}(\mathbf{C}(z, a) \times F\ z, c)$$

We can now take advantage of the adjunction between the product and the exponential:

$$\int_z \mathbf{Set}(\mathbf{C}(z, a), c^{(F\ z)})$$

We can "perform the integration" by using the Yoneda lemma to get:

$$c^{(F\ a)}$$

(Notice that we used the contravariant version of the Yoneda lemma, since the functor $c^{(Fz)}$ is contravariant in z.) This exponential object is isomorphic to the hom-set:

$$\mathbf{Set}(F\ a, c)$$

Finally, we take advantage of the Yoneda embedding to arrive at the isomorphism:

$$\int^z \mathbf{C}(z, a) \times F\ z \cong F\ a$$

26.7 Profunctor Composition

Let's explore further the idea that a profunctor describes a relation — more precisely, a proof-relevant relation, meaning that the set $p\ a\ b$ represents the set of proofs that a is related to b. If we have two relations p and q we can try to compose them. We'll say that a is related to b through the composition of q after p if there exist an intermediary object c such that both $q\ b\ c$ and $p\ c\ a$ are non-empty. The proofs of this new relation are all pairs of proofs of individual relations. Therefore, with the understanding that the existential quantifier corresponds to a coend, and the Cartesian product of two sets corresponds to "pairs of proofs," we can define composition of profunctors using the following formula:

$$(q \circ p)\, a\, b = \int^{c} p\, c\, a \times q\, b\, c$$

Here's the equivalent Haskell definition from `Data.Profunctor.Composition`, after some renaming:

```haskell
data Procompose q p a b where
    Procompose :: q a c -> p c b -> Procompose q p a b
```

This is using generalized algebraic data type, or GADT syntax, in which a free type variable (here `c`) is automatically existentially quantified. The (uncurried) data constructor `Procompose` is thus equivalent to:

```
exists c. (q a c, p c b)
```

The unit of so defined composition is the hom-functor — this immediately follows from the Ninja Yoneda lemma. It makes sense, therefore, to ask the question if there is a category in which profunctors serve as morphisms. The answer is positive, with the caveat that both associativity and identity laws for profunctor composition hold only up to natural isomorphism. Such a category, where laws are valid up to isomorphism, is called a bicategory (which is more general than a 2-category). So we have a bicategory **Prof**, in which objects are categories, morphisms are profunctors, and morphisms between morphisms (a.k.a., two-cells) are natural transformations. In fact, one can go even further, because beside profunctors, we also have regular functors as morphisms between categories. A category which has two types of morphisms is called a double category.

Profunctors play an important role in the Haskell lens library and in the arrow library.