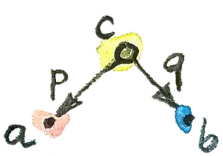
12

Limits and Colimits

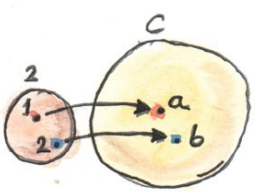
It seems like in category theory everything is related to everything and everything can be viewed from many angles. Take for instance the universal construction of the product. Now that we know more about functors and natural transformations, can we simplify and, possibly, generalize it? Let us try.



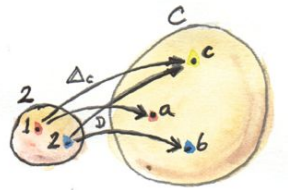
The construction of a product starts with the selection of two objects a and b, whose product we want to construct. But what does it mean to *select objects*? Can we rephrase this action in more categorical terms? Two objects form a pattern — a very simple pattern. We can abstract this pattern into a category — a very simple category, but a category

nevertheless. It's a category that we'll call **2**. It contains just two objects, 1 and 2, and no morphisms other than the two obligatory identities. Now we can rephrase the selection of two objects in **C** as the act of defining a functor D from the category **2** to **C**. A functor maps objects to objects, so its image is just two objects (or it could be one, if the functor collapses objects, which is fine too). It also maps morphisms — in this case it simply maps identity morphisms to identity morphisms.

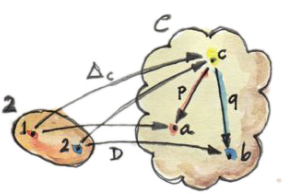
What's great about this approach is that it builds on categorical notions, eschewing the imprecise descriptions like "selecting objects," taken straight from the hunter-gatherer lexicon of our ancestors. And, incidentally, it is also easily generalized, because nothing can stop us from using categories more complex than **2** to define our patterns.



But let's continue. The next step in the definition of a product is the selection of the candidate object c. Here again, we could rephrase the selection in terms of a functor from a singleton category. And indeed, if we were using Kan extensions, that would be the right thing to do. But since we are not ready for Kan extensions yet, there is another trick we can use: a constant functor Δ from the same category **2** to **C**. The selection of c in **C** can be done with Δ_c. Remember, Δ_c maps all objects into c and all morphisms into $\mathbf{id}_c$.



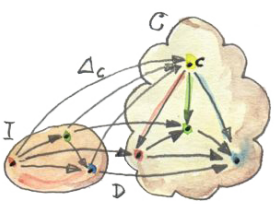
Now we have two functors, Δ_c and D going between **2** and **C** so it's only natural to ask about natural transformations between them. Since there are only two objects in **2**, a natural transformation will have two components. Object 1 in **2** is mapped to c by Δ_c and to a by D. So the component of a natural transformation between Δ_c and D at 1 is a morphism from c to a. We can call it p. Similarly, the second component is a morphism q from c to b — the image of the object 2 in **2** under D. But these are exactly like the two projections we used in our original definition of the product. So instead of talking about selecting objects and projections, we can just talk about picking functors and natural transformations. It so happens that in this simple case the naturality condition for our transformation is trivially satisfied, because there are no morphisms (other than the identities) in **2**.



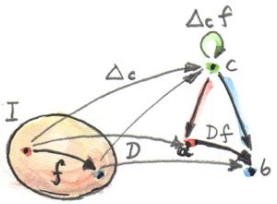
A generalization of this construction to categories other than **2** — ones that, for instance, contain non-trivial morphisms — will impose naturality conditions on the transformation between Δ_c and D. We call

such a transformation a *cone*, because the image of Δ is the apex of a cone/pyramid whose sides are formed by the components of the natural transformation. The image of D forms the base of the cone.

In general, to build a cone, we start with a category **I** that defines the pattern. It's a small, often finite category. We pick a functor D from **I** to **C** and call it (or its image) a *diagram*. We pick some c in **C** as the apex of our cone. We use it to define the constant functor Δ_c from **I** to **C**. A natural transformation from Δ_c to D is then our cone. For a finite **I** it's just a bunch of morphisms connecting c to the diagram: the image of **I** under D.



Naturality requires that all triangles (the walls of the pyramid) in this diagram commute. Indeed, take any morphism f in **I**. The functor D maps it to a morphism Df in **C**, a morphism that forms the base of some triangle. The constant functor Δ_c maps f to the identity morphism on c. Δ squishes the two ends of the morphism into one object, and the naturality square becomes a commuting triangle. The two arms of this triangle are the components of the natural transformation.

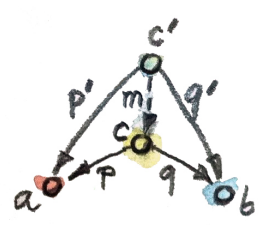


So that's one cone. What we are interested in is the *universal cone* — just like we picked a universal object for our definition of a product.

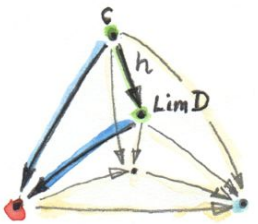
There are many ways to go about it. For instance, we may define a *category of cones* based on a given functor D. Objects in that category are cones. Not every object c in $\mathbf{C}$ can be an apex of a cone, though, because there may be no natural transformation between Δ_c and D.

To make it a category, we also have to define morphisms between cones. These would be fully determined by morphisms between their apexes. But not just any morphism will do. Remember that, in our construction of the product, we imposed the condition that the morphisms between candidate objects (the apexes) must be common factors for the projections. For instance:

```
p' = p . m
q' = q . m
```



This condition translates, in the general case, to the condition that the triangles whose one side is the factorizing morphism all commute.



The commuting triangle connecting two cones, with the factorizing morphism *h* (here, the lower cone is the universal one, with **Lim***D* as its apex)

We'll take those factorizing morphisms as the morphisms in our category of cones. It's easy to check that those morphisms indeed compose, and that the identity morphism is a factorizing morphism as well. Cones therefore form a category.

Now we can define the universal cone as the *terminal object* in the category of cones. The definition of the terminal object states that there is a unique morphism from any other object to that object. In our case it means that there is a unique factorizing morphism from the apex of any other cone to the apex of the universal cone. We call this universal cone the *limit* of the diagram *D*, **Lim***D* (in the literature, you'll often see a left arrow pointing towards *I* under the **Lim** sign). Often, as a shorthand, we call the apex of this cone the limit (or the limit object).

The intuition is that the limit embodies the properties of the whole diagram in a single object. For instance, the limit of our two-object diagram is the product of two objects. The product (together with the two projections) contains the information about both objects. And being universal means that it has no extraneous junk.

12.1 Limit as a Natural Isomorphism

There is still something unsatisfying about this definition of a limit. I mean, it's workable, but we still have this commutativity condition for the triangles that are linking any two cones. It would be so much more elegant if we could replace it with some naturality condition. But how?

We are no longer dealing with one cone but with a whole collection (in fact, a category) of cones. If the limit exists (and — let's make it clear — there's no guarantee of that), one of those cones is the universal cone. For every other cone we have a unique factorizing morphism that maps its apex, let's call it c, to the apex of the universal cone, which we named $\mathbf{Lim}D$. (In fact, I can skip the word "other", because the identity morphism maps the universal cone to itself and it trivially factorizes through itself.) Let me repeat the important part: given any cone, there is a unique morphism of a special kind. We have a mapping of cones to special morphisms, and it's a one-to-one mapping.

This special morphism is a member of the hom-set $\mathbf{C}(c, \mathbf{Lim}D)$. The other members of this hom-set are less fortunate, in the sense that they don't factorize the mapping of the two cones. What we want is to be able to pick, for each c, one morphism from the set $\mathbf{C}(c, \mathbf{Lim}D)$ — a morphism that satisfies the particular commutativity condition. Does that sound like defining a natural transformation? It most certainly does!

But what are the functors that are related by this transformation?

One functor is the mapping of c to the set $\mathbf{C}(c, \mathbf{Lim}D)$. It's a functor from $\mathbf{C}$ to **Set** — it maps objects to sets. In fact it's a contravariant functor. Here's how we define its action on morphisms: Let's take a morphism f from c' to c:

$$f :: c' \to c$$

Our functor maps c' to the set $\mathbf{C}(c', \mathbf{Lim}D)$. To define the action of this functor on f (in other words, to lift f), we have to define the corresponding mapping between $\mathbf{C}(c, \mathbf{Lim}D)$ and $\mathbf{C}(c', \mathbf{Lim}D)$. So let's pick one element u of $\mathbf{C}(c, \mathbf{Lim}D)$ and see if we can map it to some element of $\mathbf{C}(c', \mathbf{Lim}D)$. An element of a hom-set is a morphism, so we have:

$$u :: c \to \mathbf{Lim}D$$

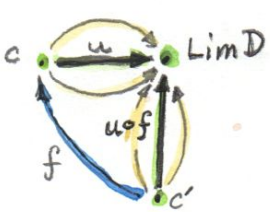
We can precompose u with f to get:

$$u.f :: c' \to \mathbf{Lim}D$$

And that's an element of $\mathbf{C}(c', \mathbf{Lim}D)$— so indeed, we have found a mapping of morphisms:

```
contramap :: (c' -> c) -> (c -> LimD) -> (c' -> LimD)
contramap f u = u . f
```

Notice the inversion in the order of c and c' characteristic of a *contravariant* functor.



To define a natural transformation, we need another functor that's also a mapping from **C** to **Set**. But this time we'll consider a set of cones. Cones are just natural transformations, so we are looking at the set of natural

transformations $Nat(\Delta_c, D)$. The mapping from $\mathbf{c}$ to this particular set of natural transformations is a (contravariant) functor. How can we show that? Again, let's define its action on a morphism:

$$f :: c' \to c$$

The lifting of f should be a mapping of natural transformations between two functors that go from **I** to **C**:

$$Nat(\Delta_c, D) \to Nat(\Delta_{c'}, D)$$

How do we map natural transformations? Every natural transformation is a selection of morphisms — its components — one morphism per element of **I**. A component of some α (a member of $Nat(\Delta_c, D)$) at a (an object in **I**) is a morphism:

$$\alpha_a :: \Delta_c a \to Da$$

or, using the definition of the constant functor Δ,

$$\alpha_a :: c \to Da$$

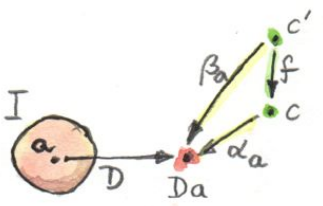
Given f and α, we have to construct a β, a member of $Nat(\Delta_{c'}, D)$. Its component at a should be a morphism:

$$\beta_a :: c' \to Da$$

We can easily get the latter (β_a) from the former (α_a) by precomposing it with f:

$$\beta_a = \alpha_a . f$$

It's relatively easy to show that those components indeed add up to a natural transformation.



Given our morphism f, we have thus built a mapping between two natural transformations, component-wise. This mapping defines `contramap` for the functor:

$$c \to Nat(\Delta_c, D)$$

What I have just done is to show you that we have two (contravariant) functors from **C** to **Set**. I haven't made any assumptions — these functors always exist.

Incidentally, the first of these functors plays an important role in category theory, and we'll see it again when we talk about Yoneda's lemma. There is a name for contravariant functors from any category **C** to **Set**: they are called "presheaves". This one is called a *representable presheaf*. The second functor is also a presheaf.

Now that we have two functors, we can talk about natural transformations between them. So without further ado, here's the conclusion: A functor D from **I** to **C** has a limit $\mathbf{Lim}D$ if and only if there is a natural isomorphism between the two functors I have just defined:

$$\mathbf{C}(c, \mathbf{Lim}D) \simeq Nat(\Delta_c, D)$$

Let me remind you what a natural isomorphism is. It's a natural transformation whose every component is an isomorphism, that is to say an invertible morphism.

I'm not going to go through the proof of this statement. The procedure is pretty straightforward if not tedious. When dealing with natural transformations, you usually focus on components, which are morphisms. In this case, since the target of both functors is **Set**, the components of the natural isomorphism will be functions. These are higher order functions, because they go from the hom-set to the set of natural transformations. Again, you can analyze a function by considering what it does to its argument: here the argument will be a morphism — a member of $\mathbf{C}(c, \mathbf{Lim}D)$ — and the result will be a natural transformation — a member of $Nat(\Delta_c, D)$, or what we have called a cone. This natural transformation, in turn, has its own components, which are morphisms. So it's morphisms all the way down, and if you can keep track of them, you can prove the statement.

The most important result is that the naturality condition for this isomorphism is exactly the commutativity condition for the mapping of cones.

As a preview of coming attractions, let me mention that the set $Nat(\Delta_c, D)$ can be thought of as a hom-set in the functor category; so our natural isomorphism relates two hom-sets, which points at an even more general relationship called an adjunction.

12.2 Examples of Limits

We've seen that the categorical product is a limit of a diagram generated by a simple category we called **2**.

There is an even simpler example of a limit: the terminal object. The first impulse would be to think of a singleton category as leading to a terminal object, but the truth is even starker than that: the terminal object is a limit generated by an empty category. A functor from an

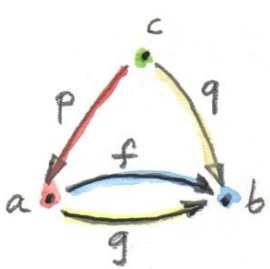
empty category selects no object, so a cone shrinks to just the apex. The universal cone is the lone apex that has a unique morphism coming to it from any other apex. You will recognize this as the definition of the terminal object.

The next interesting limit is called the *equalizer*. It's a limit generated by a two-element category with two parallel morphisms going between them (and, as always, the identity morphisms). This category selects a diagram in **C** consisting of two objects, *a* and *b*, and two morphisms:

```
f :: a -> b
g :: a -> b
```

To build a cone over this diagram, we have to add the apex, *c* and two projections:

```
p :: c -> a
q :: c -> b
```



We have two triangles that must commute:

```
q = f . p
q = g . p
```

This tells us that q is uniquely determined by one of these equations, say, `q = f . p`, and we can omit it from the picture. So we are left with just one condition:

```
f . p = g . p
```

The way to think about it is that, if we restrict our attention to **Set**, the image of the function p selects a subset of a. When restricted to this subset, the functions f and g are equal.

For instance, take a to be the two-dimensional plane parameterized by coordinates x and y. Take b to be the real line, and take:

```
f (x, y) = 2 * y + x
g (x, y) = y - x
```

The equalizer for these two functions is the set of real numbers (the apex, c) and the function:

```
p t = (t, (-2) * t)
```

Notice that $(p\ t)$ defines a straight line in the two-dimensional plane. Along this line, the two functions are equal.

Of course, there are other sets c' and functions p' that may lead to the equality:

```
f . p' = g . p'
```

but they all uniquely factor out through p. For instance, we can take the singleton set $()$ as c' and the function:

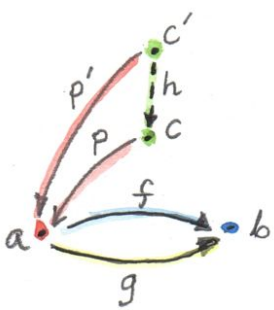
```
p'() = (0, 0)
```

It's a good cone, because $f(0,0) = g(0,0)$. But it's not universal, because of the unique factorization through h:

```
p' = p . h
```

with

```
h () = 0
```



An equalizer can thus be used to solve equations of the type $f\ x = g\ x$. But it's much more general, because it's defined in terms of objects and morphisms rather than algebraically.

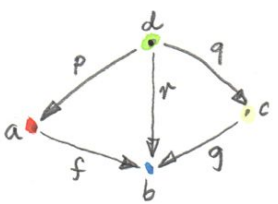
An even more general idea of solving an equation is embodied in another limit — the pullback. Here, we still have two morphisms that we want to equate, but this time their domains are different. We start with a three-object category of the shape: $1 \to 2 \leftarrow 3$. The diagram corresponding to this category consists of three objects, a, b, and c, and two morphisms:

```
f :: a -> b
g :: c -> b
```

This diagram is often called a *cospan*.

A cone built on top of this diagram consists of the apex, *d*, and three morphisms:

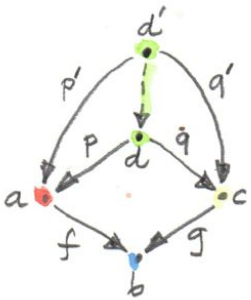
```
p :: d -> a
q :: d -> c
r :: d -> b
```



Commutativity conditions tell us that *r* is completely determined by the other morphisms, and can be omitted from the picture. So we are only left with the following condition:

```
g . q = f . p
```

A pullback is a universal cone of this shape.



Again, if you narrow your focus down to sets, you can think of the object d as consisting of pairs of elements from a and c for which f acting on the first component is equal to g acting on the second component. If this is still too general, consider the special case in which g is a constant function, say $g\ \_ = 1.23$ (assuming that b is a set of real numbers). Then you are really solving the equation:

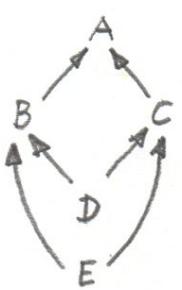
```
f x = 1.23
```

In this case, the choice of c is irrelevant (as long as it's not an empty set), so we can take it to be a singleton set. The set a could, for instance, be the set of three-dimensional vectors, and f the vector length. Then the pullback is the set of pairs $(v, ())$, where v is a vector of length 1.23 (a solution to the equation $\sqrt{(x^2 + y^2 + z^2)} = 1.23$), and $()$ is the dummy element of the singleton set.

But pullbacks have more general applications, also in programming. For instance, consider C++ classes as a category in which morphism are arrows that connect subclasses to superclasses. We'll consider inheritance a transitive property, so if `C` inherits from `B` and `B` inherits from `A` then we'll say that `C` inherits from `A` (after all, you can pass a pointer to `C`

where a pointer to `A` is expected). Also, we'll assume that `C` inherits from `C`, so we have the identity arrow for every class. This way subclassing is aligned with subtyping. C++ also supports multiple inheritance, so you can construct a diamond inheritance diagram with two classes `B` and `C` inheriting from `A`, and a fourth class `D` multiply inheriting from `B` and `C`. Normally, `D` would get two copies of `A`, which is rarely desirable; but you can use virtual inheritance to have just one copy of `A` in `D`.

What would it mean to have `D` be a pullback in this diagram? It would mean that any class `E` that multiply inherits from `B` and `C` is also a subclass of `D`. This is not directly expressible in C++, where subtyping is nominal (the C++ compiler wouldn't infer this kind of class relationship — it would require "duck typing"). But we could go outside of the subtyping relationship and instead ask whether a cast from `E` to `D` would be safe or not. This cast would be safe if `D` were the bare-bone combination of `B` and `C`, with no additional data and no overriding of methods. And, of course, there would be no pullback if there is a name conflict between some methods of `B` and `C`.



There's also a more advanced use of a pullback in type inference. There is often a need to *unify* types of two expressions. For instance, suppose that the compiler wants to infer the type of a function:

```
twice f x = f (f x)
```

It will assign preliminary types to all variables and sub-expressions. In particular, it will assign:

```
f       :: t0
x       :: t1
f x     :: t2
f (f x) :: t3
```

from which it will deduce that:

```
twice :: t0 -> t1 -> t3
```

It will also come up with a set of constraints resulting from the rules of function application:

```
t0 = t1 -> t2 -- because f is applied to x
t0 = t2 -> t3 -- because f is applied to (f x)
```

These constraints have to be unified by finding a set of types (or type variables) that, when substituted for the unknown types in both expressions, produce the same type. One such substitution is:

```
t1 = t2 = t3 = Int
twice :: (Int -> Int) -> Int -> Int
```

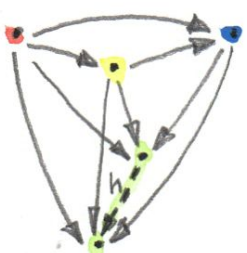
but, obviously, it's not the most general one. The most general substitution is obtained using a pullback. I won't go into the details, because they are beyond the scope of this book, but you can convince yourself that the result should be:

```
twice :: (t -> t) -> t -> t
```

with `t` a free type variable.

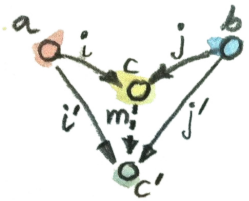
12.3 Colimits

Just like all constructions in category theory, limits have their dual image in opposite categories. When you invert the direction of all arrows in a cone, you get a co-cone, and the universal one of those is called a colimit. Notice that the inversion also affects the factorizing morphism, which now flows from the universal co-cone to any other co-cone.



Cocone with a factorizing morphism *h* connecting two apexes.

A typical example of a colimit is a coproduct, which corresponds to the diagram generated by **2**, the category we've used in the definition of the product.



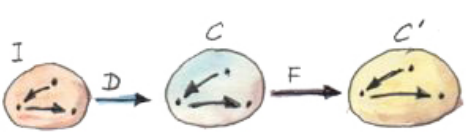
Both the product and the coproduct embody the essence of a pair of objects, each in a different way.

Just like the terminal object was a limit, so the initial object is a colimit corresponding to the diagram based on an empty category.

The dual of the pullback is called the *pushout*. It's based on a diagram called a span, generated by the category $1 \leftarrow 2 \rightarrow 3$.

12.4 Continuity

I said previously that functors come close to the idea of continuous mappings of categories, in the sense that they never break existing connections (morphisms). The actual definition of a *continuous functor* F from a category $\mathbf{C}$ to $\mathbf{C}'$ includes the requirement that the functor preserve limits. Every diagram D in $\mathbf{C}$ can be mapped to a diagram $F \circ D$ in $\mathbf{C}'$ by simply composing two functors. The continuity condition for F states that, if the diagram D has a limit $\mathbf{Lim}D$, then the diagram $F \circ D$ also has a limit, and it is equal to $F(\mathbf{Lim}D)$.



Notice that, because functors map morphisms to morphisms, and compositions to compositions, an image of a cone is always a cone. A commuting triangle is always mapped to a commuting triangle (functors preserve composition). The same is true for the factorizing morphisms: the image of a factorizing morphism is also a factorizing morphism. So every functor is *almost* continuous. What may go wrong is the uniqueness condition. The factorizing morphism in $\mathbf{C}'$ might not be unique. There may also be other "better cones" in $\mathbf{C}'$ that were not available in $\mathbf{C}$.

A hom-functor is an example of a continuous functor. Recall that the hom-functor, $\mathbf{C}(a, b)$, is contravariant in the first variable and covariant in the second. In other words, it's a functor:

$$\mathbf{C}^{op} \times \mathbf{C} \to \mathbf{Set}$$

When its second argument is fixed, the hom-set functor (which becomes the representable presheaf) maps colimits in $\mathbf{C}$ to limits in **Set**; and when its first argument is fixed, it maps limits to limits.

In Haskell, a hom-functor is the mapping of any two types to a function type, so it's just a parameterized function type. When we fix the second parameter, let's say to `String`, we get the contravariant functor:

```
newtype ToString a = ToString (a -> String)
instance Contravariant ToString where
    contramap f (ToString g) = ToString (g . f)
```

Continuity means that when `ToString` is applied to a colimit, for instance a coproduct `Either b c`, it will produce a limit; in this case a product of two function types:

```
ToString (Either b c) ~ (b -> String, c -> String)
```

Indeed, any function of `Either b c` is implemented as a case statement with the two cases being serviced by a pair of functions.

Similarly, when we fix the first argument of the hom-set, we get the familiar reader functor. Its continuity means that, for instance, any function returning a product is equivalent to a product of functions; in particular:

```
r -> (a, b) ~ (r -> a, r -> b)
```

I know what you're thinking: You don't need category theory to figure these things out. And you're right! Still, I find it amazing that such results can be derived from first principles with no recourse to bits and bytes, processor architectures, compiler technologies, or even lambda calculus.

If you're curious where the names "limit" and "continuity" come from, they are a generalization of the corresponding notions from calculus. In calculus limits and continuity are defined in terms of open neighborhoods. Open sets, which define topology, form a category (a poset).

12.5 Challenges

1. How would you describe a pushout in the category of C++ classes?
2. Show that the limit of the identity functor $\mathbf{Id} :: \mathbf{C} \to \mathbf{C}$ is the initial object.
3. Subsets of a given set form a category. A morphism in that category is defined to be an arrow connecting two sets if the first is

the subset of the second. What is a pullback of two sets in such a category? What's a pushout? What are the initial and terminal objects?

4. Can you guess what a coequalizer is?
5. Show that, in a category with a terminal object, a pullback towards the terminal object is a product.
6. Similarly, show that a pushout from an initial object (if one exists) is the coproduct.