# PROGRAMMING WITH UNIVERSAL CONSTRUCTIONS

BARTOSZ MILEWSKI

As functional programmers we are interested in functions. Category theorists are similarly interested in morphisms. There is a difference in approach, though. A programmer must implement a function, whereas a mathematician is often satisfied with the proof of existence of a morphism (unless said mathematician is a constructivist). Category theory if full of such proofs.

A lot of objects in category theory are defined using universal constructions and universality is used all over the place to show the existence (as a rule: unique, up to unique isomorphism) of morphisms between objects.

There are two major types of universal constructions: the ones asserting the mapping-out property, and the ones asserting the mapping-in property. For instance, the product has the *mapping-in* property. Recall that a product of two objects $a$ and $b$ is an object $a \times b$ together with two projections:

$$\pi_1 : a \times b \to a$$

$$\pi_2 : a \times b \to b$$

This object must satisfy the universal property: for any other object $c$ with a pair of morphisms:

$$f : c \to a$$

$$g : c \to b$$

there exists a unique morphism $h : c \to a \times b$ such that:

$$f = \pi_1 \circ h$$

$$g = \pi_2 \circ h$$

In other words, the two triangles in Fig 1 commute.

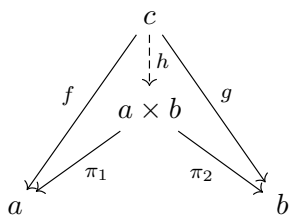


FIGURE 1. Universality of the product

This universal property can be used any time you need to find a morphism that's *mapping into* the product.

For instance, let's say you want to find a morphism from the terminal object 1 to $a \times b$. All you need is to define two morphisms $f : 1 \to a$ and $g : 1 \to b$. This is not always possible, but if it is, you are guaranteed to have a morphism $h : 1 \to a \times b$ (Fig 2).

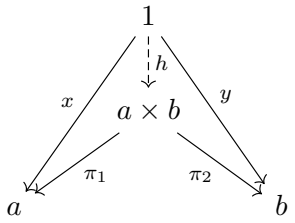


FIGURE 2. Global element of a product

Incidentally, morphisms from the terminal object are called *global elements*, so we have just shown that, as long as $a$ and $b$ have global elements, say $x$ and $y$, their product has a global element too. Moreover the projection $\pi_1$ of this global element is the same as $x$, and $\pi_2$ is the same as $y$. In other words, an element of a product is a pair of elements. But you probably knew that.

The universal construction of the product is implemented as an operator in Haskell:

```
(&&&) :: (c->a) -> (c->b) -> (c -> (a, b))
```

We can also go the other way: given a mapping-in $h : c \to a \times b$, we can always extract a pair of morphisms:

$$f = \pi_1 \circ h$$

$$g = \pi_2 \circ h$$

This bijection between $h$ and a pair of morphisms $(f, g)$ is in fact an *adjunction*.

You might think this kind of reasoning is very different from what programmers do, but it's not. Here's one possible definition of a product in Haskell (besides the built-in one, `(,)`):

```
data Product a b = MkProduct { fst :: a
                             , snd :: b }
```

It is in one-to-one correspondence with what I've just explained. The two functions `fst` and `snd` are $\pi_1$ and $\pi_2$, and `MkProduct` corresponds to our $h$. The categorical definition is just a different, and much more general, way of saying the same thing.

Here's another problem: Show that the product is functorial. Suppose that you have a pair of morphisms:

$$f : a \to a'$$

$$g : b \to b'$$

and you want to lift them to a morphism:

$$h : a \times b \to a' \times b'$$

Since we are dealing with products, we should use the mapping-in property. So we draw the universality diagram for the target $a' \times b'$, and put the source $a \times b$ at the top. The pair of functions that fits the bill is $(f \circ \pi_1, g \circ \pi_2)$ (Fig 3).

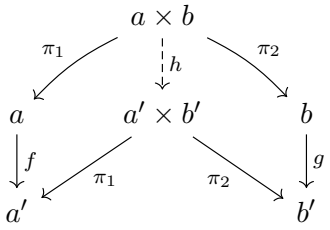


FIGURE 3. Functoriality of the product

The universal property gives us, uniquely, the $h$, which is usually written simply as $f \times g$.

*Exercise for the reader*: Show, using universality, that categorical product is symmetric.

The coproduct, being the dual of the product, is defined by the universal mapping-out property, see Fig 4.

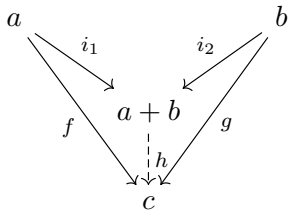


FIGURE 4. Universality of the coproduct

So if you need a morphism *from* a coproduct $a + b$ to some $c$, it's enough to define two morphisms:

$$f : a \to c$$

$$g : b \to c$$

This universal property may also be restated as the isomorphism between pairs of morphisms $(f, g)$ and morphisms of the type $a+b \to c$ (so there is a corresponding adjunction).

This is easily illustrated in Haskell:

```
h :: Either a b -> c
h (Left a)  = f a
h (Right b) = g b
```

Here `Left` and `Right` correspond to the two injections $i_1$ and $i_2$. There is a convenient function in Haskell that encapsulates this universal construction:

```
either :: (a->c) -> (b->c) -> (Either a b -> c)
```

*Exercise for the reader*: Show that coproduct is functorial.

So next time you ask yourself, what can I do with a universal construction? the answer is: use it to define a morphism, either mapping in or mapping out of your construct. Why is it useful? Because it decomposes a problem into smaller problems. In the examples above, the problem of constructing one morphism $h$ was nicely decomposed into defining $f$ and $g$ separately.

The flip side of this is that there is no simple way of defining a mapping *out* of a product or a mapping *into* a coproduct.

For instance, you might wonder if the familiar distributive law:

$$(a + b) \times c \cong a \times c + b \times c$$

holds in an arbitrary category that defines products and coproducts (so called bicartesian category). You can immediately see that defining a morphism from right to left is easy, because it involves mapping out of a coproduct. All we need is to define a pair of morphisms leading to the common target (Fig 5):

$$f : a \times c \to (a + b) \times c$$

$$g : b \times c \to (a + b) \times c$$

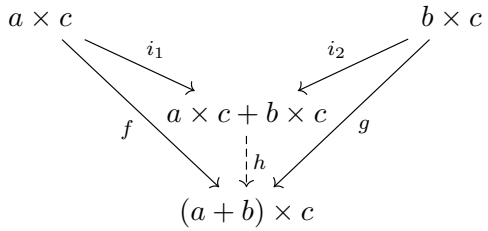


FIGURE 5. Right to left proof

The trick is to use the functoriality of the product, which we have already established, to implement $f$ and $g$ as:

$$f = i_1 \times id_c$$

$$g = i_2 \times id_c$$

But if you try to prove the other direction, from left to right, you're stuck, because it would require the mapping *out* of a product. So the distributive property does not hold in general.

"Wait a moment!" I hear you say, "I can easily implement it in Haskell."

```
f :: (Either a b, c) -> Either (a, c) (b, c)
f (Left a, c)  = Left  (a, c)
f (Right b, c) = Right (b, c)
```

That's correct, but Haskell does a little cheating behind the scenes. You can see it clearly when you convert this code to point free notation (I'll explain later how I figured it out):

```
f = uncurry (either (curry Left) (curry Right))
```

I want to direct your attention to the use of `uncurry` and `curry`. Currying is the application of another universal construction, namely that of the exponential object $c^b$, corresponding to the function type `b -> c`. This is exactly the construction that provides the missing *mapping out* of a product, `(a, b) -> c`:

```
uncurry :: (c -> (a -> b)) -> ((c, a) -> b)
```

Categorically, we have the bijection between morphisms (again, a sign of an adjunction):

$$h : c \to b^a$$

$$f : c \times a \to b$$

Universality tells us that for every $c$ and $f$ there is a unique $h$ in Fig 6 and vice versa. The arrow $h \times id_a$ is the lifting of the pair $(h, id_a)$ by the product functor (we've established the functoriality of the product earlier).

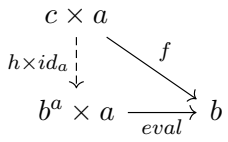


FIGURE 6. Universality of the exponential

Not every category has exponentials–the ones that do are called cartesian closed (cartesian, because they must also have products).

So how does the fact that in Haskell we have exponentials help us here? We are trying to define a morphism $f$ as a mapping out of a product:

$$f : (a + b) \times c \to a \times c + b \times c$$

This mapping exists if we can define another mapping:

$$h : (a + b) \to (a \times c + b \times c)^c$$

see Fig 7.

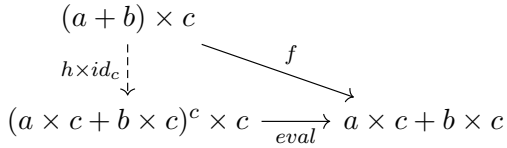


FIGURE 7. Uncurrying

This morphism is easier to define, since it involves a mapping out of a sum. We just need a pair of morphisms:

$$h_1 : a \to (a \times c + b \times c)^c$$
$$h_2 : b \to (a \times c + b \times c)^c$$

We can define the first morphism using the universal property of the exponential and picking the injection $i_1$ as the uncurried morphism:

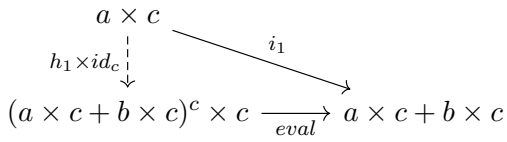


FIGURE 8. Defining $h_1$

and similarly for $h_2$.

It's possible to combine all these diagrams into one point-free definition, and that's exactly how I came up with the original code:

```
f = uncurry (either (curry Left) (curry Right))
```

Notice that `curry` is used to get from $f$ to $h$, and `uncurry` from $h$ to $f$ in the original diagram.

Products and coproducts are examples of more general constructions called limits and colimits. Importantly, the universal property of limits can be used to define mapping-in morphisms, whereas the universal property of colimits allows us to define mapping-out morphisms. I'll talk more about it in the upcoming post.