Chapter 5

Product Types

We can use sum types to enumerate possible values of a given type, but the encoding can be wasteful. We needed ten constructors just to encode numbers between zero and nine.

```
data Digit = Zero | One | Two | Three | ... | Nine
```

But if we combine two digits into a single data structure, a two-digit decimal number, we'll able to encode a hundred numbers. Or, as Lao Tzu would say, with just four digits you can encode ten thousand numbers.

A data type that combines two types in this manner is called a product. Its defining quality is the elimination rule: there are two arrows coming from $A \times B$; one called "fst" goes to A, and another called "snd" goes to B. They are called *projections*. They let us retrive A and B from the product $A \times B$.

$$\begin{tikzcd} & A \times B \arrow[dl, "\text{fst}"'] \arrow[dr, "\text{snd}"] & \\ A & & B \end{tikzcd}$$

Suppose that somebody gave you an element of a product, that is an arrow h from the terminal object 1 to $A \times B$. You can easily retrieve a pair of elements, just by using composition: an element of A given by

$$a = \text{fst} \circ h$$

and an element of B given by

$$b = \text{snd} \circ h$$

$$\begin{tikzcd} & 1 \arrow[d, dashed, "h"] \arrow[ddl, bend right, "a"'] \arrow[ddr, bend left, "b"] & \\ & A \times B \arrow[dl, "\text{fst}"'] \arrow[dr, "\text{snd}"] & \\ A & & B \end{tikzcd}$$

In fact, given an arrow from an arbitrary object C to $A \times B$, we can define, by composition, a pair of arrows $f \colon C \to A$ and $g \colon C \to B$

$$\begin{tikzcd} & C \arrow[dl, bend right, "f"'] \arrow[d, dashed, "h"] \arrow[dr, bend left, "g"] & \\ & A \times B \arrow[dl, "\text{fst}"] \arrow[dr, "\text{snd}"'] & \\ A & & B \end{tikzcd}$$

As we did before with the sum type, we can turn this idea around, and use this diagram to define the product type: A pair of functions f and g should be in one-to-one correspondence with a *mapping in* from C to $A \times B$. This is the *introduction* rule for the product.

In particular, the mapping out of the terminal object is used in Haskell to define a product type. Given two elements, `a :: A` and `b :: B`, we construct the product

```
(a, b) :: (A, B)
```

The built-in syntax for products is just that: a pair of parentheses and a comma in between. It works both for defining the product of two types `(A, B)` and the data constructor `(a, b)` that takes two elements and pairs them together.

We should never lose sight of the purpose of programming: to decompose complex problems into a series of simpler ones. We see it again in the definition of the product. Whenever we have to construct a mapping *into* the product, we decompose it into two smaller tasks of constructing a pair of functions, each mapping into one of the components of the product. This is as simple as saying that, in order to implement a function that returns a pair of values, it's enough to implement two functions, each returning one of the elements of the pair.

Logic

In logic, a product type corresponds to logical conjunction. In order to prove $A \times B$ (A and B), you need to provide the proofs of *both* A and B. These are the arrows targeting A and B. The elimination rule says that if you have a proof of $A \times B$, then you can get the proof of A (through fst) and the proof of B (through snd).

Tuples and Records

As Lao Tzu would say, a product of ten thousand objects is just an object with ten thousand projections.

We can form such products in Haskell using the tuple notation. For instance, a product of three types is written as `(A, B, C)`. A term of this type can be constructed from three elements: `(a, b, c)`.

In what mathematicians call "abuse of notation", a product of zero types is written as `()`, an empty tuple, which happens to be the same as the terminal object, or unit type. This is because the product behaves very much like multiplication of numbers, with the terminal object playing the role of unit.

In Haskell, rather than defining separate projections for all tuples, we use the pattern-matching syntax. For instance, to extract the third component from a triple we would write

```haskell
thrd :: (a, b, c) -> c
thrd (_, _, c) = c
```

We use wildcards for the components that we want to ignore.

Lao Tzu said that “Naming is the origin of all particular things.” In programming, keeping track of the meaning of the components of a particular tuple is difficult without giving them names. Record syntax allows us to give names to projections. This is the definition of a product written in record style:

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

`Pair` is the data constructor and `fst` and `snd` are the projections.

This is how it could be used to declare and initialize a particular pair:

```haskell
ic :: Product Int Char
ic = Pair 10 'A'
```

5.1 Cartesian Category

In Haskell, we can define a product of any two types. A category in which all products exist, and the terminal object exists, is called *cartesian*.

Tuple Arithmetic

The identities satisfied by the product can be derived using the mapping-in property. For instance, to show that $A \times B \cong B \times A$ consider the following two diagrams:

$$\begin{tikzcd}
& X \arrow[d, dashed, "h"] \arrow[ddl, bend right, "f"'] \arrow[ddr, bend left, "g"] & \\
& A \times B \arrow[dl, "\text{fst}"] \arrow[dr, "\text{snd}"'] & \\
A & & B
\end{tikzcd}
\qquad
\begin{tikzcd}
& X \arrow[d, dashed, "h'"] \arrow[ddl, bend right, "g"'] \arrow[ddr, bend left, "f"] & \\
& B \times A \arrow[dl, "\text{fst}"] \arrow[dr, "\text{snd}"'] & \\
B & & A
\end{tikzcd}$$

They show that, for any object X the arrows to $A \times B$ are in one-to-one correspondence with arrows to $B \times A$. This is because each of these arrows is determined by the same pair f and g.

You can check that the naturality condition is satisfied because, when you shift the perspective using $k \colon X' \to X$, all arrows originating in X are shifted by pre-composition $(- \circ k)$.

In Haskell, this isomorphism can be implemented as a function which is its own inverse:

```
swap :: (a, b) -> (b, a)
swap x = (snd x, fst x)
```

This is the same function written using pattern matching:

```
swap (x, y) = (y, x)
```

It's important to keep in mind that the product is symmetric only "up to isomorphism." It doesn't mean that swapping the order of pairs won't change the behavior of a program. Symmetry means that the information content of a swapped pair is the same, but access to it needs to be modified.

Here's the diagram that can be used to prove that the terminal object is the unit of the product, $1 \times A \cong A$.

$$\begin{tikzcd}
& X \arrow[ddl, bend right, "!"'] \arrow[d, dashed, "h"] \arrow[ddr, bend left, "f"] & & X \arrow[dd, "f"] \\
& 1 \times A \arrow[dl, "\text{fst}"'] \arrow[dr, "\text{snd}"] & & \\
1 & & A & A
\end{tikzcd}$$

The unique arrow from X to 1 is called ! (pronounced, *bang*). Because of its uniqueness, the mapping-in, h, is totally determined by f.

The invertible arrow that witnesses the isomorphism between $1 \times A$ and A is called the *left unitor*:

$$\lambda \colon 1 \times A \to A$$

Here are some other isomorphisms written in Haskell (without proofs of having the inverse). This is associativity:

```
assoc :: ((a, b), c) -> (a, (b, c))
assoc ((a, b), c) = (a, (b, c))
```

And this is the left unit

```
lunit :: ((), a) -> a
lunit (_, a) = a
```

These functions correspond to the *associator*

$$\alpha \colon (A \times B) \times C \to A \times (B \times C)$$

and the *right unitor*:

$$\rho \colon A \times 1 \to A$$

Exercise 5.1.1. *Show that the bijection in the proof of left unit is natural. Hint, change focus using an arrow $g \colon A \to B$.*

Exercise 5.1.2. *Construct an arrow*

$$h\colon B + A \times B \to (1 + A) \times B$$

Is this arrow unique?

Hint: It's a mapping into a product, so it's given by a pair of arrow. These arrows, in turn, map out of a sum, so each is given by a pair of arrows.

Hint: The mapping $B \to 1 + A$ is given by $(\mathit{Left} \circ !)$

Exercise 5.1.3. *Re-do the previous exercise, this time treating h as a mapping* out of *a sum.*

Exercise 5.1.4. *Implement a Haskell function* `maybeAB :: Either b (a, b) -> (Maybe a, b)`. *Is this function uniquely defined by its type signature or is there some leeway?*

Functoriality

Suppose that we have arrows that map A and B to some A' and B':

$$f\colon A \to A'$$

$$g\colon B \to B'$$

The composition of these arrows with the projections fst and snd, respectively, can be used to define the mapping of the products:

$$\begin{tikzcd}
 & A \times B \arrow[ld, "\text{fst}"'] \arrow[d, dashed, "h"] \arrow[rd, "\text{snd}"] & \\
A \arrow[d, "f"'] & A' \times B' \arrow[ld, "\text{fst}"] \arrow[rd, "\text{snd}"'] & B \arrow[d, "g"] \\
A' & & B'
\end{tikzcd}$$

The shorthand notation for this diagram is:

$$A \times B \xrightarrow{f \times g} A' \times B'$$

This property of the product is called *functoriality*. You can imagine it as allowing you to transform the two objects *inside* the product.

5.2 Duality

When a child sees an arrow, it knows which end points at the source, and which points at the target

$$A \to B$$

But maybe this is just a pre-conception. Would the Universe be very different if we called B the source and A the target?

We would still be able to compose this arrow with this one

$$B \to C$$

whose "target" B is the same as the same as the "source" of $A \to B$, and the result would still be an arrow

$$A \to C$$

only now we would say that it goes from C to A.

In this dual Universe, the object that we call "initial" would be called "terminal," because it's the "target" of unique arrows coming from all objects. Conversely, the terminal object would be called initial.

Now consider this diagram that we used to define the sum object:

$$\begin{tikzcd}
A \arrow[dr, bend left, "\text{Left}"'] \arrow[ddr, bend right, "f"'] & & B \arrow[dl, bend right, "\text{Right}"] \arrow[ddl, bend left, "g"] \\
& A+B \arrow[d, dashed, "h"] & \\
& C &
\end{tikzcd}$$

In the new interpretation, the arrow h maps "from" an arbitrary object C "to" the object we call $A + B$. This arrow is uniquely defined by a pair of arrows (f, g) whose "source" is C. If we rename Left to fst and Right to snd, we will get the defining diagram for a product.

A product is the sum with arrows reversed.

Conversely, a sum is the product with arrows reversed.

Every construction in category theory has its dual.

If the direction of arrows is just a matter of interpretation, then what makes sum types so different from product types in programming? The difference goes back to one assumption we made at the start: There are no incoming arrows to the initial object (other than the identity arrow). This is in contrast with the terminal object having lots of outgoing arrows, which we used to define (global) elements. In fact, we assume that every object of interest has elements, and the ones that don't are isomorphic to `Void`.

We'll see an even deeper difference when we talk about function types.

5.3 Monoidal Category

We have seen that the product satisfies these simple rules:

$$1 \times A \cong A$$

$$A \times B \cong B \times A$$

$$(A \times B) \times C \cong A \times (B \times C)$$

and is functorial.

A category in which an operation with these properties is defined is called *symmetric monoidal*. We've seen this structure before when working with sums and the initial object.

A category can have multiple monoidal structures at the same time. When you don't want to name your monoidal structure, you replace the plus sign or the product sign with a tensor sign, and the neutral element with the letter I. The rules of a symmetric monoidal category can then be written as:

$$I \otimes A \cong A$$

$$A \otimes B \cong B \otimes A$$

$$(A \otimes B) \otimes C \cong A \otimes (B \otimes C)$$

You may think of a tensor product as the lowest common denominator of product and sum. It still has an introduction rule, but it requires both objects A and B; and it has no elimination rule—no projections. Some interesting examples of tensor products are not even symmetric.

Monoids

Monoids are very simple structures equipped with a binary operation and a unit. Natural numbers with addition and zero form a monoid. So do natural numbers with multiplication and one.

The intuition is that a monoid lets you combine two things to get one thing. There is also one special thing, such that combining it with anything else gives back the same thing. And the combining must be associative.

What's not assumed is that the combining is symmetric, or that there is an inverse element.

The rules that define a monoid are reminiscent of the rules of a category. The difference is that, in a monoid, any two things are composable, whereas in a category this is usually not the case: You can only compose two arrows if the target of one is the source of another. Except, that is, when the category contains only one object, in which case all arrows are composable.

A category with a single object is called a monoid. The combining operation is the composition of arrows and the unit is the identity arrow.

This is a perfectly valid definition. In practice, however, we are often interested in monoids that are embedded in larger categories. In particular, in programming, we want to be able to define monoids inside the category of types and functions.

In a category, we are forced to define the operation in bulk, rather than looking at individual elements. We start with an object M. A binary operation is a function of two arguments. Since elements of a product are pairs of elements, we can generalize it to an arrow from a product $M \times M$ to M:

$$\mu \colon M \times M \to M$$

The unit element can be defined as an arrow from the terminal object 1:

$$\eta \colon 1 \to M$$

We can translate this description directly to Haskell by defining a class of types equipped with two methods, traditionally called `mappend` and `mempty`

```haskell
class Monoid m where
  mappend :: (m, m) -> m
  mempty  :: () -> m
```

The two arrows μ and η have to satisfy monoid laws but, again, we have to formulate them in bulk, without any recourse to elements.

To formulate the left unit law, we first create the product $1 \times M$. We then use η to "pick the unit element in M" or, in terms of arrows, turn 1 into M. Since we are operating on a product $1 \times M$, we lift the pair $\langle \eta, id_M \rangle$, which ensures that we "do not touch" the M. We then perform the "multiplication" using μ.

We want the result to be the same as the original element of M, but without mentioning elements. So we just use the left unitor λ to go from $1 \times M$ to M without "stirring things up."

$$\begin{tikzcd}
1 \times M \arrow[r, "\eta \times id_M"] \arrow[rd, "\lambda"'] & M \times M \arrow[d, "\mu"] \\
 & M
\end{tikzcd}$$

Here is the analogous law for the right unit:

$$\begin{tikzcd}
M \times M \arrow[d, "\mu"] & M \times 1 \arrow[l, "id_M \times \eta"'] \arrow[ld, "\rho"] \\
M &
\end{tikzcd}$$

To formulate the law of associativity, we have to start with a triple product and act on it in bulk. Here, α is the associator that rearranges the product without "stirring things up."

$$\begin{tikzcd}
(M \times M) \times M \arrow[rr, "\alpha"] \arrow[d, "\mu \times id"] & & M \times (M \times M) \arrow[d, "id \times \mu"] \\
M \times M \arrow[rd, "\mu"] & & M \times M \arrow[ld, "\mu"'] \\
 & M &
\end{tikzcd}$$

Notice that we didn't have to assume a lot about the categorical product that we used on the objects M and 1. In particular we never had to use projections. This suggests that the above definition will work equally well for a tensor product in a monoidal category. It doesn't even have to be symmetric. All we have to assume is that there is a unit object, that the product is functorial, and that it satisfies the unit and associativity laws up to isomorphism.

Thus if we replace $\times$ with $\otimes$ and 1 with I, we get a definition of a monoid in an arbitrary monoidal category.