

# 6

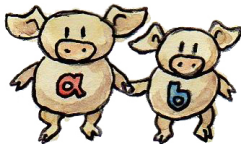
## Simple Algebraic Data Types

**W**E'VE SEEN TWO BASIC ways of combining types: using a product and a coproduct. It turns out that a lot of data structures in everyday programming can be built using just these two mechanisms. This fact has important practical consequences. Many properties of data structures are composable. For instance, if you know how to compare values of basic types for equality, and you know how to generalize these comparisons to product and coproduct types, you can automate the derivation of equality operators for composite types. In Haskell you can automatically derive equality, comparison, conversion to and from string, and more, for a large subset of composite types.

Let's have a closer look at product and sum types as they appear in programming.

## 6.1 Product Types

The canonical implementation of a product of two types in a programming language is a pair. In Haskell, a pair is a primitive type constructor; in C++ it's a relatively complex template defined in the Standard Library.



Pairs are not strictly commutative: a pair `(Int, Bool)` cannot be substituted for a pair `(Bool, Int)`, even though they carry the same information. They are, however, commutative up to isomorphism — the isomorphism being given by the `swap` function (which is its own inverse):

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

You can think of the two pairs as simply using a different format for storing the same data. It's just like big endian vs. little endian.

You can combine an arbitrary number of types into a product by nesting pairs inside pairs, but there is an easier way: nested pairs are equivalent to tuples. It's the consequence of the fact that different ways of nesting pairs are isomorphic. If you want to combine three types in a product, `a`, `b`, and `c`, in this order, you can do it in two ways:

```
((a, b), c)
```

or

```
(a, (b, c))
```

These types are different — you can't pass one to a function that expects the other — but their elements are in one-to-one correspondence. There is a function that maps one to another:

```
alpha :: ((a, b), c) -> (a, (b, c))  
alpha ((x, y), z) = (x, (y, z))
```

and this function is invertible:

```
alpha_inv :: (a, (b, c)) -> ((a, b), c)  
alpha_inv (x, (y, z)) = ((x, y), z)
```

so it's an isomorphism. These are just different ways of repackaging the same data.

You can interpret the creation of a product type as a binary operation on types. From that perspective, the above isomorphism looks very much like the associativity law we've seen in monoids:

$$(a * b) * c = a * (b * c)$$

Except that, in the monoid case, the two ways of composing products were equal, whereas here they are only equal “up to isomorphism.”

If we can live with isomorphisms, and don't insist on strict equality, we can go even further and show that the unit type, `()`, is the unit of the product the same way 1 is the unit of multiplication. Indeed, the pairing of a value of some type `a` with a unit doesn't add any information. The type:

```
(a, ())
```

is isomorphic to `a`. Here's the isomorphism:

```
rho :: (a, ()) -> a
rho (x, ()) = x
```

```
rho_inv :: a -> (a, ())
rho_inv x = (x, ())
```

These observations can be formalized by saying that **Set** (the category of sets) is a *monoidal category*. It's a category that's also a monoid, in the sense that you can multiply objects (here, take their Cartesian product). I'll talk more about monoidal categories, and give the full definition in the future.

There is a more general way of defining product types in Haskell — especially, as we'll see soon, when they are combined with sum types. It uses named constructors with multiple arguments. A pair, for instance, can be defined alternatively as:

```
data Pair a b = P a b
```

Here, `Pair a b` is the name of the type parameterized by two other types, `a` and `b`; and `P` is the name of the data constructor. You define a pair type by passing two types to the `Pair` type constructor. You construct a pair value by passing two values of appropriate types to the constructor `P`. For instance, let's define a value `stmt` as a pair of `String` and `Bool`:

```
stmt :: Pair String Bool
stmt = P "This statement is" False
```

The first line is the type declaration. It uses the type constructor **Pair**, with **String** and **Bool** replacing **a** and the **b** in the generic definition of **Pair**. The second line defines the actual value by passing a concrete string and a concrete Boolean to the data constructor **P**. Type constructors are used to construct types; data constructors, to construct values.

Since the name spaces for type and data constructors are separate in Haskell, you will often see the same name used for both, as in:

```
data Pair a b = Pair a b
```

And if you squint hard enough, you may even view the built-in pair type as a variation on this kind of declaration, where the name **Pair** is replaced with the binary operator **(,)**. In fact you can use **(,)** just like any other named constructor and create pairs using prefix notation:

```
stmt = (,) "This statement is" False
```

Similarly, you can use **(,,)** to create triples, and so on.

Instead of using generic pairs or tuples, you can also define specific named product types, as in:

```
data Stmt = Stmt String Bool
```

which is just a product of **String** and **Bool**, but it's given its own name and constructor. The advantage of this style of declaration is that you may define many types that have the same content but different meaning and functionality, and which cannot be substituted for each other.

Programming with tuples and multi-argument constructors can get messy and error prone — keeping track of which component represents what. It's often preferable to give names to components. A product type with named fields is called a *record* in Haskell, and a **struct** in C.

## 6.2 Records

Let's have a look at a simple example. We want to describe chemical elements by combining two strings, name and symbol; and an integer, the atomic number; into one data structure. We can use a tuple (**String**, **String**, **Int**) and remember which component represents what. We would extract components by pattern matching, as in this function that checks if the symbol of the element is the prefix of its name (as in **He** being the prefix of **Helium**):

```
startsWithSymbol :: (String, String, Int) -> Bool
startsWithSymbol (name, symbol, _) = isPrefixOf symbol name
```

This code is error prone, and is hard to read and maintain. It's much better to define a record:

```
data Element = Element { name :: String
                        , symbol :: String
                        , atomicNumber :: Int }
```

The two representations are isomorphic, as witnessed by these two conversion functions, which are the inverse of each other:

```
tupleToElem :: (String, String, Int) -> Element
tupleToElem (n, s, a) = Element { name = n
                                , symbol = s
```

```
, atomicNumber = a }
```

```
elemToTuple :: Element -> (String, String, Int)
elemToTuple e = (name e, symbol e, atomicNumber e)
```

Notice that the names of record fields also serve as functions to access these fields. For instance, `atomicNumber e` retrieves the `atomicNumber` field from `e`. We use `atomicNumber` as a function of the type:

```
atomicNumber :: Element -> Int
```

With the record syntax for `Element`, our function `startsWithSymbol` becomes more readable:

```
startsWithSymbol :: Element -> Bool
startsWithSymbol e = isPrefixOf (symbol e) (name e)
```

We could even use the Haskell trick of turning the function `isPrefixOf` into an infix operator by surrounding it with backquotes, and make it read almost like a sentence:

```
startsWithSymbol e = symbol e `isPrefixOf` name e
```

The parentheses could be omitted in this case, because an infix operator has lower precedence than a function call.

## 6.3 Sum Types

Just as the product in the category of sets gives rise to product types, the coproduct gives rise to sum types. The canonical implementation of a sum type in Haskell is:

```
data Either a b = Left a | Right b
```

And like pairs, **Eithers** are commutative (up to isomorphism), can be nested, and the nesting order is irrelevant (up to isomorphism). So we can, for instance, define a sum equivalent of a triple:

```
data OneOfThree a b c = Sinistral a | Medial b | Dextral c
```

and so on.

It turns out that **Set** is also a (symmetric) monoidal category with respect to coproduct. The role of the binary operation is played by the disjoint sum, and the role of the unit element is played by the initial object. In terms of types, we have **Either** as the monoidal operator and **Void**, the uninhabited type, as its neutral element. You can think of **Either** as plus, and **Void** as zero. Indeed, adding **Void** to a sum type doesn't change its content. For instance:

```
Either a Void
```

is isomorphic to **a**. That's because there is no way to construct a **Right** version of this type — there isn't a value of type **Void**. The only inhabitants of **Either a Void** are constructed using the **Left** constructors and they simply encapsulate a value of type **a**. So, symbolically,  $a + 0 = a$ .

Sum types are pretty common in Haskell, but their C++ equivalents, unions or variants, are much less common. There are several reasons for that.

First of all, the simplest sum types are just enumerations and are implemented using **enum** in C++. The equivalent of the Haskell sum type:



```
data Color = Red | Green | Blue
```

is the C++:

```
enum { Red, Green, Blue };
```

An even simpler sum type:

```
data Bool = True | False
```

is the primitive `bool` in C++.

Simple sum types that encode the presence or absence of a value are variously implemented in C++ using special tricks and “impossible” values, like empty strings, negative numbers, null pointers, etc. This kind of optionality, if deliberate, is expressed in Haskell using the `Maybe` type:

```
data Maybe a = Nothing | Just a
```

The `Maybe` type is a sum of two types. You can see this if you separate the two constructors into individual types. The first one would look like this:

```
data NothingType = Nothing
```

It’s an enumeration with one value called `Nothing`. In other words, it’s a singleton, which is equivalent to the unit type `()`. The second part:

```
data JustType a = Just a
```

is just an encapsulation of the type `a`. We could have encoded `Maybe` as:

```
data Maybe a = Either () a
```

More complex sum types are often faked in C++ using pointers. A pointer can be either null, or point to a value of specific type. For instance, a Haskell list type, which can be defined as a (recursive) sum type:

```
data List a = Nil | Cons a (List a)
```

can be translated to C++ using the null pointer trick to implement the empty list:

```
template<class A>
class List {
    Node<A> * _head;
public:
    List() : _head(nullptr) {} // Nil
    List(A a, List<A> l)      // Cons
        : _head(new Node<A>(a, l))
    {}
};
```

Notice that the two Haskell constructors **Nil** and **Cons** are translated into two overloaded **List** constructors with analogous arguments (none, for **Nil**; and a value and a list for **Cons**). The **List** class doesn't need a tag to distinguish between the two components of the sum type. Instead it uses the special **nullptr** value for **\_head** to encode **Nil**.

The main difference, though, between Haskell and C++ types is that Haskell data structures are immutable. If you create an object using one particular constructor, the object will forever remember which constructor was used and what arguments were passed to it. So a **Maybe**

object that was created as **Just "energy"** will never turn into **Nothing**. Similarly, an empty list will forever be empty, and a list of three elements will always have the same three elements.

It's this immutability that makes construction reversible. Given an object, you can always disassemble it down to parts that were used in its construction. This deconstruction is done with pattern matching and it reuses constructors as patterns. Constructor arguments, if any, are replaced with variables (or other patterns).

The **List** data type has two constructors, so the deconstruction of an arbitrary **List** uses two patterns corresponding to those constructors. One matches the empty **Nil** list, and the other a **Cons**-constructed list. For instance, here's the definition of a simple function on **Lists**:

```
maybeTail :: List a -> Maybe (List a)
maybeTail Nil = Nothing
maybeTail (Cons _ t) = Just t
```

The first part of the definition of **maybeTail** uses the **Nil** constructor as pattern and returns **Nothing**. The second part uses the **Cons** constructor as pattern. It replaces the first constructor argument with a wildcard, because we are not interested in it. The second argument to **Cons** is bound to the variable **t** (I will call these things variables even though, strictly speaking, they never vary: once bound to an expression, a variable never changes). The return value is **Just t**. Now, depending on how your **List** was created, it will match one of the clauses. If it was created using **Cons**, the two arguments that were passed to it will be retrieved (and the first discarded).

Even more elaborate sum types are implemented in C++ using polymorphic class hierarchies. A family of classes with a common ancestor may be understood as one variant type, in which the vtable serves as a

hidden tag. What in Haskell would be done by pattern matching on the constructor, and by calling specialized code, in C++ is accomplished by dispatching a call to a virtual function based on the vtable pointer.

You will rarely see `union` used as a sum type in C++ because of severe limitations on what can go into a union. You can't even put a `std::string` into a union because it has a copy constructor.

## 6.4 Algebra of Types

Taken separately, product and sum types can be used to define a variety of useful data structures, but the real strength comes from combining the two. Once again we are invoking the power of composition.

Let's summarize what we've discovered so far. We've seen two commutative monoidal structures underlying the type system: We have the sum types with `Void` as the neutral element, and the product types with the unit type, `()`, as the neutral element. We'd like to think of them as analogous to addition and multiplication. In this analogy, `Void` would correspond to zero, and unit, `()`, to one.

Let's see how far we can stretch this analogy. For instance, does multiplication by zero give zero? In other words, is a product type with one component being `Void` isomorphic to `Void`? For example, is it possible to create a pair of, say `Int` and `Void`?

To create a pair you need two values. Although you can easily come up with an integer, there is no value of type `Void`. Therefore, for any type `a`, the type `(a, Void)` is uninhabited — has no values — and is therefore equivalent to `Void`. In other words,  $a \times 0 = 0$ .

Another thing that links addition and multiplication is the distributive property:

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

Does it also hold for product and sum types? Yes, it does — up to isomorphisms, as usual. The left hand side corresponds to the type:

```
(a, Either b c)
```

and the right hand side corresponds to the type:

```
Either (a, b) (a, c)
```

Here's the function that converts them one way:

```
prodToSum :: (a, Either b c) -> Either (a, b) (a, c)
prodToSum (x, e) =
  case e of
    Left  y -> Left  (x, y)
    Right z -> Right (x, z)
```

and here's one that goes the other way:

```
sumToProd :: Either (a, b) (a, c) -> (a, Either b c)
sumToProd e =
  case e of
    Left  (x, y) -> (x, Left  y)
    Right (x, z) -> (x, Right z)
```

The **case of** statement is used for pattern matching inside functions. Each pattern is followed by an arrow and the expression to be evaluated when the pattern matches. For instance, if you call **prodToSum** with the value:

```
prod1 :: (Int, Either String Float)
prod1 = (2, Left "Hi!")
```

the `e` in `case e of` will be equal to `Left "Hi!"`. It will match the pattern `Left y`, substituting `"Hi!"` for `y`. Since the `x` has already been matched to `2`, the result of the `case of` clause, and the whole function, will be `Left (2, "Hi!")`, as expected.

I'm not going to prove that these two functions are the inverse of each other, but if you think about it, they must be! They are just trivially re-packing the contents of the two data structures. It's the same data, only different format.

Mathematicians have a name for two such intertwined monoids: it's called a *semiring*. It's not a full *ring*, because we can't define subtraction of types. That's why a semiring is sometimes called a *rig*, which is a pun on "ring without an *n*" (negative). But barring that, we can get a lot of mileage from translating statements about, say, natural numbers, which form a *rig*, to statements about types. Here's a translation table with some entries of interest:

Numbers	Types
0	<code>Void</code>
1	<code>()</code>
$a + b$	<code>Either a b = Left a   Right b</code>
$a \times b$	<code>(a, b)</code> or <code>Pair a b = Pair a b</code>
$2 = 1 + 1$	<code>data Bool = True   False</code>
$1 + a$	<code>data Maybe = Nothing   Just a</code>

The list type is quite interesting, because it's defined as a solution to an equation. The type we are defining appears on both sides of the equation:

```
data List a = Nil | Cons a (List a)
```

If we do our usual substitutions, and also replace `List a` with `x`, we get the equation:

$$x = 1 + a * x$$

We can't solve it using traditional algebraic methods because we can't subtract or divide types. But we can try a series of substitutions, where we keep replacing `x` on the right hand side with  $(1 + a * x)$ , and use the distributive property. This leads to the following series:

$$\begin{aligned} x &= 1 + a * x \\ x &= 1 + a * (1 + a * x) = 1 + a + a * a * x \\ x &= 1 + a + a * a * (1 + a * x) = 1 + a + a * a + a * a * a * x \\ &\dots \\ x &= 1 + a + a * a + a * a * a + a * a * a * a \dots \end{aligned}$$

We end up with an infinite sum of products (tuples), which can be interpreted as: A list is either empty, `1`; or a singleton, `a`; or a pair, `a*a`; or a triple, `a*a*a`; etc... Well, that's exactly what a list is — a string of `as`!

There's much more to lists than that, and we'll come back to them and other recursive data structures after we learn about functors and fixed points.

Solving equations with symbolic variables — that's algebra! It's what gives these types their name: algebraic data types.

Finally, I should mention one very important interpretation of the algebra of types. Notice that a product of two types `a` and `b` must contain

both a value of type **a** *and* a value of type **b**, which means both types must be inhabited. A sum of two types, on the other hand, contains either a value of type **a** *or* a value of type **b**, so it's enough if one of them is inhabited. Logical *and* and *or* also form a semiring, and it too can be mapped into type theory:

Logic	Types
<i>false</i>	<b>Void</b>
<i>true</i>	<b>()</b>
$a \parallel b$	<b>Either a b = Left a   Right b</b>
$a \&\& b$	<b>(a, b)</b>

This analogy goes deeper, and is the basis of the Curry-Howard isomorphism between logic and type theory. We'll come back to it when we talk about function types.

## 6.5 Challenges

1. Show the isomorphism between **Maybe a** and **Either () a**.
2. Here's a sum type defined in Haskell:

```
data Shape = Circle Float
           | Rect Float Float
```

When we want to define a function like **area** that acts on a **Shape**, we do it by pattern matching on the two constructors:

```
area :: Shape -> Float
```



```
area (Circle r) = pi * r * r
area (Rect d h) = d * h
```

Implement **Shape** in C++ or Java as an interface and create two classes: **Circle** and **Rect**. Implement **area** as a virtual function.

3. Continuing with the previous example: We can easily add a new function **circ** that calculates the circumference of a **Shape**. We can do it without touching the definition of **Shape**:

```
circ :: Shape -> Float
circ (Circle r) = 2.0 * pi * r
circ (Rect d h) = 2.0 * (d + h)
```

Add **circ** to your C++ or Java implementation. What parts of the original code did you have to touch?

4. Continuing further: Add a new shape, **Square**, to **Shape** and make all the necessary updates. What code did you have to touch in Haskell vs. C++ or Java? (Even if you're not a Haskell programmer, the modifications should be pretty obvious.)
5. Show that  $a + a = 2 \times a$  holds for types (up to isomorphism). Remember that 2 corresponds to **Bool**, according to our translation table.