Chapter 1

The Algebra of Algebraic Data Types

The term *algebraic data types* isn't just a catchy name for types we can pattern-match on. As the name suggests, there is in fact an *algebra* behind algebraic data types.

To start with, we can associate each type with its *cardinality*—the number of inhabitants it has (ignoring bottoms.) Consider the following simple type definitions:

```
data Void
```

```
data () = ()
```

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

`Void` has zero inhabitants, and so we assign it the number 0. `()` has one, and so it becomes 1. Not to belabor the point too hard, but `Bool` is assigned 2 corresponding to its two data constructors.

We can write these statements in formal notation as:

$$|\texttt{Void}| = 0$$
$$|\texttt{()}| = 1$$
$$|\texttt{Bool}| = 2$$

Any two types that have the same cardinality will always be isomorphic to one another. An *isomorphism* between types `s` and `t` is defined as a pair of functions `to` and `from`:

```
to   :: s -> t
from :: t -> s
```

such that composing either after the other gets you back where you started. In other words, such that the following two properties hold:

```
to . from = id
from . to = id
```

Given that two types have the same cardinality, any bijection (one-to-one mapping) between their elements is exactly these `to` and `from` functions. But where does such a bijection come from? Anywhere! Just choose an arbitrary ordering—not necessarily corresponding to an `Ord` instance—and then map the first element in your arbitrary ordering to the first element in your other one. And so on.

For example, we can define a new type that also has cardinality 2.

```
data Shmool = Smalse | Strew
```

By the argument above, we'd expect `Shmool` to be isomorphic to `Bool`. Indeed:

```
boolToShmool1 :: Bool -> Shmool
boolToShmool1 False = Smalse
boolToShmool1 True  = Strew
```

```
shmoolToBool1 :: Shmool -> Bool
shmoolToBool1 Smalse = False
shmoolToBool1 Strew  = True
```

There is also another bijection between `Shmool` and `Bool`, however.

```
shmoolToBool2 :: Shmool -> Bool
shmoolToBool2 Smalse = True
shmoolToBool2 Strew  = False
```

Which of the two should we prefer? Does it matter?

In general, for any two types with cardinality n, there are $n!$ unique isomorphisms between them. As far as the math goes, any of these isomorphisms is just as good as any other—and for most purposes knowing that an isomorphism *exists* is enough.

An isomorphism between two types `s` and `t` is a proof that *for all intents and purposes* `s` and `t` are the same thing. They might have different instances available, but this is more a statement about Haskell's typeclass machinery than it is about the equivalence of `s` and `t`.

Sum types—also known as coproduct types—correspond to addition of cardinalities. The canonical example of a sum type is `Either a b`, which is *either* an `a` or a `b`. As a result, the cardinality of `Either a b` is the cardinality of `a` plus the cardinality of `b`.

$$|\texttt{Either a b}| = |\texttt{a}| + |\texttt{b}|$$

This is why such things are called *sum* types. The argument generalizes to any datatype with multiple constructors—the cardinality of the type is always the sum of the cardinalities of its constructors.

```
data Deal a b
  = This a
  | That b
  | TheOther Bool
```

Giving rise to:

$$\begin{aligned} |\texttt{Deal a b}| &= |\texttt{a}| + |\texttt{b}| + |\texttt{Bool}| \\ &= |\texttt{a}| + |\texttt{b}| + 2 \end{aligned}$$

On the flip side of sum types are the product types. Again, we will look at the canonical example first—the pair type `(a, b)`. Analogously to sum types, the cardinality of a product type is the *product* of their cardinalities.

$$|\texttt{(a, b)}| = |\texttt{a}| \times |\texttt{b}|$$

An interesting consequence of all of this is that we find ourselves able to express mathematical truths of arithmetic in terms of types. For example, we can prove that $a \times 1 = a$ by way of an isomorphism between `(a, ())` and `a`. The unit type `()` acts here as an *unit* for products, in the monoidal sense of "sticking it in doesn't change anything."

```
prodUnit :: a -> (a, ())
prodUnit a = (a, ())
```

Likewise, `Void` acts as a unit for sum types. For example, the trivial statement $a + 0 = a$ can be witnessed as an isomorphism between `Either a Void`[1] and `a`:

```
sumUnit :: Either a Void -> a
sumUnit (Left a)  = a
sumUnit (Right v) = absurd v    . . . . . .    ❶
```

(the function `absurd` at ❶ has the type `Void -> a`. It's a sort of bluff saying "if you give me a `Void` I can give you anything you want"—which is trivially true because there are no `Void`s to be had in the first place.)

Finally, function types also have an encoding as statements about cardinality—they correspond to exponentialization. For example, we have exactly four (2^2) inhabitants of the type `Bool -> Bool`. These functions are `id`, `not`, `const True` and `const False`.

[1] We use `Void` here because it is the only type (up to isomorphism) whose cardinality is 0.

This knowledge culminates in the following table, which itself is an isomorphism between mathematics and types. It's known as the *Curry-Howard isomorphism*, and loosely states that logic is equivalent to computing. It's also known by the title *propositions as types*.

Algebra	Types
$a + b$	`Either a b`
$a \times b$	`(a, b)`
b^a	`a -> b`
$=$	*isomorphism*
0	`Void`
1	`()`

Exercise

Use Curry-Howard to prove the exponent law that $a^b \times a^c = a^{b+c}$. That is, provide a function of the type `(b -> a) -> (c -> a) -> Either b c -> a` and one of `(Either b c -> a) -> (b -> a, c -> a)`.

Exercise

Prove $(a \times b)^c = a^c \times b^c$.

Exercise

Give a proof of $(a^b)^c = a^{b \times c}$. Does it remind you of anything from `Prelude`?