6

Simple Algebraic Data Types

We'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

Simple Algebraic Data Types

これまで、型を組み合わせる基本的な方法として、「積」と「共積」の2つを見てきました。日常的なプログラミングで使われる多くのデータ構造は、この2つのメカニズムだけで構築できることが分かっています。この事実は、実用上重要な結果をもたらします。データ構造の多くの特性は合成可能である。例えば、基本型の値を等価に比較する方法が分かっていて、その比較を積型や共積型に一般化する方法が分かっていれば、複合型の等価演算子の導出を自動化することができる。Haskellでは、複合型の多くの部分集合に対して、等式、比較、文字列との変換などを自動的に導出することができます。プログラミングに登場する積型と和型について詳しく見てみましょう。

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:

6.1 Product Types

プログラミング言語における2つの型の積の標準的な実装はペアです。Haskellでは、pairはプリミティブな型コンストラクタです。C++では、標準ライブラリで定義された、比較的複雑なテンプレートです。

ペアは厳密には可換ではありません。同じ情報を持っていても、(Int, Bool)のペアを(Bool, Int)のペアに置き換えることはできません。しかし、同型までは可換です。同型はスワップ関数（これはそれ自身の逆関数です） で与えられます。

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

この2つのペアは、同じデータを保存するために異なるフォーマットを使っているだけだと考えることができます。ちょうど、ビッグエンディアンとリトルエンディアンのようなものです。ペアの中にペアを入れ子にして、任意の数の型を組み合わせて積にすることができますが、もっと簡単な方法があります。ペアの入れ子はタプルと等価です。これは、ペアの入れ子の方法が異なると同型になることの帰結です。a , b , c の3つの型をこの順序で積にしたい場合、2つの方法があります。

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

or

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

これらの型は異なっていて、一方を期待する関数に他方を渡すことはできませんが、その要素は一対一で対応しています。一方を他方に写す関数があります。

```
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)
```

というように、同型であることがわかります。これらは、同じデータを別の方法で再パッケージ化したものに過ぎません。製品型の作成は、型に対する二項演算と解釈してよいでしょう。その観点から見ると、上の同型性はモノイドで見た連想法則と非常によく似ています。

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

ただし、モノイドの場合は、2つの積の合成方法が等しかったのですが、ここでは、”同型まで ”しか等しくないということです。厳密な等式にこだわらずに同型性で我慢すれば、さらに進んで、1が乗法の単位であるのと同じように、単位型()が積の単位であることを示すことができる。実際、ある型aの値と単位との対は何の情報も加えない。型は

```
(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`:


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

は a と同型です。 以下、同型について説明します。

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

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

これらの観察は、□□（集合のカテゴリ）がモノイダルカテゴリであると言うことで形式化することができる。これはオブジェクトを掛け合わせる（ここではデカルト積を取る）という意味でモノイドでもあるカテゴリです。モノイダルカテゴリについては、今後、より詳しく、完全な定義を説明したいと思います。Haskellで積型を定義する、より一般的な方法があります。特に、すぐにわかるように、積型を和型と組み合わせた場合です。これは複数の引数を持つ名前付きコンストラクタを使用します。例えばペアは次のように定義することができます。

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

ここで，`Pair a b` は `a` と `b` という 2 つの型にパラメータ化された型の名前です．また，`P` はデータコンストラクタの名前です．ペア型は `Pair` 型コンストラクタに 2 つの型を渡すことで定義します．また，コンストラクタ `P` に適切な型の 2 つの値を渡すことで，ペア値を作成します．たとえば，`stmt`という値を`String`と`Bool`のペアとして定義してみましょう．

```
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.


```
stmt :: Pair String Bool
stmt = P "この文は" False
```

最初の行は型宣言です．これは，Pair の一般的な定義における a と b を String と Bool に置き換えた型コンストラクタ Pair を使っています．2行目では、データコンストラクタ P に具象文字列と具象論理値を渡して、実際の値を定義しています。 型コンストラクタは型を構築するために、データコンストラクタは値を構築するために使用されます。Haskellでは型コンストラクタとデータコンストラクタの名前空間は別々なので、次のように両方に同じ名前が使われているのをよく見かけます。

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

また、目を凝らすと、組み込みのペア型もこのような宣言のバリエーションと捉えることができます。実際、他の名前付きコンストラクタと同じように（,）を使って、プレフィックス記法でペアを作ることができます。

```
stmt = (,) "このステートメントは" False
```

同様に、(,,)を使ってトリプルを作成することもできます。一般的なペアやタプルを使う代わりに、次のように特定の名前付き商品タイプを定義することもできます。

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

これはStringとBoolの単なる積ですが、独自の名前とコンストラクタが与えられています。この宣言の利点は、内容は同じだが意味や機能が異なり、互いに代用できない型を数多く定義できることです。

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
```


タプルと複数引数のコンストラクタを使ったプログラミングは、どのコンポーネントが何を表しているかを追跡するのが面倒で、エラーが発生しやすいものです。どのコンポーネントが何を表しているのかを把握することは、しばしば困難です。名前付きフィールドを持つ製品型は、Haskellではrecord、Cではstructと呼ばれます。

6.2 Records

簡単な例を見てみましょう。名前と記号という2つの文字列と、原子番号という1つの整数を1つのデータ 構造にまとめて、化学元素を記述したいと思います。タプル(String, String, Int)を使って、どの成分が何を表すかを記憶しておくことができる。この関数は、元素記号が元素名の接頭辞であるかどうか（HeがHeliumの接頭辞であるように）をチェックするもので、パターンマッチングによって成分を抽出することができる。

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

このコードはエラーが発生しやすく、読むのもメンテナンスするのも大変です。レコードを定義する方がよっぽどましです。

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

この2つの表現は、これらの2つの変換関数が互いに逆であることからわかるように、同型である。

```
tupleToElem :: ( String , String , Int ) -> Element この2つの表現は同型である。
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:


```
                                , atomicNumber = a }
```

```
elemToTuple :: Element -> ( String , String , Int )
elemToTuple e = (名前 e, 記号 e, 原子番号 e)
```

レコードフィールドの名前は、これらのフィールドにアクセスするための関数としても機能することに注意してください。例えば、atomicNumber eはeからatomicNumberフィールドを取得する。ここでは、atomicNumberを型の関数として使っています。

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

Elementのレコード構文では、関数startWithSymbolはより読みやすくなる。

```
startsWithSymbol :: Element -> Bool
startsWithSymbol e = isPrefixOf (シンボルe) (名前e)
```

関数isPrefixOfをバッククォートで囲んでインフィックス演算子にするというHaskellのトリックも使えるし、ほとんど文のように読むこともできる。

```
startsWithSymbol e = symbol e `isPrefixOf` 名前 e
```

この場合、カッコを省略することもできます。なぜなら、インフィックス演算子は関数呼び出しよりも優先順位が低いからです。

6.3 Sum Types

集合のカテゴリで積が積型を生み出すのと同じように、共積は和型を生み出します。Haskellにおけるsum型の標準的な実装は以下の通りです。

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

And like pairs, `Either`s 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 Either a b = 左 a | 右 b
```

そして、ペアと同様に、Either sは（同型までは）可換で、入れ子にすることができ、入れ子の順序は（同型までは）関係ありません。したがって、たとえばトリプルの和に相当するものを定義することができる。

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

といった具合になる。□□も共積に関して（対称的な）モノイダルカテゴリであることが分かる。二項演算の役割は離散和で、単位要素の役割は初期オブジェクトで行う。型に関しては、モノイダル演算子としてEither、その中性要素として無住型であるVoidがある。Eitherはプラス、Voidはゼロと考えることができます。実際、和の型にVoidを加えても、その内容は変わりません。例えば

```
Either a Void
```

はaに同型です。 それは、この型のRightバージョンを構成する方法がないからです - Void型の値がないのです。Void 型の値はありません。Void 型の住民は Left コンストラクタを使って構築され、単に a 型の値をカプセル化するだけです。 ですから、記号的には　Ll_1D 44E + 0 = □ となります。和集合はHaskellではかなり一般的ですが、C++ではそれに相当するユニオンやバリアントはあまり一般的ではありません。これにはいくつかの理由があります。まず、最も単純な和型は単なる列挙型であり、C++ではenumを使って実装されています。Haskellのsum型に相当するものです。

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

is the C++:

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

An even simpler sum type:

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


は、C++のプリミティブなboolです。値の有無をコード化する単純なsum型は、C++では特殊なトリックや、空文字列、負の数、NULLポインタなどの「ありえない」値を使ってさまざまに実装されています。この種の任意性は、意図的であれば、HaskellではMaybe型を使って表現される。


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


Maybe型は2つの型の和である。2つのコンストラクタを個別の型に分離すると、これがわかる。最初のものは次のようになる。


```
data NothingType = Nothing
```


これは、Nothingという1つの値を持つ列挙型です。つまり、シングルトンであり、単位型()に相当するものです。2番目の部分です。


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


は単なる型aのカプセル化です。 Maybeを次のようにエンコードすることができました。

```haskell
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:

```haskell
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`


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

C++では、より複雑な和の型はポインターを用いてごまかされることがよくあります。ポインタは NULL か、特定の型の値を指すかのどちらかです。たとえば、Haskell のリスト型は、（再帰的な）合計型として定義することができます。

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

は、空リストを実装するためにヌルポインタのトリックを使ってC++に翻訳することができます。

```
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))
    {}
};
```

Haskell の 2 つのコンストラクタ Nil と Cons は、類似の引数（Nil では none ; Cons では value と list）を持つ 2 つのオーバーロードされた List コンストラクタに変換されていることに注意してください。List クラスは sum 型の 2 つのコンポーネントを区別するためのタグを必要としません。その代わり，_head に特別な nullptr 値を使用して Nil をエンコードします．しかし、HaskellとC++の型の主な違いは、Haskellのデータ構造が不変であることです。ある特定のコンストラクタを使ってオブジェクトを作成すると、そのオブジェクトはどのコンストラクタが使われたか、どの引数が渡されたかを永遠に覚えています。したがって、Maybeは

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

オブジェクトは、ただの「エネルギー」として作成され、決して Nothing になることはありません。同様に、空のリストは永遠に空であり、3つの要素のリストは常に同じ3つの要素を持つことになります。このような不変性があるからこそ、可逆的な構造を実現できるのです。オブジェクトがあれば、それを構成に使われた部品に分解することはいつでも可能である。この分解はパターンマッチで行われ、コンストラクタをパターンとして再利用する。コンストラクタの引数がある場合は、変数（または他のパターン）に置き換えます。Listデータ型には2つのコンストラクタがあるので、任意のListのデコンストラクションには、それらのコンストラクタに対応する2つのパターンが使用されます。1つは空のNilリストにマッチし、もう1つはConsで構成されたリストにマッチします。例えば、以下はList sに対する簡単な関数の定義です。

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

maybeTail の定義の最初の部分は、パターンとして Nil コンストラクタを使用し、Nothing を返します。2 番目の部分では、Cons コンストラクタをパターンとして使用しています。これはコンストラクタの最初の引数をワイルドカードに置き換えたもので、この引数には興味がありません。Consの第2引数は変数tに束縛されます（厳密に言えば、変数は変化しないのですが、私はこれらを変数と呼びます：一旦式に束縛されると、変数は決して変化しないのです）。返り値は Just t です．さて、リストがどのように作成されたかに応じて、いずれかの節にマッチします。もしConsを使って作られたのなら、渡された2つの引数が取り出されます（そして、最初の引数は捨てられます）。C++では、さらに複雑な和型がポリモーフィックなクラス階層を使って実装されています。共通の祖先を持つクラス群は，1つの異形型として理解することができ，その中でvtableは

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:

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

という隠しタグのようなものです。Haskellではコンストラクタのパターンマッチングと特殊なコードの呼び出しによって行われることが、C++ではvtableポインタに基づく仮想関数の呼び出しをディスパッチすることによって行われます。C++では、和集合の型として和集合が使われることはほとんどありません。和集合に入れられるものには厳しい制限があるからです。std::stringはコピーコンストラクタを持っているので、ユニオンに入れるこ とはできません。

6.4 Algebra of Types

積と和の型を別々に使えば、さまざまな有用なデータ構造を定義できますが、本当の強さは、この2つを組み合わせたときに発揮されます。もう一度、合成の力を使ってみましょう。これまでに発見したことをまとめてみましょう。型システムの基礎となる2つの可換モノイダル構造を見てきました。Voidを中性要素とする和型と、単位型である()を中性要素とする積型です。これらは足し算と掛け算のようなものであると考えたい。この例えの場合、Voidは0に、単位の()は1に対応することになります。このアナロジーをどこまで拡張できるか試してみましょう。例えば、ゼロを掛けるとゼロになるのでしょうか？つまり、Voidを1成分とする積型は、Voidと同型なのでしょうか？例えば、Int と Void のペアを作ることは可能でしょうか？ペアを作るには、2つの値が必要です。整数は簡単に思いつきますが、Void 型の値はありません。したがって、任意の型 a に対して、型（a, Void）は無人、つまり値を持たないので、Voidと等価です。つまり、□ × 0 = 0となる。足し算と掛け算を結びつけるもう一つのものとして、分配の性質がある。

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

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:

これは積や和の型でも成り立つのでしょうか？はい、成り立ちます。いつものように、同型であれば成り立ちます。左辺が型に対応します。

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

で、右辺が型に対応します。

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

これらを一方的に変換する関数は次のとおりです。

```
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)
```

そして、これがその逆方向の関数です。

```
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)
```

`case of`文は、関数内部でのパターンマッチに使われます。各パターンの後には、矢印と、パターンにマッチしたときに評価される式が続きます。たとえば、`prodToSum`を値で呼び出した場合。

```
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	`Void`
1	`()`
$a+b$	`Either a b = Left a | Right b`
$a\times b$	`(a, b)` or `Pair a b = Pair a b`
$2=1+1$	`data Bool = True | False`
$1+a$	`data Maybe = Nothing | Just a`


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

の場合のeは、`Left "Hi!"`と同じになります。これは、パターン`Left y`にマッチし、yを `"Hi!"`に置き換えます。xはすでに2にマッチしているので、`case of`節の結果、そして関数全体は、予想通り`Left (2, "Hi!")`になります。この2つの関数が互いに逆であることを証明するつもりはないが、よく考えてみれば、そうであるに違いない！この2つの関数は、2つのデータ構造の中身を単純に詰め替えただけなのです。同じデータで、形式が違うだけなのです。数学者は、このように絡み合った2つのモノイドをセマリングと呼んでいる。型の引き算を定義することができないので、完全な環ではありません。そのため、セミリングをリグと呼ぶことがあります。これは、「nのない環」（ネガティブ）をもじったものです。しかし、それを除けば、例えば、rigを形成する自然数についての記述を型についての記述に翻訳することで、多くの利益を得ることができます。以下は、いくつかの興味深いエントリを含む翻訳テーブルです。

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

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:

```
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
...
x = 1 + a + a*a + a*a*a + a*a*a*a...
```

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 `a`s!

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

リスト型は方程式の解として定義されているので、非常に興味深いものです。私たちが定義している型は、方程式の両辺に現れます。

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

通常の代入を行い、`List a`を`x`に置き換えると、次のような方程式が得られます。

```
x = 1 + a * x
```

引き算や割り算ができないので、従来の代数的な手法では解けない。しかし、右辺の`x`を`(1 + a*x)`に置き換え、分配の性質を利用した一連の代入を試してみることができます。すると、次のような一連が導かれます。

```
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
...
x = 1 + a + a*a + a*a*a + a*a*a*a...
```

と解釈できる無限の積和(タプル)に行き着く。リストは空(1)か、単項(a)か、対(a*a)か、三項(a*a*a)か、などなど・・・まさにリストとは「sの文字列」なのです。リストにはそれ以上のものがあり、ファンクタと固定点を学んだ後に、リストと他の再帰的データ構造に戻ることになります。記号変数を使った方程式の解法、それが代数学です。これが代数的データ型という名前の由来です。最後に、型の代数に関する非常に重要な解釈について述べておく。2つの型a,bの積は次のものを含まなければならないことに注意してください。

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
false	`Void`
true	`()`
$a \mid\mid b$	`Either a b = Left a | Right b`
$a \,\&\&\, b$	`(a, 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
   ```


つまり，両方の型が存在しなければならないのです．一方、2つの型の和は、a型の値かb型の値のどちらかを含むので、どちらか一方が居住していれば十分なのです。論理的なandとorもセミリングを形成し、これも型理論にマッピングすることができる。

Logic	Types
false	`Void`
true	`()`
$a \mid\mid b$	`Either a b = Left a | Right b`
$a \,\&\&\, b$	`(a, b)`

このアナロジーはさらに深く、論理学と型理論の間のカレー・ハワード同型論の基礎となるものである。関数型について話すときに、またこの話に戻ります。

6.5 Challenges

1. Maybe a と Either () a の間の同型性を示せ . 2. Haskell で定義された sum 型がここにあります。

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

   areaのようにShapeに作用する関数を定義する場合、2つのコンストラクタのパターンマッチングで行います。

   ```
   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.


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

C++またはJavaでShapeをインターフェースとして実装し、2つのクラスを作成します。CircleとRectの2つのクラスを作成します。areaを仮想関数として実装する。3. 3. 先ほどの例の続きです。Shapeの円周を計算する新しい関数circを簡単に追加することができます。Shapeの定義に触ることなく、このようなことができます。

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

あなたのC++またはJavaの実装にcircを追加してください。元のコードのどの部分を触らなければならなかったのでしょうか？4. さらに続けます。Shape に新しいシェイプ、Square 、を追加し、必要なすべての更新を行います。HaskellとC++やJavaとでは、どのようなコードを触る必要がありましたか？(あなたがHaskellプログラマーでないとしても、修正はかなり明白なはずです)。5. 型について、□ = 2 × L1_1D44E が成立することを示しなさい（同型まで）。翻訳表によると、2はBoolに対応することを思い出してください。