Chapter 4

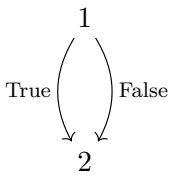
Sum Types

4.1 Bool

We know how to compose arrows. But how do we compose objects?

We have defined 0 (the initial object) and 1 (the terminal object). What is 2 if not 1 plus 1?

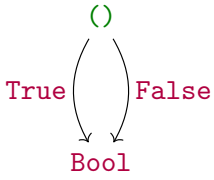
A 2 is an object with two elements: two arrows coming from 1. Let's call one arrow `True` and the other `False`. Don't confuse those names with the logical interpretations of the initial and the terminal objects. These two are *arrows*.



This simple idea can be immediately expressed in Haskell as the definition of a type, traditionally called `Bool`, after its inventor George Boole (1815-1864).

```
data Bool where
  True  :: () -> Bool
  False :: () -> Bool
```

It corresponds to the same diagram, only with some Haskell renamings:



As we've seen before, there is a shortcut notation for elements, so here's a more compact version:

```
data Bool where
  True  :: Bool
  False :: Bool
```

We can now define a term of the type `Bool`, for instance

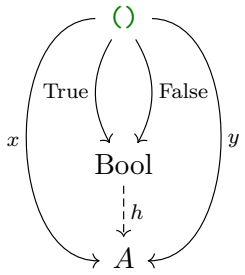
```
x :: Bool
x = True
```

The first line declares `x` to be an element of `Bool` (really, a function `()->Bool`), and the second line tells us which one of the two.

The functions `True` and `False` that we used in the definition of `Bool` are called *data constructors*. They can be used to construct specific terms, like in the example above. As a side note, in Haskell, function names start with lower-case letters, except when they are data constructors.

Our definition of the type `Bool` is still incomplete. We know how to construct a `Bool` term, but we don't know what to do with it. We have to be able to define arrows that go out of `Bool`—the *mappings out* of `Bool`.

The first observation is that, if we have an arrow `h` from `Bool` to some type `A` then we automatically get two arrows x and y from unit to `A`, just by composition. The following diagram commutes:



In other words, every function `Bool->A` produces a pair of elements of `A`.

Given a concrete type `A`:

```
h :: Bool -> A
```

we have:

```
x = h True
y = h False
```

where

```
x :: A
y :: A
```

Notice the use of the shorthand notation for the application of a function to an element:

```
h True -- meaning: h . True
```

We are now ready to complete our definition of `Bool` by adding the condition that any function from `Bool` to `A` not only produces but *is equivalent* to a pair of of elements of `A`. In other words, a pair of elements uniquely determines a function from `Bool`.

What this means is that we can interpret the diagram above in two ways: Given `h`, we can easily get `x` and `y`. But the converse is also true: a pair of elements `x` and `y` uniquely *defines* `h`.

We have a "buddy system," or a bijection, at work here. This time it's a one-to-one mapping between a pair of elements (x, y) and an arrow h.

In Haskell, this definition of `h` is encapsulated in the `if`, `then`, `else` construct. Given

```
x :: A
y :: A
```

we define the mapping out

```
h :: Bool -> A
h b = if b then x else y
```

Here, `b` is a term of type `Bool`.

In general, a data type is created using *introduction* rules and deconstructed using *elimination* rules. The `Bool` data type has two introduction rules, one using `True` and another using `False`. The `if`, `then`, `else` construct defines the elimination rule.

The fact that, given `h`, we can reconstruct the two terms used to define it, is called the *computation* rule. It tells us how to compute the result of `h`. If we call `h` with `True`, then the result is `x`; if we call it with `False`, the result is `y`.

We should never lose sight of the purpose of programming: to decompose complex problems into a series of simpler ones. The definition of `Bool` illustrates this idea. Whenever we have to construct a mapping out of `Bool`, we decompose it into two smaller tasks of constructing a pair of elements of the target type. We traded one larger problem for two simpler ones.

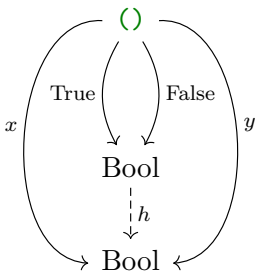
Examples

Let's do a few examples. We haven't defined many types yet, so we'll be limited to mappings out from `Bool` to either `Void`, `()`, or `Bool`. Such edge cases, however, may offer new insights into well known results.

We have decided that there can be no functions (other than identity) with `Void` as a target, so we don't expect any functions from `Bool` to `Void`. And indeed, we have zero pairs of elements of `Void`.

What about functions from `Bool` to `()`? Since `()` is terminal, there can be only one function from `Bool` to it. And, indeed, this function corresponds to the single possible pair of functions from `()` to `()`—both being identities. So far so good.

The interesting case is functions from `Bool` to `Bool`. Let's plug `Bool` in place of `A` in our diagram:



How many pairs of functions from `()` to `Bool` do we have at our disposal? There are only two such functions, `True` and `False`, so we can form four pairs. These are $(True, True)$, $(False, False)$, $(True, False)$, and $(False, True)$. Therefore there can only be four functions from `Bool` to `Bool`.

We can write them in Haskell using the `if`, `then`, `else` construct. For instance, the last one, which we'll call `not` is defined as:

```
not :: Bool -> Bool
not b = if b then False else True
```

We can also look at functions from `Bool` to `A` as elements of the object of arrows, or the exponential A^2, where 2 is the `Bool` object. Accoording to our count, we have zero elements in 0^2, one element in 1^2, and four elements in 2^2. This is exactly what we'd expect from high-school algebra, where numbers actually meant numbers.

Exercise 4.1.1. *Write the implementations of the three other functions `Bool->Bool`.*

4.2 Enumerations

What comes after 0, 1, and 2? An object with three data constructors. For instance:

```
data RGB where
  Red   :: RGB
  Green :: RGB
  Blue  :: RGB
```

If you're tired of redundant syntax, there is a shorthand for this type of definition:

```
data RGB = Red | Green | Blue
```

This introduction rule allows us to construct terms of the type `RGB`, for instance:

```
c :: RGB
c = Blue
```

To define mappings out of `RGB`, we need a more general elimination pattern. Just like a function from `Bool` was determined by two elements, a function from `RGB` to `A` is determined by a triple of elements of `A`: `x`, `y`, and `z`. We can write such function using *pattern matching*:

```
h :: RGB -> A
h Red   = x
h Green = y
h Blue  = z
```

This is just one function whose definition is split into three cases.

It's possible to use the same syntax for `Bool` as well, in place of `if`, `then`, `else`:

```
h :: Bool -> A
h True  = x
h False = y
```

In fact, there is a third way of writing the same thing using the `case` pattern:

```
h c = case c of
  Red   -> x
  Green -> y
  Blue  -> z
```

or even

```
h :: Bool -> A
h b = case b of
  True  -> x
  False -> y
```

You can use any of these at your convenience when programming.

These patterns will also work for types with four, five, and more data constructors. For instance, a decimal digit is one of:

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

There is a giant enumeration of Unicode characters called `Char`. Their constructors are given special names: you write the character itself between two apostrophes, e.g.,

```
c :: Char
c = 'a'
```

A pattern of ten thousand things would take many years to complete, therefore people came up with the wildcard pattern, the underscore, which matches everything.

Because the patterns are matched in order, you should make the wildcard pattern the last:

```
yesno :: Char -> Bool
yesno c = case c of
  'y' -> True
  'Y' -> True
  _   -> False
```

But why should we stop at that? The type `Int` could be thought of as an enumeration of integers in the range between -2^{29} and 2^{29} (or more, depending on the implementation). Of course, exhaustive pattern matching on such ranges is out of the question, but the principle holds.

In practice, the types `Char` for Unicode characters, `Int` for fixed-precision integers, `Double` for double-precision floating point numbers, and several others, are built into the language.

These are not infinite types. Their elements can be enumerated, even if it takes ten thousand years. The type `Integer` is infinite, though.

Short Haskell Digression

Since we are going to write more Haskell code, we have to establish some preliminaries. To define data types using functions, we need to use the language pragma called `GADTs` (it stands for *Generalized Algebraic Data Types*). The pragma has to be put at the top of the source file. For instance:

```
{-# language GADTs #-}

data Bool where
  True  :: () -> Bool
  False :: () -> Bool
```

The `Void` data type can be defined as:

```
data Void where
```

with the empty `where` clause (no data constructor!).

The function `absurd` works with any type as its target (it's a *polymorphic* function), so it is parameterized by a *type variable*. Unlike concrete types, type variables must start with a lowercase letter. Here, `a` is such a type variable:

```
absurd :: Void -> a
absurd v = undefined
```

We use `undefined` to placate the compiler. In this case, we are absolutely sure that the function `absurd` can never be called, because it's impossible to construct an argument of type `Void`.

You can use `undefined` when you're only interested in compiling, as opposed to running, your code. For instance, you may need to plug a function `f` to check if your definitions work together:

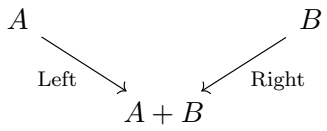
```
f :: a -> x
f = undefined
```

If you want to experiment with defining your own versions of standard types, like `Either`, you have to tell the compiler to hide the originals that are defined in the standard library called the `Prelude`. Put this line at the top of the file, after the language pragmas:

```
import Prelude hiding (Either, Left, Right)
```

4.3 Sum Types

The `Bool` type could be seen as the sum $2 = 1 + 1$. But nothing stops us from replacing 1 with another type, or even replacing each of the 1s with different types. We can define a new type $A + B$ by using two arrows. Let's call them `Left` and `Right`. The defining diagram is the introduction rule:

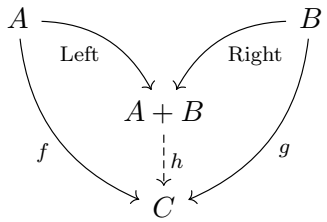


In Haskell, the type $A + B$ is called `Either a b`. By analogy with `Bool`, we can define it as

```
data Either a b where
  Left  :: a -> Either a b
  Right :: b -> Either a b
```

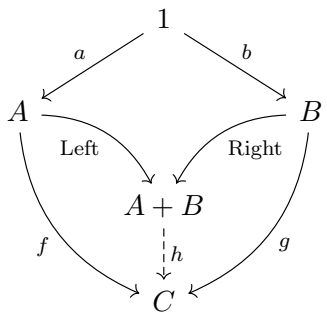
(Note the use of lower-case letters for type variables.)

Similarly, the mapping out from $A + B$ to some type C is determined by this commuting diagram:



Given a function h, we get a pair of functions f and g just by composing it with `Left` and `Right`. Conversely, such a pair of functions uniquely determines h. This is the elimination rule.

When we want to translate this diagram to Haskell, we need to select elements of the two types. We can do it by defining the arrows a and b from the terminal object.



Follow the arrows in this diagram to get

$$h \circ \text{Left} \circ a = f \circ a$$

$$h \circ \text{Right} \circ b = g \circ b$$

Haskell syntax repeats these equations almost literally, resulting in this pattern-matching syntax for the definition of `h`:

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

(Again, notice the use of lower-case letters for type variables and the same letters for terms of that type. Unlike humans, the compilers don't get confused by this.)

You can also read these equations right to left, and you will see the computation rules for sum types. The two functions that were used to define `h` can be recovered by applying `h` to terms constructed using `Left` and `Right`.

You can also use the `case` syntax to define `h`:

```
h e = case e of
  Left  a -> f a
  Right b -> g b
```

So what is the essence of a data type? It is but a recipe for manipulating arrows.

Maybe

A very useful data type, `Maybe` is defined as a sum $1 + A$, for any A. This is its definition in Haskell:

```
data Maybe a where
  Nothing :: () -> Maybe a
  Just    ::  a -> Maybe a
```

The data constructor `Nothing` is an arrow from the unit type, and `Just` constructs `Maybe a` from `a`. `Maybe a` is isomorphic to `Either () a`. It can also be defined using the shorthand notation

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

`Maybe` is mostly used to encode the return type of partial functions: ones that are undefined for some values of their arguments. In that case, instead of failing, such functions return `Nothing`. In other programming languages partial functions are often implemented using exceptions.

Logic

In logic, the proposition $A + B$ is called the alternative, or *logical or*. You can prove it by providing the proof of A or the proof of B. Either one will suffice.

If you want to prove that C follows from $A + B$, you have to be prepared for two eventualities: either somebody proved $A + B$ by proving A or by proving B. In the first case, you have to show that C follows from A. In the second case you need a proof that C follows from B. These are exactly the arrows in the elimination rule for $A + B$.

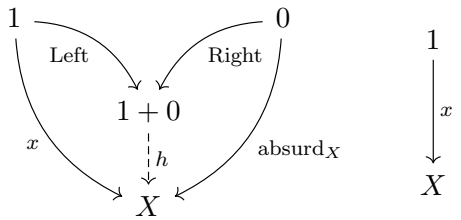
4.4 Cocartesian Categories

In Haskell, we can define a sum of any two types using `Either`. A category in which all sums exist, and the initial object exists, is called *cocartesian*, and the sum is called a *coproduct*. You might have noticed that sum types mimic addition of numbers. It turns out that the initial object plays the role of zero.

One Plus Zero

Let's first show that $1 + 0 \cong 1$, meaning the sum of the terminal object and the initial object is isomorphic to the terminal object. The standard procedure for this kind of proofs is to use the Yoneda trick. Since sum types are defined by mapping out, we should compare arrows coming *out* of either side.

Look at the definition of $1 + 0$ and it's mapping out to any object X. It's defined by a pair (x, absurd_X), where x is an element of X.



We want to establish a one-to-one mapping between arrows originating in $1 + 0$ and the ones originating in 1. Since h is determined by the pair (x, absurd_X), we can simply map it to the arrow x originating in 1. Since there is only one absurd_X, the mapping is a bijection.

So our β_X maps any pair (x, absurd_X) to x. Conversely, β_X^{-1} maps x to the pair (x, absurd_X). But is it a natural transformation?

To answer that, we need to consider what happens when we change focus from X to some Y that is connected to it through an arrow $g\colon X \to Y$. We have two options now:

- Make h switch focus by post-composing both x and absurd_X with g. We get a new pair $(y = g \circ x, \text{absurd}_Y)$. Follow it by β_Y.
- Use β_X to map (x, absurd_X) to x. Follow it with the post-composition $(g \circ -)$.

In both cases we get the same arrow $y = g \circ x$. So the transformation β is natural. Therefore $1 + 0$ is isomorphic to 1.

In Haskell, we can define the two functions that form the isomorphism, but there is no way of directly expressing the fact that they are the inverse of each other.

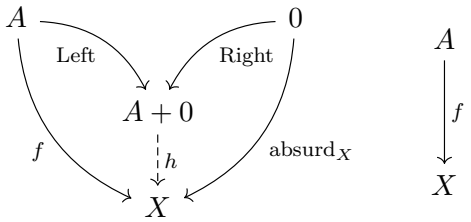
```
f :: Either () Void -> ()
f (Left ()) = ()
f (Right _) = ()

f_1 :: () -> Either () Void
f_1 _ = Left ()
```

The underscore wildcard in a function definition means that the argument is ignored. The second clause in the definition of `f` is redundant, since there are no terms of the type `Void`.

A Plus Zero

A very similar argument can be used to show that $A + 0 \cong A$. The following diagram explains it.



We can translate this argument to Haskell by implementing a (polymorphic) function `h` that works for any type `a`.

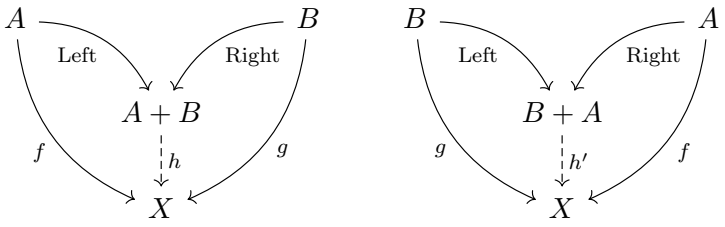
Exercise 4.4.1. *Implement, in Haskell, the two functions that form the isomorphism between* `Either a Void` *and* `a`.

We could use a similar argument to show that $0 + A \cong A$, but there is a more general property of sum types that obviates that.

Commutativity

There is a nice left-right symmetry in the diagrams that define the sum type which suggests that it satisfies the commutativity rule, $A + B \cong B + A$.

Let's consider mappings out of both sides. You can easily see that, for every h that is determined by a pair (f, g) on the left, there is a corresponding h' given by a pair (g, f) on the right. That establishes the bijection of arrows.



Exercise 4.4.2. *Show that the bijection defined above is natural. Hint: Both f and g change focus by post-composition with $k \colon X \to Y$.*

Exercise 4.4.3. *Implement, in Haskell, the function that witnesses the isomorphism between* `Either a b` *and* `Either b a`*. Notice that this function is its own inverse.*

Associativity

Just like in arithmetic, the sum that we have defined is associative:

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

It's easy to write the mapping out for the left hand side:

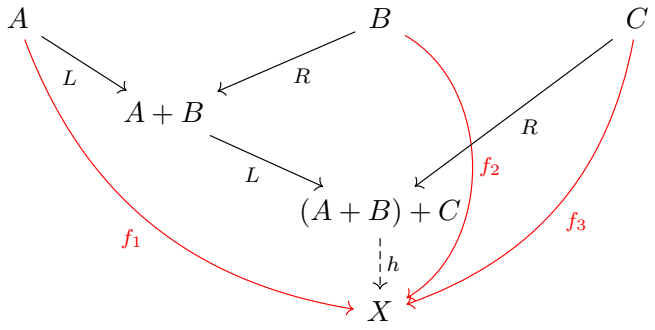
```
h :: Either (Either a b) c -> x
h (Left (Left a))  = f1 a
h (Left (Right b)) = f2 b
h (Right c)        = f3 c
```

Notice the use of nested patterns like `(Left (Left a))`, etc. The mapping is fully defined by a triple of functions. The same functions can be used to define the mapping out of the right hand side:

```
h' :: Either a (Either b c) -> x
h' (Left a)          = f1 a
h' (Right (Left b))  = f2 b
h' (Right (Right c)) = f3 c
```

This establishes a one-to-one mapping between triples of functions that define the two mappings out. This mapping is natural because all changes of focus are done using post-composition. Therefore the two sides are isomorphic.

This code can also be displayed in diagrammatical form. Here's the left hand side diagram.



Functoriality

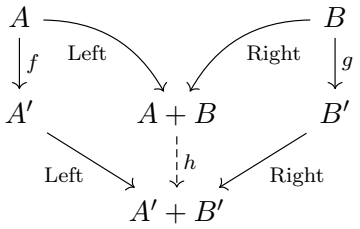
Since the sum is defined by the mapping out property, it was easy to see what happens when we change focus: it changes "naturally" with the foci of the arrows that define the product. But what happens when we move the sources of those arrows?

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 constructors Left and Right, respectively, can be used to define the mapping of the sums:



The pair of arrows, $(\text{Left} \circ f, \text{Right} \circ g)$ uniquely defines the arrow $h\colon A + B \to A' + B'$.

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

Exercise 4.4.4. *Show that functoriality preserves composition. Hint: take two composable arrows, $g\colon B \to B'$ and $g'\colon B' \to B''$ and show that applying $g' \circ g$ gives the same result as first applying g to transform $A + B$ to $A + B'$ and then applying g' to transform $A + B'$ to $A + B''$.*

Exercise 4.4.5. *Show that functoriality preserves identity. Hint: use id_B and show that it is mapped to id_{A+B}.*

Symmetric Monoidal Category

When a child learns addition we call it arithmetics. When a grownup learns addition we call it a cocartesian category.

Whether we are adding numbers, composing arrows, or constructing sums of objects, we are re-using the same idea of decomposing complex things into their simpler components.

When things come together to form a new thing, and the operation is associative, and it has a neutral element, we know how to deal with ten thousand things.

The sum type we have defined satisfies these properties:

$$A + 0 \cong A$$

$$A + B \cong B + A$$

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

and it's functorial. A category with this type of operation is called *symmetric monoidal*. When the operation is the sum (coproduct), it's called *cocartesian*. In the next chapter we'll see another monoidal structure that's called *cartesian* without the "co."