Chapter 1

Clean Slate

Programming starts with types and functions. You probably have some preconceptions about what types and functions are: get rid of them! They will cloud your mind.

Don't think about how things are implemented in hardware. What computers are is just one of the many models of computation. We shouldn't get attached to it. You can perform computations in your mind, or with pen and paper. The physical substrate is irrelevant to the idea of programming.

1.1 Types and Functions

Paraphrasing Lao Tzu: *The type that can be described is not the eternal type.* In other words, type is a primitive notion. It cannot be defined.

Instead of calling it a *type*, we could as well call it an *object* or a *proposition*. These are the words that are used to describe it in different areas of mathematics (type theory, category theory, and logic, respectively).

There may be more than one type, so we need a way to name them. We could do it by pointing fingers at them, but since we want to effectively communicate with other people, we usually name them. So we'll talk about type A, B, C; or `Int`, `Bool`, `Double`, and so on. These are just names.

A type by itself has no meaning. What makes it special is how it connects to other types. The connections are described by arrows. An arrow has one type as its source and one type as its target. The target could be the same as the source, in which case the arrow loops around.

An arrow between types is called a *function*. An arrow between objects is called a *morphism*. An arrow between propositions is called an *implication*. These are just words that are used to describe arrows in different areas of mathematics. You can use them interchangeably.

A proposition is something that may be true. We may interpret an arrow between two objects A and B as an implication between two propositions. When we say that A *implies* B, it means that if A is true then B is also true. In other words, if we can prove A, we'll know that B is true.

There may be more than one arrow between two types, so we need to name them. For instance, here's an arrow called f that goes from type A to type B

$$A \xrightarrow{f} B$$

One way to interpret this is to say that the function f takes an argument of type A and produces a result of type B. Or that f is a proof that if A is true then B is also true.

Note: The connection between type theory, lambda calcululus (which is the foundation of programming), logic, and category theory is known as Curry-Howard-Lambek isomorphism.

1.2 Yin and Yang

An object is defined by its connections. An arrow is a proof, a witness, of the fact that two objects are connected. Sometimes there's no proof, the objects are disconnected; sometimes there are many proofs; and sometimes there's a single proof—a unique arrow between two objects.

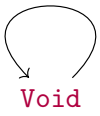
What does it mean to be *unique*? It means that if you can find two of those, then they must be equal.

An object that has a unique outgoing arrow to every object is called the *initial object*.

Its dual is an object that has a unique incoming arrow from every object. It's called the *terminal object*.

In mathematics, the initial object is often denoted by 0 and the terminal object by 1.

The initial object is the source of everything. As a type it's known as `Void`. It symbolizes the chaos from which everything arises. Since there is an arrow from `Void` to everything, there is also an arrow from `Void` to itself.



Thus `Void` begets `Void` and everything else.

The terminal object unites everything. As a type it's know as Unit. It symbolizes the ultimate order.

In logic, the terminal object signifies the ultimate truth, symbolized by T or $\top$. The fact that there's an arrow to it from any object means that $\top$ is true no matter what your assumptions are.

Dually, the initial object signifies logical falsehood, contradiction, or a counterfactual. It's written as False and symbolized by an upside down T, $\bot$. The fact that there is an arrow from it to any object means that you can prove anything starting from false premises.

In English, there is special grammatical construct for counterfactual implications. When we say, "If wishes were horses, beggars would ride," we mean that the equality between wishes and horses implies that beggars be able to ride. But we know that the premise is false.

A programming language lets us communicate with each other and with computers. Some languages are easier for the computer to understand, others are closer to the theory. We will use Haskell as a compromise.

In Haskell, the initial object corresponds to the type called `Void`. The name for the terminal type is `()`, a pair of empty parentheses, pronounced Unit. This notation will make sense later.

There are infinitely many types in Haskell, and there is a unique function/arrow from `Void` to each one of them. All these functions are known under the same name: `absurd`.

Programming	Category theory	Logic
type	object	proposition
function	morphism (arrow)	implication
`Void`	initial object, 0	False $\bot$
`()`	terminal object, 1	True $\top$

1.3 Elements

An object has no parts but it may have structure. The structure is defined by the arrows pointing at the object. We can *probe* the object with arrows.

In programming and in logic we want our initial object to have no structure. So we'll assume that it has no incoming arrows (other than the one that's looping back from it). Therefore `Void` has no structure.

The terminal object has the simplest structure. There is only one incoming arrow from any object to it: there is only one way of probing it from any direction. In this respect, the terminal object behaves like an indivisible point. Its only property is that it exists, and the arrow from any other object proves it.

Because the terminal object is so simple, we can use it to probe other, more complex objects.

If there is more than one arrow coming from the terminal object to some object A, it means that A has some structure: there is more than one way of looking at it. Since the terminal object behaves like a point, we can visualize each arrow from it as picking a different point or element of its target.

In category theory we say that x is a *global element* of A if it's an arrow

$$1 \xrightarrow{x} A$$

We'll often simply call it an element (omitting "global").

In logic, such x is called the proof of A, since it corresponds to the implication $\top \to A$ (if **True** is true then **A** is true). Notice that there may be many different proofs of A.

In type theory, $x\colon A$ means that x is of type A.

In Haskell, we use the double-colon notation instead, but it means the same:

```
x :: A
```

We say that `x` is a term of type `A`, but we can also look at it as a function `() -> A`, a global element of `A`.

Since we have assumed that there be no arrows from any other object to `Void`, there is no arrow from the terminal object to it. Therefore `Void` has no elements. This is why we think of `Void` as empty.

The terminal object has just one element, since there is a unique arrow coming from it to itself, $1 \to 1$. This is why we sometimes call it a singleton.

Note: In category theory there is no prohibition against the initial object having incoming arrows from other objects. We assume it here for convenience.

1.4 The Object of Arrows

We describe an object by looking at its arrows. Can we describe an arrow in a similar way? Could an arrow from A to B be represented as an element of some special *object of arrows*? After all, in programming we talk about the *type* of functions from `A` to `B`. In Haskell we write:

```
f :: A -> B
```

meaning that `f` is of the type "function from A to B". Here, `A->B` is just the name we are giving to this type. To fully define this type, we would have to describe its relation to other objects, in particular to A and B. We don't have the tools to do that yet, but we'll get there.

For now, let's keep in mind the following distinction: On the one hand we have an arrow which connects two objects `A` and `B`. On the other hand we have an element of the *object of arrows* from `A` to `B`. This element is itself defined as an arrow from the terminal object `()` to the object we call `A->B`.

The notation we use in programming tends to blur this distinction. This is why in category theory we call the object of arrows an *exponential* and write it as B^A (the source is in the exponent). So the statement:

```
f :: A -> B
```

is equivalent to

$$1 \xrightarrow{f} B^A$$

In logic, an arrow $A \to B$ is an implication: it states the fact that "if A then B." An exponential object B^A is the corresponding proposition. It could be true or it could be false, we don't know. You have to prove it. Such a proof is an element of B^A.

Show me an element of B^A and I'll know that B follows from A.

Consider again the statement, "If wishes were horses, beggars would ride"—this time as an object. It's not an empty object, because you can point at a proof of it—something along the lines: "A person who has a horse rides it. Beggars have wishes. Since wishes are horses, beggars have horses. Therefore beggars ride." But, even though you have a proof of this statement, it's of no use to you, because you can never prove its premise: "wish = horse".