Chapter 1

What this book is about

The *Tao Te Ching* is a classical book by Laotse which is about the way humans should follow in their life. I am borrowing the word *Tao* which means *the way* to talk about an alternative **way** to do *Mathematics*, namely using Intuitionistic Type Theory instead of Classical Set Theory.

Most Mathematicians nowadays accept Set Theory as the foundations of Mathematics. To be precise most Mathematicians don't really care very much about foundations, they just do Mathematics using sets because that's what they were taught do it. And it works. To a degree.

I am not a Mathematician but my education took place in Computer Science. Hence maybe I lack some of the skills of a good Mathematician but also some of the biases which go along with it. Having said this since I work in theoretical Computer Science most people would describe the papers I publish as *Mathematics*.

What is Set Theory? I will give a more detailed historic overview in chapter 2. Mathematicians organize mathematical objects in sets like the set of natural numbers $\mathbb{N}$ which are the numbers you use for counting $\mathbb{N} = \{0, 1, 2, \dots\}$ [1] Another example is the set of real numbers $\mathbb{R}$ which corresponds to the points on a line and for example the number $\pi = 3.1415\dots$ is a real number, we write $\pi \in \mathbb{R}$.

One of the nice things with set theory is the fact that you only need sets to construct mathematical objects. How does this work? We start with the empty set which can be written as $\{\}$ and now we can construct a new set, namely the set containing the empty set $\{\{\}\}$. Now we have these two different sets we can put them together $\{\{\}, \{\{\}\}\}$ to make a new set. Yes if you think of russian dolls that's the right idea. I don't want to write it down but continuing with this russian doll idea we can combine all the sets we have made to make a set with 3 elements and so on and thus we can create natural numbers as sets.

Ok this is the basic idea. They are some precise rules how to construct sets and how to reason about them. This is called Zermelo-Fraenkel set theory and

[1] That I start with 0 shows that I am a computer scientist. Real mathematicians start with 1.

it is a list of 8 axioms which are written in the language of predicate logic and which only use the symbol $\in$ in addition to the basic language of predicate logic. Actually there are a few extra axioms which are usually assumed such as the *axiom of choice* or you can also omit some axioms and work in a weaker system. So there is actually more than one set theory. I will explain the axioms and predicate logic in more detail in the next chapter.

Now what is Type Theory and how is it different? First of all when I say Type Theory with capital letters I mean an alternative foundation of Mathematics which is often associated with the Swedish Mathematician and Philosopher Per Martin-Löf. I say *associated* because while it is fair to say that he started it there are now lots of ideas in contemporary Type Theory which go beyond his original conception. People also use the term *type theory* (without capitals) as the theory of types in programming languages and there are some university courses with this title. The uncaptialized type theory is related to the capitalized Type Theory but it is not the same topic.

The basic idea of Type Theory is to organize mathematical objects into types instead of sets. So for example we have a type of natural numbers $\mathbb{N}$ and a type of real numbers $\mathbb{R}$ and to say that π is a real number we write $\pi : \mathbb{R}$.

Ok, I see what you are thinking: I have just replaced the word *set* by *type* and the symbol $\in$ by : and that's all. Not so. In Type Theory we can only make objects of a certain type, that is the type is first and then we can construct the element. In Set Theory we think of all the objects (which are in turn sets) are there already and then we can organize them into different sets. Hence in set theory you can just have an arbitrary object x (which is a set again because everything is a set) and then you can ask yourself wether this object is is a natural number $x \in \mathbb{N}$ or a real number $x \in \mathbb{R}$. In Type Theory if I have $x : \mathbb{N}$ then this object is a natural number by birth and we can even ask the question wether it is a real number. We say that $x : \mathbb{N}$ is a *judgement* while $x \in \mathbb{N}$ is a proposition.

That sounds rather restrictive and so it is. Type Theory is a more disciplined approach to Mathematics but this pays off in the end. There are things we cannot do in Type Theory and we are better off because of it because Type Theory allows us to view mathematical objects more abstractly. What do I mean? In set theory there is actually more than one way to define natural numbers in some of them the empty set is include $\{\} \in \mathbb{N}$ in others it is not, we write $\{\} \notin \mathbb{N}$ to express that the empty set is not a natural number. However, this question hasn't much to do with natural numbers it is about our encoding of natural numbers. And because we can talk about the details of the encoding we cannot just replace one definition of natural numbers with another equivalent one because somewhere in our reasoning we may have used properties of the encoding. In Computer Science we would say there is a lack of information hiding.

In Type Theory we cannot ask these silly questions we cannot talk about the encoding of natural numbers. Either something is a natural number or it isn't. The empty type certainly isn't a natural number it is a type! We may compare this difference to static and dynamic typing in programming languages. For

example Python is a dynamically typed language, we get an object and it may be a number. We can query this at runtime but we can also have runtime errors because we assume something is a number but then it turned out to be a string. So in a way Python is a bit like set theory. In contrast in a statically typed language like Haskell we know at compile time what is the type of an object. As a consequence we cannot have runtime errors caused by trying to add a number and a string. So Haskell is more like Type Theory.

In Programming the statically typed approach pays off because we can catch more errors earlier. In Mathematics we have to prove things so we shouldn't make errors. However, the advantage of Type Theory is that we cannot talk about the encoding of objects and as a consequence we can replace one type by another which is *equivalent*. I am going to make this precise later but in the moment you can think of *equivalent* as meaning that there is a one-to-one correspondence of objects in two types. This is the essence of the *univalence axiom* which was introduced by Vladimir Voevodsky and which basically says that two types which are equivalent are actually equal. You couldn't do this in set theory because in set theory we can actually distinguish equivalent sets by talking about details of their encodings (e.g. wether the empty set is a natural number).

It is interesting that Voevodsky came up with the univalence axiom by thinking about an abstract version of geometry which is called Homotopy Theory. To oversimplify things: Homotopy Theory classifies geometric objects by the paths you can follow on their surface. That is a ball is different from a bicycle tube because on the tube you can walk from one point to itself by going through around the inside. You can also walk around a ball but you can transform this path by many little steps into the empty path i.e. where you just stand still. Hence upto deformation of paths there is only one path on a ball. The bicycle tube is different because we cannot deform every path to the empty path because we are only allowed to deform paths on the surface we are not allowed to go through the middle.

Hence this new version of Type Theory is often called *Homotopy Type Theory* and there is a nice book (ok I was involved in writing it) about this topic but this requires a bit of mathematical background [**?**].

There is another important difference between Type Theory and Set Theory which I would like to explain. As I said above set theory uses predicate logic which codifies the rules of reasoning. It also introduces a formalism how to write logical statements (these are called propositions), e.g. using $\wedge$ for *and* and $\forall$ to say *for all*. The rules are justified by an explanation when a proposition is true. Now this makes sense if we talk about the real world because hopefully we can just check wether a statement is true. Ok, this can also be difficult but this is a different story. However, the sets do not exist in the real world they only exist in our heads. Now what does it really mean for a statement about sets to be true?

At this point we are getting philosophical and refer to the greek philosopher Plato. Plato had the view that the world of ideas is as real as the real world and that the things we see are mere shadows of ideal objects. That is a real

table is just a shadow of the ideal table. In this *Platonic universe* mathematical objects like sets are real and hence we can talk about.

Ok that doesn't sound very convincing. Maybe the reason is that I am not a Platonist. But Mathematicians would say that it just makes sense to pretend that we can talk about mathematical objects as if the were real and who cares about philosophy anyway.

The nice thing about Type Theory is that we don't need to refer to any Platonic universe or even to have the notion of truth precise. Instead of truth we should rather think of evidence. We can explain what is the evidence for a proposition by associating a type with every proposition which is the type of evidence for this proposition. This is called the *proposition-as-types* principle or the *Curry-Howard-Equivalence*.

How does this work? If A and B are propositions and we know what is evidence for A and for B but what is evidence for $A \wedge B$ that is A **and** B? We say evidence for $A \wedge B$ is a pair (a, b) where $a : A$ and $b : B$. We can also write $A \wedge B = A \times B$ where $A \times B$ is the type of pairs or tuples made from A and B. An interesting example is the explanation of if-then that is *If A then B* which we write as $A \Rightarrow B$. Now evidence for $A \Rightarrow B$ is a function from A to B. I write the type of functions from A to B as $A \to B$ and we can now say $A \Rightarrow B = A \to B$ that is the reasons that A imples B are the functions from the reasons for A to the reaosns for B.

I should say what I mean by a function. Unlike in set theory functions in Type Theory are a primitive concept. The functions in Type Theory are the same as functions in functional programming languages like Haskell but with the proviso that they have to terminate, that is they can't run forever. And indeed Type Theory is a programming language and you can execute your type-theoretic programs. This is not true in Set Theory where there a things which are called functions which you cannot implement on a computer. I always say that this is a bad name because a function which doesn't function shouldn't be called a function.

The logic of evidence we get from Type Theory is different from the logic of truth we get in predicate logic. The culprit is the *principle of excluded middle* which says that every proposition is either true or false. We write $A \vee \neg A$ where $\vee$ means *or* and $\neg$ means *not* and A is just any proposition. We can prove this by constructing a truth table and going through the possibilities that A is either true or false. In the evidence based semantics of Type Theory we cannot justify this principle because it basically says that for any proposition we have either to give evidence that is true or we have to give evidence that it is not true. But we cannot in general do this because there are certainly propositions of which we don't know wether it is true or false. Actually it is even worse: because propositions are given by types and we cannot look into a type any proof of the principle of excluded middle would either to have to prove or disprove all propositions, which makes no sense at all.

The logic we get is called *Intuitionistic Logic* for which the Dutch Mathematician Brouwer was famous. It is called intuitionistic because the basic idea are based on intuitive justification and not the platonic universe. Hence there

is also a version of predicate logic which does not include the principle of the excluded middle and this is called *Intuitionistic Predicate Logic*, while the one with excluded middle is called *Classical Predicate Logic*. I think Type Theory is much nicer because we don't need any predicate logic on top. We just have to explain the translation of propositions to types (which is straightforward) and then there is no need for a logical system on top. I think this is really cool: Type Theory is a programming language and you get logic for free.

The restriction to intuitionistic logic is not only nice philosophically (and we can always be a bit blaze about philosophy) but it is also important practically. This follows from what I said already: because our language is a programming langauge we can compute with it. So if we prove that there is a number with some property $\exists x : \mathbb{N}.P(x)$ then we can actually compute the number from the proof. That is not the case in classical logic because we cannot execute our functions. Also in an intuitionistic system you can make some useful differences: while the principle of excluded middle doesn't always hold it may hold in some particular case. For example we may have a property P of the natural numbers so that for each instance we can prove the particular instance of the excluded middle. We write $\forall x : \mathbb{N}.P(x) \vee \neg P(x)$ that is for each natural number either P holds or it doesn't. In Computer Science terms this means that the property P is decidable. As we know thanks to Turing not every property is decidable. Many are, so for example we can decide wether a number is a prime number that is greater than 1 and only divisible by 1 and itself. But we cannot decide that if we view the number as the bit pattern of a program on a computer wether this program will stop or run forever. Hence in an intuitionistic system we can express this and other properties useful in Computer Science internally.

Ok, I realize that I am trying to sell Type Theory here. Not everybody would agree and we can have long discussions about it. Also for the clarity of presentations I have avoided to hum and err too much but as we know there are two sides to everything. Mathematicians are no fools and many of them think that using intuitionistic logic would complicate things too much and that classical reasoning is easier. I don't agree but I leave it to them to make their case.