Introduction to Homotopy Type Theory

Lecture notes for a course at EWSCS 2017

Thorsten Altenkirch

March 5, 2017

1 What is this course about?

To explain what Homotopy Type Theory is, I will first talk about Type Theory and then explain what is special about Homotopy Type Theory.

The word *type theory* has at least two meanings:

- The theory of types in programming language
- Martin-Löf's Type Theory as a constructive foundation of Mathematics

We will be mainly concerned with the latter (which is emphasised by capitalising it), even though there are interactions with the design of programming languages as well.

Type Theory is the base of a number of computer systems used as the base of interactive proof systems and very advanced (functional) programming language. Here is an incomplete list:

- NuPRL
- Coq
- Agda
- Idris
- Cubical

NuPRL is based on a different version of Type Theory than the others, this is now called *Computational Type Theory.* Coq is maybe now the system most used in formal Mathematics and has been used for some impressive developments, including a formal proof of the Four Colour Theorem and the verification of an optimising C compiler. Agda is a sort of a twitter it can be used as a interactive proof assistant or as a dependently typed programming language. Idris goes further by addressing more pragmatic concerns when using Type Theory for programming. Cubical is a very new system and more a proof of concept but it is the only one (so far) that actually implements Homotopy Type Theory.

One way to introduce Type Theory is to pick one system (I usually pick Agda) and then learn Type Theory by doing it. While this is a good way to approach this subject, and do I recommend to play with a system I want to concentrate more on the conceptual issues and then I find that having to explain the intricacies of a particular system can be a bit of a distraction. Hence this course will be a paper based introduction to Type Theory.

This course can be viewed as a taster of the book on Homotopy Type Theory which was the output of a special year at the Institute for Advanced Study in Princeton. However, a few things have happened since the book was written (e.g. the construction of cubical) and I will mention them where appropriate.

2 Type Theory vs Set Theory

I view Type Theory in the first place as a intuitive foundation of Mathematics. This is similar to how most Mathematicians use Set Theory: the have an intuitive idea what sets are but they don't usually refer back to the axioms of Set Theory. This is sometimes called *naive Set Theory* and similar what I am doing here can be called *naive Type Theory.*

In Set Theory we write $3 \in \mathbb{N}$ to express that 3 is an element of the set of natural numbers. In Type Theory we write $3 : \mathbb{N}$ to express that 3 is an element of the type of natural numbers. While this looks superficially similar, there are important differences:

- While $3 \in \mathbb{N}$ is a proposition, $3 : \mathbb{N}$ is a *judgement*, that is a piece of static information.
- In Type Theory every object and every expression has a (unique) type which is statically determined. [1]
- Hence it doesn't make any sense to use $a : A$ as a proposition.
- This is similar to the distinction between statically and dynamically typed programming languages. While in dynamically typed languages there are runtime functions to check the type of an object this doesn't make sense in statically typed languages.
- In Set Theory we define $P \subseteq Q$ as $\forall x.x \in P \to x \in Q$. We can't do this in Type Theory because $x \in P$ is not a proposition.
- Also set theoretic operations like $\cup$ or $\cap$ are not operations on types. However, they can be defined as operations on predicates, aka subsets, of a given type. $\subseteq$ can be defined as a predicate on such subsets.
- Type Theory is extensional in the sense that we can't talk about details of encodings.

[1]We are not considering subtyping here, which can be understood as a notational device allowing the omission of implicit coercions.

- This is different in Set Theory where we can ask wether $\mathbb{N} \cap \mathrm{Bool} = \emptyset$? Or wether $2 \in 3$? The answer to these questions depends on the choice of representation of these objects and sets.

Apart from the judgement $a : A$ there is also the judgement $a \equiv_A b$ which means that $a, b : A$ are *definitionally* equal. We write definitions using $:\equiv$, e.g. we can define $n : \mathbb{N}$ as 3 by writing $n :\equiv 3$. As for $a : A$ definitional equality is a static property which can be determined statically and hence which doesn't make sense as a proposition. We will later introduce $a =_A b$, *propositional equality* which can be used in propositions.

While Type Theory is in some sense more restrictive than Set Theory, this does pay off. Because we cannot talk about intensional aspects, i.e. implementation details, we can identify objects which have the same extensional behaviour. This is reflected in the *univalence axiom*, which identifies extensionally equivalent types (such as unary and binary natural numbers).

Another important difference between Set Theory and Type Theory is the way propositions are treated: Set Theory is formulated using predicate logic which relies on the notion of *truth*. Type Theory is self-contained and doesn't refer to truth but to evidence. Using the propositions-as-types translation (also called the Curry-Howard equivalence) we can assign to any proposition P the type of its evidence $[\![P]\!]$ using the following table:

$$\begin{aligned}
[\![P \implies Q]\!] &\equiv [\![P]\!] \to [\![Q]\!] \\
[\![P \wedge Q]\!] &\equiv [\![P]\!] \times [\![Q]\!] \\
[\![\mathrm{True}]\!] &\equiv 1 \\
[\![P \vee Q]\!] &\equiv [\![P]\!] + [\![Q]\!] \\
[\![\mathrm{False}]\!] &\equiv 0 \\
[\![\forall x : A.P]\!] &\equiv \Pi x : A.[\![P]\!] \\
[\![\exists x : A.P]\!] &\equiv \Sigma x : A.[\![P]\!]
\end{aligned}$$

0 is the empty type, 1 is the type with exactly on element and + is the sum or disjoint union of types. $\to$ (function type) and $\times$ should be familiar but we will revisit all of them from a type theoretic perspective. Π and Σ are less familiar in Set Theory and we will have a look at them later.

We are using a typed predicate logic here. $\neg P$ is defined as $P \implies \mathrm{False}$. Logical equivalence $P \Leftrightarrow Q$ is defined as $(P \implies Q) \wedge (Q \implies P)$. Careful, equivalence such as $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$ or $\neg(\forall x : A.P) \Leftrightarrow \exists x : A.\neg P$ do not hold in Type Theory.

Later we will see a refinement of the proposition as types translation, which changes the translation of $P \vee Q$ and $\exists x : A.P$.

3 Non-dependent types

3.1 Universes

To get started we have to say what a type is. We could achieve this by introducing another judgement but instead I am going to use universes. A universe is a type of types. For example to say that $\mathbb{N}$ is a type, I write $\mathbb{N} : \mathbf{Type}$ where $\mathbf{Type}$ is a universe.

But what is the type of $\mathbf{Type}$? Do we have $\mathbf{Type} : \mathbf{Type}$? It is well known that this doesn't work in Set Theory due to Russell's paradox (the set of all sets which does not contain itself). However, in Type Theory $a : A$ is not a proposition, hence it is not immediately clear wether the paradox still works.

However, it is possible to encode Russell's paradox in a Type Theory with $\mathbf{Type} : \mathbf{Type}$ by using trees which can branch over any type. In this theory we can construct a tree of all trees which don't have themself as immediate subtree. This tree is a subtree of itself iff it isn't which enables us to derive a contradiction.

To avoid Russell's paradox we introduce a hierarchy of universes

$$\mathbf{Type}_0 : \mathbf{Type}_1 : \mathbf{Type}_2 : \ldots$$

and we decree that any type $A : \mathbf{Type}_i$ can be lifted to a type $A^+ : \mathbf{Type}_{i+1}$. Being explicit about universe levels can be quite annoying hence we are going to ignore them most of the time but try to make sure that we don't use universes in a cyclic way. That is we write $\mathbf{Type}$ as a metavariable for $\mathbf{Type}_i$ and assume that all the levels are the same unless stated explicitly.

3.2 Functions

While in Set Theory functions are a derived concept (a subset of the cartesian product with certain properties), in Type Theory functions are a primitive concept. The basic idea is the same as in functional programming: basically a function is a black box and you can feed it elements of its domain and out come elements of its codomain. Hence given $A, B : \mathbf{Type}$ we introduce the type of functions $A \to B : \mathbf{Type}$. We can define a function explicitly, e.g. we define $f : \mathbb{N} \to \mathbb{N}$ as $f(x) :\equiv x + 3$. Having defined f we can apply it, e.g. $f(2) : \mathbb{N}$ and we can evaluate this application by replacing all occurrences of the parameter x in the body of the function $x + 3$ by the actual argument 2 hence $f(2) \equiv 2 + 3$ and if we are lucky to know how to calculate $2 + 3$ we can conclude $f(2) \equiv 5$.

One word about syntax: in functional programming and in Type Theory we try to save brackets and write $f\,2$ for the application and also in the definition we write $f\,x :\equiv x + 3$.

The explicit definition of a function requires a name but we should be able to define a function without having to give it a name - this is the justification for the λ-notation. We write $\lambda x.x + 3 : \mathbb{N} \to \mathbb{N}$ avoiding to have to name the function. We can apply this $(\lambda x.x+3)(2)$ and the equality $(\lambda x.x+3)(2) \equiv 2+3$

is called β-reduction. The explicit definition $f\,x \equiv x+3$ can now be understood as a shorthand for $f \equiv \lambda x.x+3$.

In Type Theory every function has exactly one argument. To represent functions with several arguments we use *currying*, that is we use a function that returns a function. So for example the addition function $g :\equiv \lambda x.\lambda y.x+y$ has type $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$, that is if we apply it to one argument $g\,3 : \mathbb{N} \to \mathbb{N}$ it returns the function that add 3 namely $\lambda y.3+y$. We can continue and supply a further argument $(g\,3)\,2 : \mathbb{N}$ which reduces

$$\begin{aligned}(g\,3)\,2 &\equiv (\lambda y.3+y)\,2\\ &\equiv 3+2\end{aligned}$$

To avoid the proliferation of brackets we decree that application is left associative hence we can write $g\,3\,2$ and that $\to$ is left associative hence we can write $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ for the type of g.

When calculating with variables we have to be a bit careful. Assume we have a variable $y : \mathbb{N}$ hanging around, now what is $g\,y$? If we naively replace x by y we obtain $\lambda y.y+y$, that is the variable y got *captured*. This is not the intended behaviour and to avoid capture we have to rename the bound variable that is $\lambda x.\lambda y.x+y \equiv \lambda x.\lambda z.x+z$ - this equality is called α-congruence. After having done this we can β-reduce. Here is the whole story

$$\begin{aligned}g\,y &\equiv (\lambda x.\lambda y.x+y)\,y\\ &\equiv (\lambda x.\lambda z.x+z)\,y\\ &\equiv \lambda z.y+z\end{aligned}$$

Clearly the choice of z here is arbitrary, but any other choice (apart from y) would have yielded the same result upto α-congruence.

3.3 Products and sums

Given $A, B : \textbf{Type}$ we can form their product $A \times B : \textbf{Type}$ and their sum $A + B : \textbf{Type}$. The elements of a product are tuples, that is $(a, b) : A \times B$ if $a : A$ and $b : B$. The elements of a sum are injections that is $\mathrm{left}\,a : A + B$ if $a : A$ and $\mathrm{right}\,b : A + B$, if $b : B$.

To define a function from a product or a sum it is sufficient to say what the functions returns for the constructors, that is for tuples in the case of a product or the injections in the case of a sum.

As an example we derive the tautology

$$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$$

using the propositions as types translation. We assume that $P, Q, R : \textbf{Type}$, we have to construct an element of the following type

$$\begin{gathered}((P \times (Q + R) \to (P \times Q) + (P \times R))\\ \times((P \times Q) + (P \times R) \to P \times (Q + R))\end{gathered}$$

We define:

$$\begin{aligned}
&f : P \times (Q + R) \to (P \times Q) + (P \times R)\\
&f\,(p, \mathrm{left}\, q) :\equiv \mathrm{left}\,(p, q)\\
&f\,(p, \mathrm{right}\, r) :\equiv \mathrm{right}\,(p, r)\\
\\
&g : (P \times Q) + (P \times R) \to P \times (Q + R)\\
&g\,(\mathrm{left}\,(p, q)) :\equiv (p, \mathrm{left}\, q)\\
&g\,(\mathrm{right}\,(p, r))) :\equiv (p, \mathrm{right}\, r)
\end{aligned}$$

Now the tuple (f, g) is an element of the type above.

In this case the two functions are actually inverses, but this is not necessary to prove the logical equivalence.

Exercise 1 *Using the propositions as types translation, try* [2] *to prove the following tautologies:*

1. $(P \wedge Q \implies R) \Leftrightarrow (P \implies Q \implies R)$
2. $((P \vee Q) \implies R) \Leftrightarrow (P \implies R) \wedge (Q \implies R)$
3. $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$
4. $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$
5. $\neg(P \Leftrightarrow \neg P)$

where $P, Q, R : \textbf{Type}$ *are propositions represented as types.*

Exercise 2 *While the principle of excluded middle* $P \vee \neg P$ *(*tertium non datur*) is not provable, prove its double negation using the propositions as types translation:*

$$\neg\neg(P \vee \neg P)$$

If for a particular proposition P *we can establish* $P \vee \neg P$ *then we can also derive the principle of indirect proof (*reduction ad absurdo*) for the same proposition* $\neg\neg P \implies P$*. Hence show:*

$$(P \vee \neg P) \implies (\neg\neg P \implies P)$$

However, the converse does not hold (what would be a counterexample?). However, use the two tautologies to show that the two principles are equivalent.

Functions out of products and sums can be reduced to using a fixed set of combinators called non-dependent eliminators or *recursors* (even though there

[2] I didn't say they are all tautologies!

is no recursion going on).

$$\begin{aligned}
&\mathrm{R}^{\times} : (A \to B \to C) \to A \times B \to C \\
&\mathrm{R}^{\times}\, f\, (a, b) :\equiv f\, a\, b \\
\\
&\mathrm{R}^{+} : (A \to C) \to (B \to C) \to A + B \to C \\
&\mathrm{R}^{+}\, f\, g\, (\mathrm{left}\, a) :\equiv f\, a \\
&\mathrm{R}^{+}\, f\, g\, (\mathrm{right}\, b) :\equiv g\, b
\end{aligned}$$

The recursor $\mathrm{R}^{\times}$ for products maps a curried function $f : A \to B \to C$ into its uncurried form, taking tuples as arguments. The recursor R^{+} basically implements the *case* function performing case analysis over elements of $A + B$.

Exercise 3 *Show that using the recursor* $\mathrm{R}^{\times}$ *we can define the projections:*

$$\begin{aligned}
&\mathrm{fst} : A \times B \to A \\
&\mathrm{fst}\, (a, b) :\equiv a \\
&\mathrm{snd} : A \times B \to B \\
&\mathrm{snd}\, (a, b) :\equiv b
\end{aligned}$$

Vice versa: can the recursor be defined using only the projections?

We also have the case of an empty product 1, called the unit type and the empty sum 0, the empty type. There is only one element of the unit type: $() : 1$ and none in the empty type. We introduce the corresponding recursors:

$$\begin{aligned}
&\mathrm{R}^{1} : C \to (1 \to C) \\
&\mathrm{R}^{1}\, c\, () :\equiv c \\
\\
&\mathrm{R}^{0} : 0 \to C
\end{aligned}$$

The recursor for 1 is pretty useless, it just defines a constant function. The recursor for the empty type implements the logical principle *eq falso quod libet*, from false follows everything. There is no defining equation because it will never be applied to an actual element.

Exercise 4 *Construct solutions to exercises 1 and 2 using only the eliminators.*

The use of arithmetical symbols for operators on types is justified because they act like the corresponding operations on finite types. Let us identify the number n with the type of elements $0_n, 1_n, \ldots (n-1)_n : \overline{n}$, then we observe that it is indeed the case that:

$$\begin{aligned}
\overline{0} &= 0 \\
\overline{m+n} &= \overline{m} + \overline{n} \\
\overline{1} &= 1 \\
\overline{m \times n} &= \overline{m} \times \overline{n}
\end{aligned}$$

Read = here as *has the same number of elements*. This use of equality will be justified later when we introduce the univalence principle.

The arithmetic interpretation of types also extends to the function type, which corresponds to exponentiation. Indeed, in Mathematics the function type $A \to B$ is often written as B^A. And indeed we have:

$$\overline{m^n} \quad = \quad \overline{n} \to \overline{m}$$