Thesis Seminar Talk 1

Theofanis Chatzidiamantis Christoforidis

April 2024

This is the first in a series of talks aiming to motivate and introduce the ideas behind homotopy type theory (HoTT) and higher category theory, and finally to bring the two together using the extension of HoTT commonly called simplicial type theory. We assume some prior exposure to formalized mathematics and a basic understanding of type theory (the intuition one gains from using Lean should be enough), as well as basic category theory.

1 A Vague Introduction to (a) Type Theory via Proof Assistants

For a proof checking system to work, we need a system of type theory and an interpretation of mathematical statements as types (given by extending the Curry-Howard correspondence). Then, a true statement is interpreted as a type that has a term. To prove something, we need to exhibit such a term in the corresponding type.

Let us compare two very popular proof assistants: *Coq* and *Lean* (the latter of which I assume everybody knows how to use). We will now see the ways in which they are similar, and later how they differ. Both are built on a formal system of type theory called the *Calculus of Inductive Constructions* (CIC), of which we now give a brief outline. Then, we modify this system, adding the types and axioms that we need for HoTT.

1.1 Notation. *We write $x : A$ for the expression "x is of type A", or, equivalently, "x is a term of A".*

All terms have a type. Every type has a type, called a ***sort***. We require an infinite hierarchy of sorts, consisting of:

- Prop, the type of *propositions.* This is not assumed in HoTT
- Set, the type of *(small) sets* (who would've thought).
- A sort Type_i for every $i \in \mathbb{N}$.

This is a well-founded hierarchy with Prop and Set at the base of the hierarchy. Both have type Type_1.

1.2 Remark. *Such an infinite hierarchy is required to avoid Russell-type paradoxes: As in set theory, there cannot be a "type of all types".*

1.3 Remark. *For these talks, we diverge from the notation used by functional programming languages and we do not follow the CIC presentation. Instead, we use a more "natural" system, as outlined in [UF13] and [Rij22]. The exact details of the formal CIC system can be found in the Coq and Lean documentation. In particular, there is a precise exposition of the additional axioms one can use to work analogously to classical logic and set theory.*

λ notation for functions. Maybe an example of an inductive type.

The Syntax

To specify a system of type theory, we need a collection of ***structural rules*** and ***type-forming rules***. The type theory that is of interest to us is ***Martin-Löf dependent type theory*** (MLTT). For a complete formal presentation of MLTT (and HoTT), we point to [UF13, Appendix A]. The nLab article on type theory has a quick informal introduction to the general syntax and categorical semantics of type theories and an extensive list of sources to learn more.

1.4 Definition.

- *A **context** is a finite list of declarations of the form $x : A$. We add an empty context $()$ with no declarations.*
- *A **judgement** is of the form $\Gamma \vdash \mathcal{J}$, where Γ is a context. In MLTT, the possible forms of judgements are*

$$\Gamma \text{ ctx} \qquad \Gamma \vdash A \text{ type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash A \equiv A' \text{ type} \qquad \Gamma \vdash a \equiv a' : A$$

Note the symbol "$\equiv$" : This expresses the notion of ***judgemental equality***, which will be the formal specification of *equality by definition*. We will revisit this later, as we will also present a different notion of equality, having inhabited identity types. Whether or not these coincide is the difference between *extensional* and *intensional* type theories, the former of which is both inconsistent with the axioms of HoTT and also has problems when it comes to type-checking (ref?).

1.5 Notation. *To specify the allowed steps in type-theoretic constructions and proofs, we express our **inference rules** in the form*

$$\frac{\mathcal{J}_1 \ \ldots \ \mathcal{J}_n}{\mathcal{C}}$$

where all the $\mathcal{J}_i$ and $\mathcal{C}$ are judgements.

Again, it is not in anyone's best interest to specify all of the basic inference rules. Instead, we give a short overview and some examples. These first rules are called the *structural rules* of MLTT. These fall into one of five categories:

add descriptions and examples. What about the generic elements rule in Rijke's book?

1. Reflexivity, transitivity and symmetry of judgemental equality.
2. Substitution:
3. Weakening:

4. Variable conversion:

5.

We also take a hierarchy of universes $\mathcal{U}_i$, $i \in \mathbb{N}$, with the extra assumption that the universes are *cumulative*, i.e., a type in a universe is also in the next one. In formal terms, we have two rules:

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}} \qquad \frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_{i+1}}$$

In the full presentation, it is ensured that every type belongs to a universe.

Type formation, introduction, elimination and computation rules. Empty, product and coproduct and non-dependent function types. Compare to category theory. Talk about substitution!

Translating logic, Part 1

Short section of correspondence of product/coproduct/function types and logical operators. Predicates as $A \to \mathcal{U}$

2 Type Families, Dependent and Identity Types

Using that foundation, we now outline the rest of the structure of dependent type theory. We will see how, in some cases, can view these new dependent types as analogues of classical logical operators, and how the introduction of identity types provides additional "higher" structure, which we can interpret homotopically. We keep following [UF13, Appendix A.2] with some extra insights from [Rij22, I.1-I.7] (and here is the obligatory link to the relevant nLab article).

Fix a universe $\mathcal{U}$.

2.1 Definition. *A **family** of types over $A : \mathcal{U}$ in context Γ is of the form*

$$\Gamma, x : A \vdash B(x) : \mathcal{U}$$

$\prod$-Types or ***dependent function types***. Intuitively, a term of the type $\prod_{x:A} B(x)$ assigns to every term $x : A$ a term of $B(x)$. We have a formation rule

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma, x : A \vdash B(x) : \mathcal{U}}{\Gamma \vdash \prod_{x:A} B(x) : \mathcal{U}}$$

and an introduction rule

$$\frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma \vdash \lambda(x : A).b(x) : \prod_{x:A} B(x)}$$

stating formally what we just described. We additionally have elimination, computation, and uniqueness rules, expressing that evaluating a function at a certain term works as expected and respects judgemental equality.

2.2 Remark. *Consider the case where the family is constant, i.e., a type B. Then, we obtain the ordinary function type*

$$A \to B :\equiv \prod_{x:A} B$$

In our formal system, this will be the definition of the function type. In other presentations we can also define this type without using families, so we can instead take the definition of a type family to be a term of type $A \to \mathcal{U}$.

2.3 Remark. *Using the (ordinary) function type, we can express functions with multiple parameters: A function $f : A \times B \to C$ corresponds precisely to a function $f : A \to (B \to C)$. This operation is called **currying** and is the preferred way of denoting such types.*

2.4 Notation. *We usually skip the parenthesis on the right when there is no ambiguity, writing $A \to B \to C$ for $A \to (B \to C)$.*

$\sum$-Types or ***dependent pair types***. The "classical equivalent" of these would be an indexed disjoint union. Let's take a look at the details: The formation rule works as with the dependent function type. Again, we highlight the introduction rule

$$\frac{\Gamma, x : A \vdash B(x) : \mathcal{U} \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \sum_{x:A} B(x)}$$

Note the difference from the rule of the $\prod$-type: Here, we only require some term $a : A$ to "produce" a term of B, and, if this is the case, we get a pair in the $\sum$-type (some logic-minded people may now see where this is going).

2.5 Remark. *As with the $\prod$-type and the ordinary function type, we have a case that falls out of the definition of $\sum$-types: If the family B is constant, then we define the **product***

$$A \times B :\equiv \sum_{x:A} B$$

We can also define projection maps

$$\begin{aligned} \mathsf{pr}_1 : A \times B &\to A \\ \mathsf{pr}_1(a, b) &:= a \end{aligned} \qquad \textit{and} \qquad \begin{aligned} \mathsf{pr}_2 : A \times B &\to B \\ \mathsf{pr}_2(a, b) &:= b \end{aligned}$$

2.6 Remark. *As the case of the product might hint at, we can define $\sum$-types as inductive types satisfying (...) In fact, the projection maps for the product are special cases of projection maps for the $\sum$-type (pr_2 is quite complicated to define). This type of property is what we use in practice, and it will become especially important in the case of identity types.*

outline the rules.

Translating logic, Part 2

For $\prod$-Types. Since an inhabited function type can be seen as a logical implication, we can revisit our explanation of the introduction rule and interpret it as such: For every $x : A$, we have a witness of $B(x)$. We have thus found our analogue of the statement: *for all* $x : A, B(x)$ *holds.*

For $\sum$-Types. Now do the same, this time with at least one term of A such that there is a term in $B(x)$. We have the analogue of: *there exists some* $x : A$ *for which* $B(x)$ *holds.* Note that the $\sum$-type is the collection of all such pairs of terms x and proofs of $B(x)$.

Identity Types and Path Induction

It is now time to look at identifications *internal* to our type theory. The key conceptual difference is that an identity type contains *all the possible identifications* of two terms of a given type (there can be more than one!). For two terms $x, y : A$ the identity type $x =_A y$ is defined inductively: We have a formation rule

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash x : A \quad \Gamma \vdash y : A}{\Gamma \vdash x =_A y : \mathcal{U}}$$

For the introduction rule, there is only one term we can construct by default in the identity type: The term $\mathsf{refl}_x : x =_A x$ corresponding to a trivial identification of a term with itself. Formally, the rule is

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash x : A}{\Gamma \vdash \mathsf{refl}_x : x =_A x}$$

If one checks the rest of the formal presentation, the elimination and computation rules are quite complicated. However, they are outlining a powerful: ***path induction***.

The statement is as follows [UF13, 1.12.1]:
Assume we have a family

$$\Phi : \prod_{x:A} \prod_{y:A} (x =_A y) \to \mathcal{U}$$

and a (dependent) function

$$\varphi : \prod_{x:A} \Phi(x, x, \mathsf{refl}_x)$$

Then there is a function

$$f : \prod_{x:A} \prod_{y:A} \prod_{(p:x=_A y)} \Phi(x, y, p)$$

that "extends" φ:

$$f(x, x, \mathsf{refl}_x) = \varphi(x)$$

In terms of proof tactics, this is saying: If I want to prove something for all $x, y : A$ and all paths between them, it is enough to consider the case in which both elements are x and the path is refl_x!

Warning: This is *not* saying that these are the only terms in the identity type. In fact, identity types can be complicated enough to allow us to reason about homotopy theory in HoTT. Of course, it is not immediately clear why we would actually want these types to have that kind of structure. This is the first point where the classical and the homotopical perspective diverge, and so do our two proof assistants from the start.

3 Uniqueness of Identity and Homotopy Type Theory

I don't know where to put this in the notes: A proposition is a type A with a term of type

$$\mathsf{isProp}(A) :\equiv \prod_{x,y:A} x =_A y$$

i.e., A is empty or contractible.

3.1 Definition. *A **set** is a type A with a term in the following proposition:*

$$\mathsf{isSet}(A) :\equiv \prod_{(x,y:A)} \prod_{(p,q:x=_A y)} (p = q)$$

A set is then viewed as a type for which all the higher structure of the identity types has been, in some sense, collapsed. Nothing is stopping us from adding this as an axiom for our entire system of type theory. We say that such a type theory has ***uniqueness of identity proofs*** (UIP).

3.2 Example. *Coq does not assume UIP. Lean 4 does.*

The advantage of UIP is that we can do mathematics more or less exactly like we are used to in the classical, set-theoretic foundations. However, there are reasons one might want to reject it. When working, for example, in category theory, we have the two different notions of an *isomorphism* and an *equivalence* of categories. Although the latter is weaker, it preserves the fundamental properties we are usually interested in. In this case, remembering the identification of two categories via an equivalence can allow us to exploit this.

We will now explore a related, but different reason: Under a certain interpretation, the structure of identity types provides a system of foundations that allow us to naturally work in a homotopy-invariant setting. Combined with Voevodsky's univalence axiom, HoTT becomes the type-theoretic equivalent of going from manually adding homotopical structure to topological spaces to working with ∞-categories and Kan complexes, with the homotopical information requiring no additional work.

The Minimal Amount of Topology

We now present a topological interpretation (and some facts) to justify what we just said. Consider a term $p : x =_A y$ in some identity type. This first proposition states that we can really think of such terms as *paths*, in the sense that they satisfy the same fundamental properties that paths do (*up to homotopy*) in topological spaces.

3.3 Proposition. *Let A be a type. There is a concatenation operator of type*

$$\prod_{x,y,z:A} (x =_A y \to (y =_A z \to x =_A z))$$

and an inverse operator of type

$$\prod_{x,y:A} (x =_A y \to y =_A x)$$

satisfying the relevant associativity, unit and inverse laws, with the unit being refl_x.

Let us look at the type

$$\sum_{x,y:A} (x =_A y)$$

This is then analogous to the concept of ***free path space***, i.e. the path space of a topological space not bound by any endpoints. In such a space, any path is homotopic to the constant path at one of its endpoints. There is also an analogous principle of ***based path induction***, which would work in types corresponding to the behavior of the *based path space*.

We now shift our focus to functions. The next proposition states that there is a functorial way to apply a function to a path.

3.4 Proposition. *Let $f : A \to B$, $g : B \to C$ be functions. there is an operation (**action on paths**)*

$$\mathsf{ap}_f : \prod_{x,y:A} (x =_A y \to f(x) =_B f(y))$$

together with operations

$$\mathsf{ap\text{-}id}_A : \prod_{x,y:A} \prod_{(p:x=_A y)} p =_{(x=_A y)} \mathsf{ap}_{\mathsf{id}_A}(p)$$

$$\mathsf{ap\text{-}comp}(f,g) : \prod_{x,y:A} \prod_{(p:x=_A y)} \mathsf{ap}_g(\mathsf{ap}_f(p)) =_{(g\circ f(x)=_C g\circ f(y))} \mathsf{ap}_{g\circ f}(p))$$

Once again, the key here is to iteratively define the operators, each time in the case where the relevant path is refl, and use path induction.

We can have an analogous operator in the dependent function type, for which we need:

3.5 Proposition (Transport.)**.** *Let A be a type and B a type family over A. There is an operation*

$$\mathsf{tr}_B : \prod_{x,y:A} (x =_A y \to (B(x) \to B(y))$$

3.6 Definition.

- *Let $f, g : A \to B$ be two functions. A **homotopy** between f and g is a dependent function of type*

$$(f \sim g) :\equiv \prod_{x:A} (f(x) = g(x))$$

- *A function $f : A \to B$ is an **equivalence** if there is an element in the type (actually proposition!)*

$$\mathsf{isequiv}(f) :\equiv (\sum_{g:B\to A} (f \circ g \sim \mathsf{id}_B)) \times (\sum_{h:B\to A} (h \circ f \sim \mathsf{id}_B))$$

3.7 Definition. *Let $f : A \to B$ be a function and $b : B$. We define the **fiber** of f at b to be the type*

$$\mathsf{fib}_f(b) :\equiv \sum_{x:A} (f(x) = b)$$

3.8 Theorem. *A function $f : A \to B$ is an equivalence if and only if it has contractible fibers, i.e., if there is an element of the type*

$$\prod_{b:B} \mathsf{isContr}(\mathsf{fib}_f(b))$$

The Univalence Axiom

The motivation here comes from another observation about how we reason in some areas of classical mathematics: It is common to identify two isomorphic (or homotopy equivalent) objects, often without being rigorous about it. In the process of formalization, this is just false. It is, however, possible to codify this as an axiom using the structure we have built so far.

3.9 Remark. *Let $\mathcal{U}$ be a universe. For any two types $A, B : \mathcal{U}$ we can define a map*

$$\mathsf{idtoequiv} : (A =_{\mathcal{U}} B) \to (A \simeq_{\mathcal{U}} B)$$

given by sending refl_A to id_A and applying transport and path induction.

3.10 Definition. *A universe $\mathcal{U}$ is **univalent** if, for all types $A, B : \mathcal{U}$, the function $\mathsf{idtoequiv}$ is an equivalence.*

3.11 Theorem. *Univalence implies function extensionality and propositional extensionality.*

In the full notes: talk about the universal properties implied by funext.

One very powerful result implied by univalence is the following type-theoretic analogue of the Grothendieck construction:

3.12 Theorem. *For any type $A : \mathcal{U}$ where $\mathcal{U}$ is a univalent universe, the map*

$$(\sum_{X:\mathcal{U}} (X \to A)) \to (A \to \mathcal{U})$$
$$(X, f) \mapsto \mathsf{fib}_f$$

is an equivalence.

Recall that we can see $A \to \mathcal{U}$ as the type of type families indexed by A. Then this result is saying that the full homotopical structure of such families is encoded in the type of maps into A.

A Glimpse Into Higher Inductive Types

The idea behind higher inductive types is that we use the structure of identity types and manually add our own paths with some specified behavior to define new types with homotopically interesting properties. In topology, this reminds us of the way we form CW-complexes. Recall that, by CW-approximation, these are all we need when reasoning about homotopy types up to weak equivalence.

Circle.

4 Truncations and Logical Aspects

Propositions, sets, "universal property" of the propositional truncation. Infinite level of truncations, related to homotopy theory. Note that types like isSet and isEquiv are propositions. AC and LEM related to truncation levels. Univalence contradicts UIP. If this takes me more than 10 minutes to write I have failed as a logician.

UP of prop. truncation: A type, $f : A \to P$, P proposition s.t. for any other proposition Q the precomposition map (w/f) is an equivalence of the function types.

References

[UF13] The Univalent Foundations Program, Homotopy type theory: Univalent foundations of mathematics. Institute for Advanced Study, 2013.

[Rij22] Egbert Rijke, Introduction to Homotopy Type Theory. arXiv:2212.11082.