HoTTEST Summer School 2022
Lecture 1
Dependent Type Theory and Π-Types

Lecture by Paige North
Notes by Peter Fortin

July 4 2022

Contents

1 Homotopy Type Theory

Homotopy type theory brings out connections between three different aspects of mathematics: foundations, formalization, and practice. There are deep connections between all of these fields, and HoTT gives a framework to view all of these connections from a central perspective. Fields such as type theory and functional programming give computational interpretations to HoTT. The connections between logic, set theory, and topos theory allow us to view HoTT as a foundation for all of mathematics. And finally, homotopy theory, category theory, topos theory, and other areas of abstract mathematics give us useful fields to use the tools and techniques that HoTT provides in order to do actual mathematics.

$$
\begin{tikzcd}
\text{Formalization of Mathematics} \arrow[dr] \arrow[dd] & & & \text{Foundations of Mathematics} \arrow[d] \arrow[dd, bend left] \arrow[ddd, bend left] \\
& \text{Type Theory} \arrow[dr] & & \text{Logic} \arrow[dl] \\
\text{Functional Programming} \arrow[rr] & & \text{Homotopy Type Theory} & \text{Set Theory} \arrow[l] \\
& \text{Homotopy Theory} \arrow[ur] & \text{Category Theory} \arrow[u] & \text{Topos Theory} \arrow[ul] \\
& & \text{Practice of Mathematics} \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}
$$

A Map of the Territory

2 Logic and Natural Deduction

Natural Deduction is a proof system for logic which uses rules of inference to create proofs of propositions. When referring to the formal system, we will call it **ND**. Propositions are declarative statements which can be either true or false. Proofs in **ND** are trees where each branch is the application and result of some rule.

Rules in **ND** always come in pairs: introduction and elimination. Each logical notion has a way of introducing it and a way of eliminating it. These rules require some sort of hypothesis. We represent this by writing what is assumed to be true above a vertical bar, and the conclusion, or what comes after the use of the rule, on the bottom of the bar. A proof is then a tree where the root is the conclusion and all of the premises are the end leaves.

We have two rules, $\times$ and $\to$. $P \times Q$ is conjunction, which means we are taking the propositions P and Q to both be true at the same time. $P \to Q$ is implication, which means we take the truth of P to imply the truth of Q.

$$\frac{P \qquad Q}{P \times Q}\ \times\text{-intro}$$

$$\frac{P \times Q}{P}\ \times\text{-elim-l} \qquad\qquad \frac{P \times Q}{Q}\ \times\text{-elim-r}$$

ND rules for $\times$

The $\times$-intro rule means that if we have proven P and we have proven Q, we can infer$P \times Q$. $\times$-elim-l means that if we have we have proven $P \times Q$, we can infer P alone, and $\times$-elim-r means the same, but for Q. The intuition is that once we have proven a conjunction, we can get either of the conjuncts by themselves, which is the inverse of how we prove the conjunction in the first place.

$$\dfrac{\begin{array}{c}[P]\\ \vdots\\ Q\end{array}}{P \to Q}\to\text{-intro} \qquad\qquad \dfrac{P \qquad P \to Q}{Q}\to\text{-elim}$$

ND rules for $\to$

The $\to$-intro rule means that under the assumption that P is true, if we then go on to legally infer Q, we can infer that $P \to Q$. It is important to understand that we can assume P to be true for the purposes of inferring an $\to$-intro even if we don't know that P is true. This is because we are not trying to prove P, we are trying to prove $P \to Q$. The $\to$-elim rule, which is called modus ponens, means that if we have proven P, and if we have proven that $P \to Q$, then we can put those two together to derive Q.

Introduction rules tell us how to produce a proof of some statement. elimination rules tell us what we can do with that statement (which, usually, is prove more statements).

It should be noted that if we think of types as sets and terms as elements, then conjunction and all of these rules behave exactly like the set theoretic Cartesian product. Additionally, conjunction has a lot of rules simply because it involves a left and a right side.

3 The Simply Typed Lambda Calculus

The Simply Typed Lambda Calculus (**STΛC**) is a model of computation, and is closely related to **ND**. **STΛC** is what we get when we make **ND** *proof relevant*. This means that we don't only care about the abstract idea that some proposition is true. Instead, we carry around explicit proofs of the truth along with the proposition.

In **ND** we write "P" to mean"P holds", but in **STΛC** we write "$p : P$" to mean "p is a proof/witness of P". We say p is a *term* of *type* P. Because types are their own concept in **STΛC**, we also have rules for determining that something has a specific type, and we call these "formation" rules, and they exist along with the corresponding introduction and elimination rules. Finally, because **STΛC** is a model of computation, we need computation rules to tell us how to compute with our terms and types. Computation rules often come in pairs as well, β and η rules. Below are the **STΛC** rules for $\times$:

$$\frac{P \text{ Type} \qquad Q \text{ Type}}{P \times Q \text{ Type}}\ \times\text{-form}$$

$$\frac{\Gamma \vdash p : P \qquad \Gamma \vdash q : Q}{\Gamma \vdash (p, q) : P \times Q}\ \times\text{-intro}$$

$$\frac{\Gamma \vdash a : P \times Q}{\Gamma \vdash \pi_1 a : P}\ \times\text{-elim-l} \qquad\qquad \frac{\Gamma \vdash a : P \times Q}{\Gamma \vdash \pi_2 : Q}\ \times\text{-elim-r}$$

$$\frac{\Gamma \vdash p : P \qquad \Gamma \vdash q : Q}{\Gamma \vdash \pi_1(p, q) \doteq p : P}\ \times\text{-comp-}\beta\text{-l}$$

$$\frac{\Gamma \vdash p : P \qquad \Gamma \vdash q : Q}{\Gamma \vdash \pi_2(p, q) \doteq q : Q}\ \times\text{-comp-}\beta\text{-r}$$

$$\frac{\Gamma \vdash a : P \times Q}{\Gamma \vdash (\pi_1 a, \pi_2 a) \doteq a : P \times Q}\ \times\text{-comp-}\eta$$

STΛC rules for $\times$

There are a few new symbols here. Γ is called a *context*. It is a variable which stands for a collection of hypotheses, which could be empty. The idea is that we don't want our rules to only apply when we have nothing but the terms involved in the proof; we want to be able to use our rules at any point during a proof. So Γ stands for whatever has been proven before, which could be nothing at all, e.g. if we're at the first step of our proof.

$\vdash$ is a meta way of expressing implication, similar to how we use the underline bar. When we write something like $\Gamma \vdash \Delta$, we are saying that the context Γ proves the context Δ. Often times, we will make some of the terms in either of the two contexts explicit, since they are the terms we want to manipulate. These "metavariables" exist to show that we are accounting for all of the proofs which have already been derived, even if that is an empty set. The $\times$-form rule tells us that, much like in **ND**, if we have that P is a type and Q is a type, then $P \times Q$ is a type.

π_1 and π_2 are projections. They represent the idea of taking the conjunction $a : P \times Q$ and "extracting" the P and the Q parts, respectively. The elimination rules tell us how we can turn an $a : P \times Q$ into its respective projections, that is, the parts of it which have types P and Q. The combination of the introduction and elimination rules tells us that if we have two types P and Q, we can always put them together and take them apart again.

The computation rules introduce the idea of *judgmental equality*. Our type rules, as well as any time we express that a term is of a type, e.g. $p : P$, we are making a *judgement* in our theory. The expression $\Gamma \vdash p \doteq q : P$ is expressing the judgement that p and q are judgmentally equal terms of the type P. Again, much more will be said about this later. As for the computation rules, $\times$-compβ-l says that if $(p, q) : P \times Q$, then the left projection $\pi_1(p, q) : P$ is judgmentally equal to $p : P$ itself. $\times$-compβ-r says the same but for the right projection.

Finally, $\times$-comp-η says that if $a : P \times Q$, then it is judgmentally equivalent to the composition of its left and right projections. These computational rules tell us how we can move between conjunctions and their constituent parts, and that this process is guided by their types. Below are the **STΛC** rules for $\to$:

$$\frac{P \text{ Type} \qquad Q \text{ Type}}{P \to Q \text{ Type}} \to\text{-form}$$

$$\frac{\Gamma, x : P \vdash q : Q}{\Gamma \vdash \lambda x.q : P \to Q} \to\text{-intro} \qquad \frac{\Gamma \vdash p : P \qquad \Gamma \vdash f : P \to Q}{\Gamma \vdash fp : Q} \to\text{-elim}$$

$$\frac{\Gamma, x : P \vdash q : Q \qquad \Gamma \vdash p : P}{\Gamma \vdash (\lambda x.q)p \doteq q[p/x] : Q} \to\text{-comp-}\beta$$

$$\frac{\Gamma \vdash f : P \to Q}{\Gamma \vdash \lambda x.fx \doteq f : P \to Q} \to\text{-comp-}\eta$$

STΛC rules for $\to$

The meaning of the formation rule should by now be obvious. The $\to$-intro rule introduces a new symbol λ, which is called lambda abstraction, and it also has an example of more than just Γ on the left hand side of a $\vdash$. What $\to$-intro says is that if the context Γ, *as well as* the context of $x : P$ proves $p : Q$, then we can *abstract* the term $x : P$ over the term $q : Q$ as a witness to the fact that a term of type P can be used to prove $q : Q$. The computational interpretation of this makes the situation much clearer. Similarly, $\to$-elim says that we can supply a witness $p : P$ to a witness $f : P \to Q$ to get a witness $fp : Q$.

The computational interpretation of these rules comes from $\to$-comp-β and $\to$-comp-η. $\to$-comp-β introduces another new symbol, $q[p/x] : Q$, which is called substitution. The idea is to take $q : Q$ as a syntactic object, which may or may not contain the syntactic object $x : P$, and then replace each occurrence of $x : P$ with $p : P$. $\to$-comp-β says that the λ-abstraction *applied to a term of the correct type*, is judgmentally equivalent to the term after the substitution takes place. $\to$-comp-η says that an implication is judgmentally equivalent to a λ-abstraction which takes a term of the right type and then applies the implication.

As was stated above, **STΛC** is a model of computation. These last two rules can be understood in a straightforwardly computational way. Imagine that $f : P \to Q$ is a function. $\to$-comp-β says that applying f to an argument of the right type will produce a result of the right type, and moreover, that the way the result is calculated is by substituting the free variable bound by the λ-abstraction. $\to$-comp-η says that a function $\lambda x.fx$ which takes an argument and applies f to that argument, is equivalent to f itself.

The one-to-one connection between **ND** and **STΛC** is known as the "Curry-Howard Correspondence", or "Propositions as Types", since it was discovered by Haskell Curry and William Howard. The correspondence, as the second name suggests, tells us that logic and computation are two sides of the same coin, and that propositions operate in the exact same way as types. Proofs of propositions correspond to programs which return the appropriate type. Under this correspondence, the logical view of $\to$ is as implication, and the computational view is as a function. This lets us use types as a means of program specification. We can turn a logical statement such as $P \to P$ into the specification of a program $(\lambda x.x) : P \to P$ which takes an argument of type P and produces a result of type P. Clearly, the only thing that this function can do is to return what it was given. So the logical idea of a proposition implying itself gives a specification for a program that returns its input. The term of the type being specified is the program witnessing it's proof.

Finally, we can now give a good explanation for notation like $\pi_1 a$. When we have a term with a function/implication type $P \to Q$ and we set it next to a term with type P, that should be taken to mean function application/modus ponens.

4 Dependent Type Theory

In **ND**, our deductions had no terms. That is, there was nothing "inside" of the propositon variables P and Q. The **STΛC** allowed us to interpret P as a type, and consequentally a judgement such as $p : P$ as asserting that the term p has type P. Our judgemnts in **STΛC** only allow us to make terms which depend on other terms. For example, $\to$-comp-β tells us that we can form terms such as $(\lambda x.x) : P \to P$ under the condition that $x : P$. However, the rules of **STΛC** *only* allow us to have terms depend on terms.

Dependent type theory allows us to also have *types* depend on terms, and as such it is an extension of the **STΛC**. Under the interpretation of types as propositions, dependent types are predicates. Under the interpretation of types as sets, dependent types are indexed families of sets. Under the interpretation of types as program types, dependent types are program type specifications which involve a parameter.

We carry over the same rules as in the **STΛC**, except now our type judgements also contain contexts,

$$\frac{\Gamma \vdash P \text{ Type} \qquad \Gamma \vdash Q \text{ Type}}{\Gamma \vdash P \to Q \text{ Type}} \to\text{-form}$$

$$\frac{\Gamma \vdash P \text{ Type} \qquad \Gamma \vdash Q \text{ Type}}{\Gamma \vdash P \times Q \text{ Type}} \times\text{-form}$$

The main concept dependent type theory adds to **STΛC** is the idea of a dependent function type, $\prod$.

5 Dependent Functions

As an example of a dependent function, imagine that we have a type of natural numbers $\mathbb{N}$, and a type $Vect$ of all finite vectors consisting of natural numbers.

We could define a function $0 : \mathbb{N} \to Vect$ which takes a natural number n as input and returns the vector of length n whose components are all 0. But since we know that the output vector will always be of a certain length, we also know that the vector isn't just an object in $Vect$, it's also an object in $Vect(n)$, that is, the type of vectors of length n. Dependent functions let us encode this special property of the function by specifiying how the output type of the function *depends* on the input term.

We have two ways of writing this, one is a "type theory" way, and one is a computer science way.

$$0 : \prod_{n:\mathbb{N}} Vect(n) \qquad\qquad 0 : (n : \mathbb{N}) \to Vect(n)$$

The logical interpretation of $\prod$ is as $\forall$, that is, it can be read as a universal quantifier. Dependent function types are generalizations of regular function types, because in a regular function type, the result type does not depend on a term from the input type. This is a special case of a dependently typed function where the return type does not depend on the input term, such as $0 : \prod_{n:\mathbb{N}} Vect$. The product type is also a special case of the dependent function type, but the proof requires more than what we have so far. This is important, however, because it shows the parsimony of definition in type theory. We normally don't define at the basic level individual types such as $\to$ or $\times$, since we can define them with $\prod$. Of course, though, we needed to define them for **ND** and **STΛC**, because $\prod$ does not exist in those languages. We conclude with the rules for $\prod$ types.

$$\frac{\Gamma, x : P \vdash Q \text{ type}}{\Gamma \vdash \prod_{x:P} Q \text{ type}} \prod\text{-form} \qquad\qquad \frac{\Gamma, x : P \vdash q : Q}{\Gamma \vdash \lambda x.q : \prod_{x:P} Q} \prod\text{-intro}$$

$$\frac{\Gamma \vdash f : \prod_{x:P} Q \qquad \Gamma \vdash p : P}{\Gamma \vdash fp : Q[p/x]} \prod\text{-elim}$$

$$\frac{\Gamma, x : P \vdash q : Q \qquad \Gamma \vdash p : P}{\Gamma \vdash (\lambda x.q)p \doteq q[p/x] : Q[p/x]} \prod\text{-comp-}\beta$$

$$\frac{\Gamma \vdash f : \prod_{x:P} Q}{\Gamma \vdash \lambda x.fx \doteq f : \prod x : PQ} \prod\text{-comp-}\eta$$

Dependent Type Theory rules for $\prod$