A Tour of Type Theory

This talk is ...

- Part hands-on / Lean tutorial
- Part introduction to foundational / theoretical aspects of Type Theory

Which means,

- It's not a *great* Lean tutorial
- It's not a perfect formal treatment of Type Theory

Set-Theoretic Foundations

Typical set-theoretic foundations:

- First order logic
  - Syntax
    - variables, logical connectives $\neg, \wedge, \rightarrow, \leftrightarrow, \forall, \exists$
    - binary relation $\in$
  - A deductive system
    - *Judgements* of the form "p is provable"
    - *Rules of inference* to infer judgements from other judgements
    - $$\textsf{modus-ponens}\,\frac{\Gamma \vdash a \qquad \Gamma \vdash a \rightarrow b}{\Gamma \vdash b}$$
    - Axioms of FOL
- Axioms of set theory ($\mathrm{ZF}, \mathrm{ZFC}, \mathrm{ZF}^{-}$, etc.)

Big picture:

$$\text{Deductive system of FOL} \to \text{Sets} \to \text{Mathematics}$$

Primitive concept = **logic** 📝

- Decide how logic and proof works 🤔
- ... Axiomatize things called sets
- ... Use sets to implement the rest of mathematics

Type-Theoretic Foundations

Type theory takes a fundamentally different approach:

$$\text{Type theory} \begin{array}{l} \to \text{Logic and proof} \\ \\ \to \text{Mathematics} \end{array}$$

Primitive concept = **computation** 🤖

- Start with a system of "well-typed computation"
    - Implement logic and proof (using Curry-Howard)
    - Implement the rest of mathematics

(Typed) Lambda Calculus

The formal system we will describe in the next few slides is called the *Calculus of Inductive Constructions*, an extension of lambda calculus.

Syntax

- Variables are terms
- If x is a variable, A is a term, and B_x, T_x are terms, then
  - $\lambda\ x : A, B_x$ is a term (function abstraction)
  - $\prod_{x:A} T_x$ is a term (dependent function **type constructor**)
- If A and B are terms, then AB is a term (application)

Deductive system

The primary judgements of our deductive system are "typing judgements" of the form

$$x_1 : A_1, \ldots x_n : A_n \vdash x : A$$

which asserts that, in the given context, the term x has type A.

Definitional (Computational) Equality

beta-reduction is the following rule:

$$(\lambda\ x : A, B_x)\ t \rightarrow_\beta B_x[t\ /\ x]$$

this captures "performing" function application by literally inserting the argument into the body in place of the variable.

The equivalence relation $\equiv$ generated by this rule will be called *definitional equality*, and gives rise to a type of judgement

$$A \equiv B$$

that A and B are "definitionally equal terms".

We also declare the inference rule

$$\textsf{conv}\,\frac{\Gamma \vdash x : A \qquad \Gamma \vdash B : \textsf{Type} \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash x : B}$$

We will see that the assertion/judgement

$$x : A$$

will eventually be interpreted in all of the following ways:

- a is a term of type A
- When we implement mathematics...
  - a is an "element" of A $(A = \mathbb{N}, \mathbb{Q}, \mathrm{CptHaus}, C^* \mathrm{Alg}, \ldots)$
- When we implement logic...
  - a is a proof of A
  - a is a witness to A

Dependent Function Type

CIC has an infinite heirarchy of "universes":

$$\underbrace{\mathsf{Prop}}_{\text{logic}} : \underbrace{\mathsf{Type}_1 : \mathsf{Type}_2 \ldots}_{\text{data}}$$

The dependent function type constructor is governed by the following rule:

$$\mathsf{Pi}\ \frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma; x : A \vdash B_x : \mathsf{Type}_j}{\Gamma \vdash \prod_{x:A} B_x : \mathsf{Type}_{\max(i,j)}}$$

When B_x is constant (does not depend on x), the dependent function is just the type of "regular" functions from A to B, and we write

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

Terms of a function type are constructed with lambda abstraction:

$$\textsf{lam}\ \frac{\Gamma \vdash \prod_{x:A} B_x : \textsf{Type} \qquad \Gamma; x : A \vdash b_x : B_x}{\Gamma \vdash \lambda\ x : A, b_x : \prod_{x:A} B_x}$$

and come with the following application typing rule:

$$\textsf{app}\ \frac{\Gamma \vdash f : \prod_{x:A} B_x \qquad \Gamma \vdash t : A}{\Gamma \vdash f\ t : B_x[t\ /\ x]}$$

We want to interpret Prop as the type of logical propositions. If $p : \mathsf{Prop}$, we want adopt the point of view that

$x : p$ means "x is a proof of p"

And, more generally

"p is inhabited" means "p is true"

Finally,

"proving p" means "exhibiting a term of type p"

Under this point of view, we naturally get the *Brouwer-Heyting-Kolmogorov* interpretation of intuitionistic logic, as will be explained.

Under the BHK interpretation,

A proof that p implies q

should mean

A process transforming proofs of p into proofs of q

(Non-dependent) functions types capture exactly this: If $p, q : \mathsf{Prop}$, then

$$\prod_{h:p} q \equiv p \to q : \mathsf{Prop}$$

is the proposition representing "p implies q", and terms $f : p \to q$ are exactly functions taking proofs of p and returning proofs of q.

Under the BHK interpretation

A proof of "for all $x, p(x)$"

Should mean

A process transforming objects x into proofs of $p(x)$

Which is where we need *dependent* function types. Indeed, if $A : \mathsf{Type}$ and for each $x : A$ we have $p_x : \mathsf{Prop}$,

$$\prod_{x:A} p_x : \mathsf{Prop}?$$

represents the proposition "for all $x : A, p_x$", and terms $f : \prod_{x:A} p_x$ are functions taking objects x of type A to proofs (terms) of p_x.

Why Differentiate Prop?

Our typing rule for the dependent function type says that the universe of types is *predicative*:

$$\text{Pi}\ \frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma; x : A \vdash B_x : \mathsf{Type}_j}{\Gamma \vdash \prod_{x:A} B_x : \mathsf{Type}_{\max(i,j)}}$$

but, we want the type on the previous slide to "remain" a proposition! So, we make Prop *impredicative* with the following special rule:

$$\text{Pi-prop}\ \frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma; x : A \vdash B_x : \mathsf{Prop}}{\Gamma \vdash \prod_{x:A} B_x : \mathsf{Prop}}$$

This way, under our intrepretation of the type Prop, statements like "for all $n : \mathbb{N}$, ..." are propositions.

Inductive Types

(non)-dependent function is the only type constructor built into our system. The rest can be created using *inductive definitions*. Roughly, an inductively defined type I is

- A collection of constructors $A_1 \to A_2 \to \cdots \to A_n \to I$, functions that create terms of type I.
- The *only* terms of type I are those created through its constructors (An inductive type is just the initial algebra of an endofunctor, what's the problem?)

This is not only how we will define mathematical objects, like

```
inductive ℕ : Type
| zero : ℕ
| succ : ℕ → ℕ
```

(A natural number is either zero or the successor of another natural number, and nothing more)

But also the rest of the logical connectives

In the BHK interpretation, a proof of "p and q" should be a "a proof of p together with a proof of q":

```
inductive and (p : Prop) (q : Prop) : Prop
| intro : p → q → and
```

This actually results in

```
and       : Prop → Prop → Prop                     -- Type constructor
and.intro : Π p q : Prop, p → q → and p q          -- Term constructor
```

So, given p, q : Prop, there is another proposition **and** $p\ q$: Prop and proofs of this proposition are made from a proof of p and a proof of q.

Similarly, a proof of "p or q" should be a either a proof of p, or a proof of q:

```
inductive or (p : Prop) (q : Prop) : Prop
| inl : p → or
| inr : q → or

or.inl : Π p q : Prop, p → or p q
or.inr : Π p q : Prop, q → or p q
```

We can define the propositions `true : Prop` and `false : Prop` inductively:

```
inductive true : Prop
| intro : true

inductive false : Prop
```

`true` has a constructor that takes no arguments, while `false` has no constructors!

What about "not p"?

Under the BHK interpretation, to claim that "p is false" should mean

$$p \text{ cannot be true}$$

which should in turn mean

$$p \text{ implies absurdity}$$

And therefore we define

$$\neg p \equiv p \to \textsf{false}$$

Finally, a proof of "there exists x such that p_x" should mean

An object x together with a proof of p_x

So we have

```
inductive Exists (α : Type) (p : α → Prop) : Prop
| intro (w : α) (h : p w) : Exists
```

(The Type version is called a **dependent pair** in HoTT and other MLTT type theories)

Let's Prove some Propositional Logic!

Identity

"Equality" is also an inductively defined *proposition*:

```
eq : Π {α : Type}, α → α → Prop
```

With only one constructor:

```
eq.refl : ∀ {α : Type} (a : α), a = a
```

- This means that the only "proof" that two things are equal is by "reflexivity", i.e., they are (reduce to) the same thing.
- The system of inductive types ensures that equality has the behavior you expect
    - A proof of $x = y$ means they can be substituted essentially anywhere they appear
    - This is done heavily by the `rewrite` tactic in Coq and Lean

Compare the following proof to traditional foundations:

```
example : 2 + 2 = 4 := refl 4
```

("proof by computation")

Axioms (Who needs 'em?)

All we've seen so far is just consequences of our deductive system of types.

But our type theory supports classical reasoning through the addition of *axioms*. In type theory,

$$\text{"axiom"} = \text{declared but } \textbf{unimplemented} \text{ function}$$

Thus, any function that invokes an axiom is not really a "program", since there is no implementation of the axiom!

Proofs are Programs?

The big picture for CIC (Coq and Lean)...

- Prop = Logic and reasoning, Type = Data and Programs
- We can safely add axioms in Prop like excluded middle
  - to make our reasoning classical
  - to make our reasoning more expressive / easier
  - this means "proofs are not programs" 😔... 🤔
  - CS POV: Write programs, prove their correctness, but "discard" proofs
- The only axiom that has **catastrophic** consequences for programs is our pal, Axiom of Choice:
  - Magically produce a term of any nonempty Type! 🧙🏽‍♀️
  - Definitions/theorems that invoke AC must be marked with `noncomputable` 😮

Let's take a closer look:

<👋🏼> If we think of our impredicative universe Prop as a "set"... then the dependent product rule for Prop says it is closed under *arbitrary* dependent products. In other words, given a family of sets $B : A \to \mathsf{Prop}$, the product $\prod_{x \in A} B(x)$ is still in Prop, *no matter how large A is*. This is only possible if

$$\mathsf{Prop} \text{ "is" } \{\varnothing, \{\bullet\}\}$$

</👋🏼>

This suggests that it is at least consistent to add *propositional extensionality*

```
axiom propext {a b : Prop} : (a ↔ b) → a = b
```

Proof irrelevance (don't distinguish only on the content of proofs)

```
axiom proof_irrel : ∀ {a : Prop} (h₁ h₂ : a), h₁ = h₂
```

And *excluded middle*

```
axiom em : ∀ (p : Prop), p ∨ ¬p
```

Choice

The following function is illegal 👮🏾‍♂️ 🚨

```
def choice_ish : (∃ n : ℕ, true) → ℕ :=
λ ⟨n, hn⟩, n
```

- We cannot build *data* (i.e. Type stuff) out of a *proposition*
  - If Prop is completely constructive, this is fine, but...
  - If the world of Prop is classical, we can construct terms of $\exists x : A, p x : \text{Prop}$ nonconstructively
  - So extracting the data doesn't make sense.
  - The Type-level synonym of $\exists$ is the *dependent pair* type, which is always constructive.

Impredicative Prop + Large elimiation + EM = 💥

- We can only use *propositions* to build other *propositions* (stay in Prop)

Choice is exactly the axiom that allows you to "large eliminate" existential propositions.

```
axiom choice {α : Sort u} : (∃ n : α, true) → α
```

We can invoke it for our previous function:

```
noncomputable def just_do_it : (∃ n : ℕ, true) → ℕ :=
λ h, classical.some h
```

- At this point, you are completely classical
- Any function, isomorphism, bijection, you build that depends on choice is no longer an actual function that can be run.
- But the fact that our proofs type-check is still a verification that it's a "valid proof".
- mathlib's approach: it's easier to be classical 🤷🏻‍♀️
  - Entire files can be marked `noncomputable`
  - Can formalize strictly classical mathematics (Zorn's lemma, ZFC-style ordinals/cardinals, etc.)

Draw the Rest of the Owl 🦉

All the rest of the *objects* we talk about in mathematics are implemented in the Type universe with inductive types.

Expressing that the type α is a *group* amounts to creating an inductive type called $\mathsf{group}\ \alpha$, where a term is made from a binary operation and proofs of the relevant propositions:

```
inductive group (α : Type)
| mk : ((*) : α → α → α) → (∀ a b c : G, a * b * c = a * (b * c)) → ... → group
```

But this is very unwieldy. Lean implements a bunch of special behavior called *type classes* that make this sort of thing easier

Tactics

There are also some higher level tactics available in tactic mode that implement actual *automation*

- `simp`
  - Relies on a library of "simp" theorems in the standard library of the form $a \leftrightarrow b$ or $a = b$
    - Does magic on the current goal
- `linarith`
  - Automatically proves goals that are simple linear inequalities given the right hypotheses are in the context
    - Shows mathlib's nonconstructive approach: `linarith` actually negates the goal and performs a search for contradictions!
- `group`/`ring`
  - Implement "normal form" routines for group and ring stuff, solve ring arithmetic goals

What about HoTT?

- HoTT is a plain intuitionistic type theory; discards Prop
- HoTT distinguishes *particular* types as *mere propositions*, they are exactly the types that only have one inhabitant:

$$\prod_{x:A}\prod_{y:A} x =_A y$$

- Among that types that are not (automatically) mere propositions are
  - (or) $a + b$ (comes with information about *which* one is true!)
  - (dependent pair) $\Sigma_{x:A} B_x$ (comes with an actual object x)
  - **identity types**
- There is a general operation of *propositional truncation* that "turns things into" mere propositions

$$\|\Sigma_{x:A} B_x\| = \exists_{x:A} B_x$$

- Our logic effectively declared that the equality type $x =_A y$ is always a *mere proposition*, but it does not have to be.

- By removing this shackle from identity types, HoTT views the type $x =_A y$ as the "type of equivalences between x and y in type A".

- If $p : x =_A y$ and $q : x =_A y$ are equivalences between x and y, we can ask if *they* are equivalent in the type $x =_A y$. That is,

  $$p =_{x =_A y} q$$

  is the type of equivalences between p and q.

Effectively, HoTT adds the following possible interpretation of a typing judgement

$x : A$	means	x is a point of the space A

- Under this interpretation, $x =_A y$ is the type of "paths betweeen x and y in the space A"
- $p =_{x=_A y} q$ is the type of "paths between paths between x and y"
- Identity types have the structure of a *higher groupoid*
- Leads to "Synthetic Homotopy Theory"

goals accomplished 🎉

👨‍🏫 $\longrightarrow$ 🐴