# HOTT

Riley Moriss

September 29, 2026

# Contents

Dependent Type Theories

HOTT

References: [ML], [Hof97] (for the type theory), [Hom], [Rij22] (for the homotopy).

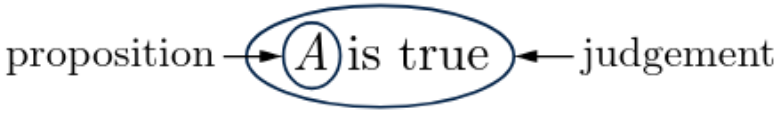
# History

| | |
|---|---|
| 1908 | Russell introduces type theory in the great foundation war. |
| 1933 - 40 | Church introduces the untyped and simply typed lambda calculus. |
| 1934-50 | Curry and others observe similarities between lambda calculus and logic. |
| 1936-43 | Gentzen introduces natural deduction and sequent calculus. |
| 1969 | Howard writes down the "Curry-Howard" isomorphism from sequent calculus to lambda calculus. |
| 1975-84 | Per Martin-Lof works on dependent type theories for constructive mathematics (MLTT). |
| 1996 | Hofman-Streicher find a model of MLTT in which the identity types have multiple inhabitants. |
| 2006-09 | Awodey-Warren and Voevodsky independently discover a model of MLTT in the homotopy category of topological spaces. Voevodsky proves that this model satisfies univalence. |
| 2010 | HOTT is created as MLTT with univalence assumed. |
| 2019 | Shulman proves that any $\infty$-topos is a model of HOTT. |

# Dependent Type Theories

# Philosophy

Type theories are formal systems that deal with the logic of deriving *judgements*, typically of the form $x$ is type $\sigma$, in a given *context*.




Propositions are combined using logical operators $\wedge, \vee, \neg$ etc.
Propositions are what are held to be true when we make a judgement.

Thus type theory makes explicit the judgemental nature of mathematics, when we write a theorem it is not merely "look a wff" it is the assertion that it is true. This is the addition of types to the syntax, we cannot merely hide the idea that everything in the syntax "makes sense" we internalise it to the syntax itself (e.g. the judgements $A$ *cntx* below).

# Alphabets

The first thing that we need is an alphabet of variables, we will use the latin letters from the end of the alphabet $\{x, y, z, \cdots\}$ and assume that we have an unbounded amount. We also assume a disjoint alphabet of constants, things like $\{., :, \Sigma, \Pi, 0, \mathrm{Suc}, \mathbb{N}, \dots\}$.

# Presyntax

We use the Backus–Naur notation to define the pre-contexts, pre-types and pre-terms:

$$
\begin{array}{lll}
\Gamma & ::= & \diamond \\
 & & \mid \Gamma, x\colon \sigma \qquad \text{provided } x \text{ is not declared in } \Gamma \\
\sigma, \tau & ::= & \Pi x\colon \sigma.\tau \mid \Sigma x\colon \sigma.\tau \mid Id_\sigma(M,N) \mid \mathbf{N} \\
M, N, H, P & ::= & x \mid \lambda x\colon \sigma.M^\tau \mid App_{[x:\sigma]\tau}(M,N) \mid \\
 & & Pair_{[x:\sigma]\tau}(M,N) \mid \mathsf{R}^{\Sigma}_{[z:(\Sigma x:\sigma.\tau)]\rho}([x\colon \sigma, y\colon \tau]H, M) \mid \\
 & & Refl_\sigma(M) \mid \mathsf{R}^{Id}_{[x:\sigma,y:\sigma,p:Id_\sigma(x,y)]\,\tau}([z\colon \sigma]H\ , M, N, P) \mid \\
 & & 0 \mid Suc(M) \mid \mathsf{R}^{\mathbf{N}}_{[n:\mathbf{N}]\sigma}(H_z, [n\colon \mathbf{N}, x\colon \sigma]H_s, M)
\end{array}
$$

# Presyntax

One should notice that the pre-sorts are recursively and interconnectedly defined, the base cases for these recursions are highlighted:

$$
\begin{array}{lll}
\Gamma & ::= & \diamond \\
 & & \mid \Gamma, x{:}\sigma \qquad \text{provided } x \text{ is not declared in } \Gamma \\
\sigma, \tau & ::= & \Pi x{:}\sigma.\tau \mid \Sigma x{:}\sigma.\tau \mid Id_\sigma(M,N) \mid \mathbf{N} \\
M, N, H, P & ::= & x \mid \lambda x{:}\sigma.M^\tau \mid App_{[x:\sigma]\tau}(M,N) \mid \\
 & & Pair_{[x:\sigma]\tau}(M,N) \mid \mathsf{R}^{\Sigma}_{[z:(\Sigma x:\sigma.\tau)]\rho}([x{:}\sigma, y{:}\tau]H, M) \mid \\
 & & Refl_\sigma(M) \mid \mathsf{R}^{Id}_{[x:\sigma,y:\sigma,p:Id_\sigma(x,y)]\tau}([z{:}\sigma]H\ , M, N, P) \mid \\
 & & 0 \mid Suc(M) \mid \mathsf{R}^{\mathbf{N}}_{[n:\mathbf{N}]\sigma}(H_z, [n{:}\mathbf{N}, x{:}\sigma]H_s, M)
\end{array}
$$

# Presyntax

The notation is "context $\vdash$ judgement"; that is the judgement on the right of the turnstile is so judged in the context of the left of the turnstile. The presyntax for judgements is

| | |
|---|---|
| $\vdash \Gamma \ ctxt$ | $\Gamma$ is a valid context |
| $\Gamma \vdash \sigma$ type | $\sigma$ is a type in context $\Gamma$ |
| $\Gamma \vdash M : \sigma$ | $M$ is a term of type $\sigma$ in context $\Gamma$ |
| $\vdash \Gamma = \Delta \ ctxt$ | $\Gamma$ and $\Delta$ are definitionally equal contexts |
| $\Gamma \vdash \sigma = \tau$ type | $\sigma$ and $\tau$ are definitionally equal types in context $\Gamma$ |
| $\Gamma \vdash M = N : \sigma$ | $M$ and $N$ are def. equal terms of type $\sigma$ in context $\Gamma$. |

# Structural rules

Now we need to know which of these pre-sorts are well typed and well formed. This is why we used Beamer so we can just screenshot all the deduction rules. Before adding any the the specific types like $\Pi, \Sigma, \mathbb{N}$ these are the rules that *define* contexts, types and terms being well formed in general. Thus a context is valid, well formed etc iff it can be derived by a series of deductions; similar for others. Here is a wall of rules

- Rules for context formation:

$$\frac{}{\vdash \diamond \ ctxt}\ \text{C-Emp} \qquad \frac{\Gamma \vdash \sigma \text{ type}}{\vdash \Gamma, x{:}\sigma \ ctxt}\ \text{C-Ext}$$

$$\frac{\vdash \Gamma = \Delta \ ctxt \qquad \Gamma \vdash \sigma = \tau \text{ type}}{\vdash \Gamma, x{:}\sigma = \Delta, y{:}\tau \ ctxt}\ \text{C-Ext-Eq}$$

  The variables $x$ and $y$ in rules C-Ext and C-Ext-Eq are assumed to be fresh.

- The variable rule

$$\frac{\vdash \Gamma, x{:}\sigma, \Delta \ ctxt}{\Gamma, x{:}\sigma, \Delta \vdash x : \sigma}\ \text{Var}$$

- Rules expressing that definitional equality is an equivalence relation:

$$\frac{\vdash \Gamma \ ctxt}{\vdash \Gamma = \Gamma \ ctxt}\ \text{C-Eq-R}$$

$$\frac{\vdash \Gamma = \Delta \ ctxt}{\vdash \Delta = \Gamma \ ctxt}\ \text{C-Eq-S}$$

$$\frac{\vdash \Gamma = \Delta \ ctxt \qquad \vdash \Delta = \Theta \ ctxt}{\vdash \Gamma = \Theta \ ctxt}\ \text{C-Eq-T}$$

$$\frac{\Gamma \vdash \sigma \text{ type}}{\Gamma \vdash \sigma = \sigma \text{ type}} \quad \text{Ty-Eq-R}$$

$$\frac{\Gamma \vdash \sigma = \tau \text{ type}}{\Gamma \vdash \tau = \sigma \text{ type}} \quad \text{Ty-Eq-S}$$

$$\frac{\Gamma \vdash \sigma = \tau \text{ type} \qquad \Gamma \vdash \tau = \rho \text{ type}}{\Gamma \vdash \sigma = \rho \text{ type}} \quad \text{Ty-Eq-T}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M = M : \sigma} \quad \text{Tm-Eq-R}$$

$$\frac{\Gamma \vdash M = N : \sigma}{\Gamma \vdash N = M : \sigma} \quad \text{Tm-Eq-S}$$

$$\frac{\Gamma \vdash M = N : \sigma \qquad \Gamma \vdash N = O : \sigma}{\Gamma \vdash M = O : \sigma} \quad \text{Tm-Eq-T}$$

- Rules relating typing and definitional equality:

$$\frac{\Gamma \vdash M : \sigma \qquad \vdash \Gamma = \Delta \ ctxt \qquad \Gamma \vdash \sigma = \tau \text{ type}}{\Delta \vdash M : \tau} \quad \text{Tm-Conv}$$

$$\frac{\vdash \Gamma = \Delta \ ctxt \qquad \Gamma \vdash \sigma \text{ type}}{\Delta \vdash \sigma \text{ type}} \quad \text{Ty-Conv}$$

- For convenience (cf. E2.7 below) we also introduce the following weakening and substitution rules where $\mathcal{J}$ ranges over one of the judgements $M : \sigma$, $\sigma$ type, $M = N : \sigma$, $\sigma = \tau$ type.

$$\frac{\Gamma, \Delta \vdash \mathcal{J} \qquad \Gamma \vdash \rho \text{ type}}{\Gamma, x{:}\,\rho, \Delta \vdash \mathcal{J}} \quad \text{Weak}$$

$$\frac{\Gamma, x{:}\,\rho, \Delta \vdash \mathcal{J} \qquad \Gamma \vdash U : \rho}{\Gamma, \Delta[U/x] \vdash \mathcal{J}[U/x]} \quad \text{Subst}$$

Here $\mathcal{J}[U/x]$ (and similarly $\Delta[U/x]$) denotes the capture-free substitution of $U$ for $x$ in $\mathcal{J}$. This means that bound variables in $\mathcal{J}$ are systematically renamed so as to prevent any free variables in $U$ from becoming bound in $\mathcal{J}[U/x]$. We will henceforth consider all contexts, types, and terms as equal if they agree up to names of bound variables and assume the existence of a capture-free substitution function on these equivalence classes. One can

# Type Formers

In this fragment one can see that the only rule without a hypothesis is the introduction of the empty context. Thus the only provable statements have hypotheses or are that the empty context is a context and that it is equal to itself. So far the only types that one can actually construct in this fragment are those that are given in the context. The only contexts that we can construct are the empty context and those coming from types. We cant really do anything. In particular the formation of types need to be axiomatised in for various types of interest.

These are type formers, more deduction rules, introduction, elimination and equality, which make types appear in our judgements (and at least once from an empty context). MLTT has the following type formers:

- Dependent function types
- Natural Numbers
- Inductive types
  - Unit
  - Empty
  - Coproduct
  - Identity type
- Universes

# Type Formers

Most type formers have several rules accompanying them,

- a **formation rule**, stating when the type former can be applied;
- some **introduction rules**, stating how to inhabit the type;
- **elimination rules**, or an induction principle, stating how to use an element of the type;
- **computation rules**, which are judgmental equalities explaining what happens when elimination rules are applied to results of introduction rules;

we will give a couple of the formation rules to see the vibes. The rest can be found in the references.

# Universes

We ensure that our alphabet includes the symbols $\mathcal{U}_i$ for each $i$ and then we add the following inference rules to our type theory

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

This type will also come with many other rules expressing compatibility, closure etc under other introduced types.
This type models set theoretic universes and will be set up in such a way that every type is an element of one of the universe types.

# Natural Numbers

The natural number type is introduced through the rules

$$\frac{\vdash \Gamma \ ctxt}{\Gamma \vdash \mathbf{N} \text{ type}} \text{ N-F} \qquad \frac{\vdash \Gamma \ ctxt}{\Gamma \vdash 0 : \mathbf{N}} \text{ N-I-0} \qquad \frac{\Gamma \vdash M : \mathbf{N}}{\Gamma \vdash Suc(M) : \mathbf{N}} \text{ N-I-S}$$

There are also a couple other rules that allow the elimination and equality to be expressed in this type.

# Function Types

The "function space" type, should be thought of as a function from $A \to B$, but the codomain can technically vary with the input

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

which says if I have a function $b(x)$ of $x$ and $x$ is type $A$ then I have a function type given by $\lambda x.b(x)$; it is just a rewriting of dependence into a function type. Note that if $A(x)$ and $B(x)$ dont depend on $x$ then we really get an 'ordinary' function type $A \to B := \prod_{(x:A)} B$.

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

Which expresses functional evaluation. There are other rules for $\beta$ and $\eta$ reduction of the $\lambda$ terms.

# Dependent Sum

This is the disjoint union analogue

$$\frac{\Gamma \vdash \sigma \text{ type} \quad \Gamma, x{:}\sigma \vdash \tau \text{ type}}{\Gamma \vdash \Sigma x{:}\sigma.\tau \text{ type}}\ \Sigma\text{-F} \qquad \frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \tau[M/x]}{\Gamma \vdash Pair_{[x:\sigma]\tau}(M, N) : \Sigma x{:}\sigma.\tau}\ \Sigma\text{-I}$$

$\Sigma x : \sigma.\tau$ should be written as $\Sigma_{x:\sigma}\tau(x)$.

# Identity

So far we can make identity judgements, but this is internalised to being contextual with the identity type,

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash Id_\sigma(M, N)\ \mathrm{type}}\ \text{Id-F}$$

which has an inhabitant given by reflexivity

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash Refl_\sigma(M) : Id_\sigma(M, M)}\ \text{Id-I}$$

and a couple other rules expressing how to eliminate and generate elements from reflexivity. It was thought for a long time that it should be provable that identity types in MLTT have a unique inhabitant. This was answered negatively in 96.

PROVE SOME SIMPLE STATEMENT HERE

# LEAN

This is a LEAN seminar I guess so the relevance is this: LEAN is built on COC "Calculus of constructions". This is another dependent type theory that I think shares the same structural fragment and presyntax. The type formers are where the two differ. For LEAN there exists non-hierarchical (non predictive) type universes (apparently). But they are largely similar, LEAN has the universes on top of the non-predictive setting.

# HOTT

# Beginnings of HOTT

What we have described so far is Martin-Lof type theory (MLTT). HOTT adds the extra assumption on the identity types:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash \mathsf{univalence}(A, B) : \mathsf{isequiv}(\mathsf{idtoeqv}_{A,B})}\ \mathcal{U}_i\text{-UNIV}$$

as well as ad hoc "higher inductive types" that may or may not be added (HOTT is more of an idea).

Note functional extensionality is implied by univalence.

# Univalence

[Hom, Lem 2.10.1] For any two types in the same universe $A, B : \mathcal{U}_i$ then we have the identity type $\mathrm{Id}_{\mathcal{U}}(A, B)$ or otherwise written as $A =_{\mathcal{U}} B$. Given two function types $f, g : \prod_{x:A}(A \to \mathcal{U})$, then a homotopy is a function type

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

Notice that this is basically the statement that the two functions agree point wise, which is different from that the two types are equal (but agrees with our normal notion). A quasi-inverse type is then a two sided inverse up to specified homotopies.

isequiv is a type such that any two inhabitants are equal and that admits maps to and from the quasi-inverse type. The homotopy equivalent type

$$(A \simeq B) :\equiv \sum_{f:A\to B} \text{isequiv}(f)$$

is then the collection of all equivalences, from $A$ to $B$.
A function type

$$\text{idtoequiv} : (A =_{\mathcal{U}} B) \to (A \simeq B)$$

can be constructed, intuitively this is just 'forgetting' the equality to an equivalence. Univalence then witnesses that this map is in fact itself an *equivalence*. Otherwise written:

$$(A = B) \simeq (A \simeq B).$$

# Inductive Types

The preceding fragment of MLTT and HOTT are often extended with new types. For HOTT these are the "higher inductive types".
But first what are the inductive types. MLTT has several, but the most obvious is the natural numbers. These are given by two "constructors" (the introduction rules for 0 and $\mathrm{Suc}$) the induction principle (the elimination rule)

$$\frac{\begin{array}{c}\Gamma, n{:}\,\mathbf{N} \vdash \sigma \text{ type} \\ \Gamma \vdash H_z : \sigma[0/n] \\ \Gamma, n{:}\,\mathbf{N}, x{:}\,\sigma \vdash H_s : \sigma[Suc(n)/n] \\ \Gamma \vdash M : \mathbf{N}\end{array}}{\Gamma \vdash \mathsf{R}^{\mathbf{N}}_{[n:\mathbf{N}]\sigma}(H_z, [n{:}\,\mathbf{N}, x{:}\,\sigma]H_s, M) : \sigma[M/n]} \text{ N-E}$$

which intuitively states how to prove something inductively, in this case that you need only show something for 0 and $\mathrm{Suc}(M)$ (gluing), and finally a computation rule which asserts that the construction agrees with the pieces that were used to construct it (restriction).

# Inductive Types

In general then an inductive type is specified by some constructors, some induction principle and some computation rules. The induction principle is always of the form: To define a dependent function $f : \prod_{x:A} B(x)$ it is sufficient to specify $f$ on the constructors of $A$. This is essentially a universal property.
Care must be taken with this vague definition and the exact concept of which types should be admitted as paradoxes easily creep in. I wont try to state anything precise because I dont know anything.

# Higher Inductive Types

The key point of higher inductive types is that they are also defined by constructors, induction rules and computation rules however the constructors are now permitted to generate points of the *identity types* of the type that we are defining and so on.
In defining inductive type $A$ we could refer to $A$ and other types *that didnt depend on* $A$, for higher inductive types we may also refer to $\mathrm{id}(A, A)$ and $\mathrm{id}(\mathrm{id}(\mathrm{A}, \mathrm{A}), \mathrm{id}(\mathrm{A}, \mathrm{A}))$ etc.

# Circle Type

HOTT is just an idea, but a basic higher inductive type that is often included is the circle. This has the two constructors

$$\frac{\vdash \Gamma ctxt}{\Gamma \vdash \mathrm{base} : S^1} \quad \text{and} \quad \frac{\vdash \Gamma ctxt}{\Gamma \vdash \mathrm{loop} : \mathrm{id}(S^1, S^1)}$$

And the rules:

$$\frac{\Gamma, x{:}\mathbb{S}^1 \vdash C : \mathcal{U}_i \qquad \Gamma \vdash b : C[\mathsf{base}/x] \qquad \Gamma \vdash \ell : b =^C_{\mathsf{loop}} b \qquad \Gamma \vdash p : \mathbb{S}^1}{\Gamma \vdash \mathsf{ind}_{\mathbb{S}^1}(x.C, b, \ell, p) : C[p/x]}\ \mathbb{S}^1\text{-ELIM}$$

$$\frac{\Gamma, x{:}\mathbb{S}^1 \vdash C : \mathcal{U}_i \qquad \Gamma \vdash b : C[\mathsf{base}/x] \qquad \Gamma \vdash \ell : b =^C_{\mathsf{loop}} b}{\Gamma \vdash \mathsf{ind}_{\mathbb{S}^1}(x.C, b, \ell, \mathsf{base}) \equiv b : C[\mathsf{base}/x]}\ \mathbb{S}^1\text{-COMP}_1$$

$$\frac{\Gamma, x{:}\mathbb{S}^1 \vdash C : \mathcal{U}_i \qquad \Gamma \vdash b : C[\mathsf{base}/x] \qquad \Gamma \vdash \ell : b =^C_{\mathsf{loop}} b}{\Gamma \vdash \mathbb{S}^1\text{-}\mathsf{loopcomp} : \mathsf{apd}_{(\lambda y.\, \mathsf{ind}_{\mathbb{S}^1}(x.C,b,\ell,y))}(\mathsf{loop}) = \ell}\ \mathbb{S}^1\text{-COMP}_2$$

In human language this means: We introduce a circle type with a canonical base point, we also introduce a loop which is a self map of the circle.
The elimination rule says given a map $C : S^1 \to \mathcal{U}$ (expressed by $C$ having dependence on $x$ in the context) such that $C(\mathrm{base}) = b$, then a loop from $b$ to $b$ in $C$ then there is a function $f : \prod_{x:S^1} C(x)$ that maps the base to $b$ and $\mathrm{loop}$ to the loop around $b$.
Note that here we are using functoriality of functions, that functions can be applied to loops and applied $\lambda$ abstraction to the deduction rules above.
Finally the computation rules intuitively say that given a point and a loop in a type there exists a function from $S^1$ assigning the base point to the point and the loop to the loop.

From this intuitive description we can see how these universal properties may be used to prove the circle is non-trivial [Hom, Lem 6.4.1]: $\mathrm{loop} \neq \mathrm{refl}_{\mathrm{base}}$, the loop is not the trivial reflective element.

*Proof.* Suppose that $\mathsf{loop} = \mathsf{refl}_{\mathsf{base}}$. Then since for any type $A$ with $x : A$ and $p : x = x$, there is a function $f : \mathbb{S}^1 \to A$ defined by $f(\mathsf{base}) :\equiv x$ and $f(\mathsf{loop}) := p$, we have

$$p = f(\mathsf{loop}) = f(\mathsf{refl}_{\mathsf{base}}) = \mathsf{refl}_x.$$

But this implies that every type is a set, which as we have seen is not the case (see Example 3.1.9). □

The moral is that the circle is not-trivial because the type theory is rich enough to allow objects with non-trivial loops, hence being the universal thing with a loop it cannot be trivial.

# Cubical Type Theory

One variation on this worth mentioning is Cubical type theory (CTT). Replaces identity types with interval types, adds a gluing type. The other axioms are derivable. [CCHM16].


Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg.
Cubical Type Theory: a constructive interpretation of the univalence axiom, November 2016.
arXiv:1611.02108 [cs.LO].


Martin Hofmann.
Syntax and Semantics of Dependent Types.
In Andrew M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 79–130. Cambridge University Press, Cambridge, 1997.


Homotopy Type Theory: Univalent Foundations of Mathematics.


Per Martin-Lof.
Intuitionistic Type Theory.


Egbert Rijke.
Introduction to Homotopy Type Theory, December 2022.
arXiv:2212.11082 [math.LO].