Homotopy Type Theory
and the Future of Mathematics

Will Barnett

16 September 2022

Introduction

Hi.

Introduction

Hi.

What is this?

Introduction

Hi.

What is this?

A talk on Homotopy Type Theory—a 21st-century foundation of mathematics.

Introduction

Hi.

What is this?

A talk on Homotopy Type Theory—a 21st-century foundation of mathematics.

Who am I?

Introduction

Hi.

What is this?
A talk on Homotopy Type Theory—a 21st-century foundation of mathematics.

Who am I?
A 2nd year Math/CS undergrad at UQ.

Introduction

Hi.

What is this?

A talk on Homotopy Type Theory—a 21st-century foundation of mathematics.

Who am I?

A 2nd year Math/CS undergrad at UQ.

Why should you listen to me?

Introduction

Hi.

What is this?

A talk on Homotopy Type Theory—a 21st-century foundation of mathematics.

Who am I?

A 2nd year Math/CS undergrad at UQ.

Why should you listen to me?

After much effort, I now understand the basic concepts of HoTT.

Introduction

Hi.

What is this?

A talk on Homotopy Type Theory—a 21st-century foundation of mathematics.

Who am I?

A 2nd year Math/CS undergrad at UQ.

Why should you listen to me?

After much effort, I now understand the basic concepts of HoTT. I do not actually understand homotopy theory (a different subject).

Foundations

What is a foundation of mathematics?

Foundations

What is a foundation of mathematics?

A system for expressing "all of mathematics" in terms of (relatively) simple building blocks.

Why?

Foundations

What is a foundation of mathematics?

A system for expressing "all of mathematics" in terms of (relatively) simple building blocks.

Why?

- Increase precision and rigour to the max.

Foundations

What is a foundation of mathematics?

A system for expressing "all of mathematics" in terms of (relatively) simple building blocks.

Why?

- Increase precision and rigour to the max.
- Eliminate (certain classes of) errors.

Foundations

What is a foundation of mathematics?

A system for expressing "all of mathematics" in terms of (relatively) simple building blocks.

Why?

- Increase precision and rigour to the max.
- Eliminate (certain classes of) errors.
- Enables computer proof-checking (formalization).

Foundations

What is a foundation of mathematics?

A system for expressing “all of mathematics” in terms of (relatively) simple building blocks.

Why?

- Increase precision and rigour to the max.
- Eliminate (certain classes of) errors.
- Enables computer proof-checking (formalization).
- Slows you down.

Foundations

What is a foundation of mathematics?

A system for expressing "all of mathematics" in terms of (relatively) simple building blocks.

Why?

- Increase precision and rigour to the max.
- Eliminate (certain classes of) errors.
- Enables computer proof-checking (formalization).
- Slows you down.

Examples

Set theories (ZFC, NBG, ETCS);

Foundations

What is a foundation of mathematics?

A system for expressing “all of mathematics” in terms of (relatively) simple building blocks.

Why?

- Increase precision and rigour to the max.
- Eliminate (certain classes of) errors.
- Enables computer proof-checking (formalization).
- Slows you down.

Examples

Set theories (ZFC, NBG, ETCS); type theories (PM, MLTT, HoTT).

Set Theory vs Type Theory

In set theory, all mathematical objects are built out of sets.

Set Theory vs Type Theory

In set theory, all mathematical objects are built out of sets.
In type theory, all mathematical objects are built out of types.

Set Theory vs Type Theory

In set theory, all mathematical objects are built out of sets.
In type theory, all mathematical objects are built out of types.
What are types?

Set Theory vs Type Theory

In set theory, all mathematical objects are built out of sets.
In type theory, all mathematical objects are built out of types.
What are types?
They're like sets.

Set Theory vs Type Theory

In set theory, all mathematical objects are built out of sets.
In type theory, all mathematical objects are built out of types.
What are types?
They're like sets. They can also be like propositions.

Set Theory vs Type Theory

In set theory, all mathematical objects are built out of sets.
In type theory, all mathematical objects are built out of types.
What are types?
They're like sets. They can also be like propositions. Or like structured sets.

Set Theory vs Type Theory

In set theory, all mathematical objects are built out of sets.
In type theory, all mathematical objects are built out of types.
What are types?
They're like sets. They can also be like propositions. Or like structured sets. Or weirder, wilder things...

Set Theory vs Type Theory

In set theory, all mathematical objects are built out of sets.
In type theory, all mathematical objects are built out of types.
What are types?
They're like sets. They can also be like propositions. Or like structured sets. Or weirder, wilder things...

"Let n be a natural number." $\quad\Rightarrow\quad$ $n : \mathbb{N}$

Set Theory vs Type Theory

In set theory, all mathematical objects are built out of sets.
In type theory, all mathematical objects are built out of types.
What are types?
They're like sets. They can also be like propositions. Or like structured sets. Or weirder, wilder things...

"Let n be a natural number."	$\Rightarrow$	$n : \mathbb{N}$
"Let $(a_n)_{n=0}^{\infty}$ be a sequence of reals."	$\Rightarrow$	$a : \mathbb{N} \to \mathbb{R}$

Set Theory vs Type Theory

In set theory, all mathematical objects are built out of sets.
In type theory, all mathematical objects are built out of types.
What are types?
They're like sets. They can also be like propositions. Or like structured sets. Or weirder, wilder things...

"Let n be a natural number."	$\Rightarrow$	$n : \mathbb{N}$
"Let $(a_n)_{n=0}^{\infty}$ be a sequence of reals."	$\Rightarrow$	$a : \mathbb{N} \to \mathbb{R}$
"Let G be a group."	$\Rightarrow$	$G : \mathsf{Group}$

Set Theory vs Type Theory

In set theory, all mathematical objects are built out of sets.
In type theory, all mathematical objects are built out of types.
What are types?
They're like sets. They can also be like propositions. Or like structured sets. Or weirder, wilder things...

"Let n be a natural number."	$\Rightarrow$	$n : \mathbb{N}$
"Let $(a_n)_{n=0}^{\infty}$ be a sequence of reals."	$\Rightarrow$	$a : \mathbb{N} \to \mathbb{R}$
"Let G be a group."	$\Rightarrow$	$G : \mathsf{Group}$
"Suppose that n is even."	$\Rightarrow$	$h : \mathsf{is\text{-}even}\ n$

Set Theory vs Type Theory

In set theory, all mathematical objects are built out of sets.
In type theory, all mathematical objects are built out of types.
What are types?
They're like sets. They can also be like propositions. Or like structured sets. Or weirder, wilder things...

"Let n be a natural number."	$\Rightarrow$	$n : \mathbb{N}$
"Let $(a_n)_{n=0}^{\infty}$ be a sequence of reals."	$\Rightarrow$	$a : \mathbb{N} \to \mathbb{R}$
"Let G be a group."	$\Rightarrow$	$G : \mathsf{Group}$
"Suppose that n is even."	$\Rightarrow$	$h : \mathsf{is\text{-}even}\ n$

But what if G is actually a triangle?

Set Theory vs Type Theory

In set theory, all mathematical objects are built out of sets.
In type theory, all mathematical objects are built out of types.
What are types?
They're like sets. They can also be like propositions. Or like structured sets. Or weirder, wilder things...

"Let n be a natural number."	$\Rightarrow$	$n : \mathbb{N}$
"Let $(a_n)_{n=0}^{\infty}$ be a sequence of reals."	$\Rightarrow$	$a : \mathbb{N} \to \mathbb{R}$
"Let G be a group."	$\Rightarrow$	$G : \mathsf{Group}$
"Suppose that n is even."	$\Rightarrow$	$h : \mathsf{is\text{-}even}\ n$

But what if G is actually a triangle? **Bad question!**

Set Theory vs Type Theory

In set theory, all mathematical objects are built out of sets.
In type theory, all mathematical objects are built out of types.
What are types?
They're like sets. They can also be like propositions. Or like structured sets. Or weirder, wilder things...

"Let n be a natural number."	$\Rightarrow$	$n : \mathbb{N}$
"Let $(a_n)_{n=0}^{\infty}$ be a sequence of reals."	$\Rightarrow$	$a : \mathbb{N} \to \mathbb{R}$
"Let G be a group."	$\Rightarrow$	$G : \mathsf{Group}$
"Suppose that n is even."	$\Rightarrow$	$h : \mathsf{is\text{-}even}\ n$

But what if G is actually a triangle? **Bad question!**
The typing "relation" isn't a proposition. A mathematical object is *always* associated with its type, by its very nature.

Dependent Function Types

Given types A, B, we can form the function type $A \longrightarrow B$.

Dependent Function Types

Given types A, B, we can form the function type $A \to B$.
The terms of a function type look like this:

$$\lambda n.n + 69 : \mathbb{N} \to \mathbb{N} \qquad ((n \mapsto n + 420) : \mathbb{N} \to \mathbb{N}).$$

Dependent Function Types

Given types A, B, we can form the function type $A \to B$.
The terms of a function type look like this:

$$\lambda n.n + 69 : \mathbb{N} \to \mathbb{N} \qquad ((n \mapsto n + 420) : \mathbb{N} \to \mathbb{N}).$$

This can be generalized to allow the type of the codomain to *depend* on an element of the domain. For example, the type of "identity matrix" (with entries from $\mathbb{R}$) is

$$I : (n : \mathbb{N}) \to M_{n\times n}(\mathbb{R}) \qquad \left(I : \prod_{n:\mathbb{N}} M_{n\times n}(\mathbb{R})\right).$$

Dependent Function Types

Given types A, B, we can form the function type $A \longrightarrow B$.
The terms of a function type look like this:

$$\lambda n.n + 69 : \mathbb{N} \longrightarrow \mathbb{N} \qquad ((n \mapsto n + 420) : \mathbb{N} \longrightarrow \mathbb{N}).$$

This can be generalized to allow the type of the codomain to *depend* on an element of the domain. For example, the type of "identity matrix" (with entries from $\mathbb{R}$) is

$$I : (n : \mathbb{N}) \longrightarrow M_{n\times n}(\mathbb{R}) \qquad \left(I : \prod_{n:\mathbb{N}} M_{n\times n}(\mathbb{R})\right).$$

This yields such believable results as $I_3 : M_{3\times 3}(\mathbb{R})$.

Dependent Pair Types

Given types A, B, we can form the pair type $A \times B$ (the type-theoretic version of the Cartesian product).

Dependent Pair Types

Given types A, B, we can form the pair type $A \times B$ (the type-theoretic version of the Cartesian product).
The terms of a pair type look like this:

$$(\pi, +) : \mathbb{R} \times (\mathbb{N} \times \mathbb{N} \to \mathbb{N}).$$

Dependent Pair Types

Given types A, B, we can form the pair type $A \times B$ (the type-theoretic version of the Cartesian product).
The terms of a pair type look like this:

$$(\pi, +) : \mathbb{R} \times (\mathbb{N} \times \mathbb{N} \to \mathbb{N}).$$

This can be generalized to a dependent pair type. For example, to represent arbitrary tuples of rational numbers:

$$\left(3, \left(\frac{1}{69}, -\frac{1}{420}, 666\right)\right) : (n : \mathbb{N}) \times \mathbb{Q}^n \qquad \left((0, ()) : \sum_{n:\mathbb{N}} \mathbb{Q}^n\right).$$

Universe Types

Everything has has a type, including types themselves.

Universe Types

Everything has has a type, including types themselves.
In most formulations of type theory, a hierarcy of *universes* is used to represent this:

$$\mathbb{N} : \mathsf{Type}, \qquad \mathsf{Type} : \mathsf{Type}_1, \qquad \mathsf{Type}_1 : \mathsf{Type}_2, \ldots .$$

Universe Types

Everything has has a type, including types themselves.
In most formulations of type theory, a hierarcy of *universes* is used to represent this:

$$\mathbb{N} : \mathsf{Type}, \qquad \mathsf{Type} : \mathsf{Type}_1, \qquad \mathsf{Type}_1 : \mathsf{Type}_2, \ldots .$$

Often, U_i is used instead of Type_i (note that i is a *meta-theoretic* natural number).

Universe Types

Everything has has a type, including types themselves.
In most formulations of type theory, a hierarcy of *universes* is used to represent this:

$$\mathbb{N} : \mathsf{Type}, \qquad \mathsf{Type} : \mathsf{Type}_1, \qquad \mathsf{Type}_1 : \mathsf{Type}_2, \ldots .$$

Often, U_i is used instead of Type_i (note that i is a *meta-theoretic* natural number).
It is a bad idea to set up a type theory where $\mathsf{Type} : \mathsf{Type}$, due to the type-theoretic version of Russel's paradox (Girard's paradox).

Propositions as Types

The primary mathematical activity in type theory is constructing a term of a type.

Propositions as Types

The primary mathematical activity in type theory is constructing a term of a type.
Proving a theorem is a special case of this, if we interpret propositions as types.

Propositions as Types

The primary mathematical activity in type theory is constructing a term of a type.
Proving a theorem is a special case of this, if we interpret propositions as types.
What are the terms of these types?

Propositions as Types

The primary mathematical activity in type theory is constructing a term of a type.
Proving a theorem is a special case of this, if we interpret propositions as types.
What are the terms of these types? Proofs of the proposition!

Propositions as Types

The primary mathematical activity in type theory is constructing a term of a type.
Proving a theorem is a special case of this, if we interpret propositions as types.
What are the terms of these types? Proofs of the proposition!

- Implication is the function type: $P \to Q$.

Propositions as Types

The primary mathematical activity in type theory is constructing a term of a type.
Proving a theorem is a special case of this, if we interpret propositions as types.
What are the terms of these types? Proofs of the proposition!

- Implication is the function type: $P \to Q$.
- Conjunction is the pair type: $P \times Q$.

Propositions as Types

The primary mathematical activity in type theory is constructing a term of a type.
Proving a theorem is a special case of this, if we interpret propositions as types.
What are the terms of these types? Proofs of the proposition!

- Implication is the function type: $P \to Q$.
- Conjunction is the pair type: $P \times Q$.
- Universal quantification is the dependent function type: $(x : A) \to P(x)$.

Propositions as Types

The primary mathematical activity in type theory is constructing a term of a type.
Proving a theorem is a special case of this, if we interpret propositions as types.
What are the terms of these types? Proofs of the proposition!

- Implication is the function type: $P \to Q$.
- Conjunction is the pair type: $P \times Q$.
- Universal quantification is the dependent function type: $(x : A) \to P(x)$.
- Disjunction and existential quantification are best discussed after *propositional truncation* has been introduced.

Equality

The behaviour of the identity type = (sometimes called Id) is what sets HoTT apart from bare Martin Löf Type Theory.

Equality

The behaviour of the identity type = (sometimes called Id) is what sets HoTT apart from bare Martin Löf Type Theory. More about this type later, but the basic information is: for a given type A,

$$=_A : A \longrightarrow A \longrightarrow \mathsf{Type}, \qquad \mathsf{refl}_A : (a : A) \longrightarrow a =_A a.$$

Equality

The behaviour of the identity type = (sometimes called Id) is what sets HoTT apart from bare Martin Löf Type Theory. More about this type later, but the basic information is: for a given type A,

$$=_A : A \longrightarrow A \longrightarrow \mathsf{Type}, \qquad \mathsf{refl}_A : (a : A) \longrightarrow a =_A a.$$

The type parameter A is often omitted, since it can be inferred from context.

Equality

The behaviour of the identity type = (sometimes called Id) is what sets HoTT apart from bare Martin Löf Type Theory. More about this type later, but the basic information is: for a given type A,

$$=_A : A \longrightarrow A \longrightarrow \mathsf{Type}, \qquad \mathsf{refl}_A : (a : A) \longrightarrow a =_A a.$$

The type parameter A is often omitted, since it can be inferred from context.
Note that *currying* is used; this is conventional in HoTT.

Equality

The behaviour of the identity type = (sometimes called Id) is what sets HoTT apart from bare Martin Löf Type Theory.
More about this type later, but the basic information is: for a given type A,

$$=_A: A \longrightarrow A \longrightarrow \mathsf{Type}, \qquad \mathsf{refl}_A : (a : A) \longrightarrow a =_A a.$$

The type parameter A is often omitted, since it can be inferred from context.
Note that *currying* is used; this is conventional in HoTT.
The identity type is also called the path type; the homotopical intuition is that the terms of $a =_A b$ are like paths from a to b in the space A.

Using Equality

Equality can be proved to be symmetric and transitive: for any type A,

$$\begin{aligned} \mathsf{sym} &: (a\ b : A) \to a = b \to b = a\,, \\ \mathsf{trans} &: (a\ b\ c : A) \to a = b \to b = c \to a = c\,. \end{aligned}$$

Using Equality

Equality can be proved to be symmetric and transitive: for any type A,

$$\begin{aligned} \mathsf{sym} &: (a\ b : A) \to a = b \to b = a\,, \\ \mathsf{trans} &: (a\ b\ c : A) \to a = b \to b = c \to a = c\,. \end{aligned}$$

These proofs of symmetry and transitivity are not just mere properties, but themselves have structure (in this case, that of an ∞-groupoid, a weird higher-categorical version of a group).

Using Equality

Equality can be proved to be symmetric and transitive: for any type A,

$$\begin{aligned} \mathsf{sym} &: (a\ b : A) \to a = b \to b = a\,, \\ \mathsf{trans} &: (a\ b\ c : A) \to a = b \to b = c \to a = c\,. \end{aligned}$$

These proofs of symmetry and transitivity are not just mere properties, but themselves have structure (in this case, that of an ∞-groupoid, a weird higher-categorical version of a group).
Some other useful operations: for any types A, B,

$$\begin{aligned} \mathsf{ap} &: (f : A \to B) \to (x\ y : A) \to x = y \to f(x) = f(y), \\ \mathsf{transport} &: A = B \to A \to B. \end{aligned}$$

Inductive Types

Where do types like $\mathbb{N}$ come from? They are "freely generated" from "constructors". The "inductive definition" of $\mathbb{N}$ is

$$\mathbb{N} : \mathsf{Type}, \qquad 0 : \mathbb{N}, \qquad S : \mathbb{N} \to \mathbb{N}.$$

Inductive Types

Where do types like $\mathbb{N}$ come from? They are "freely generated" from "constructors". The "inductive definition" of $\mathbb{N}$ is

$$\mathbb{N} : \mathsf{Type}, \qquad 0 : \mathbb{N}, \qquad S : \mathbb{N} \to \mathbb{N}.$$

The idea is that all natural numbers are "built up" from these constructors.

Inductive Types

Where do types like $\mathbb{N}$ come from? They are "freely generated" from "constructors". The "inductive definition" of $\mathbb{N}$ is

$$\mathbb{N} : \mathsf{Type}, \qquad 0 : \mathbb{N}, \qquad S : \mathbb{N} \to \mathbb{N}.$$

The idea is that all natural numbers are "built up" from these constructors.
Every natural number has a "canonical form", like $2 \equiv S(S(0))$.

Inductive Types

Where do types like $\mathbb{N}$ come from? They are "freely generated" from "constructors". The "inductive definition" of $\mathbb{N}$ is

$$\mathbb{N} : \mathsf{Type}, \qquad 0 : \mathbb{N}, \qquad S : \mathbb{N} \to \mathbb{N}.$$

The idea is that all natural numbers are "built up" from these constructors.
Every natural number has a "canonical form", like $2 \equiv S(S(0))$.

Despite the high quotation mark-density, this can be made precise; the validity of similar inductive definitions can be checked by a mechanical process.

How to use Inductive Types

Functions out of inductive types are defined by their behaviour on constructors.

How to use Inductive Types

Functions out of inductive types are defined by their behaviour on constructors.
It is permissible to use previously-defined values of the function on "structurally smaller" values.

How to use Inductive Types

Functions out of inductive types are defined by their behaviour on constructors.
It is permissible to use previously-defined values of the function on “structurally smaller” values.
Defining a function on the natural numbers by recursion, and proving a property of natural numbers by induction, are special cases of this “elimination principle”.

How to use Inductive Types

Functions out of inductive types are defined by their behaviour on constructors.
It is permissible to use previously-defined values of the function on "structurally smaller" values.
Defining a function on the natural numbers by recursion, and proving a property of natural numbers by induction, are special cases of this "elimination principle".
Explicitly, this elimination principle can be expressed in type theory itself as:

$$\begin{aligned}\mathbb{N}\text{-}\mathsf{elim} : (P : \mathbb{N} \to \mathsf{Type}) \to P(0) \to ((n : \mathbb{N}) \to P(n) \to P(S(n))) \\ \to (n : \mathbb{N}) \to P(n).\end{aligned}$$

Elimination Examples

Here is a definition by *pattern matching* (equivalent to eliminator use) of + on $\mathbb{N}$:

$$\begin{aligned} + : \mathbb{N} \to \mathbb{N} \to \mathbb{N}; \\ a + 0 := a; \\ a + S(b) := S(a + b). \end{aligned}$$

Elimination Examples

Here is a definition by *pattern matching* (equivalent to eliminator use) of $+$ on $\mathbb{N}$:

$$\begin{aligned} + : \mathbb{N} \to \mathbb{N} \to \mathbb{N}; \\ a + 0 :\equiv a; \\ a + S(b) :\equiv S(a + b). \end{aligned}$$

Here is an inductive proof that $+$ is associative:

$$\begin{aligned} \mathsf{+\text{-}is\text{-}assoc} : (a\ b\ c : \mathbb{N}) \to a + (b + c) = (a + b) + c; \\ \mathsf{+\text{-}is\text{-}assoc}\ a\ b\ 0 :\equiv \mathsf{refl}(a + b); \\ \mathsf{+\text{-}is\text{-}assoc}\ a\ b\ S(c) :\equiv \mathsf{ap}_S(\mathsf{+\text{-}is\text{-}assoc}\ a\ b\ c). \end{aligned}$$

Other Inductive Types

The empty and unit types (logical false and true):

$$\mathbb{0} : \mathsf{Type}, \qquad \mathbb{1} : \mathsf{Type}, \qquad * : \mathbb{1}.$$

Other Inductive Types

The empty and unit types (logical false and true):

$$\mathbb{0} : \mathsf{Type}, \qquad \mathbb{1} : \mathsf{Type}, \qquad * : \mathbb{1}.$$

The disjoint union of two types A, B:

$$A + B : \mathsf{Type}, \qquad \mathsf{left} : A \longrightarrow A + B, \qquad \mathsf{right} : B \longrightarrow A + B.$$

Other Inductive Types

The empty and unit types (logical false and true):

$$\mathbb{0} : \mathsf{Type}, \qquad \mathbb{1} : \mathsf{Type}, \qquad * : \mathbb{1}.$$

The disjoint union of two types A, B:

$$A + B : \mathsf{Type}, \qquad \mathsf{left} : A \longrightarrow A + B, \qquad \mathsf{right} : B \longrightarrow A + B.$$

The (dependent) sum type and the identity type can also be constructed as inductive types.

Other Inductive Types

The empty and unit types (logical false and true):

$$\mathbb{0} : \mathsf{Type}, \qquad \mathbb{1} : \mathsf{Type}, \qquad * : \mathbb{1}.$$

The disjoint union of two types A, B:

$$A + B : \mathsf{Type}, \qquad \mathsf{left} : A \longrightarrow A + B, \qquad \mathsf{right} : B \longrightarrow A + B.$$

The (dependent) sum type and the identity type can also be constructed as inductive types.
In HoTT, *higher* inductive types can be defined, which can include path constructors as well as the point constructors that we have seen so far.

Proof Assistants

What is a proof assistant?

Proof Assistants

What is a proof assistant?

It is a computer program which checks the correctness of a proof/mathematical construction, which is typically entered by a human user.

Proof Assistants

What is a proof assistant?

It is a computer program which checks the correctness of a proof/mathematical construction, which is typically entered by a human user.

Examles?

Proof Assistants

What is a proof assistant?

It is a computer program which checks the correctness of a proof/mathematical construction, which is typically entered by a human user.

Examles?

Agda, Coq, Isabelle, Lean, Mizar, …. (I recommend Agda for HoTT).

Why bother?

Proof Assistants

What is a proof assistant?

It is a computer program which checks the correctness of a proof/mathematical construction, which is typically entered by a human user.

Examles?

Agda, Coq, Isabelle, Lean, Mizar, …. (I recommend Agda for HoTT).

Why bother?

- Playing with a proof assistant is the best way to understand the previous content on an intuitive level.

Proof Assistants

What is a proof assistant?

It is a computer program which checks the correctness of a proof/mathematical construction, which is typically entered by a human user.

Examales?

Agda, Coq, Isabelle, Lean, Mizar, ... (I recommend Agda for HoTT).

Why bother?

- Playing with a proof assistant is the best way to understand the previous content on an intuitive level.
- Ensures that you don't make mistakes when things get complicated.

Proof Assistants

What is a proof assistant?

It is a computer program which checks the correctness of a proof/mathematical construction, which is typically entered by a human user.

Examales?

Agda, Coq, Isabelle, Lean, Mizar, ... (I recommend Agda for HoTT).

Why bother?

- Playing with a proof assistant is the best way to understand the previous content on an intuitive level.
- Ensures that you don't make mistakes when things get complicated.
- Turns mathematics into a fun computer game.

H-Levels

So far, only vanilla MLTT has been covered.

H-Levels

So far, only vanilla MLTT has been covered.
The h-level of a type is the number of times that $=$ must be iterated before it trivializes.

H-Levels

So far, only vanilla MLTT has been covered.
The h-level of a type is the number of times that $=$ must be iterated before it trivializes.
For historical reasons, people normally start counting at -2.

H-Levels

So far, only vanilla MLTT has been covered.

The h-level of a type is the number of times that $=$ must be iterated before it trivializes.

For historical reasons, people normally start counting at -2, but that is obviously ridiculous, so I will count from 0.

H-Levels

So far, only vanilla MLTT has been covered.
The h-level of a type is the number of times that $=$ must be iterated before it trivializes.
For historical reasons, people normally start counting at -2, but that is obviously ridiculous, so I will count from 0.
Being a 0-type (a type with h-level 0) means being contractible, which is expressed as:

$$\begin{aligned} \textsf{is-contr} &: \textsf{Type} \to \textsf{Type}, \\ \textsf{is-contr}(A) &:= (x : A) \times (y : A) \to x = y. \end{aligned}$$

H-Levels

So far, only vanilla MLTT has been covered.
The h-level of a type is the number of times that = must be iterated before it trivializes.
For historical reasons, people normally start counting at -2, but that is obviously ridiculous, so I will count from 0.
Being a 0-type (a type with h-level 0) means being contractible, which is expressed as:

$$\mathsf{is\text{-}contr} : \mathsf{Type} \to \mathsf{Type},$$
$$\mathsf{is\text{-}contr}(A) :\equiv (x : A) \times (y : A) \to x = y.$$

Rest assured that the identity type of a contractible type is itself contractible.

More H-Levels

Being a *proposition* means being a 1-type (having "at most one" term/proof):

$$\mathsf{is\text{-}prop} : \mathsf{Type} \longrightarrow \mathsf{Type},$$
$$\mathsf{is\text{-}prop}(A) :\equiv (x\ y : A) \to x = y.$$

More H-Levels

Being a *proposition* means being a 1-type (having "at most one" term/proof):

$$\mathsf{is\text{-}prop} : \mathsf{Type} \longrightarrow \mathsf{Type},$$
$$\mathsf{is\text{-}prop}(A) :\equiv (x\ y : A) \to x = y.$$

The types $\mathbb{0}$ and $\mathbb{1}$ are both propositions.

More H-Levels

Being a *proposition* means being a 1-type (having "at most one" term/proof):

$$\mathsf{is\text{-}prop} : \mathsf{Type} \longrightarrow \mathsf{Type},$$
$$\mathsf{is\text{-}prop}(A) :\equiv (x\ y : A) \longrightarrow x = y.$$

The types $\mathbb{0}$ and $\mathbb{1}$ are both propositions.
Being a *set* means being a 2-type (its identity type is a proposition)

$$\mathsf{is\text{-}set} : \mathsf{Type} \longrightarrow \mathsf{Type},$$
$$\mathsf{is\text{-}set}(A) :\equiv (x\ y : A) \longrightarrow \mathsf{is\text{-}prop}(x = y).$$

More H-Levels

Being a *proposition* means being a 1-type (having "at most one" term/proof):

$$\begin{gathered}\mathsf{is\text{-}prop} : \mathsf{Type} \longrightarrow \mathsf{Type}, \\ \mathsf{is\text{-}prop}(A) :\equiv (x\ y : A) \longrightarrow x = y.\end{gathered}$$

The types $\mathbb{0}$ and $\mathbb{1}$ are both propositions.
Being a *set* means being a 2-type (its identity type is a proposition)

$$\begin{gathered}\mathsf{is\text{-}set} : \mathsf{Type} \longrightarrow \mathsf{Type}, \\ \mathsf{is\text{-}set}(A) :\equiv (x\ y : A) \longrightarrow \mathsf{is\text{-}prop}(x = y).\end{gathered}$$

It can be proven (with some effort) that $\mathbb{N}$ is a set (this follows by Hedberg's Theorem).

Equivalence

Given types A, B, being an equivalence is a property of functions $A \to B$:

$$\begin{aligned} \mathsf{is\text{-}equiv} &: (A \to B) \to \mathsf{Type}, \\ \mathsf{is\text{-}equiv}(f) &:= (y : B) \to \mathsf{is\text{-}contr}((x : A) \times (f(x) = y)). \end{aligned}$$

Equivalence

Given types A, B, being an equivalence is a property of functions $A \to B$:

$$\begin{aligned} \mathsf{is\text{-}equiv} &: (A \to B) \to \mathsf{Type}, \\ \mathsf{is\text{-}equiv}(f) &:\equiv (y : B) \to \mathsf{is\text{-}contr}((x : A) \times (f(x) = y)). \end{aligned}$$

This is actually different from the naïve translation of "is bijective" (since there might be different *proofs* that $f(x) = y$).

Equivalence

Given types A, B, being an equivalence is a property of functions $A \longrightarrow B$:

$$\begin{aligned} \mathsf{is\text{-}equiv} &: (A \longrightarrow B) \longrightarrow \mathsf{Type}, \\ \mathsf{is\text{-}equiv}(f) &:= (y : B) \longrightarrow \mathsf{is\text{-}contr}((x : A) \times (f(x) = y)). \end{aligned}$$

This is actually different from the naïve translation of "is bijective" (since there might be different *proofs* that $f(x) = y$). We can define the type of all equivalences between two types:

$$A \simeq B := (f : A \longrightarrow B) \times \mathsf{is\text{-}equiv}(f),$$

which is a "nice" definition since $\mathsf{is\text{-}equiv}(f)$ is always a proposition.

Univalence

The Univalence Principle is a novel contribution of HoTT, which cannot even be stated in other foundations.

Univalence

The Univalence Principle is a novel contribution of HoTT, which cannot even be stated in other foundations.
Succinctly, it states that equivalence is equivalent to identity.

Univalence

The Univalence Principle is a novel contribution of HoTT, which cannot even be stated in other foundations.
Succinctly, it states that equivalence is equivalent to identity.
More precisely,

$$\mathsf{univalence} : (A\ B : \mathsf{Type}) \longrightarrow (A \simeq B) \simeq (A = B).$$

The univalence principle has the following wonderful consequences:

- Logically equivalent propositions are equal.

Univalence

The Univalence Principle is a novel contribution of HoTT, which cannot even be stated in other foundations.
Succinctly, it states that equivalence is equivalent to identity.
More precisely,

$$\mathsf{univalence} : (A\ B : \mathsf{Type}) \longrightarrow (A \simeq B) \simeq (A = B).$$

The univalence principle has the following wonderful consequences:

- Logically equivalent propositions are equal.
- Sets with the same cardinality are equal (don't confuse *sets* with *subsets*).

Univalence

The Univalence Principle is a novel contribution of HoTT, which cannot even be stated in other foundations.
Succinctly, it states that equivalence is equivalent to identity.
More precisely,

$$\mathsf{univalence} : (A\ B : \mathsf{Type}) \longrightarrow (A \simeq B) \simeq (A = B).$$

The univalence principle has the following wonderful consequences:

- Logically equivalent propositions are equal.
- Sets with the same cardinality are equal (don't confuse *sets* with *subsets*).
- Isomorphic groups are equal (don't confuse *groups* with *subgroups*).

What?

This is absurd!

What?

This is absurd!

You're welcome to feel that way, but the logicians have shown that HoTT is consistent (assuming that ZFC with countably many inaccessibles is consistent).

What?

This is absurd!

You're welcome to feel that way, but the logicians have shown that HoTT is consistent (assuming that ZFC with countably many inaccessibles is consistent).

How is this not contradictory?

What?

This is absurd!

You're welcome to feel that way, but the logicians have shown that HoTT is consistent (assuming that ZFC with countably many inaccessibles is consistent).

How is this not contradictory?

You may have heard that "isomorphic structures have the same structural properties".

What?

This is absurd!

You're welcome to feel that way, but the logicians have shown that HoTT is consistent (assuming that ZFC with countably many inaccessibles is consistent).

How is this not contradictory?

You may have heard that "isomorphic structures have the same structural properties".
In HoTT, *only* structural problems can be defined in the first place, so there is no problem.

Why use univalence?

What?

This is absurd!

You're welcome to feel that way, but the logicians have shown that HoTT is consistent (assuming that ZFC with countably many inaccessibles is consistent).

How is this not contradictory?

You may have heard that "isomorphic structures have the same structural properties".
In HoTT, *only* structural problems can be defined in the first place, so there is no problem.

Why use univalence?

It's already used informally, for convenience. If G and H are isomorphic groups, and G is sqaunchy, then you can bet that H is squanchy. HoTT provides a rigorous justification for this.

Consequence of Univalences

All types are characterized up to equivalence.

Consequence of Univalences

All types are characterized up to equivalence.
If someone inductively defines the matural numbers as follows:

$$\mathbb{M} : \mathsf{Type}, \qquad \rotatebox{90}{1} : \mathbb{M}, \qquad \rotatebox{180}{S} : \mathbb{M} \to \mathbb{M},$$

all properties/constructions of N can be interpreted in M, via univalence and transport.

Consequence of Univalences

All types are characterized up to equivalence.
If someone inductively defines the matural numbers as follows:

$$\mathbb{M} : \mathsf{Type}, \qquad \rotatebox{90}{$\mathsf{1}$} : \mathbb{M}, \qquad \rotatebox{90}{S} : \mathbb{M} \to \mathbb{M},$$

all properties/constructions of N can be interpreted in M, via univalence and transport. Univalence lets us ignore unimportant differences in representation, and see through to the mathematical objects themselves.

Consequence of Univalences

All types are characterized up to equivalence.
If someone inductively defines the matural numbers as follows:

$$\mathbb{M} : \mathsf{Type}, \qquad \rotatebox{90}{1} : \mathbb{M}, \qquad \rotatebox{90}{S} : \mathbb{M} \to \mathbb{M},$$

all properties/constructions of N can be interpreted in M, via univalence and transport. Univalence lets us ignore unimportant differences in representation, and see through to the mathematical objects themselves.

By characterizing the identity type, univalence lets us prove that $\mathsf{Set} := (A : \mathsf{Type}) \times \mathsf{is\text{-}set}(A)$ is a 3-type (i.e., its identity types are sets).

Consequence of Univalences

All types are characterized up to equivalence.
If someone inductively defines the matural numbers as follows:

$$\mathbb{M} : \mathsf{Type}, \qquad \rotatebox{90}{$\mathsf{1}$} : \mathbb{M}, \qquad \rotatebox{90}{S} : \mathbb{M} \to \mathbb{M},$$

all properties/constructions of N can be interpreted in M, via univalence and transport. Univalence lets us ignore unimportant differences in representation, and see through to the mathematical objects themselves.

By characterizing the identity type, univalence lets us prove that $\mathsf{Set} := (A : \mathsf{Type}) \times \mathsf{is\text{-}set}(A)$ is a 3-type (i.e., its identity types are sets). Given sets A, B, the type $A = B$ is equivalent (and hence equal!) to the *set* of bijections from A to B.

Higher Inductive Types

Higher inductive types are the second novel feature of HoTT.

Higher Inductive Types

Higher inductive types are the second novel feature of HoTT. The most important HIT is *propositional truncation* of a type A, which supplies paths between all points in A, making it a proposition:

$$\|A\| : \mathsf{Type}, \qquad \mathsf{squash} : (x\ y : A) \to x = y.$$

This features the *higher constructor* squash.

Higher Inductive Types

Higher inductive types are the second novel feature of HoTT. The most important HIT is *propositional truncation* of a type A, which supplies paths between all points in A, making it a proposition:

$$\|A\| : \mathsf{Type}, \qquad \mathsf{squash} : (x\ y : A) \to x = y.$$

This features the *higher constructor* squash.
Intuitively, to define a function out of $\|A\|$, you may use a term of A, provided that the value you are defining does not depend on which term you select (for example, the codomain type could be a proposition).

More Logic

Propositional truncation lets us define disjunction and existential quantification as legitimate propositions:

- Disjunction: $\|A + B\|$.

More Logic

Propositional truncation lets us define disjunction and existential quantification as legitimate propositions:

- Disjunction: $\|A + B\|$.
- Existential quantification: $\|(x : A) \times P(x)\|$.

More Logic

Propositional truncation lets us define disjunction and existential quantification as legitimate propositions:

- Disjunction: $\|A + B\|$.
- Existential quantification: $\|(x : A) \times P(x)\|$.

So now it's possible to express "we know that one of A or B is true, but we don't know which one".

More Logic

Propositional truncation lets us define disjunction and existential quantification as legitimate propositions:

- Disjunction: $\|A + B\|$.
- Existential quantification: $\|(x : A) \times P(x)\|$.

So now it's possible to express "we know that one of A or B is true, but we don't know which one". Some classical axioms that are indispensible for much of mathematics:

$$\mathsf{LEM} : (A : \mathsf{Type}) \to \mathsf{is\text{-}prop}(A) \to (A + \neg A)$$

More Logic

Propositional truncation lets us define disjunction and existential quantification as legitimate propositions:

- Disjunction: $\|A + B\|$.
- Existential quantification: $\|(x : A) \times P(x)\|$.

So now it's possible to express "we know that one of A or B is true, but we don't know which one". Some classical axioms that are indispensible for much of mathematics:

$$\mathsf{LEM} : (A : \mathsf{Type}) \longrightarrow \mathsf{is\text{-}prop}(A) \longrightarrow (A + \neg A)$$

The axiom of choice: for any *set* A and any family of *sets* $B : A \longrightarrow \mathsf{Type}$, we have

$$((x : A) \longrightarrow \|B(x)\|) \longrightarrow \|(x : A) \longrightarrow B(x)\|.$$

More HITs

Quotient Types

Given a relation $R : A \to A \to \mathsf{Type}$, you can form the quotient of A by R as a HIT:

$$[-] : A \to A/R, \qquad \mathsf{squash} : (x\ y : A) \to R(x, y) \to [x] = [y],$$
$$\mathsf{set\text{-}squash} : \mathsf{is\text{-}set}(A/R).$$

More HITs

Quotient Types

Given a relation $R : A \to A \to \mathsf{Type}$, you can form the quotient of A by R as a HIT:

$$[-] : A \to A/R, \qquad \mathsf{squash} : (x\ y : A) \to R(x,y) \to [x] = [y],$$
$$\mathsf{set\text{-}squash} : \mathsf{is\text{-}set}(A/R).$$

This is useful to define, e.g., $\mathbb{Z}$ and $\mathbb{Q}$.

More HITs

Quotient Types

Given a relation $R : A \to A \to \mathsf{Type}$, you can form the quotient of A by R as a HIT:

$$[-] : A \to A/R, \qquad \mathsf{squash} : (x\ y : A) \to R(x, y) \to [x] = [y],$$
$$\mathsf{set\text{-}squash} : \mathsf{is\text{-}set}(A/R).$$

This is useful to define, e.g., $\mathbb{Z}$ and $\mathbb{Q}$.

The Circle Type

Defined by

$$\mathbb{S}^1 : \mathsf{Type}, \qquad \mathsf{base} : \mathbb{S}^1, \qquad \mathsf{loop} : \mathsf{base} = \mathsf{base},$$

this is a "synthetic" version of the circle.

More HITs

Quotient Types

Given a relation $R : A \to A \to \mathsf{Type}$, you can form the quotient of A by R as a HIT:

$$[-] : A \to A/R, \qquad \mathsf{squash} : (x\ y : A) \to R(x, y) \to [x] = [y],$$
$$\mathsf{set\text{-}squash} : \mathsf{is\text{-}set}(A/R).$$

This is useful to define, e.g., $\mathbb{Z}$ and $\mathbb{Q}$.

The Circle Type

Defined by

$$\mathbb{S}^1 : \mathsf{Type}, \qquad \mathsf{base} : \mathbb{S}^1, \qquad \mathsf{loop} : \mathsf{base} = \mathsf{base},$$

this is a "synthetic" version of the circle. We have $(\mathsf{base} = \mathsf{base}) = \mathbb{Z}$.

Cubical Type Theory

This is an extension of HoTT.

Cubical Type Theory

This is an extension of HoTT.
Talk to me about this later...

The Future of Mathematics

The Future of Mathematics

- This slide is opinionated and conjectural.

The Future of Mathematics

- This slide is opinionated and conjectural.
- Proof assistants will become easier to use, and will probably be based on a HoTT-like system.

The Future of Mathematics

- This slide is opinionated and conjectural.
- Proof assistants will become easier to use, and will probably be based on a HoTT-like system.
- In the future, there's a good chance that HoTT-like foundations will supplant ZFC as the "implicitly-assumed foundation".

The Future of Mathematics

- This slide is opinionated and conjectural.
- Proof assistants will become easier to use, and will probably be based on a HoTT-like system.
- In the future, there's a good chance that HoTT-like foundations will supplant ZFC as the "implicitly-assumed foundation".
- In an unknown timeframe, (a decade?), AI systems will become better than humans at mathematics (and everything else).

The Future of Mathematics

- This slide is opinionated and conjectural.
- Proof assistants will become easier to use, and will probably be based on a HoTT-like system.
- In the future, there's a good chance that HoTT-like foundations will supplant ZFC as the "implicitly-assumed foundation".
- In an unknown timeframe, (a decade?), AI systems will become better than humans at mathematics (and everything else).
- If life is to continue to have meaning, we must willingly ignore the AIs (or hope that they choose to remain silent).

The Future of Mathematics

- This slide is opinionated and conjectural.
- Proof assistants will become easier to use, and will probably be based on a HoTT-like system.
- In the future, there's a good chance that HoTT-like foundations will supplant ZFC as the "implicitly-assumed foundation".
- In an unknown timeframe, (a decade?), AI systems will become better than humans at mathematics (and everything else).
- If life is to continue to have meaning, we must willingly ignore the AIs (or hope that they choose to remain silent).
- Spoilers are bad!

Further Reading

- The HoTT Book:
  https://homotopytypetheory.org/book/
- The 1lab:
  https://1lab.dev/
- The HoTT Game:
  https://thehottgameguide.readthedocs.io/en/latest/index.html