Homotopy Type Theory as an Alternative Foundation to Mathematics

Melanie Brown

Synopsis

- History and purpose of type theory
- Types and universes
- Function extensionality and the univalence axiom
- Propositions and sets

Synopsis

- History and purpose of type theory
- Types and universes
- Function extensionality and the univalence axiom
- Propositions and sets

Synopsis

- History and purpose of type theory
- Types and universes
- Function extensionality and the univalence axiom
- Propositions and sets

Synopsis

- History and purpose of type theory
- Types and universes
- Function extensionality and the univalence axiom
- Propositions and sets

History

- *Principia Mathematica*, 1910
- Simply-typed λ-calculus, 1940
- Intuitionistic type theory, 1972
- Homotopy type theory, 2007

History

Russell's paradox

Does the set of all sets that don't contain themselves contain itself?

$$\mathcal{S} = \{S \text{ set} \mid S \notin S\}$$

$$\mathcal{S} \in \mathcal{S} \text{ or } \mathcal{S} \notin \mathcal{S}?$$

History

- *Principia Mathematica*, 1910
- Simply-typed λ-calculus, 1940
- Intuitionistic type theory, 1972
- Homotopy type theory, 2007

History

- *Principia Mathematica*, 1910
- Simply-typed λ-calculus, 1940
- Intuitionistic type theory, 1972
- Homotopy type theory, 2007

History

- *Principia Mathematica*, 1910
- Simply-typed λ-calculus, 1940
- Intuitionistic type theory, 1972
- Homotopy type theory, 2007

Notation

- $\equiv$ denotes **SYNONYMY**

Notation

- $\equiv$ denotes **SYNONYMY**
- $:\equiv$ denotes **DEFINITION**

Notation

- $\equiv$ denotes **SYNONYMY**
- $:\equiv$ denotes **DEFINITION**
- $=$ has a special meaning

Types and universes

- A **TYPE** is a logical demarcation that restricts formulae

Types and universes

- A **TYPE** is a logical demarcation that restricts formulae
- A **TERM** is a formula that has a specific type: in order to use a formula α, we must have previously declared $\alpha : X$, where X is some type

Types and universes

- A **UNIVERSE** is a type whose terms are also types. There is a hierarchy
$$\mathbf{U}_0 : \mathbf{U}_1 : \mathbf{U}_2 : \cdots$$
where $\mathbf{U}_0$ is called the **BASE UNIVERSE**.

Types and universes

- A **UNIVERSE** is a type whose terms are also types. There is a hierarchy

$$\mathbf{U}_0 : \mathbf{U}_1 : \mathbf{U}_2 : \cdots$$

  where $\mathbf{U}_0$ is called the **BASE UNIVERSE**.
- We can think of types in $\mathbf{U}_i$ as belonging to every universe $\mathbf{U}_j$ where $j \geq i$.

Types and universes

- A **UNIVERSE** is a type whose terms are also types. There is a hierarchy
$$\mathbf{U}_0 : \mathbf{U}_1 : \mathbf{U}_2 : \cdots$$
where $\mathbf{U}_0$ is called the **BASE UNIVERSE**.
- We can think of types in $\mathbf{U}_i$ as belonging to every universe $\mathbf{U}_j$ where $j \geq i$.
- Constructions are valid at any universe level, so we drop the index and write $\mathbf{U}$ for the "type of types"

Types and universes

A type is specified using four pieces of information:

- formation rules: what other types are required to create it;
- construction rules: how to create standard terms;
- elimination rules: how to use generic terms in expressions;
- computation rules: how eliminators act on constructors.

Types and universes

A type is specified using four pieces of information:

- formation rules: what other types are required to create it;
- construction rules: how to create standard terms;
- elimination rules: how to use generic terms in expressions;
- computation rules: how eliminators act on constructors.

Types and universes

A type is specified using four pieces of information:

- formation rules: what other types are required to create it;
- construction rules: how to create standard terms;
- elimination rules: how to use generic terms in expressions;
- computation rules: how eliminators act on constructors.

Types and universes

A type is specified using four pieces of information:

- formation rules: what other types are required to create it;
- construction rules: how to create standard terms;
- elimination rules: how to use generic terms in expressions;
- computation rules: how eliminators act on constructors.

Types and universes: Function types

Construction rules

Let X and Y be types. The type of **FUNCTIONS**, written $X \to Y$, is formed from these two types, and its terms are constructed using **λ-EXPRESSIONS** of the form

$$\lambda(x : X).\,(y : Y) : X \to Y.$$

Let X and Y be types. The type of **FUNCTIONS**, written $X \to Y$, is formed from these two types, and its terms are constructed using **λ-EXPRESSIONS** of the form

$$\lambda(x : X).\,(y : Y) : X \to Y.$$

A **TYPE FAMILY** $Z : X \to \mathbf{U}$ is a type-valued function, where the types $Z(x)$ depend on the particular $x : X$ chosen. The type of **DEPENDENT FUNCTIONS** is then written $\prod_{(x:X)} Z(x)$, and its terms are constructed with λ-expressions of the form

$$\lambda(x : X).\,(z : Z(x)) : \prod_{(x:X)} Z(x).$$

- The elimination rule for $X \to Y$ is, given $w : X$ and $f : X \to Y$, we have a term $f(w) : Y$.

Types and universes: Function types

Elimination & computation rules

- The elimination rule for $X \to Y$ is, given $w : X$ and $f : X \to Y$, we have a term $f(w) : Y$.
- Similarly, if we are given $w : X$ and $g : \prod_{(x:X)} Z(x)$, then $g(w) : Z(w)$.

Types and universes: Function types

Elimination & computation rules

- The elimination rule for $X \to Y$ is, given $w : X$ and $f : X \to Y$, we have a term $f(w) : Y$.
- Similarly, if we are given $w : X$ and $g : \prod_{(x:X)} Z(x)$, then $g(w) : Z(w)$.
- The computation rule is, given $w : X$ and the λ-expression $f :\equiv \lambda(x : X).\,(y : Y)$, we let $f(w) \equiv y[w/x]$, where $y[w/x]$ is the formula y but with each occurrence of the term x replaced by w.

Types and universes

- For other types, the elimination and computation rules can be combined in one function definition

Types and universes

- For other types, the elimination and computation rules can be combined in one function definition
- The elimination rule is given by the type, and the computation rule by the function definition

Types and universes

- For other types, the elimination and computation rules can be combined in one function definition
- The elimination rule is given by the type, and the computation rule by the function definition
- For constant terms in another type, this function is called the **RECURSION PRINCIPLE**

Types and universes

- For other types, the elimination and computation rules can be combined in one function definition
- The elimination rule is given by the type, and the computation rule by the function definition
- For constant terms in another type, this function is called the **RECURSION PRINCIPLE**
- For terms of a type family, it is called the **INDUCTION PRINCIPLE**

Types and universes: Pair types

Construction rules

The **PAIR TYPE** is formed from two types X, Y, and is written $X \times Y$. Its terms are constructed using the function

$$(-,-) : X \to Y \to X \times Y;$$

which means standard terms are of the form $(x, y) : X \times Y$, where $x : X$ and $y : Y$.

The **PAIR TYPE** is formed from two types X, Y, and is written $X \times Y$. Its terms are constructed using the function

$$(-,-) : X \to Y \to X \times Y;$$

which means standard terms are of the form $(x, y) : X \times Y$, where $x : X$ and $y : Y$.

Let $Z : X \to \mathbf{U}$ be a type family. The **DEPENDENT PAIR TYPE** $\sum_{(x:X)} Z(x) : \mathbf{U}$, and its terms are of the form (x, z), where $x : X$ and $z : Z(x)$.

Types and universes: Pair types

Elimination & computation rules

- The **RECURSION PRINCIPLE** tells us how to create terms of a constant type $Z : \mathbf{U}$ from a pair.

$$\mathrm{rec}_\times : \prod_{(Z:\mathbf{U})} (X \to Y \to Z) \to (X \times Y \to Z)$$
$$\mathrm{rec}_\times(Z, f, (x, y)) :\equiv f(x, y).$$

Types and universes: Pair types

Elimination & computation rules

- The **RECURSION PRINCIPLE** tells us how to create terms of a constant type $Z : \mathbf{U}$ from a pair.

$$\mathrm{rec}_\times : \prod_{(Z:\mathbf{U})}(X \to Y \to Z) \to (X \times Y \to Z)$$
$$\mathrm{rec}_\times(Z, f, (x, y)) :\equiv f(x, y).$$

- The **INDUCTION PRINCIPLE** tells us how to create terms of types depending on pairs:

$$\mathrm{ind}_\times : \prod_{(Z:X\times Y\to\mathbf{U})} \left(\prod_{(x:X)}\prod_{(y:Y)} Z((x,y))\right) \to \left(\prod_{(p:X\times Y)} Z(p)\right)$$
$$\mathrm{ind}_\times(Z, f, (x, y)) :\equiv f(x, y).$$

There are some useful functions, called the **PROJECTIONS**, from pairs to their constituents. Let $X, Y : \mathbf{U}$, $Z : X \to \mathbf{U}$, and suppose that $x : X$, $y : Y$, and $z : Z(x)$. We have

$$\mathrm{pr}_1((x, y)) :\equiv x, \quad \mathrm{pr}_2((x, y)) :\equiv y;$$
$$\mathrm{pr}_1((x, z)) :\equiv x, \quad \mathrm{pr}_2((x, z)) :\equiv z.$$

Types and universes: Pair types

Extra bits

There are some useful functions, called the **PROJECTIONS**, from pairs to their constituents. Let $X, Y : \mathbf{U}$, $Z : X \to \mathbf{U}$, and suppose that $x : X$, $y : Y$, and $z : Z(x)$. We have

$$\begin{aligned} \mathrm{pr}_1((x, y)) &:\equiv x, & \mathrm{pr}_2((x, y)) &:\equiv y; \\ \mathrm{pr}_1((x, z)) &:\equiv x, & \mathrm{pr}_2((x, z)) &:\equiv z. \end{aligned}$$

exercise: Write down the type of pr_2 in the dependent case.

- Concrete types need no information to be formed

Types and universes: Concrete types

Unit type

- Concrete types need no information to be formed
- The **UNIT TYPE** $\mathbf{1} : \mathbf{U}$ is concrete and has one constructor,

$$* : \mathbf{1}$$

Types and universes: Concrete types

Unit type

- Concrete types need no information to be formed
- The **UNIT TYPE** $\mathbf{1} : \mathbf{U}$ is concrete and has one constructor,

$$* : \mathbf{1}$$

- Recursion and induction principles:

$$\mathsf{rec}_{\mathbf{1}} : \prod_{(Z:\mathbf{U})} Z \to \mathbf{1} \to Z$$
$$\mathsf{rec}_{\mathbf{1}}(Z, z, *) :\equiv z;$$

$$\mathsf{ind}_{\mathbf{1}} : \prod_{(Z:\mathbf{1}\to\mathbf{U})} Z(*) \to \prod_{(u:\mathbf{1})} Z(u)$$
$$\mathsf{ind}_{\mathbf{1}}(Z, z, *) :\equiv z.$$

- The **EMPTY TYPE** $\mathbf{0} : \mathsf{U}$ is a concrete type, with no constructors

Types and universes: Concrete types

Empty type

- The **EMPTY TYPE** $\mathbf{0} : \mathbf{U}$ is a concrete type, with no constructors
- Type families can't depend on anything, so there is only a recursion principle:

$$\mathsf{rec}_{\mathbf{0}} : \prod_{(Z:\mathbf{U})} \mathbf{0} \to Z$$

- The **EMPTY TYPE** $\mathbf{0} : \mathbf{U}$ is a concrete type, with no constructors
- Type families can't depend on anything, so there is only a recursion principle:

$$\mathrm{rec}_{\mathbf{0}} : \prod_{(Z:\mathbf{U})} \mathbf{0} \to Z$$

- A term $! : \mathbf{0}$ is called a **CONTRADICTION**, since there is no way to create a standard term

- The **EMPTY TYPE** $\mathbf{0} : \mathbf{U}$ is a concrete type, with no constructors
- Type families can't depend on anything, so there is only a recursion principle:

$$\mathsf{rec}_{\mathbf{0}} : \prod_{(Z:\mathbf{U})} \mathbf{0} \to Z$$

- A term $! : \mathbf{0}$ is called a **CONTRADICTION**, since there is no way to create a standard term
- Types with terms are called **INHABITED**; here $\mathbf{0}$ is uninhabited

- The type of **NATURAL NUMBERS** $\mathbf{N} : \mathbf{U}$ is concrete, and has two constructors:

$$0 : \mathbf{N}, \qquad \mathsf{succ} : \mathbf{N} \to \mathbf{N}.$$

Types and universes: Concrete types

Natural numbers

- The type of **NATURAL NUMBERS** $\mathbf{N} : \mathbf{U}$ is concrete, and has two constructors:

$$0 : \mathbf{N}, \qquad \mathsf{succ} : \mathbf{N} \to \mathbf{N}.$$

- Here we see the namesake of the recursion and induction principles:

$$\begin{aligned}
\mathsf{rec}_{\mathbf{N}} &: \textstyle\prod_{(Z:\mathbf{U})} Z \to (\mathbf{N} \to Z \to Z) \to (\mathbf{N} \to Z) \\
\mathsf{rec}_{\mathbf{N}}(Z, z_0, z_s, 0) &:\equiv z_0, \\
\mathsf{rec}_{\mathbf{N}}(Z, z_0, z_s, \mathsf{succ}(n)) &:\equiv z_s(n, \mathsf{rec}_{\mathbf{N}}(Z, z_0, z_s, n));
\end{aligned}$$

$$\begin{aligned}
\mathsf{ind}_{\mathbf{N}} &: \textstyle\prod_{(Z:\mathbf{N}\to\mathbf{U})} Z(0) \to (\prod_{(n:\mathbf{N})} Z(n) \to Z(\mathsf{succ}(n))) \to (\prod_{(n:\mathbf{N})} Z(n)) \\
\mathsf{ind}_{\mathbf{N}}(Z, z_0, z_s, 0) &:\equiv z_0, \\
\mathsf{ind}_{\mathbf{N}}(Z, z_0, z_s, \mathsf{succ}(n)) &:\equiv z_s(n, \mathsf{ind}_{\mathbf{N}}(Z, z_0, z_s, n)).
\end{aligned}$$

Types and universes: Coproduct types

Construction rules

Let X, Y : **U**. The **COPRODUCT TYPE** $(X + Y)$: **U** also has two constructors:

$$\mathrm{in}\ell : X \to X + Y, \qquad \mathrm{in}r : Y \to X + Y.$$

The standard terms of $X + Y$ are of the form $\mathrm{in}\ell(x)$ for some $x : X$ or $\mathrm{in}r(y)$ for some $y : Y$, but none use terms of both X and Y for their construction.

Types and universes: Coproduct types

Elimination & computation rules

- Recursion principle:

$$\mathsf{rec}_+ : \prod_{(Z:\mathsf{U})} (X \to Z) \to (Y \to Z) \to (X + Y \to Z)$$
$$\mathsf{rec}_+(Z, f, g, \mathsf{in}\ell(x)) :\equiv f(x),$$
$$\mathsf{rec}_+(Z, f, g, \mathsf{in}r(y)) :\equiv g(y).$$

- Recursion principle:

$$\mathsf{rec}_+ : \prod_{(Z:\mathsf{U})}(X \to Z) \to (Y \to Z) \to (X + Y \to Z)$$

$$\mathsf{rec}_+(Z, f, g, \mathsf{in}\ell(x)) :\equiv f(x),$$

$$\mathsf{rec}_+(Z, f, g, \mathsf{in}r(y)) :\equiv g(y).$$

- Induction principle: same definition, but with the type

$$\mathsf{ind}_+ : \prod_{(Z:X+Y\to\mathsf{U})}\Big(\prod_{(x:X)} Z(\mathsf{in}\ell(x))\Big) \to \Big(\prod_{(y:Y)} Z(\mathsf{in}r(y))\Big) \to \Big(\prod_{(p:X+Y)} Z(p)\Big).$$

Types and universes: Path types

Construction rules

- The **PATH TYPE** within a type $X : \mathbf{U}$ is formed using $= : X \to X \to \mathbf{U}$, using two terms $x, y : X$ to make $(x = y) : \mathbf{U}$

Types and universes: Path types

Construction rules

- The **PATH TYPE** within a type $X : \mathbf{U}$ is formed using $= : X \to X \to \mathbf{U}$, using two terms $x, y : X$ to make $(x = y) : \mathbf{U}$
- Terms of this type represent paths between x and y

Types and universes: Path types

Construction rules

- The **PATH TYPE** within a type $X : \mathbf{U}$ is formed using $= : X \to X \to \mathbf{U}$, using two terms $x, y : X$ to make $(x = y) : \mathbf{U}$
- Terms of this type represent paths between x and y
- The type $x = x$ has one constructor: $\mathrm{refl}_x : x = x$, called **REFLEXIVITY**

Types and universes: Path types

Construction rules

- The **PATH TYPE** within a type $X : \mathbf{U}$ is formed using $= : X \to X \to \mathbf{U}$, using two terms $x, y : X$ to make $(x = y) : \mathbf{U}$
- Terms of this type represent paths between x and y
- The type $x = x$ has one constructor: $\mathrm{refl}_x : x = x$, called **REFLEXIVITY**
- There are no standard terms of $x = y$ when $x \neq y$

Types and universes: Path types

Construction rules

- The **PATH TYPE** within a type $X : \mathbf{U}$ is formed using $= : X \to X \to \mathbf{U}$, using two terms $x, y : X$ to make $(x = y) : \mathbf{U}$
- Terms of this type represent paths between x and y
- The type $x = x$ has one constructor: $\mathrm{refl}_x : x = x$, called **REFLEXIVITY**
- There are no standard terms of $x = y$ when $x \neq y$
- Intuition for induction: terms of the type family $(x = -) : X \to \mathbf{U}$ created by "dragging" the other endpoint around the type

Types and universes: Path types

Elimination & computation rules

- No recursion principle: paths are inherently dependent

- No recursion principle: paths are inherently dependent
- The type family in the induction depends on *any* path between *any* two terms of X

Types and universes: Path types

Elimination & computation rules

- No recursion principle: paths are inherently dependent
- The type family in the induction depends on *any* path between *any* two terms of X

$$\mathsf{ind}_{=} : \prod_{(Z:\prod_{(x,y:X)}(x=y)\to\mathsf{U})} \left(\prod_{(x:X)} Z(x,x,\mathsf{refl}_x)\right) \to \prod_{(x,y:X)} \prod_{(p:x=y)} Z(x,y,p)$$
$$\mathsf{ind}_{=}(Z,f,x,x,\mathsf{refl}_x) :\equiv f(x).$$

Types and universes: Path types

Elimination & computation rules

- No recursion principle: paths are inherently dependent
- The type family in the induction depends on *any* path between *any* two terms of X

$$\mathsf{ind}_= : \prod_{(Z:\prod_{(x,y:X)}(x=y)\to\mathsf{U})} \left(\prod_{(x:X)} Z(x,x,\mathsf{refl}_x)\right) \to \prod_{(x,y:X)} \prod_{(p:x=y)} Z(x,y,p)$$

$$\mathsf{ind}_=(Z,f,x,x,\mathsf{refl}_x) :\equiv f(x).$$

- We only know how to apply the function to standard terms.

Types and universes: Path types

Important lemmas

Lemma: Path inversion

Let $X : \mathbf{U}$, $x, y : X$, and $p : x = y$. Then there is a term $p^{-1} : y = x$.

Lemma: Path inversion

Let $X : \mathbf{U}$, $x, y : X$, and $p : x = y$. Then there is a term $p^{-1} : y = x$.

Proof: Using the induction principle, we need only consider the case when $x \equiv y$ and $p \equiv \mathsf{refl}_x : x = x$. But now we can define $\mathsf{refl}_x^{-1} :\equiv \mathsf{refl}_x : x = x$. ∎

Lemma: Path inversion

Let $X : \mathbf{U}$, $x, y : X$, and $p : x = y$. Then there is a term $p^{-1} : y = x$.

Proof: Using the induction principle, we need only consider the case when $x \equiv y$ and $p \equiv \mathsf{refl}_x : x = x$. But now we can define $\mathsf{refl}_x^{-1} :\equiv \mathsf{refl}_x : x = x$. ∎

Lemma: Path concatenation

Let $X : \mathbf{U}$, $x, y, z : X$, and $p : x = y$, $q : y = z$. Then there is a term $p \bullet q : x = z$.

Types and universes: Path types

Important lemmas

Lemma: Path inversion

Let $X : \mathbf{U}$, $x, y : X$, and $p : x = y$. Then there is a term $p^{-1} : y = x$.

Proof: Using the induction principle, we need only consider the case when $x \equiv y$ and $p \equiv \mathsf{refl}_x : x = x$. But now we can define $\mathsf{refl}_x^{-1} :\equiv \mathsf{refl}_x : x = x$. ∎

Lemma: Path concatenation

Let $X : \mathbf{U}$, $x, y, z : X$, and $p : x = y$, $q : y = z$. Then there is a term $p \bullet q : x = z$.

Proof: Using the induction principle (twice), it suffices to consider when $x \equiv y \equiv z$ and $p \equiv q \equiv \mathsf{refl}_x : x = x$. But in this case, we can set $\mathsf{refl}_x \bullet \mathsf{refl}_x :\equiv \mathsf{refl}_x$. ∎

- What does it mean for two functions to be equal?

- What does it mean for two functions to be equal?
- Let $X, Y : \mathbf{U}$ and $f, g : X \to Y$. We can define

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

to be the type of **HOMOTOPIES** between f and g.

- What does it mean for two functions to be equal?
- Let $X, Y : \mathbf{U}$ and $f, g : X \to Y$. We can define

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

  to be the type of **HOMOTOPIES** between f and g.
- Homotopy is an equivalence relation. We can also define

$$\mathsf{happly} : (f = g) \to (f \sim g)$$

  by path induction, where $\mathsf{happly}(\mathsf{refl}_f) :\equiv \lambda(x : X).\ \mathsf{refl}_{f(x)}$.

Let $X, Y : \mathbf{U}$ and $f : X \to Y$. We call f an **EQUIVALENCE** if

$$\mathsf{isEquiv}(f) :\equiv \sum_{(g:Y\to X)}(g \circ f \sim \mathsf{id}_X) \times \sum_{(h:Y\to X)}(f \circ h \sim \mathsf{id}_Y)$$

Let $X, Y : \mathbf{U}$ and $f : X \to Y$. We call f an **EQUIVALENCE** if

$$\begin{aligned}\mathsf{isEquiv}(f) &:\equiv \textstyle\sum_{(g:Y\to X)}(g \circ f \sim \mathsf{id}_X) \times \sum_{(h:Y\to X)}(f \circ h \sim \mathsf{id}_Y) \\ &:\equiv \textstyle\sum_{(g:Y\to X)}(\prod_{(x:X)}(g(f(x)) = x)) \times \sum_{(h:Y\to X)}(\prod_{(y:Y)}(f(h(y)) = y))\end{aligned}$$

Let $X, Y : \mathbf{U}$ and $f : X \to Y$. We call f an **EQUIVALENCE** if

$$\begin{aligned}\mathrm{isEquiv}(f) &:\equiv \textstyle\sum_{(g:Y\to X)}(g \circ f \sim \mathrm{id}_X) \times \sum_{(h:Y\to X)}(f \circ h \sim \mathrm{id}_Y) \\ &:\equiv \textstyle\sum_{(g:Y\to X)}(\prod_{(x:X)}(g(f(x)) = x)) \times \sum_{(h:Y\to X)}(\prod_{(y:Y)}(f(h(y)) = y))\end{aligned}$$

We write the type of **EQUIVALENCES**

$$X \simeq Y :\equiv \textstyle\sum_{(f:X\to Y)} \mathrm{isEquiv}(f).$$

example: For each $X : \mathbf{U}$, $\mathrm{id}_X : X \simeq X$.

example: For each $X : \mathbf{U}$, $\mathrm{id}_X : X \simeq X$.

- Type equivalence is an equivalence relation.

Function extensionality and univalence

Type equivalence

example: For each $X : \mathbf{U}$, $\mathrm{id}_X : X \simeq X$.

- Type equivalence is an equivalence relation.
- A **QUASI-INVERSE** to $f : X \to Y$ is a function $g : Y \to X$ with both $g \circ f \sim \mathrm{id}_X$ and $f \circ g \sim \mathrm{id}_Y$.

example: For each $X : \mathbf{U}$, $\mathrm{id}_X : X \simeq X$.

- Type equivalence is an equivalence relation.
- A **QUASI-INVERSE** to $f : X \to Y$ is a function $g : Y \to X$ with both $g \circ f \sim \mathrm{id}_X$ and $f \circ g \sim \mathrm{id}_Y$.
- Having a quasi-inverse means being an equivalence; the converse also holds.

Function extensionality and univalence

Type equivalence

example: For each $X : \mathbf{U}$, $\mathrm{id}_X : X \simeq X$.

- Type equivalence is an equivalence relation.
- A **QUASI-INVERSE** to $f : X \to Y$ is a function $g : Y \to X$ with both $g \circ f \sim \mathrm{id}_X$ and $f \circ g \sim \mathrm{id}_Y$.
- Having a quasi-inverse means being an equivalence; the converse also holds.
- There may be more than one quasi-inverse, but two equivalences are always equal!

example: For each $X : \mathbf{U}$, $\mathrm{id}_X : X \simeq X$.

- Type equivalence is an equivalence relation.
- A **QUASI-INVERSE** to $f : X \to Y$ is a function $g : Y \to X$ with both $g \circ f \sim \mathrm{id}_X$ and $f \circ g \sim \mathrm{id}_Y$.
- Having a quasi-inverse means being an equivalence; the converse also holds.
- There may be more than one quasi-inverse, but two equivalences are always equal!
- This lets us ignore the term of $\mathrm{isEquiv}(f)$ and write $f : X \simeq Y$.

Axiom: Function extensionality

Let $X, Y : \mathbf{U}$ and $f, g : X \to Y$. We posit that

$$\mathsf{happly} : (f = g) \to (f \sim g)$$

is an equivalence, with a quasi-inverse

$$\mathsf{funext} : (f \sim g) \to (f = g).$$

That is, in HoTT, homotopy is equivalent to equality.

$$(f \sim g) \simeq (f = g)$$

There is a function similar to happly, but for paths in $\mathbf{U}$. We saw that $\mathsf{id}_X : X \simeq X$ for each $X : \mathbf{U}$. We can create a function

$$\mathsf{eqtoequiv} : (X = Y) \to (X \simeq Y)$$

by path induction, where $\mathsf{eqtoequiv}(\mathsf{refl}_X) :\equiv \mathsf{id}_X$.

There is a function similar to happly, but for paths in **U**. We saw that $\mathrm{id}_X : X \simeq X$ for each $X : \mathbf{U}$. We can create a function

$$\mathsf{eqtoequiv} : (X = Y) \to (X \simeq Y)$$

by path induction, where $\mathsf{eqtoequiv}(\mathsf{refl}_X) :\equiv \mathrm{id}_X$.

Axiom: Univalence

Let $X, Y : \mathbf{U}$. We posit that eqtoequiv is an equivalence, with a quasi-inverse

$$\mathsf{ua} : (X \simeq Y) \to (X = Y).$$

That is,

$$(X = Y) \simeq (X \simeq Y).$$

- Different proofs of a proposition are all equally valid at *letting us use the result*

Propositions and sets

Propositions

- Different proofs of a proposition are all equally valid at *letting us use the result*
- Different terms of a type are equally valid at showing its *inhabitedness*

Propositions and sets

Propositions

- Different proofs of a proposition are all equally valid at *letting us use the result*
- Different terms of a type are equally valid at showing its *inhabitedness*
- A **PROPOSITION** is a type that contains only the information of inhabitedness:

$$\mathsf{isProp}(X) :\equiv \prod_{(x,y:X)} (x = y).$$

Propositions and sets

Propositions

- Different proofs of a proposition are all equally valid at *letting us use the result*
- Different terms of a type are equally valid at showing its *inhabitedness*
- A **PROPOSITION** is a type that contains only the information of inhabitedness:

$$\mathrm{isProp}(X) :\equiv \prod_{(x,y:X)}(x = y).$$

- Any type $X : \mathbf{U}$ can be "truncated" to a proposition, written $\|X\| : \mathbf{U}$:

$$|\cdot| : X \to \|X\|, \qquad \mathrm{witness} : \prod_{(x,y:\|X\|)}(x = y).$$

Propositions and sets

Propositions

- Different proofs of a proposition are all equally valid at *letting us use the result*
- Different terms of a type are equally valid at showing its *inhabitedness*
- A **PROPOSITION** is a type that contains only the information of inhabitedness:

$$\mathrm{isProp}(X) :\equiv \prod_{(x,y:X)} (x = y).$$

- Any type $X : \mathbf{U}$ can be "truncated" to a proposition, written $\|X\| : \mathbf{U}$:

$$|\cdot| : X \to \|X\|, \qquad \mathrm{witness} : \prod_{(x,y:\|X\|)} (x = y).$$

- For any $X : \mathbf{U}$ and $x, y : X$, we always have $\mathrm{witness}(x, y) : |x| = |y|$, even if we don't have $x = y$.

Lemma

$\mathbf{0}$ and $\mathbf{1}$ are propositions.

Propositions and sets

Important lemmas

Lemma

$\mathbf{0}$ and $\mathbf{1}$ are propositions.

Lemma

Let $P, Q : \mathbf{U}$ be propositions. If $P \to Q$ and $Q \to P$, then $P \simeq Q$.

Propositions and sets

Important lemmas

Lemma

$\mathbf{0}$ and $\mathbf{1}$ are propositions.

Lemma

Let $P, Q : \mathbf{U}$ be propositions. If $P \to Q$ and $Q \to P$, then $P \simeq Q$.

Lemma

Let $X : \mathbf{U}$ be inhabited, and suppose $\mathsf{isProp}(X)$. Then $X \simeq \mathbf{1}$.

Propositions and sets

Traditional logical notation

If $P, Q : \mathbf{U}$ are propositions, then we can use the traditional propositional logic in the following way:

Traditional	HoTT
$\neg P$	$P \to \mathbf{0}$
$P \wedge Q$	$P \times Q$
$P \vee Q$	$\|P + Q\|$
$P \Rightarrow Q$	$P \to Q$
$P \Leftrightarrow Q$	$P = Q$.

Let $X : \mathbf{U}$ and suppose $Z : X \to \mathbf{U}$ is a family of propositions, *i.e.* $\prod_{(x:X)} \mathsf{isProp}(Z(x))$. Then we also have quantifiers:

$$
\begin{array}{cc}
\exists (x : X).\, Z(x) & \left\| \sum_{(x:X)} Z(x) \right\| \\
\forall (x : X).\, Z(x) & \prod_{(x:X)} Z(x).
\end{array}
$$

- We can make path types of path types: if we have $X : \mathbf{U}$, $x, y : X$, and $p, q : x = y$, then we can also make $p = q : \mathbf{U}$

Propositions and sets

Sets

- We can make path types of path types: if we have $X : \mathbf{U}$, $x, y : X$, and $p, q : x = y$, then we can also make $p = q : \mathbf{U}$
- When the path type on X is a proposition, we say that X is a **SET**:

$$\begin{aligned} \mathrm{isSet}(X) &:\equiv \prod_{(x,y:X)} \mathrm{isProp}(x = y) \\ &\equiv \prod_{(x,y:X)} \prod_{(p,q:x=y)} (p = q). \end{aligned}$$

Propositions and sets

Sets

- We can make path types of path types: if we have $X : \mathbf{U}$, $x, y : X$, and $p, q : x = y$, then we can also make $p = q : \mathbf{U}$
- When the path type on X is a proposition, we say that X is a **SET**:

$$\begin{aligned} \mathrm{isSet}(X) &:\equiv \prod_{(x,y:X)} \mathrm{isProp}(x = y) \\ &\equiv \prod_{(x,y:X)} \prod_{(p,q:x=y)} (p = q). \end{aligned}$$

- Any type X can be made into a set, $\|X\|_0$:

$$|\cdot|_0 : X \to \|X\|_0 , \qquad \mathrm{witness}_0 : \prod_{(x,y:\|X\|_0)} \prod_{(p,q:x=y)} (p = q).$$

Propositions and sets

Sets

- We can make path types of path types: if we have $X : \mathbf{U}$, $x, y : X$, and $p, q : x = y$, then we can also make $p = q : \mathbf{U}$
- When the path type on X is a proposition, we say that X is a **SET**:

$$\begin{aligned} \mathrm{isSet}(X) &:\equiv \prod_{(x,y:X)} \mathrm{isProp}(x = y) \\ &\equiv \prod_{(x,y:X)} \prod_{(p,q:x=y)} (p = q). \end{aligned}$$

- Any type X can be made into a set, $\|X\|_0$:

$$|\cdot|_0 : X \to \|X\|_0 \,, \qquad \mathrm{witness}_0 : \prod_{(x,y:\|X\|_0)} \prod_{(p,q:x=y)} (p = q).$$

- Propositions are sets: higher paths collapse when all terms are equal

Lemma

Let $X : \mathbf{U}$. Then $\mathsf{isProp}(X)$ and $\mathsf{isSet}(X)$ are propositions.

Lemma

Let $X : \mathbf{U}$. Then $\mathrm{isProp}(X)$ and $\mathrm{isSet}(X)$ are propositions.

- We can write

$$\begin{aligned} \mathbf{Prop} &:\equiv \textstyle\sum_{(X:\mathbf{U})} \mathrm{isProp}(X), \\ \mathbf{Set} &:\equiv \textstyle\sum_{(X:\mathbf{U})} \mathrm{isSet}(X) \end{aligned}$$

and also $X : \mathbf{Prop}$ or $Y : \mathbf{Set}$

Propositions and sets

Subsets and powersets

Lemma

Let $X : \mathbf{U}$. Then $\mathrm{isProp}(X)$ and $\mathrm{isSet}(X)$ are propositions.

- We can write

$$\begin{aligned} \mathbf{Prop}_i &:\equiv \textstyle\sum_{(X:\mathbf{U}_i)} \mathrm{isProp}(X), \\ \mathbf{Set}_i &:\equiv \textstyle\sum_{(X:\mathbf{U}_i)} \mathrm{isSet}(X) \end{aligned}$$

  and also $X : \mathbf{Prop}_i$ or $Y : \mathbf{Set}_i$
- Sub-universes: $\mathbf{Prop}_i, \mathbf{Set}_i : \mathbf{U}_{i+1}$

Lemma

Let $X : \mathbf{U}$. Then $\mathsf{isProp}(X)$ and $\mathsf{isSet}(X)$ are propositions.

- We can write

$$\begin{aligned}\mathbf{Prop}_i &:\equiv \textstyle\sum_{(X:\mathbf{U}_i)} \mathsf{isProp}(X),\\ \mathbf{Set}_i &:\equiv \textstyle\sum_{(X:\mathbf{U}_i)} \mathsf{isSet}(X)\end{aligned}$$

  and also $X : \mathbf{Prop}_i$ or $Y : \mathbf{Set}_i$
- Sub-universes: $\mathbf{Prop}_i, \mathbf{Set}_i : \mathbf{U}_{i+1}$
- For $X : \mathbf{Set}_i$, we write $\mathbf{P}(X) :\equiv X \to \mathbf{Prop}_i$ for the type of **SUBSETS**

- We can use traditional notation for subset operations:

$$
\begin{aligned}
-^{\complement} &: \mathbf{P}(X) \to \mathbf{P}(X) \\
A^{\complement} &:\equiv \lambda(x : X).\,(\neg A(x))
\end{aligned}
$$

$$
\begin{aligned}
\cup,\ \cap,\ \setminus &: \mathbf{P}(X) \to \mathbf{P}(X) \to \mathbf{P}(X) \\
A \cup B &:\equiv \lambda(x : X).\,(A(x) + B(x)) \\
A \cap B &:\equiv \lambda(x : X).\,(A(x) \times B(x)) \\
A \setminus B &:\equiv A \cap B^{\complement} \\
&\equiv \lambda(x : X).\,(A(x) \times (B(x) \to \mathbf{0}))
\end{aligned}
$$

Propositions and sets

Differences from classical sets

- A classical set uses an equivalence relation to define equality, that may not coincide with the path type

Propositions and sets

Differences from classical sets

- A classical set uses an equivalence relation to define equality, that may not coincide with the path type
- Sets in HoTT are *setoids* in traditional mathematics, since they have no such equivalence relation *a priori*

Propositions and sets

Differences from classical sets

- A classical set uses an equivalence relation to define equality, that may not coincide with the path type
- Sets in HoTT are *setoids* in traditional mathematics, since they have no such equivalence relation *a priori*
- No global membership operator (terms must have a type), but terms can be members of subsets:

$$\in : X \to \mathbf{P}(X) \to \mathbf{Prop}$$
$$x \in A :\equiv A(x).$$

Membership is the adjoint to evaluation!

Propositions and sets

Differences from classical sets

- A classical set uses an equivalence relation to define equality, that may not coincide with the path type
- Sets in HoTT are *setoids* in traditional mathematics, since they have no such equivalence relation *a priori*
- No global membership operator (terms must have a type), but terms can be members of subsets:

$$\in : X \to \mathbf{P}(X) \to \mathbf{Prop}$$
$$x \in A :\equiv A(x).$$

Membership is the adjoint to evaluation!

example: The type $\mathbf{N}$ is a set(oid), but not a proposition.

Conclusion

- Types were developed to restrict the application of formulae to sensible domains

Conclusion

- Types were developed to restrict the application of formulae to sensible domains
- Type formers behave similarly to set constructions

Conclusion

- Types were developed to restrict the application of formulae to sensible domains
- Type formers behave similarly to set constructions
- Extensionality and univalence clarify how functions and types can be interchanged

Conclusion

- Types were developed to restrict the application of formulae to sensible domains
- Type formers behave similarly to set constructions
- Extensionality and univalence clarify how functions and types can be interchanged
- HoTT models constructive propositional logic, and is consistent with AC and LEM

Conclusion

- Types were developed to restrict the application of formulae to sensible domains
- Type formers behave similarly to set constructions
- Extensionality and univalence clarify how functions and types can be interchanged
- HoTT models constructive propositional logic, and is consistent with AC and LEM
- Sets can also be modelled in HoTT, but they differ slightly from classical interpretations