Emily Riehl

Johns Hopkins University

Path induction and the indiscernibility of identicals

Plan

1. Induction over the natural numbers
2. Dependent type theory
3. Identity types
4. Path induction
5. Epilogue: univalent foundations

1

Induction over the natural numbers

Peano's postulates

In Dedekind's 1888 book "Was sind und was sollen die Zahlen" and Peano's 1889 paper "Arithmetices principia, nova methodo exposita," the natural numbers $\mathbb{N}$ are characterized by:

- There is a natural number $0 \in \mathbb{N}$.
- Every natural number $n \in \mathbb{N}$ has a successor $\mathsf{suc}\, n \in \mathbb{N}$.
- 0 is not the successor of any natural number.
- No two natural numbers have the same successor.
- The principle of mathematical induction:

$$\forall P, P(0) \to (\forall k \in \mathbb{N}, P(k) \to P(\mathsf{suc}\, k)) \to (\forall n \in \mathbb{N}, P(n))$$

By Dedekind's categoricity theorem, all triples given by a set $\mathbb{N}$, an element $0 \in \mathbb{N}$, and a function $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$ satisfying the Peano postulates are isomorphic.

Natural numbers induction

In the statement of the principle of mathematical induction:

$$\forall P, P(0) \to (\forall k \in \mathbb{N}, P(k) \to P(\mathsf{suc}\,k)) \to (\forall n \in \mathbb{N}, P(n))$$

the variable P is a predicate over the natural numbers.

A predicate over the natural numbers is a function

$$P : \mathbb{N} \to \{\top, \bot\}$$

that associates a truth value $\top$ or $\bot$ to each $n \in \mathbb{N}$.

Natural numbers induction

In the statement of the principle of mathematical induction:

$$\forall P, P(0) \to (\forall k \in \mathbb{N}, P(k) \to P(\mathsf{suc}\,k)) \to (\forall n \in \mathbb{N}, P(n))$$

the variable P is a predicate over the natural numbers.

A predicate over the natural numbers is a function

$$P \colon \mathbb{N} \to \{\top, \bot\}$$

that associates a truth value $\top$ or $\bot$ to each $n \in \mathbb{N}$.

Thus, to prove a sentence of the form $\forall n \in \mathbb{N}, P(n)$ it suffices to:

- prove the base case, showing that $P(0)$ is true, and
- prove the inductive step, showing for each $k \in \mathbb{N}$ that $P(k)$ implies $P(\mathsf{suc}\,k)$.

A proof by induction

Theorem. For any $n \in \mathbb{N}$, $n^2 + n$ is even.

Proof: By induction on $n \in \mathbb{N}$:

A proof by induction

Theorem. For any $n \in \mathbb{N}$, $n^2 + n$ is even.

Proof: By induction on $n \in \mathbb{N}$:

- In the base case, when $n = 0$, $0^2 + 0 = 2 \times 0$, which is even.

A proof by induction

Theorem. For any $n \in \mathbb{N}$, $n^2 + n$ is even.

Proof: By induction on $n \in \mathbb{N}$:

- In the base case, when $n = 0$, $0^2 + 0 = 2 \times 0$, which is even.
- For the inductive step, assume for $k \in \mathbb{N}$ that $k^2 + k = 2 \times m$ is even. Then

$$\begin{aligned} (k+1)^2 + (k+1) &= (k^2 + k) + ((2 \times k) + 2) \\ &= (2 \times m) + (2 \times (k+1)) \\ &= 2 \times (m + k + 1) \qquad \text{is even.} \end{aligned}$$

A proof by induction

Theorem. For any $n \in \mathbb{N}$, $n^2 + n$ is even.

Proof: By induction on $n \in \mathbb{N}$:

- In the base case, when $n = 0$, $0^2 + 0 = 2 \times 0$, which is even.
- For the inductive step, assume for $k \in \mathbb{N}$ that $k^2 + k = 2 \times m$ is even. Then

$$\begin{aligned}(k+1)^2 + (k+1) &= (k^2 + k) + ((2 \times k) + 2) \\ &= (2 \times m) + (2 \times (k+1)) \\ &= 2 \times (m + k + 1) \qquad \text{is even.}\end{aligned}$$

By the principle of mathematical induction

$$\forall P, P(0) \to (\forall k \in \mathbb{N}, P(k) \to P(\mathsf{suc}\,k)) \to (\forall n \in \mathbb{N}, P(n))$$

this proves that $n^2 + n$ is even for all $n \in \mathbb{N}$. $\square$

A construction by induction

The induction proof not only demonstrates for all $n \in \mathbb{N}$ that $n^2 + n$ is even but also defines a function $m : \mathbb{N} \to \mathbb{N}$ so that $n^2 + n = 2 \times m(n)$.

A construction by induction

The induction proof not only demonstrates for all $n \in \mathbb{N}$ that $n^2 + n$ is even but also defines a function $m : \mathbb{N} \to \mathbb{N}$ so that $n^2 + n = 2 \times m(n)$.

Construction: By induction on $n \in \mathbb{N}$:

A construction by induction

The induction proof not only demonstrates for all $n \in \mathbb{N}$ that $n^2 + n$ is even but also defines a function $m : \mathbb{N} \to \mathbb{N}$ so that $n^2 + n = 2 \times m(n)$.

Construction: By induction on $n \in \mathbb{N}$:

- In the base case, $0^2 + 0 = 2 \times 0$, so we define $m(0) \coloneqq 0$.

A construction by induction

The induction proof not only demonstrates for all $n \in \mathbb{N}$ that $n^2 + n$ is even but also defines a function $m : \mathbb{N} \to \mathbb{N}$ so that $n^2 + n = 2 \times m(n)$.

Construction: By induction on $n \in \mathbb{N}$:

- In the base case, $0^2 + 0 = 2 \times 0$, so we define $m(0) \coloneqq 0$.
- For the inductive step, assume for $k \in \mathbb{N}$ that $k^2 + k = 2 \times m(k)$. Then

$$\begin{aligned}
(k+1)^2 + (k+1) &= (k^2 + k) + ((2 \times k) + 2) \\
&= (2 \times m(k)) + (2 \times (k+1)) \\
&= 2 \times (m(k) + k + 1)
\end{aligned}$$

  so we define $m(k+1) \coloneqq m(k) + k + 1$.

A construction by induction

The induction proof not only demonstrates for all $n \in \mathbb{N}$ that $n^2 + n$ is even but also defines a function $m : \mathbb{N} \to \mathbb{N}$ so that $n^2 + n = 2 \times m(n)$.

Construction: By induction on $n \in \mathbb{N}$:

- In the base case, $0^2 + 0 = 2 \times 0$, so we define $m(0) \coloneqq 0$.
- For the inductive step, assume for $k \in \mathbb{N}$ that $k^2 + k = 2 \times m(k)$. Then

$$\begin{aligned}(k+1)^2 + (k+1) &= (k^2 + k) + ((2 \times k) + 2) \\ &= (2 \times m(k)) + (2 \times (k+1)) \\ &= 2 \times (m(k) + k + 1)\end{aligned}$$

  so we define $m(k+1) \coloneqq m(k) + k + 1$.

By the principle of mathematical recursion, this defines a function $m : \mathbb{N} \to \mathbb{N}$ so that $n^2 + n = 2 \times m(n)$ for all $n \in \mathbb{N}$.

Induction and recursion

Recursion can be thought of as the constructive form of induction

$$\forall P, P(0) \to (\forall k \in \mathbb{N}, P(k) \to P(\mathsf{suc}\, k)) \to (\forall n \in \mathbb{N}, P(n))$$

in which the predicate

$$P : \mathbb{N} \to \{\top, \bot\} \quad \text{such as} \quad P(n) \coloneqq \exists m \in \mathbb{N}, n^2 + n = 2 \times m$$

is replaced by an arbitrary family of sets

$$P : \mathbb{N} \to \mathsf{Set} \quad \text{such as} \quad P(n) \coloneqq \{m \in \mathbb{N} \mid n^2 + n = 2 \times m\}.$$

Induction and recursion

Recursion can be thought of as the constructive form of induction

$$\forall P, P(0) \to (\forall k \in \mathbb{N}, P(k) \to P(\mathsf{suc}\,k)) \to (\forall n \in \mathbb{N}, P(n))$$

in which the predicate

$$P : \mathbb{N} \to \{\top, \bot\} \quad \text{such as} \quad P(n) :\equiv \exists m \in \mathbb{N}, n^2 + n = 2 \times m$$

is replaced by an arbitrary family of sets

$$P : \mathbb{N} \to \mathsf{Set} \quad \text{such as} \quad P(n) :\equiv \{m \in \mathbb{N} \mid n^2 + n = 2 \times m\}.$$

The output of a recursive construction is a dependent function $p \in \prod_{n \in \mathbb{N}} P(n)$ which specifies a value $p(n) \in P(n)$ for each $n \in \mathbb{N}$.

$$\forall P, (p_0 \in P(0)) \to (p_s \in \prod_{k \in \mathbb{N}} P(k) \to P(\mathsf{suc}\,k)) \to (p \in \prod_{n \in \mathbb{N}} P(n))$$

Induction and recursion

Recursion can be thought of as the constructive form of induction

$$\forall P, P(0) \to (\forall k \in \mathbb{N}, P(k) \to P(\mathsf{suc}\, k)) \to (\forall n \in \mathbb{N}, P(n))$$

in which the predicate

$$P : \mathbb{N} \to \{\top, \bot\} \quad \text{such as} \quad P(n) :\equiv \exists m \in \mathbb{N}, n^2 + n = 2 \times m$$

is replaced by an arbitrary family of sets

$$P : \mathbb{N} \to \mathsf{Set} \quad \text{such as} \quad P(n) :\equiv \{m \in \mathbb{N} \mid n^2 + n = 2 \times m\}.$$

The output of a recursive construction is a dependent function $p \in \prod_{n \in \mathbb{N}} P(n)$ which specifies a value $p(n) \in P(n)$ for each $n \in \mathbb{N}$.

$$\forall P, (p_0 \in P(0)) \to (p_s \in \prod_{k \in \mathbb{N}} P(k) \to P(\mathsf{suc}\, k)) \to (p \in \prod_{n \in \mathbb{N}} P(n))$$

The recursive function $p \in \prod_{n \in \mathbb{N}} P(n)$ satisfies computation rules:

$$p(0) :\equiv p_0 \qquad p(\mathsf{suc}\, n) :\equiv p_s(n, p(n)).$$

The natural numbers in dependent type theory

While Peano's postulates characterize the natural numbers in set theory, the following rules characterize the natural numbers in dependent type theory:

The natural numbers in dependent type theory

While Peano's postulates characterize the natural numbers in set theory, the following rules characterize the natural numbers in dependent type theory:

- There is a type $\mathbb{N}$.

The natural numbers in dependent type theory

While Peano's postulates characterize the natural numbers in set theory, the following rules characterize the natural numbers in dependent type theory:

- There is a type $\mathbb{N}$.
- There is a term $0 : \mathbb{N}$ and a function $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$.

The natural numbers in dependent type theory

While Peano's postulates characterize the natural numbers in set theory, the following rules characterize the natural numbers in dependent type theory:

- There is a type $\mathbb{N}$.
- There is a term $0 : \mathbb{N}$ and a function $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$.
- For any family of types $P : \mathbb{N} \to \mathsf{Type}$ there is a term

$$\mathbb{N}\text{-ind} : (p_0 : P(0)) \to (p_s : \prod_{k:\mathbb{N}} P(k) \to P(\mathsf{suc}\,k)) \to (p : \prod_{n:\mathbb{N}} P(n))$$

The natural numbers in dependent type theory

While Peano's postulates characterize the natural numbers in set theory, the following rules characterize the natural numbers in dependent type theory:

- There is a type $\mathbb{N}$.
- There is a term $0 : \mathbb{N}$ and a function $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$.
- For any family of types $P : \mathbb{N} \to \mathsf{Type}$ there is a term

$$\mathbb{N}\text{-ind} : (p_0 : P(0)) \to (p_s : \prod_{k:\mathbb{N}} P(k) \to P(\mathsf{suc}\, k)) \to (p : \prod_{n:\mathbb{N}} P(n))$$

- Computation rules $p(0) :\equiv p_0$ and $p(\mathsf{suc}\, n) :\equiv p_s(n, p(n))$.

The natural numbers in dependent type theory

While Peano's postulates characterize the natural numbers in set theory, the following rules characterize the natural numbers in dependent type theory:

- There is a type $\mathbb{N}$.
- There is a term $0 : \mathbb{N}$ and a function $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$.
- For any family of types $P : \mathbb{N} \to \mathsf{Type}$ there is a term

$$\mathbb{N}\text{-ind} : (p_0 : P(0)) \to (p_s : \prod_{k:\mathbb{N}} P(k) \to P(\mathsf{suc}\,k)) \to (p : \prod_{n:\mathbb{N}} P(n))$$

- Computation rules $p(0) \coloneqq p_0$ and $p(\mathsf{suc}\,n) \coloneqq p_s(n, p(n))$.

Note the final two postulates — that 0 is not a successor and suc is injective — are missing because they are provable.

2

Dependent type theory

Types, terms, and contexts

Dependent type theory is a formal system for mathematical statements and proofs that has the following primitive notions:

- types, e.g., $\mathbb{N}$, $\mathbb{R}$, Group

Types, terms, and contexts

Dependent type theory is a formal system for mathematical statements and proofs that has the following primitive notions:

- types, e.g., $\mathbb{N}$, $\mathbb{R}$, Group
- terms, e.g., $17 : \mathbb{N}$, $\sqrt{2} : \mathbb{R}$, K_4 : Group

Types, terms, and contexts

Dependent type theory is a formal system for mathematical statements and proofs that has the following primitive notions:

- types, e.g., $\mathbb{N}$, $\mathbb{R}$, Group
- terms, e.g., $17 : \mathbb{N}$, $\sqrt{2} : \mathbb{R}$, $K_4 : \mathsf{Group}$
- type families, e.g., $\mathbb{R}^- : \mathbb{N} \to \mathsf{Type}$, $\mathsf{Mat}_{-\times-}(-) : \mathbb{N} \to \mathbb{N} \to \mathsf{Ring} \to \mathsf{Type}$

Types, terms, and contexts

Dependent type theory is a formal system for mathematical statements and proofs that has the following primitive notions:

- types, e.g., $\mathbb{N}$, $\mathbb{R}$, Group
- terms, e.g., $17 : \mathbb{N}$, $\sqrt{2} : \mathbb{R}$, $K_4 : \mathsf{Group}$
- type families, e.g., $\mathbb{R}^- : \mathbb{N} \to \mathsf{Type}$, $\mathsf{Mat}_{-\times -}(-) : \mathbb{N} \to \mathbb{N} \to \mathsf{Ring} \to \mathsf{Type}$
- dependent functions, e.g., $\vec{0}^{\bullet} : \prod_{n:\mathbb{N}} \mathbb{R}^n$, $I_\bullet : \prod_{n:\mathbb{N}} Mat_{n,n}(\mathbb{R})$

Types, terms, and contexts

Dependent type theory is a formal system for mathematical statements and proofs that has the following primitive notions:

- types, e.g., $\mathbb{N}$, $\mathbb{R}$, Group
- terms, e.g., $17 : \mathbb{N}$, $\sqrt{2} : \mathbb{R}$, $K_4 : \mathsf{Group}$
- type families, e.g., $\mathbb{R}^- : \mathbb{N} \to \mathsf{Type}$, $\mathsf{Mat}_{-\times-}(-) : \mathbb{N} \to \mathbb{N} \to \mathsf{Ring} \to \mathsf{Type}$
- dependent functions, e.g., $\vec{0}^{\bullet} : \prod_{n:\mathbb{N}} \mathbb{R}^n$, $I_\bullet : \prod_{n:\mathbb{N}} Mat_{n,n}(\mathbb{R})$

all of which can occur in an arbitrary context of variables from previously-defined types.

In a mathematical statement of the form "Let …be …then …" The stuff following the "let" likely declares the names of the variables in the context described after the "be", while the stuff after the "then" most likely describes a type or term in that context.

Type constructors

Type constructors build new types from given ones:

- given $A, B \rightsquigarrow$ products $A \times B$, coproducts $A + B$, function types $A \to B$,
- given $P : A \to \mathrm{Type} \rightsquigarrow$ dependent pairs $\sum_{x:A} P(x)$, dependent functions $\prod_{x:A} P(x)$
- given $A \rightsquigarrow$ identity types $- =_A - : A \to A \to \mathrm{Type}$

Type constructors

Type constructors build new types from given ones:

- given $A, B \rightsquigarrow$ products $A \times B$, coproducts $A + B$, function types $A \to B$,
- given $P : A \to \mathrm{Type} \rightsquigarrow$ dependent pairs $\sum_{x:A} P(x)$, dependent functions $\prod_{x:A} P(x)$
- given $A \rightsquigarrow$ identity types $- =_A - : A \to A \to \mathrm{Type}$

Each type constructor comes with rules:

(i) formation: a way to construct new types

(ii) introduction: ways to construct terms of these types

(iii) elimination: ways to use them to construct other terms

(iv) computation: the way (ii) and (iii) relate

Type constructors

Type constructors build new types from given ones:

- given $A, B \rightsquigarrow$ products $A \times B$, coproducts $A + B$, function types $A \to B$,
- given $P : A \to \mathrm{Type} \rightsquigarrow$ dependent pairs $\sum_{x:A} P(x)$, dependent functions $\prod_{x:A} P(x)$
- given $A \rightsquigarrow$ identity types $- =_A - : A \to A \to \mathrm{Type}$

Each type constructor comes with rules:

(i) formation: a way to construct new types

(ii) introduction: ways to construct terms of these types

(iii) elimination: ways to use them to construct other terms

(iv) computation: the way (ii) and (iii) relate

The rules suggest a logical naming for certain types:

$A \times B$	"A and B"	$\sum_{x:A} P(x)$	"$\exists x.P(x)$"
$A + B$	"A or B"	$\prod_{x:A} P(x)$	"$\forall x.P(x)$"
$A \to B$	"A implies B"	$x =_A y$	"x equals y"

Product types and function types

Product types are governed by the rules

$\times$-form: given types A and B there is a type $A \times B$

$\times$-intro: given terms $a : A$ and $b : B$ there is a term $(a, b) : A \times B$

$\times$-elim: given $p : A \times B$ there are terms $\mathrm{pr}_1 p : A$ and $\mathrm{pr}_2 p : B$

plus computation rules that relate pairings and projections.

Product types and function types

Product types are governed by the rules

$\times$-form: given types A and B there is a type $A \times B$

$\times$-intro: given terms $a : A$ and $b : B$ there is a term $(a, b) : A \times B$

$\times$-elim: given $p : A \times B$ there are terms $\mathrm{pr}_1 p : A$ and $\mathrm{pr}_2 p : B$

plus computation rules that relate pairings and projections.

Function types are governed by the rules

$\to$-form: given types A and B there is a type $A \to B$

$\to$-intro: if in the context of a variable $x : A$ there is a term $b_x : B$

there is a term $\lambda x.b_x : A \to B$

$\to$-elim: given terms $f : A \to B$ and $a : A$ there is a term $f(a) : B$

plus computation rules that relate λ-abstractions and evaluations.

Mathematics in dependent type theory

$^{\times}$-form: $A, B \rightsquigarrow A \times B$	$^{\rightarrow}$-form: A and $B \rightsquigarrow A \rightarrow B$
$^{\times}$-intro: $a : A,\ b : B \rightsquigarrow (a, b) : A \times B$	$^{\rightarrow}$-intro: $x : A \vdash b_x : B \rightsquigarrow \lambda x.b_x : A \rightarrow B$
$^{\times}$-elim: $p : A \times B \rightsquigarrow \mathrm{pr}_1 p : A,\ \mathrm{pr}_2 p : B$	$^{\rightarrow}$-elim: $f : A \rightarrow B,\ a : A \rightsquigarrow f(a) : B$

To prove a mathematical proposition in dependent type theory, one constructs a term in the type that encodes its statement.

Mathematics in dependent type theory

$^{\times}$-form: $A, B \rightsquigarrow A \times B$
$^{\times}$-intro: $a : A,\ b : B \rightsquigarrow (a, b) : A \times B$
$^{\times}$-elim: $p : A \times B \rightsquigarrow \mathrm{pr}_1 p : A,\ \mathrm{pr}_2 p : B$

$^{\rightarrow}$-form: A and $B \rightsquigarrow A \rightarrow B$
$^{\rightarrow}$-intro: $x : A \vdash b_x : B \rightsquigarrow \lambda x.b_x : A \rightarrow B$
$^{\rightarrow}$-elim: $f : A \rightarrow B,\ a : A \rightsquigarrow f(a) : B$

To prove a mathematical proposition in dependent type theory, one constructs a term in the type that encodes its statement.

Proposition. For any types A and B, $\mathsf{modus\text{-}ponens} : (A \times (A \rightarrow B)) \rightarrow B$.

Mathematics in dependent type theory

$^{\times}$-form: $A, B \rightsquigarrow A \times B$	$^{\to}$-form: A and $B \rightsquigarrow A \to B$
$^{\times}$-intro: $a : A,\ b : B \rightsquigarrow (a, b) : A \times B$	$^{\to}$-intro: $x : A \vdash b_x : B \rightsquigarrow \lambda x.b_x : A \to B$
$^{\times}$-elim: $p : A \times B \rightsquigarrow \mathrm{pr}_1 p : A,\ \mathrm{pr}_2 p : B$	$^{\to}$-elim: $f : A \to B,\ a : A \rightsquigarrow f(a) : B$

To prove a mathematical proposition in dependent type theory, one constructs a term in the type that encodes its statement.

Proposition. For any types A and B, $\text{modus-ponens} : (A \times (A \to B)) \to B$.

Construction: By $^{\to}$-intro, it suffices to assume given a term $p : (A \times (A \to B))$ and define a term of type B.

Mathematics in dependent type theory

$^{\times}$-form: $A, B \rightsquigarrow A \times B$
$^{\times}$-intro: $a : A,\ b : B \rightsquigarrow (a, b) : A \times B$
$^{\times}$-elim: $p : A \times B \rightsquigarrow \mathrm{pr}_1 p : A,\ \mathrm{pr}_2 p : B$

$^{\rightarrow}$-form: A and $B \rightsquigarrow A \to B$
$^{\rightarrow}$-intro: $x : A \vdash b_x : B \rightsquigarrow \lambda x.b_x : A \to B$
$^{\rightarrow}$-elim: $f : A \to B,\ a : A \rightsquigarrow f(a) : B$

To prove a mathematical proposition in dependent type theory, one constructs a term in the type that encodes its statement.

Proposition. For any types A and B, modus-ponens $: (A \times (A \to B)) \to B$.

Construction: By $^{\rightarrow}$-intro, it suffices to assume given a term $p : (A \times (A \to B))$ and define a term of type B. By $^{\times}$-elim, p provides terms $\mathrm{pr}_1 p : A$ and $\mathrm{pr}_2 p : A \to B$.

Mathematics in dependent type theory

$^{\times}$-form: $A, B \rightsquigarrow A \times B$
$^{\times}$-intro: $a : A,\ b : B \rightsquigarrow (a, b) : A \times B$
$^{\times}$-elim: $p : A \times B \rightsquigarrow \mathrm{pr}_1 p : A,\ \mathrm{pr}_2 p : B$

$^{\rightarrow}$-form: A and $B \rightsquigarrow A \rightarrow B$
$^{\rightarrow}$-intro: $x : A \vdash b_x : B \rightsquigarrow \lambda x.b_x : A \rightarrow B$
$^{\rightarrow}$-elim: $f : A \rightarrow B,\ a : A \rightsquigarrow f(a) : B$

To prove a mathematical proposition in dependent type theory, one constructs a term in the type that encodes its statement.

Proposition. For any types A and B, modus-ponens $: (A \times (A \rightarrow B)) \rightarrow B$.

Construction: By $^{\rightarrow}$-intro, it suffices to assume given a term $p : (A \times (A \rightarrow B))$ and define a term of type B. By $^{\times}$-elim, p provides terms $\mathrm{pr}_1 p : A$ and $\mathrm{pr}_2 p : A \rightarrow B$. By $^{\rightarrow}$-elim, these combine to give a term $\mathrm{pr}_2 p(\mathrm{pr}_1 p) : B$.

Mathematics in dependent type theory

$^{\times}$-form: $A, B \rightsquigarrow A \times B$	$^{\rightarrow}$-form: A and $B \rightsquigarrow A \rightarrow B$
$^{\times}$-intro: $a : A,\ b : B \rightsquigarrow (a, b) : A \times B$	$^{\rightarrow}$-intro: $x : A \vdash b_x : B \rightsquigarrow \lambda x.b_x : A \rightarrow B$
$^{\times}$-elim: $p : A \times B \rightsquigarrow \mathrm{pr}_1 p : A,\ \mathrm{pr}_2 p : B$	$^{\rightarrow}$-elim: $f : A \rightarrow B,\ a : A \rightsquigarrow f(a) : B$

To prove a mathematical proposition in dependent type theory, one constructs a term in the type that encodes its statement.

Proposition. For any types A and B, modus-ponens $: (A \times (A \rightarrow B)) \rightarrow B$.

Construction: By $^{\rightarrow}$-intro, it suffices to assume given a term $p : (A \times (A \rightarrow B))$ and define a term of type B. By $^{\times}$-elim, p provides terms $\mathrm{pr}_1 p : A$ and $\mathrm{pr}_2 p : A \rightarrow B$. By $^{\rightarrow}$-elim, these combine to give a term $\mathrm{pr}_2 p(\mathrm{pr}_1 p) : B$. Thus we have

$$\lambda p.\mathrm{pr}_2 p(\mathrm{pr}_1 p) : (A \times (A \rightarrow B)) \rightarrow B. \quad \square$$

3

Identity types

The traditional view of equality

In first order logic, the binary relation "$=$" is governed by the following rules:

- Reflexivity: $\forall x,\ x = x$.
- Indiscernibility of Identicals:

$$\forall x, y,\ x = y \text{ implies that for all predicates } P,\ P(x) \leftrightarrow P(y)$$

The traditional view of equality

In first order logic, the binary relation "$=$" is governed by the following rules:

- Reflexivity: $\forall x,\ x = x$.
- Indiscernibility of Identicals:

  $\forall x, y,\ x = y$ implies that for all predicates P, $P(x) \leftrightarrow P(y)$

Symmetry and transitivity of equality can be proven from these rules.

Interpreting identity types

The formation and introduction rules for identity types are:

$=$-form: given a type A and terms $x, y : A$, there is a type $x =_A y$

$=$-intro: given a type A and term $x : A$ there is a term $\mathrm{refl}_x : x =_A x$

Interpreting identity types

The formation and introduction rules for identity types are:

$=$-form: given a type A and terms $x, y : A$, there is a type $x =_A y$

$=$-intro: given a type A and term $x : A$ there is a term $\mathrm{refl}_x : x =_A x$

Note that identity types can be iterated:

given $x, y : A$ and $p, q : x =_A y$ there is a type $p =_{x =_A y} q$.

Does this type always have a term? In other words, are identity proofs unique?

Interpreting identity types

The formation and introduction rules for identity types are:

$^{=}$-form: given a type A and terms $x, y : A$, there is a type $x =_A y$

$^{=}$-intro: given a type A and term $x : A$ there is a term $\mathsf{refl}_x : x =_A x$

Note that identity types can be iterated:

given $x, y : A$ and $p, q : x =_A y$ there is a type $p =_{x =_A y} q$.

Does this type always have a term? In other words, are identity proofs unique?

No! From the existence of homotopical models of dependent type theory, in which

- types are interpreted as "spaces",
- terms are interpreted as points,

Interpreting identity types

The formation and introduction rules for identity types are:

$=$-form: given a type A and terms $x, y : A$, there is a type $x =_A y$

$=$-intro: given a type A and term $x : A$ there is a term $\mathrm{refl}_x : x =_A x$

Note that identity types can be iterated:

given $x, y : A$ and $p, q : x =_A y$ there is a type $p =_{x =_A y} q$.

Does this type always have a term? In other words, are identity proofs unique?

No! From the existence of homotopical models of dependent type theory, in which

- types are interpreted as "spaces",
- terms are interpreted as points,
- a term $p : x =_A y$ may be thought of as a path from x to y in A, and
- a term $h : p =_{x =_A y} q$ is interpreted as a homotopy between paths,

we know that iterated identity types can have interesting higher structure.

Path induction

We now introduce Martin-Löf's rules for identity types in full:

$=$-form: given a type A and terms $x, y : A$, there is a type $x =_A y$

$=$-intro: given a type A and term $x : A$ there is a term $\mathrm{refl}_x : x =_A x$

The elimination rule for the identity type defines an induction principle analogous to recursion over the natural numbers: it provides sufficient conditions for which to define a dependent function out of the identity type family.

Path induction

We now introduce Martin-Löf's rules for identity types in full:

$=$-form: given a type A and terms $x, y : A$, there is a type $x =_A y$

$=$-intro: given a type A and term $x : A$ there is a term $\mathsf{refl}_x : x =_A x$

The elimination rule for the identity type defines an induction principle analogous to recursion over the natural numbers: it provides sufficient conditions for which to define a dependent function out of the identity type family.

Now that terms $p : x =_A y$ are called paths, we re-brand $=$-elim as:

Path induction: For any type family $P(x, y, p)$ over $x, y : A, p : x =_A y$, to prove $P(x, y, p)$ for all x, y, p it suffices to assume y is x and p is refl_x. That is

$$\mathsf{path\text{-}ind} : \left(\prod_{x:A} P(x, x, \mathsf{refl}_x)\right) \to \left(\prod_{x,y:A} \prod_{p:x=_A y} P(x, y, p)\right).$$

Path induction

We now introduce Martin-Löf's rules for identity types in full:

$=$-form: given a type A and terms $x, y : A$, there is a type $x =_A y$

$=$-intro: given a type A and term $x : A$ there is a term $\mathsf{refl}_x : x =_A x$

The elimination rule for the identity type defines an induction principle analogous to recursion over the natural numbers: it provides sufficient conditions for which to define a dependent function out of the identity type family.

Now that terms $p : x =_A y$ are called paths, we re-brand $=$-elim as:

Path induction: For any type family $P(x, y, p)$ over $x, y : A, p : x =_A y$, to prove $P(x, y, p)$ for all x, y, p it suffices to assume y is x and p is refl_x. That is

$$\mathsf{path\text{-}ind} : \left(\prod_{x:A} P(x, x, \mathsf{refl}_x)\right) \to \left(\prod_{x,y:A} \prod_{p:x=_A y} P(x, y, p)\right).$$

A computation rule establishes that the proof of $P(x, x, \mathsf{refl}_x)$ is the given one.

4

Path induction

Reversal and concatenation of paths

Path induction: For any type family $P(x, y, p)$ over $x, y : A$, $p : x =_A y$

$$\text{path-ind} : \Big(\prod_{x:A} P(x, x, \mathsf{refl}_x)\Big) \to \Big(\prod_{x,y:A} \prod_{p:x=_A y} P(x, y, p)\Big).$$

Proposition. Paths can be reversed: $(-)^{-1} : \prod_{x,y:A} x =_A y \to y =_A x$.

Reversal and concatenation of paths

Path induction: For any type family $P(x, y, p)$ over $x, y : A$, $p : x =_A y$

$$\text{path-ind} : \left(\prod_{x:A} P(x, x, \mathsf{refl}_x)\right) \to \left(\prod_{x,y:A} \prod_{p:x=_A y} P(x, y, p)\right).$$

Proposition. Paths can be reversed: $(-)^{-1} : \prod_{x,y:A} x =_A y \to y =_A x$.

Construction: It suffices to assume $p : x =_A y$ and then define a term in the type $P(x, y, p) :\equiv y =_A x$.

Reversal and concatenation of paths

Path induction: For any type family $P(x, y, p)$ over $x, y : A, p : x =_A y$

$$\mathsf{path\text{-}ind} : \left(\prod_{x:A} P(x, x, \mathsf{refl}_x)\right) \to \left(\prod_{x,y:A} \prod_{p:x=_A y} P(x, y, p)\right).$$

Proposition. Paths can be reversed: $(-)^{-1} : \prod_{x,y:A} x =_A y \to y =_A x$.

Construction: It suffices to assume $p : x =_A y$ and then define a term in the type $P(x, y, p) :\equiv y =_A x$. By path induction, we may reduce to the case $P(x, x, \mathsf{refl}_x) :\equiv x =_A x$, for which we have the term $\mathsf{refl}_x : x =_A x$. $\square$

Reversal and concatenation of paths

Path induction: For any type family $P(x, y, p)$ over $x, y : A, p : x =_A y$

$$\text{path-ind} : \left(\prod_{x:A} P(x, x, \text{refl}_x)\right) \to \left(\prod_{x,y:A} \prod_{p:x=_A y} P(x, y, p)\right).$$

Proposition. Paths can be reversed: $(-)^{-1} : \prod_{x,y:A} x =_A y \to y =_A x$.

Construction: It suffices to assume $p : x =_A y$ and then define a term in the type $P(x, y, p) :\equiv y =_A x$. By path induction, we may reduce to the case $P(x, x, \text{refl}_x) :\equiv x =_A x$, for which we have the term $\text{refl}_x : x =_A x$. $\square$

Proposition. Paths can be concatenated: $* : \prod_{x,y,z:A} x =_A y \to (y =_A z \to x =_A z)$.

Reversal and concatenation of paths

Path induction: For any type family $P(x,y,p)$ over $x, y : A$, $p : x =_A y$

$$\text{path-ind} : \left(\prod_{x:A} P(x,x,\text{refl}_x)\right) \to \left(\prod_{x,y:A} \prod_{p:x=_A y} P(x,y,p)\right).$$

Proposition. Paths can be reversed: $(-)^{-1} : \prod_{x,y:A} x =_A y \to y =_A x$.

Construction: It suffices to assume $p : x =_A y$ and then define a term in the type $P(x,y,p) :\equiv y =_A x$. By path induction, we may reduce to the case $P(x,x,\text{refl}_x) :\equiv x =_A x$, for which we have the term $\text{refl}_x : x =_A x$. $\square$

Proposition. Paths can be concatenated: $* : \prod_{x,y,z:A} x =_A y \to (y =_A z \to x =_A z)$.

Construction: It suffices to assume $p : x =_A y$ and then define a term in the type $Q(x,y,p) :\equiv \prod_{z:A} y =_A z \to x =_A z$.

Reversal and concatenation of paths

Path induction: For any type family $P(x, y, p)$ over $x, y : A$, $p : x =_A y$

$$\text{path-ind} : \left(\prod_{x:A} P(x, x, \mathsf{refl}_x)\right) \to \left(\prod_{x,y:A} \prod_{p:x=_A y} P(x, y, p)\right).$$

Proposition. Paths can be reversed: $(-)^{-1} : \prod_{x,y:A} x =_A y \to y =_A x$.

Construction: It suffices to assume $p : x =_A y$ and then define a term in the type $P(x, y, p) :\equiv y =_A x$. By path induction, we may reduce to the case $P(x, x, \mathsf{refl}_x) :\equiv x =_A x$, for which we have the term $\mathsf{refl}_x : x =_A x$. $\square$

Proposition. Paths can be concatenated: $* : \prod_{x,y,z:A} x =_A y \to (y =_A z \to x =_A z)$.

Construction: It suffices to assume $p : x =_A y$ and then define a term in the type $Q(x, y, p) :\equiv \prod_{z:A} y =_A z \to x =_A z$. By path induction, we may reduce to the case $Q(x, x, \mathsf{refl}_x) :\equiv \prod_{z:A} x =_A z \to x =_A z$, for which we have the term $\mathsf{id} :\equiv \lambda q.q : x =_A z \to x =_A z$. $\square$

The ∞-groupoid of paths

Identity types can be iterated: given $x, y : A$ and $p, q : x =_A y$ there is a type $p =_{x =_A y} q$.

The ∞-groupoid of paths

Identity types can be iterated: given $x, y : A$ and $p, q : x =_A y$ there is a type $p =_{x =_A y} q$.

Theorem (Lumsdaine,Garner–van den Berg). The terms belonging to the iterated identity types of any type A form an ∞-groupoid.

The ∞-groupoid of paths

Identity types can be iterated: given $x, y : A$ and $p, q : x =_A y$ there is a type $p =_{x =_A y} q$.

Theorem (Lumsdaine, Garner–van den Berg). The terms belonging to the iterated identity types of any type A form an ∞-groupoid.

The ∞-groupoid structure of A has

- terms $x : A$ as objects
- paths $p : x =_A y$ as 1-morphisms
- paths of paths $h : p =_{x =_A y} q$ as 2-morphisms, ...

The ∞-groupoid of paths

Identity types can be iterated: given $x, y : A$ and $p, q : x =_A y$ there is a type $p =_{x =_A y} q$.

Theorem (Lumsdaine, Garner–van den Berg). The terms belonging to the iterated identity types of any type A form an ∞-groupoid.

The ∞-groupoid structure of A has

- terms $x : A$ as objects
- paths $p : x =_A y$ as 1-morphisms
- paths of paths $h : p =_{x =_A y} q$ as 2-morphisms, ...

The required structures are proven from the path induction principle:

- constant paths (reflexivity) $\mathrm{refl}_x : x = x$
- reversal (symmetry) $p : x = y$ yields $p^{-1} : y = x$
- concatenation (transitivity) $p : x = y$ and $q : y = z$ yield $p * q : x = z$

The ∞-groupoid of paths

Identity types can be iterated: given $x, y : A$ and $p, q : x =_A y$ there is a type $p =_{x =_A y} q$.

Theorem (Lumsdaine,Garner–van den Berg). The terms belonging to the iterated identity types of any type A form an ∞-groupoid.

The ∞-groupoid structure of A has

- terms $x : A$ as objects
- paths $p : x =_A y$ as 1-morphisms
- paths of paths $h : p =_{x =_A y} q$ as 2-morphisms, ...

The required structures are proven from the path induction principle:

- constant paths (reflexivity) $\mathrm{refl}_x : x = x$
- reversal (symmetry) $p : x = y$ yields $p^{-1} : y = x$
- concatenation (transitivity) $p : x = y$ and $q : y = z$ yield $p * q : x = z$

and furthermore concatenation is associative and unital, the associators are coherent ...

The higher coherences in path algebra

Path induction proves the (higher) coherences in the ∞-groupoid of paths:

Proposition. For any type A and terms $w, x, y, z : A$

$$\mathsf{assoc} : \prod_{p:w=_A x} \prod_{q:x=_A y} \prod_{r:y=_A z} (p * q) * r =_{w=_A z} p * (q * r).$$

The higher coherences in path algebra

Path induction proves the (higher) coherences in the ∞-groupoid of paths:

Proposition. For any type A and terms $w, x, y, z : A$

$$\mathsf{assoc} : \prod_{p : w =_A x} \prod_{q : x =_A y} \prod_{r : y =_A z} (p * q) * r =_{w =_A z} p * (q * r).$$

Construction: By path induction, it suffices to assume x is w and p is refl_w, reducing to the case

$$\prod_{q : w =_A y} \prod_{r : y =_A z} (\mathsf{refl}_w * q) * r =_{w =_A z} \mathsf{refl}_w * (q * r).$$

The higher coherences in path algebra

Path induction proves the (higher) coherences in the ∞-groupoid of paths:

Proposition. For any type A and terms $w, x, y, z : A$

$$\mathsf{assoc} : \prod_{p : w =_A x} \prod_{q : x =_A y} \prod_{r : y =_A z} (p * q) * r =_{w =_A z} p * (q * r).$$

Construction: By path induction, it suffices to assume x is w and p is refl_w, reducing to the case

$$\prod_{q : w =_A y} \prod_{r : y =_A z} (\mathsf{refl}_w * q) * r =_{w =_A z} \mathsf{refl}_w * (q * r).$$

By the computation rules for path induction $\mathsf{refl}_w * -$ is the identity function.

The higher coherences in path algebra

Path induction proves the (higher) coherences in the ∞-groupoid of paths:

Proposition. For any type A and terms $w, x, y, z : A$

$$\mathsf{assoc} : \prod_{p : w =_A x} \prod_{q : x =_A y} \prod_{r : y =_A z} (p * q) * r =_{w =_A z} p * (q * r).$$

Construction: By path induction, it suffices to assume x is w and p is refl_w, reducing to the case

$$\prod_{q : w =_A y} \prod_{r : y =_A z} (\mathsf{refl}_w * q) * r =_{w =_A z} \mathsf{refl}_w * (q * r).$$

By the computation rules for path induction $\mathsf{refl}_w * -$ is the identity function. Thus, we must show

$$\prod_{q : w =_A y} \prod_{r : y =_A z} q * r =_{w =_A z} q * r,$$

for which we have the proof $\mathsf{refl}_{q * r} : q * r =_{w =_A z} q * r$. $\square$

Indiscernibility of Identicals as path lifting

Indiscernibility of Identicals: $x = y$ implies that for all predicates P, $P(x) \leftrightarrow P(y)$

Indiscernibility of Identicals as path lifting

Indiscernibility of Identicals: $x = y$ implies that for all predicates P, $P(x) \leftrightarrow P(y)$

Let $P : A \to \mathsf{Type}$ be any family of types over A.

Proposition. For any $x, y : A$ if $p : x =_A y$ then $\mathsf{tr}_{P,p} : P(x) \to P(y)$.

Indiscernibility of Identicals as path lifting

Indiscernibility of Identicals: $x = y$ implies that for all predicates P, $P(x) \leftrightarrow P(y)$

Let $P : A \to \mathsf{Type}$ be any family of types over A.

Proposition. For any $x, y : A$ if $p : x =_A y$ then $\mathsf{tr}_{P,p} : P(x) \to P(y)$.

Construction: By path induction, it suffices to assume y is x and p is refl_x, in which case we have the identity function $\lambda x.x : P(x) \to P(x)$. $\square$

Indiscernibility of Identicals as path lifting

Indiscernibility of Identicals: $x = y$ implies that for all predicates P, $P(x) \leftrightarrow P(y)$

Let $P : A \to \mathsf{Type}$ be any family of types over A.

Proposition. For any $x, y : A$ if $p : x =_A y$ then $\mathrm{tr}_{P,p} : P(x) \to P(y)$.

Construction: By path induction, it suffices to assume y is x and p is refl_x, in which case we have the identity function $\lambda x.x : P(x) \to P(x)$. $\square$

Corollary. For any $x, y : A$ if $p : x =_A y$ then $P(x) \simeq P(y)$.

Indiscernibility of Identicals as path lifting

Indiscernibility of Identicals: $x = y$ implies that for all predicates P, $P(x) \leftrightarrow P(y)$

Let $P : A \to \mathsf{Type}$ be any family of types over A.

Proposition. For any $x, y : A$ if $p : x =_A y$ then $\mathsf{tr}_{P,p} : P(x) \to P(y)$.

Construction: By path induction, it suffices to assume y is x and p is refl_x, in which case we have the identity function $\lambda x.x : P(x) \to P(x)$. $\square$

Corollary. For any $x, y : A$ if $p : x =_A y$ then $P(x) \simeq P(y)$.

Construction: By path induction, it suffices to assume y is x and p is refl_x, in which case we have the identity equivalence. $\square$

5

Epilogue: univalent foundations

The homotopy type theoretic Rosetta stone

type theory	logic	set theory	homotopy theory
A	proposition	set	space
$x : A$	proof	element	point
$\emptyset, 1$	$\bot, \top$	$\emptyset, \{\emptyset\}$	$\emptyset, *$
$A \times B$	A and B	set of pairs	product space
$A + B$	A or B	disjoint union	coproduct
$A \to B$	A implies B	set of functions	function space
$P : A \to \mathsf{Type}$	predicate	family of sets	fibration
$f : \prod_{x:A} P(x)$	conditional proof	family of elements	section
$\prod_{x:A} P(x)$	$\forall x.P(x)$	product	space of sections
$\sum_{x:A} P(x)$	$\exists x.P(x)$	disjoint union	total space
$p : x =_A y$	proof of equality	$x = y$	path from x to y
$\sum_{x,y:A} x =_A y$	equality relation	diagonal	path space for A

Contractible types

The homotopical perspective on type theory suggests new definitions:

Contractible types

The homotopical perspective on type theory suggests new definitions:

A type A is contractible if it comes with a term of type

$$\text{is-contr}(A) :\equiv \sum_{a:A} \prod_{x:A} a =_A x$$

Contractible types

The homotopical perspective on type theory suggests new definitions:

A type A is contractible if it comes with a term of type

$$\text{is-contr}(A) := \sum_{a:A} \prod_{x:A} a =_A x$$

By $^{\Sigma}$-elim a proof of contractibility provides:

- a term $c : A$ called the center of contraction and
- a dependent function $h : \prod_{x:A} c =_A x$ called the contracting homotopy, which can be thought of as a continuous choice of paths $h(x) : c =_A x$ for each $x : A$.

The hierarchy of types

Contractible types, those types A for which the type

$$\text{is-contr}(A) := \sum_{a:A} \prod_{x:A} a =_A x$$

has a term, form the bottom level of Voevodsky's hierarchy of types.

The hierarchy of types

Contractible types, those types A for which the type

$$\text{is-contr}(A) := \sum_{a:A} \prod_{x:A} a =_A x$$

has a term, form the bottom level of Voevodsky's hierarchy of types.

A type A is

- a proposition if

$$\text{is-prop}(A) := \prod_{x,y:A} \text{is-contr}(x =_A y)$$

The hierarchy of types

Contractible types, those types A for which the type

$$\text{is-contr}(A) := \sum_{a:A} \prod_{x:A} a =_A x$$

has a term, form the bottom level of Voevodsky's hierarchy of types.

A type A is

- a proposition if
$$\text{is-prop}(A) := \prod_{x,y:A} \text{is-contr}(x =_A y)$$
- a set or 0-type if
$$\text{is-set}(A) := \prod_{x,y:A} \text{is-prop}(x =_A y)$$

The hierarchy of types

Contractible types, those types A for which the type

$$\text{is-contr}(A) :\equiv \sum_{a:A} \prod_{x:A} a =_A x$$

has a term, form the bottom level of Voevodsky's hierarchy of types.

A type A is

- a proposition if

$$\text{is-prop}(A) :\equiv \prod_{x,y:A} \text{is-contr}(x =_A y)$$

- a set or 0-type if

$$\text{is-set}(A) :\equiv \prod_{x,y:A} \text{is-prop}(x =_A y)$$

- a succ(n)-type for $n : \mathbb{N}$ if

$$\text{is-succ}(n)\text{-type}(A) :\equiv \prod_{x,y:A} \text{is-}n\text{-type}(x =_A y)$$

Equivalences

Similarly, homotopy theory suggests definitions of when two types A and B are equivalent or when a function $f : A \to B$ is an equivalence:

An equivalence between types A and B is a term of type:

$$A \simeq B :\equiv \sum_{f:A\to B} \left(\sum_{g:B\to A} \prod_{a:A} g(f(a)) =_A a \right) \times \left(\sum_{h:B\to A} \prod_{b:B} f(h(b)) =_B b \right)$$

A term of type $A \simeq B$ provides:

- functions $f : A \to B$ and $g, h : B \to A$ and
- homotopies α and β relating $g \circ f$ and $f \circ h$ to the identity functions.

Equivalences

Similarly, homotopy theory suggests definitions of when two types A and B are equivalent or when a function $f : A \to B$ is an equivalence:

An equivalence between types A and B is a term of type:

$$A \simeq B :\equiv \sum_{f:A\to B} \left(\sum_{g:B\to A} \prod_{a:A} g(f(a)) =_A a \right) \times \left(\sum_{h:B\to A} \prod_{b:B} f(h(b)) =_B b \right)$$

A term of type $A \simeq B$ provides:

- functions $f : A \to B$ and $g, h : B \to A$ and
- homotopies α and β relating $g \circ f$ and $f \circ h$ to the identity functions.

Using this data, one can define a homotopy from g to h.

Equivalences

Similarly, homotopy theory suggests definitions of when two types A and B are equivalent or when a function $f : A \to B$ is an equivalence:

An equivalence between types A and B is a term of type:

$$A \simeq B :\equiv \sum_{f:A\to B} \left(\sum_{g:B\to A} \prod_{a:A} g(f(a)) =_A a\right) \times \left(\sum_{h:B\to A} \prod_{b:B} f(h(b)) =_B b\right)$$

A term of type $A \simeq B$ provides:

- functions $f : A \to B$ and $g, h : B \to A$ and
- homotopies α and β relating $g \circ f$ and $f \circ h$ to the identity functions.

Using this data, one can define a homotopy from g to h.

So why not say $f : A \to B$ is an equivalence just when:

$$\sum_{g:B\to A} \left(\prod_{a:A} g(f(a)) =_A a\right) \times \left(\prod_{b:B} f(g(b)) =_B b\right)?$$

Equivalences

Similarly, homotopy theory suggests definitions of when two types A and B are equivalent or when a function $f : A \to B$ is an equivalence:

An equivalence between types A and B is a term of type:

$$A \simeq B :\equiv \sum_{f:A\to B} \left(\sum_{g:B\to A} \prod_{a:A} g(f(a)) =_A a\right) \times \left(\sum_{h:B\to A} \prod_{b:B} f(h(b)) =_B b\right)$$

A term of type $A \simeq B$ provides:

- functions $f : A \to B$ and $g, h : B \to A$ and
- homotopies α and β relating $g \circ f$ and $f \circ h$ to the identity functions.

Using this data, one can define a homotopy from g to h.

So why not say $f : A \to B$ is an equivalence just when:

$$\sum_{g:B\to A} \left(\prod_{a:A} g(f(a)) =_A a\right) \times \left(\prod_{b:B} f(g(b)) =_B b\right)?$$

This type is not a proposition and may have non-trivial higher structure.

The univalence axiom

Another notion of sameness between types is provided by the universe $\mathcal{U}$ of types, which has (small) types A, B as its terms $\quad\rightsquigarrow\quad A, B : \mathcal{U}$.

Q: How do the types $A =_{\mathcal{U}} B$ and $A \simeq B$ compare?

The univalence axiom

Another notion of sameness between types is provided by the universe $\mathcal{U}$ of types, which has (small) types A, B as its terms $\rightsquigarrow$ $A, B : \mathcal{U}$.

Q: How do the types $A =_{\mathcal{U}} B$ and $A \simeq B$ compare?

By path induction, there is a canonical function

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

defined by sending refl_A to the identity equivalence id_A.

The univalence axiom

Another notion of sameness between types is provided by the universe $\mathcal{U}$ of types, which has (small) types A, B as its terms $\quad\rightsquigarrow\quad A, B : \mathcal{U}$.

Q: How do the types $A =_{\mathcal{U}} B$ and $A \simeq B$ compare?

By path induction, there is a canonical function

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

defined by sending refl_A to the identity equivalence id_A.

Univalence Axiom: The function $\text{id-to-equiv} : (A =_{\mathcal{U}} B) \to (A \simeq B)$ is an equivalence.

"Identity is equivalent to equivalence."

$$(A =_{\mathcal{U}} B) \simeq (A \simeq B)$$

The univalence axiom

Another notion of sameness between types is provided by the universe $\mathcal{U}$ of types, which has (small) types A, B as its terms $\rightsquigarrow$ $A, B : \mathcal{U}$.

Q: How do the types $A =_{\mathcal{U}} B$ and $A \simeq B$ compare?

By path induction, there is a canonical function

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

defined by sending refl_A to the identity equivalence id_A.

Univalence Axiom: The function $\text{id-to-equiv} : (A =_{\mathcal{U}} B) \to (A \simeq B)$ is an equivalence.

"Identity is equivalent to equivalence."

$$(A =_{\mathcal{U}} B) \simeq (A \simeq B)$$

Consequences of univalence

There are myriad consequences of the univalence axiom $(A =_{\mathcal{U}} B) \simeq (A \simeq B)$:

Consequences of univalence

There are myriad consequences of the univalence axiom $(A =_{\mathcal{U}} B) \simeq (A \simeq B)$:

- The structure-identity principle, which specializes to the statement that for set-based structures (monoids, groups, rings) isomorphic structures are identical.

Consequences of univalence

There are myriad consequences of the univalence axiom $(A =_{\mathcal{U}} B) \simeq (A \simeq B)$:

- The structure-identity principle, which specializes to the statement that for set-based structures (monoids, groups, rings) isomorphic structures are identical.
- Function extensionality: for any $f, g : A \to B$, the canonical function defines an equivalence between the identity type and the type of homotopies:

$$\text{id-to-htpy} : (f =_{A \to B} g) \to \left(\prod_{a:A} f(a) =_B g(a)\right)$$

Consequences of univalence

There are myriad consequences of the univalence axiom $(A =_{\mathcal{U}} B) \simeq (A \simeq B)$:

- The structure-identity principle, which specializes to the statement that for set-based structures (monoids, groups, rings) isomorphic structures are identical.
- Function extensionality: for any $f, g : A \to B$, the canonical function defines an equivalence between the identity type and the type of homotopies:

$$\text{id-to-htpy} : (f =_{A \to B} g) \to \left(\prod_{a:A} f(a) =_B g(a) \right)$$

- By indiscernibility of identicals, if $x, y : A$ and $x =_A y$ then $P(x) \simeq P(y)$ for any $P : A \to \mathrm{Type}$. By univalence, whenever $A \simeq B$ then $A =_{\mathcal{U}} B$ and thus any type constructed from A is equivalent to the corresponding type constructed from B.

Consequences of univalence

There are myriad consequences of the univalence axiom $(A =_{\mathcal{U}} B) \simeq (A \simeq B)$:

- The structure-identity principle, which specializes to the statement that for set-based structures (monoids, groups, rings) isomorphic structures are identical.
- Function extensionality: for any $f, g : A \to B$, the canonical function defines an equivalence between the identity type and the type of homotopies:

$$\text{id-to-htpy} : (f =_{A \to B} g) \to \left(\prod_{a:A} f(a) =_B g(a)\right)$$

- By indiscernibility of identicals, if $x, y : A$ and $x =_A y$ then $P(x) \simeq P(y)$ for any $P : A \to \text{Type}$. By univalence, whenever $A \simeq B$ then $A =_{\mathcal{U}} B$ and thus any type constructed from A is equivalent to the corresponding type constructed from B.

Via path induction, Voevodsky's univalence axiom — which is justified by the homotopical model of type theory — captures the common mathematical practice of applying results proven about one object to any other object that is equivalent to it!

What justifies the path induction principle?

Path induction: For any type family $P(x, y, p)$ over $x, y : A$, $p : x =_A y$, to prove $P(x, y, p)$ for all x, y, p it suffices to assume y is x and p is refl_x.

$$\mathsf{path\text{-}ind} : \left(\prod_{x:A} P(x, x, \mathsf{refl}_x)\right) \to \left(\prod_{x,y:A} \prod_{p:x=_A y} P(x, y, p)\right).$$

What justifies the path induction principle?

Path induction: For any type family $P(x, y, p)$ over $x, y : A$, $p : x =_A y$, to prove $P(x, y, p)$ for all x, y, p it suffices to assume y is x and p is refl_x.

$$\text{path-ind} : \left(\prod_{x:A} P(x, x, \mathrm{refl}_x)\right) \to \left(\prod_{x,y:A} \prod_{p:x=_A y} P(x, y, p)\right).$$

Path induction asserts that to map out of a path space $\sum_{x,y:A} x =_A y$ it suffices to define the images of the reflexivity paths.

What justifies the path induction principle?

Path induction: For any type family $P(x, y, p)$ over $x, y : A$, $p : x =_A y$, to prove $P(x, y, p)$ for all x, y, p it suffices to assume y is x and p is refl_x.

$$\mathsf{path\text{-}ind} : \left(\prod_{x:A} P(x, x, \mathsf{refl}_x)\right) \to \left(\prod_{x,y:A} \prod_{p:x=_A y} P(x, y, p)\right).$$

Path induction asserts that to map out of a path space $\sum_{x,y:A} x =_A y$ it suffices to define the images of the reflexivity paths.

Proposition. For each $x : A$, the based path space $\sum_{y:A} x =_A y$ is contractible with center of contraction given by the point (x, refl_x).

What justifies the path induction principle?

Path induction: For any type family $P(x, y, p)$ over $x, y : A$, $p : x =_A y$, to prove $P(x, y, p)$ for all x, y, p it suffices to assume y is x and p is refl_x.

$$\text{path-ind} : \left(\prod_{x:A} P(x, x, \mathrm{refl}_x)\right) \to \left(\prod_{x,y:A} \prod_{p:x=_A y} P(x, y, p)\right).$$

Path induction asserts that to map out of a path space $\sum_{x,y:A} x =_A y$ it suffices to define the images of the reflexivity paths.

Proposition. For each $x : A$, the based path space $\sum_{y:A} x =_A y$ is contractible with center of contraction given by the point (x, refl_x).

Corollary. The function $\lambda x.(x, x, \mathrm{refl}_x) : A \to \left(\sum_{x,y:A} x =_A y\right)$ is an equivalence.

What justifies the path induction principle?

Path induction: For any type family $P(x, y, p)$ over $x, y : A$, $p : x =_A y$, to prove $P(x, y, p)$ for all x, y, p it suffices to assume y is x and p is refl_x.

$$\mathsf{path\text{-}ind} : \left(\prod_{x:A} P(x, x, \mathsf{refl}_x)\right) \to \left(\prod_{x,y:A} \prod_{p:x=_A y} P(x, y, p)\right).$$

Path induction asserts that to map out of a path space $\sum_{x,y:A} x =_A y$ it suffices to define the images of the reflexivity paths.

Proposition. For each $x : A$, the based path space $\sum_{y:A} x =_A y$ is contractible with center of contraction given by the point (x, refl_x).

Corollary. The function $\lambda x.(x, x, \mathsf{refl}_x) : A \to \left(\sum_{x,y:A} x =_A y\right)$ is an equivalence.

By univalence, the equivalence $A \simeq \left(\sum_{x,y:A} x =_A y\right)$ gives rise to an equivalence

$$\left(\prod_{x:A} P(x, x, \mathsf{refl}_x)\right) \simeq \left(\prod_{(x,y,p):\sum_{x,y:A} x=_A y} P(x, y, p)\right) \simeq \left(\prod_{x,y:A} \prod_{p:x=_A y} P(x, y, p)\right).$$

References

Homotopy Type Theory: Univalent Foundations of Mathematics

homotopytypetheory.org/book/

Egbert Rijke, Introduction to Homotopy Type Theory

arXiv:2212.11082

HoTTEST Summer School, July–August 2022

discord.gg/tkhJ9zCGs9

Thank you!