[Homotopy (Type] Theory)

Nikola Tomić

January 25, 2023

Classical Logic and Set Theory

Classical logic can be formally defined using :

Classical logic can be formally defined using :

- Logical operators : $\wedge$, $\vee$, $\bot$, $\top$, $\Rightarrow$, $\neg$, $\forall$, $\exists$.

Classical logic can be formally defined using :

- Logical operators : $\wedge$, $\vee$, $\bot$, $\top$, $\Rightarrow$, $\neg$, $\forall$, $\exists$.
- Propositions : A, B, ...

Classical logic can be formally defined using :

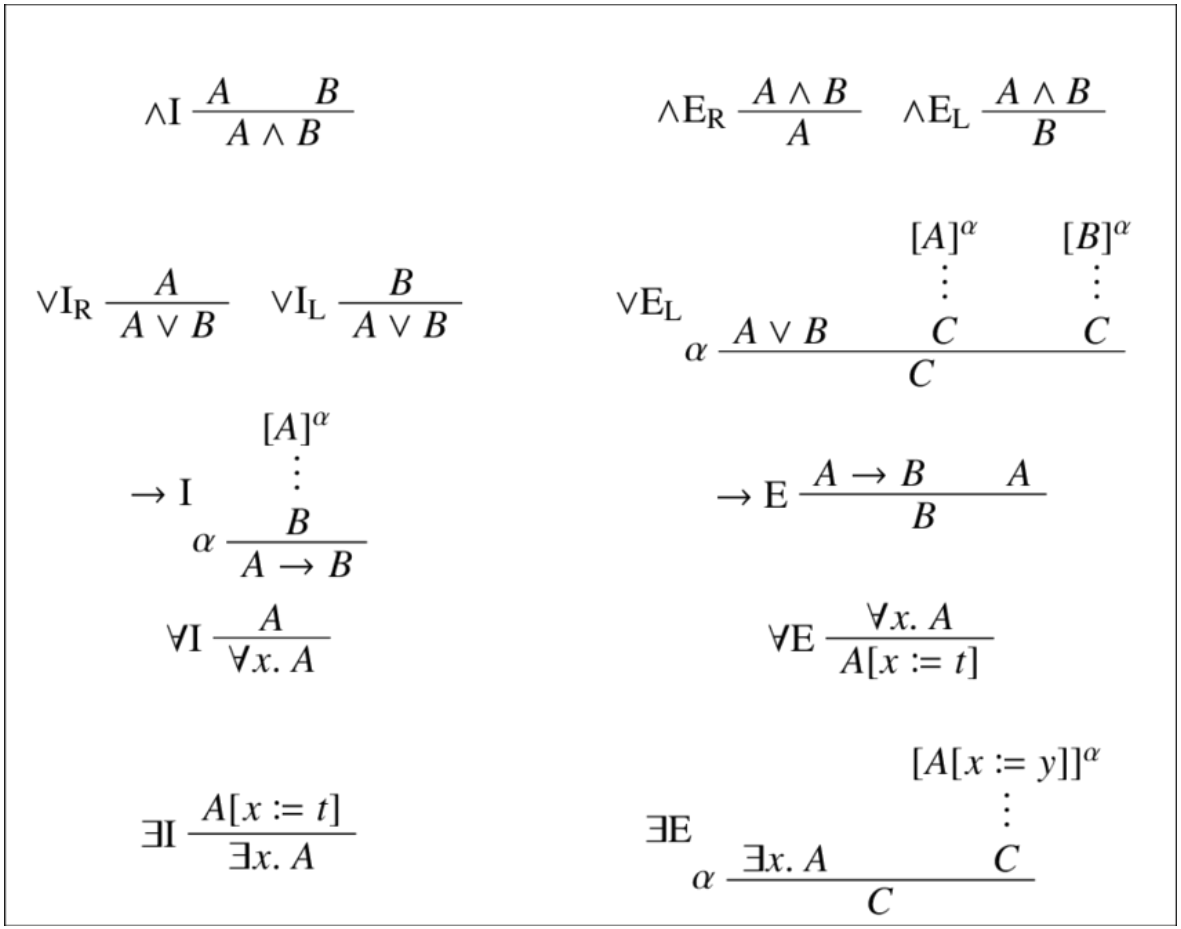
- Logical operators : $\wedge$, $\vee$, $\bot$, $\top$, $\Rightarrow$, $\neg$, $\forall$, $\exists$.
- Propositions : A, B, ...
- Variables : x, y, ...

Classical logic can be formally defined using :

- Logical operators : $\wedge$, $\vee$, $\bot$, $\top$, $\Rightarrow$, $\neg$, $\forall$, $\exists$.
- Propositions : A, B, ...
- Variables : x, y, ...

Satisfying some rules called **judgments**.

Here is an example of a system modelizing rules of classical logic (this is called natural deduction) :



Set theory

Set theory is the theory used by the usual mathematicians, it is defined *in* classical logic.

Set theory is the theory used by the usual mathematicians, it is defined *in* classical logic.
To be defined we need three things :

Set theory is the theory used by the usual mathematicians, it is defined *in* classical logic.
To be defined we need three things :

- A theory of equality. It is the data of a symbol $=$ that satisfies the Leibniz's law : $\forall x \forall y, x = y \iff (\forall P, P(x) \iff P(y))$.

Set theory is the theory used by the usual mathematicians, it is defined *in* classical logic.
To be defined we need three things :

- A theory of equality. It is the data of a symbol $=$ that satisfies the Leibniz's law : $\forall x \forall y, x = y \iff (\forall P, P(x) \iff P(y))$.
- 2 Symbols : $\in$ and $\varnothing$.

Set theory is the theory used by the usual mathematicians, it is defined *in* classical logic.
To be defined we need three things :

- A theory of equality. It is the data of a symbol $=$ that satisfies the Leibniz's law : $\forall x \forall y, x = y \iff (\forall P, P(x) \iff P(y))$.
- 2 Symbols : $\in$ and $\varnothing$.
- Axioms. (Usually ZF(+C for everyone else here I believe...))

The axioms of Zermelo-Fraenkel set theory with choice ZFC

In principle all of mathematics can be derived from these axioms

Extensionality	$\forall X \forall Y [X = Y \quad \Leftrightarrow \quad \forall z(z \in X \Leftrightarrow z \in Y)]$
Pairing	$\forall x \forall y \exists Z \forall z [z \in Z \quad \Leftrightarrow \quad z = x \text{ or } z = y]$
Union	$\forall X \exists Y \forall y [y \in Y \quad \Leftrightarrow \quad \exists Z(Z \in X \text{ and } y \in Z)]$
Empty set	$\exists X \forall y [y \notin X]$ (this set X is denoted by $\emptyset$)
Infinity	$\exists X [\emptyset \in X \text{ and } \forall x(x \in X \Rightarrow x \cup \{x\} \in X)]$
Power set	$\forall X \exists Y \forall Z [Z \in Y \quad \Leftrightarrow \quad \forall z(z \in Z \Rightarrow z \in X)]$
Replacement	$\forall x \in X \exists! y P(x, y) \quad \Rightarrow \quad [\exists Y \forall y (y \in Y \Leftrightarrow \exists x \in X (P(x, y)))]$
Regularity	$\forall X [X \neq \emptyset \quad \Rightarrow \quad \exists Y \in X (X \cap Y = \emptyset)]$
Axiom of choice	$\forall X [\emptyset \notin X \text{ and } \forall Y, Z \in X(Y \neq Z \Rightarrow Y \cap Z = \emptyset) \Rightarrow \quad \exists Y \forall Z \in X \exists! z \in Z (z \in Y)]$

$\forall =$ for all $\quad \exists! =$ there exists a unique $\quad P$ is any formula that does not contain Y

$z \in X \cup Y \Leftrightarrow z \in X$ or $z \in Y \qquad z \in X \cap Y \Leftrightarrow z \in X$ and $z \in Y$

Set theory axioms — Math Poster 2007 — math.chapman.edu

The axioms of Zermelo-Fraenkel set theory with choice ZFC

In principle all of mathematics can be derived from these axioms

Extensionality	$\forall X\,\forall Y\,[\,X = Y \;\Leftrightarrow\; \forall z(z \in X \;\Leftrightarrow\; z \in Y)\,]$
Pairing	$\forall x\,\forall y\,\exists Z\,\forall z\,[\,z \in Z \;\Leftrightarrow\; z = x$ or $z = y\,]$
Union	$\forall X\,\exists Y\,\forall y\,[\,y \in Y \;\Leftrightarrow\; \exists Z(Z \in X$ and $y \in Z)\,]$
Empty set	$\exists X\,\forall y\,[\,y \notin X\,]$ (this set X is denoted by $\emptyset$)
Infinity	$\exists X\,[\,\emptyset \in X$ and $\forall x(x \in X \Rightarrow x \cup \{x\} \in X)\,]$
Power set	$\forall X\,\exists Y\,\forall Z\,[\,Z \in Y \;\Leftrightarrow\; \forall z(z \in Z \;\Rightarrow\; z \in X)\,]$
Replacement	$\forall x \in X\,\exists! y\,P(x,y) \;\Rightarrow\; [\,\exists Y\,\forall y\,(y \in Y \;\Leftrightarrow\; \exists x \in X\,(P(x,y)))\,]$
Regularity	$\forall X\,[\,X \neq \emptyset \;\Rightarrow\; \exists Y \in X\,(X \cap Y = \emptyset)\,]$
Axiom of choice	$\forall X\,[\,\emptyset \notin X$ and $\forall Y, Z \in X(Y \neq Z \;\Rightarrow Y \cap Z = \emptyset)$ $\Rightarrow \;\exists Y\,\forall Z \in X\,\exists! z \in Z\,(z \in Y)\,]$

$\forall =$ for all $\quad \exists! =$ there exists a unique $\quad P$ is any formula that does not contain Y

$z \in X \cup Y \;\Leftrightarrow\; z \in X$ or $z \in Y \qquad z \in X \cap Y \;\Leftrightarrow\; z \in X$ and $z \in Y$

Set theory axioms — Math Poster 2007 — math.chapman.edu

One does not define *what* is a set but instead defines how they behave.

Flaws :

Flaws (for constructivists) :

Flaws (for constructivists) :

- You can't know how an object is built : proofs of existence don't give you any algorithm to get the object wanted.

Flaws (for constructivists) :

- You can't know how an object is built : proofs of existence don't give you any algorithm to get the object wanted.
- Has well-formed propositions with no meaning at all : e.g. $1 = \mathbb{N}$.

Type Theory

Type Theory is also system of symbols and rules but covers both (constructivist) Classical Logic *and* Set Theory.

Type Theory is also system of symbols and rules but covers both (constructivist) Classical Logic *and* Set Theory.

- Variables x, y, ... are given with a **type** : $x : X$, $y : Y$, ... When $x : X$ is fixed we may think of x as an element of type X.

Type Theory is also system of symbols and rules but covers both (constructivist) Classical Logic *and* Set Theory.

- Variables x, y, ... are given with a **type** : $x : X$, $y : Y$, ... When $x : X$ is fixed we may think of x as an element of type X.
- Satisfying also some judgment rules that are used to build new types : $X \times Y$, $X \to Y$, $x =_X y$, $* : \mathbf{1}$, ...

Type Theory is also system of symbols and rules but covers both (constructivist) Classical Logic *and* Set Theory.

- Variables x, y, ... are given with a **type** : $x : X$, $y : Y$, ... When $x : X$ is fixed we may think of x as an element of type X.
- Satisfying also some judgment rules that are used to build new types : $X \times Y$, $X \to Y$, $x =_X y$, $* : \mathbf{1}$, ...
- There is an interesting feature, there are *two* kind of equalities.

Type Theory is also system of symbols and rules but covers both (constructivist) Classical Logic *and* Set Theory.

- Variables x, y, ... are given with a **type** : $x : X$, $y : Y$, ... When $x : X$ is fixed we may think of x as an element of type X.
- Satisfying also some judgment rules that are used to build new types : $X \times Y$, $X \to Y$, $x =_X y$, $* : \mathbf{1}$, ...
- There is an interesting feature, there are *two* kind of equalities.
  One is the **equality type** $x =_X y$ where $x, y : X$, which is a type and elements $p : x =_X y$ of that type can be thought as proofs that x is "equal" to y.

Type Theory is also system of symbols and rules but covers both (constructivist) Classical Logic *and* Set Theory.

- Variables x, y, ... are given with a **type** : $x : X$, $y : Y$, ... When $x : X$ is fixed we may think of x as an element of type X.
- Satisfying also some judgment rules that are used to build new types : $X \times Y$, $X \to Y$, $x =_X y$, $* : \mathbf{1}$, ...
- There is an interesting feature, there are *two* kind of equalities.
  One is the **equality type** $x =_X y$ where $x, y : X$, which is a type and elements $p : x =_X y$ of that type can be thought as proofs that x is "equal" to y.
  The other one is the **judgmental equality** $x \equiv y$ which means that x and y are the *same* by definition.

Type Theory is also system of symbols and rules but covers both (constructivist) Classical Logic *and* Set Theory.

- Variables x, y, ... are given with a **type** : $x : X$, $y : Y$, ... When $x : X$ is fixed we may think of x as an element of type X.
- Satisfying also some judgment rules that are used to build new types : $X \times Y$, $X \to Y$, $x =_X y$, $* : \mathbf{1}$, ...
- There is an interesting feature, there are *two* kind of equalities.
  One is the **equality type** $x =_X y$ where $x, y : X$, which is a type and elements $p : x =_X y$ of that type can be thought as proofs that x is "equal" to y.
  The other one is the **judgmental equality** $x \equiv y$ which means that x and y are the *same* by definition.

For instance, in Set theory you may write 0 for $\varnothing$, we shall write $0 \equiv \varnothing$. Or $1 \equiv \{\varnothing\}$. It is written when there is nothing to prove.

Examples of types

Examples of types

- The type $\mathbf{1}$ with one unique element of this type $* : \mathbf{1}$, it corresponds to the singleton set.

Examples of types

- The type $\mathbf{1}$ with one unique element of this type $* : \mathbf{1}$, it corresponds to the singleton set.
- The type $\mathbf{0}$. There isn't any element of this type, it corresponds to the empty set $\varnothing$.

Examples of types

- The type $\mathbf{1}$ with one unique element of this type $* : \mathbf{1}$, it corresponds to the singleton set.
- The type $\mathbf{0}$. There isn't any element of this type, it corresponds to the empty set $\varnothing$.
- The type of integers $\mathbb{N}$. Integers are elements of type $\mathbb{N}$. e.g. $2 : \mathbb{N}$.

Examples of types

- The type $\mathbf{1}$ with one unique element of this type $* : \mathbf{1}$, it corresponds to the singleton set.
- The type $\mathbf{0}$. There isn't any element of this type, it corresponds to the empty set $\varnothing$.
- The type of integers $\mathbb{N}$. Integers are elements of type $\mathbb{N}$. e.g. $2 : \mathbb{N}$.
- The type of groups **Grp**. Groups are elements of that type. e.g. $S_n : \mathbf{Grp}$.

Examples of types

- The type $\mathbf{1}$ with one unique element of this type $* : \mathbf{1}$, it corresponds to the singleton set.
- The type $\mathbf{0}$. There isn't any element of this type, it corresponds to the empty set $\varnothing$.
- The type of integers $\mathbb{N}$. Integers are elements of type $\mathbb{N}$. e.g. $2 : \mathbb{N}$.
- The type of groups **Grp**. Groups are elements of that type. e.g. S_n : **Grp**.
- The type of booleans $\mathbf{2}$ with two elements : **true** : $\mathbf{2}$ and **false** : $\mathbf{2}$.

Construction of types/Judgment rules

Product type

Product type

$$(x : X, y : Y) \vdash ((x, y) : X \times Y)$$

Product type

$$(x : X, y : Y) \vdash ((x, y) : X \times Y)$$

$$(w : X \times Y) \vdash (x : X, y : Y, (x, y) \equiv w)$$

Sum type

Sum type

$$(x : X) \vdash (x : X + Y),\ (y : Y) \vdash (y : X + Y)$$

Function type

Function type

$$(f : X \to Y, x : X) \vdash (f(x) : Y)$$

Function type

$$(f : X \to Y, x : X) \vdash (f(x) : Y)$$

$$(X, Y, \Phi \text{ an expression involving } x : X) \vdash (f : X \to Y, f(x) \equiv \Phi)$$

Function type

$$(f : X \to Y, x : X) \vdash (f(x) : Y)$$

$$(X, Y, \Phi \text{ an expression involving } x : X) \vdash (f : X \to Y, f(x) \equiv \Phi)$$

We have automatically a function $\mathrm{id}_X : X \to X$ and a composition $\circ : (X \to Y) \times (Y \to Z) \to (X \to Z)$.

Equality type

Equality type

$$(x, y : X) \vdash (x =_X y)$$

Equality type

$$(x, y : X) \vdash (x =_X y)$$

The type $x =_X y$ satisfies a rule called **path induction** (This is the type-theoretic Leibniz's law). We have always an element $\mathrm{refl}_x : x =_X x$.

Families of types

Families of types

If X is a type, we can index by X a family $\mathcal{P}(-)$ of types. Hence if $x : X$, $\mathcal{P}(x)$ is a type. We will write $\mathcal{P} : X \to \mathcal{U}$. $X \to \mathcal{U}$ is the type of families indexed by X : $(\mathcal{P}(x))_{x:X}$.

Families of types

If X is a type, we can index by X a family $\mathcal{P}(-)$ of types. Hence if $x : X$, $\mathcal{P}(x)$ is a type. We will write $\mathcal{P} : X \to \mathcal{U}$. $X \to \mathcal{U}$ is the type of families indexed by X : $(\mathcal{P}(x))_{x:X}$.
(We can think of $\mathcal{U}$ as an universe, where each type X is an element of type $\mathcal{U}$, $X : \mathcal{U}$)

Families of types

If X is a type, we can index by X a family $\mathcal{P}(-)$ of types. Hence if $x : X$, $\mathcal{P}(x)$ is a type. We will write $\mathcal{P} : X \to \mathcal{U}$. $X \to \mathcal{U}$ is the type of families indexed by X : $(\mathcal{P}(x))_{x:X}$.
(We can think of $\mathcal{U}$ as an universe, where each type X is an element of type $\mathcal{U}$, $X : \mathcal{U}$)
With families of types, we can build two new types.

Dependant product

Dependant product

If $\mathcal{P} : X \to \mathcal{U}$ is a family of types indexed by X, we can form the **dependant product** :

$$\prod_{x:X} \mathcal{P}(x)$$

Dependant product

If $\mathcal{P} : X \to \mathcal{U}$ is a family of types indexed by X, we can form the **dependant product** :

$$\prod_{x:X} \mathcal{P}(x)$$

An element $f : \prod_{x:X} \mathcal{P}(x)$ can be evaluated in $x : X$, $f(x) : \mathcal{P}(x)$.

Dependant product

If $\mathcal{P} : X \to \mathcal{U}$ is a family of types indexed by X, we can form the **dependant product** :

$$\prod_{x:X} \mathcal{P}(x)$$

An element $f : \prod_{x:X} \mathcal{P}(x)$ can be evaluated in $x : X$, $f(x) : \mathcal{P}(x)$. We can also see an element $f : \prod_{x:X} \mathcal{P}(x)$ as a family $(f_x : \mathcal{P}(x))_{x:X}$ indexed by X.

Dependant sum

Dependant sum

If $\mathcal{P} : X \to \mathcal{U}$ is a family of types indexed by X, we can form the **dependant sum** :

$$\sum_{x:X} \mathcal{P}(x)$$

Dependant sum

If $\mathcal{P} : X \to \mathcal{U}$ is a family of types indexed by X, we can form the **dependant sum** :

$$\sum_{x:X} \mathcal{P}(x)$$

An element $w : \sum_{x:X} \mathcal{P}(x)$ is the data of $x : X$ and $a : \mathcal{P}(x)$, $w \equiv (x, a)$.

Quick comment : proposition as type

Quick comment : proposition as type

Each proposition can be thought as a type. If A is a proposition corresponding to a type X the existence of an element x of type X is a witness of the truth value of A (Being able to write $x : X$ means that A is true, x is a **proof** of A).

Quick comment : proposition as type

Each proposition can be thought as a type. If A is a proposition corresponding to a type X the existence of an element x of type X is a witness of the truth value of A (Being able to write $x : X$ means that A is true, x is a **proof** of A).
Here is a list of basic constructions of types and how they relate to propositions :

Quick comment : proposition as type

Each proposition can be thought as a type. If A is a proposition corresponding to a type X the existence of an element x of type X is a witness of the truth value of A (Being able to write $x : X$ means that A is true, x is a **proof** of A).
Here is a list of basic constructions of types and how they relate to propositions :

$$\begin{aligned} \top &\leftrightarrow \mathbf{1} \\ \bot &\leftrightarrow \mathbf{0} \\ A \wedge B &\leftrightarrow X \times Y \\ A \vee B &\leftrightarrow X + Y \\ A \Rightarrow B &\leftrightarrow X \to Y \\ \forall x, \mathcal{P}(x) &\leftrightarrow \Pi_{x:X} \mathcal{P}(x) \\ \exists x, \mathcal{P}(x) &\leftrightarrow \Sigma_{x:X} \mathcal{P}(x) \end{aligned}$$

Homotopy Type Theory

Types as Homotopy types/Spaces

Types as Homotopy types/Spaces

In HoTT (Homotopy Type Theory), types are thought as homotopy types, these are topological spaces **up to homotopy equivalences**.

Types as Homotopy types/Spaces

In HoTT (Homotopy Type Theory), types are thought as homotopy types, these are topological spaces **up to homotopy equivalences**.

- X is a **space**.
- $x : X$ are **points** of X.
- $p : x =_X y$ are **paths** between points.

Example : Dependant Sum as a Fibration

Example : Dependant Sum as a Fibration

Feature : Path lifting

Feature : Path lifting

Homotopies between functions

Homotopies between functions

Let $f, g : X \to Y$, we define $f \sim g :\equiv \prod_{x:X} (f(x) =_Y g(x))$.

Homotopies between functions

Let $f, g : X \to Y$, we define $f \sim g :\equiv \prod_{x:X} (f(x) =_Y g(x))$. An element of that type is the data for each $x : X$, of a path $p_x : f(x) =_Y g(x)$.

Homotopies between functions

Let $f, g : X \to Y$, we define $f \sim g :\equiv \prod_{x:X} (f(x) =_Y g(x))$.
An element of that type is the data for each $x : X$, of a path $p_x : f(x) =_Y g(x)$.
Because of the nature of HoTT, p_x depends automatically "continuously" of x.

Equivalences of type

Equivalences of type

Let $f : X \to Y$ be a function, we want to state that f is an equivalence. Naturally one way to say that f is an equivalence is if f has a quasi-inverse, that means that we can build $g : Y \to X$, and two homotopies $H : f \circ g \sim \mathrm{id}_Y$, $H' : g \circ f \sim \mathrm{id}_X$, but it is ill behaved due to higher homotopies reasons (f does not have an unique (up to homotopy) quasi-inverse).

Equivalences of type

Let $f : X \to Y$ be a function, we want to state that f is an equivalence. Naturally one way to say that f is an equivalence is if f has a quasi-inverse, that means that we can build $g : Y \to X$, and two homotopies $H : f \circ g \sim \mathrm{id}_Y$, $H' : g \circ f \sim \mathrm{id}_X$, but it is ill behaved due to higher homotopies reasons (f does not have an unique (up to homotopy) quasi-inverse).
f is an equivalence if it is provided by a left inverse $h : Y \to X$ with a homotopy $H : h \circ f \sim \mathrm{id}_X$ and a right inverse $h' : Y \to X$ with a homotopy $H' : f \circ h \sim \mathrm{id}_Y$. The type $X \simeq Y$ is defined by :

Equivalences of type

Let $f : X \to Y$ be a function, we want to state that f is an equivalence. Naturally one way to say that f is an equivalence is if f has a quasi-inverse, that means that we can build $g : Y \to X$, and two homotopies $H : f \circ g \sim \mathrm{id}_Y$, $H' : g \circ f \sim \mathrm{id}_X$, but it is ill behaved due to higher homotopies reasons (f does not have an unique (up to homotopy) quasi-inverse).
f is an equivalence if it is provided by a left inverse $h : Y \to X$ with a homotopy $H : h \circ f \sim \mathrm{id}_X$ and a right inverse $h' : Y \to X$ with a homotopy $H' : f \circ h \sim \mathrm{id}_Y$. The type $X \simeq Y$ is defined by :

$$X \simeq Y :\equiv \sum_{f:X\to Y} \left(\left(\sum_{h:Y\to X} h \circ f \sim \mathrm{id}_X \right) \times \left(\sum_{h':Y\to X} f \circ h \sim \mathrm{id}_Y \right) \right)$$

Making Type Theory "homotopic" : The Univalence axiom

Making Type Theory "homotopic" : The Univalence axiom

(Homotopy) pullback

(Homotopy) pullback

(Homotopy) pushout

(Homotopy) pushout