Univalent foundations

an introduction

Benedikt Ahrens

6th Workshop on Formal Topology
University of Birmingham, UK
2019-04-08

What are univalent foundations?

"Univalent foundations" may refer to:

1. any foundation of mathematics satisfying the equivalence principle
2. a particular foundation of mathematics (which was designed to satisfy the equivalence principle)

- For the purpose of this talk, we adopt meaning 2
- An explanation of the equivalence principle will be given later

At the origin of univalent foundations is Vladimir Voevodsky. Voevodsky died on 30 September 2017.

What is a foundation of mathematics?

According to Voevodsky, a foundation of mathematics is specified by three things:

1. Syntax for mathematical objects
2. Notion of proposition and proof
3. Interpretation of the syntax into the world of mathematical objects

By "univalent foundations" I refer to the following foundation:

1. Univalent type theory
2. Logic of h-propositions
3. Interpretation of types as Kan complexes

Outline

Outline

Voevodsky's interest in computer-checked proofs

> *PS I am thinking again about the applications of computers to pure math. Do you know of anyone working in this area? I mean mostly some kind of a computer language to describe mathematical structures, their properties and proofs in such a way that ultimately one may have mathematical knowledge archived and logically verified in a fixed format.*

Email to Dan Grayson, Sept 2002

Voevodsky's ideas about computer-checked proofs I

Let us start with what would be a perfect system of such sort [. . .].

Ideally one wants a math oracle. A user inputs a type expression and the system either returns a term of this type or says that it has no terms. The type expression may be "of level 0" i.e. it can correspond to a statement in which case a term is a proof and the absence of terms means that the statement is not provable. It may be "of level 1" e.g. it might be the type of solutions of an equation. In that case the system produces a solution or says that there are non. It may be "of level 2" e.g. It might be the type of all solvable groups of order 35555. In that case the system produces an example of such a group etc.

Voevodsky's ideas about computer-checked proofs II

> *A realistic approximation [. . .] may look as follows. Imagine a web-based system with a lot of users (both "creators" and "consumers") and a very large "database". Originally the database is empty (or, rather, contains only the primitive "knowledge" a-la axioms). User A (say in Princeton) inputs a type expression and builds up a term of this type either in one step (just types it in) or in many steps using the standard proof-assistant capabilities of the system. Both the original and all the intermediate type expressions which occur in the process are filed (in the real time) in the database. Enter user B (somewhere in Brazil), who inputs another type expression and begins the process of constructing a term of his type.*

Email to Grayson, Sept 2006

Voevodsky learning how to use the proof assistant Coq

> *I am thinking a lot these days about foundations of math and automated proof verification. My old idea about a "univalent" homotopy theoretical models of Martin-Lof type systems survived the verification stage an I am in the process of writing things up.*
>
> *I also took a course at the Princeton CS department which was for most part about Coq and was very impressed both by how much can be proved in it in a reasonable time and by how many young students attended (45, 35 undergrad + 10 grad!).*

Email to Grayson, Dec 2009

The Foundations library of computer-checked mathematics

In Feb 2010, Voevodsky started writing the Coq library *Foundations*, making precise his ideas conceived during three years and collected in *A very short note on homotopy λ-calculus*.

```
Fixpoint isofhlevel (n:nat) (X:UU): UU:=
match n with
O => iscontr X |
S m => forall x:X, forall x':X, (isofhlevel m (paths _ x x'))
end.


Theorem hlevelretract (n:nat)(X:UU)(Y:UU)(p:X -> Y)(s:Y ->X)(eps: forall y:Y, paths _  (p (s y)) y): (isofhlevel n X) -> (isofhlevel n Y).
Proof. intro. induction n. intros. apply (contrl1' _ _ p s eps X0).
intros. unfold isofhlevel. intros. unfold isofhlevel in X0. assert (is: isofhlevel n (paths _ (s x) (s x'))).  apply X0.
set (s':= maponpaths _ _ s x x'). set (p':= pathssec2 _ _ s p eps x x'). set (eps':= pathssec3 _ _ s p eps x x'). apply (IHn _ _ p' s' eps' is). Defined.

Corollary  hlevelweqf (n:nat)(X:UU)(Y:UU)(f:X -> Y)(is: isweq _ _ f): (isofhlevel n X) -> (isofhlevel n Y).
Proof. intros.  apply (hlevelretract n _ _ f (invmap _ _ f is) (weqfg _ _ f is)). assumption. Defined.

Corollary  hlevelweqb (n:nat)(X:UU)(Y:UU)(f:X -> Y)(is: isweq _ _ f): (isofhlevel n Y) -> (isofhlevel n X).
Proof. intros.  apply (hlevelretract n _ _ (invmap _ _ f is) f (weqgf _ _ f is)). assumption. Defined.


Definition isaprop (X:UU): UU := isofhlevel (S O) X.
```

This library, and other libraries built on top of *Foundations*, were later combined into the UniMath (Univalent Mathematics) library, which continues to be developed.

On univalent foundations

- Mathematics is the study of structures on sets and their higher analogs.
- Set-theoretic mathematics constitutes a subset of the mathematics that can be expressed in univalent foundations.
- Classical mathematics is a subset of univalent mathematics consisting of the results that require LEM and/or AC among their assumptions.

see Voevodsky, Talk at HLF, Sept 2016

Outline

Overview of type theory

(Dependent) Type theory

- Is a (functional programming) language of types and terms
- Has infrastructure to write mathematical statements and proofs

Write $a : A$ to say that term a has type A (e.g., $1 : \mathsf{Nat}$)

Writing well-typed programs

In type theory, both the activities of

- implementing an algorithm
- proving a mathematical statement

are done by writing well-typed programs.

Dependent types and functions

Examples of dependent types

- $\mathsf{Vect}(A, n)$ — type of vectors of length $n : \mathsf{Nat}$ of elements in type A
- $\mathsf{GrpStr}(X)$ — type of group structures on a set X

Examples of dependent functions

$$\mathsf{zero} : \prod_{m:\mathsf{Nat}} \mathsf{Vect}(\mathsf{Nat}, m)$$

$$\mathsf{tail} : \prod_{n:\mathsf{Nat}} \mathsf{Vect}(A, 1+n) \to \mathsf{Vect}(A, n)$$

$$\mathsf{GrpStrTransfer} : \prod_{X,Y:\mathsf{Set}} (X \cong Y) \times \mathsf{GrpStr}(X) \to \mathsf{GrpStr}(Y)$$

This section

In this section, I give a brief overview of type theory:

- judgements
- inference rules
- derivations
- overview of the type constructions available in type theory

Syntax of type theory

Fundamental: **judgment**

$$\text{context} \vdash \text{conclusion}$$

Contexts & judgments

Γ	sequence of variable declarations $(x_1 : A_1), (x_2 : A_2(x_1)), \ldots, (x_n : A_n(\vec{x}_i))$
$\Gamma \vdash A$	A is well–formed **type** in context Γ
$\Gamma \vdash a : A$	**term** a is well-formed and of type A
$\Gamma \vdash A \equiv B$	types A and B are **convertible**
$\Gamma \vdash a \equiv b : A$	a is convertible to b in type A

$$(x : \mathsf{Nat}), (f : \mathsf{Nat} \to \mathsf{Bool}) \vdash f@x : \mathsf{Bool}$$

Type dependency

In particular: dependent type B over A

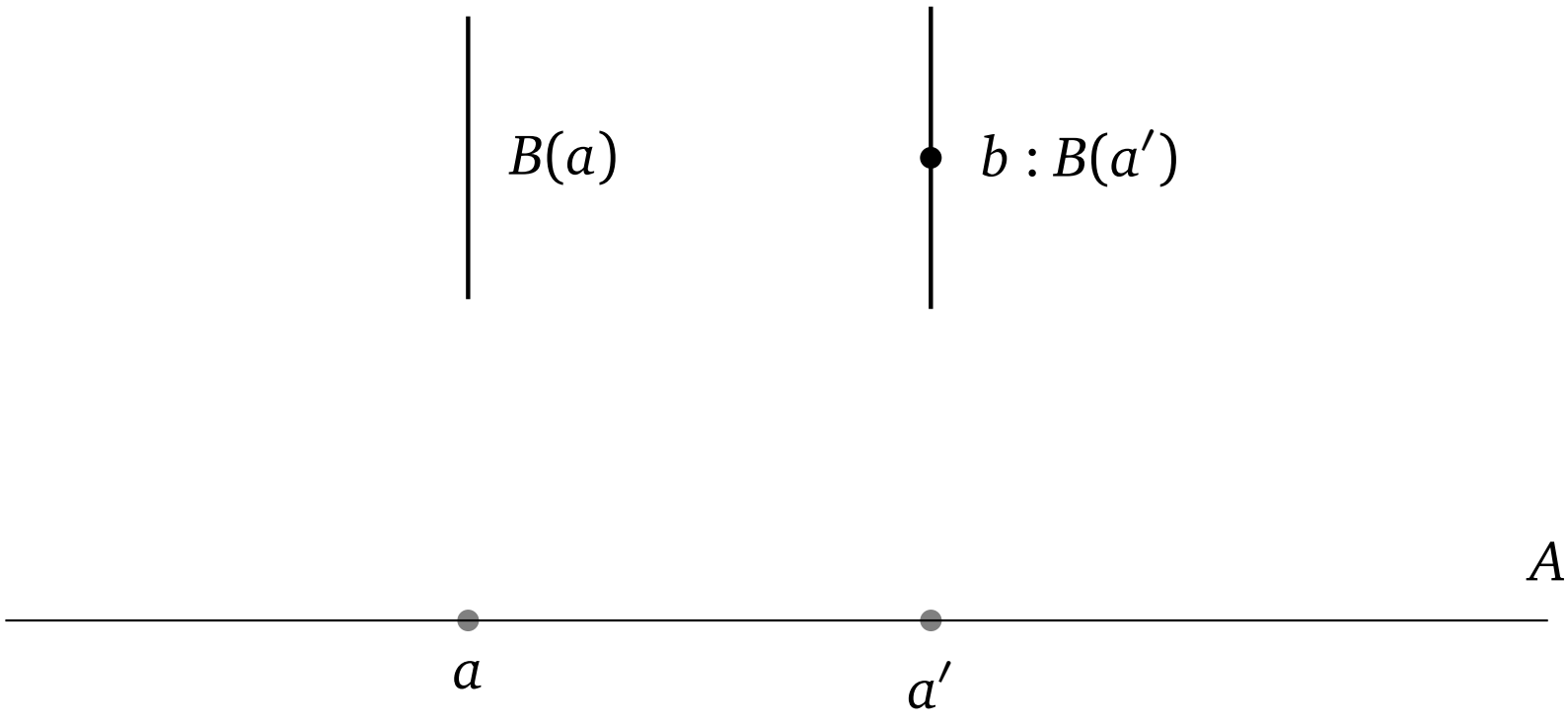
$$x : A \;\vdash\; B(x)$$

"family B of types indexed by A"

- A type can depend on several variables
- Example: type of vectors of length n with elements in type A

$$(A : \mathsf{Type}), (n : \mathsf{Nat}) \;\vdash\; \mathsf{Vect}(A, n)$$

Dependent types in pictures



Inference rules and derivations

An inference rule

is an implication of judgments,

$$\frac{J_1 \qquad J_2 \qquad \dots}{J}$$

e.g.,

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f@a : B} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash a : B}$$

Inference rules and derivations

An **inference rule**

is an implication of judgments,

$$\frac{J_1 \qquad J_2 \qquad \ldots}{J}$$

e.g.,

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f@a : B} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash a : B}$$

- Abbreviate rules to, e.g., " If $a : A$ and $A \equiv B$, then $a : B$".

Derivations

Example:

$$\dfrac{\dfrac{\dfrac{}{(f : A \to B) \vdash f : A \to B}}{(f : A \to B), (x : A) \vdash f : A \to B}}{(x : A), (f : A \to B) \vdash f : A \to B} \qquad \dfrac{\dfrac{}{(x : A) \vdash x : A}}{(x : A), (f : A \to B) \vdash x : A}$$

$$(x : A), (f : A \to B) \vdash f@x : B$$

Derivations

Example:

$$\dfrac{\dfrac{\dfrac{}{(f : A \to B) \;\vdash\; f : A \to B}}{(f : A \to B), (x : A) \;\vdash\; f : A \to B}}{(x : A), (f : A \to B) \;\vdash\; f : A \to B} \qquad \dfrac{\dfrac{}{(x : A) \;\vdash\; x : A}}{(x : A), (f : A \to B) \;\vdash\; x : A}$$

$$\dfrac{(x : A), (f : A \to B) \;\vdash\; f : A \to B \qquad (x : A), (f : A \to B) \;\vdash\; x : A}{(x : A), (f : A \to B) \;\vdash\; f@x : B}$$

Derivation

Term a is well-typed of type A in a context Γ if there is a derivation of the judgement $\Gamma \;\vdash\; a : A$

Interlude: Interpreting types as sets?

- Can interpret types and terms as sets
- $a : A$ is interpreted as $[a] \in [A]$

Differences between $a : A$ and $a \in A$

- Judgement $\Gamma \vdash a : A$ is **not** a statement that can be proved or disproved within type theory
- Term a does not exist independently of its type A
- A term has exactly one type (up to $\equiv$)

Declaring types & terms

The inference rules of type theory consist of

infrastructural rules: substitution, weakening,. . .

logical rules: to build new types and terms

Declaring types & terms

The inference rules of type theory consist of

infrastructural rules: substitution, weakening,. . .

logical rules: to build new types and terms

The logical rules mostly come in groups of 4 rules:

Formation	way to construct a type
Introduction	way(s) to construct **canonical terms** of that type
Elimination	ways to use a term of the introduced type to construct other terms
Computation	what happens when one does Introduction followed by Elimination

Universes

Universes

- There is a type Type. Its elements are types, written $A : \mathsf{Type}$.
- Using dependent function types (cf. later), the dependent type $x : A \vdash B$ can be considered as a function

$$\lambda x.B : A \to \mathsf{Type}$$

Universes

Universes

- There is a type Type. Its elements are types, written $A : \mathsf{Type}$.
- Using dependent function types (cf. later), the dependent type $x : A \vdash B$ can be considered as a function

$$\lambda x.B : A \to \mathsf{Type}$$

- Actually, hierarchy $(\mathsf{Type}_i)_{i \in I}$ to avoid paradoxes.
- But we ignore this here, and only write Type.

$$(A : \mathsf{Type}), (n : \mathsf{Nat}) \vdash \mathsf{Vect}(A, n) : \mathsf{Type}$$

The singleton type

Formation	way to construct a type
Introduction	way(s) to construct **canonical terms** of that type
Elimination	ways to use a term of the introduced type to construct other terms
Computation	what happens when one does Introduction followed by Elimination

Singleton type

Formation	1 is a type
Introduction	$\mathsf{t} : 1$
Elimination	If $x : 1$ and C is a type and $c : C$, then $\mathsf{rec}_1(C, c, x) : C$
Computation	$\mathsf{rec}_1(C, c, \mathsf{t}) \equiv c$

The empty type

Formation	way to construct a type
Introduction	way(s) to construct **canonical terms** of that type
Elimination	ways to use a term of the introduced type to construct other terms
Computation	what happens when one does Introduction followed by Elimination

Empty type

Formation	$\mathbf{0}$ is a type
Introduction	
Elimination	If $x : \mathbf{0}$ and C is a type, then $\mathrm{rec}_{\mathbf{0}}(C, x) : C$
Computation	

Exercises

- Define the type Bool of booleans.
- Define the type Nat of natural numbers.

The type of dependent pairs $\sum_{x:A} B$

Formation If $x : A \vdash B$, then $\sum_{x:A} B(x)$ is a type

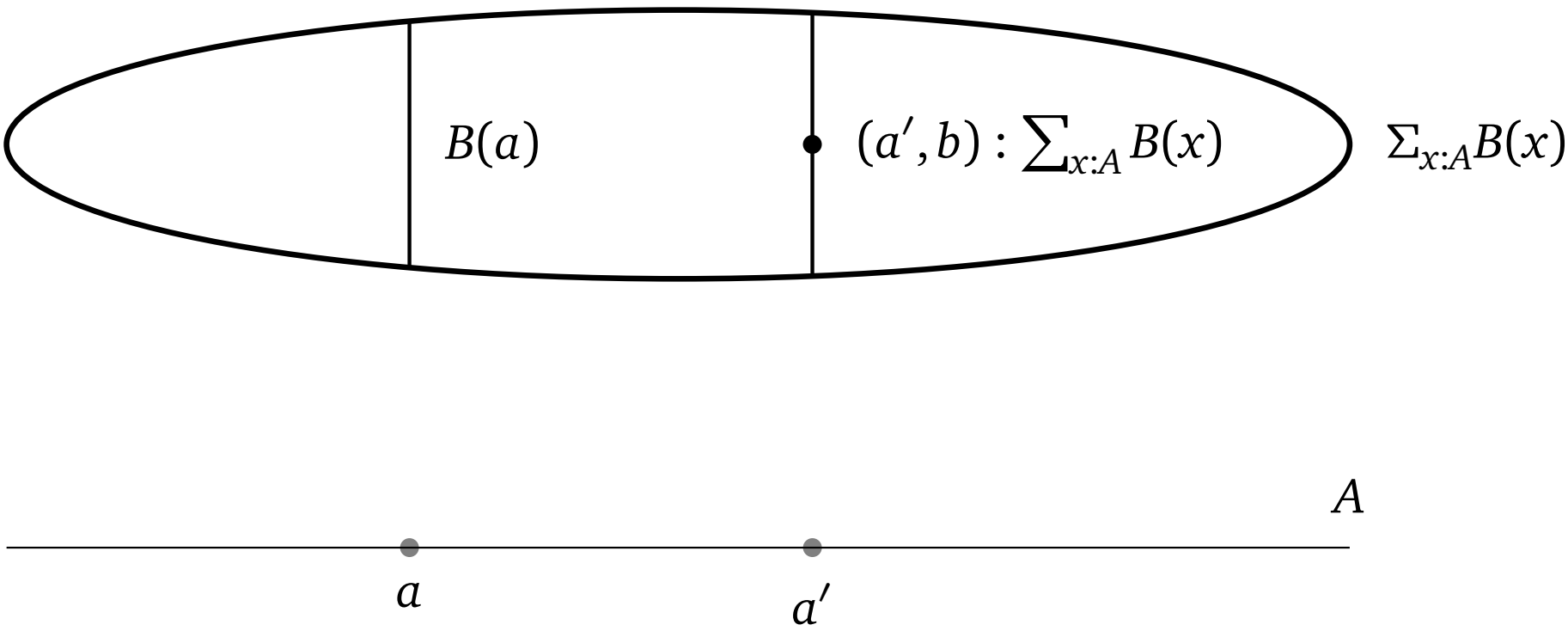
Introduction If $a : A$ and $b : B(a)$, then $\mathsf{pair}(a, b) : \sum_{x:A} B(x)$

Elimination If $t : \sum_{x:A} B$, then $\mathsf{fst}(t) : A$ and $\mathsf{snd}(t) : B(\mathsf{fst}(t))$

Computation $\mathsf{fst}(\mathsf{pair}(a, b)) \equiv a$ and $\mathsf{snd}(\mathsf{pair}(a, b)) \equiv b$

- Special case: $A \times B$

Σ-type in pictures



The type of dependent functions $\prod_{x:A} B$

Formation If $x : A \vdash B$, then $\prod_{x:A} B$ is a type

Introduction If $x : A \vdash b : B$, then $\lambda(x : A).b : \prod_{x:A} B$

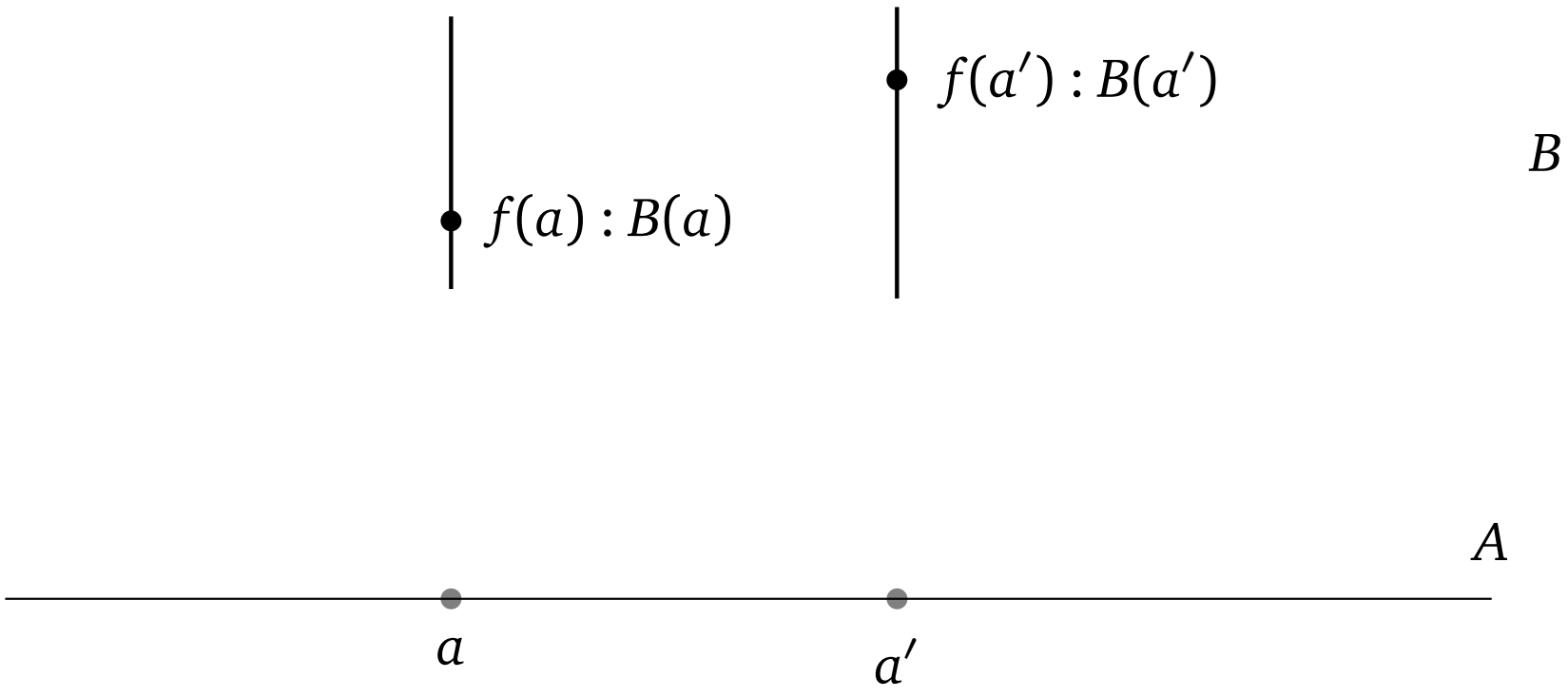
Elimination If $f : \prod_{x:A} B$ and $a : A$, then $f@a : B[x := a]$

Computation $(\lambda(x : A).b)@a \equiv b[x := a]$

- Special case: $A \to B$

A dependent function in pictures

$f : \prod_{x:A} B(x)$



The identity type

Formation If $a : A$ and $b : A$, then $\mathsf{Id}_A(a, b)$ is a type

Introduction If $a : A$, then $\mathsf{refl}(a) : \mathsf{Id}_A(a, a)$

Elimination If

$$(x, y : A), (p : \mathsf{Id}_A(x, y)) \vdash C(x, y, p)$$

and

$$(x : A) \vdash t(x) : C(x, x, \mathsf{refl}(x))$$

then

$$(x, y : A), (p : \mathsf{Id}_A(x, y)) \vdash \mathsf{ind}_{\mathsf{Id}}(x.t(x); x, y, p) : C(x, y, p)$$

Computation $\mathsf{ind}_{\mathsf{Id}}(x.t(x); x, x, \mathsf{refl}(x)) \equiv t(x)$

Overview of types in Martin-Löf type theory

Type former	Notation	(special case)
Inhabitant	$a : A$	
Dependent type	$x : A \vdash B(x)$	
Sigma type	$\sum_{x:A} B(x)$	$A \times B$
Product type	$\prod_{x:A} B(x)$	$A \to B$
Coproduct type	$A + B$	
Identity type	$\mathsf{Id}_A(a, b)$	
Universe	Type	
Base types	Nat, Bool, $\mathsf{1}$, $\mathsf{0}$	

Some Identity terms that can be defined

$$\mathsf{refl} : \prod_{x:A} \mathsf{Id}_A(x,x)$$

$$\mathsf{sym} : \prod_{x,y:A} \mathsf{Id}(x,y) \to \mathsf{Id}(y,x)$$

$$\mathsf{trans} : \prod_{x,y,z:A} \mathsf{Id}(x,y) \times \mathsf{Id}(y,z) \to \mathsf{Id}(x,z)$$

$$\mathsf{subst} : \prod_{x,y:A} \mathsf{Id}(x,y) \to B(x) \to B(y)$$

Some Identity terms that can be defined

$$\mathsf{refl} : \prod_{x:A} \mathsf{Id}_A(x,x)$$

$$\mathsf{sym} : \prod_{x,y:A} \mathsf{Id}(x,y) \to \mathsf{Id}(y,x)$$

$$\mathsf{trans} : \prod_{x,y,z:A} \mathsf{Id}(x,y) \times \mathsf{Id}(y,z) \to \mathsf{Id}(x,z)$$

$$\mathsf{subst} : \prod_{x,y:A} \mathsf{Id}(x,y) \to B(x) \to B(y)$$

$\rightsquigarrow$ motivates interpretation of $\mathsf{Id}_A(x,y)$ as equality $x = y$

Outline

Interpreting types as sets

Syntax	Set interpretation
A	set A
$a : A$	$a \in A$
$A \times B$	cartesian product
$A \to B$	set of functions $A \to B$
$A + B$	disjoint union $A \amalg B$
$x : A \vdash B(x)$	family B of sets indexed by A
$\sum_{x:A} B(x)$	disjoint union $\amalg_{x:A} B(x)$
$\prod_{x:A} B(x)$	product of sets
$\mathsf{Id}_A(a, b)$	equality $a = b$

Equalities between equalities

- Identity type former can be iterated:

$$A : \mathsf{Type},\ \ x,y : A,\ \ p,q : \mathsf{Id}(x,y) \ \vdash\ \ \mathsf{Id}_{\mathsf{Id}_A(x,y)}(p,q)$$

- What are its terms?

Question

Can one construct a term of type

$$\mathsf{UIP} : \prod_{(x,y:A)} \prod_{(p,q:\mathsf{Id}(x,y))} \mathsf{Id}(p,q)$$

Answer

No, cf. Hofmann & Streicher's groupoid model of type theory.

Axiom K

However, UIP is independent of type theory; the set model of type theory satisfies

Axiom K

$$\mathsf{axiomK} : \prod_{(x:A)} \prod_{(p:\mathsf{Id}(x,x))} \mathsf{Id}(p, \mathsf{refl}(x))$$

One can then show

Uniqueness of Identity Proofs

$$\mathsf{UIP} : \prod_{(x,y:A)} \prod_{(p,q:\mathsf{Id}(x,y))} \mathsf{Id}(p, q)$$

A different interpretation of identities

- A different interpretation of identities as **structure rather than property** is given in Hofmann & Streicher's groupoid model:

  $\mathsf{Id}_A(x,y)$ interpreted as (invertible) morphisms $x \to y$

  $\mathsf{Id}_{\mathsf{Id}_A(x,y)}(f,g)$ interpreted as equalities $f = g$ for $x \overset{f,g}{\rightrightarrows} y$

- Voevodsky's model in simplicial sets takes this idea a step further.

A different interpretation of identities

- A different interpretation of identities as **structure rather than property** is given in Hofmann & Streicher's groupoid model:

  $\mathsf{Id}_A(x,y)$ interpreted as (invertible) morphisms $x \to y$

  $\mathsf{Id}_{\mathsf{Id}_A(x,y)}(f,g)$ interpreted as equalities $f = g$ for $x \overset{f,g}{\rightrightarrows} y$

- Voevodsky's model in simplicial sets takes this idea ω steps further.

Outline

Interpretation of identities as paths

Inhabitants of $\mathsf{Id}(x,y)$ behave like equality in many ways

- reflexivity, symmetry, transitivity
- subst

Inhabitants of $\mathsf{Id}(x,y)$ behave **un**like equality

- Can iterate identity type
- Cannot show that any two identities are identical

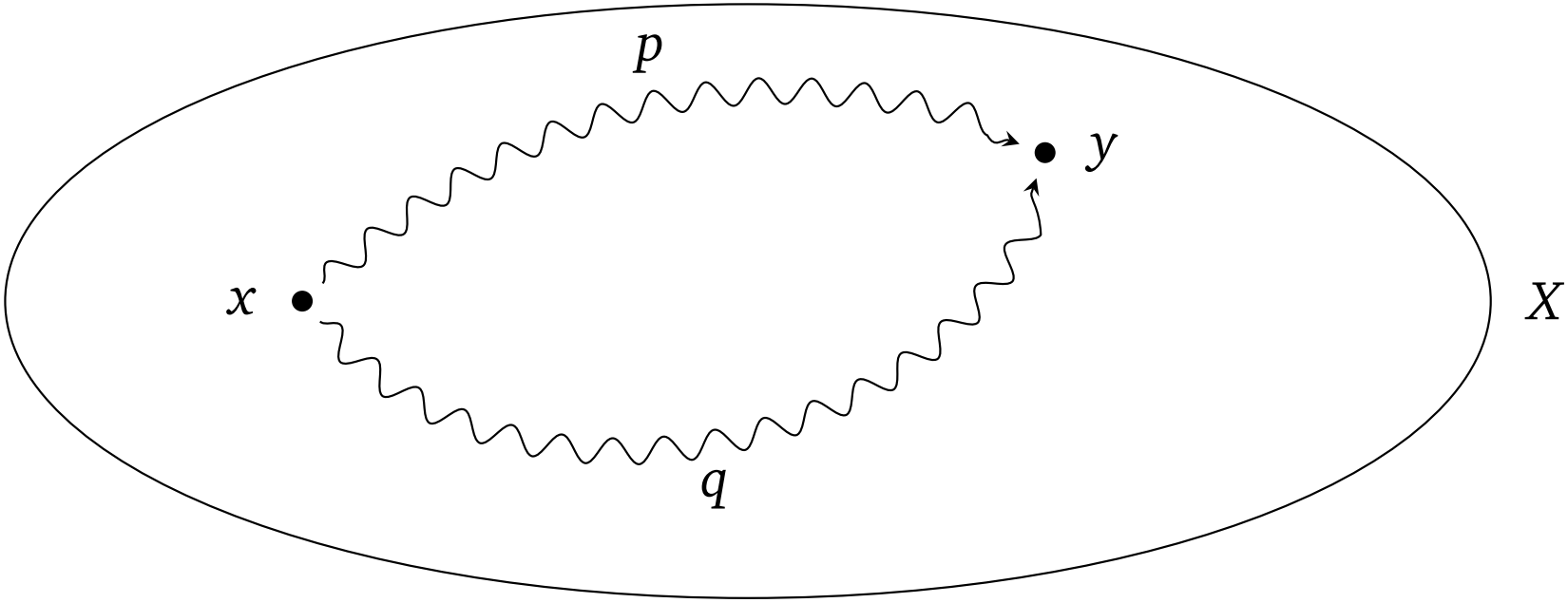
Lack of uniqueness motivates interpretation of elements of $\mathsf{Id}_X(x,y)$ as **paths from** x **to** y **in** X.
A new notation, to reflect the new interpretation:

$$x \rightsquigarrow_X y$$

Identities interpreted as paths in a space

$x, y : X$

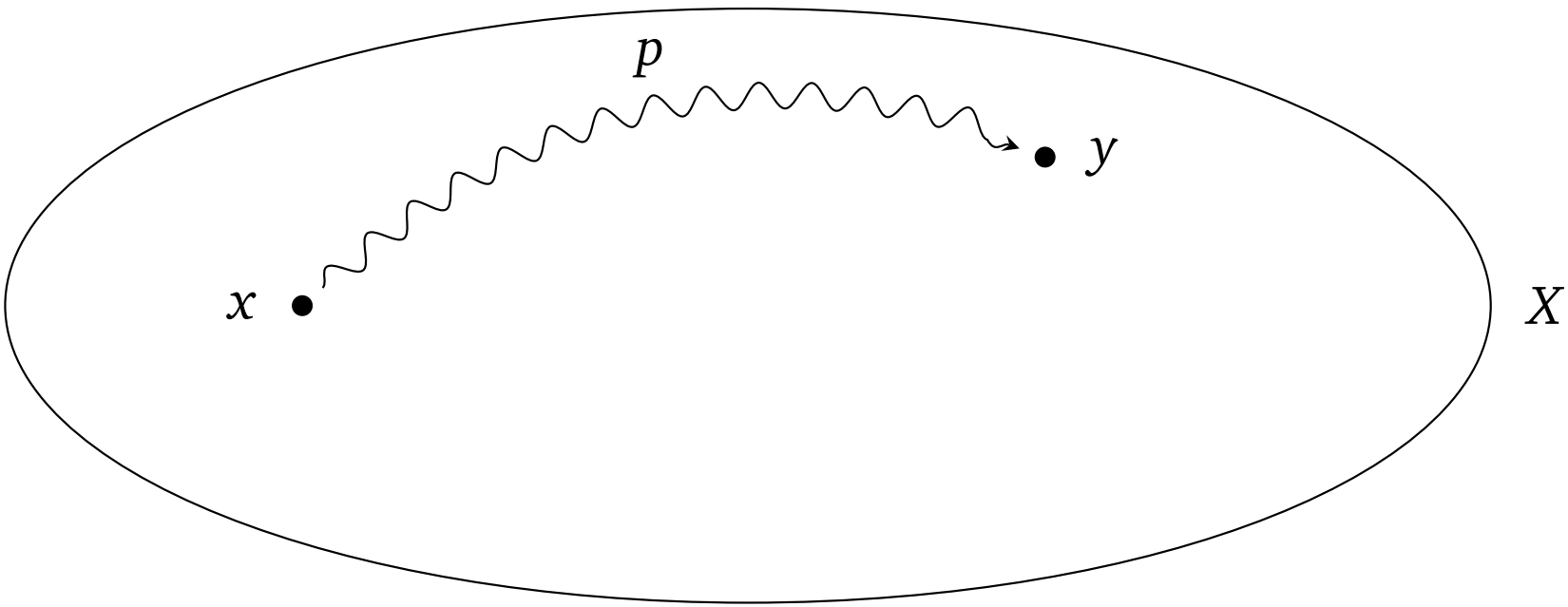
$p, q : x \rightsquigarrow_X y$



'Reflexivity' interpreted as the constant path on a point x.

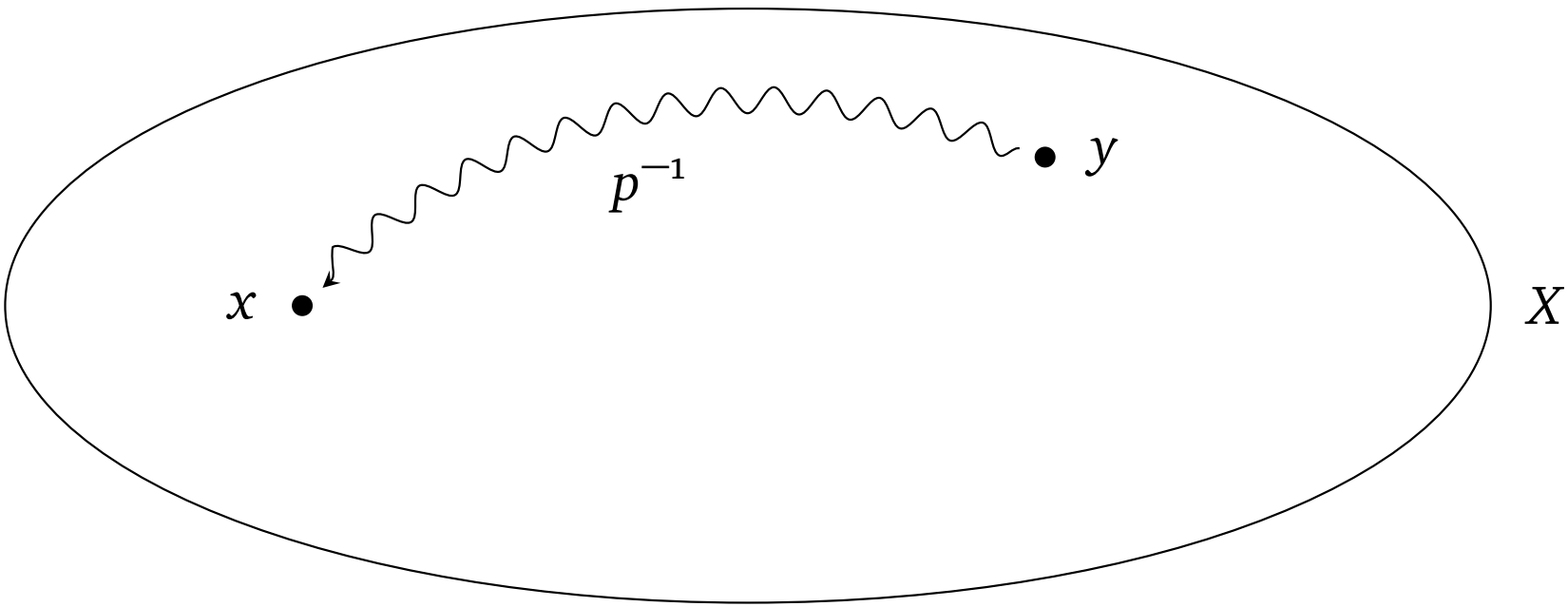
Operations on paths

- $p : x \rightsquigarrow y$



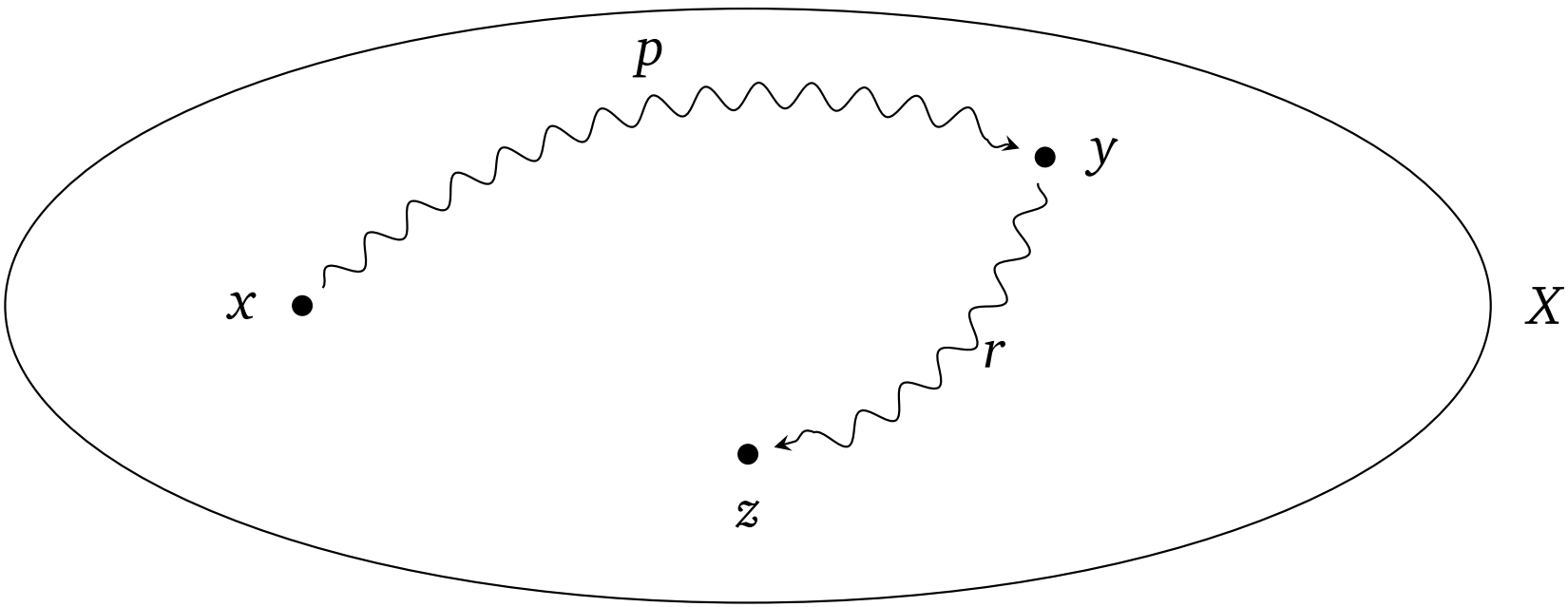
Operations on paths

- $p : x \rightsquigarrow y$
- $p^{-1} : y \rightsquigarrow x$



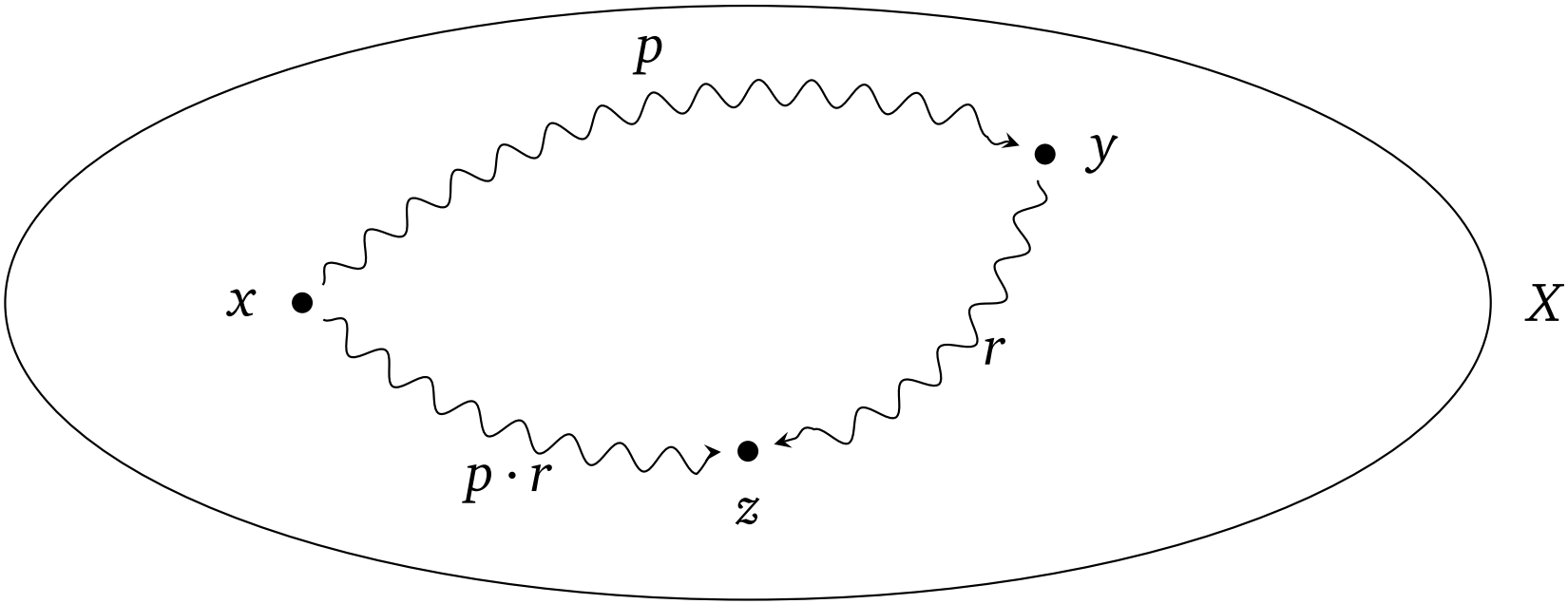
Operations on paths

- $p : x \rightsquigarrow y$
- $p^{-1} : y \rightsquigarrow x$
- $r : y \rightsquigarrow z$



Operations on paths

- $p : x \rightsquigarrow y$
- $p^{-1} : y \rightsquigarrow x$
- $r : y \rightsquigarrow z$
- $p \cdot r : x \rightsquigarrow z$



More operations on paths

Transport

$$\mathsf{transport} : \prod_{x,y:A} x \rightsquigarrow y \to \prod_{B:A\to\mathsf{Type}} \big(B(x) \leftrightarrow B(y)\big)$$

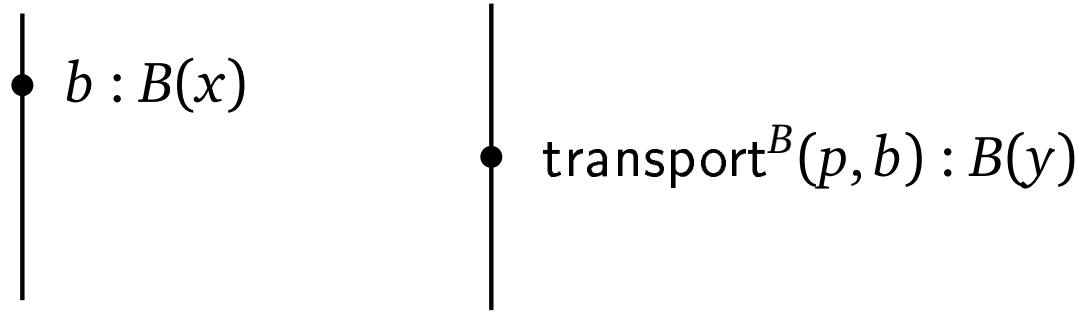
Congruence

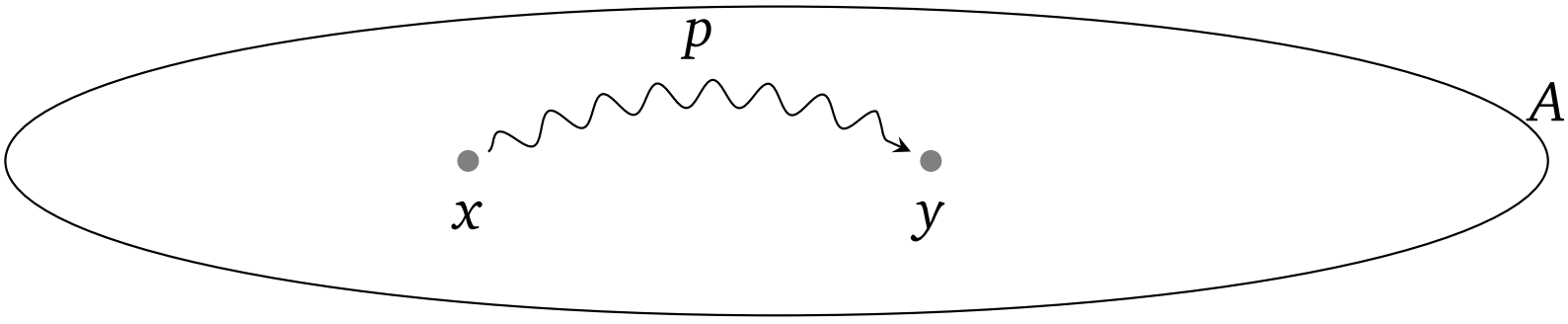
Given $f : A \to B$, have

$$\mathsf{ld}(f) : \prod_{x,y:A} x \rightsquigarrow y \to f(x) \rightsquigarrow f(y)$$

Transport in pictures

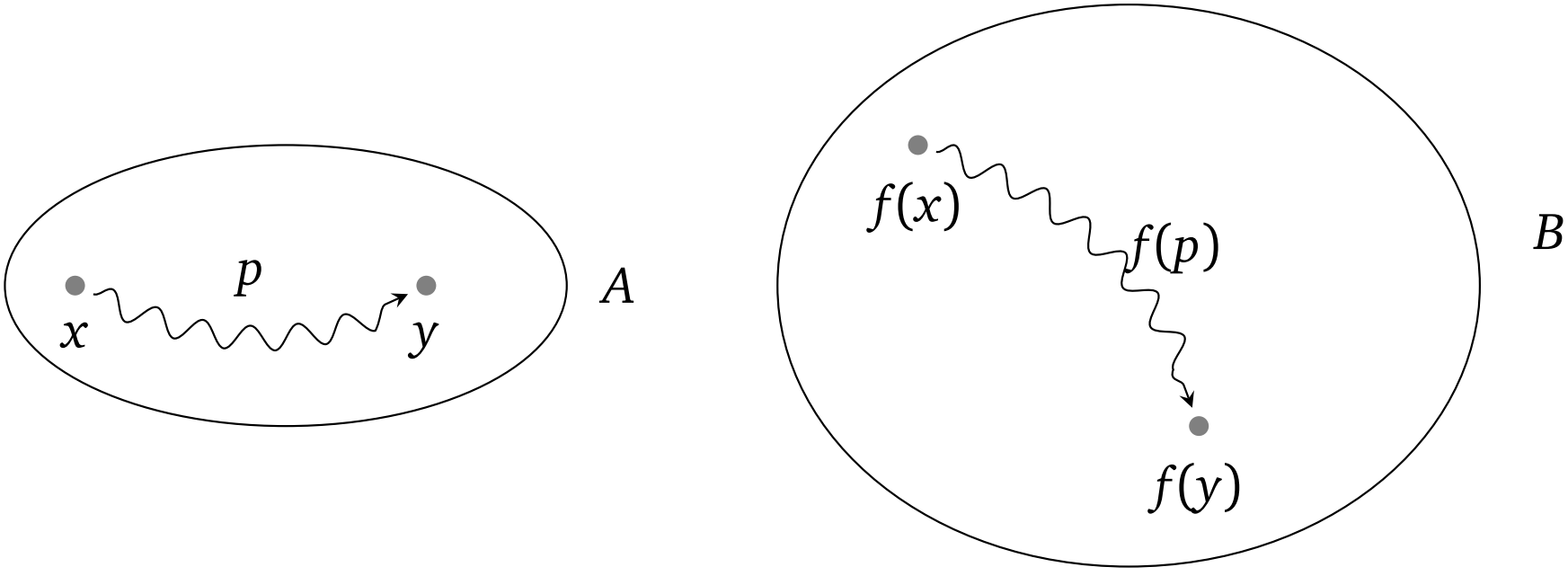
$\mathsf{transport}^B : x \rightsquigarrow y \to B(x) \to B(y)$





Functions map paths, not just points

$f : A \to B$



Exercise

Given $f : A \to B$, construct a term of type

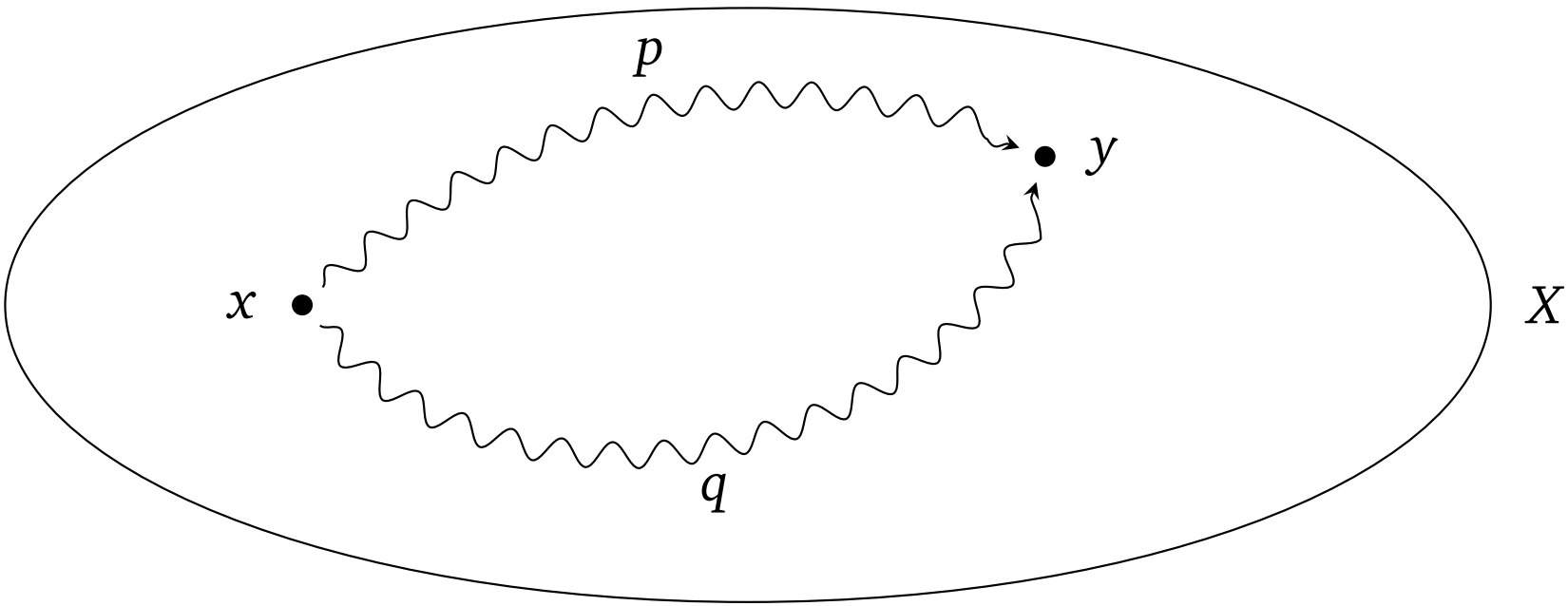
$$\prod_{x,y:A} x \rightsquigarrow_A y \to f(x) \rightsquigarrow_B f(y)$$

Paths between paths

What is a path

$$h : p \rightsquigarrow_{x \rightsquigarrow y} q$$

between paths?

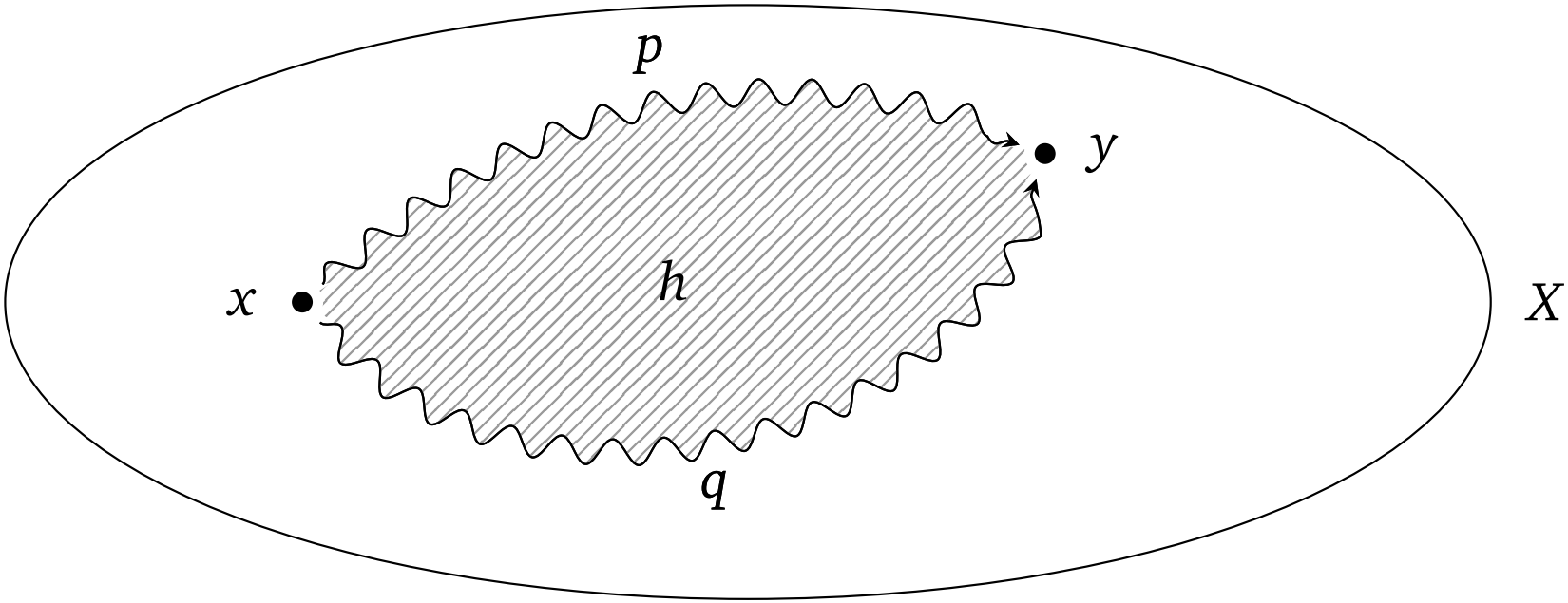


Paths between paths

What is a path

$$h : p \rightsquigarrow_{x \rightsquigarrow y} q$$

between paths?



Laws satisfied by path operations

Can construct homotopies

- $(p \cdot q) \cdot r \rightsquigarrow p \cdot (q \cdot r)$
- $p \cdot 1_y \rightsquigarrow p$
- $1_x \cdot p \rightsquigarrow p$
- $p \cdot p^{-1} \rightsquigarrow 1_x$
- $p^{-1} \cdot p \rightsquigarrow 1_y$

Theorem (van den Berg & Garner)

$$(A, \rightsquigarrow_A, \rightsquigarrow_{\rightsquigarrow_A}, \ldots)$$

forms weak ω-groupoid, i.e., groupoid laws hold up to "higher" paths

Interpreting types as topological spaces?

Interpreting types as Kan complexes

Voevodsky gives a model of univalent type theory in the category of Kan complexes.
Modulo a proof of a suitable initiality conjecture, this gives rise to an interpretation of type theory in “spaces”.

Interpreting types as topological spaces?

It seems difficult (impossible?) to give a formal interpretation of type theory in a convenient category of topological spaces.

There is a ‘Quillen equivalence’ between the category of simplicial sets and the category of topological spaces, justifying the intuition of ‘types as (topological) spaces’.

Interpreting types as simplicial sets

Syntax	Simpl. set interpretation
$(A, \rightsquigarrow_A, \rightsquigarrow_{\rightsquigarrow_A}, \ldots)$	Kan complex A
$a : A$	$a \in A_0$
$A \times B$	binary product
$A \to B$	space of maps
$A + B$	binary coproduct
$x : A \vdash B(x)$	fibration $B \to A$ with fibers $B(x)$
$\sum_{x:A} B(x)$	total space of fibration $B \to A$
$\prod_{x:A} B(x)$	space of sections of fibration $B \to A$

Outline

Contractible types

Definition

The type A is **contractible** if we can construct a term of type

$$\mathsf{isContr}(A) :\equiv \sum_{x:A} \prod_{y:A} y \rightsquigarrow x$$

A contractible type. . .

- is also called **singleton** type.
- has a point and a path from any point to that point.

By path inversion and concatenation, there is a path between any two points of a contractible type.

Equivalences

Definition

A map $f : A \to B$ is an **equivalence** if it has contractible fibers, i.e.,

$$\mathsf{isequiv}(f) :\equiv \prod_{b:B} \mathsf{isContr}\left(\sum_{a:A} f(a) \rightsquigarrow b\right)$$

The type of equivalences:

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

Exercise

Given an equivalence $f : A \simeq B$, define a function $g : B \to A$.
Construct paths $f(g(y)) \rightsquigarrow y$ and $g(f(x)) \rightsquigarrow x$.

Exercises

- Show that 1 is contractible.
- Let A be a contractible type. Construct an equivalence $A \simeq 1$.
- Given types A and B, let $f : A \to B$ and $g : B \to A$. Suppose having families of paths $\eta_x : g(f(x)) \rightsquigarrow x$ and $\epsilon_y : f(g(y)) \rightsquigarrow y$. Show that f is an equivalence.

Path types of pairs

Can construct equivalences (i.e., terms of type)

- for $a, a' : A$ and $b, b' : B$,

$$(a, b) \rightsquigarrow (a', b') \;\simeq\; (a \rightsquigarrow a') \times (b \rightsquigarrow b')$$

- for $a, a' : A$ and $b : B(a)$ and $b' : B(a')$,

$$(a, b) \rightsquigarrow (a', b') \;\simeq\; \sum_{p : a \rightsquigarrow a'} \mathsf{transport}^B(p, b) \rightsquigarrow b'$$

The equivalences map $\mathsf{refl}(a, b)$ to $\big(\mathsf{refl}(a), \mathsf{refl}(b)\big)$.

Path types of pairs

Can construct equivalences (i.e., terms of type)

- for $a, a' : A$ and $b, b' : B$,

$$(a, b) \rightsquigarrow (a', b') \;\simeq\; (a \rightsquigarrow a') \times (b \rightsquigarrow b')$$

- for $a, a' : A$ and $b : B(a)$ and $b' : B(a')$,

$$(a, b) \rightsquigarrow (a', b') \;\simeq\; \sum_{p : a \rightsquigarrow a'} \mathsf{transport}^B(p, b) \rightsquigarrow b'$$

The equivalences map $\mathsf{refl}(a, b)$ to $\big(\mathsf{refl}(a), \mathsf{refl}(b)\big)$.

Exercise

Given functions $f, g : A \to B$, complete

$$f \rightsquigarrow g \;\simeq\; ???$$

Path types of function spaces

For $f, g : A \to B$ cannot construct

$$f \rightsquigarrow g \;\simeq\; \prod_{a:A} f(a) \rightsquigarrow g(a)$$

Define

$$\mathsf{toPointwisePath} : \prod_{f,g:A\to B} f \rightsquigarrow g \to \left(\prod_{a:A} f(a) \rightsquigarrow g(a)\right)$$

$$\mathsf{toPointwisePath}(f,f,\mathsf{refl}(f)) :\equiv \lambda a.\mathsf{refl}(f(a))$$

Axiom (function extensionality)

$$\mathsf{toPointwisePath}(f,g) : f \rightsquigarrow g \;\to\; \left(\prod_{a:A} f(a) \rightsquigarrow g(a)\right)$$

is an equivalence for any f, g.

Outline

Some types are propositions

Curry–Howard

- Types are propositions.
- Terms are proofs.

Univalent logic

- **Some** types are propositions.
- Terms **of those types** are proofs.

Definition (Propositions in univalent type theory)

Type A is a **proposition** if

$$\mathsf{isProp}(A) :\equiv \prod_{x,y:A} x \rightsquigarrow y$$

is inhabited.

Examples of propositions

Exercise: show that

- 1 is a proposition.
- any contractible type is a proposition.
- 0 is a proposition.
- if A and B are propositions, then $A \times B$ is a proposition.
- if B is a proposition, then $A \to B$ is a proposition.

Connectives in univalent logic

Definition

$$\mathsf{Prop} :\equiv \sum_{X:\mathsf{Type}} \mathsf{isProp}(X)$$

We want logical connectives

$$\begin{aligned}
\top, \bot &: \mathsf{Prop} \\
\vee, \wedge, \Rightarrow &: \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} \\
\neg &: \mathsf{Prop} \to \mathsf{Prop} \\
\forall_X, \exists_X &: (X \to \mathsf{Prop}) \to \mathsf{Prop} \qquad \text{(binding a variable)}
\end{aligned}$$

Univalent logic

- 1 and 0 are propositions. Hence

$$\top :\equiv 1 \quad \bot :\equiv 0$$

- If A and B are propositions, so is $A \times B$. Hence

$$A \wedge B :\equiv A \times B$$

- If B is a proposition, so is $A \to B$. Hence

$$A \Rightarrow B :\equiv A \to B$$

- 0 is a proposition, hence $A \to 0$ is. Hence

$$\neg A :\equiv A \to 0$$

- If $B(a)$ (for any a) are propositions, so is $\prod_{a:A} B(a)$. Hence

$$\forall (a : A), B(a) :\equiv \prod_{a:A} B(a)$$

$\vee$ and $\exists$ in univalent logic

- Exercise: Find a type T that is a proposition such that $T + T$ is not a proposition.
  Conclusion: can **not** set

$$A \vee B \quad :\equiv \quad A + B$$

- $\Sigma_{n:\mathsf{Nat}} \mathsf{isEven}(n)$ is the type of all even natural numbers. It is not a proposition.
  Conclusion: can **not** set

$$\exists (a : A), B(a) \quad :\equiv \quad \Sigma_{a:A} B(a)$$

Solution: introduce a type former that makes propositions.

Propositional truncation

Formation If A is a type, then $||A||$ is a type

Introduction If $a : A$, then $\overline{a} : ||A||$
$p(A) : \prod_{x,y:||A||} x \rightsquigarrow y$

Elimination If $f : A \to B$ and B is a proposition, then $\overline{f} : ||A|| \to B$

Computation $\overline{f}(\overline{a}) \equiv f(a)$

- $p(A)$ turns $||A||$ into a proposition.
- Intuitively, $||A||$ is empty if A is, and contractible if A has at least one element.

Propositional truncation

Formation If A is a type, then $\|A\|$ is a type

Introduction If $a : A$, then $\bar{a} : \|A\|$
$p(A) : \prod_{x,y:\|A\|} x \rightsquigarrow y$

Elimination If $f : A \to B$ and B is a proposition, then $\bar{f} : \|A\| \to B$

Computation $\bar{f}(\bar{a}) \equiv f(a)$

- $p(A)$ turns $\|A\|$ into a proposition.
- Intuitively, $\|A\|$ is empty if A is, and contractible if A has at least one element.

Exercise

Show that $\|A\| :\equiv \prod_{P:\mathsf{Prop}} (A \to P) \to P$ satisfies the rules.

$\vee$ and $\exists$ in univalent logic

- $$A \vee B \quad :\equiv \quad \|A + B\|$$
- $$\exists(a : A), B(a) \quad :\equiv \quad \|\Sigma_{a:A} B(a)\|$$

For example:

$$\mathsf{isSurjective}(f) :\equiv \prod_{b:B} \|\Sigma_{a:A} f(a) \rightsquigarrow b\|$$

Propositional extensionality

We would like to consider two propositions to be equal if they are logically equivalent:

$$\prod_{P,Q:\mathsf{Prop}} (P \rightsquigarrow Q) \simeq (P \leftrightarrow Q)$$

Propositional extensionality

We would like to consider two propositions to be equal if they are logically equivalent:

$$\prod_{P,Q:\mathsf{Prop}} (P \rightsquigarrow Q) \simeq (P \leftrightarrow Q)$$

Axiom: propositional extensionality

The family of maps

$$\prod_{P,Q:\mathsf{Prop}} (P \rightsquigarrow Q) \to (P \leftrightarrow Q)$$

is an equivalence pointwise.

Outline

Contractible types, propositions and sets

- A is **contractible** if we can construct a term of type

$$\mathsf{isContr}(A) :\equiv \sum_{x:A} \prod_{y:A} y \rightsquigarrow x$$

- A is a **proposition** if $\prod_{x,y:A} x \rightsquigarrow y$ is inhabited

$$\mathsf{isProp}(A) :\equiv \prod_{x,y:A} x \rightsquigarrow y$$

- A is a **set** if, for any $x, y : A$, the type $x \rightsquigarrow y$ is a proposition

$$\mathsf{isSet}(A) :\equiv \prod_{x,y:A} \mathsf{isProp}(x \rightsquigarrow y)$$

Contractible types, propositions and sets

- A is **contractible** if we can construct a term of type

$$\mathsf{isContr}(A) :\equiv \sum_{x:A} \prod_{y:A} y \rightsquigarrow x$$

- A is a **proposition** if $\prod_{x,y:A} \mathsf{isContr}(x \rightsquigarrow y)$ is inhabited

$$\mathsf{isProp}(A) :\equiv \prod_{x,y:A} \mathsf{isContr}(x \rightsquigarrow y)$$

- A is a **set** if, for any $x, y : A$, the type $x \rightsquigarrow y$ is a proposition

$$\mathsf{isSet}(A) :\equiv \prod_{x,y:A} \mathsf{isProp}(x \rightsquigarrow y)$$

Exercises

- For a type A, show that $\prod_{x,y:A} \mathsf{isContr}(x \rightsquigarrow y) \leftrightarrow \prod_{x,y:A} x \rightsquigarrow y$.
- Show that Bool is a set. Is it contractible? Is it a proposition?
- Show that Nat is a set. Is it contractible? Is it a proposition?

Homotopy level of a type

Definition

$$\mathsf{isofhlevel} : \mathsf{Nat} \to \mathsf{Type} \to \mathsf{Type}$$

$$\mathsf{isofhlevel}(0)(X) :\equiv \mathsf{isContr}(X)$$

$$\mathsf{isofhlevel}(S(n))(X) :\equiv \prod_{x,y:X} \mathsf{isofhlevel}(n)(x \rightsquigarrow y)$$

Homotopy level of a type

Definition

$$\mathsf{isofhlevel} : \mathsf{Nat} \to \mathsf{Type} \to \mathsf{Prop}$$

$$\mathsf{isofhlevel}(0)(X) :\equiv \mathsf{isContr}(X)$$

$$\mathsf{isofhlevel}(S(n))(X) :\equiv \prod_{x,y:X} \mathsf{isofhlevel}(n)(x \rightsquigarrow y)$$

Exercise

Show that $\mathsf{isofhlevel}(n)(X)$ is a proposition.

Preservation of levels

. . . by type constructors

- If A and B are of hlevel n, then so is $A \times B$.
- If B is of hlevel n, then so is $A \to B$.
- If A and $B(a)$ (for any $a : A$) are of hlevel n, then so is $\sum_{a:A} B(a)$.
- If $B(a)$ (for any $a : A$) are of hlevel n, then so is $\prod_{a:A} B(a)$.

. . . under equivalence of types

If A is of level n and $A \simeq B$ then B is of hlevel n.

Cumulativity

If type A is of hlevel n, then it is also of hlevel $S(n)$.

Outline

Universes

Reminder:

Universes

There is also a type Type that contains types, i.e., $A : \mathsf{Type}$.

Exercise: What is the path type of two types in Type?

Fill in

$$A \rightsquigarrow_{\mathsf{Type}} B \quad \simeq$$

Universes

Reminder:

Universes

There is also a type Type that contains types, i.e., $A : \mathsf{Type}$.

Exercise: What is the path type of two types in Type?

Fill in

$$A \rightsquigarrow_{\mathsf{Type}} B \quad \simeq \quad (A \simeq B)$$

The type theory seen so far does not allow one to construct such an equivalence.

Voevodsky's Univalence Axiom

Definition

$$\mathsf{idtoequiv} : \prod_{A,B:\mathsf{Type}} (A \rightsquigarrow B) \to (A \simeq B)$$

$$\mathsf{idtoequiv}(A, A, \mathsf{refl}(A)) :\equiv \mathsf{id}_A$$

Univalence Axiom

$$\mathsf{univalence} : \prod_{A,B:\mathsf{Type}} \mathsf{isequiv}(\mathsf{idtoequiv}_{A,B})$$

Note that univalence is stated for a fixed universe Type.

Consequences of the Univalence Axiom

- Function extensionality

$$\prod_{f,g:A\to B}\left(f \rightsquigarrow g\right) \simeq \left(\prod_{a:A} f(a) \rightsquigarrow g(a)\right)$$

- Propositional extensionality:

$$\prod_{P,Q:\mathsf{Prop}} (P \rightsquigarrow Q) \simeq (P \leftrightarrow Q)$$

- Paths are isomorphisms for sets:

$$\prod_{X,Y:\mathsf{Set}} (X \rightsquigarrow Y) \simeq (X \cong Y)$$

  where

$$X \cong Y :\equiv \sum_{f:X\to Y} \sum_{g:Y\to X} \dots$$

Summary: Univalent Foundations

- Univalent type theory with an interpretation in spaces (precisely: Kan complexes)

Type theory	Interpretation
A type	space A
$a : A$ (term a of type A)	point a in space A
$f : A \to B$	map from A to B
$p : a \rightsquigarrow b$	path (1-morphism) from a to b in A
$\alpha : p \rightsquigarrow_{a \rightsquigarrow b} q$	homotopy from p to q in A

- "World" of **logic** (propositions and proofs) given by Prop
- "World" of **sets** given by $\mathsf{Set} :\equiv \sum_{X:\mathsf{Type}} \mathsf{isSet}(X)$

Outline

Indiscernability of identicals

Indiscernability of identicals

$$x = y \quad \rightarrow \quad \forall P\,(P(x) \leftrightarrow P(y))$$

- Reasoning **in logic** is invariant under equality
- **In mathematics**, reasoning should be invariant under weaker notion of sameness!

The equivalence principle

Equivalence principle

Reasoning in mathematics should be **invariant under** the appropriate notion of **sameness**.

Notion of sameness depends on the objects under consideration:

- **equal** numbers, functions,...
- **isomorphic** sets, groups, rings,...
- **equivalent** categories
- **biequivalent** bicategories
- ...

A language for invariant properties

Michael Makkai, *Towards a Categorical Foundation of Mathematics*:

> *The basic character of the Principle of Isomorphism is that of a* ***constraint on the language*** *of Abstract Mathematics; a welcome one, since it provides for the separation of sense from nonsense.*

A language for invariant properties

Michael Makkai, *Towards a Categorical Foundation of Mathematics*:

> *The basic character of the Principle of Isomorphism is that of a* ***constraint on the language*** *of Abstract Mathematics; a welcome one, since it provides for the separation of sense from nonsense.*

Makkai's FOLDS

a language in which only those properties of structures can be expressed that are invariant under equivalence of structures

Voevodsky's take on the equivalence principle

> *[. . .] My homotopy lambda calculus is an attempt to create a system which is very good at dealing with equivalences. In particular it is supposed to have the property that given any type expression F(T) depending on a term subexpression t of type T and an equivalence t->t' (a term of the type Eq(T;t,t')) there is a mechanical way to create a new expression F' now depending on t' and an equivalence between F(T) and F'(T') (note that to get F' one can not just substitute t' for t in F – the resulting expression will most likely be syntactically incorrect).*

Email to Grayson, Sept 2006

Transport along isomorphism

Transport revisited

$$\mathsf{transport}_{x,y} : (x \rightsquigarrow y) \to \prod_{B:A\to\mathsf{Type}} \big(B(x) \simeq B(y)\big)$$

Want:

$$\mathsf{transport}_{x,y} : (x \cong y) \to \prod_{B:A\to\mathsf{Type}} \big(B(x) \simeq B(y)\big)$$

Suffices:

$$(x \rightsquigarrow y) \quad \simeq \quad (x \cong y)$$

Transport along isomorphism

Transport revisited

$$\mathsf{transport}_{x,y} : (x \rightsquigarrow y) \to \prod_{B:A\to\mathsf{Type}} \big(B(x) \simeq B(y)\big)$$

Want:

$$\mathsf{transport}_{x,y} : (x \cong y) \to \prod_{B:A\to\mathsf{Type}} \big(B(x) \simeq B(y)\big)$$

Suffices:

$$(x \rightsquigarrow y) \xrightarrow{\simeq} (x \cong y)$$

Transport along isomorphism

Transport revisited

$$\mathsf{transport}_{x,y} : (x \rightsquigarrow y) \to \prod_{B:A\to\mathsf{Type}} \big(B(x) \simeq B(y)\big)$$

Want:

$$\mathsf{transport}_{x,y} : (x \cong y) \to \prod_{B:A\to\mathsf{Type}} \big(B(x) \simeq B(y)\big)$$

Suffices:

$$(x \rightsquigarrow y) \quad \simeq \quad (x \cong y)$$

We establish this equivalence for the example of monoids.

Monoids

Traditionally (in set theory), a monoid is a triple (M, μ, e) of

- a set M
- a multiplication $\mu : M \times M \to M$
- a unit $e \in M$

subject to the usual axioms: associativity, and left and right neutrality.

Monoids in type theory

In type theory, a monoid is a tuple $(M, \mu, e, \alpha, \lambda, \rho)$ where

1. $M : \mathsf{Set}$
2. $\mu : M \times M \to M$ (multiplication)
3. $e : M$ (neutral element)
4. $\alpha : \Pi_{(a,b,c:M)} \mu(\mu(a,b),c) \rightsquigarrow \mu(a,\mu(b,c))$ (associativity)
5. $\lambda : \Pi_{(a:M)} \mu(e,a) \rightsquigarrow a$ (left neutrality)
6. $\rho : \Pi_{(a:M)} \mu(a,e) \rightsquigarrow a$ (right neutrality)

Monoids in type theory

In type theory, a monoid is a tuple $(M, \mu, e, \alpha, \lambda, \rho)$ where

1. $M : \mathsf{Set}$
2. $\mu : M \times M \to M$ (multiplication)
3. $e : M$ (neutral element)
4. $\alpha : \Pi_{(a,b,c:M)} \mu(\mu(a,b),c) \rightsquigarrow \mu(a,\mu(b,c))$ (associativity)
5. $\lambda : \Pi_{(a:M)} \mu(e,a) \rightsquigarrow a$ (left neutrality)
6. $\rho : \Pi_{(a:M)} \mu(a,e) \rightsquigarrow a$ (right neutrality)

Why $M : \mathsf{Set}$?

Monoids in type theory

In type theory, a monoid is a tuple $(M, \mu, e, \alpha, \lambda, \rho)$ where

1. $M : \mathsf{Set}$
2. $\mu : M \times M \to M$ (multiplication)
3. $e : M$ (neutral element)
4. $\alpha : \Pi_{(a,b,c:M)} \mu(\mu(a,b),c) \rightsquigarrow \mu(a,\mu(b,c))$ (associativity)
5. $\lambda : \Pi_{(a:M)} \mu(e,a) \rightsquigarrow a$ (left neutrality)
6. $\rho : \Pi_{(a:M)} \mu(a,e) \rightsquigarrow a$ (right neutrality)

Why $M : \mathsf{Set}$?

Abstractly, a monoid is a (dependent) pair $(data, proof)$ where

- $data$ is a triple (M, μ, e) as above
- $proof$ is a triple (α, λ, ρ) saying that $(data)$ satisfy the usual axioms.

The type of monoids

- We want two monoids $(data, proof)$ and $(data', proof')$ to be the same if $data$ is the same as $data'$.
- This is ensured when the type of $proof$ and $proof'$ is a **proposition**.
- This in turn is guaranteed when the underlying type M is a **set**.

Summarily:

$$\mathsf{Monoid} :\equiv \sum_{(M:\mathsf{Set})} \sum_{(\mu,e):\mathsf{MonoidStr}(M)} \mathsf{MonoidAxioms}(M,(\mu,e))$$

Can show

$$\mathsf{isProp}(\mathsf{MonoidAxioms}(M,(\mu,e)))$$

Monoid isomorphisms

Given monoids $\mathbf{M} \equiv (M, \mu, e, \alpha, \lambda, \rho)$ and $\mathbf{M}' \equiv (M', \mu', e', \alpha', \lambda', \rho')$, a **monoid isomorphism** is a bijection $f : M \cong M'$ preserving multiplication and neutral element.

Monoid isomorphisms

Given monoids $\mathbf{M} \equiv (M, \mu, e, \alpha, \lambda, \rho)$ and $\mathbf{M}' \equiv (M', \mu', e', \alpha', \lambda', \rho')$, a **monoid isomorphism** is a bijection $f : M \cong M'$ preserving multiplication and neutral element.

$$
\begin{aligned}
\mathbf{M} \rightsquigarrow \mathbf{M}' &\simeq (M, \mu, e) \rightsquigarrow (M', \mu', e') \\
&\simeq \sum_{p : M \rightsquigarrow M'} (\mathsf{transport}^{Y \mapsto (Y \times Y \to Y)}(p, \mu) \rightsquigarrow \mu') \\
&\qquad \times (\mathsf{transport}^{Y \mapsto Y}(p, e) \rightsquigarrow e') \\
&\simeq \sum_{f : M \cong M'} \left(f \circ \mu \circ (f^{-1} \times f^{-1}) \rightsquigarrow \mu'\right) \\
&\qquad \times (f \circ e \rightsquigarrow e') \\
&\simeq \mathbf{M} \cong \mathbf{M}'
\end{aligned}
$$

Transport along monoid isomorphism

We now have two ingredients:

1. $$(\mathbf{M} \rightsquigarrow \mathbf{M}') \simeq (\mathbf{M} \cong \mathbf{M}')$$

2. $$\mathsf{transport}_{\mathbf{M},\mathbf{M}'} : (\mathbf{M} \rightsquigarrow \mathbf{M}') \to \prod_{B:\mathsf{Monoid}\to\mathsf{Type}} \left(B(\mathbf{M}) \simeq B(\mathbf{M}')\right)$$

Composing these, we get

$$\mathsf{transport}_{\mathbf{M},\mathbf{M}'} : (\mathbf{M} \cong \mathbf{M}') \to \prod_{B:\mathsf{Monoid}\to\mathsf{Type}} \left(B(\mathbf{M}) \simeq B(\mathbf{M}')\right)$$

I.e., any structure on monoids that can be expressed in univalent type theory can be transported along isomorphism of monoids.

Equivalence principle for set-level structures

EP for set-level structures (Coquand&Danielsson)

For many set-level structures in univalent foundations, paths are isomorphisms.

Examples include:

- monoids, groups, rings
- posets
- discrete fields
- sets with fixpoint operator

What about **categories**?

EP for categories

Conjecture

For categories $\mathscr{C}$ and $\mathscr{D}$, the canonical map

$$(\mathscr{C} \rightsquigarrow \mathscr{D}) \;\to\; \mathsf{AdjEquiv}(\mathscr{C}, \mathscr{D})$$

is an equivalence.

EP for categories

Conjecture

For categories $\mathscr{C}$ and $\mathscr{D}$, the canonical map

$$(\mathscr{C} \rightsquigarrow \mathscr{D}) \quad \to \quad \mathsf{AdjEquiv}(\mathscr{C}, \mathscr{D})$$

is an equivalence.

Counter-example

Categories in univalent foundations

Definition

A **category** $\mathscr{C}$ is given by

- a **type** $\mathscr{C}_0$: Type of **objects**
- for any $a, b : \mathscr{C}_0$, a **set** $\mathscr{C}(a,b)$: Set of **morphisms**
- operations: identity & composition

$$1_a : \mathscr{C}(a,a)$$
$$(\circ)_{a,b,c} : \mathscr{C}(b,c) \times \mathscr{C}(a,b) \to \mathscr{C}(a,c)$$

- axioms: unitality & associativity

$$1 \circ f \rightsquigarrow f \qquad f \circ 1 \rightsquigarrow f \qquad (h \circ g) \circ f \rightsquigarrow h \circ (g \circ f)$$

From paths to isomorphisms

Definition (univalent category)

For a category $\mathscr{C}$ we define

$$\mathsf{idtoiso} : \prod_{a,b:\mathscr{C}_0} (a \rightsquigarrow b) \to \mathsf{iso}(a,b)$$

$$\mathsf{idtoiso}(a,a,\mathsf{refl}(a)) :\equiv 1_a$$

We call the category $\mathscr{C}$ **univalent** if, for any objects $a, b : \mathscr{C}_0$,

$$\mathsf{idtoiso}_{a,b} : (a \rightsquigarrow b) \to \mathsf{iso}(a,b)$$

is an equivalence of types.

Examples of univalent categories

- Set
- Category of monoids (as proved above)
- Groups, rings, . . . (Structure Identity Principle)
- Functor category $[\mathscr{C}, \mathscr{D}]$, if $\mathscr{D}$ is univalent
- Full subcategories of univalent categories

More examples of univalent categories

- A preorder is univalent iff it is antisymmetric
- If X is of h-level 3, then there is a univalent category with X as objects and $\hom(x,y) :\equiv (x \rightsquigarrow y)$
- If $\mathscr{C}$ is univalent, then the category of cones of shape $F : \mathscr{J} \to \mathscr{C}$ is
  - $\rightsquigarrow$ limits (limiting cones) in a univalent category are unique **up to paths**

Non-univalent categories

- Any "chaotic" category $\mathscr{C}$ with $\mathscr{C}(x,y) \simeq 1$, for $\mathscr{C}_0$ not a proposition

- Any chaotic category $\mathscr{C}$ with an object $c : \mathscr{C}_0$ is **equivalent** to the terminal category $\mathbf{1}$
  - $\rightsquigarrow$ a category can be equivalent to a univalent one without being univalent itself

(Adjoint) equivalence of categories

An **equivalence** $\mathscr{C} \simeq \mathscr{D}$ is given by

- a functor $F : \mathscr{C} \to \mathscr{D}$
- a functor $G : \mathscr{D} \to \mathscr{C}$
- a natural isomorphism $\eta : 1_{\mathscr{C}} \xrightarrow{\cong} GF$
- a natural isomorphism $\epsilon : FG \xrightarrow{\cong} 1_{\mathscr{D}}$

(F, G, η, ϵ) is an adjoint equivalence if F and G form an adjunction.

(Adjoint) equivalence of categories

An **equivalence** $\mathscr{C} \simeq \mathscr{D}$ is given by

- a functor $F : \mathscr{C} \to \mathscr{D}$
- a functor $G : \mathscr{D} \to \mathscr{C}$
- a natural isomorphism $\eta : 1_{\mathscr{C}} \xrightarrow{\cong} GF$
- a natural isomorphism $\epsilon : FG \xrightarrow{\cong} 1_{\mathscr{D}}$

(F, G, η, ϵ) is an adjoint equivalence if F and G form an adjunction.

Theorem

*For **univalent** categories $\mathscr{C}$ and $\mathscr{D}$, the canonical map*

$$(\mathscr{C} \rightsquigarrow \mathscr{D}) \;\to\; \mathsf{AdjEquiv}(\mathscr{C}, \mathscr{D})$$

is an equivalence.

Research goal: a higher-categorical EP in UF

Prove a similar theorem for other categorical structures.

Envisioned result

Given a signature $\mathscr{L}$, and two $\mathscr{L}$-univalent $\mathscr{L}$-structures M and N, then

$$(M \rightsquigarrow N) \quad \to \quad (M \simeq_{\mathscr{L}} N)$$

is an equivalence.

Need notions of

- signature
- structure for a signature
- equivalence of structures
- $\mathscr{L}$-isomorphism and $\mathscr{L}$-univalence (a.k.a. saturation)

Outline

Goals

- To define (higher) loop spaces and homotopy groups
- To declare/construct types that have non-trivial higher homotopy groups

Loop space

A **pointed type** is a pair (A, a) with $A : \mathsf{Type}$ and $a : A$.

Definition

The **(first) loop space** $\Omega(A, a)$ is defined as the pointed type

$$\Omega(A, a) \quad :\equiv \quad (a \rightsquigarrow_A a, 1_a)$$

The $(n+1)$**-th loop space** is defined as

$$\Omega^{n+1}(A, a) \quad :\equiv \quad \Omega(\Omega^n(A, a))$$

We have operations

- $1_a : \Omega(A, a)$
- $\_ \cdot \_ : \Omega(A, a) \times \Omega(A, a) \to \Omega(A, a)$
- $\_^{-1} : \Omega(A, a) \to \Omega(A, a)$
- $\Omega(f) : \Omega(A, a) \to \Omega(B, fa)$ for $f : A \to B$

Fundamental group

Definition

Let $n \geq 1$. Given a pointed type (A, a), its **n-th homotopy group** has, as underlying set,

$$\pi_n(A, a) = ||\Omega^n(A, a)||_2$$

$\pi_n(A, a)$ inherits group structure from operations of inversion and concatenation on $\Omega^n(A, a)$.

Theorem

The composition on the iterated loop space $\Omega^2(A, a)$ is commutative.

Corollary

$\pi_n(A, a)$ is abelian for $n \geq 2$.

Truncated Whitehead's Theorem

Theorem (HoTT book, Theorem 8.8.3)

Let A and B be types of homotopy level n, and let $f : A \to B$ such that

- $\|f\|_2 : \|A\|_2 \to \|B\|_2$ *is a bijection; and*
- $\pi_k(f) : \pi_k(A, a) \to \pi_k(B, fa)$ *is a bijection for* $k \geq 1$ *and* $a : A$

Then f is an equivalence.

The encode-decode method for computing path spaces

- HITs are axiomatically declared types, usually with interesting homotopy groups
- The encode-decode method is a systematic way to characterize path spaces of types, in particular of HITs

Now: walk through characterization of path spaces via encode-decode for two examples:

- coproduct type $A + B$
- HIT $\mathbb{S}^1$ (circle)

Coproduct

Formation If A and B are types, so is $A + B$

Introduction $\mathsf{inl} : A \to A + B$, $\mathsf{inr} : B \to A + B$

Elimination
$$\begin{aligned} t : A + B &\vdash C(t) \\ x : A &\vdash l : C(\mathsf{inl}(x)) \\ y : B &\vdash r : C(\mathsf{inr}(y)) \\ &\text{then} \\ t : A + B &\vdash \mathsf{ind}_+(l, r, t) : C(t) \end{aligned}$$

Computation ...

The identity type of a coproduct

Would like to construct

$$\begin{aligned} \mathsf{inl}(a) \rightsquigarrow \mathsf{inl}(a') &\simeq a \rightsquigarrow a' \\ \mathsf{inr}(b) \rightsquigarrow \mathsf{inr}(b') &\simeq b \rightsquigarrow b' \\ \mathsf{inl}(a) \rightsquigarrow \mathsf{inr}(b) &\simeq \mathrm{o} \end{aligned}$$

Method:

1. construct maps in both directions
2. show that they are inverses

Problem: map from left to right not easy to define, even harder to reason about.

Solution: “encode-decode” method

Codes for identities in coproduct

Fix $a_0 : A$. We want to show

$$\begin{aligned} \mathsf{inl}(a_0) \rightsquigarrow \mathsf{inl}(a) &\simeq a_0 \rightsquigarrow a \\ \mathsf{inl}(a_0) \rightsquigarrow \mathsf{inr}(b) &\simeq 0 \end{aligned}$$

Define two functions $l, r : A + B \to \mathsf{Type}$ such that

$$l(t) \simeq r(t)$$

and

$$\begin{aligned} l(\mathsf{inl}(a)) &\equiv \mathsf{inl}(a_0) \rightsquigarrow \mathsf{inl}(a) \\ l(\mathsf{inr}(b)) &\equiv \mathsf{inl}(a_0) \rightsquigarrow \mathsf{inr}(b) \\ r(\mathsf{inl}(a)) &\equiv a_0 \rightsquigarrow a \\ r(\mathsf{inr}(b)) &\equiv 0 \end{aligned}$$

The function r represents "codes" for the identity type to be characterized, hence is also called code.

Encode-decode for coproduct

$$\mathsf{decode}(t) : \mathsf{code}(t) \to l(t)$$

$$\mathsf{encode}(t) : l(t) \to \mathsf{code}(t)$$

Encode-decode for coproduct

$$\mathsf{decode}(t) : \mathsf{code}(t) \to l(t)$$

$$\mathsf{decode}(\mathsf{inl}(a)) : a_{\mathrm{o}} \rightsquigarrow a \to \mathsf{inl}(a_{\mathrm{o}}) \rightsquigarrow \mathsf{inl}(a)$$

$$\mathsf{encode}(t) : l(t) \to \mathsf{code}(t)$$

Encode-decode for coproduct

$$\mathsf{decode}(t) : \mathsf{code}(t) \to l(t)$$

$$\mathsf{decode}(\mathsf{inl}(a)) : a_{\mathrm{o}} \rightsquigarrow a \to \mathsf{inl}(a_{\mathrm{o}}) \rightsquigarrow \mathsf{inl}(a)$$

$$\mathsf{decode}(\mathsf{inr}(b)) : \mathrm{o} \to \mathsf{inl}(a_{\mathrm{o}}) \rightsquigarrow \mathsf{inr}(b)$$

$$\mathsf{encode}(t) : l(t) \to \mathsf{code}(t)$$

Encode-decode for coproduct

$$\mathsf{decode}(t) : \mathsf{code}(t) \to l(t)$$

$$\mathsf{decode}(\mathsf{inl}(a)) : a_{\mathrm{o}} \rightsquigarrow a \to \mathsf{inl}(a_{\mathrm{o}}) \rightsquigarrow \mathsf{inl}(a)$$

$$\mathsf{decode}(\mathsf{inr}(b)) : \mathrm{o} \to \mathsf{inl}(a_{\mathrm{o}}) \rightsquigarrow \mathsf{inr}(b)$$

$$\mathsf{encode}(t) : \mathsf{inl}(a_{\mathrm{o}}) \rightsquigarrow t \to \mathsf{code}(t)$$

Encode-decode for coproduct

$$\mathsf{decode}(t) : \mathsf{code}(t) \to l(t)$$

$$\mathsf{decode}(\mathsf{inl}(a)) : a_{\mathrm{o}} \rightsquigarrow a \to \mathsf{inl}(a_{\mathrm{o}}) \rightsquigarrow \mathsf{inl}(a)$$

$$\mathsf{decode}(\mathsf{inr}(b)) : \mathrm{o} \to \mathsf{inl}(a_{\mathrm{o}}) \rightsquigarrow \mathsf{inr}(b)$$

$$\mathsf{encode}(t) : \mathsf{inl}(a_{\mathrm{o}}) \rightsquigarrow t \to \mathsf{code}(t)$$

$$\mathsf{encode}(t) : p \mapsto \mathsf{transport}^{\mathsf{code}}\Big(p, \mathsf{refl}(\mathsf{inl}(a_{\mathrm{o}}))\Big)$$

Encode-decode for coproduct

$$\mathsf{decode}(t) : \mathsf{code}(t) \to l(t)$$
$$\mathsf{decode}(\mathsf{inl}(a)) : a_{\mathrm{o}} \rightsquigarrow a \to \mathsf{inl}(a_{\mathrm{o}}) \rightsquigarrow \mathsf{inl}(a)$$
$$\mathsf{decode}(\mathsf{inr}(b)) : \mathrm{o} \to \mathsf{inl}(a_{\mathrm{o}}) \rightsquigarrow \mathsf{inr}(b)$$
$$\mathsf{encode}(t) : \mathsf{inl}(a_{\mathrm{o}}) \rightsquigarrow t \to \mathsf{code}(t)$$
$$\mathsf{encode}(t) : p \mapsto \mathsf{transport}^{\mathsf{code}}\Big(p, \mathsf{refl}(\mathsf{inl}(a_{\mathrm{o}}))\Big)$$

Then show that

$$\mathsf{encode}(t) \circ \mathsf{decode}(t) \rightsquigarrow 1$$

by induction on t (and then path induction in one case) and

$$\mathsf{decode}(t) \circ \mathsf{encode}(t) \rightsquigarrow 1$$

by path induction without induction on t.

Higher Inductive Types (HITs)

- A HIT has both term formation and path formation rules.
- Elimination rule must take the path formation rule into account.
- Computation rule on canonical paths only given by a path, not modulo $\equiv$.

We look at the example of the circle, and compute its loop space using the encode-decode method.

The circle

Formation $\mathbb{S}^1$ is a type

Introduction $\mathsf{base} : \mathbb{S}^1$

$\mathsf{loop} : \mathsf{base} \rightsquigarrow \mathsf{base}$

Elimination $b : X$

$l : b \rightsquigarrow b$

yields

$\mathsf{rec}_{\mathbb{S}^1}(b, l) : \mathbb{S}^1 \to X$

Computation $\mathsf{rec}_{\mathbb{S}^1}(b, l)(\mathsf{base}) \equiv b$

$\mathsf{rec}_{\mathbb{S}^1}(b, l)(\mathsf{loop}) \rightsquigarrow l$

The loop space of the circle

Problem

Construct an equivalence

$$\mathsf{base} \rightsquigarrow \mathsf{base} \;\simeq\; \mathbb{Z}$$

Construction: We define

$$l : \mathbb{S}^1 \to \mathsf{Type}$$

$$l : x \mapsto \mathsf{base} \rightsquigarrow x$$

$$\mathsf{code} : \mathbb{S}^1 \to \mathsf{Type}$$

$$\mathsf{code} : \mathsf{base} \mapsto \mathbb{Z}$$

$$\mathsf{code} : \mathsf{loop} \mapsto \mathsf{univalence}(+1 : \mathbb{Z} \simeq \mathbb{Z}) : \mathbb{Z} \rightsquigarrow \mathbb{Z}$$

Encode-decode for $\mathbb{S}^1$

$$\mathsf{encode}(x) : l(x) \to \mathsf{code}(x)$$

$$\mathsf{decode}(x) : \mathsf{code}(t) \to l(t)$$

Encode-decode for $\mathbb{S}^1$

$$\mathsf{encode}(x) : \mathsf{base} \rightsquigarrow x \to \mathsf{code}(x)$$

$$\mathsf{decode}(x) : \mathsf{code}(t) \to l(t)$$

Encode-decode for $\mathbb{S}^1$

$$\mathsf{encode}(x) : \mathsf{base} \rightsquigarrow x \to \mathsf{code}(x)$$

$$\mathsf{encode}(x) : p \mapsto \mathsf{transport}^{\mathsf{code}}(p, \mathsf{refl}(\mathsf{base}))$$

$$\mathsf{decode}(x) : \mathsf{code}(t) \to l(t)$$

Encode-decode for $\mathbb{S}^1$

$$\mathsf{encode}(x) : \mathsf{base} \rightsquigarrow x \to \mathsf{code}(x)$$

$$\mathsf{encode}(x) : p \mapsto \mathsf{transport}^{\mathsf{code}}(p, \mathsf{refl}(\mathsf{base}))$$

$$\mathsf{decode}(x) : \mathsf{code}(t) \to l(t)$$

$$\mathsf{decode}(\mathsf{base}) : \mathsf{code}(\mathsf{base}) \to l(\mathsf{base})$$

Encode-decode for $\mathbb{S}^1$

$$\mathsf{encode}(x) : \mathsf{base} \rightsquigarrow x \to \mathsf{code}(x)$$

$$\mathsf{encode}(x) : p \mapsto \mathsf{transport}^{\mathsf{code}}(p, \mathsf{refl}(\mathsf{base}))$$

$$\mathsf{decode}(x) : \mathsf{code}(t) \to l(t)$$

$$\mathsf{decode}(\mathsf{base}) : \mathbb{Z} \to \mathsf{base} \rightsquigarrow \mathsf{base}$$

Encode-decode for $\mathbb{S}^1$

$$\mathsf{encode}(x) : \mathsf{base} \rightsquigarrow x \to \mathsf{code}(x)$$

$$\mathsf{encode}(x) : p \mapsto \mathsf{transport}^{\mathsf{code}}(p, \mathsf{refl}(\mathsf{base}))$$

$$\mathsf{decode}(x) : \mathsf{code}(t) \to l(t)$$

$$\mathsf{decode}(\mathsf{base}) : \mathbb{Z} \to \mathsf{base} \rightsquigarrow \mathsf{base}$$

$$\mathsf{decode}(\mathsf{base}) : z \mapsto \mathsf{loop}^z$$

Encode-decode for $\mathbb{S}^1$

$$\mathsf{encode}(x) : \mathsf{base} \rightsquigarrow x \to \mathsf{code}(x)$$

$$\mathsf{encode}(x) : p \mapsto \mathsf{transport}^{\mathsf{code}}(p, \mathsf{refl}(\mathsf{base}))$$

$$\mathsf{decode}(x) : \mathsf{code}(t) \to l(t)$$

$$\mathsf{decode}(\mathsf{base}) : \mathbb{Z} \to \mathsf{base} \rightsquigarrow \mathsf{base}$$

$$\mathsf{decode}(\mathsf{base}) : z \mapsto \mathsf{loop}^z$$

$$\mathsf{decode}(\mathsf{loop}) : \mathsf{transport}^P(\mathsf{loop}, \mathsf{decode}(\mathsf{base})) \rightsquigarrow \mathsf{decode}(\mathsf{base})$$

$$\mathsf{decode}(\mathsf{loop}) :\equiv \ldots$$

with $P(x') :\equiv x' \mapsto \mathsf{code}(x') \to \mathsf{base} \rightsquigarrow x'$

Encode-decode for $\mathbb{S}^1$

Lemma

$$\mathsf{encode}(x) \circ \mathsf{decode}(x) \rightsquigarrow 1$$

(by circle induction on x)
and

$$\mathsf{decode}(x) \circ \mathsf{encode}(x) \rightsquigarrow 1$$

(by path induction).

Other HITs

- Other HITs are discussed extensively in Chapter 8 of the HoTT book.
- The encode-decode method can be used to characterize the identity types of many (H)ITs.

Remark

- HITs constitute an extension of Voevodsky's univalent foundations
- Bezem, Buchholtz, and Grayson have shown that $B(\mathbb{Z})$, defined as a type of $\mathbb{Z}$-torsors, is equivalent to the circle.

The End

Thanks for your attention!

Thanks in particular to Ingo Blechschmidt for preparing a list of typos during the talk.

References

- Voevodsky's emails to Grayson
  https://groups.google.com/forum/#!topic/homotopytypetheory/K_4bAZEDRvE
- Voevodsky's library *Foundations*
  https://github.com/vladimirias/Foundations,
  archived at https://github.com/UniMath/Foundations
- HoTT book https://homotopytypetheory.org/book/
- Coquand & Danielsson, Isomorphism is equality
  doi:10.1016/j.indag.2013.09.002
- Circle in UF: https://groups.google.com/d/msg/HomotopyTypeTheory/DgeCOl9mbtg/t67GfvcLBQAJ
- Hofmann & Streicher, The groupoid model of type theory