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

1. The origins of univalent foundations
2. Dependent type theory
3. Set interpretation of type theory & Axiom K
4. Simplicial set interpretation of type theory
5. Contractible types, equivalences, function extensionality
6. Logic in univalent foundations
7. Homotopy levels
8. Universes and the Univalence Axiom
9. Equivalence Principle
10. Synthetic homotopy theory
Outline

1. The origins of univalent foundations
2. Dependent type theory
3. Set interpretation of type theory & Axiom K
4. Simplicial set interpretation of type theory
5. Contractible types, equivalences, function extensionality
6. Logic in univalent foundations
7. Homotopy levels
8. Universes and the Univalence Axiom
9. Equivalence Principle
10. Synthetic homotopy theory
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
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 none. 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*.

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

1. The origins of univalent foundations
2. Dependent type theory
3. Set interpretation of type theory & Axiom K
4. Simplicial set interpretation of type theory
5. Contractible types, equivalences, function extensionality
6. Logic in univalent foundations
7. Homotopy levels
8. Universes and the Univalence Axiom
9. Equivalence Principle
10. Synthetic homotopy theory
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., \( 1 : \text{Nat} \))

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

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

Examples of dependent functions

- \( \text{zero} : \prod_{m: \text{Nat}} \text{Vect}(\text{Nat}, m) \)
- \( \text{tail} : \prod_{n: \text{Nat}} \text{Vect}(A, 1 + n) \to \text{Vect}(A, n) \)
- \( \text{GrpStrTransfer} : \prod_{X, Y: \text{Set}} (X \cong Y) \times \text{GrpStr}(X) \to \text{GrpStr}(Y) \)
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

\[ \text{context} \vdash \text{conclusion} \]

<table>
<thead>
<tr>
<th>Contexts &amp; judgments</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma )</td>
<td>sequence of variable declarations</td>
</tr>
<tr>
<td>( (x_1 : A_1), (x_2 : A_2(x_1)), \ldots, (x_n : A_n(\bar{x}_i)) )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash A )</td>
<td>( A ) is well-formed type in context ( \Gamma )</td>
</tr>
<tr>
<td>( \Gamma \vdash a : A )</td>
<td>term ( a ) is well-formed and of type ( A )</td>
</tr>
<tr>
<td>( \Gamma \vdash A \equiv B )</td>
<td>types ( A ) and ( B ) are convertible</td>
</tr>
<tr>
<td>( \Gamma \vdash a \equiv b : A )</td>
<td>( a ) is convertible to ( b ) in type ( A )</td>
</tr>
</tbody>
</table>

\( (x : \text{Nat}), (f : \text{Nat} \rightarrow \text{Bool}) \vdash f@x : \text{Bool} \)
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 $A$

$$(A : Type), (n : Nat) \vdash Vect(A, n)$$
Dependent types in pictures

\[
\begin{align*}
A & \quad a \\
\quad \downarrow B(a) & \quad \downarrow b : B(a') \\
A & \quad a' \\
\end{align*}
\]
Inference rules and derivations

An inference rule

is an implication of judgments,

\[ \frac{J_1 \quad J_2 \quad \ldots}{J} \]

e.g.,

\[ \Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash a : A \quad \Gamma \vdash a : A \quad \Gamma \vdash A \equiv B \]

\[ \Gamma \vdash f @ a : B \quad \Gamma \vdash a : B \]
An inference rule is an implication of judgments,

\[
\begin{array}{c}
J_1 \\
J_2 \\
\vdots
\end{array}
\quad \frac{\quad \ldots \quad}{J}
\]

e.g.,

\[
\begin{array}{c}
\Gamma \vdash f : A \rightarrow B \\
\Gamma \vdash a : A
\end{array}
\quad \frac{\quad \Gamma \vdash f @ a : B
}{\Gamma \vdash a : B}
\quad \Gamma \vdash a : A
\quad \frac{\quad \Gamma \vdash A \equiv B
}{\Gamma \vdash a : B}
\]

• Abbreviate rules to, e.g., “If \( a : A \) and \( A \equiv B \), then \( a : B \)”. 
Example:

\[
\begin{array}{c}
\frac{(f : A \rightarrow B) \vdash f : A \rightarrow B}{(f : A \rightarrow B), (x : A) \vdash f : A \rightarrow B} \\
\frac{(x : A), (f : A \rightarrow B) \vdash f : A \rightarrow B}{(x : A), (f : A \rightarrow B) \vdash f \@ x : B}
\end{array}
\]

\[
\begin{array}{c}
\frac{(x : A) \vdash x : A}{(x : A), (f : A \rightarrow B) \vdash x : A}
\end{array}
\]
Example:

\[
\frac{(f : A \rightarrow B) \vdash f : A \rightarrow B}{(f : A \rightarrow B), (x : A) \vdash f : A \rightarrow B}
\]

\[
\frac{(x : A), (f : A \rightarrow B) \vdash f : A \rightarrow B}{(x : A), (f : A \rightarrow B) \vdash f \@ x : B}
\]

Derivation

Term \( a \) is well-typed of type \( A \) in a context \( \Gamma \) 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 \([a] \in [A]\)

<table>
<thead>
<tr>
<th>Differences between ( a : A ) and ( a \in A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Judgement ( \Gamma \vdash a : A ) is <strong>not</strong> a statement that can be proved or disproved within type theory</td>
</tr>
<tr>
<td>2. Term ( a ) does not exist independently of its type ( A )</td>
</tr>
<tr>
<td>3. A term has exactly one type (up to ( \equiv ))</td>
</tr>
</tbody>
</table>
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
There is a type Type. Its elements are types, written $A : \text{Type}$.

Using dependent function types (cf. later), the dependent type $x : A \vdash B$ can be considered as a function

$$\lambda x.B : A \to \text{Type}$$
There is a type $Type$. Its elements are types, written $A : Type$.

Using dependent function types (cf. later), the dependent type $x : A \vdash B$ can be considered as a function

$$\lambda x. B : A \rightarrow Type$$

Actually, hierarchy $(Type_i)_{i \in I}$ to avoid paradoxes.

But we ignore this here, and only write $Type$.

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

<table>
<thead>
<tr>
<th>Formation</th>
<th>1 is a type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>t : 1</td>
</tr>
<tr>
<td>Elimination</td>
<td>If x : 1 and C is a type and c : C, then ( \text{rec}_1(C, c, x) : C )</td>
</tr>
<tr>
<td>Computation</td>
<td>( \text{rec}_1(C, c, t) \equiv c )</td>
</tr>
</tbody>
</table>
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 o is a type

Introduction

Elimination If $x : o$ and $C$ is a type, then $\text{rec}_o(C, x) : C$

Computation
Exercises

• Define the type \texttt{Bool} of booleans.
• Define the type \texttt{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 $\text{pair}(a, b) : \sum_{x:A} B(x)$

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

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

- 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: $A \rightarrow B$
A dependent function in pictures

\[
f : \Pi_{x : A} B(x)
\]

\[
f(a) : B(a)
\]

\[
f(a') : B(a')
\]
The identity type

**Formation**  If \( a : A \) and \( b : A \), then \( \text{id}_A(a, b) \) is a type

**Introduction**  If \( a : A \), then \( \text{refl}(a) : \text{id}_A(a, a) \)

**Elimination**  If
\[
(x, y : A), (p : \text{id}_A(x, y)) \vdash C(x, y, p)
\]
and
\[
(x : A) \vdash t(x) : C(x, x, \text{refl}(x))
\]
then
\[
(x, y : A), (p : \text{id}_A(x, y)) \vdash \text{ind}_{\text{id}}(x.t(x); x, y, p) : C(x, y, p)
\]

**Computation**  \( \text{ind}_{\text{id}}(x.t(x); x, x, \text{refl}(x)) \equiv t(x) \)
## Overview of types in Martin-Löf type theory

<table>
<thead>
<tr>
<th>Type former</th>
<th>Notation</th>
<th>(special case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inhabitant</td>
<td>$a : A$</td>
<td></td>
</tr>
<tr>
<td>Dependent type</td>
<td>$x : A \vdash B(x)$</td>
<td></td>
</tr>
<tr>
<td>Sigma type</td>
<td>$\sum_{x : A} B(x)$</td>
<td>$A \times B$</td>
</tr>
<tr>
<td>Product type</td>
<td>$\prod_{x : A} B(x)$</td>
<td>$A \rightarrow B$</td>
</tr>
<tr>
<td>Coproduct type</td>
<td>$A + B$</td>
<td></td>
</tr>
<tr>
<td>Identity type</td>
<td>$\text{Id}_A(a, b)$</td>
<td></td>
</tr>
<tr>
<td>Universe</td>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>Base types</td>
<td>Nat, Bool, 1, 0</td>
<td></td>
</tr>
</tbody>
</table>
Some Identity terms that can be defined

\[
\text{refl} : \prod_{x:A} \text{id}_A(x,x) \\
\text{sym} : \prod_{x,y:A} \text{id}(x,y) \to \text{id}(y,x) \\
\text{trans} : \prod_{x,y,z:A} \text{id}(x,y) \times \text{id}(y,z) \to \text{id}(x,z) \\
\text{subst} : \prod_{x,y:A} \text{id}(x,y) \to B(x) \to B(y)
\]
Some Identity terms that can be defined

\[
\text{refl} : \prod_{x : A} \text{id}_A(x, x)
\]

\[
\text{sym} : \prod_{x, y : A} \text{id}(x, y) \rightarrow \text{id}(y, x)
\]

\[
\text{trans} : \prod_{x, y, z : A} \text{id}(x, y) \times \text{id}(y, z) \rightarrow \text{id}(x, z)
\]

\[
\text{subst} : \prod_{x, y : A} \text{id}(x, y) \rightarrow B(x) \rightarrow B(y)
\]

\[\rightsquigarrow\] motivates interpretation of \(\text{id}_A(x, y)\) as equality \(x = y\)
Outline

1. The origins of univalent foundations
2. Dependent type theory
3. Set interpretation of type theory & Axiom K
4. Simplicial set interpretation of type theory
5. Contractible types, equivalences, function extensionality
6. Logic in univalent foundations
7. Homotopy levels
8. Universes and the Univalence Axiom
9. Equivalence Principle
10. Synthetic homotopy theory
### Interpreting types as sets

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Set interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>set $A$</td>
</tr>
<tr>
<td>$a : A$</td>
<td>$a \in A$</td>
</tr>
<tr>
<td>$A \times B$</td>
<td>cartesian product</td>
</tr>
<tr>
<td>$A \rightarrow B$</td>
<td>set of functions $A \rightarrow B$</td>
</tr>
<tr>
<td>$A + B$</td>
<td>disjoint union $A \coprod B$</td>
</tr>
<tr>
<td>$x : A \vdash B(x)$</td>
<td>family $B$ of sets indexed by $A$</td>
</tr>
<tr>
<td>$\sum_{x:A} B(x)$</td>
<td>disjoint union $\coprod_{x:A} B(x)$</td>
</tr>
<tr>
<td>$\prod_{x:A} B(x)$</td>
<td>product of sets</td>
</tr>
<tr>
<td>$\text{id}_A(a, b)$</td>
<td>equality $a = b$</td>
</tr>
</tbody>
</table>
Equalities between equalities

• Identity type former can be iterated:

\[ A : \text{Type}, \ x, y : A, \ p, q : \text{Id}(x, y) \vdash \text{Id}_{\text{Id}_A(x, y)}(p, q) \]

• What are its terms?

**Question**

Can one construct a term of type

\[ \text{UIP} : \prod_{(x, y : A)} \prod_{(p, q : \text{Id}(x, y))} \text{Id}(p, q) \]

**Answer**

No, cf. Hofmann & Streicher’s groupoid model of type theory.
However, UIP is independent of type theory; the set model of type theory satisfies

\[ \text{axiomK} : \prod_{(x:A)} \prod_{(p:Id(x,x))} Id(p, refl(x)) \]

One can then show

\[ \text{UIP} : \prod_{(x,y:A)} \prod_{(p,q:Id(x,y))} Id(p, q) \]
A different interpretation of identities

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

  \[ \text{id}_A(x,y) \text{ interpreted as (invertible) morphisms } x \to y \]

  \[ \text{id}_{\text{id}_A(x,y)}(f,g) \text{ interpreted as equalities } f = g \text{ for } x \Rightarrow y \]

- 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:

  \[ \text{Id}_A(x, y) \text{ interpreted as (invertible) morphisms } x \to y \]

  \[ \text{Id}_{\text{Id}_A(x, y)}(f, g) \text{ interpreted as equalities } f = g \text{ for } x \Rightarrow y \]

- Voevodsky’s model in simplicial sets takes this idea \( \omega \) steps further.
<table>
<thead>
<tr>
<th></th>
<th>The origins of univalent foundations</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Dependent type theory</td>
</tr>
<tr>
<td>3</td>
<td>Set interpretation of type theory &amp; Axiom K</td>
</tr>
<tr>
<td>4</td>
<td><strong>Simplicial set interpretation of type theory</strong></td>
</tr>
<tr>
<td>5</td>
<td>Contractible types, equivalences, function extensionality</td>
</tr>
<tr>
<td>6</td>
<td>Logic in univalent foundations</td>
</tr>
<tr>
<td>7</td>
<td>Homotopy levels</td>
</tr>
<tr>
<td>8</td>
<td>Universes and the Univalence Axiom</td>
</tr>
<tr>
<td>9</td>
<td>Equivalence Principle</td>
</tr>
<tr>
<td>10</td>
<td>Synthetic homotopy theory</td>
</tr>
</tbody>
</table>
Interpretation of identities as paths

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

- reflexivity, symmetry, transitivity
- subst

Inhabitants of \( \text{Id}(x, y) \) behave unlike equality

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

Lack of uniqueness motivates interpretation of elements of \( \text{Id}_X(x, y) \) 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

- $p : x ightsquigarrow y$
- $p^{-1} : y ightsquigarrow x$
- $r : y ightsquigarrow z$
- $p \cdot r : x ightsquigarrow z$
More operations on paths

**Transport**

\[
\text{transport : } \prod_{x,y:A} x \rightsquigarrow y \to \prod_{B:A \to \text{Type}} (B(x) \leftrightarrow B(y))
\]

**Congruence**

Given \( f : A \to B \), have

\[
\text{id}(f) : \prod_{x,y:A} x \rightsquigarrow y \to f(x) \rightsquigarrow f(y)
\]
Transport in pictures

\[ \text{transport}^B : x \rightsquigarrow y \to B(x) \to B(y) \]

\[ b : B(x) \]

\[ \text{transport}^B(p, b) : B(y) \]

\[ x \]

\[ y \]

\[ A \]

\[ p \]
Functions map paths, not just points

\[ f : A \rightarrow B \]

Exercise

Given \( f : A \rightarrow B \), construct a term of type

\[
\prod_{x,y : A} x \rightsquigarrow_A y \rightarrow f(x) \rightsquigarrow_B f(y)
\]
Paths between paths

What is a path

\[ h : p \leadsto_{x \leadsto y} q \]

between paths?
Paths between paths

What is a path between paths?

\[ h : p \rightsquigarrow_{x \rightsquigarrow y} q \]
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 \(\omega\)-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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Simpl. set interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (A, \rightsquigarrow_A, \rightsquigarrow_A, \ldots) )</td>
<td>Kan complex ( A )</td>
</tr>
<tr>
<td>( a : A )</td>
<td>( a \in A_0 )</td>
</tr>
<tr>
<td>( A \times B )</td>
<td>binary product</td>
</tr>
<tr>
<td>( A \to B )</td>
<td>space of maps</td>
</tr>
<tr>
<td>( A + B )</td>
<td>binary coproduct</td>
</tr>
<tr>
<td>( x : A \vdash B(x) )</td>
<td>fibration ( B \to A ) with fibers ( B(x) )</td>
</tr>
<tr>
<td>( \sum_{x:A} B(x) )</td>
<td>total space of fibration ( B \to A )</td>
</tr>
<tr>
<td>( \prod_{x:A} B(x) )</td>
<td>space of sections of fibration ( B \to A )</td>
</tr>
</tbody>
</table>
Outline

1. The origins of univalent foundations
2. Dependent type theory
3. Set interpretation of type theory & Axiom K
4. Simplicial set interpretation of type theory
5. **Contractible types, equivalences, function extensionality**
6. Logic in univalent foundations
7. Homotopy levels
8. Universes and the Univalence Axiom
9. Equivalence Principle
10. Synthetic homotopy theory
Contractible types

**Definition**

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

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

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.,

\[
\text{isequiv}(f) \equiv \prod_{b : B} \text{isContr} \left( \sum_{a : A} f(a) \rightsquigarrow b \right)
\]

The type of equivalences:

\[
A \simeq B \equiv \sum_{f : A \to B} \text{isequiv}(f)
\]

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 \(b, b' : B\),

\[(a, b) \mapsto (a', b') \simeq (a \mapsto a') \times (b \mapsto b')\]

• for \(a, a' : A\) and \(b : B(a)\) and \(b' : B(a')\),

\[(a, b) \mapsto (a', b') \simeq \sum_{p : a \mapsto a'} \text{transport}^B(p, b) \mapsto b'\]

The equivalences map \(\text{refl}(a, b)\) to \((\text{refl}(a), \text{refl}(b))\).
Path types of pairs

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

- for \(a, a' : A\) and \(b, b' : B\),
  \[(a,b) \leadsto (a',b') \simeq (a \leadsto a') \times (b \leadsto b')\]

- for \(a, a' : A\) and \(b : B(a)\) and \(b' : B(a')\),
  \[(a,b) \leadsto (a',b') \simeq \sum_{p:a \leadsto a'} \text{transport}^B(p,b) \leadsto b'\]

The equivalences map \(\text{refl}(a,b)\) to \((\text{refl}(a), \text{refl}(b))\).

**Exercise**

Given functions \(f, g : A \to B\), complete

\[f \leadsto g \simeq ???\]
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

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

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

**Axiom (function extensionality)**

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

is an equivalence for any $f, g$. 
Outline

1. The origins of univalent foundations
2. Dependent type theory
3. Set interpretation of type theory & Axiom K
4. Simplicial set interpretation of type theory
5. Contractible types, equivalences, function extensionality
6. **Logic in univalent foundations**
7. Homotopy levels
8. Universes and the Univalence Axiom
9. Equivalence Principle
10. Synthetic homotopy theory
Some types are propositions

<table>
<thead>
<tr>
<th>Curry–Howard</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Types are propositions.</td>
</tr>
<tr>
<td>• Terms are proofs.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Univalent logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Some types are propositions.</td>
</tr>
<tr>
<td>• Terms of those types are proofs.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Definition (Propositions in univalent type theory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type $A$ is a <strong>proposition</strong> if</td>
</tr>
<tr>
<td>$\text{isProp}(A) \equiv \prod_{x,y:A} x \rightsquigarrow y$</td>
</tr>
<tr>
<td>is inhabited.</td>
</tr>
</tbody>
</table>
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

\[ \text{Prop} \equiv \sum_{X : \text{Type}} \text{isProp}(X) \]

We want logical connectives

\[ \top, \bot : \text{Prop} \]
\[ \lor, \land, \Rightarrow : \text{Prop} \to \text{Prop} \to \text{Prop} \]
\[ \neg : \text{Prop} \to \text{Prop} \]
\[ \forall X, \exists X : (X \to \text{Prop}) \to \text{Prop} \] (binding a variable)
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

\[ A \land B \equiv A \times B \]

• 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

\[ \forall (a : A), B(a) \equiv \prod_{a:A} B(a) \]
Exercise: Find a type $T$ that is a proposition such that $T + T$ is not a proposition.

Conclusion: can not set

\[ A \lor B \equiv A + B \]

- $\Sigma_{n: \mathbb{Nat}} \text{isEven}(n)$ is the type of all even natural numbers. It is not a proposition.

Conclusion: can not set

\[ \exists (a : A), B(a) \equiv \Sigma_{a : A} B(a) \]

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 $\bar{a} : ||A||$

$$p(A) : \prod_{x,y:||A||} x \rightsquigarrow y$$

**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.
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 $f : A \rightarrow B$ and $B$ is a proposition, then $\bar{f} : ||A|| \rightarrow B$

**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 $||A|| := \prod_{P : \text{Prop}} (A \rightarrow P) \rightarrow P$ satisfies the rules.
\( \lor \) and \( \exists \) in univalent logic

\[
A \lor B \equiv \|A + B\|
\]

\[
\exists(a : A), B(a) \equiv \|\Sigma_{a : A} B(a)\|
\]

For example:

\[
isSurjective(f) \equiv \prod_{b : B} \|\Sigma_{a : A} f(a) \rightsquigarrow b\|
\]
Propositional extensionality

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

\[ \prod_{P,Q:\text{Prop}} (P \Rightarrow Q) \simeq (P \leftrightarrow Q) \]
Propositional extensionality

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

\[ 
\prod_{P,Q:\text{Prop}} (P \iff Q) \simeq (P \leftrightarrow Q) 
\]

**Axiom: propositional extensionality**

The family of maps

\[ 
\prod_{P,Q:\text{Prop}} (P \iff Q) \to (P \leftrightarrow Q) 
\]

is an equivalence pointwise.
Outline

1. The origins of univalent foundations
2. Dependent type theory
3. Set interpretation of type theory & Axiom K
4. Simplicial set interpretation of type theory
5. Contractible types, equivalences, function extensionality
6. Logic in univalent foundations
7. **Homotopy levels**
8. Universes and the Univalence Axiom
9. Equivalence Principle
10. Synthetic homotopy theory
Contractible types, propositions and sets

- **A is contractible** if we can construct a term of type
  
  $$
  \text{isContr}(A) \equiv \sum_{x:A} \prod_{y:A} y \rightsquigarrow x
  $$

- **A is a proposition** if \( \prod_{x,y:A} x \rightsquig-arrow y \) is inhabited
  
  $$
  \text{isProp}(A) \equiv \prod_{x,y:A} x \rightsquig-arrow y
  $$

- **A is a set** if, for any \( x, y : A \), the type \( x \rightsquigarrow y \) is a proposition
  
  $$
  \text{isSet}(A) \equiv \prod_{x,y:A} \text{isProp}(x \rightsquig-arrow y)
  $$
Contractible types, propositions and sets

- $A$ is **contractible** if we can construct a term of type
  \[
  \text{isContr}(A) : \equiv \sum_{x:A} \prod_{y:A} y \rightsquigarrow x
  \]

- $A$ is a **proposition** if $\prod_{x,y:A} \text{isContr}(x \rightsquigarrow y)$ is inhabited
  \[
  \text{isProp}(A) : \equiv \prod_{x,y:A} \text{isContr}(x \rightsquigarrow y)
  \]

- $A$ is a **set** if, for any $x,y:A$, the type $x \rightsquigarrow y$ is a proposition
  \[
  \text{isSet}(A) : \equiv \prod_{x,y:A} \text{isProp}(x \rightsquigarrow y)
  \]
Exercises

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

**Definition**

\[
isofhlevel : \text{Nat} \to \text{Type} \to \text{Type}
\]

\[
isofhlevel(o)(X) \equiv \text{isContr}(X)
\]

\[
isofhlevel(S(n))(X) \equiv \prod_{x,y:X} isofhlevel(n)(x \rightsquigarrow y)
\]
Homotopy level of a type

**Definition**

\[
\text{isofhlevel} : \text{Nat} \to \text{Type} \to \text{Prop}
\]

\[
\text{isofhlevel}(0)(X) \equiv \text{isContr}(X)
\]

\[
\text{isofhlevel}(S(n))(X) \equiv \prod_{x,y:X} \text{isofhlevel}(n)(x \rightsquigarrow y)
\]

**Exercise**

Show that \( \text{isofhlevel}(n)(X) \) 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

1. The origins of univalent foundations
2. Dependent type theory
3. Set interpretation of type theory & Axiom K
4. Simplicial set interpretation of type theory
5. Contractible types, equivalences, function extensionality
6. Logic in univalent foundations
7. Homotopy levels
8. **Universes and the Univalence Axiom**
9. Equivalence Principle
10. Synthetic homotopy theory
Universes

Reminder:

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

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

Fill in

$$A \rightsquigarrow_{\text{Type}} B \simeq$$
Universes

Reminder:

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

Exercise: What is the path type of two types in $\text{Type}$?

Fill in

$$A \rightsquigarrow_{\text{Type}} B \simeq (A \simeq B)$$

The type theory seen so far does not allow one to construct such an equivalence.
Voevodsky’s Univalence Axiom

Definition

\[\text{idtoequiv} : \prod_{A,B: \text{Type}} (A \rightsquigarrow B) \to (A \simeq B)\]

\[\text{idtoequiv}(A, A, \text{refl}(A)) := \text{id}_A\]

Univalence Axiom

\[\text{univalence} : \prod_{A,B: \text{Type}} \text{isequiv}(\text{idtoequiv}_{A,B})\]

Note that univalence is stated for a fixed universe \(\text{Type}\).
Consequences of the Univalence Axiom

• Function extensionality

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

• Propositional extensionality:

\[
\prod_{P,Q:\text{Prop}} (P \rightsquigarrow Q) \simeq (P \leftrightarrow Q)
\]

• Paths are isomorphisms for sets:

\[
\prod_{X,Y:\text{Set}} (X \rightsquigarrow Y) \simeq (X \simeq Y)
\]

where

\[
X \simeq Y \equiv \sum_{f:X \to Y} \sum_{g:Y \to X} \ldots
\]
Summary: Univalent Foundations

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

<table>
<thead>
<tr>
<th>Type theory</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$ type</td>
<td>space $A$</td>
</tr>
<tr>
<td>$a : A$ (term $a$ of type $A$)</td>
<td>point $a$ in space $A$</td>
</tr>
<tr>
<td>$f : A \to B$</td>
<td>map from $A$ to $B$</td>
</tr>
<tr>
<td>$p : a \rightsquigarrow b$</td>
<td>path (1-morphism) from $a$ to $b$ in $A$</td>
</tr>
<tr>
<td>$\alpha : p \rightsquigarrow_{a \rightsquigarrow b} q$</td>
<td>homotopy from $p$ to $q$ in $A$</td>
</tr>
</tbody>
</table>

- “World” of logic (propositions and proofs) given by $\text{Prop}$

- “World” of sets given by $\text{Set} \equiv \sum_{X: \text{Type}} \text{isSet}(X)$
Outline

1. The origins of univalent foundations
2. Dependent type theory
3. Set interpretation of type theory & Axiom K
4. Simplicial set interpretation of type theory
5. Contractible types, equivalences, function extensionality
6. Logic in univalent foundations
7. Homotopy levels
8. Universes and the Univalence Axiom
9. Equivalence Principle
10. Synthetic homotopy theory
Indiscernability of identicals

\[ x = y \rightarrow \forall P (P(x) \leftrightarrow P(y)) \]

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

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

\[
\text{transport}_{x,y} : (x \rightsquigarrow y) \to \prod_{B : A \to \text{Type}} (B(x) \simeq B(y))
\]

Want:

\[
\text{transport}_{x,y} : (x \equiv y) \to \prod_{B : A \to \text{Type}} (B(x) \simeq B(y))
\]

Suffices:

\[
(x \rightsquigarrow y) \simeq (x \equiv y)
\]
Transport along isomorphism

Transport revisited

\[
\text{transport}_{x,y} : (x \rightsquigarrow y) \to \prod_{B:A \to \text{Type}} (B(x) \simeq B(y))
\]

Want:

\[
\text{transport}_{x,y} : (x \cong y) \to \prod_{B:A \to \text{Type}} (B(x) \simeq B(y))
\]

Suffices:

\[
(x \rightsquigarrow y) \rightsquigarrow (x \cong y)
\]
Transport along isomorphism

Transport revisited

\[
\text{transport}_{x,y} : (x \leadsto y) \to \prod_{B:A \to \text{Type}} (B(x) \simeq B(y))
\]

Want:

\[
\text{transport}_{x,y} : (x \equiv y) \to \prod_{B:A \to \text{Type}} (B(x) \simeq B(y))
\]

Suffices:

\[
(x \leadsto y) \simeq (x \equiv y)
\]

We establish this equivalence for the example of monoids.
Monoids

Traditionally (in set theory), a monoid is a triple \((M, \mu, e)\) 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. \(M : \text{Set}\)
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. \(M : \text{Set}\)
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 \(M : \text{Set}\)?
Monoids in type theory

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

1. \(M : \text{Set}\)
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 \(M : \text{Set}\)?

Abstractly, a monoid is a (dependent) pair \((\text{data}, \text{proof})\) where

- \text{data} is a triple \((M, \mu, e)\) as above
- \text{proof} is a triple \((\alpha, \lambda, \rho)\) saying that \((\text{data})\) satisfy the usual axioms.
The type of monoids

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

Summarily:

\[
\text{Monoid} \equiv \sum_{(M:\text{Set})} \sum_{(\mu, e):\text{MonoidStr}(M)} \text{MonoidAxioms}(M, (\mu, e))
\]

Can show

\[
isProp(\text{MonoidAxioms}(M, (\mu, e)))
\]
Monoid isomorphisms

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

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

$$M \cong M' \cong (M, \mu, e) \cong (M', \mu', e')$$

$$\cong \sum_{p : M \cong M'} \text{transport}_{Y \to (Y \times Y \to Y)} (p, \mu) \cong \mu' \times \text{transport}_{Y \to Y} (p, e) \cong e'$$

$$\cong \sum_{f : M \cong M'} (f \circ \mu \circ (f^{-1} \times f^{-1}) \cong \mu') \times (f \circ e \cong e')$$

$$\cong M \cong M'$$
Transport along monoid isomorphism

We now have two ingredients:

1.\[(M \cong M') \cong (M \cong M')\]

2.\[\text{transport}_{M,M'} : (M \cong M') \to \prod_{B : \text{Monoid} \to \text{Type}} (B(M) \cong B(M'))\]

Composing these, we get

\[\text{transport}_{M,M'} : (M \cong M') \to \prod_{B : \text{Monoid} \to \text{Type}} (B(M) \cong B(M'))\]

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*?
Conjecture

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

$$(\mathcal{C} \rightsquigarrow \mathcal{D}) \rightarrow \text{AdjEquiv}(\mathcal{C}, \mathcal{D})$$

is an equivalence.
Conjecture

For categories \( \mathcal{C} \) and \( \mathcal{D} \), the canonical map

\[
(\mathcal{C} \cong \mathcal{D}) \rightarrow \text{AdjEquiv}(\mathcal{C}, \mathcal{D})
\]

is an equivalence.

Counter-example

\[
\bullet \quad \approx \quad \bullet
\]
Categories in univalent foundations

**Definition**

A category \( \mathcal{C} \) is given by

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

\[
1_a : \mathcal{C}(a, a)
\]

\[
(\circ)_{a,b,c} : \mathcal{C}(b, c) \times \mathcal{C}(a, b) \rightarrow \mathcal{C}(a, c)
\]

- axioms: unitality & associativity

\[
1 \circ f \rightsquigarrow f \quad f \circ 1 \rightsquigarrow f \quad (h \circ g) \circ f \rightsquigarrow h \circ (g \circ f)
\]
From paths to isomorphisms

**Definition (univalent category)**

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

\[
\text{idtoiso} : \prod_{a, b : \mathcal{C}_0} (a \rightsquigarrow b) \to \text{iso}(a, b)
\]

\[
\text{idtoiso}(a, a, \text{refl}(a)) \equiv 1_a
\]

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

\[
\text{idtoiso}_{a, b} : (a \rightsquigarrow b) \to \text{iso}(a, b)
\]

is an equivalence of types.
Examples of univalent categories

- Set
- Category of monoids (as proved above)
- Groups, rings, ... (Structure Identity Principle)
- Functor category $[\mathcal{C}, \mathcal{D}]$, if $\mathcal{D}$ 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 $\text{hom}(x,y) \equiv (x \rightsquigarrow y)$

- If $\mathcal{C}$ is univalent, then the category of cones of shape $F : \mathcal{I} \to \mathcal{C}$ is
  - $\rightsquigarrow$ limits (limiting cones) in a univalent category are unique up to paths
Non-univalent categories

• Any “chaotic” category $\mathcal{C}$ with $\mathcal{C}(x,y) \simeq 1$, for $\mathcal{C}_0$ not a proposition

  $\bullet \quad \bullet$  

• Any chaotic category $\mathcal{C}$ with an object $c : \mathcal{C}_0$ is **equivalent** to the terminal category $\mathbf{1}$

  $\leadsto$  a category can be equivalent to a univalent one without being univalent itself
(Adjoint) equivalence of categories

An equivalence \( \mathcal{C} \simeq \mathcal{D} \) is given by

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

\((F, G, \eta, \epsilon)\) is an adjoint equivalence if \( F \) and \( G \) form an adjunction.

Theorem

For univalent categories \( \mathcal{C} \) and \( \mathcal{D} \), the canonical map \( \mathcal{C} \to \text{AdjEquiv}(\mathcal{C}, \mathcal{D}) \) is an equivalence.
(Adjont) equivalence of categories

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

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

$(F, G, \eta, \varepsilon)$ is an adjoint equivalence if $F$ and $G$ form an adjunction.

**Theorem**

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

$$(\mathcal{C} \simeq \mathcal{D}) \to \text{AdjEquiv}(\mathcal{C}, \mathcal{D})$$

is an equivalence.
Research goal: a higher-categorical EP in UF

Prove a similar theorem for other categorical structures.

Envisioned result

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

$$(M \leadsto N) \rightarrow (M \simeq_{\mathcal{L}} N)$$

is an equivalence.

Need notions of

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

1. The origins of univalent foundations
2. Dependent type theory
3. Set interpretation of type theory & Axiom K
4. Simplicial set interpretation of type theory
5. Contractible types, equivalences, function extensionality
6. Logic in univalent foundations
7. Homotopy levels
8. Universes and the Univalence Axiom
9. Equivalence Principle
10. Synthetic homotopy theory
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 \(A : \text{Type}\) and \(a : A\).

**Definition**

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

\[
\Omega(A, a) \equiv (a \mapsto_a a, 1_a)
\]

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

\[
\Omega^{n+1}(A, a) \equiv \Omega(\Omega^n(A, a))
\]

We have operations

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

1. $||f||_2 : ||A||_2 \to ||B||_2$ is a bijection; and
2. $\pi_k(f) : \pi_k(A, a) \to \pi_k(B, fa)$ 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 $S^1$ (circle)
Coproduct

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

Introduction  $\text{inl} : A \rightarrow A + B$, $\text{inr} : B \rightarrow A + B$

Elimination  $t : A + B \vdash C(t)$

$x : A \vdash l : C(\text{inl}(x))$

$y : B \vdash r : C(\text{inr}(y))$

then

$t : A + B \vdash \text{ind}_+(l, r, t) : C(t)$

Computation  ...
The identity type of a coproduct

Would like to construct

\[
\begin{align*}
\text{inl}(a) & \rightsquigarrow \text{inl}(a') \cong a \rightsquigarrow a' \\
\text{inr}(b) & \rightsquigarrow \text{inr}(b') \cong b \rightsquigarrow b' \\
\text{inl}(a) & \rightsquigarrow \text{inr}(b) \cong \text{zero}
\end{align*}
\]

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

\[
\begin{align*}
\text{inl}(a_0) \rightsquigarrow \text{inl}(a) & \equiv a_0 \rightsquigarrow a \\
\text{inl}(a_0) \rightsquigarrow \text{inr}(b) & \equiv o
\end{align*}
\]

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

\[
l(t) \equiv r(t)
\]

and

\[
\begin{align*}
l(\text{inl}(a)) & \equiv \text{inl}(a_0) \rightsquigarrow \text{inl}(a) \\
l(\text{inr}(b)) & \equiv \text{inl}(a_0) \rightsquigarrow \text{inr}(b) \\
r(\text{inl}(a)) & \equiv a_0 \rightsquigarrow a \\
r(\text{inr}(b)) & \equiv o
\end{align*}
\]

The function \( r \) represents “codes” for the identity type to be characterized, hence is also called code.
Encode-decode for coproduct

\[
\text{decode}(t) : \text{code}(t) \rightarrow l(t)
\]

\[
\text{encode}(t) : l(t) \rightarrow \text{code}(t)
\]
Encode-decode for coproduct

\[
\begin{align*}
\text{decode}(t) &: \text{code}(t) \rightarrow l(t) \\
\text{decode}(\text{inl}(a)) &: a_0 \rightsquigarrow a \rightarrow \text{inl}(a_0) \rightsquigarrow \text{inl}(a) \\
\text{encode}(t) &: l(t) \rightarrow \text{code}(t)
\end{align*}
\]
Encode-decode for coproduct

\[
\begin{align*}
\text{decode}(t) : \text{code}(t) & \to l(t) \\
\text{decode}(\text{inl}(a)) : a_0 \leadsto a & \to \text{inl}(a_0) \leadsto \text{inl}(a) \\
\text{decode}(\text{inr}(b)) : o \to \text{inl}(a_0) & \leadsto \text{inr}(b) \\
\text{encode}(t) : l(t) & \to \text{code}(t)
\end{align*}
\]
Encode-decode for coproduct

\[ \text{decode}(t) : \text{code}(t) \to l(t) \]
\[ \text{decode} \left( \text{inl}(a) \right) : a_0 \leadsto a \to \text{inl}(a_0) \leadsto \text{inl}(a) \]
\[ \text{decode} \left( \text{inr}(b) \right) : o \to \text{inl}(a_0) \leadsto \text{inr}(b) \]
\[ \text{encode}(t) : \text{inl}(a_0) \leadsto t \to \text{code}(t) \]
Encode-decode for coproduct

\[
\begin{align*}
\text{decode}(t) &: \text{code}(t) \rightarrow l(t) \\
\text{decode}(\text{inl}(a)) &: a_{o} \rightsquigarrow a \rightarrow \text{inl}(a_{o}) \rightsquigarrow \text{inl}(a) \\
\text{decode}(\text{inr}(b)) &: o \rightarrow \text{inl}(a_{o}) \rightsquigarrow \text{inr}(b) \\
\text{encode}(t) &: \text{inl}(a_{o}) \rightsquigarrow t \rightarrow \text{code}(t) \\
\text{encode}(t) &: p \mapsto \text{transport}^{\text{code}}(p, \text{refl}(\text{inl}(a_{o})))
\end{align*}
\]
Encode-decode for coproduct

\[
\begin{align*}
\text{decode}(t) : \text{code}(t) & \rightarrow l(t) \\
\text{decode}(\text{inl}(a)) : a_o & \rightsquigarrow a \rightarrow \text{inl}(a_o) \rightsquigarrow \text{inl}(a) \\
\text{decode}(\text{inr}(b)) : o & \rightarrow \text{inl}(a_o) \rightsquigarrow \text{inr}(b) \\
\text{encode}(t) : \text{inl}(a_o) & \rightsquigarrow t \rightarrow \text{code}(t) \\
\text{encode}(t) : p & \rightsquigarrow \text{transport}^{\text{code}}(p, \text{refl}(\text{inl}(a_o)))
\end{align*}
\]

Then show that
\[
\text{encode}(t) \circ \text{decode}(t) \rightsquigarrow 1
\]
by induction on \( t \) (and then path induction in one case) and
\[
\text{decode}(t) \circ \text{encode}(t) \rightsquigarrow 1
\]
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  $S^1$ is a type

Introduction  $\text{base} : S^1$

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

Elimination  $b : X$

$l : b \rightsquigarrow b$

yields

$\text{rec}_{S^1}(b, l) : S^1 \rightarrow X$

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

$\text{rec}_{S^1}(b, l)(\text{loop}) \rightsquigarrow l$
The loop space of the circle

**Problem**

Construct an equivalence

\[ \text{base} \rightsquigarrow \text{base} \cong \mathbb{Z} \]

**Construction:** We define

\[ l : S^1 \rightarrow \text{Type} \]

\[ l : x \mapsto \text{base} \rightsquigarrow x \]

\[ \text{code} : S^1 \rightarrow \text{Type} \]

\[ \text{code} : \text{base} \mapsto \mathbb{Z} \]

\[ \text{code} : \text{loop} \mapsto \text{univalence}(+1 : \mathbb{Z} \cong \mathbb{Z}) : \mathbb{Z} \rightsquigarrow \mathbb{Z} \]
Encode-decode for $S^1$

encode($x$) : $l(x) \rightarrow \text{code}(x)$

decode($x$) : code($t$) $\rightarrow l(t)$
Encode-decode for $S^1$

\[
\text{encode}(x) : \text{base} \rightsquigarrow x \to \text{code}(x)
\]

\[
\text{decode}(x) : \text{code}(t) \to l(t)
\]
Encode-decode for $S^1$

\[
\begin{align*}
\text{encode}(x) & : \text{base} \rightsquigarrow x \rightarrow \text{code}(x) \\
\text{encode}(x) & : p \mapsto \text{transport}^{\text{code}}(p, \text{refl}(\text{base})) \\
\text{decode}(x) & : \text{code}(t) \rightarrow l(t)
\end{align*}
\]
Encode-decode for $S^1$

\[
\begin{align*}
\text{encode}(x) & : \text{base} \rightsquigarrow x \rightarrow \text{code}(x) \\
\text{encode}(x) & : p \mapsto \text{transport}^{\text{code}}(p, \text{refl(base)}) \\
\text{decode}(x) & : \text{code}(t) \rightarrow l(t) \\
\text{decode(base)} & : \text{code}(\text{base}) \rightarrow l(\text{base})
\end{align*}
\]
Encode-decode for $S^1$

encode($x$) : base $\rightsquigarrow$ x $\rightarrow$ code($x$)

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

decode($x$) : code(t) $\rightarrow$ l(t)

decode(base) : $\mathbb{Z}$ $\rightarrow$ base $\rightsquigarrow$ base
Encode-decode for $S^1$

encode$(x) : \text{base} \rightsquigarrow x \rightarrow \text{code}(x)$

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

decode$(x) : \text{code}(t) \rightarrow l(t)$

decode$(\text{base}) : \mathbb{Z} \rightarrow \text{base} \rightsquigarrow \text{base}$

decode$(\text{base}) : \mathbb{Z} \mapsto \text{loop}^\mathbb{Z}$
Encode-decode for $S^1$

$$
\text{encode}(x) : \text{base} \leadsto x \rightarrow \text{code}(x)
$$
$$
\text{encode}(x) : p \mapsto \text{transport}^{\text{code}}(p, \text{refl}(\text{base}))
$$
$$
\text{decode}(x) : \text{code}(t) \rightarrow l(t)
$$
$$
\text{decode}(\text{base}) : \mathbb{Z} \rightarrow \text{base} \leadsto \text{base}
$$
$$
\text{decode}(\text{base}) : \mathbb{Z} \mapsto \text{loop}^{\mathbb{Z}}
$$
$$
\text{decode}(\text{loop}) : \text{transport}^P(\text{loop}, \text{decode}(\text{base})) \leadsto \text{decode}(\text{base})
$$
$$
\text{decode}(\text{loop}) \equiv \ldots
$$

with $P(x') \equiv x' \mapsto \text{code}(x') \rightarrow \text{base} \leadsto x'$
Encode-decode for $S^1$

Lemma

$$\text{encode}(x) \circ \text{decode}(x) \leadsto 1$$

(by circle induction on $x$)

and

$$\text{decode}(x) \circ \text{encode}(x) \leadsto 1$$

(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*
- Coquand & Danielsson, Isomorphism is equality
- Circle in UF: https://groups.google.com/d/msg/HomotopyTypeTheory/DgeC0l9mbtg/t67GfvcLBQAJ
- Hofmann & Streicher, The groupoid model of type theory