Homotopy Type Theory

Steve Awodey
Carnegie Mellon University

6th Hari V Sahasrabuddhe
Inflections in Computing
Indian Institute of Technology Kanpur
14 January 2015

Overview

- *Homotopy Type Theory* is a recently discovered connection between Logic and Topology.

Overview

- *Homotopy Type Theory* is a recently discovered connection between Logic and Topology.
- It is based on an interpretation of intensional Martin-Löf type theory into homotopy theory.

Overview

- *Homotopy Type Theory* is a recently discovered connection between Logic and Topology.
- It is based on an interpretation of intensional Martin-Löf type theory into homotopy theory.
- *Univalent Foundations* is an ambitious new program of foundations of mathematics based on HoTT.

Overview

- *Homotopy Type Theory* is a recently discovered connection between Logic and Topology.
- It is based on an interpretation of intensional Martin-Löf type theory into homotopy theory.
- *Univalent Foundations* is an ambitious new program of foundations of mathematics based on HoTT.
- New constructions based on homotopical intuitions are added as Higher Inductive Types, providing many classical spaces, quotient types, truncations, etc.

Overview

- *Homotopy Type Theory* is a recently discovered connection between Logic and Topology.
- It is based on an interpretation of intensional Martin-Löf type theory into homotopy theory.
- *Univalent Foundations* is an ambitious new program of foundations of mathematics based on HoTT.
- New constructions based on homotopical intuitions are added as Higher Inductive Types, providing many classical spaces, quotient types, truncations, etc.
- The new Univalence Axiom is also added. It implies that isomorphic structures are equal, in a certain sense.

Overview

- And a new “synthetic” style of axiomatics is used, simplifying and shortening many proofs.

Overview

- And a new "synthetic" style of axiomatics is used, simplifying and shortening many proofs.
- A large amount of classical mathematics has been developed in this new system: basic homotopy theory, higher category theory, analysis, algebra, cumulative hierarchy of set theory, ...

Overview

- And a new "synthetic" style of axiomatics is used, simplifying and shortening many proofs.
- A large amount of classical mathematics has been developed in this new system: basic homotopy theory, higher category theory, analysis, algebra, cumulative hierarchy of set theory, ...
- Proofs are formalized and verified in computerized proof assistants (e.g. Coq and Agda).

Overview

- And a new "synthetic" style of axiomatics is used, simplifying and shortening many proofs.
- A large amount of classical mathematics has been developed in this new system: basic homotopy theory, higher category theory, analysis, algebra, cumulative hierarchy of set theory, ...
- Proofs are formalized and verified in computerized proof assistants (e.g. Coq and Agda).
- Applications to software verification are current work in progress.

Overview

- And a new "synthetic" style of axiomatics is used, simplifying and shortening many proofs.
- A large amount of classical mathematics has been developed in this new system: basic homotopy theory, higher category theory, analysis, algebra, cumulative hierarchy of set theory, ...
- Proofs are formalized and verified in computerized proof assistants (e.g. Coq and Agda).
- Applications to software verification are current work in progress.
- There is a comprehensive book containing the informal development, which was written at a year-long special research program at the Institute for Advanced Study in Princeton.

Type theory

Martin-Löf constructive type theory consists of:

Type theory

Martin-Löf constructive type theory consists of:

- **Types**: $X, Y, \ldots, A \times B,\ A \rightarrow B, \ldots$

Type theory

Martin-Löf constructive type theory consists of:

- **Types**: $X, Y, \ldots, A \times B,\ A \to B, \ldots$
- **Terms**: $x : A,\ b : B,\ \langle a, b \rangle,\ \lambda x.b(x), \ldots$

Type theory

Martin-Löf constructive type theory consists of:

- **Types**: $X, Y, \ldots, A \times B,\ A \to B, \ldots$
- **Terms**: $x : A,\ b : B,\ \langle a, b \rangle,\ \lambda x.b(x), \ldots$
- **Dependent Types**: $x : A \vdash B(x)$

Type theory

Martin-Löf constructive type theory consists of:

- **Types**: $X, Y, \ldots, A \times B,\ A \to B, \ldots$
- **Terms**: $x : A,\ b : B,\ \langle a, b \rangle,\ \lambda x.b(x), \ldots$
- **Dependent Types**: $x : A \vdash B(x)$
  - $\sum_{x:A} B(x)$
  - $\prod_{x:A} B(x)$

Type theory

Martin-Löf constructive type theory consists of:

- **Types**: $X, Y, \ldots, A \times B,\ A \to B, \ldots$
- **Terms**: $x : A,\ b : B,\ \langle a, b \rangle,\ \lambda x.b(x), \ldots$
- **Dependent Types**: $x : A \vdash B(x)$
  - $\sum_{x:A} B(x)$
  - $\prod_{x:A} B(x)$
- **Equations** $s = t : A$

Type theory

Martin-Löf constructive type theory consists of:

- **Types**: $X, Y, \ldots, A \times B,\ A \to B, \ldots$
- **Terms**: $x : A,\ b : B,\ \langle a, b \rangle,\ \lambda x.b(x), \ldots$
- **Dependent Types**: $x : A \vdash B(x)$
  - $\sum_{x:A} B(x)$
  - $\prod_{x:A} B(x)$
- **Equations** $s = t : A$

Formal calculus of typed terms and equations, presented as a deductive system by rules of inference.
Intended as a foundation for constructive mathematics, but now used also in the theory of programming languages and as the basis of computational proof assistants.

Propositions as Types

The system has a dual interpretation:

- once as **mathematical** objects: types are “sets” and their terms are “elements”, which are being constructed,

Propositions as Types

The system has a dual interpretation:

- once as **mathematical** objects: types are “sets” and their terms are “elements”, which are being constructed,
- once as **logical** objects: types are “propositions” and their terms are “proofs”, which are being derived.

Propositions as Types

The system has a dual interpretation:

- once as **mathematical** objects: types are "sets" and their terms are "elements", which are being constructed,
- once as **logical** objects: types are "propositions" and their terms are "proofs", which are being derived.

This is known as the **Curry-Howard correspondence**:

0	1	$A + B$	$A \times B$	$A \rightarrow B$	$\sum_{x:A} B(x)$	$\prod_{x:A} B(x)$
$\bot$	$\top$	$A \vee B$	$A \wedge B$	$A \Rightarrow B$	$\exists_{x:A} B(x)$	$\forall_{x:A} B(x)$

Propositions as Types

The system has a dual interpretation:

- once as **mathematical** objects: types are "sets" and their terms are "elements", which are being constructed,
- once as **logical** objects: types are "propositions" and their terms are "proofs", which are being derived.

This is known as the **Curry-Howard correspondence**:

0	1	$A + B$	$A \times B$	$A \to B$	$\sum_{x:A} B(x)$	$\prod_{x:A} B(x)$
$\bot$	$\top$	$A \vee B$	$A \wedge B$	$A \Rightarrow B$	$\exists_{x:A} B(x)$	$\forall_{x:A} B(x)$

Gives the system its **constructive character**.

Identity types

It's natural to add a primitive relation of **identity** between any terms of the same type:

$$x, y : A \vdash \mathtt{Id}_A(x, y)$$

Logically this is the proposition "x is identical to y".

Identity types

It's natural to add a primitive relation of **identity** between any terms of the same type:

$$x, y : A \vdash \mathtt{Id}_A(x, y)$$

Logically this is the proposition "x is identical to y".
But what is it **mathematically**?

Identity types

It's natural to add a primitive relation of **identity** between any terms of the same type:

$$x, y : A \vdash \mathtt{Id}_A(x, y)$$

Logically this is the proposition "x is identical to y".
But what is it **mathematically**?

The **introduction** rule says that $a : A$ is always identical to itself:

$$\mathtt{r}(a) : \mathtt{Id}_A(a, a)$$

Identity types

It's natural to add a primitive relation of **identity** between any terms of the same type:

$$x, y : A \vdash \mathtt{Id}_A(x, y)$$

Logically this is the proposition "x is identical to y".
But what is it **mathematically**?

The **introduction** rule says that $a : A$ is always identical to itself:

$$\mathtt{r}(a) : \mathtt{Id}_A(a, a)$$

The **elimination** rule is a form of "Lawvere's law":

$$\frac{c : \mathtt{Id}_A(a, b) \qquad x : A \ \vdash \ d(x) : R\big(x, x, \mathtt{r}(x)\big)}{\mathtt{J}_d(a, b, c) : R(a, b, c)}$$

Identity types

It's natural to add a primitive relation of **identity** between any terms of the same type:

$$x, y : A \vdash \mathtt{Id}_A(x, y)$$

Logically this is the proposition "x is identical to y".
But what is it **mathematically**?

The **introduction** rule says that $a : A$ is always identical to itself:

$$\mathtt{r}(a) : \mathtt{Id}_A(a, a)$$

The **elimination** rule is a form of "Lawvere's law":

$$\frac{c : \mathtt{Id}_A(a, b) \qquad x : A \vdash d(x) : R\big(x, x, \mathtt{r}(x)\big)}{\mathtt{J}_d(a, b, c) : R(a, b, c)}$$

Schematically:

$$\text{“ } a = b \ \& \ R(x, x) \ \Rightarrow \ R(a, b) \text{ ”}$$

Intensionality

The rules are such that if a and b are **equal** as terms:

$$a = b$$

then they are also logically **identical**:

$$t : \mathtt{Id}_A(a, b) \quad \text{(for some } t\text{).}$$

Intensionality

The rules are such that if a and b are **equal** as terms:

$$a = b$$

then they are also logically **identical**:

$$t : \mathtt{Id}_A(a, b) \quad \text{(for some } t\text{)}.$$

- But the converse is not true — this is called **intensionality**.

Intensionality

The rules are such that if a and b are **equal** as terms:

$$a = b$$

then they are also logically **identical**:

$$t : \mathtt{Id}_A(a, b) \quad \text{(for some } t\text{)}.$$

- But the converse is not true — this is called **intensionality**.
- Terms that are identified logically may nonetheless remain distinct syntactically — e.g. different expressions may determine "the same" function.

Intensionality

The rules are such that if a and b are **equal** as terms:

$$a = b$$

then they are also logically **identical**:

$$t : \mathtt{Id}_A(a, b) \quad \text{(for some } t\text{)}.$$

- But the converse is not true — this is called **intensionality**.
- Terms that are identified logically may nonetheless remain distinct syntactically — e.g. different expressions may determine "the same" function.
- Allowing such distinctions gives the system good computational and proof-theoretic properties.

Intensionality

The rules are such that if a and b are **equal** as terms:

$$a = b$$

then they are also logically **identical**:

$$t : \mathtt{Id}_A(a, b) \quad \text{(for some } t\text{)}.$$

- But the converse is not true — this is called **intensionality**.
- Terms that are identified logically may nonetheless remain distinct syntactically — e.g. different expressions may determine "the same" function.
- Allowing such distinctions gives the system good computational and proof-theoretic properties.
- It also gives rise to a structure of great combinatorial complexity.

The homotopy interpretation (Awodey-Warren)

Suppose we have terms of ascending identity types:

$$\begin{aligned} a,\ b &: A \\ p,\ q &: \mathtt{Id}_A(a, b) \\ \alpha,\ \beta &: \mathtt{Id}_{\mathtt{Id}_A(a,b)}(p, q) \\ \ldots &: \mathtt{Id}_{\mathtt{Id}_{\mathtt{Id}_{\ldots}}}(\ldots) \end{aligned}$$

The homotopy interpretation (Awodey-Warren)

Suppose we have terms of ascending identity types:

$$
\begin{aligned}
a,\ \ b &: A \\
p,\ \ q &: \mathtt{Id}_A(a,b) \\
\alpha,\ \ \beta &: \mathtt{Id}_{\mathtt{Id}_A(a,b)}(p,q) \\
\ldots &: \mathtt{Id}_{\mathtt{Id}_{\mathtt{Id}_{\ldots}}}(\ldots)
\end{aligned}
$$

Consider the following interpretation:

$$
\begin{array}{rcl}
\text{Types} & \rightsquigarrow & \text{Spaces} \\
\text{Terms} & \rightsquigarrow & \text{Maps} \\
a : A & \rightsquigarrow & \text{Points } a : 1 \to A \\
p : \mathtt{Id}_A(a,b) & \rightsquigarrow & \text{Paths } p : a \Rightarrow b \\
\alpha : \mathtt{Id}_{\mathtt{Id}_A(a,b)}(p,q) & \rightsquigarrow & \text{Homotopies } \alpha : p \Rrightarrow q \\
\vdots & &
\end{array}
$$

The homotopy interpretation: Type dependency

We still need to interpret dependent types $x : A \vdash B(x)$.

The homotopy interpretation: Type dependency

We still need to interpret dependent types $x : A \vdash B(x)$.
The identity rules imply the following:

The homotopy interpretation: Type dependency

We still need to interpret dependent types $x : A \vdash B(x)$.
The identity rules imply the following:

$$\frac{p : \mathtt{Id}_A(a, a') \qquad b : B(a)}{p_* b : B(a')}$$

The homotopy interpretation: Type dependency

We still need to interpret dependent types $x : A \vdash B(x)$.
The identity rules imply the following:

$$\frac{p : \mathtt{Id}_A(a, a') \qquad b : B(a)}{p_* b : B(a')}$$

Logically, this just says "$a = a'$ & $B(a) \Rightarrow B(a')$".

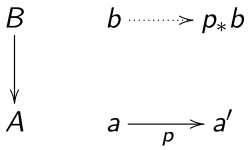
The homotopy interpretation: Type dependency

We still need to interpret dependent types $x : A \vdash B(x)$.
The identity rules imply the following:

$$\frac{p : \mathtt{Id}_A(a, a') \qquad b : B(a)}{p_* b : B(a')}$$

Logically, this just says "$a = a'$ & $B(a) \Rightarrow B(a')$".

But topologically, it is a familiar **lifting property**:



The homotopy interpretation: Type dependency

We still need to interpret dependent types $x : A \vdash B(x)$.
The identity rules imply the following:

$$\frac{p : \mathtt{Id}_A(a, a') \qquad b : B(a)}{p_* b : B(a')}$$

Logically, this just says "$a = a'$ & $B(a) \Rightarrow B(a')$".

But topologically, it is a familiar **lifting property**:

$$\begin{array}{ccc} B & \quad & b \cdots\cdots\!\!\!\!> p_* b \\ \Big\downarrow & & \\ A & & a \xrightarrow[p]{\qquad} a' \end{array}$$

This is the notion of a "fibration" of spaces.

The homotopy interpretation: Type dependency

Thus we continue the homotopy interpretation as follows:

$$\text{Dependent types} \quad x : A \vdash B(x) \quad \rightsquigarrow \quad \text{Fibrations} \quad \begin{array}{c} B \\ \downarrow \\ A \end{array}$$

The homotopy interpretation: Type dependency

Thus we continue the homotopy interpretation as follows:

$$\text{Dependent types} \quad x : A \vdash B(x) \quad \rightsquigarrow \quad \text{Fibrations} \quad \begin{array}{c} B \\ \downarrow \\ A \end{array}$$

The type $B(a)$ is the fiber of $B \to A$ over the point $a : A$

$$\begin{array}{ccc} B(a) & \longrightarrow & B \\ \downarrow & & \downarrow \\ 1 & \xrightarrow[a]{} & A. \end{array}$$

The homotopy interpretation: Identity types

To interpret the identity type $x, y : A \vdash \mathtt{Id}_A(x, y)$, we thus require a fibration over $A \times A$.

The homotopy interpretation: Identity types

To interpret the identity type $x, y : A \vdash \mathtt{Id}_A(x, y)$, we thus require a fibration over $A \times A$.
Take the space A^I of all paths in A:

$$\text{Identity type} \quad x, y : A \vdash \mathtt{Id}_A(x, y) \quad \rightsquigarrow \quad \text{Path space} \quad \begin{array}{c} A^I \\ \downarrow \\ A \times A \end{array}$$

The homotopy interpretation: Identity types

To interpret the identity type $x, y : A \vdash \mathtt{Id}_A(x, y)$, we thus require a fibration over $A \times A$.
Take the space A^I of all paths in A:

$$\text{Identity type} \quad x, y : A \vdash \mathtt{Id}_A(x, y) \quad \rightsquigarrow \quad \text{Path space} \quad \begin{array}{c} A^I \\ \downarrow \\ A \times A \end{array}$$

The fiber $\mathtt{Id}_A(a, b)$ over a point $(a, b) \in A \times A$ is the space of paths from a to b in A.

$$\begin{array}{ccc} \mathtt{Id}_A(a, b) & \longrightarrow & A^I \\ \downarrow & & \downarrow \\ 1 & \xrightarrow[(a,b)]{} & A \times A. \end{array}$$

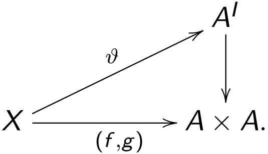
The homotopy interpretation: Identity types

The path space A^I classifies homotopies $\vartheta : f \Rightarrow g$ between maps $f, g : X \to A$,

$$\begin{array}{ccc} & & A^I \\ & \overset{\vartheta}{\nearrow} & \downarrow \\ X & \xrightarrow[(f,g)]{} & A \times A. \end{array}$$

The homotopy interpretation: Identity types

The path space A^I classifies homotopies $\vartheta : f \Rightarrow g$ between maps $f, g : X \to A$,



So given any terms $x : X \vdash f, g : A$, an identity term

$$x : X \vdash \vartheta : \mathtt{Id}_A(f, g)$$

is interpreted as a **homotopy** between f and g.

The homotopy interpretation: Summary

This takes the familiar **topological interpretation** of the *simply-typed* λ-calculus:

$$\text{types} \rightsquigarrow \text{spaces}$$
$$\text{terms} \rightsquigarrow \text{continuous functions}$$

The homotopy interpretation: Summary

This takes the familiar **topological interpretation** of the *simply-typed* λ-calculus:

$$\text{types} \rightsquigarrow \text{spaces}$$
$$\text{terms} \rightsquigarrow \text{continuous functions}$$

and extends it to *dependently typed* λ-calculus with `Id`-types, via the **basic idea**:

$$p : \mathtt{Id}_X(a, b) \;\; \rightsquigarrow \;\; p \text{ is a path from point } a \text{ to point } b \text{ in } X$$

The homotopy interpretation: Summary

This takes the familiar **topological interpretation** of the *simply-typed* λ-calculus:

$$\text{types} \rightsquigarrow \text{spaces}$$
$$\text{terms} \rightsquigarrow \text{continuous functions}$$

and extends it to *dependently typed* λ-calculus with `Id`-types, via the **basic idea**:

$$p : \mathtt{Id}_X(a, b) \;\; \rightsquigarrow \;\; p \text{ is a path from point } a \text{ to point } b \text{ in } X$$

This forces:

- *dependent types to be fibrations*,

The homotopy interpretation: Summary

This takes the familiar **topological interpretation** of the *simply-typed* λ-calculus:

$$\begin{aligned} \text{types} &\rightsquigarrow \text{spaces} \\ \text{terms} &\rightsquigarrow \text{continuous functions} \end{aligned}$$

and extends it to *dependently typed* λ-calculus with `Id`-types, via the **basic idea**:

$$p : \mathtt{Id}_X(a, b) \;\rightsquigarrow\; p \text{ is a path from point } a \text{ to point } b \text{ in } X$$

This forces:

- *dependent types to be fibrations*,
- `Id`*-types to be path spaces*,

The homotopy interpretation: Summary

This takes the familiar **topological interpretation** of the *simply-typed* λ-calculus:

$$\text{types} \rightsquigarrow \text{spaces}$$

$$\text{terms} \rightsquigarrow \text{continuous functions}$$

and extends it to *dependently typed* λ-calculus with `Id`-types, via the **basic idea**:

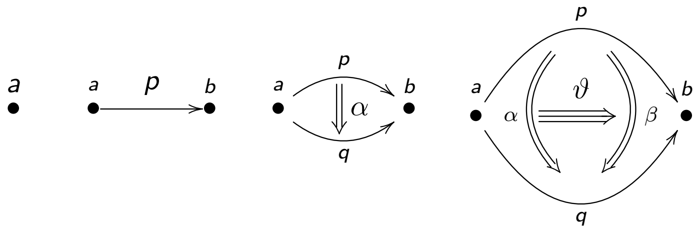
$$p : \mathtt{Id}_X(a, b) \quad \rightsquigarrow \quad p \text{ is a path from point } a \text{ to point } b \text{ in } X$$

This forces:

- *dependent types to be fibrations*,
- `Id`*-types to be path spaces*,
- *homotopic maps to be identical*.

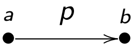
The fundamental groupoid of a type (Hofmann-Streicher)

Like path spaces in topology, identity types endow each type in the system with the structure of a (higher-) groupoid:



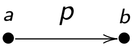
Fundamental groupoids

As in topology, the terms of order 0 and 1, ("points" and "paths") bear the structure of a **groupoid**.



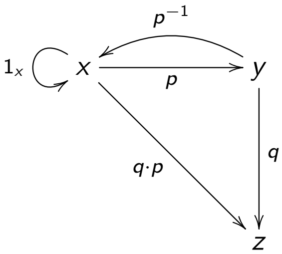
Fundamental groupoids

As in topology, the terms of order 0 and 1, ("points" and "paths") bear the structure of a **groupoid**.



Definition

A *groupoid* is a category in which every arrow has an inverse.



The fundamental groupoid of a type

The laws of identity are then the **groupoid operations**:

$r : \mathtt{Id}(a, a)$	reflexivity	$a \rightarrow a$
$s : \mathtt{Id}(a, b) \rightarrow \mathtt{Id}(b, a)$	symmetry	$a \leftrightarrows b$
$t : \mathtt{Id}(a, b) \times \mathtt{Id}(b, c) \rightarrow \mathtt{Id}(a, c)$	transitivity	$a \rightarrow b \rightarrow c$

The fundamental groupoid of a type

But also just as in topology, the **groupoid equations** of associativity, inverse, and unit:

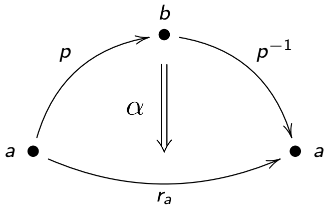
$$p \cdot (q \cdot r) = (p \cdot q) \cdot r$$
$$p^{-1} \cdot p = 1 = p \cdot p^{-1}$$
$$1 \cdot p = p = p \cdot 1$$

do not hold strictly, but only "up to homotopy".

The fundamental groupoid of a type

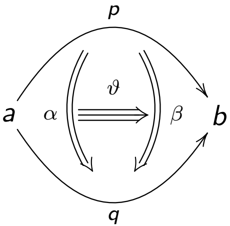
This means they are witnessed by terms of the next higher order:

$$\alpha : \mathtt{Id}_{\mathtt{Id}}\left(p^{-1} \cdot p, r_a\right)$$



The fundamental groupoid of a type

In this way, each type in the system is endowed with the structure of an "∞-groupoid", with terms, identities between terms, identities between identities, ...



Homotopy n-types (Voevodsky)

The universe of all types is naturally stratified by "homotopical dimension" or "truncation level": the level at which the fundamental groupoid becomes trivial.

Homotopy n-types (Voevodsky)

The universe of all types is naturally stratified by "homotopical dimension" or "truncation level": the level at which the fundamental groupoid becomes trivial.

A type X is called:

contractible iff $\sum_{x:X} \prod_{y:X} \mathtt{Id}_X(x, y)$,
Such a type has *essentially* one term.

Homotopy *n*-types (Voevodsky)

The universe of all types is naturally stratified by "homotopical dimension" or "truncation level": the level at which the fundamental groupoid becomes trivial.

A type X is called:

contractible iff $\sum_{x:X} \prod_{y:X} \mathtt{Id}_X(x, y)$,
Such a type has *essentially* one term.

A type X is called a:

proposition iff $\prod_{x,y:X} \text{Contr}(\mathtt{Id}_X(x, y))$,
Such a type has at most one term.

Homotopy n-types (Voevodsky)

The universe of all types is naturally stratified by "homotopical dimension" or "truncation level": the level at which the fundamental groupoid becomes trivial.

A type X is called:

contractible iff $\sum_{x:X} \prod_{y:X} \mathtt{Id}_X(x, y)$,
Such a type has *essentially* one term.

A type X is called a:

proposition iff $\prod_{x,y:X}$ Contr$(\mathtt{Id}_X(x, y))$,
Such a type has at most one term.

set iff $\prod_{x,y:X}$ Prop$(\mathtt{Id}_X(x, y))$,
Identity of terms in such a type is a proposition.

Homotopy n-types (Voevodsky)

The universe of all types is naturally stratified by "homotopical dimension" or "truncation level": the level at which the fundamental groupoid becomes trivial.

A type X is called:

contractible iff $\sum_{x:X} \prod_{y:X} \mathtt{Id}_X(x, y)$,
Such a type has *essentially* one term.

A type X is called a:

proposition iff $\prod_{x,y:X} \mathsf{Contr}(\mathtt{Id}_X(x, y))$,
Such a type has at most one term.

set iff $\prod_{x,y:X} \mathsf{Prop}(\mathtt{Id}_X(x, y))$,
Identity of terms in such a type is a proposition.

1-type iff $\prod_{x,y:X} \mathsf{Set}(\mathtt{Id}_X(x, y))$,
Identity of identity terms in such a type is a proposition.

Homotopy n-types (Voevodsky)

The universe of all types is naturally stratified by "homotopical dimension" or "truncation level": the level at which the fundamental groupoid becomes trivial.

A type X is called:

contractible iff $\sum_{x:X} \prod_{y:X} \mathtt{Id}_X(x,y)$,
Such a type has *essentially* one term.

A type X is called a:

proposition iff $\prod_{x,y:X} \mathsf{Contr}(\mathtt{Id}_X(x,y))$,
Such a type has at most one term.

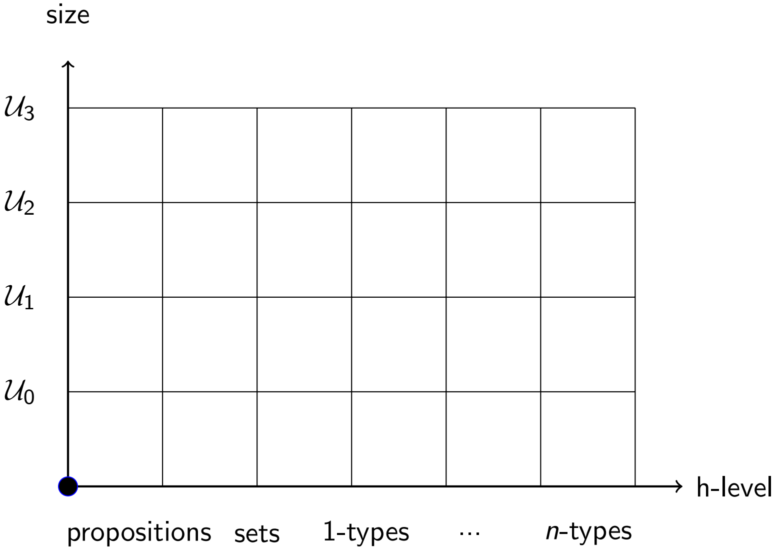
set iff $\prod_{x,y:X} \mathsf{Prop}(\mathtt{Id}_X(x,y))$,
Identity of terms in such a type is a proposition.

1-type iff $\prod_{x,y:X} \mathsf{Set}(\mathtt{Id}_X(x,y))$,
Identity of identity terms in such a type is a proposition.

(n+1)-type iff $\prod_{x,y:X} \mathsf{nType}(\mathtt{Id}_X(x,y))$.

The Hierarchy of n-Types

This gives a new view of the mathematical universe, in which some types have intrinsic higher-dimensional structure.



Machine implementation

Now one can combine:

- the representation of homotopy theory and other mathematics in constructive type theory,

Machine implementation

Now one can combine:

- the representation of homotopy theory and other mathematics in constructive type theory,
- the well-developed implementations of type theory in computational proof assistants like Coq and Agda.

Machine implementation

Now one can combine:

- the representation of homotopy theory and other mathematics in constructive type theory,
- the well-developed implementations of type theory in computational proof assistants like Coq and Agda.

This allows for computer verified proofs in homotopy theory and related fields, in addition to classical and constructive mathematics. This is being very actively pursued right now.

A computational example

A classical result states that the higher homotopy groups of a space are always abelian.

A computational example

A classical result states that the higher homotopy groups of a space are always abelian.
We can formalize this very simply in homotopy type theory:

A computational example

A classical result states that the higher homotopy groups of a space are always abelian.
We can formalize this very simply in homotopy type theory:

- the fundamental group $\pi_1(X, b)$ of a type X at basepoint $b : X$ consists of all terms of type $\mathtt{Id}_X(b, b)$.

A computational example

A classical result states that the higher homotopy groups of a space are always abelian.
We can formalize this very simply in homotopy type theory:

- the fundamental group $\pi_1(X, b)$ of a type X at basepoint $b : X$ consists of all terms of type $\mathtt{Id}_X(b, b)$.
- the second homotopy group $\pi_2(X, b)$ consists of all terms of type $\mathtt{Id}_{\mathtt{Id}_X(b,b)}(\mathtt{r}(b), \mathtt{r}(b))$.

A computational example

A classical result states that the higher homotopy groups of a space are always abelian.
We can formalize this very simply in homotopy type theory:

- the fundamental group $\pi_1(X, b)$ of a type X at basepoint $b : X$ consists of all terms of type $\mathtt{Id}_X(b, b)$.
- the second homotopy group $\pi_2(X, b)$ consists of all terms of type $\mathtt{Id}_{\mathtt{Id}_X(b,b)}(\mathtt{r}(b), \mathtt{r}(b))$.
- Each of these types has a group structure, and so the second one has *two* group structures that are compatible.

A computational example

A classical result states that the higher homotopy groups of a space are always abelian.
We can formalize this very simply in homotopy type theory:

- the fundamental group $\pi_1(X, b)$ of a type X at basepoint $b : X$ consists of all terms of type $\mathtt{Id}_X(b, b)$.
- the second homotopy group $\pi_2(X, b)$ consists of all terms of type $\mathtt{Id}_{\mathtt{Id}_X(b,b)}(\mathtt{r}(b), \mathtt{r}(b))$.
- Each of these types has a group structure, and so the second one has *two* group structures that are compatible.
- Now the Eckmann-Hilton argument shows that the two structures on $\pi_2(X, b)$ agree, and are abelian.

A computational example

A classical result states that the higher homotopy groups of a space are always abelian.
We can formalize this very simply in homotopy type theory:

- the fundamental group $\pi_1(X, b)$ of a type X at basepoint $b : X$ consists of all terms of type $\mathtt{Id}_X(b, b)$.
- the second homotopy group $\pi_2(X, b)$ consists of all terms of type $\mathtt{Id}_{\mathtt{Id}_X(b,b)}(\mathtt{r}(b), \mathtt{r}(b))$.
- Each of these types has a group structure, and so the second one has *two* group structures that are compatible.
- Now the Eckmann-Hilton argument shows that the two structures on $\pi_2(X, b)$ agree, and are abelian.

This argument can be formalized in Coq and checked by a computer. In this way, we can use the homotopical interpretation to give machine-checked proofs in homotopy theory.

A computational example

```
(** ** The 2-dimensional groupoid structure *)

(** Horizontal composition of 2-dimensional paths. *)
Definition concat2 {A} {x y z : A} {p p' : x = y} {q q' : y = z} (h : p = p') (h' : q = q')
: p @ q = p' @ q'
:= match h, h' with idpath, idpath => 1 end.

Notation ``p @@ q" := (concat2 p q)

(** 2-dimensional path inversion *)
Definition inverse2 {A : Type} {x y : A} {p q : x = y} (h : p = q) : p^ = q^
:= match h with idpath => 1 end.

(** *** Whiskering *)

Definition whiskerL {A : Type} {x y z : A} (p : x = y) {q r : y = z} (h : q = r) : p @ q = p @ r
:= 1 @@ h.

Definition whiskerR {A : Type} {x y z : A} {p q : x = y} (h : p = q) (r : y = z) : p @ r = q @ r
:= h @@ 1.

(** *** Unwhiskering, a.k.a. cancelling. *)

Lemma cancelL {A} {x y z : A} (p : x = y) (q r : y = z) : (p @ q = p @ r) -> (q = r).
Proof.
  destruct p, r. intro a. exact ((concat_1p q)^ @ a).
Defined.

Lemma cancelR {A} {x y z : A} (p q : x = y) (r : y = z) : (p @ r = q @ r) -> (p = q).
Proof.
  destruct r, p. intro a. exact (a @ concat_p1 q).
Defined.
```

```
(** Whiskering and identity paths. *)

Definition whiskerR_p1 {A : Type} {x y : A} {p q : x = y} (h : p = q) :
  (concat_p1 p) ^ @ whiskerR h 1 @ concat_p1 q = h
  :=
  match h with idpath =>
    match p with idpath =>
      1
    end end.

Definition whiskerR_1p {A : Type} {x y z : A} (p : x = y) (q : y = z) :
  whiskerR 1 q = 1 :> (p @ q = p @ q)
  :=
  match q with idpath => 1 end.

Definition whiskerL_p1 {A : Type} {x y z : A} (p : x = y) (q : y = z) :
  whiskerL p 1 = 1 :> (p @ q = p @ q)
  :=
  match q with idpath => 1 end.

Definition whiskerL_1p {A : Type} {x y : A} {p q : x = y} (h : p = q) :
  (concat_1p p) ^ @ whiskerL 1 h @ concat_1p q = h
  :=
  match h with idpath =>
    match p with idpath =>
      1
    end end.

Definition concat2_p1 {A : Type} {x y : A} {p q : x = y} (h : p = q) :
  h @@ 1 = whiskerR h 1 :> (p @ 1 = q @ 1)
  :=
  match h with idpath => 1 end.

Definition concat2_1p {A : Type} {x y : A} {p q : x = y} (h : p = q) :
  1 @@ h = whiskerL 1 h :> (1 @ p = 1 @ q)
  :=
  match h with idpath => 1 end.
```

```
(** The interchange law for concatenation. *)
Definition concat_concat2 {A : Type} {x y z : A} {p p' p'' : x = y} {q q' q'' : y = z}
  (a : p = p') (b : p' = p'') (c : q = q') (d : q' = q'') :
  (a @@ c) @ (b @@ d) = (a @ b) @@ (c @ d).
Proof.
  case d.
  case c.
  case b.
  case a.
  reflexivity.
Defined.

(** The interchange law for whiskering.  Special case of [concat_concat2]. *)
Definition concat_whisker {A} {x y z : A} (p p' : x = y) (q q' : y = z) (a : p = p') (b : q = q') :
  (whiskerR a q) @ (whiskerL p' b) = (whiskerL p b) @ (whiskerR a q')
  :=
  match b with
    idpath =>
    match a with idpath =>
      (concat_1p _)^
    end
  end.

(** Structure corresponding to the coherence equations of a bicategory. *)

(** The "pentagonator": the 3-cell witnessing the associativity pentagon. *)
Definition pentagon {A : Type} {v w x y z : A} (p : v = w) (q : w = x) (r : x = y) (s : y = z)
  : whiskerL p (concat_p_pp q r s)
      @ concat_p_pp p (q@r) s
      @ whiskerR (concat_p_pp p q r) s
  = concat_p_pp p q (r@s) @ concat_p_pp (p@q) r s.
Proof.
  case p, q, r, s.  reflexivity.
Defined.
```

```
(** The 3-cell witnessing the left unit triangle. *)
Definition triangulator {A : Type} {x y z : A} (p : x = y) (q : y = z)
  : concat_p_pp p 1 q @ whiskerR (concat_p1 p) q
  = whiskerL p (concat_1p q).
Proof.
  case p, q.  reflexivity.
Defined.

(** The Eckmann-Hilton argument *)
Definition eckmann_hilton {A : Type} {x:A} (p q : 1 = 1 :> (x = x)) : p @ q = q @ p :=
  (whiskerR_p1 p @@ whiskerL_1p q)^
  @ (concat_p1 _ @@ concat_p1 _)
  @ (concat_1p _ @@ concat_1p _)
  @ (concat_whisker _ _ _ _ p q)
  @ (concat_1p _ @@ concat_1p _)^
  @ (concat_p1 _ @@ concat_p1 _)^
  @ (whiskerL_1p q @@ whiskerR_p1 p).
```

Formalization of mathematics

- The idea of logical foundations of math has great conceptual and philosophical interest, but in the past this was too lengthy and cumbersome to be of much use.

Formalization of mathematics

- The idea of logical foundations of math has great conceptual and philosophical interest, but in the past this was too lengthy and cumbersome to be of much use.
- Explicit formalization of math is finally *feasible*, because computers can now take over what was once too tedious or complicated to be done by hand.

Formalization of mathematics

- The idea of logical foundations of math has great conceptual and philosophical interest, but in the past this was too lengthy and cumbersome to be of much use.
- Explicit formalization of math is finally *feasible*, because computers can now take over what was once too tedious or complicated to be done by hand.
- Formalization can provide a *practical tool* for working mathematicians and scientists: increased certainty and precision, supports remote collaborative work, cumulativity of results, searchable library of code, ... perhaps mathematics will one day be fully formalized.

Formalization of mathematics

- The idea of logical foundations of math has great conceptual and philosophical interest, but in the past this was too lengthy and cumbersome to be of much use.
- Explicit formalization of math is finally *feasible*, because computers can now take over what was once too tedious or complicated to be done by hand.
- Formalization can provide a *practical tool* for working mathematicians and scientists: increased certainty and precision, supports remote collaborative work, cumulativity of results, searchable library of code, … perhaps mathematics will one day be fully formalized.
- UF uses a "synthetic" method involving high-level axiomatics and structural descriptions. Allows for shorter, more abstract proofs that are closer to mathematical practice than the "analytic" method of ZFC.

Formalization of mathematics

- The idea of logical foundations of math has great conceptual and philosophical interest, but in the past this was too lengthy and cumbersome to be of much use.
- Explicit formalization of math is finally *feasible*, because computers can now take over what was once too tedious or complicated to be done by hand.
- Formalization can provide a *practical tool* for working mathematicians and scientists: increased certainty and precision, supports remote collaborative work, cumulativity of results, searchable library of code, ... perhaps mathematics will one day be fully formalized.
- UF uses a "synthetic" method involving high-level axiomatics and structural descriptions. Allows for shorter, more abstract proofs that are closer to mathematical practice than the "analytic" method of ZFC.
- Software verification should also benefit from higher dimensional methods: current work in progress at CMU.

Homotopy Type Theory: Summary

- Under the new homotopical interpretation, constructive type theory provides a “logic of homotopy”.

Homotopy Type Theory: Summary

- Under the new homotopical interpretation, constructive type theory provides a "logic of homotopy".
- Logical methods capture some homotopically significant structures: e.g. the fundamental ∞-groupoid of a space is a *logical construction*, and the notion of an n-type is *logically definable*.

Homotopy Type Theory: Summary

- Under the new homotopical interpretation, constructive type theory provides a "logic of homotopy".
- Logical methods capture some homotopically significant structures: e.g. the fundamental ∞-groupoid of a space is a *logical construction*, and the notion of an n-type is *logically definable*.
- Many basic results have already been proved and formalized in computational proof assistants, e.g. calculations of many homotopy groups of spheres.

Homotopy Type Theory: Summary

- Under the new homotopical interpretation, constructive type theory provides a "logic of homotopy".
- Logical methods capture some homotopically significant structures: e.g. the fundamental ∞-groupoid of a space is a *logical construction*, and the notion of an n-type is *logically definable*.
- Many basic results have already been proved and formalized in computational proof assistants, e.g. calculations of many homotopy groups of spheres.
- Other areas are also being developed:
  - Foundations: quotient types, inductive types, cumulative hierarchy of sets, ...

Homotopy Type Theory: Summary

- Under the new homotopical interpretation, constructive type theory provides a "logic of homotopy".
- Logical methods capture some homotopically significant structures: e.g. the fundamental ∞-groupoid of a space is a *logical construction*, and the notion of an n-type is *logically definable*.
- Many basic results have already been proved and formalized in computational proof assistants, e.g. calculations of many homotopy groups of spheres.
- Other areas are also being developed:
  - Foundations: quotient types, inductive types, cumulative hierarchy of sets, ...
  - Elementary mathematics: basic algebra, real numbers, cardinal arithmetic, ...

Homotopy Type Theory: Summary

- Under the new homotopical interpretation, constructive type theory provides a "logic of homotopy".
- Logical methods capture some homotopically significant structures: e.g. the fundamental ∞-groupoid of a space is a *logical construction*, and the notion of an n-type is *logically definable*.
- Many basic results have already been proved and formalized in computational proof assistants, e.g. calculations of many homotopy groups of spheres.
- Other areas are also being developed:
  - Foundations: quotient types, inductive types, cumulative hierarchy of sets, ...
  - Elementary mathematics: basic algebra, real numbers, cardinal arithmetic, ...
- Some new logical ideas are suggested by the homotopy interpretation: Higher inductive types, Univalence axiom.

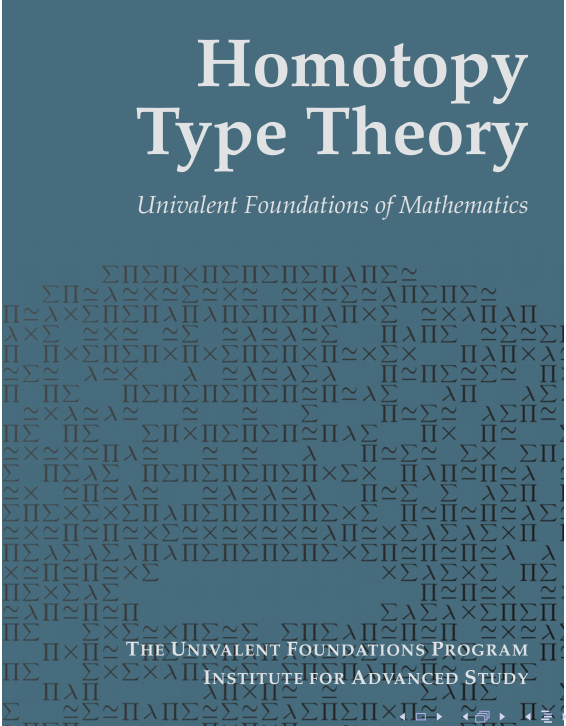
References and Further Information

More Information:

`www.HomotopyTypeTheory.org`

The Book:

Homotopy Type Theory:
Univalent Foundations of Mathematics


Homotopy
Type Theory
Univalent Foundations of Mathematics
The Univalent Foundations Program
Institute for Advanced Study

Higher inductive types (Lumsdaine-Shulman)

The natural numbers $\mathbb{N}$ are implemented as an (ordinary) inductive type:

$$\mathbb{N} := \begin{cases} 0 : \mathbb{N} \\ s : \mathbb{N} \to \mathbb{N} \end{cases}$$

Higher inductive types (Lumsdaine-Shulman)

The natural numbers $\mathbb{N}$ are implemented as an (ordinary) inductive type:

$$\mathbb{N} := \begin{cases} 0 : \mathbb{N} \\ s : \mathbb{N} \to \mathbb{N} \end{cases}$$

The **recursion property** is captured by an elimination rule:

$$\frac{a : X \qquad f : X \to X}{\mathrm{rec}(a, f) : \mathbb{N} \to X}$$

Higher inductive types (Lumsdaine-Shulman)

The natural numbers $\mathbb{N}$ are implemented as an (ordinary) inductive type:

$$\mathbb{N} := \begin{cases} 0 : \mathbb{N} \\ s : \mathbb{N} \to \mathbb{N} \end{cases}$$

The **recursion property** is captured by an elimination rule:

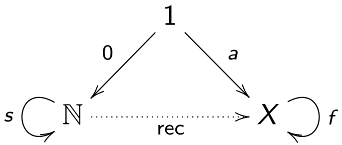
$$\frac{a : X \qquad f : X \to X}{\mathrm{rec}(a, f) : \mathbb{N} \to X}$$

with computation rules:

$$\begin{aligned} \mathrm{rec}(a, f)(0) &= a \\ \mathrm{rec}(a, f)(sn) &= f(\mathrm{rec}(a, f)(n)) \end{aligned}$$

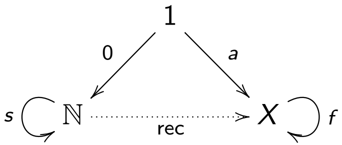
Higher inductive types (Lumsdaine-Shulman)

In other words, $(\mathbb{N}, 0, s)$ is the **free** structure of this type:



Higher inductive types (Lumsdaine-Shulman)

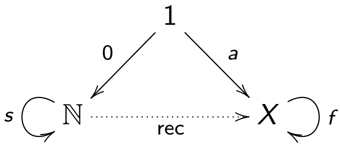
In other words, $(\mathbb{N}, 0, s)$ is the **free** structure of this type:



The map $\mathsf{rec}(a, f) : \mathbb{N} \to X$ is unique.

Higher inductive types (Lumsdaine-Shulman)

In other words, $(\mathbb{N}, 0, s)$ is the **free** structure of this type:



The map $\mathrm{rec}(a, f) : \mathbb{N} \to X$ is unique.

Theorem

$\mathbb{N}$ *is a set (i.e. a* 0*-type).*

Higher inductive types: The circle S^1

The homotopical circle $\mathbb{S} = S^1$ can be given as an inductive type involving a "higher-dimensional" generator:

$$\mathbb{S} := \begin{cases} \mathsf{base} : \mathbb{S} \\ \mathsf{loop} : \mathtt{Id}_{\mathbb{S}}(\mathsf{base}, \mathsf{base}) \end{cases}$$

where we think of $\mathsf{loop} : \mathtt{Id}_{\mathbb{S}}(\mathsf{base}, \mathsf{base})$ as a path

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

Higher inductive types: The circle S^1

$$\mathbb{S} := \begin{cases} \mathsf{base} : \mathbb{S} \\ \mathsf{loop} : \mathtt{Id}_{\mathbb{S}}(\mathsf{base}, \mathsf{base}) \end{cases}$$

The **recursion principle** of $\mathbb{S}$ is given by its elimination rule:

$$\frac{a : X \qquad p : \mathtt{Id}_{\mathbb{X}}(a, a)}{\mathsf{rec}(a, p) : \mathbb{S} \to X}$$

Higher inductive types: The circle S^1

$$\mathbb{S} := \begin{cases} \text{base} : \mathbb{S} \\ \text{loop} : \mathtt{Id}_{\mathbb{S}}(\text{base}, \text{base}) \end{cases}$$

The **recursion principle** of $\mathbb{S}$ is given by its elimination rule:

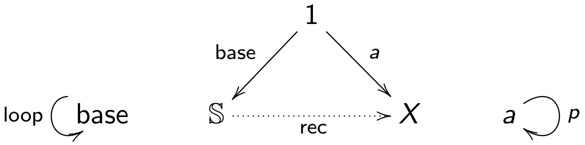
$$\frac{a : X \qquad p : \mathtt{Id}_{\mathbb{X}}(a, a)}{\text{rec}(a, p) : \mathbb{S} \to X}$$

with computation rules:

$$\text{rec}(a, p)(\text{base}) = a$$
$$\text{rec}(a, p)(\text{loop}) = p$$

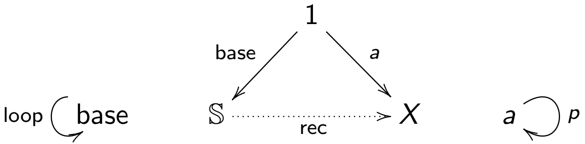
Higher inductive types: The circle $\mathbb{S}^1$

In other words, $(\mathbb{S}, \mathsf{base}, \mathsf{loop})$ is the **free** structure of this type:



Higher inductive types: The circle $\mathbb{S}^1$

In other words, $(\mathbb{S}, \mathsf{base}, \mathsf{loop})$ is the **free** structure of this type:



The map $\mathsf{rec}(a, p) : \mathbb{S} \to X$ is unique up to homotopy.

Higher inductive types: The circle S^1

Here is a sanity check:

Theorem (Shulman 2011)

The type-theoretic circle $\mathbb{S}$ has the correct homotopy groups:

$$\pi_n(\mathbb{S}) = \begin{cases} \mathbb{Z}, & \text{if } n = 1, \\ 0, & \text{if } n \neq 1. \end{cases}$$

Higher inductive types: The circle S^1

Here is a sanity check:

Theorem (Shulman 2011)

The type-theoretic circle $\mathbb{S}$ has the correct homotopy groups:

$$\pi_n(\mathbb{S}) = \begin{cases} \mathbb{Z}, & \text{if } n = 1, \\ 0, & \text{if } n \neq 1. \end{cases}$$

The proof combines classical homotopy theory with methods from constructive type theory, and uses Voevodsky's Univalence Axiom. It has been formalized in Coq.

Higher inductive types: The interval I

The unit interval $\mathbb{I} = [0, 1]$ is also an inductive type, on the data:

$$\mathbb{I} := \begin{cases} 0, 1 : \mathbb{I} \\ p : \mathtt{Id}_{\mathbb{I}}(0, 1) \end{cases}$$

now thinking of $p : \mathtt{Id}_{\mathbb{I}}(0, 1)$ as a "free path"

$$p : 0 \rightsquigarrow 1.$$

Higher inductive types: The interval I

The unit interval $\mathbb{I} = [0,1]$ is also an inductive type, on the data:

$$\mathbb{I} := \begin{cases} 0, 1 : \mathbb{I} \\ p : \mathtt{Id}_{\mathbb{I}}(0,1) \end{cases}$$

now thinking of $p : \mathtt{Id}_{\mathbb{I}}(0,1)$ as a "free path"

$$p : 0 \rightsquigarrow 1.$$

Slogan:

In topology, we start with the **interval** and use it to define the notion of a **path**.

Higher inductive types: The interval I

The unit interval $\mathbb{I} = [0, 1]$ is also an inductive type, on the data:

$$\mathbb{I} := \begin{cases} 0, 1 : \mathbb{I} \\ p : \mathtt{Id}_{\mathbb{I}}(0, 1) \end{cases}$$

now thinking of $p : \mathtt{Id}_{\mathbb{I}}(0, 1)$ as a "free path"

$$p : 0 \rightsquigarrow 1.$$

Slogan:

In topology, we start with the **interval** and use it to define the notion of a **path**.

In HoTT, we start with the notion of a **path**, and use it to define the **interval**.

Higher inductive types: Conclusion

Many basic spaces and constructions can be introduced as HITs:

- higher spheres S^n, cylinders, tori, cell complexes, ...,
- suspensions ΣA,
- homotopy pullbacks, pushouts, etc.,
- truncations, such as connected components $\pi_0(A)$ and "bracket" types $[A]$,
- quotients by equivalence relations and general quotients,
- free algebras, algebras for a monad,
- (higher) homotopy groups π_n, Eilenberg-MacLane spaces $K(G, n)$, Postnikov systems,
- Quillen model structure,
- real numbers,
- cumulative hierarchy of sets.

Univalence

Voevodsky has proposed a new foundational axiom to be added to HoTT: the **Univalence Axiom**.

Univalence

Voevodsky has proposed a new foundational axiom to be added to HoTT: the **Univalence Axiom**.

- It captures the informal mathematical practice of **identifying isomorphic objects**.

Univalence

Voevodsky has proposed a new foundational axiom to be added to HoTT: the **Univalence Axiom**.

- It captures the informal mathematical practice of **identifying isomorphic objects**.
- It is very **useful** from a **practical** point of view, especially when combined with HITs.

Univalence

Voevodsky has proposed a new foundational axiom to be added to HoTT: the **Univalence Axiom**.

- It captures the informal mathematical practice of **identifying isomorphic objects**.
- It is very **useful** from a **practical** point of view, especially when combined with HITs.
- It is **formally incompatible** with the assumption that all types are **sets**.

Univalence

Voevodsky has proposed a new foundational axiom to be added to HoTT: the **Univalence Axiom**.

- It captures the informal mathematical practice of **identifying isomorphic objects**.
- It is very **useful** from a **practical** point of view, especially when combined with HITs.
- It is **formally incompatible** with the assumption that all types are **sets**.
- Its status as a **constructive principle** is the focus of much current research.

Isomorphism and Equivalence

In type theory, the usual notion of *isomorphism* $A \cong B$ is definable:

$$A \cong B \;\Leftrightarrow\; \textit{there are } f : A \to B \textit{ and } g : B \to A$$
$$\textit{such that } gf(x) = x \textit{ and } fg(y) = y.$$

Isomorphism and Equivalence

In type theory, the usual notion of *isomorphism* $A \cong B$ is definable:

$$A \cong B \;\Leftrightarrow\; \textit{there are } f : A \to B \textit{ and } g : B \to A \\ \textit{such that } gf(x) = x \textit{ and } fg(y) = y.$$

Formally, there is the type of isomorphisms:

$$\mathrm{Iso}(A, B) \;:=\; \sum_{f:A\to B} \sum_{g:B\to A} \Big(\prod_{x:A} \mathtt{Id}_A(gf(x), x) \times \prod_{y:B} \mathtt{Id}_B(fg(y), y) \Big)$$

Isomorphism and Equivalence

In type theory, the usual notion of *isomorphism* $A \cong B$ is definable:

$$A \cong B \;\Leftrightarrow\; \textit{there are } f : A \to B \textit{ and } g : B \to A \textit{ such that } gf(x) = x \textit{ and } fg(y) = y.$$

Formally, there is the type of isomorphisms:

$$\mathrm{Iso}(A, B) \;:=\; \sum_{f:A\to B}\,\sum_{g:B\to A} \Big(\prod_{x:A} \mathtt{Id}_A(gf(x), x) \times \prod_{y:B} \mathtt{Id}_B(fg(y), y) \Big)$$

Thus $A \cong B$ iff this type is inhabited by a closed term, which is then just an isomorphism between A and B.

Isomorphism and Equivalence

- There is also a more refined notion of *equivalence* of types,

$$A \simeq B$$

which adds a further "coherence" condition relating the *proofs* of $gf(x) = x$ and $fg(y) = y$.

Isomorphism and Equivalence

- There is also a more refined notion of *equivalence* of types,

$$A \simeq B$$

  which adds a further "coherence" condition relating the *proofs* of $gf(x) = x$ and $fg(y) = y$.
- Under the homotopy interpretation, this is the type of *homotopy equivalences*.

Isomorphism and Equivalence

- There is also a more refined notion of *equivalence* of types,

$$A \simeq B$$

  which adds a further "coherence" condition relating the *proofs* of $gf(x) = x$ and $fg(y) = y$.
- Under the homotopy interpretation, this is the type of *homotopy equivalences*.
- This subsumes *categorical equivalence* (for 1-types), *isomorphism of sets* (for 0-types), and *logical equivalence* (for (-1)-types).

Invariance

One can show that all *definable properties* $P(X)$ of types X *respect type equivalence*:

$$A \simeq B \text{ and } P(A) \text{ implies } P(B)$$

Invariance

One can show that all *definable properties* $P(X)$ of types X *respect type equivalence*:

$$A \simeq B \text{ and } P(A) \text{ implies } P(B)$$

In this sense, *all properties are invariant*.

Invariance

One can show that all *definable properties* $P(X)$ of types X *respect type equivalence*:

$$A \simeq B \text{ and } P(A) \text{ implies } P(B)$$

In this sense, *all properties are invariant*.
Moreover, therefore, *equivalent types* $A \simeq B$ *are indiscernable*:

$$P(A) \Rightarrow P(B), \text{for all } P$$

Invariance

One can show that all *definable properties* $P(X)$ of types X *respect type equivalence*:

$$A \simeq B \text{ and } P(A) \text{ implies } P(B)$$

In this sense, *all properties are invariant*.
Moreover, therefore, *equivalent types* $A \simeq B$ *are indiscernable*:

$$P(A) \Rightarrow P(B), \text{for all } P$$

How is this related to **identity of types** A and B?

Univalence

To reason about **identity of types**, we need a *type universe* $\mathcal{U}$, with an identity type,

$$\mathtt{Id}_{\mathcal{U}}(A, B).$$

Univalence

To reason about **identity of types**, we need a *type universe* $\mathcal{U}$, with an identity type,

$$\mathtt{Id}_{\mathcal{U}}(A, B).$$

Since identity implies equivalence there is a comparison map:

$$\mathtt{Id}_{\mathcal{U}}(A, B) \to (A \simeq B).$$

The *Univalence Axiom* asserts that this map is an equivalence:

$$\mathtt{Id}_{\mathcal{U}}(A, B) \;\simeq\; (A \simeq B) \qquad \text{(UA)}$$

Univalence

To reason about **identity of types**, we need a *type universe* $\mathcal{U}$, with an identity type,

$$\mathtt{Id}_{\mathcal{U}}(A, B).$$

Since identity implies equivalence there is a comparison map:

$$\mathtt{Id}_{\mathcal{U}}(A, B) \to (A \simeq B).$$

The *Univalence Axiom* asserts that this map is an equivalence:

$$\mathtt{Id}_{\mathcal{U}}(A, B) \ \simeq \ (A \simeq B) \qquad \text{(UA)}$$

So UA can be stated: *"Identity is equivalent to equivalence."*

The Univalence Axiom: Remarks

- Since UA is an equivalence, there is a map coming back:

$$\mathtt{Id}_{\mathcal{U}}(A, B) \longleftarrow (A \simeq B)$$

  In this sense, **equivalent objects are identical**.
- So logically equivalent propositions are identical, and isomorphic sets, groups, etc., can be identified.

The Univalence Axiom: Remarks

- Since UA is an equivalence, there is a map coming back:

$$\mathtt{Id}_{\mathcal{U}}(A, B) \longleftarrow (A \simeq B)$$

  In this sense, **equivalent objects are identical**.
- So logically equivalent propositions are identical, and isomorphic sets, groups, etc., can be identified.
- UA implies that $\mathcal{U}$, in particular, is not a set (0-type).

The Univalence Axiom: Remarks

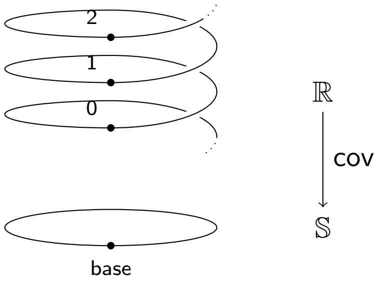
- Since UA is an equivalence, there is a map coming back:

$$\mathtt{Id}_{\mathcal{U}}(A, B) \longleftarrow (A \simeq B)$$

  In this sense, **equivalent objects are identical**.
- So logically equivalent propositions are identical, and isomorphic sets, groups, etc., can be identified.
- UA implies that $\mathcal{U}$, in particular, is not a set (0-type).
- The computational character of UA is still an open question.

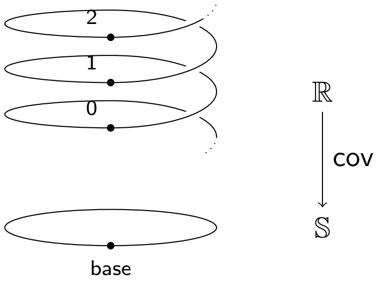
The Univalence Axiom: How it works

To compute the fundamental group of the circle $\mathbb{S}$, we shall construct the universal cover:



The Univalence Axiom: How it works

To compute the fundamental group of the circle $\mathbb{S}$, we shall construct the universal cover:



This will be a dependent type over $\mathbb{S}$, i.e. a type family

$$\mathsf{cov} : \mathbb{S} \longrightarrow \mathcal{U}.$$

The Univalence Axiom: How it works

To define a type family

$$\mathsf{cov} : \mathbb{S} \longrightarrow \mathcal{U},$$

by the recursion property of the circle, we just need the following data:

- a point $A : \mathcal{U}$
- a loop $p : A \rightsquigarrow A$

The Univalence Axiom: How it works

To define a type family

$$\mathsf{cov} : \mathbb{S} \longrightarrow \mathcal{U},$$

by the recursion property of the circle, we just need the following data:

- a point $A : \mathcal{U}$
- a loop $p : A \rightsquigarrow A$

For the point A we take the integers $\mathbb{Z}$.

The Univalence Axiom: How it works

To define a type family

$$\mathsf{cov} : \mathbb{S} \longrightarrow \mathcal{U},$$

by the recursion property of the circle, we just need the following data:

- a point $A : \mathcal{U}$
- a loop $p : A \rightsquigarrow A$

For the point A we take the integers $\mathbb{Z}$.
By Univalence, to give a loop $p : \mathbb{Z} \rightsquigarrow \mathbb{Z}$ in $\mathcal{U}$, it suffices to give an equivalence $\mathbb{Z} \simeq \mathbb{Z}$.

The Univalence Axiom: How it works

To define a type family

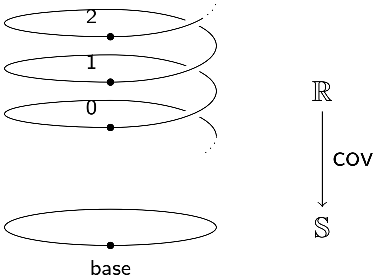
$$\mathsf{cov} : \mathbb{S} \longrightarrow \mathcal{U},$$

by the recursion property of the circle, we just need the following data:

- a point $A : \mathcal{U}$
- a loop $p : A \rightsquigarrow A$

For the point A we take the integers $\mathbb{Z}$.
By Univalence, to give a loop $p : \mathbb{Z} \rightsquigarrow \mathbb{Z}$ in $\mathcal{U}$, it suffices to give an equivalence $\mathbb{Z} \simeq \mathbb{Z}$.
Since $\mathbb{Z}$ is a set, equivalences are just isomorphisms, so we can take the successor function $\mathsf{succ} : \mathbb{Z} \cong \mathbb{Z}$.

The Univalence Axiom: How it works

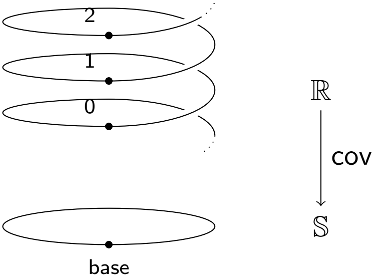


Definition (Universal Cover of $\mathbb{S}^1$)

The dependent type $\mathsf{cov} : \mathbb{S} \longrightarrow \mathcal{U}$ is given by circle-recursion, with

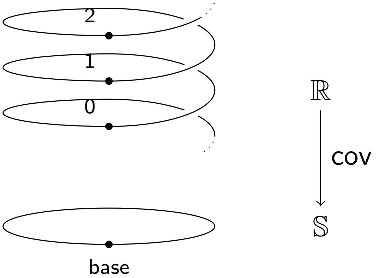
$$\mathsf{cov}(\mathsf{base}) := \mathbb{Z}$$
$$\mathsf{cov}(\mathsf{loop}) := \mathsf{ua}(\mathsf{succ}).$$

The Univalence Axiom: How it works



As in classical homotopy theory, we use the universal cover to define the "winding number" of any path $p : \mathsf{base} \rightsquigarrow \mathsf{base}$ by $\mathsf{wind}(p) = p_*(0)$.

The Univalence Axiom: How it works



As in classical homotopy theory, we use the universal cover to define the "winding number" of any path $p : \mathsf{base} \rightsquigarrow \mathsf{base}$ by $\mathsf{wind}(p) = p_*(0)$. This gives a map

$$\mathsf{wind} : \Omega(\mathbb{S}) \longrightarrow \mathbb{Z},$$

which is inverse to the map $\mathbb{Z} \longrightarrow \Omega(\mathbb{S})$ given by

$$z \mapsto \mathsf{loop}^z.$$

The formal proof

```
(** * Theorems about the circle S^1. *)

Require Import Overture PathGroupoids Equivalences Trunc HSet.
Require Import Paths Forall Arrow Universe Empty Unit.
Local Open Scope path_scope.
Local Open Scope equiv_scope.
Generalizable Variables X A B f g n.

(* *** Definition of the circle. *)

Module Export Circle.

Local Inductive S1 : Type :=
| base : S1.

Axiom loop : base = base.

Definition S1_rect (P : S1 -> Type) (b : P base) (l : loop # b = b)
  : forall (x:S1), P x
  := fun x => match x with base => b end.

Axiom S1_rect_beta_loop
  : forall (P : S1 -> Type) (b : P base) (l : loop # b = b),
  apD (S1_rect P b l) loop = l.

End Circle.
```

```
(* *** The non-dependent eliminator *)

Definition S1_rectnd (P : Type) (b : P) (l : b = b)
  : S1 -> P
  := S1_rect (fun _ => P) b (transport_const _ _ @ l).

Definition S1_rectnd_beta_loop (P : Type) (b : P) (l : b = b)
  : ap (S1_rectnd P b l) loop = l.
Proof.
  unfold S1_rectnd.
  refine (cancelL (transport_const loop b) _ _ _).
  refine ((apD_const (S1_rect (fun _ => P) b _) loop)^ @ _).
  refine (S1_rect_beta_loop (fun _ => P) _ _).
Defined.

(* *** The loop space of the circle is the Integers. *)

(* First we define the appropriate integers. *)

Inductive Pos : Type :=
| one : Pos
| succ_pos : Pos -> Pos.

Definition one_neq_succ_pos (z : Pos) : ~ (one = succ_pos z)
  := fun p => transport (fun s => match s with one => Unit | succ_pos t => Empty end) p tt.

Definition succ_pos_injective {z w : Pos} (p : succ_pos z = succ_pos w) : z = w
  := transport (fun s => z = (match s with one => w | succ_pos a => a end)) p (idpath z).

Inductive Int : Type :=
| neg : Pos -> Int
| zero : Int
| pos : Pos -> Int.
```

```
Definition neg_injective {z w : Pos} (p : neg z = neg w) : z = w
  := transport (fun s => z = (match s with neg a => a | zero => w | pos a => w end)) p (idpath z).

Definition pos_injective {z w : Pos} (p : pos z = pos w) : z = w
  := transport (fun s => z = (match s with neg a => w | zero => w | pos a => a end)) p (idpath z).

Definition neg_neq_zero {z : Pos} : ~ (neg z = zero)
  := fun p => transport (fun s => match s with neg a => z = a | zero => Empty
  | pos _ => Empty end) p (idpath z).

Definition pos_neq_zero {z : Pos} : ~ (pos z = zero)
  := fun p => transport (fun s => match s with pos a => z = a
  | zero => Empty | neg _ => Empty end) p (idpath z).

Definition neg_neq_pos {z w : Pos} : ~ (neg z = pos w)
  := fun p => transport (fun s => match s with neg a => z = a
  | zero => Empty | pos _ => Empty end) p (idpath z).

(* And prove that they are a set. *)

Instance hset_int : IsHSet Int.
Proof.
  apply hset_decidable.
  intros [n | | n] [m | | m].
  revert m; induction n as [|n IHn]; intros m; induction m as [|m IHm].
  exact (inl 1).
  exact (inr (fun p => one_neq_succ_pos _ (neg_injective p))).
  exact (inr (fun p => one_neq_succ_pos _ (symmetry _ _ (neg_injective p)))).
  destruct (IHn m) as [p | np].
  exact (inl (ap neg (ap succ_pos (neg_injective p)))).
  exact (inr (fun p => np (ap neg (succ_pos_injective (neg_injective p))))).
  exact (inr neg_neq_zero).
  exact (inr neg_neq_pos).
  exact (inr (neg_neq_zero o symmetry _ _)).
  exact (inl 1).
```

```
  exact (inr (pos_neq_zero o symmetry _ _)).
  exact (inr (neg_neq_pos o symmetry _ _)).
  exact (inr pos_neq_zero).
  revert m; induction n as [|n IHn]; intros m; induction m as [|m IHm].
  exact (inl 1).
  exact (inr (fun p => one_neq_succ_pos _ (pos_injective p))).
  exact (inr (fun p => one_neq_succ_pos _ (symmetry _ _ (pos_injective p)))).
  destruct (IHn m) as [p | np].
  exact (inl (ap pos (ap succ_pos (pos_injective p)))).
  exact (inr (fun p => np (ap pos (succ_pos_injective (pos_injective p))))).
Defined.

(* Successor is an autoequivalence of [Int]. *)

Definition succ_int (z : Int) : Int
  := match z with
       | neg (succ_pos n) => neg n
       | neg one => zero
       | zero => pos one
       | pos n => pos (succ_pos n)
     end.

Definition pred_int (z : Int) : Int
  := match z with
       | neg n => neg (succ_pos n)
       | zero => neg one
       | pos one => zero
       | pos (succ_pos n) => pos n
     end.

Instance isequiv_succ_int : IsEquiv succ_int
  := isequiv_adjointify succ_int pred_int _ _.
Proof.
  intros [[|n] | | [|n]]; reflexivity.
  intros [[|n] | | [|n]]; reflexivity.
Defined.
```

```
(* Now we do the encode/decode. *)

Section AssumeUnivalence.
Context `{Univalence} `{Funext}.

Definition S1_code : S1 -> Type
  := S1_rectnd Type Int (path_universe succ_int).

(* Transporting in the codes fibration is the successor autoequivalence. *)

Definition transport_S1_code_loop (z : Int)
  : transport S1_code loop z = succ_int z.
Proof.
  refine (transport_compose idmap S1_code loop z @ _).
  unfold S1_code; rewrite S1_rectnd_beta_loop.
  apply transport_path_universe.
Defined.

Definition transport_S1_code_loopV (z : Int)
  : transport S1_code loop^ z = pred_int z.
Proof.
  refine (transport_compose idmap S1_code loop^ z @ _).
  rewrite ap_V.
  unfold S1_code; rewrite S1_rectnd_beta_loop.
  rewrite <- path_universe_V.
  apply transport_path_universe.
Defined.
```

```
(* Encode by transporting *)

Definition S1_encode (x:S1) : (base = x) -> S1_code x
  := fun p => p # zero.

(* Decode by iterating loop. *)

Fixpoint loopexp {A : Type} {x : A} (p : x = x) (n : Pos) : (x = x)
  := match n with
       | one => p
       | succ_pos n => loopexp p n @ p
     end.

Definition looptothe (z : Int) : (base = base)
  := match z with
       | neg n => loopexp (loop^) n
       | zero => 1
       | pos n => loopexp (loop) n
     end.

Definition S1_decode (x:S1) : S1_code x -> (base = x).
Proof.
  revert x; refine (S1_rect (fun x => S1_code x -> base = x) looptothe _).
  apply path_forall; intros z; simpl in z.
  refine (transport_arrow _ _ _ @ _).
  refine (transport_paths_r loop _ @ _).
  rewrite transport_S1_code_loopV.
  destruct z as [[|n] | | [|n]]; simpl.
  by apply concat_pV_p.
  by apply concat_pV_p.
  by apply concat_Vp.
  by apply concat_1p.
  reflexivity.
Defined.
```

```
(* The nontrivial part of the proof that decode and encode are equivalences is showing that decoding
followed by encoding is the identity on the fibers over [base]. *)

Definition S1_encode_looptothe (z:Int)
  : S1_encode base (looptothe z) = z.
Proof.
  destruct z as [n | | n]; unfold S1_encode.
  induction n; simpl in *.
  refine (moveR_transport_V _ loop _ _ _).
  by apply symmetry, transport_S1_code_loop.
  rewrite transport_pp.
  refine (moveR_transport_V _ loop _ _ _).
  refine (_ @ (transport_S1_code_loop _)^).
  assumption.
  reflexivity.
  induction n; simpl in *.
  by apply transport_S1_code_loop.
  rewrite transport_pp.
  refine (moveR_transport_p _ loop _ _ _).
  refine (_ @ (transport_S1_code_loopV _)^).
  assumption.
Defined.
```

```
(* Now we put it together. *)

Definition S1_encode_isequiv (x:S1) : IsEquiv (S1_encode x).
Proof.
  refine (isequiv_adjointify (S1_encode x) (S1_decode x) _ _).
  (* Here we induct on [x:S1].  We just did the case when [x] is [base]. *)
  refine (S1_rect (fun x => Sect (S1_decode x) (S1_encode x))
    S1_encode_looptothe _ _).
  (* What remains is easy since [Int] is known to be a set. *)
  by apply path_forall; intros z; apply set_path2.
  (* The other side is trivial by path induction. *)
  intros []; reflexivity.
Defined.

Definition equiv_loopS1_int : (base = base) <~> Int
  := BuildEquiv _ _ (S1_encode base) (S1_encode_isequiv base).

End AssumeUnivalence.
```

Final Example: The cumulative hierarchy

Given a universe $\mathcal{U}$, we can make the *cumulative hierarchy* V of sets in $\mathcal{U}$ as a HIT:

- for any small A and any map $f : A \to V$, there is a "set":

  $$\mathsf{set}(A, f) : V$$

  We think of $\mathsf{set}(A, f)$ as the image of A under f, i.e. the classical set $\{f(a) \mid a \in A\}$
- For all $A, B : \mathcal{U}$, $f : A \to V$ and $g : B \to V$ such that

  $$\big(\forall a : A \, \exists b : B \;\, f(a) = g(b)\big) \wedge \big(\forall b : B \, \exists a : A \;\, f(a) = g(b)\big)$$

  we put in a path in V from $\mathsf{set}(A, f)$ to $\mathsf{set}(B, g)$.
- The 0-truncation constructor: for all $x, y : V$ and $p, q : x = y$, we have $p = q$.

The cumulative hierarchy of sets

Membership $x \in y$ is then defined for elements of V by:

$$(x \in \mathrm{set}(A, f)) \;:=\; (\exists a : A.\ x = f(a)).$$

The cumulative hierarchy of sets

Membership $x \in y$ is then defined for elements of V by:

$$(x \in \mathrm{set}(A, f)) \;:=\; (\exists a : A.\ x = f(a)).$$

One can show that the resulting structure $(V, \in)$ satisfies most of the axioms of Aczel's constructive set theory CZF.

The cumulative hierarchy of sets

Membership $x \in y$ is then defined for elements of V by:

$$(x \in \mathrm{set}(A, f)) \;:=\; (\exists a : A.\ x = f(a)).$$

One can show that the resulting structure $(V, \in)$ satisfies most of the axioms of Aczel's constructive set theory CZF.

Assuming AC for sets (0-types), one gets a model of ZFC set theory.

The cumulative hierarchy of sets

Membership $x \in y$ is then defined for elements of V by:

$$(x \in \mathsf{set}(A, f)) \;:=\; (\exists a : A.\ x = f(a)).$$

One can show that the resulting structure $(V, \in)$ satisfies most of the axioms of Aczel's constructive set theory CZF.

Assuming AC for sets (0-types), one gets a model of ZFC set theory.

The proofs make essential use of UA.