What is HoTT?

Steve Awodey

Carnegie Mellon University

Royal Society Wolfson Visiting Fellow

Department of Computer Science and Technology

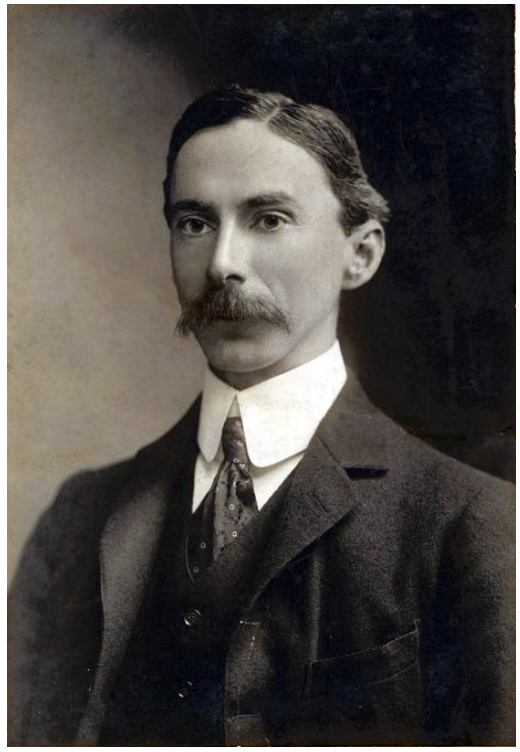
University of Cambridge

28 January 2026

Overview

- Homotopy Type Theory (HoTT) is a system of type theory with an interpretation into homotopy theory.
- It extends Martin-Löf's constructive type theory (MLTT) with new principles that strengthen this connection.
- MLTT underlies several computer theorem provers such as Lean, Roq, and Agda.
- HoTT allows such systems to formalize proofs in higher maths, like homotopy theory and higher category theory.
- HoTT preserves the constructive character of MLTT, allowing program extraction from proofs and computation.

Type Theory



Constructive Type Theory

Constructive type theory replaces logical formulas and their proofs by types and terms.

- *types*: $1,\ \mathbb{N},\ A \times B,\ A + B,\ A \to B$
- *terms*: $*,\ n,\ \langle a, b\rangle,\ [a, b],\ \lambda x.b(x)$
- *dependent types and terms*: $x : A \vdash b(x) : B(x)$
- *sum and product types*: $\Sigma_{x:A} B(x),\ \Pi_{x:A} B(x)$

A term e.g. of the form

$$t : \Pi_{x:A} \Sigma_{y:B} R(x, y)$$

determines a computable function $t_1 : A \to B$ and a term

$$t_2 a : R(a, t_1 a).$$

So we can extract computations from proof terms.

The Curry-Howard Correspondence

The type constructors replace the logical operations.

0	1	$A + B$	$A \times B$	$A \to B$	$\Sigma_{x:A} B(x)$	$\Pi_{x:A} B(x)$
$\bot$	T	$\alpha \vee \beta$	$\alpha \wedge \beta$	$\alpha \Rightarrow \beta$	$\exists_{x:\alpha} \beta(x)$	$\forall_{x:\alpha} \beta(x)$

The Curry-Howard Correspondence

The type constructors replace the logical operations.

0	1	$A + B$	$A \times B$	$A \to B$	$\Sigma_{x:A} B(x)$	$\Pi_{x:A} B(x)$	$\mathsf{Id}(a, b)$
$\bot$	T	$\alpha \vee \beta$	$\alpha \wedge \beta$	$\alpha \Rightarrow \beta$	$\exists_{x:\alpha} \beta(x)$	$\forall_{x:\alpha} \beta(x)$	$a = b$

Martin-Löf introduced *identity types* $\mathsf{Id}(a, b)$, with rules that preserves the constructive character of the system.

But their meaning was mysterious, as there may be different terms:

$$p, q : \mathsf{Id}(a, b)$$

The Homotopy Interpretation

We extended Dana Scott's topological interpretation of λ-calculus:

$$\begin{array}{rcl} \text{types } X & \rightsquigarrow & \text{spaces} \\ \text{terms } t : X \to Y & \rightsquigarrow & \text{continuous functions} \\ \text{identities } p : \mathsf{Id}_X(a, b) & \rightsquigarrow & \text{paths } p : a \sim b \text{ in } X \end{array}$$

A *path* $p : a \sim b$ from point a to point b in a space X is a continuous function

$$p : [0, 1] \to X$$

with $p(0) = a$ and $p(1) = b$.

The Homotopy Interpretation

Theorem (Awodey-Warren 2006)

This interpretation satisfies Martin-Löf's rules for identity types.

Homotopy

The relation $a \sim b$ satisfies the usual **laws of identity**,

$$a \sim a$$

$$a \sim b \;\Rightarrow\; b \sim a$$

$$a \sim b,\; b \sim c \;\Rightarrow\; a \sim c$$

Homotopy

The relation $a \sim b$ satisfies the usual **laws of identity**,

$$r : a \sim a$$

$$p : a \sim b \;\Rightarrow\; p^{-1} : b \sim a$$

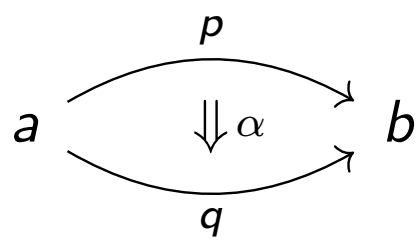
$$p : a \sim b,\; q : b \sim c \;\Rightarrow\; p.q : a \sim c$$

They are witnessed by paths $p : a \sim b$.

And these paths satisfy **higher laws**,

$$\alpha : p.(q.r) \approx (p.q).r$$

Such higher paths are called **homotopies**,



and they satisfy **even higher laws** ....

The Homotopy Interpretation of Identity Types

The identity types in MLTT endowed each type X with **higher structure**:

$$\begin{aligned} a, b &: X \\ p, q &: \mathsf{Id}_X(a, b) \\ \alpha, \beta &: \mathsf{Id}_{\mathsf{Id}_X(a,b)}(p, q), \\ &\dots \end{aligned}$$

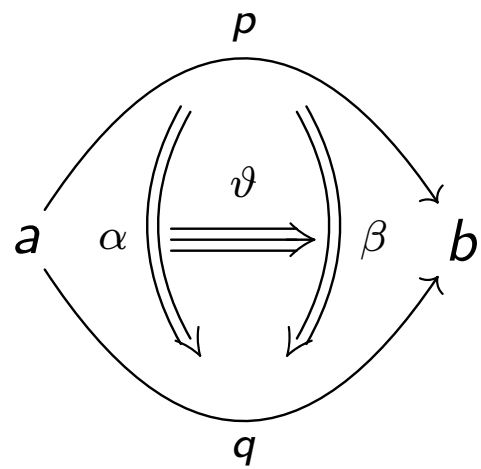
These satisfy the same laws as **homotopies**:

$$\begin{aligned} a, b : X &\quad\rightsquigarrow\quad \text{points of } X \\ p : \mathsf{Id}_X(a, b) &\quad\rightsquigarrow\quad \text{paths } p : a \sim b \\ \alpha : \mathsf{Id}_{\mathsf{Id}_X(a,b)}(p, q) &\quad\rightsquigarrow\quad \text{homotopies } \alpha : p \approx q, \dots \end{aligned}$$

The Homotopy Interpretation: ∞-Groupoids

Theorem (Lumsdaine, van den Berg-Garner)

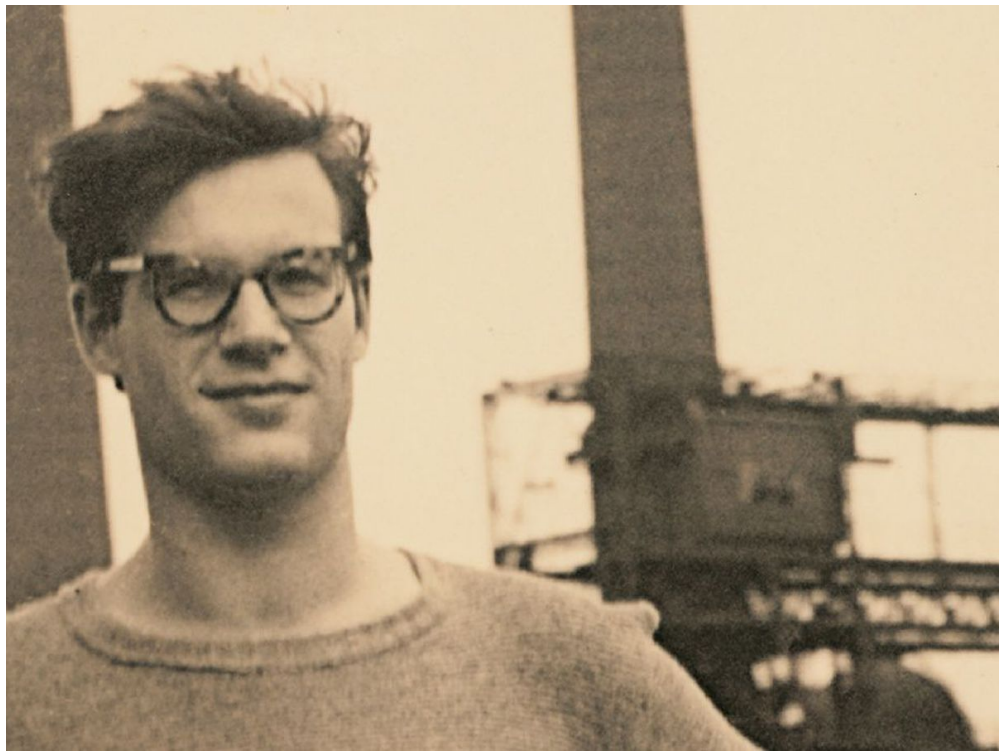
The identity types of a type in MLTT form an ∞-**groupoid**.



The Homotopy Interpretation: ∞-Groupoids

Such structures arose in Grothendieck's **Homotopy Hypothesis**:

∞-Groupoids classify homotopy types of spaces



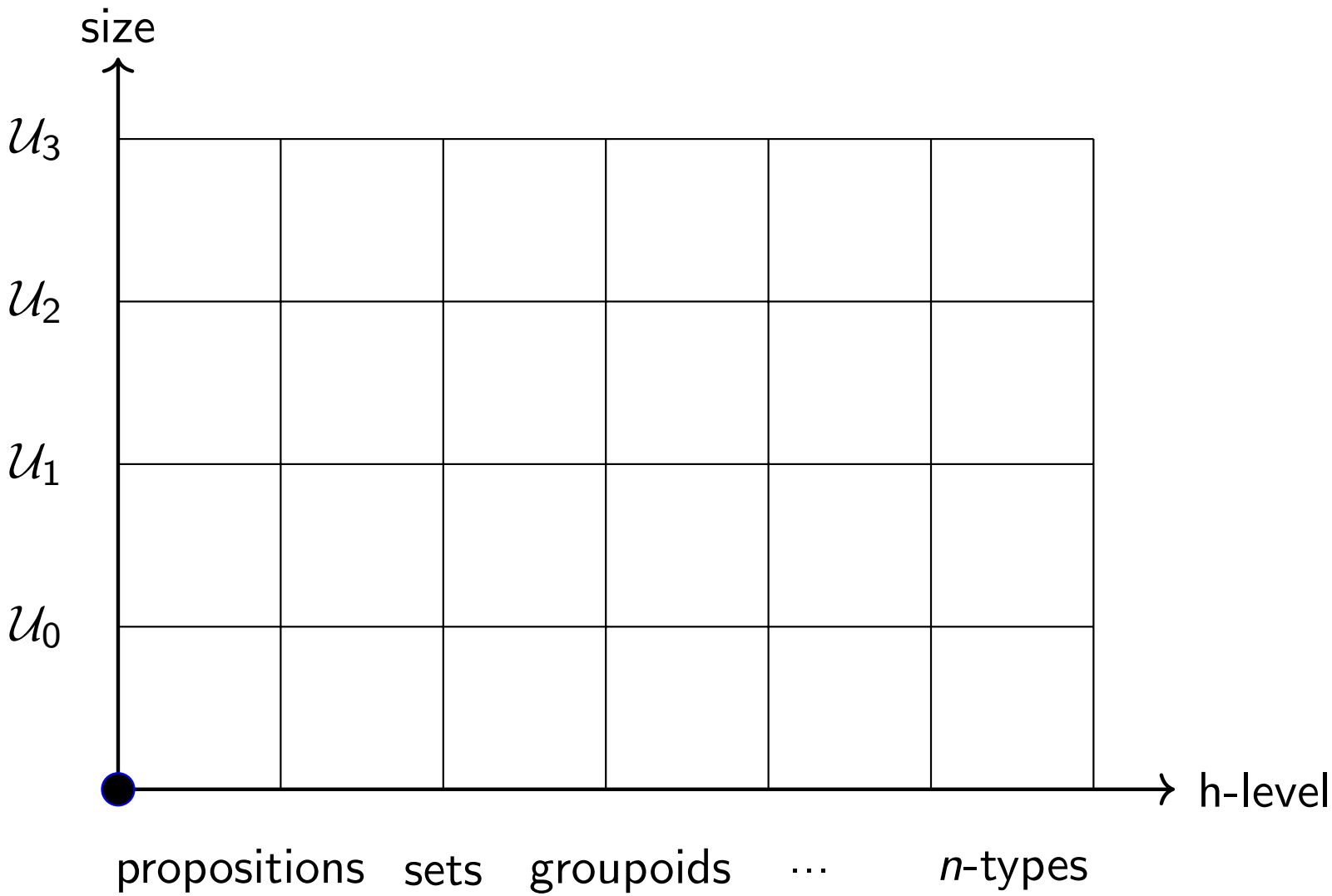
Homotopy Levels (Voevodsky)

Like spaces, types of MLTT are therefore stratified by the level at which their ∞-groupoid becomes trivial.

contractible:	$\Sigma_{x:X}\Pi_{y:X}\mathtt{Id}_X(x,y)$	*(X is a point)*
propositions:	$\Pi_{x,y:X}\ \mathrm{Contr}(\mathtt{Id}_X(x,y))$	*(identity is contractible)*
sets:	$\Pi_{x,y:X}\ \mathrm{Prop}(\mathtt{Id}_X(x,y))$	*(identity is a proposition)*
1-*groupoids*:	$\Pi_{x,y:X}\ \mathrm{Set}(\mathtt{Id}_X(x,y))$	*(identity is a set)*
$(n+1)$-*groupoids*:	$\Pi_{x,y:X}\ \mathrm{nGpd}(\mathtt{Id}_X(x,y))$	*(identity is an n-groupoid)*

This refines the **Propositions-as-Types** interpretation of MLTT. Types are now *structures*, rather than mere *propositions*.

Propositions as Homotopy Types



Univalence

Voevodsky also proposed the *Univalence Axiom*:

$$\mathsf{Id}_{\mathcal{U}}(X, Y) \;\simeq\; (X \simeq Y)$$

It has many remarkable consequences: function extensionality, identification of isomorphic structures, invariance under equivalence, ...

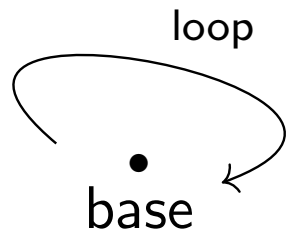
But is it still **constructive**?

Higher Inductive Types

Higher inductive types (HITs) define spaces like the spheres S^n.
The circle S^1 is an inductive type with a single higher generator:

$$S^1 := \begin{cases} \text{base} : S^1 \\ \text{loop} : \mathtt{Id}_{S^1}(\text{base}, \text{base}) \end{cases}$$

We think of $\text{loop} : \mathtt{Id}_{S^1}(\text{base}, \text{base})$ as the "free generator" of S^1,



Fundamental Groups

The *fundamental group* $\pi_1(X)$ of a space X was introduced by Poincaré in the influential paper *Analysis situs* (1895).
For a point $* \in X$ it consists of all loops $\ell : * \sim *$ up to homotopy.

Homotopy Groups of Spheres

Shulman calculated the fundamental group of the circle S^1 in HoTT to be

$$\pi_1(S^1) \simeq \mathsf{Id}_{S^1}(\mathsf{base}, \mathsf{base}) \;\cong\; \mathbb{Z}\,,$$

and formalized the proof in Coq-HoTT in 2011.

The higher homotopy groups of the spheres $\pi_k(S^n)$ are defined in HoTT as sets of *pointed* maps $S^k \dot{\to} S^n$ identified up to homotopy,

$$\pi_k(S^n) \;=\; ||S^k \to_* S^n||_0$$

Some of these were calculated at the IAS special year on Univalent Foundations in 2012–13.

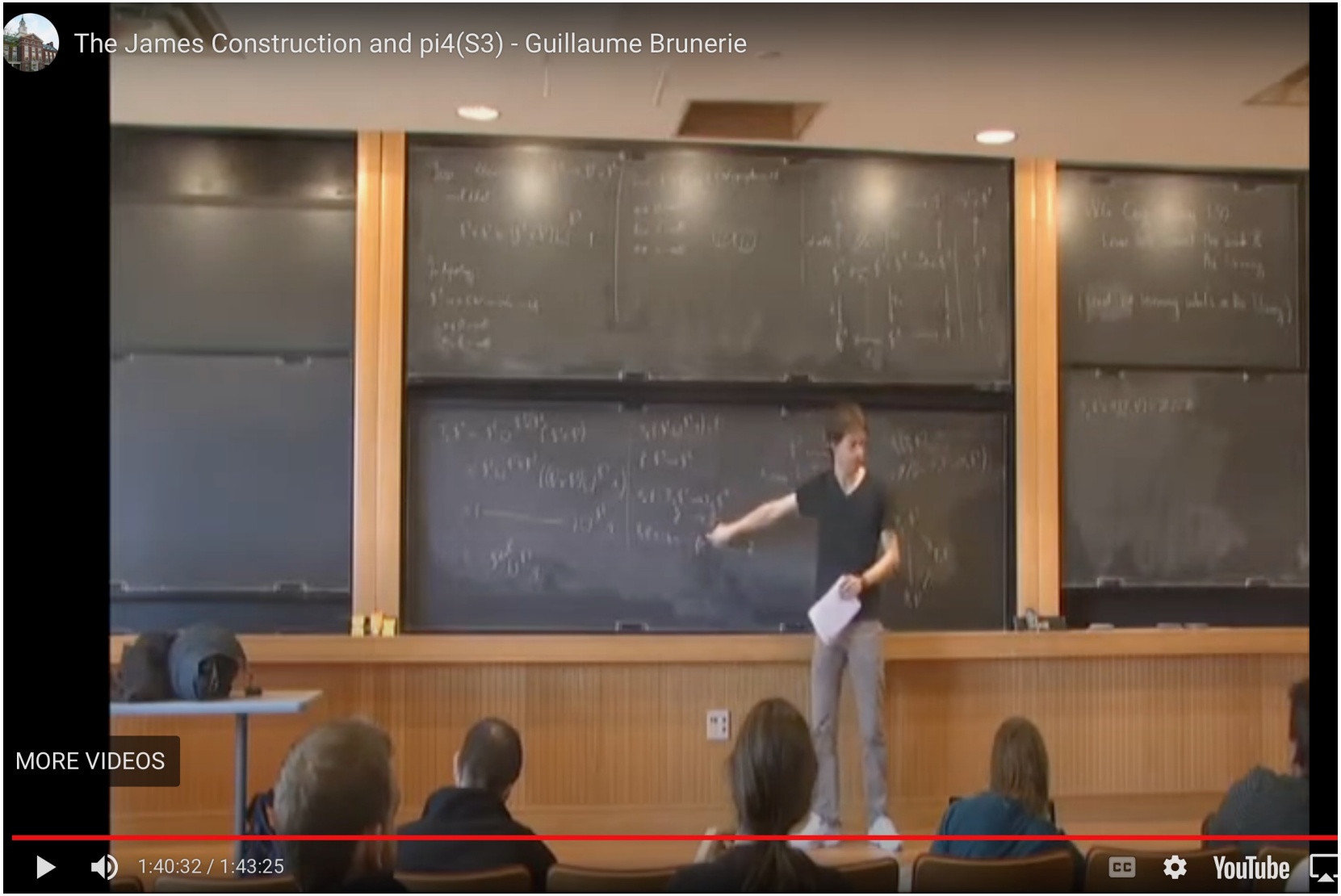
The IAS Special Year

An Open Problem

At the end of the special year Brunerie showed in HoTT that the 4th homotopy group of the 3-sphere is

$$\pi_4(\mathsf{S}^3) \;\cong\; \mathbb{Z}/n\mathbb{Z}\,.$$

But the value of n could not be **computed** from the proof without a *constructive implementation* of HoTT in a proof assistant.


The James Construction and pi4(S3) - Guillaume Brunerie
MORE VIDEOS
1:40:32 / 1:43:25
YouTube

Brunerie's "Perfect World"

So what we get is that $\pi_4(\mathsf{S}^3)$ *... is equal to* $\mathbb{Z} \bmod n$ *for* **this** *n. And this is one very concrete and non-trivial example of why we may want to have canonicity, because this n is a closed term of type* $\mathbb{Z}$*, defined with a lot of univalence and higher inductive types. So in a perfect world, if you formalize that in a proof assistant with a computational interpretation of univalence ... you can just ask "what is the value of n?" and you will get* 2*.*

Guillaume Brunerie, 23 May 2013, IAS

Computation of Brunerie's Number

Since 2013:

1. Constructivity of Univalence and HITs

   Coquand and collaborators developed a constructive version of HoTT with univalence and HITs (2014-16).

2. Implementation in a computational proof assistant
3. Computation of $\pi_4(\mathsf{S}^3)$

Computation of Brunerie's Number

Since 2013:

1. Constructivity of Univalence and HITs (2014-16) ✓
2. Implementation in a computational proof assistant

   A new proof assistant Cubical Agda that computes with Univalence and HITs was developed on that basis (2019).
3. Computation of $\pi_4(\mathsf{S}^3)$

Computation of Brunerie's Number

Since 2013:

1. Constructivity of Univalence and HITs (2014–16) ✓
2. Implementation in a computational proof assistant (2019) ✓
3. Computation of $\pi_4(\mathsf{S}^3)$

   Brunerie's IAS proof that, for some $n : \mathbb{Z}$,

   $$\pi_4(\mathsf{S}^3) = \mathbb{Z}/_{n\,\mathbb{Z}}$$

   was formalized in Cubical Agda and the value of $n = 2$ was computed from the proof term (2022).

Computation of Brunerie's Number

Since 2013:

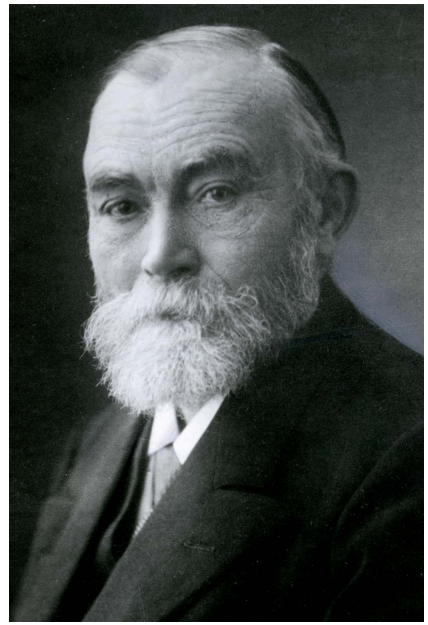
1. Constructivity of Univalence and HITs (2014–16) ✓
2. Implementation in a computational proof assistant (2019) ✓
3. Computation of $\pi_4(S^3)$ (2022) ✓

Recently, the Serre Finiteness Theorem was formalized (2025), computing $\pi_k(S^n)$ for all $k, n > 0$.

Summary

1. The idea that computability is modeled by continuity extends to all of constructive type theory.
2. Type theoretic constructions become homotopy invariant structures and theorems.
3. Constructive proofs yield programs for calculating e.g. homotopy invariants in a computational proof system.
4. The calculations of $\pi_k(S^n)$ were a proof of concept of HoTT, which still remains experimental.
5. Classical foundations using *sets* (as in Lean) form a subsystem of constructive foundations using ∞-*groupoids* and *homotopy types* (as in HoTT).

Gottlob Frege



I am convinced that my Begriffsschrift will find successful application wherever particular value is placed on the rigor of proofs, as in the foundations of the differential and integral calculus. It seems to me that it would be even easier to extend the domain of this formal language to geometry. Only a few more symbols would need to be added for the intuitive relations occurring there. In this way, one would obtain a kind of analysis situs.

Preface to Begriffsschrift, 1879

Thanks!

For more information:

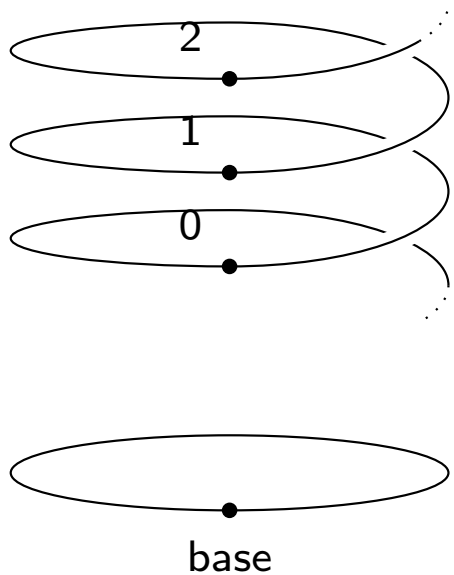
`HomotopyTypeTheory.org`

Some References

- A. Abel, A. Vezzosi and A. Mörtberg. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. J. Functional Programming (2019).
- S. Awodey, T. Coquand. Univalent foundations and the large scale formalization of mathematics. The IAS Letter (Spring 2013).
- S. Awodey, M. Warren. Homotopy theoretic models of identity types, Mathematical Proceedings of the Cambridge Philosophical Society (2009).
- M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. TYPES 2014.
- G. Brunerie. The James construction and $\pi_4(S^3)$. Institute for Advanced Study, March 2013.
- C. Cohen, T. Coquand, S. Huber and A. Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. TYPES 2015.
- B. van den Berg, R. Garner. Types are weak ω-groupoids, Proceedings of the London Mathematical Society (2011).
- A. Ljungström and A. Mörtberg. Formalizing $\pi_4(S^3) = \mathbb{Z}/_{2\mathbb{Z}}$ and Computing a Brunerie Number in Cubical Agda (2023).
- P. LeFanu Lumsdaine. Weak ω-categories from intensional type theory, Logical Methods in Computer Science (2010).
- The Univalent Foundations Program. Homotopy Type Theory: Univalent Foundations of Mathematics, Institute for Advanced Study (2013).

Appendix: $\pi_1(S^1)$ in HoTT

To compute the fundamental group of the circle S^1, as in classical algebraic topology we shall use the universal cover:



In HoTT, this will be a dependent type over S^1, so a type family,

$$\mathrm{cov} : S^1 \longrightarrow \mathcal{U}.$$

Appendix: $\pi_1(S^1)$ in HoTT

To define such a type family $\mathrm{cov} : S^1 \longrightarrow \mathcal{U}$, by the recursion property of the circle, we need the following data:

- a point $A : \mathcal{U}$
- a loop $p : \mathtt{Id}_{\mathcal{U}}(A, A)$

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

By the univalence axiom, to give a loop $p : \mathtt{Id}_{\mathcal{U}}(\mathbb{Z}, \mathbb{Z})$ 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 $\mathrm{succ} : \mathbb{Z} \cong \mathbb{Z}$.

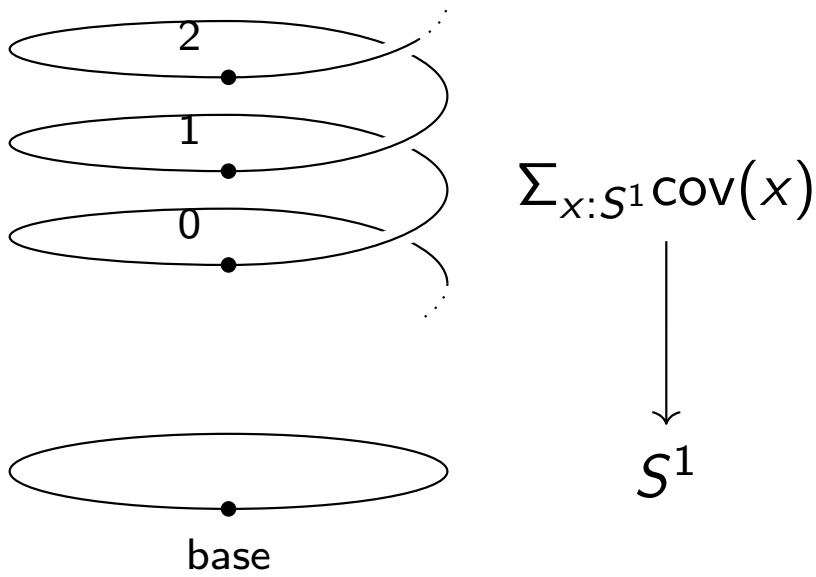
Appendix: $\pi_1(S^1)$ in HoTT

Definition (Universal Cover of S^1)

The dependent type $\mathsf{cov} : S^1 \longrightarrow \mathcal{U}$ is given by circle recursion, with:

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

Appendix: $\pi_1(S^1)$ in HoTT



Now we use cov to define the "winding number" of any path $p : \mathtt{Id}_{S^1}(\mathsf{base}, \mathsf{base})$ by $\mathsf{wind}(p) = p_*(0)$. This gives a map

$$\mathsf{wind} : \mathtt{Id}_{S^1}(\mathsf{base}, \mathsf{base}) \longrightarrow \mathbb{Z}.$$

The map wind is inverse to the map $\mathbb{Z} \longrightarrow \mathtt{Id}_{S^1}(\mathsf{base}, \mathsf{base})$ given by iterated composition of paths,

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

Shulman's Coq proof

```
(* *** 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.
```

```
(* 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.
```

Appendix 2: Dependent Types in HoTT

A consequence of the interpretation of identity terms as *paths* is the interpretation of dependent types as *fibrations*.

A type family $x : X \vdash P(x)$ should be interpreted as a "continuously varying family of spaces", which we can take to be a continuous map:

$$x : X \vdash P(x) \quad \rightsquigarrow \quad \begin{array}{c} P \\ \big\downarrow \\ X \end{array}$$

Appendix 2: Dependent Types in HoTT

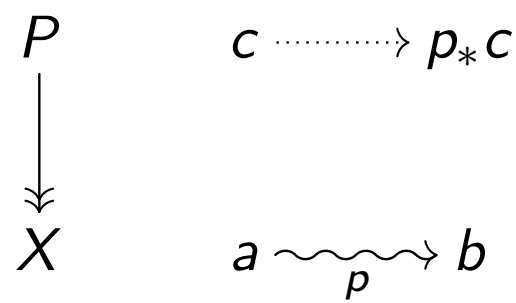
The rules for identity types permit the inference:

$$\frac{p : \mathrm{Id}_X(a,b) \qquad\qquad c : P(a)}{p_* c : P(b)}$$

This says the predicate $P(x)$ *respects identity*:

$$\mathrm{Id}_X(a,b)\ \&\ P(a)\ \Rightarrow\ P(b)$$

Topologically, it is the *path lifting property* of a fibration:



To lift the path $p : a \sim b$ use the pathspace $x : X \vdash \mathrm{Id}_X(a,x)$.