Recent Work in Homotopy Type Theory

Steve Awodey
Carnegie Mellon University

AMS Baltimore
January 2014

Introduction

- *Homotopy Type Theory* is a newly discovered connection between logic and topology, based on an interpretation of constructive type theory into homotopy theory.
- *Univalent Foundations* is a program for comprehensive foundations of mathematics based on HoTT.
- A large amount of mathematics has already been developed in this new foundational system, including some basic results in homotopy theory.
- Proofs are formalized and verified in an extension of the Coq proof assistant, suitably modified for UF.

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** $s = t : A$

Intended as a foundation for constructive mathematics, but now also used extensively in programming languages.

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 also 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 terms:

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

This type represents the **logical** proposition "x is identical to y".

Question: What is the mathematical interpretation of $\mathtt{Id}_A(x, y)$?

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:

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

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 (Awodey-Warren)

This extends the topological interpretation of the (**simply-typed**) λ-calculus:

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

to (**dependently-typed**) λ-calculus with `Id`-**types** via the **new idea**:

$p : \mathtt{Id}_X(a, b) \;\Leftrightarrow$

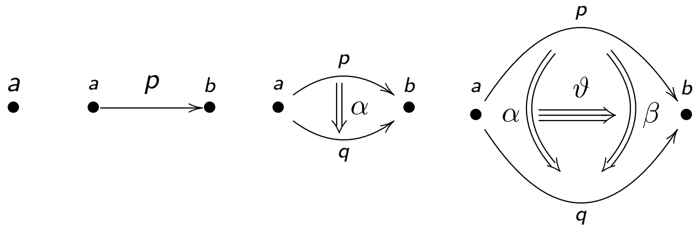
p is a path from point a to point b in the space X

This forces:

- dependent types to be *fibrations*,
- `Id`-types to be *path spaces*,
- general terms of `Id`-types to be *homotopies*.

The fundamental groupoid of a type (Hofmann-Streicher)

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



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

$$
\begin{array}{rll}
r : \mathtt{Id}(a, a) & \text{reflexivity} & a \to a \\
s : \mathtt{Id}(a, b) \to \mathtt{Id}(b, a) & \text{symmetry} & a \leftrightarrows b \\
t : \mathtt{Id}(a, b) \times \mathtt{Id}(b, c) \to \mathtt{Id}(a, c) & \text{transitivity} & a \to b \to c
\end{array}
$$

The **groupoid equations** only hold "up to homotopy".

Fundamental ∞-groupoids

The entire system of identity terms of all orders forms an infinite-dimensional graph, or **globular set**:

$$A \leftleftarrows \mathtt{Id}_A \leftleftarrows \mathtt{Id}_{\mathtt{Id}_A} \leftleftarrows \mathtt{Id}_{\mathtt{Id}_{\mathtt{Id}_A}} \leftleftarrows \dots$$

It has the structure of a (weak), infinite-dimensional, groupoid, as occurring homotopy theory:

Theorem (Lumsdaine, Garner & van den Berg, 2009)

The system of identity terms of all orders over any fixed type is a weak ∞-groupoid.

Every type has a **fundamental weak ∞-groupoid**.

Homotopy *n*-types (Voevodsky)

The universe of all types is stratified by homotopical truncation, which is logically definable.

A type X is called:

contractible iff $\sum_{x:X} \prod_{y:X} \mathtt{Id}_X(x, y)$ is inhabited,

A type X is called a:

proposition iff $\mathtt{Id}_X(x, y)$ is contractible for all $x, y : X$,

set iff $\mathtt{Id}_X(x, y)$ is a proposition for all $x, y : X$,

1-type iff $\mathtt{Id}_X(x, y)$ is a set for all $x, y : X$,

(n+1)-type iff $\mathtt{Id}_X(x, y)$ is an n-type for all $x, y : X$.

We then let set = 0-type, and proposition = (-1)-type.
This corresponds to the homotopical notion of **truncation**,
the level at which the fundamental groupoid becomes trivial.

Homotopy type theory: Summary

- Constructive type theory has an interpretation into homotopy theory.
- Logical methods capture some homotopical concepts: e.g. the fundamental ∞-groupoid of a space and the notion of a homotopy n-type are *logically definable*.
- Many basic results have already been formalized: Homotopy groups of spheres $\pi_k(S^n)$, Hopf fibration, Freudenthal suspension theorem, Eilenberg–Mac Lane spaces, ...
- Other areas are 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.

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:

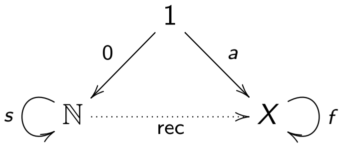
$$\frac{a : X \qquad f : X \to X}{\mathrm{rec}(a, f) : \mathbb{N} \to X}$$

with computation rules:

$$\mathrm{rec}(a, f)(0) = a$$
$$\mathrm{rec}(a, f)(sn) = f(\mathrm{rec}(a, f)(n))$$

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.

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:

$$\mathbb{S} := \begin{cases} \mathsf{base} : \mathbb{S} \\ \mathsf{loop} : \mathsf{base} \rightsquigarrow \mathsf{base} \end{cases}$$

where we write "base $\rightsquigarrow$ base" for "$\mathtt{Id}_{\mathbb{S}}(\mathsf{base}, \mathsf{base})$".

Higher inductive types: The circle S^1

$$\mathbb{S} := \begin{cases} \mathsf{base} : \mathbb{S} \\ \mathsf{loop} : \mathsf{base} \rightsquigarrow \mathsf{base} \end{cases}$$

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

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

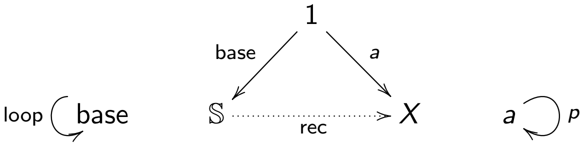
with computation rules:

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

(The map $\mathsf{rec}(a, p)$ acts on loop via the `Id`-elimination rule.)

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:

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

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

Higher inductive types: The interval I

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

$$\mathbb{I} := \begin{cases} 0, 1 : \mathbb{I} \\ p : 0 \rightsquigarrow 1 \end{cases}$$

again writing $0 \rightsquigarrow 1$ for the type $\mathtt{Id}_{\mathbb{I}}(0, 1)$.

Slogan:

In classical 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 more general quotients
- higher homotopy groups, Eilenberg-Mac Lane spaces, Postnikov systems
- Quillen model structure.

These are mostly ad hoc — the general theory is still a work in progress.

Univalence

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

- It captures the informal practice of **identifying isomorphic objects**.
- It is formally **incompatible** with set theoretic foundations.
- It is formally **consistent** with homotopy type theory.
- It has powerful consequences, especially together with HITs.

Isomorphism and Equivalence

The notion of *type isomorphism* $A \cong B$ is definable as usual:

$$A \cong B \;\Leftrightarrow\; \textit{there are } f : A \to B \textit{ and } g : B \to A \\ \textit{such that } gfx = x \textit{ and } fgy = y.$$

Formally, there is a type of isomorphisms:

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

We say that $A \cong B$ if this type is inhabited by a closed term, which is then 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 $gfx = x$ and $fgy = y$.
- Under the homotopy interpretation, this is the type of *homotopy equivalences* between the spaces A and B.
- Depending on the n-type of A and B, this also subsumes:
  - *categorical equivalence* $(n = 1)$,
  - *isomorphism of sets* $(n = 0)$,
  - *logical equivalence* $(n = -1)$.

Univalence

Question: How is equivalence related to *identity of types*?

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:

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

It can thus be stated: *"Identity is equivalent to equivalence."*

The Univalence Axiom: Remarks

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

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

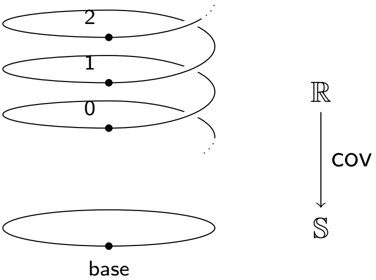
  So **equivalent objects are identical**.
  (In particular, isomorphic sets, groups, etc., get identified.)
- UA is *equivalent* to the following **invariance property**:

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

  for all definable properties $P(-)$ of types.
- UA is incompatible with the assumption that everything is a set (0-type), but it is consistent with general HoTT.
- The **computational character** of UA is an open question.

The Univalence Axiom: How it works

To compute the fundamental group of the circle $\mathbb{S}$, we first 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

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

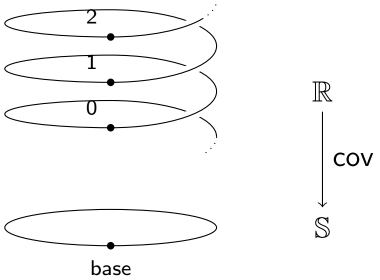
by the recursion property of the circle, we just need the following data:

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

We have:

- For the point A we take the integers $\mathbb{Z}$.
- By UA, 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

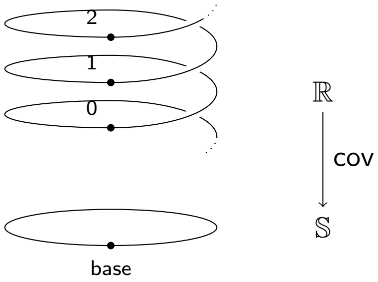


Definition (Universal Cover of $\mathbb{S}^1$)

The dependent type 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



Then, as usual, we can define the "winding number" of a path $p : \mathsf{base} \rightsquigarrow \mathsf{base}$ to give a map

$$\mathsf{wind} : (\mathsf{base} \rightsquigarrow \mathsf{base}) \longrightarrow \mathbb{Z},$$

which is inverse to the map $z \mapsto \mathsf{loop}^z$.

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.
```

Univalent Foundations: Summary

- Explicit logical foundations are now *feasible*, because computers can take over what was once too tedious or complicated to be done by hand.
- Formalization can provide a *practical tool* for working mathematicians: increased certainty and precision, supports collaborative work, cumulativity of results, searchable library of code, ... Mathematics could eventually be fully formalized.
- UF uses a “synthetic” method involving high-level axiomatics and direct, structural descriptions; allows shorter, more abstract proofs; closer to mathematical practice than the “analytic” method of ZFC.
- Use of UA is very powerful.

References and Further Information

General information:

`www.HomotopyTypeTheory.org`

Current state of the Univalent Foundations Program:

`uf-ias-2012.wikispaces.com`

The Book:

Homotopy Type Theory:
Univalent Foundations of Mathematics

Homotopy Type Theory

Univalent Foundations of Mathematics

THE UNIVALENT FOUNDATIONS PROGRAM
INSTITUTE FOR ADVANCED STUDY