2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science

Calculating the Fundamental Group of the Circle in Homotopy Type Theory

Daniel R. Licata
Institute for Advanced Study
drl@cs.cmu.edu

Michael Shulman
Institute for Advanced Study
mshulman@ias.edu

***Abstract*—Recent work on homotopy type theory exploits an exciting new correspondence between Martin-Lof's dependent type theory and the mathematical disciplines of category theory and homotopy theory. The mathematics suggests new principles to add to type theory, while the type theory can be used in novel ways to do computer-checked proofs in a proof assistant. In this paper, we formalize a basic result in algebraic topology, that the fundamental group of the circle is the integers. Our proof illustrates the new features of homotopy type theory, such as higher inductive types and Voevodsky's univalence axiom. It also introduces a new method for calculating the path space of a type, which has proved useful in many other examples.**

I. Introduction

Recently, researchers have discovered an exciting new correspondence between Martin-Löf's intensional dependent type theory and the mathematical disciplines of category theory and homotopy theory [2, 4, 5, 6, 10, 11, 22, 23, 24]. Under this correspondence, a type A in dependent type theory carries the structure of an ∞-groupoid, or a topological space up to homotopy. Terms $M : A$ correspond to objects of the groupoid, or points in the topological space. Terms of the identity type, written $\alpha : \mathsf{Id}_A(M,N)$, correspond to morphisms, or paths in the topological space, between M and N. Iterating the identity type gives further structure; for example, the type $\mathsf{Id}_{\mathsf{Id}_A(M,N)}(\alpha,\beta)$ represents higher-dimensional morphisms, or homotopies (continuous deformations) between paths. This correspondence has many applications: The category theory and homotopy theory suggest new principles to add to type theory, such as higher-dimensional inductive types [12, 13, 18] and Voevodsky's univalence axiom [7, 23]. Proof assistants such as Coq [3] and Agda [15], especially when extended with these new principles, can be used in novel ways to formalize category theory, homotopy theory, and mathematics in general.

Here, we consider the use of type theory for computer-checked proofs in homotopy theory, using a *synthetic* approach. Synthetic geometry is geometry in the style of Euclid, where one starts from some basic notions (points and lines), constructions (two points determine a line), and axioms (two right angles are equal), and deduces consequences logically. In synthetic homotopy theory, we take homotopy-theoretic concepts such as spaces, points, paths, and homotopies as basic notions/constructions, and deduce consequences logically, using type theory as a *logic of homotopy theory*. To illustrate this synthetic approach, we compute what is called the *fundamental group* of the circle: Given a topological space X with a particular *base point* $x_0 \in X$, the *loops* in X are the continuous paths from x_0 to itself. These loops (considered up to homotopy) have the structure of a group: there is an identity path (standing still), composition (go along one loop and then another), and inverses (go backwards along a loop). The group of homotopy-equivalence-classes of loops is called the *fundamental group* of X at x_0, denoted $\pi_1(X,x_0)$ or just $\pi_1(X)$. More generally, by considering higher-dimensional paths and deformations, one obtains the *higher homotopy groups* $\pi_n(X)$. Characterizing these is a central question in homotopy theory; they are surprisingly complex even for a space as simple as the sphere.

Consider the circle (written S^1) with some fixed base point base. What paths are there from base to base? Some possibilities include standing still, going around clockwise once, or twice, or three times in a row; or going counterclockwise once, twice, etc. However, up to homotopy, going around clockwise and then counterclockwise (or vice versa) is the identity: we can deform this path continuously back to the constant one. Thus, the clockwise and counterclockwise paths are inverses in $\pi_1(\mathsf{S}^1)$. This suggests that $\pi_1(\mathsf{S}^1)$ should be isomorphic to $\mathbb{Z}$, the additive group of the on the integers: a path is determined by its *winding number*, which determines whether it stands still (0), goes around counterclockwise n times ($+n$), or goes around clockwise n times ($-n$). Proving this is one of the first basic theorems of algebraic topology.

In this paper, we give a computer-checked synthetic proof of this result in type theory, using Agda. In the synthetic approach, the circle is not defined concretely as a subset of the real plane, but abstractly, as a *higher inductive type* inductively generated by a point and a loop, making use of the ∞-groupoid structure provided by the identity type. Homotopy type theory has a model in Kan simplicial sets [7], and combining our proof, this model, and the *geometric realization* functor from simplicial sets to topological spaces does show that $\pi_1(\mathsf{S}^1) = \mathbb{Z}$ for the familiar topological circle. But the synthetic approach is much more amenable to formalization in a proof assistant,

This material is based in part upon work supported by the National Science Foundation under grants CCF-1116703 and DMS-1128155 and a Mathematical Sciences Postdoctoral Research Fellowship, by the Institute for Advanced Study's Oswald Veblen fund, and by the CMU-Microsoft Center for Computational Thinking. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or any other sponsoring entity.

1043-6871/13 $26.00 © 2013 IEEE
DOI 10.1109/LICS.2013.28

and has several additional interesting aspects.

First, our proof shows that the result holds not only in topological spaces, but in any model of homotopy type theory [7, 20]. Second, the proof has computational content: it is also a program that converts a path on the circle to its winding number, and vice versa. While the computational interpretation of full homotopy type theory is an open problem, existing work [10] covers this particular proof. Third, and most importantly, the proof mixes classical homotopy-theoretic techniques with new type-theoretic ones. In this paper, we introduce a technique we call the *encode-decode method* for characterizing path spaces. In subsequent work [21], this method has proved useful for formalizing a variety of basic results in algebraic topology, including calculations of many more homotopy groups of spheres ($\pi_{k<n}(S^n)$, $\pi_n(S^n)$), the Freudenthal suspension theorem, the Blakers–Massey theorem, and van Kampen's theorem, and a construction of Eilenberg–Mac Lane spaces. This method can be seen as a generalization of techniques for proving injectivity of disjointness of the constructors of an inductive type [14], and provides a context for understanding the familiar puzzle that a universe (type of types) is necessary to prove these properties.

The remainder of this paper is organized as follows. In Section II, we introduce the basic definitions of homotopy type theory. In Section III, we introduce the encode-decode method, proving injectivity and disjointness for the constructors of the coproduct type. In Section IV, we define the circle as a higher-dimensional inductive type, and in Section V we prove that its fundamental group is $\mathbb{Z}$.

II. Basics of Homotopy Type Theory

In this section, we introduce the basics of homotopy type theory (see the Homotopy Type Theory book [21] for a more thorough introduction).

A. Types as Spaces

The identity type (or *propositional equality* type) in Martin-Löf's intensional type theory equips each type with the structure of a *homotopy type*, or, equivalently, an ∞-*groupoid*. To emphasize the homotopy-theoretic interpretation, we write the identity type between elements $\mathsf{M},\mathsf{N} : \mathsf{A}$ as $\mathsf{Path\ M\ N}$ (leaving A as an implicit argument[1]). Path is defined as an inductive family of types, with one constructor id:

```
data Path {A : Type} : A → A → Type where
  id : {M : A} → Path M M
```

id (reflexivity of propositional equality) represents the identity path in the type A.

[1]We assume basic familiarity with Agda; see Norell [16] for an introduction. Curly-braces are Agda notation for an implicit parameter: we write Path M N when A can be inferred, or Path{A} M N to explicitly notate it. We rename Agda's universe Set (the type of smaller types) to Type, because the word "set" has another meaning in this context [21]. Agda is a predicative theory, with explicit universe polymorphism, and ordinarily one would define Path in a universe-polymorphic manner. To avoid cluttering the presentation, we use Agda's (inconsistent) `--type-in-type` flag to suppress universe levels, but the development can be done without this cheat.

The standard J elimination rule for the identity type (in the Paulin-Mohring "one-sided" form [17]) expresses that the paths from a fixed point M, to a variable endpoint x, are inductively generated by M and id. We call this *path induction*:

```
path-induction : {A : Type} {M : A}
  (C : (x : A) → Path M x → Type) (b : C M id)
  {N : A} (α : Path M N) → C N α
path-induction _ b id = b
```

This function is defined by *pattern-matching*: we give cases for all possible constructors for α, which in this case is just id. In each case, the type of the right-hand side is specialized to that constructor; in this case, C N α is specialized to C M id. It is crucial that path-induction only applies to a family C that is parametrized by a variable x standing for the second endpoint—otherwise, it would imply that all loops of type Path M M are generated by id, which would be incompatible with the homotopy-theoretic interpretation. Topologically, the intuition is that a path with a *free endpoint* can be retracted back to the other, but a path with two fixed endpoints cannot.

Paths have a groupoid structure, with inverses and composition defined as follows:

```
! : {A : Type} {M N : A} → Path M N → Path N M
! id = id
_∘_ : {A : Type} {M N P : A} → Path N P → Path M N → Path M P
β ∘ id = β
```

The groupoid laws hold only up to homotopy; in type theory, this means they are not definitional equalities, but are witnessed by propositional equalities, or paths between paths. For example, we can prove associativity, unit, and inverse laws for ∘ ! and id (we omit the symmetric ∘-unit-r and !-inv-r):

```
∘-unit-l : {A : Type} {M N : A} (α : Path M N)
         → Path (id ∘ α) α
∘-unit-l id = id
∘-assoc : {A : Type} {M N P Q : A}
          (γ : Path P Q) (β : Path N P) (α : Path M N)
          → Path (γ ∘ (β ∘ α)) ((γ ∘ β) ∘ α)
∘-assoc id id id = id
!-inv-l    : {A : Type} {M N : A} (α : Path M N)
           → Path (! α ∘ α) id
!-inv-l id = id
```

B. Dependent Types as Fibrations

Just as types act like groupoids, type families act like indexed families of groupoids. In particular, they vary functorially with paths in the indexing type:

```
transport : {B : Type} (E : B → Type)
            {b1 b2 : B} → Path b1 b2 → (E b1 → E b2)
transport C id = λ x → x
```

transport is functorial up to homotopy; e.g. there is a path between transport C ($\beta \circ \alpha$) and transport C β o transport C α, where o is function composition. Logically, transport is a "coercion" by propositional equality.

While the category-theoretic viewpoint on a type family $\mathsf{E} : \mathsf{B} \to \mathsf{Type}$ is as a functor from B to types, the topological

viewpoint is as a *fibration*. Given two spaces $\tilde{E}$ and B, a fibration over B is a continuous map $p : \tilde{E} \to B$ such that for any $e \in \tilde{E}$, any path in B starting at $p(e)$ has a lifting to a path in $\tilde{E}$ starting at e (and these liftings are continuous/functorial). $\tilde{E}$ is called the *total space*, while B is called the *base space*.

To make the connection with type theory, it is helpful to consider a slightly different characterization of fibrations: Given a point b in the base space B, the *fiber over* b is the space of points in the total space $\tilde{E}$ that p maps to b (i.e. the inverse image $p^{-1}(b)$). Any path β from b_1 to b_2 in B induces an operation from the fiber over b_1 to the fiber over b_2. Namely, given $e \in p^{-1}(b_1)$, we lift β starting at e and consider the other endpoint of the resulting path; this is well-defined up to homotopy. These operations respect path composition, inversion, and so on, in a homotopical way, yielding a "functor" E sending each $b \in B$ to its fiber $p^{-1}(b)$. Conversely, from any such functor E we can assemble a total space $\tilde{E}$ and a fibration $p : \tilde{E} \to B$ with the specified fibers and path-lifting functions (for groupoids, this is called the *Grothendieck construction*).

In type theory, the functor description E corresponds directly to a dependent type $\mathsf{E} : \mathsf{B} \to \mathsf{Type}$, where the type $\mathsf{E}\,\mathsf{b}$ represents the fiber over b, and $\mathsf{transport}\,\mathsf{E}\,\alpha$ represents the operation induced by path lifting. Given such an E, the fibration $p : \tilde{E} \to B$ is modeled by the Σ-type $\Sigma b{:}B.E(b)$ and its first projection $\mathsf{fst} : \Sigma b{:}B.E(b) \to B$. Thus total spaces are Σ-types, which we will sometimes write as $\Sigma\,\mathsf{E}$.

For each specific dependent type, we can give a rule stating how $\mathsf{transport}$ acts, expressing the definition of path lifting for the corresponding fibration. These rules are like computation steps for $\mathsf{transport}$, though they are paths, not definitional equalities. Here, we need the following two rules:

$$\begin{aligned}
&\mathsf{transport\text{-}Path\text{-}right} : \{\mathsf{A} : \mathsf{Type}\}\,\{\mathsf{M}\,\mathsf{N}\,\mathsf{P} : \mathsf{A}\} \\
&\quad (\alpha' : \mathsf{Path}\,\mathsf{N}\,\mathsf{P})\,(\alpha : \mathsf{Path}\,\mathsf{M}\,\mathsf{N}) \\
&\quad \to \mathsf{Path}\,(\mathsf{transport}\,(\lambda\,\mathsf{x} \to \mathsf{Path}\,\mathsf{M}\,\mathsf{x})\,\alpha'\,\alpha)\,(\alpha' \circ \alpha) \\
&\mathsf{transport}\text{-}{\to} : \{\Gamma : \mathsf{Type}\}\,(\mathsf{A}\,\mathsf{B} : \Gamma \to \mathsf{Type})\,\{\theta 1\,\theta 2 : \Gamma\} \\
&\qquad\qquad (\delta : \theta 1 \simeq \theta 2)\,(\mathsf{f} : \mathsf{A}\,\theta 1 \to \mathsf{B}\,\theta 1) \\
&\quad \to \mathsf{Path}\,(\mathsf{transport}\,(\lambda\,\gamma \to (\mathsf{A}\,\gamma) \to \mathsf{B}\,\gamma)\,\delta\,\mathsf{f}) \\
&\qquad\qquad (\mathsf{transport}\,\mathsf{B}\,\delta \circ \mathsf{f} \circ (\mathsf{transport}\,\mathsf{A}\,(!\,\delta)))
\end{aligned}$$

The former says that transporting with the family $\mathsf{Path}\,\mathsf{M}$ - is post-composition of paths; the latter that transporting at $\mathsf{A} \to \mathsf{B}$ is given by pre-composing with transport at A (on the inverse) and post-composing with transport at B. Both are proved by matching the input paths as id and returning id.

C. Functions are Functorial

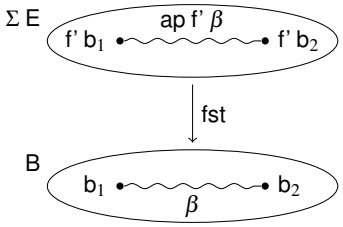
Simply-typed functions correspond to functors between ∞-groupoids, and have an action at all levels: a function $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ has an action on points, paths, paths between paths, etc. The action on points is ordinary function application ($\mathsf{f}\,\mathsf{a} : \mathsf{B}$ when $\mathsf{a} : \mathsf{A}$), while the action on paths is given by

$$\begin{aligned}
&\mathsf{ap} : \{\mathsf{A}\,\mathsf{B} : \mathsf{Type}\}\,\{\mathsf{M}\,\mathsf{N} : \mathsf{A}\} \\
&\qquad (\mathsf{f} : \mathsf{A} \to \mathsf{B}) \to \mathsf{Path}\,\{\mathsf{A}\}\,\mathsf{M}\,\mathsf{N} \to \mathsf{Path}\,\{\mathsf{B}\}\,(\mathsf{f}\,\mathsf{M})\,(\mathsf{f}\,\mathsf{N}) \\
&\mathsf{ap}\,\mathsf{f}\,\mathsf{id} \;=\; \mathsf{id}
\end{aligned}$$

ap acts functorially on identities and composition of paths, and also on identities and composition of functions.

Dependently typed functions $\mathsf{f} : (\mathsf{b} : \mathsf{B}) \to \mathsf{E}\,(\mathsf{b})$ correspond to sections of the fibration E. In type-theoretic language, such a section is a simply-typed function $\mathsf{f'} : \mathsf{B} \to \Sigma\,\mathsf{E}$ such that $\mathsf{fst} \circ \mathsf{f'} = (\lambda\,\mathsf{x} \to \mathsf{x})$ *definitionally*—the first component of the result of $\mathsf{f'}$ must be its argument. Given a dependently typed f, we can define a section $\mathsf{f'} : \mathsf{B} \to \Sigma\,\mathsf{E}$ by $\lambda\,\mathsf{b} \to (\mathsf{b}, \mathsf{f}\,\mathsf{b})$.

This correspondence helps us arrive at the dependently typed analogue of ap. When f has a dependent function type, and $\beta : \mathsf{Path}\,\{\mathsf{B}\}\,\mathsf{b}_1\,\mathsf{b}_2$, the applications $\mathsf{f}\,\mathsf{b}_1$ and $\mathsf{f}\,\mathsf{b}_2$ should still be related by a path, but $\mathsf{f}\,\mathsf{b}_1 : \mathsf{E}\,\mathsf{b}_1$ and $\mathsf{f}\,\mathsf{b}_2 : \mathsf{E}\,\mathsf{b}_2$ live in different fibers, i.e. they have different types. By turning f into a section $\mathsf{f'}$, we have $\mathsf{ap}\,\mathsf{f'}\,\beta : \mathsf{Path}\,\{\Sigma\,\mathsf{E}\}\,(\mathsf{b}_1, \mathsf{f}\,\mathsf{b}_1)\,(\mathsf{b}_2, \mathsf{f}\,\mathsf{b}_2)$—there is a path *in the total space* between $(\mathsf{b}_1, \mathsf{f}\,\mathsf{b}_1)$ and $(\mathsf{b}_2, \mathsf{f}\,\mathsf{b}_2)$. However, because $\mathsf{fst} \circ \mathsf{f'} = (\lambda\,\mathsf{x} \to \mathsf{x})$, it follows from functoriality of ap that $\mathsf{ap}\,\mathsf{fst}\,(\mathsf{ap}\,\mathsf{f'}\,\beta)$ equals β. Topologically, this means that $\mathsf{ap}\,\mathsf{f'}\,\beta$ *projects down to* β, or *sits above* β:



Thus, one way to describe the result of applying f to β is as a path $\tilde{\eta}$ in $\Sigma\,\mathsf{B}$ such that $\mathsf{ap}\,\mathsf{fst}\,\tilde{\eta}$ is β. However, there is a more direct representation: we can represent a path between $\mathsf{e}_1 : \mathsf{E}\,\mathsf{b}_1$ and $\mathsf{e}_2 : \mathsf{E}\,\mathsf{b}_2$ sitting above $\beta : \mathsf{Path}\,\mathsf{b}_1\,\mathsf{b}_2$ by an $\eta : \mathsf{Path}\,\{\mathsf{E}\,\mathsf{b}_2\}\,(\mathsf{transport}\,\mathsf{E}\,\beta\,\mathsf{e}_1)\,\mathsf{e}_2$. That is, we use $\mathsf{transport}$ to move e_1 into the fiber over b_2 and then give a path in $\mathsf{E}\,\mathsf{b}_2$. This representation is correct because $(\mathsf{b}_1, \mathsf{e}_1)$ is always connected to $(\mathsf{b}_2, \mathsf{transport}\,\mathsf{E}\,\beta\,\mathsf{e}_1)$ by a path over β (this follows by path induction on β), and any path $\tilde{\eta}$ from $(\mathsf{b}_1, \mathsf{e}_1)$ to $(\mathsf{b}_2, \mathsf{e}_2)$ in $\Sigma\,\mathsf{E}$ over β factors as this path followed by our path η in the fiber over b_2.

This discussion motivates the following dependently typed analogue of ap: applying f to β must determine a path between $\mathsf{f}\,\mathsf{b}_1$ and $\mathsf{f}\,\mathsf{b}_2$ that sits above β, which is represented by the following type:

$$\begin{aligned}
&\mathsf{apd} : \{\mathsf{B} : \mathsf{Type}\}\,\{\mathsf{E} : \mathsf{B} \to \mathsf{Type}\}\,\{\mathsf{b}_1\,\mathsf{b}_2 : \mathsf{B}\} \\
&\qquad (\mathsf{f} : (\mathsf{x} : \mathsf{B}) \to \mathsf{E}\,\mathsf{x})\,(\beta : \mathsf{Path}\,\mathsf{b}_1\,\mathsf{b}_2) \\
&\quad \to \mathsf{Path}\,(\mathsf{transport}\,\mathsf{E}\,\beta\,(\mathsf{f}\,\mathsf{b}_1))\,(\mathsf{f}\,\mathsf{b}_2) \\
&\mathsf{apd}\,\mathsf{f}\,\mathsf{id} \;=\; \mathsf{id}
\end{aligned}$$

D. Paths Between Functions

Another kind of application is applying a path between functions to an argument, to get a path between results:

$$\begin{aligned}
&\mathsf{ap}{\simeq} : \forall\,\{\mathsf{A}\}\,\{\mathsf{B} : \mathsf{A} \to \mathsf{Type}\}\,\{\mathsf{f}\,\mathsf{g} : (\mathsf{x} : \mathsf{A}) \to \mathsf{B}\,\mathsf{x}\} \\
&\quad \to \mathsf{Path}\,\mathsf{f}\,\mathsf{g} \to \{\mathsf{x} : \mathsf{A}\} \to \mathsf{Path}\,(\mathsf{f}\,\mathsf{x})\,(\mathsf{g}\,\mathsf{x}) \\
&\mathsf{ap}{\simeq}\,\alpha\,\{\mathsf{x}\} \;=\; \mathsf{ap}\,(\lambda\,\mathsf{f} \to \mathsf{f}\,\mathsf{x})\,\alpha
\end{aligned}$$

$$\begin{aligned}
&\lambda{\simeq} : \forall\,\{\mathsf{A}\}\,\{\mathsf{B} : \mathsf{A} \to \mathsf{Type}\}\,\{\mathsf{f}\,\mathsf{g} : (\mathsf{x} : \mathsf{A}) \to \mathsf{B}\,\mathsf{x}\} \\
&\quad \to ((\mathsf{x} : \mathsf{A}) \to \mathsf{Path}\,(\mathsf{f}\,\mathsf{x})\,(\mathsf{g}\,\mathsf{x})) \\
&\quad \to \mathsf{Path}\,\mathsf{f}\,\mathsf{g}
\end{aligned}$$

Function extensionality $\lambda\simeq$ is the converse—functions are equal if they are pointwise equal, or a path between functions is a homotopy between them. Agda does not provide this, so we postulate it.[2] We should also give β/η-like equations relating $\lambda\simeq$ and ap$\simeq$, but we do not need them in this paper.

E. Paths in the Universe

Voevodsky's univalence axiom says roughly that *isomorphic types are equal*, where "equal" means Path, and "isomorphic" means a *homotopy equivalence*,[3] or a pair of mutually inverse functions:

```
record HEquiv (A B : Type) : Type where
  constructor hequiv
  field
    f : A → B
    g : B → A
    α : (x : A) → Path (g (f x)) x
    β : (y : B) → Path (f (g y)) y
```

Rather than stating the univalence axiom in full generality, we specify only the consequences we need. First, a homotopy equivalence determines a path between types:

```
univalence : {A B : Type} → HEquiv A B → Path A B
```

We need two facts about this axiom:

```
transport-univ : {A B : Type} (e : HEquiv A B)
  → Path (transport (λ (A : Type) → A) (univalence e))
         (HEquiv.f e)
!-univalence : {A B : Type} (e : HEquiv A B)
  → Path (! (univalence e))
         (univalence (!-equiv e))
```

The first says that transporting with the identity type family $\lambda\ A \to A$ on an application of univalence applies the forward direction of the equivalence. The second says that the inverse of an application of univalence is the inverse equivalence of e, where !-equiv (hequiv f g α β) is hequiv g f β α.

F. Integers

We represent integers as follows:

```
data Positive : Type where
  One : Positive
  S : (n : Positive) → Positive
data Int : Type where
  Pos : (n : Positive) → Int
  Zero : Int
  Neg : (n : Positive) → Int
```

It is straightforward to define the successor, predecessor, and addition functions by case-analysis/induction, and to prove that succ and pred are mutually inverse, so we have

```
succ : Int → Int
pred : Int → Int
_+_ : Int → Int → Int
succ-pred : (n : Int) → Path (succ (pred n)) n
pred-succ : (n : Int) → Path (pred (succ n)) n
```

Therefore, we have a homotopy equivalence between Int and itself given by adding and subtracting 1:

```
succEquiv : HEquiv Int Int
succEquiv = hequiv succ pred pred-succ succ-pred
```

III. Injectivity and Disjointness for Coproducts

Consider the type of coproducts (binary sums):

```
data _+_ (A B : Type) : Type where
  Inl : A → A + B
  Inr : B → A + B
```

Given a universe (or large eliminations), one can prove that the constructors are disjoint (Inl a is never equal to Inr b) and injective (Inl a equals Inl a' only if a equals a'). In this section, we use injectivity and disjointness to introduce the method we will use to calculate $\pi_1(S^1)$.

Fix A, B, and a:A. Injectivity and disjointness of Inl can be phrased as follows (where Void is the empty type):

- *Injectivity:* If Path (Inl a) (Inl a') then Path a a'.
- *Disjointness:* If Path (Inl a) (Inr b) then Void.

Thus, we can regard the injectivity-and-disjointness problem as the question of characterizing the types Path (Inl a) e for all e. One way to do this is to define a family of types describing the desired characterization, which we call Code:

```
Code : A + B → Type
Code (Inl a') = Path a a'
Code (Inr _) = Void
```

Code is a family of types defined by case analysis (this is the step that is not possible without a universe). An element of the type Code e is a piece of data that represents a path in A + B from the fixed *base point* Inl a to the point e, so we say that such an element is a *code* for such a path.

Suppose that we can encode every path as a code:

```
encode : {e : A + B} → Path (Inl a) e → Code e
```

Then injectivity and disjointness follow immediately, by the expanding the definition of Code:

```
inj : {a' : A} → Path (Inl a) (Inl a') → Path a a'
inj {a'} = encode {Inl a'}
dis : {b : B} → Path (Inl a) (Inr b) → Void
dis {b} = encode {Inr b}
```

However, we can easily define encode by transport at Code:

```
encode α = transport Code α id
```

To encode α : Path (Inl a) e, we transport along α with the type family Code, which reduces the goal of constructing an element of Code e to that of Code (Inl a). But Code (Inl a) is just Path a a, so we can choose id to complete the proof.

[2] It is not strictly necessary to postulate it separately, because it follows from Voevodsky's univalence axiom.

[3] Homotopy equivalences have a slightly undesirable property: there can be multiple different g, α, and β showing that a given f is a homotopy equivalence. It is common to include an additional coherence cell that fixes this problem. However, any homotopy equivalence can be *improved* by constructing this coherence cell (at the cost of changing α or β), so we can suppress this detail.

Thus far, our proof is very similar to an instance of the "no confusion" construction by McBride et al. [14]: Code corresponds to their NoConfusion type family, and encode to showing that it is provable. Previous work stops here, having proved injectivity and disjointness—which suffices when equality has no meaningful computational content. However, when paths have real content, it is important to know not only that injectivity and disjointness exist, but that they are equivalences. For example, one might like to know that the paths in the coproduct between Inls are equivalent to the paths in A (i.e. that Path {A + B} (Inl a) (Inl a') is equivalent to Path a a'), so that one may reason about paths in A through their injection into the coproduct. To this end, we will show that encode is an equivalence:

```
enceqv : {e : A + B} → HEquiv (Path (Inl a) e) (Code e)
enceqv = hequiv encode decode
                decode-encode encode-decode
```

by defining decode and proofs decode-encode and encode-decode. decode is defined as follows:

```
decode : {e : A + B} → Code e → Path (Inl a) e
decode {Inl a'} α = ap Inl α
decode {Inr _} ()
```

When e is Inl a', we are given α of type Code (Inl a'), or Path a a'. Thus, we get a Path (Inl a) (Inl a') by applying Inl to α. When e is Inr, α has type Code (Inr -), or Void, so the case is vacuously true, which we notate with an *absurd pattern* ().

Next, we show that these two functions are mutually inverse. First, we show that starting from a code, decoding it as a path, and then re-encoding it gives back the original code. We write the proof using a chain of equations, where the notation x ≃⟨ a ⟩ y ≃⟨ b ⟩ z ■ means there is a path a from x to y and then b from y to z.

```
encode-decode : {e : A + B} (c : Code e)
  → encode {e} (decode {e} c) ≃ c
encode-decode {Inl a'} α =
  encode (decode α)                          -- (1)
    ≃⟨ id ⟩
  transport Code (ap Inl α) id               -- (2)
    ≃⟨ ap≃ (! (transport-ap-assoc' Code Inl α)) ⟩
  transport (Code o Inl) α id                -- (3)
    ≃⟨ id ⟩
  transport (λ a' → Path a a') α id          -- (4)
    ≃⟨ transport-Path-right α id ⟩
  α ∘ id                                     -- (5)
    ≃⟨ id ⟩
  α ■                                        -- (6)
encode-decode {Inr _} ()
```

The proof begins by casing on e. In the Inl a' case, α is a path from a to a'. Between steps (1) and (2), we expand the definitions of encode and decode, where for decode we know the case for Inl is selected. Between (2) and (3), we reassociate transport and ap: in general, transport (C o f) α is the same as transport C (ap f α). Between (3) and (4), we reduce the definition of transport on Inl. Between (4) and (5), we apply the fact that transporting at Path a - is post-composition. Between (5) and (6), we apply one of the unit laws for composition, which gives the result.

The Inr case is vacuously true, because in this case we have an element of Code (Inr -), which is the empty type.

Next, we show the opposite compostion:

```
decode-encode : {e : A + B} (α : Path (Inl a) e)
                → Path (decode {e} (encode {e} α)) α
```

Expanding the defnition of encode, we need a path from (decode (transport Code α id)) to α, where α is an arbitrary path from Inl (a) to e. The key idea is to apply *path induction*: To prove the goal for an arbitrary e : A + B and α : Path (Inl a) e, it suffices to consider the case where e is Inl a and α is id. In this case, we have to show that decode (encode id) is id, which is easy: it holds definitionally, because both transport and ap compute to id on id, so id proves the result. This argument is formalized as follows:

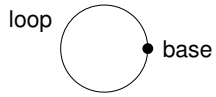
```
decode-encode {e} α =
  path-induction
  (λ e' α' → Path (decode {e'} (encode {e'} α')) α')
  id α
```

In summary, the structure of this proof is: (1) Fix a base point, and define a type of codes for paths from the base point. (2) Define encode by transporting along the codes family, with an appropriate base case. (3) Define decode by case-analysis/induction. (4) Prove that encoding after decoding is the identity, using the induction principle for codes. (5) Prove that decoding after encoding is the identity, using path induction. We follow this same *encode-decode method* for circle.

IV. The Circle

In this section, we introduce the representation of the circle in homotopy type theory. The natural numbers is an inductive type, with *generators* zero and successor. One of the new ingredients in homotopy type theory is *higher-dimensional inductive types* (or just *higher inductive types*) [12, 13, 18]: inductive types specified by generators not only for points (terms), but also for paths.

One might draw the circle like this:



This drawing has a single point, and a single non-identity loop from this base point to itself. This translates to a higher inductive type with two generators:

```
base : S¹
loop : Path {S¹} base base
```

base is like an ordinary constructor for an inductive type, with the same status as zero or successor. loop is similar, except it generates a path on the circle. This generator can be used with the generic groupoid structure to form additional paths, such as ! loop, loop ∘ loop, etc. Some of these paths are "reducible"—for example, loop ∘ ! loop is homotopic to id.

A. Simple Elimination

That the type of natural numbers is *inductively* generated by zero and successor is expressed by its elimination rule: to map from the natural numbers into a type X, it suffices to specify an element and an endomorphism of X, to be the images of zero and successor. Similarly, the elimination rule for S^1 expresses that the circle is inductively generated by base and loop: to map from the circle into any other type, it suffices to find a point and a loop in that type:

$$\begin{aligned} S^1\text{-recursion} : \{X : \mathsf{Type}\} \\ (\mathsf{base'} : X)\ (\mathsf{loop'} : \mathsf{Path\ base'\ base'}) \\ \to S^1 \to X \end{aligned}$$

For an inductive type, the general pattern of the β-reduction rules is that an application of the elimination rule to a generator computes to the corresponding branch. Applying this pattern to this case, S^1-recursion should compute to base' when applied to base and to loop' when applied to loop:

$$\begin{aligned} (S^1\text{-recursion base' loop'})\ \mathsf{base} &= \mathsf{base'} \\ (S^1\text{-recursion base' loop'})\ \mathsf{loop} &= \mathsf{loop'} \end{aligned}$$

However, the second equation does not quite make sense, because S^1-recursion base' loop' is a function $S^1 \to X$ but loop is a *path* in S^1. The solution is to use ap to apply this function to the path:

$$\mathsf{ap}\ (S^1\text{-recursion base' loop'})\ \mathsf{loop} = \mathsf{loop'}$$

The left-hand side of this equation is a loop at $((S^1\text{-recursion base' loop'})\ \mathsf{base})$. By the first β-reduction, this type equals Path base' base', so loop' has the same type. Thus, the computation rules for higher inductives follow the same general pattern as for ordinary inductive types, except we need to use the notion of application appropriate for the level of the generator.

B. Dependent Elimination

To fully characterize an inductive type, we need not just recursion, but an induction principle (dependent elimination). The induction principle for natural numbers says that to prove a property for all natural numbers, it suffices to prove that it holds for zero and is preserved by successors. Similarly, the induction rule for S^1 says that to prove a property for all points on the circle, it suffices to give a proof base' that it holds for base, and to show that this proof base' is "preserved by going around the loop":

$$\begin{aligned} S^1\text{-induction} : \ & (X : S^1 \to \mathsf{Type}) \\ & (\mathsf{base'} : X\ \mathsf{base}) \\ & (\mathsf{loop'} : \mathsf{Path}\ (\mathsf{transport}\ X\ \mathsf{loop\ base'})\ \mathsf{base'}) \\ & \to (y : S^1) \to X\ y \end{aligned}$$

Here, X is a dependent type/fibration over the circle. The image of base, which we call base', shows that X holds for base—it is a point in the fiber over base. The image of loop, loop', shows that base' is preserved by going around the loop, which formally means that loop' is a path from base' to itself that projects down to loop, in the sense described in Section II-C.

The topological perspective on why this elimination rule is correct is that base' and loop' determine a dependent function $(y : S^1) \to X\ y$, which represents a section of the fibration $\Sigma\ X \to S^1$ (see Section II-C). Such a section must take each point or path in S^1 to a point or path lying above it, so it is natural to expect a path lying above loop as input. The syntactic perspective is that the second computation rule for S^1-induction must be well-typed:

$$\begin{aligned} (S^1\text{-induction base' loop'})\ \mathsf{base} &= \mathsf{base'} \\ \mathsf{apd}\ (S^1\text{-induction base' loop'})\ \mathsf{loop} &= \mathsf{loop'} \end{aligned}$$

The dependent elimination rule also characterizes an inductive type up to equivalence. S^1-induction is equivalent to asserting that for all X, the type of functions $S^1 \to X$ is naturally equivalent to the type of pairs $(\mathsf{base'} : X, \mathsf{loop'} : \mathsf{Path\ base'\ base'})$ (the premises of S^1-recursion). Because the type theory ensures that functions are groupoid homomorphisms, this is exactly the universal property of the free ∞-groupoid generated by one object and one loop on it.

C. Agda Implementation

Computer proof assistants such as Agda and Coq include ordinary inductive types, but not yet higher inductive types. One way to implement the latter is to simply postulate the generators, the elimination rule, and the computation rules. With this representation, the computation rules are paths (propositional equalities), rather than definitional equalities.

Though the question of whether these rules should be definitional equalities or paths has not yet been settled, it is more convenient to do proofs if they are definitional equalities. While it is not possible to achieve this in Agda, there is a trick using private data types that allows the base (but not loop) rule to be definitional [8]. Because this simplifies the proofs, we use this implementation, with a postulate

$$\begin{aligned} \beta\mathsf{loop/rec} : \ & \{X : \mathsf{Type}\} \\ & (\mathsf{base'} : X)\ (\mathsf{loop'} : \mathsf{Path\ base'\ base'}) \\ & \to \mathsf{Path}\ (\mathsf{ap}\ (S^1\text{-recursion base' loop'})\ \mathsf{loop})\ \mathsf{loop'} \end{aligned}$$

for the β-rule for loop (and similarly for S^1-induction).

V. The Fundamental Group of the Circle

Given a space X with a specified base point x_0, the fundamental group $\pi_1(X, x_0)$ (or just $\pi_1(X)$ when x_0 is clear from context) is the group of homotopy classes of loops from x_0 to itself, with path composition as the group operation. In type theory, this corresponds to the type $\mathsf{Path}\ \{X\}\ x_0\ x_0$, except for one caveat: For $\pi_1(X)$, the group has a *set* of elements, which are paths *quotiented by homotopy*. This means that any two paths that are homotopic are equal, but any non-trivial *structure* of paths between paths has been collapsed by quotienting. On the other hand, the *type* $\mathsf{Path}\ x_0\ x_0$ may have interesting paths between paths (i.e., the type $\mathsf{Path}\ \{\mathsf{Path}\ x_0\ x_0\}\ \alpha\ \beta$ might not be trivial). Thus, $\mathsf{Path}\ x_0\ x_0$ corresponds more closely to what is called the *loop space* $\Omega_1(X, x_0)$, the *space* of loops in X based at x_0, which also may still have non-trivial structure.

It is possible to construct the set $\pi_1(X)$ from the loop space $\Omega_1(X)$ by an operation called *truncation* (in classical topology,

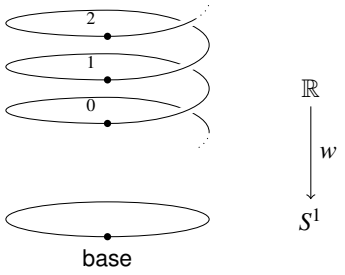
this is just the set of connected components). However, for the example we consider here, this is not necessary: what we will prove is that the loop space of the circle $\Omega_1(\mathsf{S}^1)$ is $\mathbb{Z}$. Because the truncation of $\mathbb{Z}$ is $\mathbb{Z}$, applying truncation to both sides shows that $\pi_1(S^1)$ is also $\mathbb{Z}$. Moreover, proving that $\Omega_1(\mathsf{S}^1)$ is $\mathbb{Z}$ immediately characterizes all the *higher homotopy groups* of the circle, because the higher homotopy groups are determined by iterating the loop space construction. Thus, if $\Omega_1(\mathsf{S}^1)$ is $\mathbb{Z}$, then the higher homotopy groups of S^1 must be the same as the higher homotopy groups of $\mathbb{Z}$, which are trivial.

In type theoretic terms, this means that our first goal is to prove that the type $\mathsf{Path}\,\{\mathsf{S}^1\}\,\mathsf{base}\,\mathsf{base}$ is equivalent to Int. We then check that this equivalence is a group homomorphism, taking path composition to addition.

A. Classical and Type-theoretic Proofs

Our proof that $\Omega_1(\mathsf{S}^1)$ is $\mathbb{Z}$ can be seen as a type-theoretic version of a proof in classical homotopy theory. The classical proof we start from is usually formulated using "universal covering spaces", but we will give an equivalent sketch using *fibrations* which transfers more directly to type theory. Recall the notion of a fibration from Section II-B. For any point $x_0 \in B$, there is a canonical *path fibration* $p : P_{x_0}B \to B$, where the points of $P_{x_0}B$ are paths in B starting at x_0, and the map p selects the other endpoint of such a path. The space $P_{x_0}B$ is *contractible*, i.e. homotopy equivalent to a point, since we can "retract" any path to its initial endpoint x_0. Moreover, the fiber over x_0 is the loop space $\Omega_1(B, x_0)$.

Now consider the "winding" map $w : \mathbb{R} \to S^1$, which looks like a helix projecting down onto the circle:



The map w sends each point on the helix to the point on the circle that it is "sitting above"; this map is a fibration, and the fiber over each point is isomorphic to the integers. If we lift the path that goes counterclockwise around the loop on the bottom, we go up one level in the helix, incrementing the integer in the fiber. Similarly, going clockwise around the loop on the bottom corresponds to going down one level in the helix, decrementing this count. This fibration is called the *universal cover* of the circle.

Now a basic fact is that a map $E_1 \to E_2$ of fibrations over B which is a homotopy equivalence between E_1 and E_2 induces a homotopy equivalence on fibers. Since $\mathbb{R}$ and $P_{\mathsf{base}}S^1$ are both contractible, they are homotopy equivalent, and thus the fibers over base, $\mathbb{Z}$ and $\Omega_1(S^1)$, are isomorphic.

It is possible to formalize this classical proof directly in type theory, by (1) defining the universal cover, (2) proving that a homotopy equivalence between total spaces induces an equivalence on fibers, and (3) proving that the total spaces of both the path fibration and the cover are contractible (this was the first proof of this result in homotopy type theory [19]). However, it turns out to be simpler to explicitly construct the encoding-decoding equivalence, following the template introduced in Section III [9]. Both proofs use the same construction of the cover (step 1 above). Where the classical proof induces an equivalence on fibers from an equivalence between total spaces (step 2), the type-theoretic proof constructs the inverse map explicitly as a map between fibers. Where the classical proof uses contractibility (step 3), the type-theoretic proof uses path induction, circle induction, and integer induction. These are the same tools used to prove contractibility—indeed, path induction *is* contractibility of the path fibration composed with $\mathsf{transport}$—but it is more convenient to use them to prove inverses directly. This proof is a good example of how combining insights from homotopy theory and type theory can simplify proofs and yield deeper insight.

B. The Universal Cover of the Circle

Recall that fibrations are represented by dependent types (Section II-B). The path fibration $P_{\mathsf{base}}\mathsf{S}^1 \to \mathsf{S}^1$ is easy to represent: it is the dependent type sending $\mathsf{x} : \mathsf{S}^1$ to $\mathsf{Path}\,\mathsf{base}\,\mathsf{x}$. The universal cover of the circle (the helix) requires a bit more thought: since our "circle" is not a topological one, we don't have a "real line" that we can wrap around it. However, our inductive definition of the circle gives us a different way to define dependent types over it: by circle-recursion.

```
Cover : S¹ → Type
Cover x = S¹-recursion Int (univalence succEquiv) x
```

To define a function by circle recursion, we need to find a point and a loop in the target. In this case, the target is Type, and the point we choose is Int, corresponding to our expectation that the fiber of the universal cover should be the integers. The loop we choose is the successor/predecessor isomorphism on Int, $\mathsf{succEquiv}$ (Section II), which by univalence determines a path from Int to Int. Univalence is necessary for this part of the proof, because we need a *non-trivial* loop from Int to Int.

It is immediate from this definition that $\mathsf{Cover}\,\mathsf{base}$ is Int, and we can verify that choosing $\mathsf{succEquiv}$ as the image of loop gives the desired path-lifting action: transporting one way along the cover is successor (going up one level in the helix), and the other way is predecessor (going down one level):

```
transport-Cover-loop : Path (transport Cover loop) succ
transport-Cover-loop =
  transport Cover loop
    ≃⟨ transport-ap-assoc Cover loop ⟩
  transport (λ x → x) (ap Cover loop)
    ≃⟨ ap (transport (λ x → x))
          (βloop/rec Int (univalence succEquiv)) ⟩
  transport (λ x → x) (univalence succEquiv)
    ≃⟨ transport-univ _ ⟩
  succ ■
```

```
transport-Cover-!loop : Path (transport Cover (! loop)) pred
```

For transport-Cover-loop, we re-associate, which creates a β-redex ap Cover loop for S^1-recursion. After reducing this, we have a term of of the form transport (λ x → x) (univalence e), which is a β-redex for univalence, and selects the "forward" direction of the equivalence. We omit the proof for transport-Cover-!loop, which is similar except for additional reasoning about inverses, using !-univalence from Section II.

The cover can be seen as a type of codes for paths on the circle, analogous to the family Code in Section III. The next step is to define "encoding" and "decoding" functions and prove that they are an equivalence between Path base x and Cover x for all x : S^1. This is a generalization of the original statement we intended to prove, which was that Path base base is equivalent to Cover base (the latter being, by definition, Int).

C. Encoding

As in Section III, encode is defined by transporting in the cover. The starting point needs to be an element of Cover base, which is Int. In this case, a good choice is Zero[4]:

```
encode : {x : S¹} → Path base x → Cover x
encode α = transport Cover α Zero
```

The instance encode' = encode {base} has type Path base base → Int, as we originally intended.

The interesting thing about this function is that it computes a concrete number from a loop on the circle, when this loop is represented using the abstract groupoidal framework of homotopy type theory. To gain an intuition for how it does this, observe that by the above lemmas, transport Cover loop is succ and transport Cover (! loop) is pred. Further, transport is functorial (Section II), so transport Cover (loop ∘ loop) is (transport Cover loop) o (transport Cover loop), etc. Thus, when α is a composition like loop ∘ ! loop ∘ loop ∘ ..., the application transport Cover α will compute a composition of functions like succ o pred o succ o .... Applying this composition of functions to Zero will compute the winding number of the path—how many times it goes around the circle, with orientation marked by sign, after inverses have been canceled.

Thus, the computational content of encode follows from the β-like rules for higher-inductive types and univalence, and the action of transport on compositions and inverses. This "computation" happens only up to paths in current homotopy type theory, so we cannot actually run this program in Agda, but an alternate formulation might take these equations as definitional equalities [10].

D. Decoding

The first step in decoding is that, given an integer n, we compute the n-fold composition loop^n:

```
loop^ : Int → Path base base
loop^ Zero        = id
loop^ (Pos One)   = loop
loop^ (Pos (S n)) = loop ∘ loop^ (Pos n)
loop^ (Neg One)   = ! loop
loop^ (Neg (S n)) = ! loop ∘ loop^ (Neg n)
```

[4]We could choose another number as the base case besides Zero, but then we would need to subtract it off when decoding.

At this point, if we were working naively, rather than with Section III or the classical proof in mind, we might think that this is enough. That is, since what we want overall is an equivalence between Path base base and Int, we might expect to be able to prove that encode' : Path base base → Int and loop^ : Int → Path base base give an equivalence. The problem comes in trying to prove the "decode after encode" direction:

```
decode-encode : {α : Path base base}
                → Path (loop^ (encode' α)) α
```

In Section III, we proved this step using path induction to reduce α to the identity, which depends crucially on α having one endpoint free—recall that path induction does not apply to loops like a Path base base with both endpoints fixed! The way to solve this problem is to state decode-encode generally for all x:S^1 and α : Path base x:

```
decode-encode : {x:S¹} {α : Path base x}
                → Path (loop^ (encode {x} α)) α
```

However, this does not type check as is, because loop^ works only for Path base base, whereas here we need Path base x. This (along with the template from Section III and the proof from classical homotopy theory) suggests extending loop^ to a function with a more general type:

```
decode : {x : S¹} → Cover x → Path base x
```

Here is the definition of decode:

```
decode : {x : S¹} → Cover x → Path base x
decode {x} =
S¹-induction
  (λ x' → Cover x' → Path base x')
  loop^
  (  transport (λ x' → Cover x' → Path base x') loop loop^ (1)
  ≃ transport (λ x' → Path base x') loop
      o loop^
      o transport Cover (! loop)                            (2)
  ≃ (λ p → loop ∘ p) o loop^ o transport Cover (! loop)     (3)
  ≃ (λ p → loop ∘ p) o loop^ o pred                         (4)
  ≃ (λ n → loop ∘ (loop^ (pred n)))                         (5)
  ≃ (λ n → loop^ n)                                         (6)
  ■)
  x
```

decode's first argument is an arbitrary point on the circle. Thus, we proceed by circle induction, which requires, first, a function Cover base → Path base base, which is just loop^, and, second, a path showing that this function is preserved by going around the loop. Formally, this means a path from transport (λ x' → Cover x' → Path base x') loop loop^ to loop^.

Above, we have shown the steps of reasoning required to give such a path, eliding the proof terms, which we now discuss informally. The path is constructed by composing five steps of reasoning, between terms (1) through (6) above. From (1) to (2), we apply the definition of transport when the

outer connective of the type family is $\to$, using the lemma transport-$\to$ from Section II. This reduces the transport to pre- and post-composition with transport at the domain and range types. From (2) to (3), we apply the definition of transport when the type family is Path base - (called transport-Path-right above). From (3) to (4), we apply transport-Cover-!loop. From (4) to (5), we simply reduce the function composition. The final step is the only significant one: it follows from associativity and inverses of $\circ$, together with a lemma loop^-preserves-pred which gives a Path (loop^ (pred n)) (! loop $\circ$ loop^ n) for all n. This lemma is proved by a simple case analysis, again using associativity/unit/inverse laws.

E. Encoding after Decoding

Computing encode o loop^ is comparatively straightforward.

```
encode-loop^ : (n : Int) → Path (encode (loop^ n)) n
encode-loop^ Zero = id
encode-loop^ (Pos One) = ap≃ transport-Cover-loop
encode-loop^ (Pos (S n)) =
  encode (loop^ (Pos (S n)))
    ≃⟨ id ⟩
  transport Cover (loop ∘ loop^ (Pos n)) Zero
    ≃⟨ ap≃ (transport-∘ Cover loop (loop^ (Pos n))) ⟩
  transport Cover loop
            (transport Cover (loop^ (Pos n)) Zero)
    ≃⟨ ap≃ transport-Cover-loop ⟩
  succ (transport Cover (loop^ (Pos n)) Zero)
    ≃⟨ id ⟩
  succ (encode (loop^ (Pos n)))
    ≃⟨ ap succ (encode-loop^ (Pos n)) ⟩
  succ (Pos n) ■
```

The proof is a simple induction on Int, using functoriality of transport and the transport-Cover-loop lemmas. We omit the cases for Neg, which are analogous.

To prove the equivalence of Path base base and Int, this is sufficient. If we additionally want an equivalence between Path base x and Cover x for all x, then we need to show that encode-loop^ extends to

```
encode-decode : {x : S¹} → (c : Cover x)
                → Path (encode (decode {x} c)) c
```

which we do in the companion code.

F. Decoding after Encoding

As in Section III, decoding after encoding is easy:

```
decode-encode : {x : S¹} (α : Path base x)
                → Path (decode (encode α)) α
decode-encode {x} α =
path-induction (λ (x' : S¹) (α' : Path base x')
                  → Path (decode (encode α')) α')
               id α
```

By path induction, it suffices to consider the case when α is id : Path base base. Then encode {base} id = Zero, and decode {base} Zero = loop^ Zero = id, so we need a Path id id—which can be id.

decode-encode can be seen as an η-rule/induction principle for paths on the circle, which states that every loop on the circle is of the form loop^ n for some n:

```
all-loops : (α : Path base base) → Path α (loop^ (encode α))
all-loops α = ! (decode-encode α)
```

Consequently, to prove a statement for every Path base base, it suffices to prove the statement for loop^ n for every n, which can be done using integer induction. Recall that the proof of decode-encode depends crucially on the fact that decode is defined for a path with a free endpoint. Thus, the essential parts of this proof are the path induction used here, and the circle induction used to define decode from loop^. It is important that this combination of circle and path induction suffices to *prove* this induction principle for loops on the circle, as we discuss further in Section VI.

G. Summary

These lemmas give a homotopy equivalence between Path base base and Int, establishing that the loop space of the circle is equivalent to Int:

```
Ω₁ [S¹] -is-Int : HEquiv (Path base base) Int
Ω₁ [S¹] -is-Int =
  hequiv encode decode decode-encode encode-loop^
```

To identify the fundamental group of S^1 with Int as a *group*, we also must check that this equivalence is a group homomorphism. A bijection between carriers is a group homomorphism if one of the functions preserves composition (it then necessarily preserves inverses and the unit because these are unique). Thus, it suffices to show that

```
preserves-composition : (n m : Int)
  → Path (loop^ (n + m)) (loop^ n ∘ loop^ m)
```

The proof is an easy induction, using associativity and unit of $\circ$, the lemma loop^-preserves-pred defined above, and an analogous lemma that loop^ (succ n) is loop $\circ$ loop^ n.

Categorically, we can understand the proof we have just given as follows: The higher-inductive type S^1 is a *presentation* of the free ∞-groupoid with one morphism, using the groupoidal framework of type theory. The proof we have given shows that type theory is sufficiently powerful to relate this abstract description to an *explicit* description of the free group on one generator, as the inductive type Int equipped with the function +.

VI. Conclusion

In this paper, we have described a method for characterizing the path spaces of inductive types in type theory, and applied it to two examples. For coproducts, we obtain injectivity and disjointness of constructors. For the circle, we compute its fundamental group, a basic theorem of algebraic topology. The proof for the circle illustrates the use of homotopy type theory as a logic of homotopy theory, and introduces the encode-decode method, which we have subsequently applied to more examples, as discussed in Section I.

Seeing injectivity-and-disjointness of inductive types in this context provides a topological explanation for the use of a universe to prove them: Injectivity and disjointness characterize a path space. Topological proofs characterizing a

path space typically consider an entire path fibration at once (like $\mathsf{Path}\,(\mathsf{Inl}\,a)\,-$), rather than a path with both endpoints fixed (like $\mathsf{Path}\,(\mathsf{Inl}\,a)\,(\mathsf{Inl}\,a')$), and show that the entire path fibration is equivalent to an alternate fibration (our "codes"). The codes fibration (like the universal cover of the circle) is represented in type theory using induction, which requires a universe or large elimination.

Moreover, the fact that a universe is *necessary* has analogues in higher dimensions: it is the first rung on a ladder of categorical nondegeneracy. Without a universe, the category of types could be a poset, in which case disjointness at least would fail. Without a *univalent* universe, the ∞-category of types could be a 1-category, in which case the computation of the fundamental group of the circle would fail. In general, path spaces of inductive types are only "correct" when the category of types is sufficiently rich to support them.

In this paper, we have taken an approach to inductive types where the characterization of the path space is a *theorem*, not part of the *definition* (as in [1]). One might wonder whether we could take the opposite approach: For coproducts, we might include injectivity and disjointness of Inl and Inr in the definition; for the circle, we might include an elimination rule for paths $\mathsf{Path}\{S^1\}\,x\,y$ expressing that they are freely generated by loop. However, there are two problems with this. Conceptually, a (higher) inductive type is *one* freely generated structure, even though it may have more than one kind of generator. As such, it should have only one elimination rule, expressing its universal property. More practically, calculating homotopy groups of a space in algebraic topology can be a significant mathematical theorem. For example, for the 2-dimensional sphere, π_1 is trivial, π_2 is $\mathbb{Z}$ (like the circle, one level up), but π_3 is also $\mathbb{Z}$, even though the description of the sphere does not include any generators at this level. This is due to something called the Hopf fibration, which arises from the interaction of the lower-dimensional generators with the ∞-groupoid laws. Indeed, there is no known formula for the homotopy groups of higher-dimensional spheres, so we would not know what characterization to include in the definition, even if we wanted to.

Fortunately, the examples we have done so far suggest that the paths in inductive types will always be *determined* by the inductive description and ambient ∞-groupoid laws—so characterizing the path spaces explicitly in the definition would be at best redundant, and at worst inconsistent. Thus, we can pose these questions about homotopy groups using higher inductive types, and use homotopy type theory to investigate them.

Acknowledgments We thank the following people for helpful feedback on this work: Steve Awodey, Robert Harper, Martin Hofmann, Chris Kapulkin, Peter Lumsdaine, many other people in Carnegie Mellon's HoTT group and the Institute for Advanced Study special year, and the anonymous reviewers.

References

[1] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *Programming Languages meets Program Verification Workshop*, 2007.

[2] S. Awodey and M. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 2009.

[3] Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.2*. INRIA, 2009. Available from http://coq.inria.fr/.

[4] N. Gambino and R. Garner. The identity type weak factorisation system. *Theoretical Computer Science*, 409(3):94–109, 2008.

[5] R. Garner. Two-dimensional models of type theory. *Mathematical. Structures in Computer Science*, 19(4):687–736, 2009.

[6] M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory*. Oxford University Press, 1998.

[7] C. Kapulkin, P. L. Lumsdaine, and V. Voevodsky. The simplicial model of univalent foundations. arXiv:1211.2851, 2012.

[8] D. R. Licata. Running circles around (in) your proof assistant; or, quotients that compute. http://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/, April 2011.

[9] D. R. Licata. A simpler proof that $\pi_1(S^1)$ is $\mathbb{Z}$. http://homotopytypetheory.org/2012/06/07/a-simpler-proof-that-$\pi_1 s^1$-is-z/, June 2012.

[10] D. R. Licata and R. Harper. Canonicity for 2-dimensional type theory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[11] P. L. Lumsdaine. Weak ω-categories from intensional type theory. In *International Conference on Typed Lambda Calculi and Applications*, 2009.

[12] P. L. Lumsdaine. Higher inductive types: a tour of the menagerie. http://homotopytypetheory.org/2011/04/24/higher-inductive-types-a-tour-of-the-menagerie/, April 2011.

[13] P. L. Lumsdaine and M. Shulman. Higher inductive types. In preparation, 2013.

[14] C. McBride, H. Goguen, and J. Mckinna. A few constructions on constructors. In *Types for Proofs and Programs (TYPES'04)*, page 30. Springer-Verlag, 2005.

[15] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[16] U. Norell. Dependently typed programming in agda. *Summer school on Advanced Functional Programming*, 2008.

[17] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris 7, 1989.

[18] M. Shulman. Homotopy type theory VI: higher inductive types. http://golem.ph.utexas.edu/category/2011/04/homotopy_type_theory_vi.html, April 2011.

[19] M. Shulman. A formal proof that $\pi_1(S^1) = \mathbb{Z}$. http://homotopytypetheory.org/2011/04/29/a-formal-proof-that-pi1s1-is-z/, April 2011.

[20] M. Shulman. Univalence for inverse diagrams, oplax limits, and gluing, and homotopy canonicity. arXiv:1203.3253, 2013.

[21] The Univalent Foundations Program, Institute for Advanced Study. *Homotopy Type Theory: Univalent Foundations Of Mathematics*. Available from homotopytypetheory.org/book, 2013.

[22] B. van den Berg and R. Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011.

[23] V. Voevodsky. Univalent foundations of mathematics. Invited talk at WoLLIC 2011 18th Workshop on Logic, Language, Information and Computation, 2011.

[24] M. A. Warren. *Homotopy theoretic aspects of constructive type theory*. PhD thesis, Carnegie Mellon University, 2008.