Formalizing Computational Paths and Fundamental Groups in Lean

Arthur F. Ramos[*] Anjolina G. de Oliveira[†] Ruy J. G. B. de Queiroz[‡]
Tiago Mendonça Lucena de Veras[§]

November 23, 2025

Abstract

Computational paths treat propositional equality as explicit paths built from labelled deduction steps and rewrite rules. This view originates in work by de Queiroz and collaborators [dQdOR16] and yields a weak groupoid structure for equality, together with a computational account of homotopy inspired by homotopy type theory. In this paper we present a complete mechanization of this framework in Lean 4 and show how it supports concrete homotopy theoretic computations.

Our contributions are threefold. First, we formalize the theory of computational paths in Lean, including path formation, composition, inverses, and a rewrite system that identifies redundant or trivial paths. We prove that equality types with computational paths carry a weak groupoid structure in the sense of the original theory.

Second, we organize this material into a reusable Lean library, `ComputationalPathsLean`, which exposes an interface for paths, rewrites, and loop spaces. This library allows later developments to treat computational paths as a drop in replacement for propositional equality when reasoning about homotopical structure.

Third, we apply the library to two canonical examples in algebraic topology. We give Lean proofs that the fundamental group of the circle is isomorphic to the integers and that the fundamental group of the torus is isomorphic to the product of two copies of the integers, both via computational paths. These case studies demonstrate that the computational paths approach scales to nontrivial homotopical computations in a modern proof assistant.

All the definitions and proofs described here are available in an open source Lean 4 repository [RdVdQdO25].

1 Introduction

Homotopy type theory and higher category theory have reshaped our understanding of equality in type theory [Awo12, Uni13]. In this perspective, an equality between elements of a type is not a mere proposition but carries a structure that behaves like a path in a space. This insight links type theory to homotopy theory and suggests that proof assistants based on type theory can serve as practical tools for mechanized homotopy theory.

Computational paths offer a complementary viewpoint [dQdOR16]. Instead of treating equalities abstractly, they describe equalities as explicitly labelled paths generated by inference rules in a natural deduction system. These paths can be composed, inverted, and simplified by a rewrite system. The resulting structure forms a weak groupoid of equalities, where associativity and inverse laws hold up to path rewrites.

[*]Microsoft, Florida, USA. Email: `arfreita@microsoft.com`.

[†]Centro de Informática, Universidade Federal de Pernambuco, Recife, Brasil. Email: `ago@cin.ufpe.br`.

[‡]Centro de Informática, Universidade Federal de Pernambuco, Recife, Brasil. Email: `ruy@cin.ufpe.br`.

[§]Departamento de Matemática, Universidade Federal Rural de Pernambuco, Recife, Brasil. Email: `tiago.veras@ufrpe.br`.

The theory of computational paths has been developed in a series of papers that describe the underlying logical system, the associated rewrite rules, and applications to homotopy theoretic constructions such as fundamental groups [dQdOR16, dVRdQdO19, Ram18]. However, until now, this theory has lived mostly on paper. Its mechanization in a modern proof assistant brings several benefits. It increases confidence in the correctness of the constructions, exposes hidden subtleties in the interplay of rewrite rules, and provides a reusable platform for further developments.

In this paper we report on the first full formalization of computational paths in Lean 4. We build a Lean library that captures the main ideas of the original theory and we use it to compute nontrivial fundamental groups. The work reported here combines logical, algebraic, and engineering aspects and is distributed as the `ComputationalPathsLean` repository [RdVdQdO25].

Contributions More concretely, the main contributions of this work are:

- A Lean formalization of computational paths for propositional equality, including:
  - a type of paths between elements of a type,
  - operations for reflexivity, symmetry, and transitivity,
  - congruence rules for functions and dependent functions,
  - a rewrite system that removes trivial or redundant path structure.
- A proof, mechanized in Lean, that equality types equipped with computational paths form a weak groupoid. In particular, we formalize path composition and inverses and show that the groupoid laws hold up to rewrite equality.
- A Lean implementation of loop spaces and fundamental groups based on computational paths.
- Two full case studies:
  - a computation of the fundamental group of the circle and an isomorphism with the integers,
  - a computation of the fundamental group of the torus and an isomorphism with the product of two copies of the integers.
- A reusable Lean library, structured as a hierarchy of modules, which can be extended with further spaces and homotopy theoretic constructions.

To our knowledge this is the first mechanization of the computational paths framework in any major proof assistant and the first demonstration that this framework can support the computation of classical fundamental groups inside a fully checked environment.

Structure of the paper Section 2 recalls computational paths and their weak groupoid structure, as well as the basic notions of loop spaces and fundamental groups. Section 3 describes our Lean environment and overall library layout. Section 4 gives the core definitions of paths and rewrites. Section 5 presents the formalization of the weak groupoid structure. Section 6 describes the computation of the fundamental group of the circle. Section 7 does the same for the torus. Section 8 discusses engineering aspects of the development, including library design and automation. Section 9 situates our work in the landscape of homotopy inspired formalizations. Section 10 concludes and sketches directions for future work.

2 Background

2.1 Computational paths

In the traditional formulation of intensional Martin Löf type theory [ML84], equality between elements of a type is given by the identity type. A proof of equality can be seen as a path connecting its endpoints. Homotopy type theory makes this interpretation explicit and develops a rich story about higher path structure [Uni13].

Computational paths start from a different point of view [dQdOR16, Ram18]. Instead of treating equality as a primitive type with constructors and eliminators, one derives equalities from a natural deduction system for a typed lambda calculus. Each equality step is labelled by the corresponding reduction or expansion rule, such as beta, eta, or reflexivity. By composing these labelled steps one obtains paths.

Formally, given a type A and elements $a, b : A$, a computational path from a to b is a finite sequence of rewrite steps that transforms a term denoting a into a term denoting b. The syntax of paths records the labels of these rewrite steps. There are basic constructors for reflexive paths, for the inverse of a path, and for the concatenation of paths.

The concrete syntax used in [dQdOR16] is based on the labelled natural deduction system LNDEQ. Path terms are built from labels for the equality rules, together with composition operators that keep track of the shape of the derivation. This representation is closer to actual proof terms than to the abstract identity type of Martin Löf type theory, and it allows the rewrite system on paths to mirror the proof theoretic properties of equality reasoning.

However, paths contain redundant structure. For example, the identity path composed with another path should be considered equal to the second path, and a path composed with its inverse should be trivial. The theory therefore introduces a rewrite system on paths that contracts such redundancies. The quotient of paths by this rewrite system carries the structure of a weak groupoid.

From the point of view of term rewriting [BN98, KB70], the computational paths system consists of a first layer of rewrites on object terms (beta, eta, and other reductions of the underlying calculus) and a second layer of rewrites on equality proofs. The second layer is where the weak groupoid structure is encoded.

2.2 Weak groupoids of equality

A groupoid is a category where every morphism is invertible [HS94]. In the context of computational paths, the objects of the groupoid are terms and the morphisms are equivalence classes of paths under the rewrite relation.

The groupoid is weak because the associativity of composition and the laws relating identities and inverses do not hold strictly, but only up to the rewrite relation [vdBG11, Lum09]. For example, the concatenation of three paths $(p \cdot q) \cdot r$ and $p \cdot (q \cdot r)$ are not syntactically equal but are related by a sequence of path rewrites. In categorical language, the associator and the unitors are given by higher paths that witness the equality between composites.

The original theory of computational paths develops this structure and shows that it recovers many of the intuitions of homotopy type theory in a more concrete, proof theoretic setting [dQdOR16, dVRdQdO19]. For instance, the groupoid laws correspond to canonical LNDEQ derivations that eliminate detours in equality proofs. Weakness appears when only the rewrite relation on equality proofs identifies different derivations.

One advantage of the computational paths approach is that it exposes the combinatorics of equality proofs in a way that lends itself to algorithmic manipulation. The rewrite relation is finitary and admits a careful critical pair analysis [New42], so one can investigate confluence and termination properties. Our Lean formalization instantiates these abstract properties in a concrete proof assistant setting.

2.3 Loop spaces and fundamental groups

Given a type A and a point $a : A$, the loop space at a consists of paths from a to itself. In homotopy type theory this is the identity type $a =_A a$; in the computational paths framework it is the type of computational paths from a to a, modulo rewrites [Uni13].

The set of connected components of the loop space, with composition induced by path concatenation, forms the fundamental group of A at a. For spaces like the circle and torus these groups are well known: the fundamental group of the circle is isomorphic to the integers and the fundamental group of the torus is isomorphic to the direct product of two copies of the integers [LS13, dVRdQdO19].

In homotopy type theory, the classical computation of the fundamental group of the circle uses an encode and decode construction. One defines a map that sends loops to integers and another map that sends integers to loops, and proves that they are inverse equivalences in the homotopy sense. We follow the same high level structure, but we phrase all constructions in terms of computational paths and their rewrites, and we mechanize the entire argument in Lean.

Our goal is to realize these constructions concretely in Lean using computational paths, and to do so in a way that is faithful to the original LNDEQ based presentations.

3 Lean environment and library layout

3.1 Lean and mathlib setting

We use Lean 4 as our proof assistant [dMKA$^+$15]. Lean provides a dependent type theory with inductive types, type classes, and an extensible elaboration and tactic system. The core language is expressive enough to encode the LNDEQ style path syntax directly, and its universe polymorphism is sufficient for the kinds of small groupoid constructions that we consider.

Our development depends on Lean's core libraries and on standard components of the mathlib ecosystem [BCM20]. We rely on mathlib for basic algebraic structures such as groups and group homomorphisms, the integer type and its algebraic properties, and some standard infrastructure for quotients and setoids. We deliberately avoid deeper parts of mathlib that implement homotopy type theory specific features, since one of our aims is to show that computational paths can be realized in an otherwise ordinary dependent type theory.

3.2 Repository structure

The code is organized as a standalone library called `ComputationalPathsLean`[1]. Within this repository the main directory `ComputationalPaths` contains the definitions and proofs described in this paper. At a high level the directory is split into the following components:

- `Path.Basic`: core definitions of the type of computational paths and basic operations.
- `Path.Rewrite.Rw`: the rewrite relation on paths, together with its basic properties.
- `Path.Rewrite.Quot`: the quotient of paths by the rewrite relation and the induced groupoid structure.
- `LoopSpace`: definitions and lemmas about loop spaces and fundamental groups.
- `Examples.Circle` and `Examples.Torus`: concrete case studies for the circle and torus.

In practice there are several auxiliary modules that support these main components, for instance modules containing helper lemmas about integers, about repeated composition of loops, and about normal forms in the torus case study. We keep these auxiliary modules separate to avoid cyclic dependencies and to make it easier for users to import only what they need.

The code follows Lean's module system so that users can import only the parts they need. For example, a development that only needs paths and rewrites can import `Path.Basic` and `Path.Rewrite.Rw`, while a development interested in fundamental groups can additionally import the loop space modules.

[1]See [RdVdQdO25].

3.3 Design choices

There are several design choices in the Lean formalization that are not forced by the mathematics but have significant impact on usability.

First, we chose an inductive representation of paths where the endpoints are indices of the type. This keeps type checking simple and allows Lean's pattern matching to recover endpoints from constructors. An alternative would be to use a separate object language and a typing relation, but this would make basic manipulations more cumbersome.

Second, we split the rewrite system into small, focused constructors, each corresponding to a single conceptual equality such as eliminating a left identity or cancelling a path with its inverse. This makes proofs that a given composite reduces to a simpler form more predictable and easier to inspect.

Third, we expose both the raw path syntax and the quotient by rewrites as first class types. In some situations it is convenient to reason about specific representatives, for example when defining a function by recursion on the path constructors. In others it is more convenient to work modulo rewrite equivalence. We use Lean's `Quot` and `Quot.lift` infrastructure to move between these levels.

4 Paths and rewrites in Lean

4.1 The type of computational paths

The code in `ComputationalPaths/Path/Basic/Core.lean`[2] mirrors the paper definitions but keeps the combinatorial data that witnesses a path. Instead of closing over the built-in identity type, we store the finite list of elementary rewrites that was used to travel from a to b together with the induced equality proof. The basic syntax is:

```
namespace ComputationalPaths

structure Step (A : Type u) where
  src : A
  tgt : A
  proof : src = tgt

structure Path {A : Type u} (a b : A) where
  steps : List (Step A)
  proof : a = b

namespace Path

@[simp] def toEq (p : Path a b) : a = b :=
  p.proof

@[simp] def refl (a : A) : Path a a :=
  Path.mk [] rfl

@[simp] def ofEq (h : a = b) : Path a b :=
  Path.mk [Step.mk a b h] h

@[simp] def trans (p : Path a b) (q : Path b c) : Path a c :=
  Path.mk (p.steps ++ q.steps) (p.proof.trans q.proof)

@[simp] def symm (p : Path a b) : Path b a :=
  Path.mk (p.steps.reverse.map Step.symm) p.proof.symm
```

[2]File paths refer to the public repository [RdVdQdO25].

```
end Path
end ComputationalPaths
```

The extra `steps` field gives us the exact rewrite certificate emphasized in the LNDEQ literature [dQdOR16, dVRdQdO19], while `proof` keeps the bridge to Lean's identity type. All of the usual constructions—reflexivity, inversion, concatenation, and transport—are defined using the record operations above. Because `Path.toEq` is definitionally the stored equality proof, we can freely switch between computational paths and propositional equalities whenever needed.

4.2 Loop spaces

The abstraction layer for loops lives in `ComputationalPaths/Path/Homotopy/Loops.lean`[3]. Raw loop spaces are just paths whose endpoints coincide, while rewrite classes of loops (the objects that model fundamental groups) are quotients by rewrite equivalence:

```
namespace ComputationalPaths
namespace Path

abbrev LoopSpace (A : Type u) (a : A) : Type u :=
  Path (A := A) a a

abbrev LoopQuot (A : Type u) (a : A) : Type u :=
  PathRwQuot A a a

namespace LoopQuot

@[simp] def ofLoop (p : LoopSpace A a) : LoopQuot A a :=
  Quot.mk _ p

@[simp] def id : LoopQuot A a :=
  PathRwQuot.refl (A := A) a

@[simp] def comp (x y : LoopQuot A a) : LoopQuot A a :=
  PathRwQuot.trans (A := A) x y

@[simp] def inv (x : LoopQuot A a) : LoopQuot A a :=
  PathRwQuot.symm (A := A) x

@[simp] theorem comp_assoc (x y z : LoopQuot A a) :
    comp (comp x y) z = comp x (comp y z) :=
  PathRwQuot.trans_assoc _ _ _

@[simp] theorem inv_comp (x : LoopQuot A a) :
    comp (inv x) x = id :=
  PathRwQuot.symm_trans _ _

end LoopQuot
end Path
end ComputationalPaths
```

The quotient version exposes strictly associative composition, a genuine identity, and strict inverses, so downstream developments can reason as if they were working in a classical fundamental group. The normalisation API connecting raw loops to `LoopQuot` objects handles the bookkeeping required to lift functions and proofs through the quotient.

[3]All Lean source files live in the public repository [RdVdQdO25].

4.3 The rewrite system

Simplifying computational paths follows the two-layer presentation of LNDEQ: first we enumerate single-step rewrites (cancellation, associativity witnesses, β/η reductions, congruence for contexts, *etc.*), and then we close this relation under reflexive/transitive (and later symmetric) closure. The primitive steps live in `ComputationalPaths/Path/Rewrite/Step.lean`[4] and are represented by an inductive family:

```
namespace ComputationalPaths
namespace Path

/-- Primitive rewrite steps on computational paths. -/
inductive Step :
  {A : Type u} -> {a b : A} ->
    Path a b -> Path a b -> Prop
  | trans_refl_left {p : Path a b} :
      Step (Path.trans (Path.refl a) p) p
  | trans_refl_right {p : Path a b} :
      Step (Path.trans p (Path.refl b)) p
  | symm_trans {p : Path a b} :
      Step (Path.trans (Path.symm p) p) (Path.refl b)
  | symm_trans_congr {p : Path a b} {q : Path b c} :
      Step (Path.symm (Path.trans p q))
           (Path.trans (Path.symm q) (Path.symm p))
  | prod_fst_beta {p : Path a1 a2} {q : Path b1 b2} :
      Step (Path.congrArg Prod.fst (Path.prodMk p q)) p
  | -- many more constructors (beta/eta rules, context substitution, ...)

/-- Reflexive/transitive closure of primitive steps. -/
inductive Rw {A : Type u} {a b : A} :
    Path a b -> Path a b -> Prop
  | refl (p : Path a b) : Rw p p
  | tail {p q r : Path a b} :
      Rw p q -> Step q r -> Rw p r

end Path
end ComputationalPaths
```

The long tail of constructors (indicated by the ellipsis) covers every β and η reduction used in the LNDEQ presentation [dQdOR16]: projections and recursors for products, sums, and Σ-types; application and abstraction for Π-types; transport rules; and the context-substitution steps needed for the confluence proof. Because each rule has its own constructor we can quote the precise rewrite when transporting proofs between the raw path level and the quotient level. The reflexive/transitive closure `Rw` plays the role of the rewrite equality relation from that presentation; its symmetric closure `RwEq` and the quotient `PathRwQuot` (Section 5) provide the weak groupoid structure used later in the paper. In practice we rarely reason directly with `Rw`'s constructors—instead we rely on the large collection of derived lemmas and automation living next to the definition.

5 Weak groupoid structure

Quotienting computational paths by rewrite equality provides the weak (in fact strict, after quotienting) groupoid structure promised in the original papers. The entire interface is implemented in `ComputationalPaths/Path/Rewrite/Quot.lean`[5]. We recap the key components below.

[4]See [RdVdQdO25].
[5]See [RdVdQdO25].

5.1 Quotienting by rewrites

Rewrite equality `RwEq` is a symmetric, reflexive, and transitive closure of the single-step relation `Rw`. Lean packages it as a `Setoid` and defines the quotient type `PathRwQuot` $A\ a\ b$ together with the usual operations:

```
namespace ComputationalPaths
namespace Path

/-- Paths modulo rewrite equality. -/
abbrev PathRwQuot (A : Type u) (a b : A) :=
  Quot (rwEqSetoid A a b)

namespace PathRwQuot

@[simp] def refl (a : A) : PathRwQuot A a a :=
  Quot.mk _ (Path.refl a)

@[simp] def symm :
    PathRwQuot A a b -> PathRwQuot A b a :=
  Quot.lift (fun p => Quot.mk _ (Path.symm p))
    (by intro p q h; simpa using rweq_symm h)

@[simp] def trans :
    PathRwQuot A a b -> PathRwQuot A b c -> PathRwQuot A a c :=
  Quot.inductionOn_2
    (fun p q => Quot.mk _ (Path.trans p q))
    (by
      intro p_1 p_2 q_1 q_2 hp hq
      exact Quot.sound (rweq_trans hp (rweq_comp_right hq)))

@[simp] def ofEq (h : a = b) : PathRwQuot A a b :=
  Quot.mk _ (Path.ofEq h)

@[simp] def toEq : PathRwQuot A a b -> a = b :=
  Quot.inductionOn (fun p => p.toEq)

@[simp] theorem toEq_trans (x : PathRwQuot A a b) (y : PathRwQuot A b c) :
    toEq (trans x y) = (toEq x).trans (toEq y) := rfl

end PathRwQuot
end Path
end ComputationalPaths
```

Besides these constructors, the development provides normalisation maps that pick a canonical representative for every class, together with proofs that `PathRwQuot` is equivalent to the ordinary identity type. We nevertheless keep `PathRwQuot` around explicitly because later arguments (loop normal forms, encode/decode equivalences) use the stored rewrite witnesses.

5.2 Groupoid laws up to rewrite

Once rewriting is quotiented away, path composition becomes strictly associative with strict inverses. The file `ComputationalPaths/Path/Groupoid.lean`[6] instantiates mathlib's `Groupoid` structure with objects A and morphisms `PathRwQuot` $A\ a\ b$:

```
def strictGroupoid (A : Type u) : StrictGroupoid A where
  comp := fun p q => PathRwQuot.trans (A := A) p q
```

[6]See [RdVdQdO25].

```
  id := fun {a} => PathRwQuot.refl (A := A) a
  assoc := PathRwQuot.trans_assoc
  id_left := PathRwQuot.trans_refl_left
  id_right := PathRwQuot.trans_refl_right
  inv := fun p => PathRwQuot.symm (A := A) p
  inv_left := PathRwQuot.symm_trans
  inv_right := PathRwQuot.trans_symm
```

For readability we typically work with the `LoopQuot` and `PiOne` aliases introduced in Section 4; these reuse the same definitions but specialise to the case where $a = b$. The strict groupoid interface supplies all of the algebra needed later: cancellation lemmas, iteration via natural and integer powers, and transport of functions through the quotient. In particular, the encode/decode arguments for the circle and torus only have to reason about `PathRwQuot` values; the heavy rewrite machinery remains encapsulated behind the quotient API.

6 Fundamental group of the circle

The first case study is the circle. The Lean file `ComputationalPaths/Path/HIT/Circle.lean`[7] introduces an axiomatic higher-inductive interface so that downstream code can already depend on a stable API:

```
axiom Circle : Type u
axiom circleBase : Circle
axiom circleLoop : Path circleBase circleBase

structure CircleRecData (C : Type v) where
  base : C
  loop : Path base base

axiom circleRec {C : Type v} (data : CircleRecData C) :
    Circle -> C
axiom circleRec_base : circleRec data circleBase = data.base
axiom circleRec_loop :
  Path.trans (Path.symm (Path.ofEq circleRec_base))
    (Path.trans (Path.congrArg (circleRec data) circleLoop)
      (Path.ofEq circleRec_base)) = data.loop
```

Dependent elimination data and axioms (`CircleIndData`, `circleInd`) are provided as well. These recursors give us the usual "code" family into the integers and let us transport loops into an arithmetic setting. The core definitions that drive the winding-number computation are:

```
/-- Universal cover code family landing in the integers. -/
noncomputable def circleCode : Circle -> Type _ :=
  circleRec circleCodeData

/-- Encode a raw loop as an integer using transport in ‘circleCode‘. -/
@[simp] def circleEncodePath :
    Path circleBase circleBase -> Int :=
  fun p => circleCodeToInt (circleEncodeRaw circleBase p)
```

6.1 Loop powers and winding numbers

Loop spaces and their quotients were introduced in Section 4. Specialising those constructions to the circle gives the abbreviations `CircleLoopSpace`, `CircleLoopQuot`, and `circlePiOne`. The

[7]See [RdVdQdO25].

file proceeds by defining natural and integer powers of the fundamental loop, both on raw paths and on their rewrite classes:

```
def circleLoopPathPow : Nat -> Path circleBase circleBase
  | 0 => Path.refl circleBase
  | Nat.succ n => Path.trans (circleLoopPathPow n) circleLoop

def circleLoopPow (n : Nat) : CircleLoopQuot :=
  LoopQuot.pow circleLoopClass n

def circleLoopZPow (z : Int) : CircleLoopQuot :=
  LoopQuot.zpow circleLoopClass z
```

Encoding a loop class is simply the quotient-lift of the raw encoding:

```
/-- Quotient-level winding number. -/
@[simp] def circleEncodeLift : CircleLoopQuot -> Int :=
  Quot.lift (fun p => circleEncodePath p)
    (by intro p q h; exact circleEncodePath_rweq h)
```

The computation rules mirror the informal algebra: composing with the fundamental loop adds 1, composing with its inverse subtracts 1, and the value on n-fold concatenations is n (lemmas `circleEncodeLift_comp_loop`, `circleEncodeLift_comp_inv_loop`, `circleEncodeLift_circleLoopPow`). Integer powers use `circleLoopZPow` and satisfy the expected addition and negation laws. All of these facts are proved directly from the rewrite system by induction on the syntax of loops and on the integer argument.

6.2 Isomorphism with the integers

At this point the encode/decode maps are simple wrappers around the loop quotient infrastructure:

```
@[simp] def circleEncode : CircleLoopQuot -> Int := circleEncodeLift

@[simp] def circleDecodeConcrete : Int -> CircleLoopQuot :=
  circleLoopZPow

@[simp] def circleDecode : Int -> circlePiOne :=
  fun z => PiOne.ofLoop (circleLoopPathZPow z)
```

The addition and subtraction rules for `circleDecodeConcrete` (e.g. `circleDecodeConcrete_add`) show that integer sums correspond to concatenation of iterated loops. Dually, the lemmas `circleEncode_comp_loop` and `circleEncode_comp_inv_loop` witness that conjugating by the generator increments or decrements the winding number.

Two inverse laws finish the job: `circleEncode_circleDecode` proves that $\mathbb{Z} \to \pi_1(S^1)$ followed by encoding is the identity on integers, while `circleDecode_circleEncode` proves the other composite is the identity on the quotient of loops. These statements reduce to integer induction on one side and to normalisation of representative loops on the other. The final equivalence is packaged as:

```
noncomputable def circlePiOneEquivInt :
    SimpleEquiv circlePiOne Int where
  toFun := circleWindingNumber
  invFun := circleDecode
  left_inv := circleDecode_circleEncode
  right_inv := circleEncode_circleDecode
```

The field `circleWindingNumber` is just `circleEncode` seen as a map $\pi_1(S^1) \to \mathbb{Z}$; the equivalence above shows that $\pi_1(S^1)$ is canonically the additive group of integers inside Lean.

All downstream reasoning about the circle—step laws, naturality results, or transport to other homotopy invariants—reuse these lemmas without ever returning to the raw rewrite system.

7 Fundamental group of the torus

The torus development follows the same template as the circle but has two commuting generators. The axiomatic skeleton in `ComputationalPaths/Path/HIT/Torus.lean`[8] provides the data needed for encode/decode arguments:

```
axiom Torus : Type u
axiom torusBase : Torus
axiom torusLoop1 : Path torusBase torusBase
axiom torusLoop2 : Path torusBase torusBase

structure TorusRecData (C : Type v) where
  base : C
  loop1 : Path base base
  loop2 : Path base base

axiom torusRec {C : Type v} (data : TorusRecData C) : Torus -> C
axiom torusRec_loop1 :
  Path.trans (Path.symm (Path.ofEq (torusRec_base data)))
    (Path.trans (Path.congrArg (torusRec data) torusLoop1)
      (Path.ofEq (torusRec_base data))) = data.loop1
axiom torusRec_loop2 : -- analogous computation rule for ‘torusLoop2‘
```

Dependent data (`TorusIndData`) gives elimination principles for families over the torus. Using these recursors we build a universal cover `torusCode : Torus -> Type` whose fibre over the base point is $\mathbb{Z} \times \mathbb{Z}$. Transporting along the two fundamental loops increments the first or second coordinate respectively, while transporting along their inverses decrements. The encoding of raw loops into integer pairs is therefore:

```
/-- Raw encode map ‘Path torusBase torusBase -> \(\mathbb{Z}\) \times \(\mathbb{Z}\)‘.
     -/
@[simp] def torusEncodePath :
    Path torusBase torusBase -> Int \times Int :=
  fun p => torusCodeToProd (torusEncodeRaw torusBase p)

/-- Interpret a pair of integers as a raw torus loop. -/
def torusDecodePath (z : Int \times Int) :
    Path torusBase torusBase :=
  Path.trans (torusLoop1PathZPow z.1) (torusLoop2PathZPow z.2)
```

The helper paths `torusLoop1PathZPow` and `torusLoop2PathZPow` mirror the circle's integer powers but track each generator separately. Crucially, the commuting relation between the two loops is encoded in the rewrite system, so integer pairs form a normal form for torus loops without the need for an additional word language.

7.1 Loop quotients and decode/encode

The quotient-level definitions follow the same pattern as before:

```
abbrev TorusLoopQuot := LoopQuot Torus torusBase
abbrev torusPiOne    := PiOne   Torus torusBase

@[simp] def torusEncode : torusPiOne -> Int \times Int :=
```

[8]See [RdVdQdO25].

```
  torusEncodeLift

@[simp] def torusDecode : Int \times Int -> torusPiOne :=
  fun z => LoopQuot.ofLoop (torusDecodePath z)
```

Here `torusEncodeLift` is the quotient lift of `torusEncodePath`. The key lemmas establish step laws analogous to those on the circle, for example:

```
theorem torusEncode_comp_loop1 (x : torusPiOne) :
    torusEncode (LoopQuot.comp x torusLoop1Class) =
      ((torusEncode x).1 + 1, (torusEncode x).2)

theorem torusEncode_comp_loop2 (x : torusPiOne) :
    torusEncode (LoopQuot.comp x torusLoop2Class) =
      ((torusEncode x).1, (torusEncode x).2 + 1)
```

and similarly for inverses, subtracting in the appropriate coordinate. These statements are proved by unfolding the definitions, picking representatives, and applying the transport lemmas that describe how `torusCode` behaves along the two generators.

On the decoding side we iterate `torusLoop1` and `torusLoop2` according to the integer pair and show that this process respects addition and subtraction in each component. The raw encode/decode pair satisfies:

$$\texttt{torusEncodePath (torusDecodePath z)} = z, \qquad \texttt{torusDecode (torusEncode x)} = x,$$

with the proofs again following from normalization and the fact that the rewrite system can shuffle generator occurrences past each other. Passing to the quotient yields the desired group isomorphism:

```
noncomputable def torusPiOneEquivIntProd :
    SimpleEquiv torusPiOne (Int \times Int) where
  toFun := torusEncode
  invFun := torusDecode
  left_inv := torusDecode_torusEncode
  right_inv := torusEncode_torusDecode
```

Consequently $\pi_1(T^2)$ is definitionally the product $\mathbb{Z} \times \mathbb{Z}$ inside Lean, and the entire encode/decode algebra (addition laws, inversion, cancellation) is available as '[simp]' lemmas for subsequent developments.

8 Engineering and automation

The formalization described in this paper spans a substantial Lean code base. While the underlying mathematics is classical, encoding it in a modern proof assistant raises several engineering challenges.

8.1 Library design

A central design goal was to make the computational paths library usable beyond the specific case studies of the circle and the torus. To this end we separated the core path and rewrite machinery from the topological examples. The path library itself does not mention specific spaces; it only knows about types, paths, and rewrites. The circle and torus are defined in separate modules that can be imported selectively.

We also made deliberate choices about where to expose quotient types. In some parts of the library it is convenient to reason about raw paths and rewrites explicitly. In others it is more convenient to work modulo rewrite equivalence from the beginning. We expose both viewpoints and provide translation lemmas. For instance, we have lemmas that transport a function defined on raw paths to a function on equivalence classes, provided it respects the rewrite relation.

Another design consideration is universe management. The `Path` type is universe polymorphic, and the groupoid structure is formulated at an arbitrary universe level. This makes it possible to instantiate the theory both at small types such as `Circle` and `Torus` and at more complex types that may live in higher universes.

8.2 Automation and proof search

Several lemmas in the development involve long sequences of rewrite steps that are conceptually simple but tedious to write by hand. For these we rely on a combination of Lean tactics, bespoke rewrite tactics tailored to the computational path system, and external assistance from large language models that suggest candidate proof scripts.

We implemented small custom tactics that mimic the informal reasoning used in the original computational paths papers. One such tactic repeatedly applies the identity elimination rules and inverse cancellation rules until no further simplification is possible. Another tactic searches for opportunities to use the torus commuting lemma to swap adjacent loops and bring a word closer to normal form.

For example, a typical lemma might state that a complex composite of loops on the torus can be rewritten to a normal form. The high level structure of the proof is clear: repeatedly apply commuting and identity rewrites until the path is normalized. We encapsulate these patterns into a tactic that searches for applicable rewrite rules and applies them until no further progress is possible. This is particularly important in the torus development, where naive proofs quickly become unreadable.

8.3 Role of AI assistance

During the development we experimented with AI assisted proof search. Large language models can suggest candidate Lean proof scripts that are then checked by the Lean kernel. This proved particularly useful in proofs that require many small and routine steps. We emphasize, however, that all final proofs are fully checked by Lean and that the logical soundness of the development does not depend on the internal reasoning of the AI tools.

Our workflow typically proceeds as follows. For a given lemma, we write a precise statement and a rough outline of the proof in comments. We then ask an AI system to propose a Lean script that follows this outline. The proposed script rarely works unchanged, but it often provides a useful starting point that we refine by hand. We then verify the final script with Lean. In some cases the AI suggestion was substantially different from our outline but still valid, which highlighted alternative proof strategies.

This experience suggests that computational paths are a good target for AI assisted formalization: the proofs are rich enough to be mathematically interesting, yet structured enough that local search and pattern based heuristics work well.

9 Related work

Our work sits at the intersection of several lines of research.

First, it contributes to the formalization of homotopy theoretic concepts in type theory based proof assistants. Formalizations of the fundamental group of the circle and torus exist in homotopy type theory and cubical type theory, often using path types as primitive [Uni13, LS13, CCHM18]. Our work offers an alternative based on computational paths and a rewrite based weak groupoid. It shows that the encode and decode arguments can be carried out in a setting where paths are explicit proof objects rather than primitive identity types.

Second, it complements the original theoretical work on computational paths. That work develops the logical system, the rewrite rules, and several applications, but it does not provide a mechanization in a contemporary proof assistant [dQdOR16, dVRdQdO19]. Our Lean development validates these constructions and exposes the computational details of the rewrite system. For example, the need to prove invariance of winding numbers under `RwStar` forces one to make precise which rewrite rules are allowed and how they interact.

Third, it connects to the growing body of work on AI assisted theorem proving. We use AI tools to generate candidate proofs and refactor code, while relying on Lean's kernel for final verification [dMKA+15, BCM20]. This workflow illustrates how large language models can assist in building substantial mathematical libraries without compromising soundness. It also suggests that computational paths, with their explicit proof terms and finite rewrite systems, may be a good playground for studying proof search strategies.

Finally, our work adds to a broader effort to formalize algebraic topology in proof assistants. There are formalizations of fundamental groups, covering spaces, and homology theories in various systems, each with its own balance between abstraction and concreteness. Computational paths occupy an interesting point in this spectrum: they are close to traditional equality reasoning, yet they admit a rich homotopical interpretation.

10 Conclusion and future work

We have presented a Lean 4 formalization of computational paths and their weak groupoid structure. Building on this foundation, we formalized loop spaces and fundamental groups and carried out two classical computations: the fundamental groups of the circle and of the torus. The development is packaged as a reusable library that can serve as a basis for further work on computational paths and homotopy theory in proof assistants.

There are several promising directions for future work.

- **Additional spaces.** One can extend the library with further examples such as wedges of circles, higher dimensional tori, and more general cell complexes. This would test the scalability of the computational paths framework and its implementation. In particular, free products of groups arising from wedges of circles are a natural target.
- **Higher structures.** The weak groupoid structure considered here is 1 dimensional. Extending the Lean formalization to higher groupoids and higher paths would bring computational paths closer to the higher dimensional structures considered in homotopy type theory. This raises interesting questions about how to represent higher rewrite rules and coherence conditions.
- **Bridging to homotopy type theory.** It would be interesting to compare the computational paths formalization with existing homotopy type theory libraries, both conceptually and through concrete translations between the two frameworks. For instance, one could attempt to construct a functor from the groupoid of computational paths to a path groupoid built from the identity type.
- **Improved automation.** The rewrite heavy nature of the development suggests that more sophisticated automation could be helpful. Designing tactics that understand the algebra of paths more deeply is a natural next step. For example, one could implement a tactic that recognizes words in generators and decides equality by normalizing them.
- **Integration with other libraries.** Since the formalization builds on mathlib, there is scope for integrating computational paths with other algebraic and categorical developments. For example, one could connect the fundamental groups computed here with homology computations or with higher categorical structures.

We hope that this development will serve as a starting point for a broader exploration of computational paths in formalized mathematics and that it will stimulate further interactions between proof theory, homotopy theory, and interactive theorem proving.

Acknowledgements

We thank the Lean community for developing the tools that made this formalization possible.

References

[Awo12] Steve Awodey. Type theory and homotopy. In Peter Dybjer, Sten Lindstr"om, Erik Palmgren, and G"oran Sundholm, editors, *Epistemology versus Ontology*, volume 27 of *Logic, Epistemology, and the Unity of Science*, pages 183–201. Springer, 2012.

[BCM20] Kevin Buzzard, Johan Commelin, and Patrick Massot. The lean mathematical library. In *Intelligent Computer Mathematics*, volume 12236 of *Lecture Notes in Computer Science*, pages 109–123. Springer, 2020.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[CCHM18] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders M"ortberg. Cubical type theory: A constructive interpretation of the univalence axiom. *Mathematical Structures in Computer Science*, 28(7):1228–1269, 2018.

[dMKA$^+$15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction – CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015.

[dQdOR16] Ruy J. G. B. de Queiroz, Anjolina G. de Oliveira, and Arthur F. Ramos. Propositional equality, identity types, and direct computational paths. *South American Journal of Logic*, 2(2):245–296, 2016. Special Issue: A Festschrift for Francisco Miraglia.

[dVRdQdO19] Tiago M. L. de Veras, Arthur F. Ramos, Ruy J. G. B. de Queiroz, and Anjolina G. de Oliveira. An alternative approach to the calculation of fundamental groups based on labeled natural deduction. *arXiv preprint arXiv:1906.09107*, 2019.

[HS94] Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 208–212. IEEE, 1994.

[KB70] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon, 1970.

[LS13] Daniel R. Licata and Michael Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *Proceedings of the 28th Annual IEEE Symposium on Logic in Computer Science*, pages 223–232. IEEE, 2013.

[Lum09] Peter LeFanu Lumsdaine. Weak ω-categories from intensional type theory. In *Typed Lambda Calculi and Applications*, volume 5608 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2009.

[ML84] Per Martin-L"of. *Intuitionistic Type Theory.* Bibliopolis, Naples, 1984.

[New42] M. H. A. Newman. On theories with a combinatorial definition of “equivalence”. *Annals of Mathematics*, 43(2):223–243, 1942.

[Ram18] Arthur F. Ramos. *Explicit Computational Paths in Type Theory.* PhD thesis, Universidade Federal de Pernambuco, Recife, Brazil, 2018.

[RdVdQdO25] Arthur F. Ramos, Tiago M. L. de Veras, Ruy J. G. B. de Queiroz, and Anjolina G. de Oliveira. Computational paths in lean. `https://github.com/Arthur742Ramos/ComputationalPathsLean`, 2025. Version accessed November 22, 2025.

[Uni13] Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study, 2013.

[vdBG11] Benno van den Berg and Richard Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011.