Cubical Synthetic Homotopy Theory

Anders Mörtberg
Department of Mathematics
Stockholm University
Stockholm, Sweden
anders.mortberg@math.su.se

Loïc Pujet
Équipe Gallinette
Inria, LS2N
France
loic.pujet@inria.fr

Abstract

Homotopy type theory is an extension of type theory that enables synthetic reasoning about spaces and homotopy theory. This has led to elegant computer formalizations of multiple classical results from homotopy theory. However, many proofs are still surprisingly complicated to formalize. One reason for this is the axiomatic treatment of univalence and higher inductive types which complicates synthetic reasoning as many intermediate steps, that could hold simply by computation, require explicit arguments. Cubical type theory offers a solution to this in the form of a new type theory with native support for both univalence and higher inductive types. In this paper we show how the recent cubical extension of Agda can be used to formalize some of the major results of homotopy type theory in a direct and elegant manner.

CCS Concepts • **Theory of computation → Constructive mathematics**; **Type theory**.

Keywords Synthetic Homotopy Theory, Cubical Type Theory, Homotopy Type Theory, Constructive Mathematics.

ACM Reference Format:
Anders Mörtberg and Loïc Pujet. 2020. Cubical Synthetic Homotopy Theory. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3372885.3373825

1 Introduction

Homotopy type theory (HoTT) is a new foundation that has emerged from recently discovered connections between type theory and homotopy theory [27]. These foundations are based on the observation that the inductively defined *equality types* in type theory behave like *paths* in homotopy theory. On the one hand, HoTT provides type theory with extensionality principles that were previously not typically available, including function and propositional extensionality (pointwise equal functions and logically equivalent propositions are equal). These principles follow from a more general extensionality principle called *univalence*. This principle was originally proposed by Vladimir Voevodsky in the form of his celebrated *univalence axiom* [31] that can be consistently assumed in type theory [17]. On the other hand, HoTT provides methods for reasoning formally about homotopical notions in a synthetic manner. This is made possible through *higher inductive types* (HITs) that allow spaces to be directly represented as types with nontrivial equality/path constructors.

In recent years HoTT has been developed axiomatically using various proof assistants. Indeed, all of the major proof assistants based on type theory have HoTT libraries: HoTT-Agda [7], Coq-HoTT [4], Lean-HoTT [29], UniMath [33]. Many impressive results can be found in these libraries, for example, the formalization of computations of many homotopy groups of spheres [5, 18, 19], the Seifert-van Kampen theorem [15], Blakers-Massey theorem [14] and Serre's spectral sequence [28]. However, despite these successes some constructions have turned out to be surprisingly difficult. An example of this is the proof that the torus is equivalent to the product of two circles: the first version required an impressive amount of complicated path algebra, worked out by Sojakova [25]. The proof was later simplified by Licata and Brunerie [20] using cubical ideas, but it was still highly nontrivial. The main reason for the difficulties in formalizing this proof is the complicated path algebra arising from the fact that many equalities do not hold definitionally. These problems can be traced back to univalence and HITs being axiomatically added in HoTT; in particular, the computation rules for the higher constructors of HITs are typically postulated and do not hold definitionally.

These issues have been resolved in a uniform way by the recent invention of *cubical* type theories [2, 3, 12]. This is a family of type theories with native support for all of the notions of HoTT, in particular univalence and HITs. The fundamental idea underlying cubical type theory is to take the homotopical intuitions from HoTT seriously and incorporate them in the judgmental framework of the type theory. As these homotopical notions are built into the theory, they have computational meaning, which simplifies many proofs. Indeed, many proofs that involve complicated path

CPP '20, January 20–21, 2020, New Orleans, LA, USA

ACM ISBN 978-1-4503-7097-4/20/01.
https://doi.org/10.1145/3372885.3373825

algebra in HoTT may simply be proved by reflexivity in cubical type theory. For instance, the proof that the torus is equivalent to the product of two circles is trivial to prove in cubical type theory using pattern matching and reflexivity as shown in section 3.

There are multiple variations and implementations of cubical type theory—the original formulation of Cohen et al. [12] used an interval endowed with the structure of a *De Morgan algebra* while the more recent *cartesian* cubical type theories use an interval with less structure [2, 3]. All of the results in this paper have been formalized[1] using the recent cubical extension [30] of the Agda proof assistant [1] that is based on the De Morgan variation of cubical type theory. However, despite some technical differences between the underlying theories, all of the proofs could be carried with similar complexity in a system based on cartesian cubical type theory, like redtt [26].

While the proofs in cubical type theory often resemble the original HoTT proofs some work is typically required to make them more "cubical" in order to take full advantage of the cubical primitives. In particular, the use of path-induction that is ubiquitous in HoTT is kept to a minimum, and one typically instead uses the more elementary operations of cubical type theory such as transport and composition. This results in a new way of mechanizing synthetic homotopy theory, leading to (arguably) more direct and elegant proofs that closely resemble the homotopical arguments.

Contributions/Outline The main goal of this paper is to show the practical gains of using a system with native support for univalence and higher inductive types for formalizing synthetic homotopy theory. We exemplify this by formalizing the following results from HoTT:

- The equivalence of the torus and two circles together with the computation of their respective fundamental groups (section 3).
- The equivalence between direct definitions of low dimensional spheres, and alternative definitions using iterated suspensions (section 4.1).
- Definition of pushout together with a direct proof of the "3×3 lemma" (section 4.2).
- Definition of the join of two types and a proof that it is associative (section 4.3). Using this we get two proofs, one inspired by HoTT and a new direct cubical proof, that $\mathbb{S}^3$ is equivalent to the join of two circles.
- Definitions of the Hopf fibration and a proof that its total space is $\mathbb{S}^3$ (section 5).

The paper ends with some discussions of future work and a comparison with related work in the HoTT libraries of the major proof assistants based on type theory (section 6).

[1]The developments can be found in the AGDA/CUBICAL library located at: https://github.com/agda/cubical

2 Cubical Agda

The cubical type theory that Cubical Agda is based on is heavily inspired by the one of Cohen et al. [12], more precisely on the variation outlined by Coquand et al. [13] that is well-suited for HITs. The goal of this section is to give sufficient background for readers not familiar with cubical type theory and Cubical Agda to be able to understand the examples in this paper. Readers who are familiar with Cubical Agda can hence skip this section and we refer curious readers to the paper of Vezzosi et al. [30] for a comprehensive technical treatment of all of the features of Cubical Agda.

2.1 The Interval and Path Types

The first addition to make Agda *cubical* is an interval type I with endpoints i0 and i1. This plays the role of the real interval $[0, 1] \subset \mathbb{R}$ in homotopy theory, however in Cubical Agda this is a purely formal object. A variable $i : \mathsf{I}$ represents a point varying continuously between the two endpoints. The interval is equipped with three basic operations: *minimum* (_∧_ : I → I → I), *maximum* (_∨_ : I → I → I) and *reversal* (~_ : I → I). These operations form a *De Morgan algebra*, that is, a bounded distributive lattice (i0, i1, _∧_, _∨_) with a De Morgan involution ~_.

A function out of the interval into one of Agda's universes of types (Set ℓ) represents a line between two types. By iterating this we obtain squares, cubes and hypercubes of types making Agda inherently *cubical*. It is often useful to specify the endpoints of a line, which is done via *path types* (we omit the quantification of the universe level ℓ for readability):

```
PathP : (A : I → Set ℓ) → A i0 → A i1 → Set ℓ
```

Paths are introduced by lambda abstractions

$$\lambda\, i \longrightarrow t : \mathsf{PathP}\ A\ t[\mathsf{i0}\ /\ i]\ t[\mathsf{i1}\ /\ i]$$

provided that $t : A\ i$ for $i : \mathsf{I}$. Given $p : \mathsf{PathP}\ A\ a_0\ a_1$ we can apply it to $r : \mathsf{I}$ and obtain $p\ r : A\ r$. Also, we always have that $p\ \mathsf{i0}$ reduces to a_0 and $p\ \mathsf{i1}$ reduces to a_1.

The PathP types should be thought of as representing heterogeneous equalities since the two endpoints are in different types; this is similar to the dependent paths in HoTT [27, Sect. 6.2]. We define the type of non-dependent paths/equalities in terms of PathP as follows:

```
_≡_ : {A : Set ℓ} → A → A → Set ℓ
_≡_ {A = A} x y = PathP (λ _ → A) x y
```

The syntax $\{A = A\}$ tells Agda to bind the hidden argument A (first A) to a variable A (second A) that can be used on the right hand side of the definition. Viewing equalities as paths allows us to reason about equality; for instance, the constant path represents a proof of reflexivity.

```
refl : {x : A} → x ≡ x
refl {x = x} = λ i → x
```

Path types also let us prove new things that are not provable in standard Agda. For example, function extensionality, stating that pointwise equal functions are equal themselves, has a very simple proof:

```
funExt : {f g : A → B} → ((x : A) → f x ≡ g x) → f ≡ g
funExt p i x = p x i
```

The proof of function extensionality for dependent and n-ary functions is equally direct. Since funExt is a *definable notion* in Cubical Agda it has computational content: it simply swaps the arguments to p.

We can also use the basic operations on I to construct various useful operations, for example the reversal of a path is defined using ~_ and represents the fact that ≡ is symmetric.

```
_⁻¹ : {x y : A} → x ≡ y → y ≡ x
p ⁻¹ = λ i → p (~ i)
```

2.2 Transport and Composition

One of the key operations with type theoretic equality is *transport*: given an equality/path between types we get a function between these types. In Cubical Agda this is defined using another primitive called transp. However, for the examples in this paper the transport function suffices.

```
transport : A ≡ B → A → B
transport p a = transp (λ i → p i) i0 a
```

The substitution principle is an instance of transport.

```
subst : (B : A → Set ℓ) {x y : A} → x ≡ y → B x → B y
subst B p b = transport (λ i → B (p i)) b
```

This function invokes transport with a proof that the family B respects the equality p:

$$\lambda\ i \to B\ (p\ i) : B\ x \equiv B\ y$$

By combining the transport and minimum operations we can define an induction principle for paths that resemble the eliminator for Martin-Löf's inductively defined identity types [21].

```
J : {x : A} (P : ∀ y → x ≡ y → Set ℓ) (d : P x refl)
    {y : A} (p : x ≡ y) → P y p
J P d p = transport (λ i → P (p i) (λ j → p (i ∧ j))) d
```

However, an important difference between the cubical path types and Martin-Löf's identity types (and the path types in HoTT) is that cubical path types do not behave like inductive types. In particular, the above definition does not definitionally satisfy the computation rule when applied to refl. Nevertheless, we can still prove that it holds up to a path:

```
JRefl : {x : A} (P : ∀ y → x ≡ y → Set ℓ) (d : P x refl) →
        J P d refl ≡ d
```

This is a subtle, but important difference between cubical type theory and HoTT. Readers familiar with HoTT might be worried that the failure of this equality holding definitionally complicates many proofs, however in our experience this is rarely the case because many proofs that require path induction in HoTT can be proved in more direct ways using cubical primitives. Furthermore, for closed terms (potentially depending on interval variables) the lack of this definitional equality in cubical type theory does not affect canonicity, in particular any closed term of type natural numbers is definitionally equal to a numeral as proved by Huber [16].

Cubical Agda also has a primitive operation for composing paths and, more generally, for composing higher dimensional cubes. This operation is called *homogeneous composition* and to describe it we need to be able to write partially specified n-dimensional cubes, i.e., cubes where some faces are missing. For this Cubical Agda has partial cubical types, written Partial r A. The idea is that Partial r A is the type of cubes in A that are only defined when (r = i1) holds. For example, Partial (i ∨ ~ i) A is a type that is only defined when i is i1 or i0 (this means that it corresponds to the two endpoints of a line). Elements of these partial cubical types are introduced using pattern matching lambdas. For this purpose Cubical Agda supports a new form of patterns, here (i = i1) and (i = i0), that specify the cases of a partial element:

```
partialBool : ∀ i → Partial (i ∨ ~ i) Bool
partialBool i = λ { (i = i1) → true ; (i = i0) → false }
```

The term partialBool should be thought of as a boolean depending on i with different values when i is i1 and when i is i0. This is only defined on the two endpoints of I and there is no way to extend this partial type to a regular dependent boolean, since true is not equal to false.

Cubical Agda also has cubical subtypes as in Cohen et al. [12]; given A : Set ℓ, r : I and u : Partial r A we can form the type A [$r \mapsto u$]. A term v of this type is a term of type A that is definitionally equal to u when (r = i1) is satisfied. Any term u : A can be seen as a term of type A [$r \mapsto u$] that agrees with itself when (r = i1):

```
inS : {r : I} (a : A) → A [ r ↦ (λ _ → a) ]
```

We can also forget that a partial element agrees with u when (r = i1):

```
outS : {r : I} {u : Partial r A} → A [ r ↦ u ] → A
```

These two operations are inverse to each other when well-typed. Using this cubical infrastructure we can now give the type of the homogeneous composition operation.

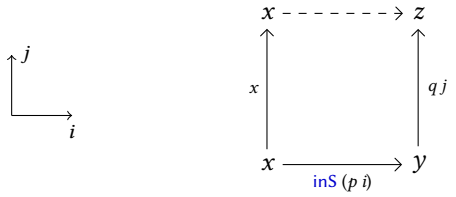
```
hcomp : {r : I} (u : I → Partial r A) (u₀ : A [ r ↦ u i0 ]) → A
```

When calling hcomp {r = r} u u_0, Cubical Agda makes sure that u_0 agrees with u i0 on r; this is specified in the type of u_0. The idea is that u_0 is the base and u specifies the sides of an open box where the side opposite of u_0 is missing.

The hcomp operation then gives us the missing side. For example binary composition of paths can be written as:

```
_·_ : {x y z : A} → x ≡ y → y ≡ z → x ≡ z
_·_ {x = x} p q i = hcomp {r = i ∨ ~ i}
                         (λ j → λ { (i = i0) → x
                                  ; (i = i1) → q j })
                         (inS (p i))
```

Pictorially we are given $p : x \equiv y$ and $q : y \equiv z$, and the composite of the two paths is obtained by computing the dashed top of the following square.



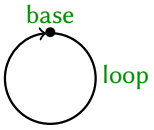
By composing paths and higher cubes using hcomp, we can reason about equalities/paths in a very direct way, avoiding the use of path induction.

2.3 Higher Inductive Types

The fact that Cubical Agda has native support for HITs means that we can define the circle HIT using an Agda data declaration.

```
data S¹ : Set where
  base : S¹
  loop : base ≡ base
```

This is a type with a point constructor base and a nontrivial equality/path constructor loop connecting base to itself as shown in the drawing below.



Functions out of HITs are written using normal Agda pattern matching equations. The following function wraps the circle twice around itself, by composing the loop constructor with itself:

```
double : S¹ → S¹
double base = base
double (loop i) = (loop · loop) i
```

The second case defines a *definitional* equality, while in HoTT this would have had to be defined using the postulated recursion principle for the circle. This means that in HoTT double applied to loop would not reduce automatically, but a proof of equality would instead have to be applied manually. This leads to rather bureaucratic proofs as one then has to handle these explicit applications of the computation rule when reasoning about double. Furthermore, this is not very natural if one wants to use HITs for programming.

In order for the circle to be the free type generated by base and loop it also needs to have elements of the form loop · loop as in the above definition of double. In general, for HITs hcomp $(\lambda\, i \to \lambda\, \{\ (r = \mathsf{i1}) \to u\})\ u_0$ only reduces to u[i1 / i] when r is i1, and is to be considered a canonical element otherwise. The circle hence has constructors of the form hcomp $u\ u_0$ in addition to base and loop. When typechecking definitions like the one for double, Cubical Agda checks that the boundary of all cases are correct, in particular that the defining equation

```
double (loop i) = (loop · loop) i
```

agrees with double base when i is i0 or i1.

2.4 Glue Types and Univalence

The final extension of Cubical Agda relevant for this paper are the Glue types of Cohen et al. [12] that let us give computational content to univalence. Given that a type in cubical type theory represents a higher dimensional cube, Glue types let us construct a cube where some sides have been replaced by equivalent types.

There are many ways to define the notion of "equivalence of types" in HoTT; Cubical Agda uses the definition that two types are equivalent if there is a function between them with *"contractible fibers"* following the terminology of Voevodsky [32] and the HoTT book [27]. Spelled out, a map $f : A \to B$ is an equivalence if the preimage of any point in B is a singleton type. We write $A \simeq B$ if there is a chosen equivalence $f : A \to B$ between A and B. A key result is that isomorphisms, in the sense of a section-retraction pair of functions, give rise to equivalences. In particular, the identity function is an equivalence: idEquiv $A : A \simeq A$.

The Glue types take a partial family of types A that are equivalent to the base type B. These types are then "glued" onto B and the equivalence data gets packaged up into a new type.

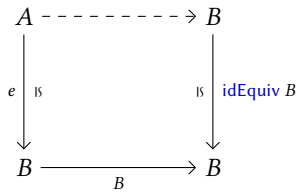
```
Glue : (B : Set ℓ) {r : I} →
       Partial r (Σ[ A ∈ Set ℓ ] (A ≃ B)) → Set ℓ
```

These types are introduced using the glue constructor and eliminated using the unglue operation. Examples of this will be discussed in the proof of theorem 5.2. Using Glue types we can turn an equivalence of types into a path:

```
ua : {A B : Set ℓ} → A ≃ B → A ≡ B
ua {A = A} {B = B} e i =
  Glue B (λ { (i = i0) → (A , e) ; (i = i1) → (B , idEquiv B) })
```

The idea is that we glue A onto B when i is i0 using e and B onto itself when i is i1 using the identity equivalence. The term ua e is a path from A to B as the Glue type reduces when the face conditions are satisfied—when i is i0 this reduces to A and when i is i1 it reduces to B. Pictorially we

can describe ua e as the dashed top line in:



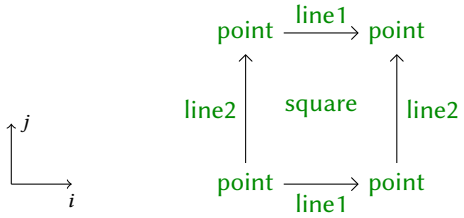
This lets us prove the standard formulation of the univalence axiom, however, for all of the examples in this paper the ua function suffices.

3 The Circle and Torus

We already gave the definition of the circle as a HIT in section 2.3. To define the torus as a datatype in Cubical Agda we write the following:

```
data Torus : Set where
  point  : Torus
  line1  : point ≡ point
  line2  : point ≡ point
  square : PathP (λ i → line1 i ≡ line1 i) line2 line2
```

The idea is that the Torus has a base point with two nontrivial path constructors connecting it to itself and a square relating the two paths. This square can be illustrated by:



The type of the square constructor captures this by identifying line2 with itself *over* line1. In order to see that this represents a torus imagine square being made of a piece of paper that is folded so that the opposite sides are matched up. The proof that the Torus type is equivalent to two circles in Cubical Agda was given by Vezzosi et al. [30, Section 2.4.1], but we recall it here for completeness. We first write a function from the torus to two circles using pattern matching.

```
t2c : Torus → S¹ × S¹
t2c point        = (base  , base)
t2c (line1 i)    = (loop i , base)
t2c (line2 j)    = (base  , loop j)
t2c (square i j) = (loop i , loop j)
```

To prove that this is an equivalence we need to define its inverse c2t : S¹ × S¹ → Torus. This function is also defined by pattern matching in the obvious way. Proving that the two maps cancel is then simply done by pattern matching with refl in all cases.

```
c2t-t2c : (t : Torus) → c2t (t2c t) ≡ t
c2t-t2c point       = refl
c2t-t2c (line1 _)   = refl
c2t-t2c (line2 _)   = refl
c2t-t2c (square _ _) = refl
```

The converse, t2c-c2t : $(p : S^1 \times S^1) \to$ t2c (c2t p) ≡ p, is equally trivial to prove. We can then package this up as an equality using isoToPath which combines ua with the proof that any isomorphism is an equivalence.

```
Torus≡S¹×S¹ : Torus ≡ S¹ × S¹
Torus≡S¹×S¹ = isoToPath (iso t2c c2t t2c-c2t c2t-t2c)
```

The simplicity of this cubical proof relies on the computation rules for all constructors of HITs holding definitionally in Cubical Agda. In HoTT the computation rules for the higher constructors would have to be postulated which means that they do not hold definitionally. This is exactly what makes the proofs of c2t-t2c and t2c-c2t surprisingly nontrivial in HoTT.

3.1 The Loop Spaces of the Circle and Torus

The loop spaces of the circle and torus are defined as follows:

```
ΩS¹ : Set
ΩS¹ = base ≡ base


ΩTorus : Set
ΩTorus = point ≡ point
```

The goal of this section is to prove that ΩS¹ is equivalent to the integers. This proof is a cubical adaptation of the proof of Licata and Shulman [18]. We can then combine this with the above equivalence between the torus and two circles to also compute the loop space of the torus. Note that we are computing loop spaces and not fundamental groups. However, as the fundamental group is defined as the set-truncation of the loop space and these loop spaces are both sets, we get that they coincide with the fundamental groups.

The first step in computing the loop space of the circle is to define a function computing "winding numbers", i.e. the net number of times an element of ΩS¹ goes around the circle clockwise. To do this we first prove that the successor function on the integers, Int, is an equivalence (its inverse is the predecessor function). By applying ua we then get a nontrivial equality/path from Int to Int that we call sucPathInt. Using this we can define:

```
helix : S¹ → Set
helix base = Int
helix (loop i) = sucPathInt i

winding : ΩS¹ → Int
winding p = subst helix p (pos 0)
```

Applying the winding function to an element of ΩS¹ will compute its winding number. For example, we can compute the winding numbers +3 and −1 as follows:

```
_ : winding (loop · loop · loop) ≡ pos 3
_ = refl

_ : winding (loop ⁻¹ · loop · loop ⁻¹) ≡ negsuc 0
_ = refl
```

The term negsuc n represents the number $-(n+1)$, so that negsuc 0 is indeed −1. Note that none of these examples would have reduced to a numeral in HoTT as they would have been stuck on transporting along ua. The proofs of these would hence not be proved by refl, but rather by manually rewriting with the postulated computation rules for univalence.

In order to prove that ΩS¹ ≡ Int we just have to define an inverse function to winding. This is easily done via pattern matching.

```
loopn : Int → ΩS¹
loopn (pos zero) = refl
loopn (pos (suc n)) = loopn (pos n) · loop
loopn (negsuc zero) = loop ⁻¹
loopn (negsuc (suc n)) = loopn (negsuc n) · loop ⁻¹
```

It is then easy to prove that the winding number of an n-fold loop is n.

```
winding-loopn : (n : Int) → winding (loopn n) ≡ n
winding-loopn (pos zero) = refl
winding-loopn (pos (suc n)) i =
  sucInt (winding-loopn (pos n) i)
winding-loopn (negsuc zero) = refl
winding-loopn (negsuc (suc n)) i =
  predInt (winding-loopn (negsuc n) i)
```

Note that we parameterize by i in the second and fourth case of winding-loopn. The reason for this is that we are constructing an element of ≡, i.e. a function out of I. This use of interval variables hence lets us inline cong/ap.

However, when trying to prove the other composition one quickly realizes that it is not as easy as there is no direct induction principle for ΩS¹. Luckily there is an ingenious solution to this offered by HoTT in the form of the *encode-decode method* [27, Section 8.9]. The trick is to generalize the loopn and winding functions as follows:

```
encode : ∀ x → base ≡ x → helix x
encode x p = subst helix p (pos 0)

decode : (x : S¹) → helix x → base ≡ x
decode base = loopn
decode (loop i) = {- ... -}
```

Note that decode base is loopn and that encode base is winding definitionally. The loop case of decode is a bit longer to define and we omit it due to space constraints. However, it is fairly direct to define using the cubical primitives and we refer the interested reader to the formalization. The main reason for generalizing the functions as above is that we can now use path induction to prove the following:

```
decodeEncode : (x : S¹) (p : base ≡ x) →
                 decode x (encode x p) ≡ p
decodeEncode x p =
  J (λ y q → decode y (encode y q) ≡ q) (λ x → refl) p
```

The special case decodeEncode base proves the desired composition, i.e. that loopn (winding x) ≡ x for all x : ΩS¹. We may then package this up to get the desired equality:

```
ΩS¹≡Int : ΩS¹ ≡ Int
ΩS¹≡Int = isoToPath (iso winding loopn
                        winding-loopn
                        (decodeEncode base))
```

By combining this result with the equality between the torus and two circles we obtain:

```
ΩTorus≡Int×Int : ΩTorus ≡ Int × Int
```

It is now possible to transport along this equality to compute winding numbers on the torus. However, this will not result in the most efficient function possible and we can easily write a more direct function for computing these numbers as follows.

```
windingTorus : ΩTorus → Int × Int
windingTorus l = ( winding (λ i → t2c (l i) .fst)
                 , winding (λ i → t2c (l i) .snd))
```

Just like winding this function also computes as expected:

```
_ : windingTorus (line1 · line2) ≡ (pos 1 , pos 1)
_ = refl

_ : windingTorus (line1 ⁻¹ · line2 · line1) ≡ (pos 0 , pos 1)
_ = refl
```

4 Suspension, Spheres and Pushouts

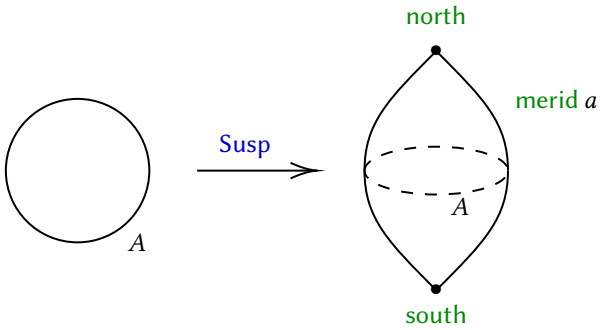
In this section we discuss various results about spheres, culminating in two proofs that the 3-dimensional sphere is equal to the join of two circles — a result that we will need when computing the total space of the Hopf fibration later on. On the way to this result we introduce the pushout and join HITs. We also prove some useful results about these, including the "3×3 lemma" for pushouts and associativity of the join.

4.1 Suspension

The suspension of a type A is built from two distinguished points north and south, along with a path from north to south for every point of A (and a homotopy square for every path in A, etc.):

```
data Susp (A : Set) : Set where
  north : Susp A
  south : Susp A
  merid : (a : A) → north ≡ south
```

The suspension of a type A is depicted below.



In this drawing A is a circle, in which case the suspension is a sphere. This generalizes well, and we can define the higher spheres as iterated suspensions starting from the booleans.

```
_-sphere : ℕ → Set
(0)-sphere = Bool
(suc n)-sphere = Susp ((n)-sphere)
```

We may now prove that our initial definition for the circle is equal to the suspension of the booleans.

Lemma 4.1. *The (1)-sphere is equal to the circle S¹.*

Proof. We will prove this by constructing an isomorphism and then converting it to an equality using univalence. We first define a map from (1)-sphere to S¹ in the following way, collapsing merid false to base:

```
s2c : (1)-sphere → S¹
s2c north          = base
s2c south          = base
s2c (merid false i) = base
s2c (merid true i)  = loop i
```

In the other direction, we use composition to go around the (1)-sphere in one go.

```
c2s : S¹ → (1)-sphere
c2s base = north
c2s (loop i) = (merid true · merid false ⁻¹) i
```

To construct the first canceling homotopy, we have to find a homotopy between s2c (c2s loop) = loop · refl and loop. This is proved in one of the lemmas in the cubical library that says that refl is the right unit for _·_.

```
s2c-c2s : (x : S¹) → s2c (c2s x) ≡ x
s2c-c2s base = refl
s2c-c2s (loop i) j = rUnit loop (~ j) i
```

The second homotopy is slightly more involved. After going back and forth, merid false has been collapsed to the north pole, while merid true has been stretched to go around the whole circle. As such, we need to move the south pole back into place along merid false, and deform the two meridians accordingly:

```
c2s-s2c : (x : (1)-sphere) → c2s (s2c x) ≡ x
c2s-s2c north j = north
c2s-s2c south j = merid false j
c2s-s2c (merid false i) j = h1 i j
c2s-s2c (merid true i) j = h2 i j
```

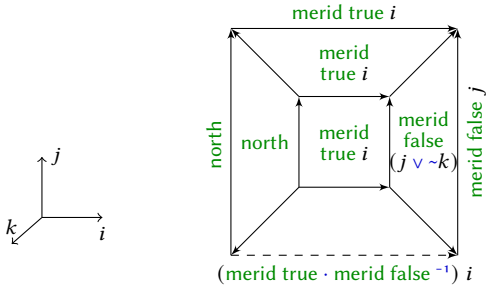
We need h1 to be a homotopy from the constant path at north to the original path merid false, such that the restriction to $(i = \mathsf{i1})$ matches merid false j and the restriction to $(i = \mathsf{i0})$ matches north. This is easily achieved using _∧_:

```
h1 : I → I → (1)-sphere
h1 i j = merid false (i ∧ j)
```

On the other hand, h2 has to be a homotopy from merid true · merid false ⁻¹ to merid true, with the same condition on the restrictions to $(i = \mathsf{i0})$ and $(i = \mathsf{i1})$. We can do that using hcomp to paste several homotopies together:

```
h2 : I → I → (1)-sphere
h2 i j = hcomp (λ k → λ { (i = i0) → north
                        ; (i = i1) → merid false (j ∨ ~ k)
                        ; (j = i1) → merid true i })
               (merid true i)
```

The composition can be pictured as follows, showing both the outer edge constraints and the inner faces we used:



The $(j = \mathsf{i0})$ face of the open cube matches the square used to define the composition merid true · merid false ⁻¹ — this breaks the composition into refl, merid true, and merid false ⁻¹ on the inner edges. Thus, we use the $(i = \mathsf{i1})$ face to retract the merid false ⁻¹ part into the south pole, using a connection. The other faces are just constant in directions j and k. From there, hcomp provides us with the front face of the cube, which is the homotopy we need.

This completes the definition of c2s-s2c, providing us with an isomorphism than can be converted into an equality with ua. □

Inspired by the direct definition of S^1 we can also give direct definitions of some higher spheres, for example S^2 and S^3 are defined as follows.

```
data S² : Set where
  base₂ : S²
  surf₂ : PathP (λ i → base₂ ≡ base₂) refl refl

data S³ : Set where
  base₃ : S³
  surf₃ : PathP (λ j → PathP (λ i → base₃ ≡ base₃) refl refl)
                refl refl
```

As expected we can prove that these are equal to the definitions using iterated suspensions. The proof of this lemma is very similar to the one of lemma 4.1 so we omit it.

Lemma 4.2. *The (2)-sphere is equal to S^2 and the (3)-sphere is equal to S^3.*

One might wonder if we can also give a direct definition of S^n in a similar fashion and prove that it is equal to the (n-1)-sphere. However, it is currently not possible to write the higher constructor corresponding to surf_n in Cubical Agda as this kind of HIT is not supported by any of the proposed schemas for cubical HITs [10, 13]. Interestingly it is in fact possible to *postulate* this HIT in HoTT or cubical type theory, and prove properties about it — for instance Licata and Brunerie [19] prove that $\pi_n(\mathbb{S}^n) = \mathbb{Z}$ using this direct definition.

4.2 Pushouts and the 3×3 Lemma

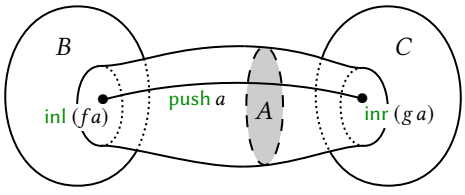
Suppose we are given a span of types:

$$B \xleftarrow{f} A \xrightarrow{g} C$$

Then the homotopical pushout of this span is a type that contains both a copy of B and a copy of C, and a path identifying $f\,a$ and $g\,a$ for every a in A. It can be defined as a HIT in the following way.

```
data Pushout {A B C : Set} (f : A → B) (g : A → C) : Set where
  inl : B → Pushout f g
  inr : C → Pushout f g
  push : (a : A) → inl (f a) ≡ inr (g a)
```

A generic homotopical pushout is illustrated below: it consists of an embedded copy of B and C, and a copy of $A \times I$ whose ends have been identified with corresponding points in B and C according to f and g.

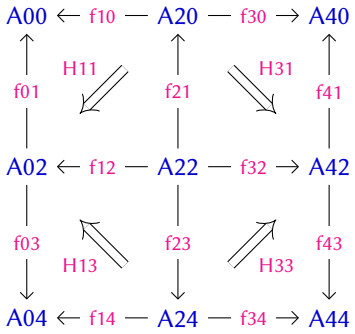


Pushouts can be used to construct various classical objects such as suspensions, joins and many more—hence their importance. For instance, the suspension of a type A is the homotopical pushout of the following span:

$$1 \longleftarrow A \longrightarrow 1$$

where 1 is the inductive type with only one constructor, and the arrows are the unique maps to 1.

Pushouts are often nested, and when dealing with them a fact known as the 3×3 lemma comes in very handy. Suppose we are given a double span of types



where the Hij are homotopies ensuring the commutativity of the diagram: for instance, H11 is of type f10 ∘ f21 ≡ f01 ∘ f12.

Then, there are two canonical ways to build a type from this diagram out of pushouts. Indeed, one can start by taking the pushouts of the three rows, to get the following (vertical) span

$$A\square 0 \xleftarrow{f\square 1} A\square 2 \xrightarrow{f\square 3} A\square 4 \tag{1}$$

in which we write A□0, A□2 and A□4 for the three pushouts of the rows. The maps between them, f□1 and f□3, are then defined using pattern matching.

```
f□1 : A□2 → A□0
f□1 (inl a) = inl (f01 a)
f□1 (inr a) = inr (f41 a)
f□1 (push a j) =
  hcomp (λ i → λ { (j = i0) → inl (H11 a (~ i))
                 ; (j = i1) → inr (H31 a (~ i)) })
        (push (f21 a) j)
```

Finally, we write A□○ for the pushout of the span in (1). Similarly, we could have started by taking the pushout of the columns and then computed the pushout A○□ of the resulting "vertical" span. The 3×3 lemma for pushouts then states that the results of these two constructions are equal.

Lemma 4.3. *(3×3 lemma) The two pushouts A□○ and A○□ are equal.*

Proof. In order to prove this we construct an isomorphism via pattern matching and apply univalence. To define a map A□○ → A○□, we will first need maps for the two sides of our pushout span:

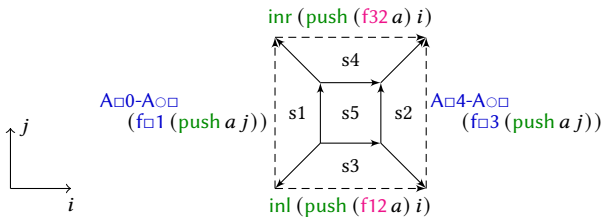
```
A□0-A○□ : A□0 → A○□
A□0-A○□ (inl x) = inl (inl x)
A□0-A○□ (inr x) = inr (inl x)
A□0-A○□ (push a i) = push (inl a) i
```

The map A□4-A○□ is defined analogously. Using these maps we can define one of the main maps:

```
A□○-A○□ : A□○ → A○□
A□○-A○□ (inl x) = A□0-A○□ x
A□○-A○□ (inr x) = A□4-A○□ x
A□○-A○□ (push (inl x) i) = inl (push x i)
A□○-A○□ (push (inr x) i) = inr (push x i)
A□○-A○□ (push (push a j) i) = filler a j i
```

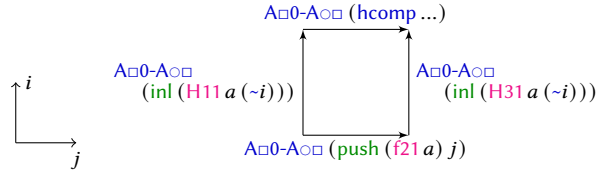
Note how we did not define a map A□2-A○□ to handle the push case, but instead proceeded via nested pattern matching. The reason is that when Agda checks boundary constraints, it does not perform η-expansion on its own so we have to force it in the function definition.

Most cases boil down to swapping the constructors, but this straightforward strategy will not be sufficient to handle the last case (which has been temporarily called filler). Indeed, we need to construct a square inside A○□ with a prescribed boundary. To do this, the most natural guess would be push (push a i) j, but its boundary for $i = i0$ is push (inl (f21 a)) j, which is not definitionally equal to the prescribed A□0-A○□(f□1 (push a j)). In order to fix this, we will use hcomp to glue the four squares $s1$–$s4$ around our candidate in order to ensure it matches the dashed boundary:

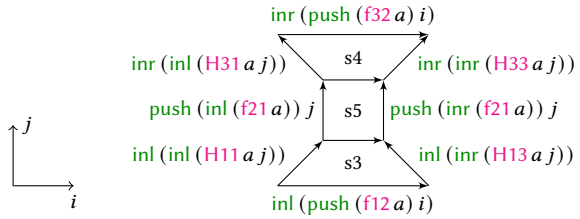


To do this we hence need to construct the squares $s1$–$s5$, in this order.

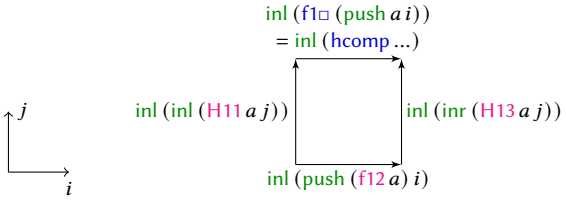
To construct $s1$, we unfold f□1 (push a j) on the left side, obtaining a composition of three paths. But functions commute with hcomp's up to a homotopy, so A□0-A○□ applied to the composition is homotopic to the composition of the images of the paths. That is, we can find a square with the following boundary:



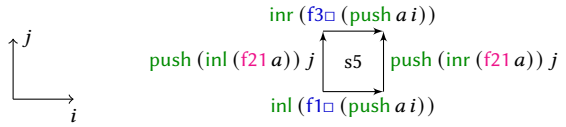
Once rotated counter-clockwise, we can use this square for $s1$ and the analogue for $s2$. After simplifying the applications of A□0-A○□, we are left with the following boundary:



Just like before, we can use the functoriality of inl to get a square with the following boundary:



We use this square for $s3$ and the appropriate analogue for $s4$. The only square left to construct now is



which we can fill with push (push a i) j. By combining these five squares using hcomp we obtain the definition of filler, and thus of our first map.

The opposite direction, A○□ → A□○, is the same up to a transposition of the 3×3 span. The proofs that these maps cancel are also similar, but the central fillers are more difficult to illustrate as they require one more dimension. We refer the interested reader to the formalization. By combining all of this we obtain the desired equality between A□○ and A○□. □

The formal proof of the 3×3 lemma is under 200 lines of code (LOC) in Cubical Agda. The corresponding result in HoTT-Agda is about 3000 LOC.[2] These numbers should of course be taken with a grain of salt as those files are not self-contained and rely on other results in the library. Regardless of this we still believe that it indicates the complexity of the involved path algebra in the HoTT proof compared to the cubical one.

[2]This number has been calculated by counting the number of lines (excluding comments) in: https://github.com/HoTT/HoTT-Agda/tree/master/theorems/homotopy/3x3

4.3 The Join and S^3

Another interesting example of a pushout is the join. The main importance of this HIT in this paper is that it gives us another characterization of S^3 as the join of two circles which will be useful when proving that the total space of the Hopf fibration is S^3. However, this construction also has many other interesting uses in HoTT as explored by Rijke [23].

The join of two types A and B is built from one copy of A, one copy of B, and a path from every $a : A$ to every $b : B$.

```
data Join (A : Set) (B : Set) : Set where
  inl : A → Join A B
  inr : B → Join A B
  push : ∀ a b → inl a ≡ inr b
```

It is easy to see that Join A B can alternatively be defined as the pushout of the following span—the inductive definitions only differ by currying.

$$A \xleftarrow{\text{fst}} A \times B \xrightarrow{\text{snd}} B$$

We now proceed by proving a few lemmas relating joins and spheres.

Lemma 4.4. *Join Bool $A \equiv$ Susp A.*

Proof. In Join Bool A the two points inr true and inr false play the role of north and south, with a path connecting them to every point inl a. □

The proof of the following result is taken from Brunerie [5, Proposition 1.8.6], as such we will only sketch it here.

Lemma 4.5. *Join is associative:*

$$\text{Join } A \text{ (Join } B\ C) \equiv \text{Join (Join } A\ B) \ C$$

Proof. One starts by applying the 3×3 lemma to the following diagram.

$$\begin{array}{ccccc}
A & \longleftarrow & A \times B & \longrightarrow & B \\
\uparrow & & \uparrow & & \uparrow \\
A \times C & \longleftarrow & A \times B \times C & \longrightarrow & B \times C \\
\downarrow & & \downarrow & & \downarrow \\
A \times C & \longleftarrow & A \times C & \longrightarrow & C
\end{array}$$

All of the arrows in the diagram are the obvious projections, and the homotopies are refl's. One can then show that A□○ ≡ Join (Join A B) C, and that A○□ ≡ Join A (Join B C), which implies the desired result. □

We now have all of the ingredients to relate S^3 and the Join of two circles.

Lemma 4.6. *Join S^1 S^1 ≡ S^3*

Proof. This is a composition of equalities we already proved.

```
Join S¹ S¹ ≡ Join (Susp Bool) S¹          (4.1)
           ≡ Join (Join Bool Bool) S¹     (4.4)
           ≡ Join Bool (Join Bool S¹)     (4.5)
           ≡ Susp (Susp S¹)               (4.4)
           ≡ S³                                  □
```

The above proof is an elegant application of the 3×3 lemma, but it results in rather complicated maps between Join S^1 S^1 and S^3. Evan Cavallo has found a more direct cubical proof that Join S^1 S^1 ≡ S^3 by simply defining the maps directly and proving that they cancel.[3] We have ported his proof to Cubical Agda and the resulting proof is very short (~ 60 LOC). However, the proofs that the maps cancel require a 5-dimensional(!) composition making it rather difficult to visualize.

5 The Hopf Fibration

In this section we define the Hopf fibration: Hopf : $S^2 \to$ Set. It is a fibration over the sphere whose fibers are equal to S^1 and whose total space is equal to S^3. It is a very useful construction that allows one to compute homotopical properties of S^3 from the properties of S^2 and S^1.

We begin by motivating the definition using some geometry while keeping in mind that we are actually working synthetically. Since the two hemispheres of S^2 are contractible, the fibration is trivial on each and the important datum is how to glue the fibers at the equator. Therefore, we have to pick an equivalence $S^1 \to S^1$ at every point of the equator. Since the equator is equal to S^1, it amounts to defining a map $S^1 \to S^1 \to S^1$. A natural choice is the binary product we get on S^1 when seeing it as the set of unitary complex numbers:

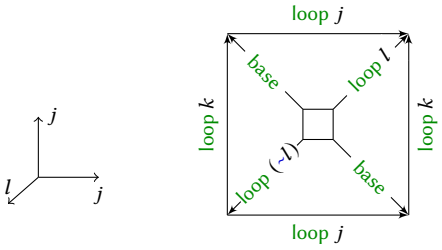
$$e^{2i\pi\theta} \cdot e^{2i\pi\varphi} = e^{2i\pi(\theta+\varphi)}$$

This is defined in Cubical Agda as follows:

```
rot : S¹ → S¹ → S¹
rot base y = y
rot (loop j) base = loop j
rot (loop j) (loop k) =
  hcomp (λ l → λ { (k = i0) → loop (j ∨ ~ l)
                 ; (k = i1) → loop (j ∧ l)
                 ; (j = i0) → loop (k ∨ ~ l)
                 ; (j = i1) → loop (k ∧ l)} ) base
```

The final case can be illustrated by the following diagram, where we only annotated the edges and reduced the (constant) central face to a tiny square:

[3] For details see the redtt proof at:
https://github.com/RedPRL/redtt/blob/master/library/cool/s3-to-join.red



Indeed, our intuition from the complex numbers tells us that rot (loop j) (loop k) is analogous to $e^{2i\pi(j+k)}$. So it is only natural that the $j + k = 1$ diagonal is constant at base, and the $j = k$ diagonal follows loop twice.

This results in a binary operation on S¹, that we often write as _*_ instead of rot. We can even get a complete (higher) abelian group structure on S¹, with an inverse operation that we call inv.

Lemma 5.1. *For every $x : S^1$ we have an equivalence rotEquiv $x : S^1 \simeq S^1$ given by rot x.*

Proof. The inverse is induced by path reversal. □

We now define the Hopf fibration using Susp S¹ (which is equivalent to S² by lemma 4.2) as our base space, using Glue to glue the fibers as discussed above. However, we have to depart slightly from the topological intuition as there is no actual equator in Susp S¹. As instead, we use rot to glue the fibers around the north pole, and the identity to glue them around the south pole:

```
Hopf : Susp S¹ → Set
Hopf north = S¹
Hopf south = S¹
Hopf (merid x i) = ua (rotEquiv x) i
```

In fact, we could have directly defined the Hopf fibration for S² by gluing on the identity equivalence on three of the sides of the 2-cell and the rot equivalence on the fourth side.

```
HopfS² : S² → Set
HopfS² base = S¹
HopfS² (surf i j) =
  Glue S¹ (λ { (i = i0) → (S¹ , idEquiv S¹)
             ; (i = i1) → (S¹ , idEquiv S¹)
             ; (j = i0) → (S¹ , idEquiv S¹)
             ; (j = i1) → (S¹ , rotEquiv (loop i)) } )
```

However, it turns out that the version using Susp S¹ is easier to work with as we have more wiggle room with the constructors.

We can now form the total space of the Hopf fibration using a Σ-type: Σ Hopf = $\sum_{x:\text{Susp } S^1}$ Hopf x. Our main theorem can be stated as:

Theorem 5.2. *The total space of the Hopf fibration is S^3, that is, Σ Hopf ≡ S^3.*

Proof. By lemma 4.6, which states that S³ ≡ Join S¹ S¹, it suffices to prove that Σ Hopf ≡ Join S¹ S¹. From the topological intuition of the Hopf fibration, the union of the fibers at the equator form a torus S¹ × S¹—the first coordinate being the longitude in S², the second parameterizing the fiber. When transporting along the meridian merid x, the fiber above north will transform into the fiber above the equator according to rot x, and the one above south will be transported to the equator according to the identity. Conversely, transporting fibers from the equator to the north (resp. south) pole will transform them according to rot (inv x) (resp. the identity). This informal observation relates Σ Hopf to the pushout of the following span:

$$S^1 \xleftarrow{\text{rot'}} S^1 \times S^1 \xrightarrow{\pi_2} S^1$$

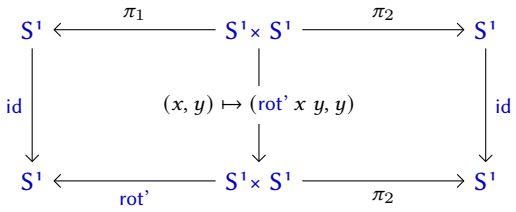
where rot' (x, y) = inv x * y.

But recall that Join S¹ S¹ is the pushout of the following span:

$$S^1 \xleftarrow{\pi_1} S^1 \times S^1 \xrightarrow{\pi_2} S^1$$

We will now describe a span isomorphism (that is, a natural equivalence between these two spans) and use it as intuition to build a formal isomorphism between the two types.

To go from Join S¹ S¹ to Σ Hopf, we want to mimic the following natural equivalence:

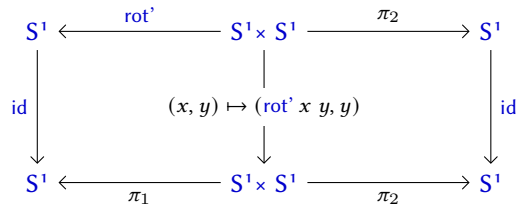


We can use this intuition to construct the following map:

```
j2h : Join S¹ S¹ → ΣHopf
j2h (inl x) = (north , x)
j2h (inr y) = (south , y)
j2h (push x y i) =
  let p : rot' (x , y) * x ≡ y
      p = lem-rot' x y
  in ( merid (rot' (x , y)) i
     , glue (λ { (i = i0) → x ; (i = i1) → y }) (p i))
```

Here, x is the point in the fiber above north, y is the point in the fiber above south, and $p\,i$ is the point in the fiber above merid (rot' (x, y)) i. However, as Hopf (merid (rot' (x, y)) i) is ua (rotEquiv (merid (rot' (x, y)))) i) which in turn is a Glue type we need to use the glue constructor to package it up into a well-typed term. The term lem-rot' x y proves that rot' (x , y) * $x \equiv y$, that is, inv x * y * $x \equiv y$. This equality is clearly true and we prove it by induction on x and y.

Conversely, to go from Σ Hopf to Join S¹ S¹, we use the following natural isomorphism:



To do this we need to map out of a Glue type. This is made possible through the unglue primitive in Cubical Agda. Given y : ua (rotEquiv x) i the term unglue ($i \vee \sim i$) y is an element of S¹ that is x * y when i is i0 and y when i is i1. Using this we can write the inverse map.

```
h2j : ΣHopf → Join S¹ S¹
h2j (north , y) = inl y
h2j (south , y) = inr y
h2j (merid x i , y) =
  hcomp (λ j → λ { (i = i0) → inl (lem-rot-inv x y j)
                 ; (i = i1) → inr y })
        (push (unglue (i ∨ ~ i) y * inv x)
              (unglue (i ∨ ~ i) y) i)
```

The lemma lem-rot-inv proves that x * y * inv $x \equiv y$ for all x, y : S¹. This hence defines the second map. Proving that they cancel requires some rather involved path algebra, and we refer the interested reader to the formalization. □

This hence concludes our direct cubical proof that Hopf is indeed the Hopf fibration.

6 Conclusions and Future Work

We have in this paper shown how some of the main results in synthetic homotopy theory can be formalized in cubical type theory. We have used a variation of cubical type theory implemented by the Cubical Agda system, however it would have been possible to formalize all of these examples with comparable complexity in other cubical systems. Indeed, many of these examples have also been formalized in the redtt system [26] and in the precursor of Cubical Agda called cubicaltt [11].

One might wonder to what extent the lack of reversals and connections in redtt, which is based on cartesian cubical type theory [2, 3], affects the length of proofs. In our experience the lack of this additional structure on the interval is often made up for by the more powerful composition operations of cartesian cubical type theory. For instance, the direct proof that Join S¹ S¹ ≡ S³ is of more or less exactly the same complexity as the Cubical Agda proof. However, in order to draw any definite conclusions more experiments are necessary.

The results in this paper have also been formalized in the various HoTT libraries available in the major proof assistants based on type theory: HoTT-Agda [7], Coq-HoTT [4], Lean-HoTT [29].[4] It is very difficult to make an accurate quantitative comparison of the complexity between the formalized results as they have been performed in different systems based on different type theories. However, it is interesting to note that all of these HoTT libraries contain cubical sublibraries for conveniently reasoning about squares and cubes inspired by the work of Licata and Brunerie [20]. In a cubical system like Cubical Agda we do not need to write such a library as the cubical primitives provide us with it for free.

Despite the difficulty of comparing the complexity of formal proofs between different systems we have made some estimates of the size of some of the proofs (in terms of lines of code) in table 1.[5] Some of the libraries contain multiple proofs of the relevant results and in such cases we picked the one that most closely resemble the cubical proof. We only include those examples where we can make reasonably accurate estimates of the proof size, but the numbers in the table should still be taken with a large grain of salt as they do not count self-contained proofs and many of the results rely on cubical sublibraries that are not necessary in Cubical Agda. The line count also involve relevant comments and we haven't counted the definitions of the involved HITs. The length and style of proofs also vary quite a bit between the various systems in general, for instance, both Coq and Lean proofs are written using tactics while Agda proofs are typically not.

Table 1. Results in the HoTT libraries

	Agda	Coq	Lean	Cubical
$\Omega(\mathbb{S}^1) = \mathbb{Z}$	90	160	80	50
$\mathbb{T} = \mathbb{S}^1 \times \mathbb{S}^1$	150	150	-	25
3×3 lemma	3000	-	-	200
Join assoc. via 3×3	320	-	-	240
Join assoc. direct	210	-	230	90

The table indicates that not too much is gained in the proof that the loop space of the circle is the integers by doing it cubically, while the proof that the torus is equivalent to two circles is about 6 times longer in HoTT-Agda and Coq-HoTT compared to the cubical proof presented here. The major difference is for the 3×3 lemma for pushouts where the HoTT-Agda proof is about 15 times longer than

[4] We omit the UniMath library as it does not focus on synthetic homotopy theory.

[5] The numbers in the table have been computed from the master branches of the following libraries on 2019-12-16:
https://github.com/HoTT/HoTT-Agda
https://github.com/HoTT/HoTT/
https://github.com/leanprover/lean2/tree/master/hott

the cubical proof. This result is not yet formalized in Coq-HoTT and Lean-HoTT, however a Coq-HoTT formalization is currently underway. The HoTT-Agda library has two proofs that Join is associative. The longer one, 320 lines, is more or less the same as the one we discussed in this paper and relies on the 3×3 lemma for pushouts. There is very little gain from the cubical machinery in this proof as it is simply a matter of reorganizing data so that we can apply an already proved result. The other HoTT-Agda and Lean-HoTT proofs are more direct and construct the maps in an ingenious way following Cavallo [9, Theorem 4.21]. Evan Cavallo has recently proved this result directly in Cubical Agda as well, leading to a proof in only 90 lines of code.

Another interesting observation is that when working in a cubical system we often state the theorems as paths while in HoTT one often instead just uses equivalences. These are of course equivalent by univalence, but by invoking the univalence axiom in HoTT one hides the computational content of these equivalences making them harder to work with. In a cubical system where univalence has computational content this is not the case and it is in fact often more convenient to convert the equivalences into paths using univalence as we may then use the cubical primitives to manipulate them. This indicates that cubical type theory might be better suited for doing *univalent* mathematics than HoTT.

A crucial property when doing synthetic mathematics is the existence of interesting models of the theory. Ideally we would like to be able to interpret all of the results in this paper in topological spaces or even any (Grothendieck) ∞-topos. Currently these questions have not been fully resolved for the various cubical type theories that have been considered. In fact, it has been shown that the standard model of Cubical Agda is *not* equivalent[6] to spaces [24]. However, if one drops the reversal operation (~_) from Cubical Agda any internal result about homotopy groups of spheres corresponds to a result about the homotopy groups of spheres in spaces.[7] Furthermore, there has been recent progress on an "equivariant" cubical set model that is equivalent to spaces [22]. We are hence very optimistic that these issues will be resolved in the near future. Furthermore, as soon as a satisfactory cubical type theory with a model in spaces has been developed we expect it to be straightforward to adapt the formalizations in this paper to that theory. Indeed, the main features that we rely on—computational univalence and higher inductive types with definitional computation rules for all constructors—should also be satisfied by that cubical type theory.

Future work Some results have so far been out of reach for conventional HoTT because of the complexity of the involved path algebra. An example of this is the symmetric monoidal structure of the smash product HIT [6, 28]. It might be possible to make progress on this using cubical type theory as the path algebra should be more manageable. This would be a very interesting direction for future work.[8]

Another interesting possibility offered by cubical type theory comes from the fact that the theory is constructive and hence satisfies the existence property. This means that we should be able to write down existence statements (expressed using Σ-types) and extract witnesses automatically. An example of this is the so called *"Brunerie number"*: a concrete synthetic definition of $n \in \mathbb{Z}$ such that $\pi_4(\mathbb{S}^3) = \mathbb{Z}/n\mathbb{Z}$ [5]. We have formalized this construction in Cubical Agda, but have so far not been able to compute this numeral due to the computational complexity of the involved constructions. However, it may be more feasible to use Cubical Agda for computing other simpler topological invariants like cohomology groups [8, 9].

Acknowledgments

The second author is grateful to Steve Awodey for hosting a visit to Carnegie Mellon University during the 2018/2019 academic year where the work leading to this paper was started. The authors are also very grateful to Evan Cavallo for his many ingenious cubical proofs and impressive higher dimensional compositions.

This material is based upon work supported by the Swedish Research Council (Vetenskapsrådet) under Grant No. 2016-06828 and Grant No. 2019-04545.

References

[1] Agda development team. 2019. *Agda 2.6.0.1 documentation.* https://agda.readthedocs.io/en/v2.6.0.1/

[2] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Kuen-Bang Hou (Favonia), Robert Harper, and Daniel R. Licata. 2019. Syntax and Models of Cartesian Cubical Type Theory. (February 2019). https://github.com/dlicata335/cart-cube Preprint.

[3] Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. 2018. Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. In *27th EACSL Annual Conference on Computer Science Logic (CSL 2018) (Leibniz International Proceedings in Informatics (LIPIcs))*, Dan Ghica and Achim Jung (Eds.), Vol. 119. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 6:1–6:17. https://doi.org/10.4230/LIPIcs.CSL.2018.6

[4] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. 2017. The HoTT Library: A Formalization of Homotopy Type Theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017)*. ACM, New York, NY, USA, 164–172. https://doi.org/10.1145/3018610.3018615

[5] Guillaume Brunerie. 2016. *On the homotopy groups of spheres in homotopy type theory.* Ph.D. Dissertation. Université de Nice.

[6] By "equivalent" we mean that the notion of fibration in the cubical set model gives rise to a model structure that is Quillen equivalent to the Quillen model structure on spaces.

[7] For further details and discussions about this result see: https://groups.google.com/forum/#!topic/homotopytypetheory/imPb56IqxOI

[8] Some preliminary work in redtt can be found at: https://github.com/RedPRL/redtt/blob/master/library/pointed/smash.red

[6] Guillaume Brunerie. 2018. Computer-generated proofs for the monoidal structure of the smash product. (Nov. 2018). https://www.uwo.ca/math/faculty/kapulkin/seminars/hottest.html *Homotopy Type Theory Electronic Seminar Talks.*

[7] Guillaume Brunerie, Kuen-Bang Hou (Favonia), Evan Cavallo, Tim Baumann, Eric Finster, Jesper Cockx, Christian Sattler, Chris Jeris, Michael Shulman, et al. [n. d.]. Homotopy Type Theory in Agda. https://github.com/HoTT/HoTT-Agda

[8] Ulrik Buchholtz and Kuen-Bang Hou Favonia. 2018. Cellular Cohomology in Homotopy Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*. ACM, New York, NY, USA, 521–529. https://doi.org/10.1145/3209108.3209188

[9] Evan Cavallo. 2015. *Synthetic cohomology in homotopy type theory.* Master's thesis. Carnegie Mellon University.

[10] Evan Cavallo and Robert Harper. 2019. Higher Inductive Types in Cubical Computational Type Theory. *Proc. ACM Program. Lang.* 3, POPL, Article 1 (Jan. 2019), 27 pages. https://doi.org/10.1145/3290314

[11] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2015. Cubicaltt. (2015). https://github.com/mortberg/cubicaltt.

[12] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In *Types for Proofs and Programs (TYPES 2015) (LIPIcs)*, Vol. 69. 5:1–5:34.

[13] Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. On Higher Inductive Types in Cubical Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*. ACM, New York, NY, USA, 255–264. https://doi.org/10.1145/3209108.3209197

[14] Kuen-Bang Hou (Favonia), Eric Finster, Daniel R. Licata, and Peter LeFanu Lumsdaine. 2016. A Mechanization of the Blakers-Massey Connectivity Theorem in Homotopy Type Theory. In *31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16*. ACM, 565–574.

[15] Kuen-Bang Hou (Favonia) and Michael Shulman. 2016. The Seifert-van Kampen Theorem in Homotopy Type Theory. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Jean-Marc Talbot and Laurent Regnier (Eds.), Vol. 62. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:16. https://doi.org/10.4230/LIPIcs.CSL.2016.22

[16] Simon Huber. 2019. Canonicity for Cubical Type Theory. *Journal Automated Reasoning* 63, 2 (Aug. 2019), 173–210. https://doi.org/10.1007/s10817-018-9469-1

[17] Chris Kapulkin and Peter LeFanu Lumsdaine. 2012. The Simplicial Model of Univalent Foundations (after Voevodsky). (Nov. 2012). Preprint arXiv:1211.2851v4.

[18] Daniel Licata and Michael Shulman. 2013. Calculating the Fundamental Group of the Circle in Homotopy Type Theory. *Proceedings - Symposium on Logic in Computer Science*, 223–232. https://doi.org/10.1109/LICS.2013.28

[19] Daniel R. Licata and Guillaume Brunerie. 2013. $\pi_n(S^n)$ in Homotopy Type Theory. In *Certified Programs and Proofs*, Georges Gonthier and Michael Norrish (Eds.). Springer International Publishing, 1–16.

[20] Daniel R. Licata and Guillaume Brunerie. 2015. A Cubical Approach to Synthetic Homotopy Theory. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS'15*. ACM, 92–103.

[21] Per Martin-Löf. 1975. An Intiutionistic Theory of Types: Predicative Part. In *Logic Colloquium '73*, H. E. Rose and J. Shepherdson (Eds.). North–Holland, Amsterdam, 73–118.

[22] Emily Riehl. 2019. The equivariant uniform Kan fibration model of cubical homotopy type theory. (Aug. 2019). https://hott.github.io/HoTT-2019//programme/#riehl Talk given at *The International Conference on Homotopy Type Theory (HoTT 2019)* at Carnegie Mellon University.

[23] Egbert Rijke. 2017. The join construction. (2017). Preprint arXiv:1701.07538v1.

[24] Christian Sattler. 2018. Do cubical models of type theory also model homotopy types? (2018). https://www.him.uni-bonn.de/programs/past-programs/past-trimester-programs/types-sets-constructions/workshop-types-homotopy-type-theory-and-verification/schedule/#c13292 Talk given at *Types, Homotopy Type theory, and Verification* at the Hausdorff Center for Mathematics in Bonn.

[25] Kristina Sojakova. 2016. The Equivalence of the Torus and the Product of Two Circles in Homotopy Type Theory. *ACM Transactions on Computational Logic* 17, 4 (Nov. 2016), 29:1–29:19.

[26] The RedPRL Development Team. 2018. The redtt Proof Assistant. https://github.com/RedPRL/redtt/

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

[28] Floris van Doorn. 2018. *On the Formalization of Higher Inductive Types and Synthetic Homotopy Theory.* Ph.D. Dissertation. Carnegie Mellon University.

[29] Floris van Doorn, Jakob von Raumer, and Ulrik Buchholtz. 2017. Homotopy Type Theory in Lean. In *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings (Lecture Notes in Computer Science)*, Mauricio Ayala-Rincón and César A. Muñoz (Eds.), Vol. 10499. Springer, 479–495. https://doi.org/10.1007/978-3-319-66107-0_30

[30] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 87 (July 2019), 29 pages. https://doi.org/10.1145/3341691

[31] Vladimir Voevodsky. 2014. The equivalence axiom and univalent models of type theory. (Talk at CMU on February 4, 2010). (2014). Preprint arXiv:1402.5556.

[32] Vladimir Voevodsky. 2015. An experimental library of formalized Mathematics based on the univalent foundations. *Mathematical Structures in Computer Science* 25 (2015), 1278–1294.

[33] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. [n. d.]. UniMath: Univalent Mathematics. Available at https://github.com/UniMath.