The Proof of the Fundamental Group of the Circle in Homotopy Type Theory's Dependence on the Univalence Axiom

A.C. Linssen
Supervisors: B. P. Ahrens, K. F. Wullaert
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

The incorporation of the univalence axiom into homotopy type theory has facilitated a new way of proving a basic result in algebraic topology: that the fundamental group of the circle is the integers ($\pi_1(\mathbb{S}^1) \simeq \mathbb{Z}$). This proof is formalised by Licata and Shulman [1]. However, while the authors note that the univalence axiom is essential, it is not clearly stated where and why. A step-by-step analysis of their proof, the purpose of which is to increase the readers' understanding of the univalence axiom and its implications in homotopy type theory, shows that it relies on the univalence axiom for constructing the circle. When assuming axiom K instead of the univalence axiom, we can no longer construct the circle in the same way, leading to $\pi_1(\mathbb{S}^1) \simeq 1$.

1 Introduction

The fusing of concepts from mathematics into computer science and vice versa has led to new fields of study like type theory. Type theory is a formal system that blends mathematical logic with computer science's type systems [2, 3]. The concept of type theory has been added to and adapted, resulting in different versions such as Church's λ-calculus [4, 5] and Martin-Löf's constructive type theory, also known as intuitionistic type theory [6].

Similar to type theory, constructive type theory has been used and modified many times [7]. Its dependent types form the foundation for proof assistants [8] like Agda [9] and Coq [10]. It likewise has been fundamental, along with category theory and homotopy theory, in creating homotopy type theory (HoTT) [11]. In HoTT, types can be interpreted as either spaces in homotopy theory or higher-dimensional groupoids in category theory.

A contribution to HoTT that must be noted is that of Voevodsky: the univalence axiom [12, 13, 14]. In short, the univalence axiom is a way to enforce a universe $\mathcal{U}$ of types where each type has a function that maps equivalence to equality, that is $(A \simeq B) \to (A =_{\mathcal{U}} B)$. This map from equivalence to equality is itself an equivalence, namely $(A \simeq B) \simeq (A =_{\mathcal{U}} B)$.

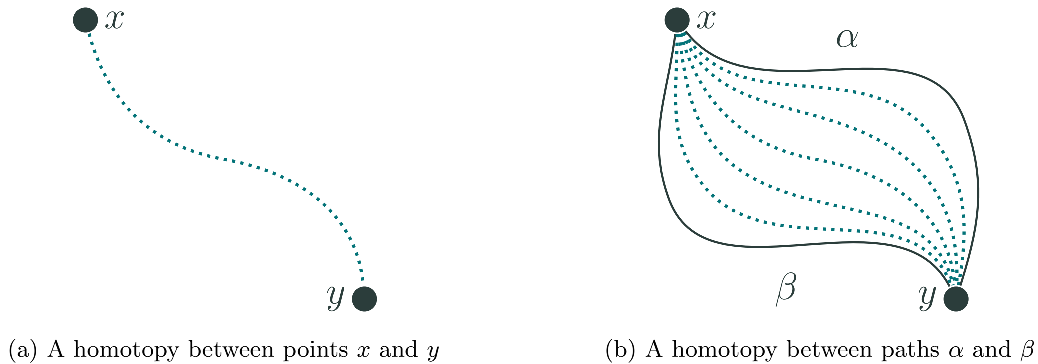
These developments have come together to form a new set of tools for mathematicians and computer scientists to use. One of the simpler examples that combines proof assistants, HoTT and the univalence axiom is the calculation of an algebraic invariant: the fundamental group of the circle. The proof, formalised by Licata and Shulman, takes a more straightforward approach compared to classical proof [15]. Furthermore, the authors note that the univalence axiom "plays an essential role" [1, p. 1].

As exciting as this new set of tools is, it can be challenging to see how the different concepts play into one another, especially for novices. Hence, to deepen the understanding of homotopy type theory and especially the univalence axiom and its implications, this paper analyses how the proof given by Licata and Shulman depends on the univalence axiom. It divides the proof into smaller components and checks for each how it relates to the univalence axiom and what the consequences of omitting it are.

While this paper does touch on some basics of homotopy type theory and the univalence axiom, it is by no means a complete guide to these topics. Instead, the short explanations ensure clarity in this intersection of disciplines. For a more in-depth introduction and explanation of these concepts, we advise starting with the book "Homotopy Type Theory" [11]. The rest of this introduction gives a brief overview of tokens and types, homotopies, fundamental groups and the homotopy lifting property.

The fundamental building blocks of type theory, and hence of HoTT, are tokens and types. Throughout this paper they will be indicated as `a : A`, meaning `a` is a token of type `A`. There are many different interpretations for what tokens and types represent. They can be interpreted as *points* in *spaces* in homotopy theory or *objects* in *categories* in category theory. Furthermore, they can be seen as *proofs* for *predicates* according to the Curry-Howard isomorphism[16].

In homotopy theory, and therefore in HoTT, paths and points in topological space can be morphed by continuous maps. These continuous maps are called homotopies. A homotopy is an equivalence relation respecting reflexivity (constant path), symmetry (inversion of paths) and transitivity (concatenation of paths). Examples of a homotopy between points and between paths (also known as a homotopy between homotopies) are shown in figures 1a and 1b, respectively. There can be homotopies between homotopies between homotopies (and so forth), but this is outside the scope of this paper.



(a) A homotopy between points x and y

(b) A homotopy between paths α and β

Figure 1: Two examples of homotopies

Fundamental groups are a basic algebraic invariant of topological spaces. They describe the sets of equivalence classes of homotopic paths starting and ending at point *base*. Fundamental groups are of interest because homotopy equivalent spaces have isomorphic fundamental groups[11]. The corollary is that if the fundamental group is different, the spaces are not equivalent up to homotopy.

Path lifting, more formally known as the homotopy lifting property, is similar to homotopies in that it is a continuous function from one thing to another. The difference is that instead of mapping within a space, it maps between spaces: from a topological space E to a different one B. E, the *total space*, is commonly referred to as being 'above' B, the *base space*. The function p maps E to B, as seen in figure 2. For our purposes, the most important characteristic is that the path lifting maps any homotopy $\tilde{f}$ in E, to a homotopy f in B.

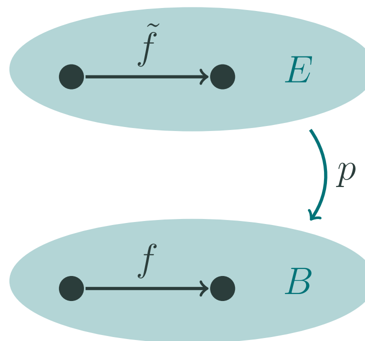


Figure 2: Space E above B, where p maps E to B and $\tilde{f}$ in E to f in B.

The remaining part of the paper has been divided into four parts. The first part deals with the univalence axiom. The second part deconstructs Licata and Shulman's proof into components, clarifying each one. The third part then analyses the previously defined components' dependence on the univalence axiom. And finally, the fourth part gives a brief discussion on responsible research in mathematics and concludes the paper.

2 The Univalence Axiom

As noted in the introduction, the univalence axiom is a way to enforce a universe $\mathcal{U}$ of types where each type has a function that maps equivalence to equality. To properly analyse how the proof of the fundamental group of the circle depends on Voevodsky's addition in section 4, this section presents the univalence axiom in more detail. It does so by first refining the notions of equivalence and equality in HoTT. And second, by briefly looking at what we may assume about the universe $\mathcal{U}$ in its absence.

2.1 Equivalence

The intuitive way to interpret equivalence is "equal for all intents and purposes". Two objects may not be identical, but their relevant, whatever that might mean in a given context, properties are. Throughout this paper, the context for equivalences is indicated by phrases such as "equivalent up to homotopy" or "homotopy equivalent".

While intuitive interpretations are valuable, they are of little use in formal proofs. Hence, we now define equivalence in the context of homotopy: two spaces X and Y are homotopy equivalent if there is a morphism f that (1) maps X to Y and (2) has a homotopy inverse $g : Y \to X$. From the definition of homotopy inverse, we also have $g \circ f \sim id_X$ and $f \circ g \sim id_Y$ [11].

Equivalences also occur in the context of the univalence axiom itself. As noted in the introduction, the mapping from equivalence to equality is itself an equivalence. To see why we consider that equivalences can map one thing to the other and vice versa. Going from equivalence to equality is possible due to the univalence axiom. The reverse while not explicitly stated, is also possible; equal things are also "equal for all intents and purposes" or equivalent. Consequently, the univalence axiom gives a mapping and its inverse which together can be considered an equivalence.

2.2 Equality

To understand the impact of the univalence axiom on the notion of equality, we first characterise equality in basic HoTT, that is HoTT without any additional axioms. Equality in (basic) homotopy type theory is denoted by the identity type $Id_A(a, b)$ or $a =_A b$ where $a, b : A$. This type, according to the Curry-Howard correspondence [16], represents the proposition that a and b are equal. This proposition can be made for any a and b, in contrast to the token $Id_A(a, b)$, which represents the proof that a and b are equal [17]. While this is simple enough, the only guaranteed identifications are self-identifications [17]. Self-identifications state nothing more than "tokens are equal to themselves", which is known as reflexivity. At first blush, this seems ineffective, but it does facilitate path induction which we will discuss in section 3.

To add the univalence axiom, then, is to add another token constructor that takes a token of $a \simeq b$ and produces $a =_A b$. In effect, the univalence axiom expands the notion of equality to include equivalence [18]. This is precisely what makes Voevodsky's addition so powerful.

2.3 Alternative Universe

In the absence of the univalence axiom, we are left with the more narrow definition of equality. More specifically, there is only one constructor for identity types. Given there is only one constructor, it is consistent to say there is only one way to associate something with itself and thus that each identity type is inhabited by only one token. This is also referred to as uniqueness of identity proofs (UIP). Axiom K is logically equivalent to UIP and it states that all tokens of identity types are reflexivity (trivial self-identifications)[11].

3 Components of the Proof

To systematically examine the proof of the fundamental group of the circle given by Licata and Shulman [1], this section divides it into smaller components. This both increases the understanding and makes the investigation into each component's dependence on the univalence axiom in section 4 more straightforward.

The components are grouped into two main categories: the preliminaries and the actual proof. The general concepts and operations are introduced and defined in the preliminaries. In addition, this section contains the accompanying proof that the given operations adhere to specific mathematical laws. The components in the second category build on the preliminaries and deliver the proof that the fundamental group of the circle is the integers.

Each component is described from multiple angles depending, like its contribution to the proof as a whole, the mathematical reasoning, the intuition behind it and the Agda code, depending on what is appropriate. The contribution to the proof as a whole is meant to keep the reader informed about the goal of that component. The mathematical reasoning describes the formulaic version[1], while the intuition behind the component serves to inform those less mathematically inclined. Lastly, the Agda code is the official proof.

3.1 Preliminaries

The preliminaries contain all the building blocks for the proof. It creates the context for our universe of types and specifies which operations are possible. Furthermore, it already gives some small proofs which will be used later on.

Types As Spaces

We start from the bottom: by giving the types meaning. Since we are interested in algebraic topology, we choose the homotopy interpretation of HoTT. That is, interpreting types as spaces and tokens as points in those spaces.

The Identity Type
As mentioned in section 2 the identity type in HoTT denotes equality, where the type is a proposition of equality and the token a proof of equality. The authors have chosen to represent the identity type between elements `M, N : A` as `Path M N` to "emphasize the homotopy theoretic interpretation" [1, p. 2]. Through this lens, the identity type `Path M N` symbolises the path space which consists of all paths from `M` to `N` [17]. Its only constructor `id` is for the identity path, a trivial self-identification. A key observation is that while it can only construct a token representing the identity path, the type also represents non-trivial self-identifications, which we'll see later when discussing the circle.

Equivalent Points
Directly after defining the identity type, they add the notion of equivalence ($\simeq$) which under the hood is also represented by `Path`. This, in essence, adds another constructor for `Path` which, unlike `id`, does not require having the same start- and end-point. It does, however, require the points to have the same type (`A`).

Path Induction
A crucial part of the homotopy interpretation is the elimination rule (induction principle) for identity types: path induction [11]. It is based on the principle of *substitution salva veritate*, meaning if `p` and `q` are identical then "one can be substituted for the another in any statement while *saving the*

[1] In general Π can be interpreted as $\forall$ and Σ as $\exists$ for those more familiar with set theory.

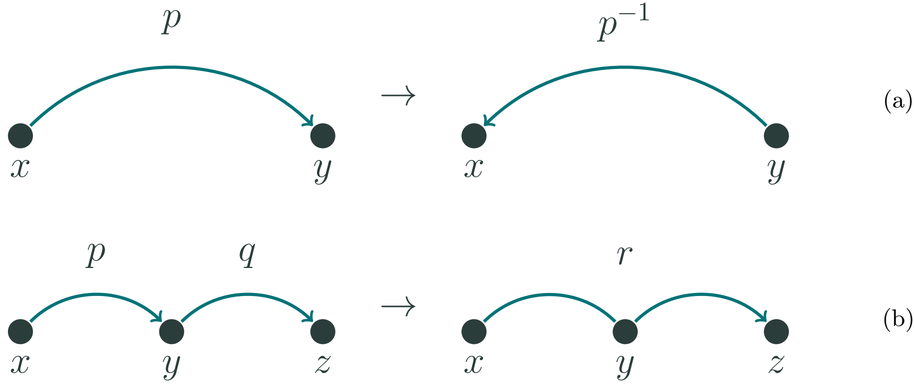


Figure 3: Operations on paths: (a) path inversion and (b) path concatenation

truth of that statement" [17, p. 20]. Hence, to prove a statement by induction it is enough to prove the base case where the points are the same and the paths are the constant path.

The paths in based path induction, with a fixed starting point and a variable endpoint, are inductively generated by the starting point and the constant path. As a function:

$$\Pi_{a:A}\Pi_{C:\Pi_{x:A}(a=_A x)\to\mathcal{U})}C(a,(a =_A a)) \to \Pi_{x:A}\Pi_{p:a=_A x}C(x,p) \tag{1}$$

While, at first blush, this makes it seem like the only path is the constant path, this is not the case[11]. The identity type `x =A y`, as `y` varies over `A`, is the based path space. And for any point in that space is homotopic to the constant path at some point [17]. This is a result of being able to move the free endpoint along the given path until it is the same as the starting point and the resulting path is the constant path.

Inversion of Paths
Next, the authors define operations on paths starting with inversion. Intuitively, if for two points `x, y : A` there is a path `p` from `x` to `y`, then following that path in reverse gives you a path from `y` to `x` (Figure 3a). Hence, for every path `p` there is an inverse `p-1` or `!p`. Mathematically speaking, it is:

$$\Pi_{(A:\mathcal{U})}\Pi_{x,y:A}(x =_A y) \to (y =_A x) \tag{2}$$

By path induction, it suffices to assume `x` is y and hence the inverse of `Path x x` is `Path x x = id`.

Composition of Paths
Another operation on paths is path composition or concatenation, indicated by $\circ$. In the code this operation is right-associative; `f ∘ g` is read as "f after g" and g is applied first, then f. Since in mathematics the convention is that concatenation is left-associative, we will distinguish the two by using $\centerdot$ for the left-associative version and $\circ$ for the right-associative variant. If for three points `x, y, z : A` there is a path `p` from `x` to `y` and a path `q` from `y` to `z`, then first following `p` and then following `q` gives a new path `r` from `x` to `z` (Figure 3b). For those more mathematically inclined:

$$\Pi_{(A:\mathcal{U})}\Pi_{x,y,z:A}(x =_A y) \centerdot (y =_A z) \to (x =_A z) \tag{3}$$

Again by induction, it suffices to assume `x` is `y` and `y` is `z` and hence `p, q` are `Path x x = id`, making `r Path x x = id`.

Types as Spaces

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

3  _≃_ : {A : Type} → A → A → Type
4  _≃_ = Path

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

9  ! : {A : Type} {M N : A} → Path M N → Path N M
10 ! id = id

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

Groupoid Laws

Having defined path inversion (`!`) and composition (∘), it must be proven that these methods satisfy the groupoid laws up to homotopy: the unit, associativity and inverse law. The laws are described in more detail below, but their proofs all follow the same pattern: path induction.

Unit Law
The unit law states that composing a path `p` with `id` results in `p`. It must be proven in both directions, i.e. `p ▪ id = p` and `id ▪ p = p`. More formally, this is respectively:

$$\Pi_{(A:\mathcal{U})}\Pi_{x,y:A}(x =_A y) \centerdot (y =_A y) \to (x =_A y) \tag{4}$$

$$\Pi_{(A:\mathcal{U})}\Pi_{x,y:A}(x =_A x) \centerdot (x =_A y) \to (x =_A y) \tag{5}$$

Associativity Law
The associativity law states that for paths `p, q, r` the order in which they are composed does not matter: `p ▪ (q ▪ r) = (p ▪ q) ▪ r`. Translated to formal mathematics results in the following:

$$\Pi_{(A:\mathcal{U})}\Pi_{w,x,y,z:A}(w =_A x) \centerdot ((x =_A y) \centerdot (y =_A z)) \to ((w =_A x) \centerdot (x =_A y)) \centerdot (y =_A z) \tag{6}$$

Inverse Law
The inverse law states that composing a path `p` with its inverse `p`$^{-1}$ is `id`, which like the unit law must be proven in both directions:

$$\Pi_{(A:\mathcal{U})}\Pi_{x,y:A}(x =_A y)^{-1} \centerdot (x =_A y) \to (y =_A y) \tag{7}$$

$$\Pi_{(A:\mathcal{U})}\Pi_{x,y:A}(x =_A y) \centerdot (x =_A y)^{-1} \to (x =_A x) \tag{8}$$

Groupoid Laws

```
13 ∘-unit-l  : {A : Type} {M N : A} (α : Path M N)
14           → Path (id ∘ α) α
15 ∘-unit-l id = id
```

```
16  ∘-unit-r  : {A : Type} {M N : A} (α : Path M N)
17            → Path (α ∘ id) α
18  ∘-unit-r id = id

19  ∘-assoc  : {A : Type} {M N P Q : A}
20           (γ : Path P Q) (β : Path N P) (α : Path M N)
21           → Path (γ ∘ (β ∘ α)) ((γ ∘ β) ∘ α)
22  ∘-assoc id id id = id

23  !-inv-l  : {A : Type} {M N : A} (α : Path M N)
24           → Path (! α ∘ α) id
25  !-inv-l id = id

26  !-inv-r : {A : Type} {M N : A} (α : Path M N) → Path (α ∘ (! α)) id
27  !-inv-r id = id
```

Type Families As Dependent Spaces

Again in correspondence with the homotopy interpretation of HoTT, we give meaning to the concept of type families. Type families are types indexed over some other type. In other words, they are dependent spaces.

Transport
Recall from the introduction the homotopy lifting property that maps between spaces (see also Figure 2). `transport` is the method that given mapping from `B` to its dependent space `E` and a path between two points in `B` produces a function that takes as input the starting point in `E` and outputs the endpoint in `E`.

Properties of Transport
Transport has two properties that will prove important later on, which are dependent on what you are transporting. The first, `transport-Path-right`, is applicable when transporting a family of paths: paths with the same start point and a variable endpoint. It can be converted to post-composition of the paths given as the second and third arguments. The second is applicable when transporting functions. Here the function can be broken up as shown in `transport-→`.

--- Type Families As Dependent Spaces ---

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

31  transport-Path-right :  {A : Type} {M N P : A}
32    (α' : Path N P) (α : Path M N)
33    → Path (transport (\ x → Path M x) α' α) (α' ∘ α)
34  transport-Path-right id id = id

35  transport-→ :  {Γ : Type} (A B : Γ → Type) {θ1 θ2 : Γ}
36                 (δ : θ1 ≃ θ2) (f : A θ1 → B θ1)
37    → Path  (transport (\ γ → (A γ) → B γ) δ f)
38            (transport B δ o f o (transport A (! δ)))
39  transport-→ _ _ id f = id
```

Functions Are Functorial

The previous sections have defined actions on points, but functors in HoTT can act on all levels: points, paths, paths between paths, and so forth. Hence, we now move on to these higher-level applications.

Applying Functions to Paths
Applying functions to `Path`s must be done in a functorial matter, that is "propagated" to the points of the path (which could themselves be paths). More concretely applying `f: A → B` to a path `Path N M` based in `A`, produces `Path fN fM` based in `B`. The method `ap` does precisely that.

Dependent Function Application
Where `ap` can be used for mapping from one space to another, `apd` can be used for mapping from one space to another *dependent* space. Recall figure 2 which shows path lifting, where the space `E` is dependent on `B`. The function used to morph the path `f` based in `B` into $\tilde{f}$ in the dependent space `E` is `apd` p^{-1} `f`.

Function are Functorial

```
40 ap :  {A B : Type} {M N : A}
41       (f : A → B) → Path{A} M N → Path{B} (f M) (f N)
42 ap f id = id

43 apd  :  {B : Type} {E : B → Type} {b₁ b₂ : B}
44         (f : (x : B) → E x) (β : Path b₁ b₂)
45      → Path (transport E β (f b₁)) (f b₂)
46 apd f id = id
```

Paths Between More Than Points

We can create paths between a variety of things. In fact, given a path between functions, we can create one between those functions applied to an argument. This is precisely what `ap≃` does. $\lambda\simeq$, or function extensionality, does the opposite; it takes the argument and a path between the functions applied to that argument, and reconstructs the path between the unapplied functions.

Paths Between More Than Points

```
47 ap≃ : ∀ {A} {B : A → Type} {f g : (x : A) → B x}
48       → Path f g → {x : A} → Path (f x) (g x)
49 ap≃ α {x} = ap (\ f → f x) α

50 postulate
51   λ≃  : ∀ {A} {B : A → Type} {f g : (x : A) → B x}
52       → ((x : A) → Path (f x) (g x))
53       → Path f g
```

Equivalences

We now turn to homotopy equivalences and the univalence axiom. Though this is the proof's first explicit mention of the univalence axiom, we'll refrain from scrutinising it until section 4. Instead,

we simply observe that the authors have chosen to specify only the consequences of the univalence axiom that are needed and discuss how those have been implemented.

Homotopy Equivalences
As explained in section 2, a homotopy equivalence requires two inverses, which by definition return the identity types after composition. From this, it follows that the implementation requires the two functions `f` and `g`, a proof that `g∘f x` $\simeq$ $\mathtt{id}_{\mathtt{x}}$, which is referred to as α in the code, and a proof that `f∘g y` $\simeq$ $\mathtt{id}_{\mathtt{y}}$, β in the code. Since $\simeq$ is represented by `Path`, we get the definition of `HEquiv`.

Of course, if there is a homotopy equivalence between `A` and `B`, there is also a homotopy equivalence between `B` and `A`. To make constructing this "inverse equivalence" easier, the authors have added `!-equiv` which simply reorders the arguments to construct an `HEquiv` between `B` and `A` from one between `A` and `B`.

Univalence
Next is a method which by now should look somewhat familiar: `univalence`. It maps an equivalence (`HEquiv`) to equality (`Path`). The keyword `postulate` just before it announces that it is a declaration of a type without an accompanying definition. The two needed consequences are expressed in `transport-univ` and `!-univalence`. The first stipulates that when transport is applied to a family of identity types on an application of univalence, it applies the forward direction of the equivalence. The second states that inverting an equivalence and then transforming it into an equality is the same as first transforming it to an equality and then inverting it.

Equivalences

```
54 record HEquiv (A B : Type) : Type where
55   constructor hequiv
56   field
57   f : A → B
58   g : B → A
59   α : (x : A) → Path (g (f x)) x
60   β : (y : B) → Path (f (g y)) y

61 !-equiv : ∀ {A B} → HEquiv A B → HEquiv B A
62 !-equiv (hequiv f g α β) = hequiv g f β α

63 postulate

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

65  transport-univ : {A B : Type} (e : HEquiv A B)
66   → Path  (transport (\ (A : Type) → A) (univalence e))
67           (HEquiv.f e)

68  !-univalence : {A B : Type} (e : HEquiv A B)
69               → Path  (! (univalence e))
70                       (univalence (!-equiv e))
```

The Integers

As the notion of integers is not native to Agda, we need to define it ourselves. We first create a data structure which uses recursion. Next, we then define the operations which can be performed on the integers. Last, we lay some groundwork for the proof by proving the defined operations are each other's inverses.

Data Structures
The data structure is based on `Positive`, which corresponds to the natural numbers. It is simple enough with a constructor `One` for the base case and `S` for recursively generating successors. This is then used as the basis for the integers. `Int` is like a wrapper around `Positive` to denote the sign (and include the notion of zero).

Operations on Integers
The only two operations on `Int` are `succ` and `pred` to produce the successor and predecessor, respectively. Like earlier with the groupoid laws, we have to prove correctness of the methods `succ` and `pred`. That is, we have to prove they are each others' inverse in order to say `succ (pred n) ≃ n` and `pred (succ n) ≃ n`. Then, we can construct an `HEquiv`.

The Integers

```
71 data Positive : Type where
72   One : Positive
73   S   : (n : Positive) → Positive

74 data Int : Type where
75   Pos  : (n : Positive) → Int
76   Zero : Int
77   Neg  : (n : Positive) → Int

78 succ : Int → Int
79 pred : Int → Int

80 pred-succ : (n : Int) → Path (pred (succ n)) n
81 pred-succ (Pos y) = id
82 pred-succ (Zero) = id
83 pred-succ (Neg One) = id
84 pred-succ (Neg (S y)) = id

85 succ-pred : (n : Int) → Path (succ (pred n)) n
86 succ-pred (Pos One) = id
87 succ-pred (Pos (S y)) = id
88 succ-pred (Zero) = id
89 succ-pred (Neg y) = id

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

The Circle

We can not reason about the fundamental group of the circle if the concept of the circle is not defined. We use the mathematical notion of a circle, S^1, where the space consists of only the border. This is not to be confused with a disk which is a plane bounded by a circle.

Data Structure
The circle is a higher-dimensional inductive type, meaning that it not only has generators for points but also for paths. In fact, it has one for each: `base` is a point and `loop` is a path (see Figure 4. The key thing to note about `loop` is that it is a *non-trivial* path from `base` to `base`. Furthermore, in combination with the path operations described earlier, it can be used to form additional paths. Examples of additional paths are `!loop` and `loop ∘ loop` but there are many more. For reasons

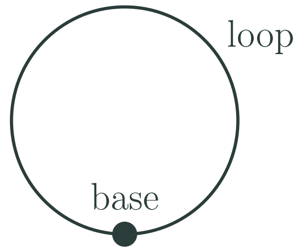


Figure 4: The circle as a higher inductive type with its two generators

we'll discuss later we consider `loop` to go around the circle counterclockwise and `!loop` to go around clockwise.

While `loop` itself is not equivalent to `id`, that does not mean that no path on the circle is equivalent to it. More specifically, the concatenation `loop` with its inverse is equivalent to `id`. The deformation from `loop ∘ !loop` to `id` is visualised in Figure 5.

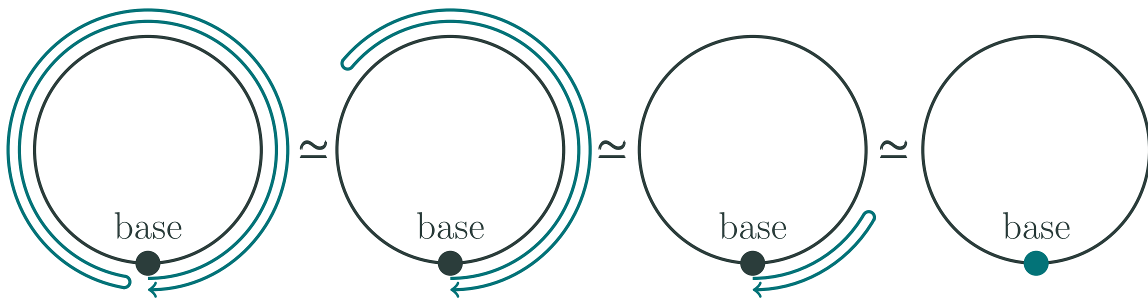


Figure 5: Morphing two concatenated inverse paths to the identity path

Circle Recursion
Where identity types have (based) path induction, the circle has circle induction. An outline of its elimination rule is "to map from the circle into any other type, it suffices to find a point and a loop in that type"[1, p. 5]. Indeed, we can see that `S¹-recursion` takes a "surrogate" base (`base'`) and "surrogate" loop (`Path base' base'`) as well as a circle space. How `S¹-recursion` can be applied is specified by the β-reduction rules. Its application on points is straightforward but requires more steps for paths; to ensure the types check, we use `ap` from the preliminaries.

Circle Induction
To be able to use the principle of induction on the circle, we must define dependent elimination. In short, a property needs to (1) hold for `base` and (2) be preserved going around the loop. Its β-elimination rules are analogous to those of circle recursion, the only difference being that `ap` is replaced with `apd` to allow *dependent* function application.

The Circle

```
92  data S¹' : Set where
93      Base : S¹'

94      S¹ : Set
95      S¹ = S¹'

96      base : S¹
97      base = Base
```

```
98      postulate
99        loop : Path base base

100     S¹-recursion : {C : Set}
101            -> (c : C)
102            -> (α : c ≃ c)
103            -> S¹ -> C
104     S¹-recursion a _ Base = a

105     S¹-induction :  (C : S¹ -> Set)
106             -> (c : C base)
107                (α : Path (transport C loop c) c)
108             -> (x : S¹) -> C x
109     S¹-induction _ x _ Base = x

110     postulate
111       βloop/rec : {C : Set}
112            -> (c : C)
113            -> (α : Path c c)
114            -> Path (ap (S¹-recursion c α) loop) α

115       βloop/elim : {C : S¹ -> Set}
116                  -> (c : C base) (α : Path (transport C loop c) c)
117                  -> Path (apd (S¹-induction C c α) loop) α
```

The Universal Cover

To gather some intuition behind how paths on the circle relate to the integers, we look at its universal cover, which is the helix. In short, the helix is a dependent space above the circle created by path lifting (see Figure 6). In other words, we can map paths on the circle to paths on the helix. These paths on the helix can then be translated into integers.

The intuition behind this mapping is as follows: we map `base` to the point marked `0` on the helix. Next, we map `loop`, which goes counterclockwise, to going around the helix counterclockwise one rotation. This has the effect of moving up one "level". In the same vein, we map `!loop` to going around the helix clockwise, which moves down one "level". Lastly, we equate each possible endpoint of a path on the helix with an integer, as shown in the figure.

Having gathered some intuition behind the cover, we now look at its implementation. The cover is a dependent space which is constructed using circle recursion (`S¹-recursion`). We need to provide an alternative to `base` and `loop`. Based on the expectation that the circle will map to the integers we give `Int` and the equivalence between different operations on integers we have created before (`succEquiv`).

We now prove that this definition of `Cover` gives the correct path-lifting action. There should be correspondences between `loop` and moving up one level, and `!loop` and moving down one level. Using the methods defined before we can construct `transport-Cover-loop` and `transport-Cover-!loop` to do exactly that.

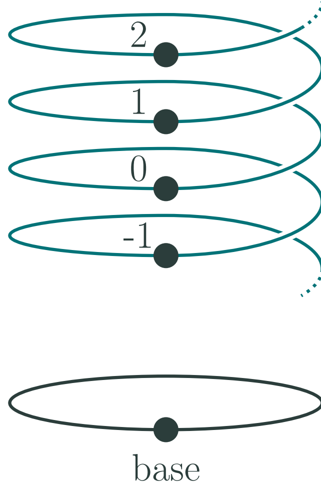


Figure 6: The helix, with possible endpoints marked with integers, shown "above" the circle (adapted from [11])

The Universal Cover

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

120 transport-Cover-loop : Path (transport Cover loop) succ
121 transport-Cover-loop =
122   transport Cover loop
123     ≃⟨ transport-ap-assoc Cover loop ⟩
124   transport (λ x → x) (ap Cover loop)
125     ≃⟨ ap   (transport (λ x → x))
126           (βloop/rec Int (univalence succEquiv)) ⟩
127   transport (λ x → x) (univalence succEquiv)
128     ≃⟨ transport-univ _ ⟩
129   succ ■

130 transport-Cover-!loop : Path (transport Cover (! loop)) pred
131 transport-Cover-!loop =
132   transport Cover (! loop)
133     ≃⟨ transport-ap-assoc Cover (! loop) ⟩
134   transport (λ x → x) (ap Cover (! loop))
135     ≃⟨ ap (transport (λ x → x)) (ap-! Cover loop)⟩
136   transport (λ x → x) (! (ap Cover loop))
137     ≃⟨ ap  (λ y → transport (λ x → x) (! y))
138          (βloop/rec Int (univalence succEquiv)) ⟩
139   transport (λ x → x) (! (univalence succEquiv))
140     ≃⟨ ap (transport (λ x → x)) (!-univalence succEquiv) ⟩
141   transport (λ x → x) (univalence (!-equiv succEquiv))
142     ≃⟨ transport-univ _ ⟩
143   pred ■
```

3.2 Proof

The goal of the proof is to construct an `HEquiv` between paths on the circle and the integers. Recall that the constructor for `HEquiv` takes four methods as arguments: `f` which maps some type `A` to some type `B`, `g` which maps `B` to `A`, α which shows there is a path between `g (f x))` and `x`, and finally β which shows there is a path between `f (g y))` and `y`. Here, `A` is paths on the circle and `B` the integers. The sections below show `f` (`encode`), `g` (`decode`), α (`encode-decode`) and β (`decode-encode`), respectively.

Encoding

`encode` is the function that takes a path on the circle and computes the corresponding element (integer) on the cover. Encoding happens in two stages. The first stage creates a function with the help of `transport` which we pass the dependent space (the cover) and the path which we want to encode. This function is then able, given a starting point in the dependent space, to compute the endpoint in it. Predictably, the second stage is providing this starting point to apply the function.

The authors have chosen `Zero` as the starting point, but remark that really any number could be chosen as long as it is taken into account in `decode`. The benefit of `Zero` is that no additional operations are required in decode. Furthermore, the use of this starting point means that `encode` computes the winding number or how many times a path goes around the circle. The orientation of the path is marked by whether the winding number is positive or negative.

Encoding

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

146 encode' : Path base base → Int
147 encode' α = encode {base} α

148 loop^ : Int → Path base base
149 loop^ Zero        = id
150 loop^ (Pos One)   = loop
151 loop^ (Pos (S n)) = loop ∘ loop^ (Pos n)
152 loop^ (Neg One)   = ! loop
153 loop^ (Neg (S n)) = ! loop ∘ loop^ (Neg n)

154 loop^-preserves-pred
155   : (n : Int) → Path (loop^ (pred n)) (! loop ∘ loop^ n)
156 loop^-preserves-pred (Pos One) = ! (!-inv-l loop)
157 loop^-preserves-pred (Pos (S y)) =
158      ! (∘-assoc (! loop) loop (loop^ (Pos y)))
159    ∘ ! (ap (λ x → x ∘ loop^ (Pos y)) (!-inv-l loop))
160    ∘ ! (∘-unit-l (loop^ (Pos y)))
161 loop^-preserves-pred Zero = id
162 loop^-preserves-pred (Neg One) = id
163 loop^-preserves-pred (Neg (S y)) = id
```

Decoding

Since `encode` essentially computes the winding number, `decode` must construct a path on the circle that corresponds to the given winding number. In other words, it computes the `n`-fold composition $\texttt{loop}^{\texttt{n}}$, for example $\texttt{loop}^{\texttt{1}}$ is `loop` and $\texttt{loop}^{\texttt{-2}}$ is `!loop ∘ !loop`. This part is given by the function `loop^`.

However, this would cause trouble in proving that decoding after encoding is correct. Usually, this would be done with path induction, but because neither endpoint is free, this is not possible. The type of `decode` must be generalised to `Path base x`, as opposed to `Path base base` like `loop^`. Thus we need to prove it holds not just for `base` but for any point on the circle. This can be achieved by using circle induction (`S¹-induction`).

We will first go through the three arguments passed to `S¹-induction` and then go into more detail. The first argument defines the property that we want to prove, the second gives the base case and the third shows the function is preserved going around the loop (the inductive step). The property we want to prove is that for any point on the circle which has been lifted on the helix, we can create a path on the circle starting at `base` and ending at the original point. Or in short: we want to prove we can decode it. The base case is relatively simple because we have already done the work to prove we can decode to `Path base base` with `loop^`. Hence, we pass this method as an argument. Matters get somewhat more complicated in the inductive step, where we must prove the function is preserved by going around the loop.

The formal notation of this is given in lines 166-169. We can then use a property of `transport` applied to functions which we proved earlier: `transport-→`. The result of this is lines 171-173. Similarly we can exploit the property of `transport` applied to families of paths which is defined in `transport-Path-right`. The argument α`'` is loop (recall that `loop` is `Path base base`) but we do not have a second path for α. The solution is to add a lambda statement that receives this path as an argument and passes it on for the post-composition. This simplifies the equation to what we see in lines 175-177. We then use `transport-Cover-!loop` which gives us a path from `!loop` to `pred` (lines 179-181). Now we can simplify the resulting composition of functions to serve as the path we were missing earlier giving line 183. Lastly, by the groupoid laws and `loop^preserves-pred` we can cancel out `pred` and `loop`. Now we have shown that there exists a path between where we started and `loop^`.

Decoding

```
164 decode : {x : S¹} → Cover x → Path base x
165 decode {x} =
166  S¹-induction
167   (λ x' → Cover x' → Path base x')
168   loop^
169   (transport (λ x' →  Cover x' → Path base x') loop loop^
170         ≃⟨transport-→ Cover (Path base) loop loop^⟩
171   transport (λ x' → Path base x') loop
172   o loop^
173   o transport Cover (! loop)
174         ≃⟨λ≃ (λ y → transport-Path-right loop (loop^ (transport Cover (! loop) y)))⟩
175   (λ p → loop ∘ p)
176   o loop^
177   o transport Cover (! loop)
178         ≃⟨λ≃ (λ y → ap (λ x' → loop ∘ loop^ x') (ap≃ transport-Cover-!loop))⟩
179   (λ p → loop ∘ p)
180   o loop^
181   o pred
182         ≃⟨id⟩
183   (λ n → loop ∘ (loop^ (pred n)))
184         ≃⟨λ≃ (λ y → move-left-! _ loop (loop^ y) (loop^-preserves-pred y))⟩
185   (λ n → loop^ n)
186   ■)
187  x
```

Encoding After Decoding

Next we turn to proving that given an element of the cover, first decoding it and then encoding it gives back the original element. We begin by expanding the definition of both `encode` and `decode` (line 193). Then, we can use the interaction between `ap` and `transport` together with `ap≃` to get the result we see on line 195. By expanding the definition of `transport` we get line 197. Lastly we can use the property of transport that applying it to families of paths is post-compositoin (`transport-Path-right`) to get α ∘ `id` which of course is equal to α. We now have a path between `encode (decode` α`)` and α.

Encoding After Decoding

```
188 encode-decode : {e : A + B} (c : Cover e)
189 → encode {e} (decode {e} c) ≃ c
190 encode-decode {Inl a'} α =
191 encode (decode α)
192     ≃⟨id⟩
193 transport Cover (ap Inl α) id
194     ≃⟨ ap≃ (! (transport-ap-assoc' Cover Inl α))⟩
195 transport(Cover o Inl) α id
196     ≃⟨id⟩
197 transport(λ a' → Path a a') α id
198     ≃⟨transport-Path-right α id⟩
199 α ∘ id
200     ≃⟨id⟩
201 α ■
```

Decoding After Encoding

The last component for this proof is providing a path between `decode (encode` α`)` and α. Here we can use path induction after expanding the definition of `encode`. This leaves us to prove there is a path between `decode (encode id)` and `id`, which is true by the definition of `id`. We now have all the components necessary to construct an `HEquiv` between the integers and paths on the circle which we can transform into a path by using `univalence`.

Decoding After Encoding

```
202 decode-encode  : {x : S¹} (α : Path base x)
203                → Path (decode (encode α)) α
204 decode-encode {x} α =
205  path-induction
206   (λ  (x' : S¹) (α' : Path base x')
207       → Path (decode (encode α')) α')
208   id α
```

4 Components' Dependence on the Univalence Axiom

What follows is an examination to which extent the components defined in section 3 depend on Voevodsky's univalence axiom (UA). Furthermore, this section investigates the consequences of omitting the univalence axiom. That is to say, what may be concluded from the proof if the axiom is not assumed to be true.

In order to fully grasp the scope of the impact, it is important to not only view each component in isolation but also in relation to the components it builds on. Hence, for each component, its dependency on previous components is investigated first. This is followed by the examination of its direct or indirect dependence on the univalence axiom. Finally, for each component, it is discussed what conclusions may still be drawn from it, if any.

4.1 Types As Spaces

The constructor `id` for `Path` is not dependent on the univalence axiom. `Path` itself is technically also not dependent on the UA, but without it can only contain trivial self-identifications. Interestingly, this impacts only tokens and not types since the latter are mere propositions. Path as a type can facilitate both constant paths and others without undermining any rules precisely because we can only create tokens for constant paths.

What complicates matters is the introduction of $\simeq$ which adds a way of creating tokens for other paths. This constructor is very much dependent on the UA. As a result, whenever we encounter $\simeq$ as a type, we can replace it with `Path`, which is consistent with its definition. In contrast, if we encounter $\simeq$ used as a constructor, the best we can do is replace it with `id`, which potentially reduces the applicability.

Path induction also still holds without the assumption of univalence. In actuality, it becomes simpler because there would only be trivial-self identifications. Hence, to prove a property, it only needs to be proven for `id`. With univalence, it holds as described in section 3.

The operations on paths `!` and $\circ$ both depend on path induction. They become somewhat trivial without univalence because they can only operate on constant paths, meaning regardless of inversion of concatenation, the result is still the constant path. Since path induction holds and the operations do not depend directly on univalence, the operations also hold.

4.2 Groupoid Laws

As noted in the introduction of the groupoid laws, they use path induction to reduce the proof to only the base case. Since without univalence, that is the only case that can exist, it arguably does not depend on path induction anymore. In other words, if the base case is the only case, no inductive step is needed. Since proving the base case does not require univalence, the groupoid laws are still valid.

4.3 Type Families As Dependent Spaces

`transport` only depends on `Path` as a data structure and not on the univalence axiom. The rules for post- and pre-composition additionally depend on `!` and $\circ$. Analogous to the groupoid laws, the rules no longer depend on path induction. The one detail that must be changed is replacing δ : $\theta 1 \simeq \theta 2$ with δ : `Path` $\theta 1$ $\theta 2$. This does not break any rules because without the UA the only paths are constant and thus don't need the $\simeq$ constructor. In summary, `transport` and its associated rules are still well founded.

4.4 Functions Are Functorial

Both `ap` and `apd` act as functors. This "propagation" of functions does not depend on the univalence axiom. That might, however, become more complicated if the function given as an argument is dependent on univalence. In that case, this dependency is also "propagated".

4.5 Paths Between More Than Points

Both `ap≃` and $\lambda\simeq$ are limited in their functionality in the absence of the univalence axiom. The reason is similar to what we have seen with regular operations on paths. The only path in a universe without univalence is the constant path, and applying any function to `id` results in `id`. Thus, both functions always return `id`.

4.6 Equivalences

Without the univalence axiom, we can still construct `HEquiv`s with the caveat that the paths α and β will be trivial self-identifications or constant paths. To see why we recall the discussion of tokens versus types from the beginning of this section. Since `!-equiv` only reorders the arguments, it only depends on `HEquiv` and not additionally on the univalence axiom.

It should come as no surprise that a method named `univalence` is dependent on the univalence axiom. And indeed, it maps equivalence to equality which is not guaranteed to be possible in a universe without univalence; it is only possible if the equivalence is between two already equal things. The methods `transport-univ` and `!-univalence` hinge on `univalence`. Consequentially, all three depend on the univalence axiom.

4.7 The Integers

Since the data structures that create the integers do not have any notion of equivalence or equality, they do not build on the univalence axiom. The same holds for the operations we can perform on integers: `succ` and `pred`. The question of whether `succEquiv` is dependent on univalence is a little more interesting; the third and fourth arguments require paths, but we can only construct constant paths. In other words, `pred-succ` and `succ-pred` should be constant paths. Indeed, we see from their definition that they are and thus may conclude the integers and operations on integers are not dependent on the UA.

4.8 The Circle

In section 3 we saw that the circle was defined as a point `base` and a path `loop`. This seemingly is still doable without the univalence axiom. However, `loop` was specifically defined to be a non-trivial self-identification. In other words, a path from `base` to `base` that was not equal to `id`. Without univalence, there is no way of constructing such a path: the only self-identifications are trivial.

Another way of looking at this issue is to recall from section 2 what conclusions we may draw in a universe without univalence. In the absence of the univalence axiom, it is consistent to assume axiom K. When we interpret axiom K's meaning from a homotopy viewpoint it confirms that the only `Path base base` is `id`.

It might seem like semantics to distinguish between trivial and non-trivial self-identifications. Changing `loop` to be the same as `id` does, however, impact the fundamental group of the circle. For example, we could previously conclude that there were different paths on the circle: `id`, `!loop` and `loop` $\circ$ `loop` to name a few. Now, regardless of which operation you perform on `id`, the resulting path will always still be `id` by its very definition.

Here we come to the root of why without the univalence axiom, the fundamental group of the circle is `1`; we have no way of creating any other paths than the constant path on the circle. Where there used to be different paths we could map to different integers, we are left with only one. And we can not construct an equivalence between one path and the integers.

5 Conclusion

The goal of this paper was two-fold: to understand how the proof of the fundamental group of the circle depends on the univalence axiom and why without it, the fundamental group is `1` as opposed to the integers. In short, the proof depends on the univalence axiom for constructing the circle. In a universe without univalence, it is consistent to assume axiom K. Consequently, we can no longer construct the circle in the same manner. There is now only one path on the circle, which we can map to `1`.

The original proof, given by Licata and Shulman [1], defines the circle as a higher-inductive type with a point `base` and a non-trivial path `loop`. The operations on `loop` include concatenation (∘) and inversion (`!`) and can be used to create different paths such as `!loop` and `loop ∘ loop`.

Using path lifting, we can create a dependent space above the circle in the shape of a helix. We create a mapping between `loop` and moving up one level on the helix and `!loop` and going down one level. If we now label each of the levels on the helix with an integer, we have a mapping between paths on the circle and the integers.

Without the univalence axiom, however, it is not possible to define the circle in the same way; we have no means of constructing a non-trivial path for `loop`. We can approximate the circle by using `id` instead, but by its very definition, any operations on it always result in `id`. As a consequence, there is only one path on the circle. And hence, the fundamental group of the circle without the univalence axiom is `1`.

A natural progression of this analysis would be to write a formal proof in Agda or Coq, similar to what Licata and Shulman have done but assuming K instead. This is outside the scope of this paper but would ensure we not only depend on the intuition provided here. Furthermore, if any alternatives to the univalence axiom are discovered in the future, by which we mean other axioms that influence the notion of equality, it would be interesting to see the fundamental group of the circle in that context.

6 Responsible Research

The ethics in mathematics seem, at first glance, much simpler compared to other fields. There are no data sets with implicit bias, no questionable experiments and no reproducibility problems. The first two are due to the nature of the field and the last by virtue of mathematical proofs. These are, in their essence, step-by-step expositions of the ideas presented. And where, historically, these were susceptible to human error, we now have computer assistance to avoid that.

There is, however, another side to this story. Part of the charm of abstract mathematics is, usually, not knowing what the benefits of discoveries are. This gives rise to two concerns in terms of social responsibility. The first is the question of whether the time and energy invested in mathematics could not be better invested in other fields. The second is purist tendencies which may arise. That is, to value purity above applicability [19].

Because we can not know how the field of homotopy type theory might contribute to society in the future, we will have to postpone judgement on the first issue; only time will tell. Turning now to the second concern. One of the benefits of HoTT is its flexibility. There are many interpretations and many different additional options such as the univalence axiom. As a result, it is unlikely to present purist tendencies.

References

[1] D. R. Licata and M. Shulman, "Calculating the fundamental group of the circle in homotopy type theory," *arXiv.org*, 1 2013. [Online]. Available: https://arxiv.org/abs/1301.3443

[2] P. B. Andrews, *An introduction to mathematical logic and type theory: to truth through proof.* Springer Science & Business Media, 2013, vol. 27.

[3] S. Thompson, *Type theory and functional programming.* Addison Wesley, 1991.

[4] A. Church, “A formulation of the simple theory of types,” *The journal of symbolic logic*, vol. 5, no. 2, pp. 56--68, 1940.

[5] H. P. Barendregt *et al.*, *The lambda calculus.* North-Holland Amsterdam, 1984, vol. 3.

[6] P. Martin-Löf and G. Sambin, *Intuitionistic type theory.* Bibliopolis Naples, 1984, vol. 9.

[7] G. Sambin and J. M. Smith, *Twenty five years of constructive type theory.* Clarendon Press, 1998, vol. 36.

[8] H. Barendregt and H. Geuvers, *Proof-Assistants Using Dependent Type Systems.* Elsevier Ltd, 2001, vol. 2, pp. 1149--1238.

[9] A. Bove, P. Dybjer, and U. Norell, “A brief overview of agda--a functional language with dependent types,” in *International Conference on Theorem Proving in Higher Order Logics.* Springer, 2009, pp. 73--78.

[10] G. Huet, G. Kahn, and C. Paulin-Mohring, “The coq proof assistant a tutorial,” *Rapport Technique*, vol. 178, 1997.

[11] T. Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics.* Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[12] V. Voevodsky, “The equivalence axiom and univalent models of type theory.(talk at cmu on february 4, 2010),” *arXiv preprint arXiv:1402.5556*, 2014.

[13] S. Awodey, Á. Pelayo, and M. A. Warren, “Voevodsky’s univalence axiom in homotopy type theory,” *Notices of the AMS*, vol. 60, no. 9, pp. 1164--1167, 2013.

[14] M. H. Escardó, “A self-contained, brief and complete formulation of voevodsky’s univalence axiom,” *arXiv.org*, 10 2018. [Online]. Available: https://arxiv.org/abs/1803.02294

[15] S. Dooley, “Basic algebraic topology: The fundamental group of circle,” *preprint, University of Chicago*, 2011.

[16] W. A. Howard, “The formulae-as-types notion of construction,” *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, vol. 44, pp. 479--490, 1980.

[17] J. Ladyman and S. Presnell, “Identity in homotopy type theory, part i: The justification of path induction,” *Philosophia Mathematica*, vol. 23, no. 3, pp. 386--406, 2015.

[18] S. Awodey, “Structuralism, invariance, and univalence,” *Philosophia Mathematica*, vol. 22, no. 1, pp. 1--11, 2014.

[19] P. Ernest, “Mathematics, ethics and purism: an application of macintyre’s virtue theory,” *Synthese*, vol. 199, no. 1, pp. 3137--3167, 2021.