Issue III: Homotopy Type Theory

Maksym Sokhatskyi [1]

[1] National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnical Institute

April 27, 2025

Abstract

Here is presented destinctive points of Homotopy Type Theory as an extension of Martin-Löf Type Theory but without higher inductive types which will be given in the next issue. The study of identity system is given. Groupoid (categorical) interpretation is presented as categories of spaces and paths between them as invertible morphisms. At last constructive proof $\Omega(S^1) = \mathbb{Z}$ is given through helix.

Keywords: Homotopy Theory, Type Theory

Contents

1 Groupoid Interpretation

1.1 Introduction: Type Theory

Type theory is a universal programming language for pure mathematics, designed for theorem proving. It supports an arbitrary number of consistent axioms, structured as pseudo-isomorphisms consisting of *encode* functions (methods for constructing type elements), *decode* functions (dependent eliminators of the universal induction principle), and their equations—beta and eta rules governing computability and uniqueness.

As a programming language, type theory includes basic primitives (axioms as built-in types) and accompanying documentation, such as lecture notes or textbooks, explaining their applications, including:

- Function ($\mathbf{\Pi}$)
- Context ($\mathbf{\Sigma}$)
- Identification ($=$)
- Polynomial ($\mathbf{W}$)
- Path (Ξ)
- Gluing (**Glue**)
- Infinitesimal ($\Im$)
- Complex (**HIT**)

Students (10) are tasked with applying type theory to prove an initial but non-trivial result addressing an open problem in one of the following areas offered by the Department of Pure Mathematics (KM-111):

$$\text{Mathematics} := \begin{cases} \text{Homotopy Theory} \\ \text{Homological Algebra} \\ \text{Category Theory} \\ \text{Functional Analysis} \\ \text{Differential Geometry} \end{cases}.$$

1.2 Motivation: Homotopy Type Theory

The primary motivation of homotopy type theory is to provide computational semantics for homotopic types and CW-complexes. The central idea, as described in, is to combine function spaces (Π), context spaces (Σ), and path spaces (Ξ) to form a fiber bundle, proven within HoTT to coincide with the Π type itself.

Key definitions include:

```
def contr (A: U) : U := Σ (x: A), Π (y: A), Ξ A x y
def fiber (A B: U) (f: A → B) (y: B): U := Σ (x: A), Path B y (f x)
def isEquiv (A B: U) (f: A → B): U := Π (y: B), contr(fiber A B f y)
def equiv (X Y: U): U := Σ (f: X → Y), isEquiv X Y f
def ua (A B : U) (p : Ξ U A B) : equiv A B
 := transp (<i> equiv A (p @ i)) 0 (idEquiv A)
```

The absence of an eta-rule for equality implies that not all proofs of the same path space are equal, resulting in a multidimensional ∞-groupoid structure for path spaces. Further definitions include:

```
def isProp (A : U) : U
 := Π (a b : A), Ξ A a b

def isSet (A : U) : U
 := Π (a b : A) (x y : Ξ A a b), Ξ (Ξ A a b) x y

def isGroupoid (A : U) : U
 := Π (a b : A) (x y : Ξ A a b) (i j : Ξ (Ξ A a b) x y),
    Ξ (Ξ (Ξ A a b) x y) i j
```

The groupoid interpretation raises questions about the existence of a language for mechanically proving all properties of the categorical definition of a groupoid:

```
def CatGroupoid (X : U) (G : isGroupoid X)
  : isCatGroupoid (PathCat X)
 := ( idp X,
      comp-Path X,
      G,
      sym X,
      comp-inv-Path^-1 X,
      comp-inv-Path X,
      comp-Path-left X,
      comp-Path-right X,
      comp-Path-assoc X,
      ⋆
    )
```

1.3 Metatheory: Adjunction Triples

The course is divided into four parts, each exploring type-axioms and their meta-theoretical adjunctions.

1.3.1 Fibrational Proofs

$$\Sigma \dashv f_\star \dashv \Pi$$

Fibrational proofs are modeled by primitive axioms, which are type-theoretic representations of categorical meta-theoretical models of adjunctions of three Cockett-Reit functors, giving rise to function spaces (Π) and pair spaces (Σ). These proof methods enable direct analysis of fibrations.

1.3.2 Equality Proofs

$$\mathrm{Q} \dashv \Xi \dashv \mathrm{C}$$

In intensional type theory, the equality type is embedded as type-theoretic primitives of categorical meta-theoretical models of adjunctions of three Jacobs-Lambek functors: quotient space (Q), identification system (Ξ), and contractible space (C). These methods allow direct manipulation of identification systems, strict for set theory and homotopic for homotopy theory.

1.3.3 Inductive Proofs

$$\mathrm{W} \dashv \odot \dashv \mathrm{M}$$

Inductive types in type theory can be embedded as polynomial functors (W, M) or general inductive type schemes (Calculus of Inductive Constructions), with properties including: 1) Verification of program finiteness; 2) Verification of strict positivity of parameters; 3) Verification of mutual recursion.

In this course, induction and coinduction are introduced as type-theoretic primitives of categorical meta-theoretical models of adjunctions of polynomial functors (Lambek-Bohm), enabling manipulation of initial and terminal algebras, algebraic recursive data types, and infinite processes. Higher inductive proofs, where constructors include path spaces, are modeled by polynomial functors using monad-algebras and comonad-coalgebras (Lumsdaine-Shulman).

Historical Notes

Homotypy Type Theory takes its origins in 1996 from groupoid interpretation by Hofmann and Streicher's, and later (in 10 years) was formalized by Awodey, Warren and Voevodsky. Voevodsky constrtucted Kan simplicial sets interpretation of type theory and discovered the property of this model, that was named univalence. This property allows to identify isomorphic structures in terms of type theory.

Homotopy type theory to classical homotopy theory is like Euclidian syntethic geometry (points, lines, axioms and deduction rules) to analytical geometry with cartesian coordinates on $\mathbb{R}^n$ (geometric and algebraic)[1].

In the same way as inductive types extends MLTT for inductive programming, the higher inductive types (HIT) extend homotopy type theory for geometry programming. You can directly encode CW-complexes by using HIT. The definition of HIT syntax will be given in the next **Issue IV: Higher Inductive Types**.

Cubical with HITs has very lightweight core and syntax, and is an internal language of $(\infty, 1)$-topos. Cubical with $[0, 1]$ Path types but without HITs is an internal language of $(\infty, 1)$-categories, while MLTT is an internal language of locally cartesian closed categories.

Acknowledgement

This article is dedicated to Ihor Horobets and written on his request for clarification and direct intoduction to HoTT.

[1]We will denote geometric, type theoretical and homotopy constants bold font $\mathbf{R}$ while analitical will be denoted with double lined letters $\mathbb{R}$.

2 Homotopy Type Theory

2.1 Identity Systems

Definition 1. (Identity System). An identity system over type A in universe X_i is a family $R : A \to A \to X_i$ with a function $r_0 : \Pi_{a:A} R(a,a)$ such that any type family $D : \Pi_{a,b:A} R(a,b) \to X_i$ and $d : \Pi_{a:A} D(a,a,r_0(a))$, there exists a function $f : \Pi_{a,b:A} \Pi_{r:R(a,b)} D(a,b,r)$ such that $f(a,a,r_0(a)) = d(a)$ for all $a : A$.

```
def IdentitySystem (A : U) : U
 := Σ (=-form  : A → A → U)
      (=-ctor  : Π (a : A), =-form a a)
      (=-elim  : Π (a : A) (C: Π (x y : A)
                 (p : =-form x y), U)
                 (d : C a a (=-ctor a)) (y : A)
                 (p : =-form a y), C a y p)
      (=-comp  : Π (a : A) (C: Π (x y : A)
                 (p : =-form x y), U)
                 (d : C a a (=-ctor a)),
                 Ξ (C a a (=-ctor a)) d
                      (=-elim a C d a (=-ctor a))), 1
```

Example 1. There are number of equality signs used in this tutorial, all of them listed in the following table of identity systems:

Sign	Meaning
$=_{def}$	Definition
$=$	Id
$\equiv$	Path
$\simeq$	Equivalence
$\cong$	Isomorphism
$\sim$	Homotopy
$\approx$	Bisimulation

Theorem 1. (Fundamental Theorem of Identity System).

Definition 2. (Strict Identity System). An identity system over type A and universe of pretypes V_i is called strict identity system ($=$), which respects UIP.

Definition 3. (Homotopy Identity System). An identity system over type A and universe of homotopy types U_i is called homotopy identity system ($\equiv$), which models discrete infinity groupoid.

2.2 Path (Ξ)

The homotopy identity system defines a **Path** space indexed over type A with elements as functions from interval $[0,1]$ to values of that path space $[0,1] \to A$. HoTT book defines two induction principles for identity types: path induction and based path induction.

Definition 4. (Path Formation).

$$\equiv : U =_{def} \prod_{A:U} \prod_{x,y:A} \mathbf{Path}_A(x,y).$$

```
def Ξ (A : U) (x y : A) : U
 := PathP (<_> A) x y

def Ξ′ (A : U) (x y : A)
 := Π (i : I),
     A [∂ i |-> [(i = 0) → x ,
                  (i = 1) → y ]]
```

Definition 5. (Path Introduction). Returns a reflexivity path space for a given value of the type. The inhabitant of that path space is the lambda on the homotopy interval $[0,1]$ that returns a constant value x. Written in syntax as $[i]x$.

$$\mathrm{id}_{\equiv} : x \equiv_A x =_{def} \prod_{A:U} \prod_{x:A} [i]x$$

```
def idp (A: U) (x: A)
  : Ξ A x x := <_> x
```

Definition 6. (Path Application).

```
def at0 (A: U) (a b: A)
    (p: Path A a b) : A := p @ 0

def at1 (A: U) (a b: A)
    (p: Path A a b): A := p @ 1
```

Definition 7. (Path Connections). Connections allow you to build a square with only one element of path: i) $[i,j]p$ @ $min(i,j)$; ii) $[i,j]p$ @ $max(i,j)$.

$$\begin{tikzcd}
b \arrow[r, "{[i]b}"] & b & a \arrow[r, "p"] & b \\
a \arrow[u, "p"] \arrow[r, "p"] & b \arrow[u, "{[i]b}"] & a \arrow[u, "{[i]a}"] \arrow[r, "{[i]a}"] & a \arrow[u, "p"]
\end{tikzcd}$$

```
def join (A: U) (a b: A) (p: Path A a b)
  : PathP (<x> Path A (p@x) b) p (<i> b)
 := <y x> p @ (x \/ y)

def meet (A: U) (a b: A) (p: Path A a b)
  : PathP (<x> Path A a (p@x)) (<i> a) p
 := <x y> p @ (x /\ y)
```

Definition 8. (Path Inversion).

Theorem 2. (Congruence).

$$\text{ap} : f(a) \equiv f(b) =_{def}$$

$$\prod_{A:U} \prod_{a,x:A} \prod_{B:A\rightarrow U} \prod_{f:\Pi(A,B)} \prod_{p:a\equiv_A x} [i]f(p@i).$$

```
def ap (A B: U) (f: A -> B)
     (a b: A) (p: Path A a b)
  : Path B (f a) (f b)

def apd (A: U) (a x: A) (B: A -> U)
     (f: A -> B a) (b: B a) (p: Path A a x)
  : Path (B a) (f a) (f x)
```

Maps a given path space between values of one type to path space of another type using an encode function between types. Implemented as a lambda defined on $[0,1]$ that returns application of encode function to path application of the given path to lamda argument $[i]f(p@i)$ in both cases.

Definition 9. (Generalized Transport Kan Operation). Transports a value of the left type to the value of the right type by a given path element of the path space between left and right types.

$$\text{transport} : A(0) \to A(1) =_{def}$$

$$\prod_{A:I\to U} \prod_{r:I}$$

$$\lambda x, \mathbf{transp}([i]A(i), 0, x).$$

```
def transp' (A: U) (x y: A) (p : PathP (&lt;_>A) x y) (i: I)
 := transp (&lt;i> (\(_:A),A) (p @ i)) i x
```

```
def transp^U (A B: U) (p : PathP (&lt;_>U) A B) (i: I)
 := transp (&lt;i> (\(_:U),U) (p @ i)) i A
```

Definition 10. (Partial Elements).

$$\text{Partial} : V =_{def} \prod_{A:U} \prod_{i:I} \mathbf{Partial}(A, i).$$

```
def Partial' (A : U) (i : I)
  : V := Partial A i
```

Definition 11. (Cubical Subtypes).

$$\text{Subtype} : V =_{def}$$

$$\prod_{A:U} \prod_{i:I} \prod_{u:\mathbf{Partial}(A,i)} A[i \mapsto u].$$

```
def sub (A : U) (i : I) (u : Partial A i)
  : V := A [i ↦ u]
```

Definition 12. (Cubical Elements).

$$\text{inS} : A\ [(i = 1) \mapsto a] =_{def}$$

$$\prod_{A:U} \prod_{i:I} \prod_{a:A} \mathbf{inc}(A, i, a).$$

$$\text{outS} : A\ [i \mapsto u] \to A =_{def}$$

$$\prod_{A:U} \prod_{i:I} \prod_{u:\mathbf{Partial}(A,i)} \mathbf{ouc}(a).$$

```
def inS (A : U) (i : I) (a : A)
  : sub A i [(i = 1) → a] := inc A i a
```

```
def outS (A : U) (i : I) (u : Partial A i)
  : A [i ↦ u] -> A := λ (a: A[i ↦ u]), ouc a
```

Theorem 3. (Heterogeneous Composition Kan Operation).

$$\text{comp}_{\text{CCHM}} : A(0)\ [r \mapsto u(0)] \to A(1) =_{def}$$

$$\prod_{A:U} \prod_{r:I} \prod_{u:\Pi_{i:I}\mathbf{Partial}(A(i),r)}$$

$$\lambda u_0, \mathbf{hcomp}(A(1), r, \lambda i.$$

$$[(r{=}1) \to \mathbf{transp}([j]A(i/j), i, u(i, 1{=}1))],$$

$$\mathbf{transp}([i]A(i), 0, \mathbf{ouc}(u_0))).$$

```
def compCCHM (A : I → U) (r : I)
     (u : Π (i : I), Partial (A i) r)
     (u₀ : (A 0)[r ↦ u 0]) : A 1
 := hcomp (A 1) r (λ (i : I),
     [ (r = 1) → transp (<j> A (i ∨ j)) i (u i 1=1)])
                     (transp (<i> A i) 0 (ouc u₀))
```

Theorem 4. (Homogeneous Composition Kan Operation).

$$\text{comp}_{\text{CHM}} : A\ [r \mapsto u(0)] \to A =_{def}$$

$$\prod_{A:U} \prod_{r:I} \prod_{u:I \to \mathbf{Partial}(A,r)}$$

$$\lambda u_0, \mathbf{hcomp}(A, r, u, \mathbf{ouc}(u_0)).$$

```
def compCHM (A : U) (r : I)
     (u : I → Partial A r) (u₀ : A[r ↦ u 0]) : A
 := hcomp A r u (ouc u₀)
```

Theorem 5. (Substitution).

$$\text{subst} : P(x) \to P(y) =_{def}$$

$$\prod_{A:U} \prod_{P:A \to U} \prod_{x,y:A} \prod_{p:x=y}$$

$$\lambda e.\mathbf{transp}([i]P(p@i), 0, e).$$

```
def subst (A: U) (P: A -> U) (x y: A) (p: Path A x y)
  : P x -> P y
 := λ (e: P x), transp (<i> P (p @ i)) 0 e
```

Other synonyms are mapOnPath and cong.

Theorem 6. (Path Composition).

$$\begin{tikzcd} a \arrow[r, "pcomp"] & c \\ a \arrow[u, "{[i]a}"] \arrow[r, "p\ @\ i"'] & b \arrow[u, "q"'] \end{tikzcd}$$

```
def pcomp (A: U) (a b c: A)
      (p: Path A a b) (q: Path A b c)
   : Path A a c := subst A (Path A a) b c q p
```

Composition operation allows building a new path from two given paths in a connected point. The proofterm is $\mathbf{comp}([i]\mathbf{Path}_A(a, q@i), p, [])$.

Theorem 7. (J by Paulin-Mohring).

```
def J (A: U) (a b: A)
      (P: singl A a -> U)
      (u: P (a,refl A a))
   : Π (p: Path A a b), P (b,p)
```

J is formulated in a form of Paulin-Mohring and implemented using two facts that singletons are contractible and dependent function transport.

Theorem 8. (Contractability of Singleton).

```
def singl (A: U) (a: A) : U
 := Σ (x: A), Path A a x

def contr (A: U) (a b: A) (p: Path A a b)
   : Path (singl A a) (a,<_>a) (b,p)
```

Proof that singleton is contractible space. Implemented as $[i](p@i, [j]p@(i \wedge j))$.

Theorem 9. (HoTT Dependent Eliminator).

```
def J (A: U) (a: A)
      (C: (x: A) -> Path A a x -> U)
      (d: C a (refl A a)) (x: A)
   : Π (p: Path A a x) : C x p
```

Theorem 10. (Diagonal Path Induction).

```
def D (A: U) : U
 := Π (x y: A), Path A x y -> U

def J (A: U) (x: A) (C: D A)
      (d: C x x (refl A x))
      (y: A)
   : Π (p: Path A x y), C x y p
```

Theorem 11. (Path Computation).

```
def trans_comp (A: U) (a: A)
  : Path A a (trans A A (<_> A) a)

def subst_comp (A: U) (P: A -> U) (a: A) (e: P a)
  : Path (P a) e (subst A P a a (refl A a) e)

def J_comp (A: U) (a: A)
    (C: (x: A) -> Path A a x -> U)
    (d: C a (refl A a))
  : Path (C a (refl A a)) d
    (J A a C d a (refl A a))
```

Note that in HoTT there is no Eta rule, otherwise Path between element would requested to be unique applying UIP at any Path level which is prohibited. UIP in HoTT is defined only as instance of n-groupoid, see the PROP type.

2.3 Glue

Glue types defines composition structure for fibrant universes that allows partial elements to be extended to fibrant types. In other words it turns equivalences in the multidensional cubes to path spaces. Unlike ABCHFL, CCHM needn't another universe for that purpose.

Definition 13. (Glue Formation). 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.

$$\mathbf{Glue}(A, \varphi, e) : U.$$

```
def Glue' (A : U) (φ : I)
    (e : Partial (Σ (T : U), equiv T A) φ) : U
 := Glue A φ e
```

Definition 14. (Glue Introduction).

$$\mathbf{glue}\ \varphi\ u\ (\mathbf{ouc}\ a) : \mathbf{Glue}\ A\ [\varphi{=}1 \mapsto (T, f)].$$

```
def glue' (A : U) (φ : I)
    (u : Partial (Σ (T : U), equiv T A × T) φ)
    (a : A [φ ↦ [(φ = 1) → (u 1=1).2.1.1 (u 1=1).2.2]])
 := glue φ u (ouc a)
```

Definition 15. (Glue Elimination).

$$\mathbf{unglue}(b) : A\ [\varphi \mapsto f(b)].$$

```
def unglue′ (A : U) (φ : I)
    (e : Partial (Σ (T : U), equiv T A) φ)
    (a : Glue A φ e) : A
 := unglue φ e a
```

Theorem 12. (Glue Computation).

$$b = \mathbf{glue}\ [\varphi \mapsto b]\ (\mathbf{unglue}\ b).$$

Theorem 13. (Glue Uniqueness).

$$\mathbf{unglue}\ (\mathbf{glue}\ [\varphi \mapsto t]\ a) = a : A.$$

2.4 Fibration

Definition 16 (Fiber)**.** The fiber of the map $p : E \to B$ at a point $y : B$ is the set of all points $x : E$ such that $p(x) = y$.

```
fiber (E B: U) (p: E -> B) (y: B): U
    = (x: E) * Ξ B y (p x)
```

Definition 17 (Fiber Bundle)**.** The fiber bundle $F \to E \xrightarrow{p} B$ on a total space E with fiber layer F and base B is a structure (F, E, p, B), where $p : E \to B$ is a surjective map with the following property: for any point $y : B$ there exists a neighborhood U_b for which there is a homeomorphism

$$f : p^{-1}(U_b) \to U_b \times F$$

making the following diagram commute:

$$\begin{tikzcd} p^{-1}(U_b) \arrow[r, "f"] \arrow[d, "p"'] & U_b \times F \arrow[dl, "pr_1"] \\ U_b & \end{tikzcd}$$

Definition 18 (Trivial Fiber Bundle)**.** When the total space E is the cartesian product $\Sigma(B, F)$ and $p = pr_1$, then such a bundle is called trivial: $(F, \Sigma(B, F), pr_1, B)$.

```
Family (B: U): U = B -> U

total (B: U) (F: Family B): U = Sigma B F
trivial (B: U) (F: Family B): total B F -> B = \(x: total B F) -> x.1
homeo (B E: U) (F: Family B) (p: E -> B) (y: B):
    fiber E B p y -> total B F
```

Theorem 14 (Fiber Bundle $\equiv \Pi$)**.** The inverse image (fiber) of the trivial bundle $(F, B \times F, pr_1, B)$ at a point $y : B$ equals $F(y)$. Proof sketch:

```
F y = (_: isContr B) * (F y)
    = (x y: B) * (_: Ξ B x y) * (F y)
    = (z: B) * (k: F z) * Ξ B z y
    = (z: E) * Ξ B z.1 y
    = fiber (total B F) B (trivial B F) y
```

The equality is shown using the *isoPath* lemma and *encode/decode* functions.

```
def Family (B : U) : U₁ := B → U
def Fibration (B : U) : U₁ := Σ (X : U), X → B

def encode-Pi (B : U) (F : B → U) (y : B)
  : fiber (Sigma B F) B (pr₁ B F) y → F y
 := \ (x : fiber (Sigma B F) B (pr₁ B F) y),
      subst B F x.1.1 y (<i> x.2 @ -i) x.1.2

def decode-Pi (B : U) (F : B → U) (y : B)
  : F y → fiber (Sigma B F) B (pr₁ B F) y
 := \ (x : F y), ((y, x), idp B y)

def decode-encode-Pi (B : U) (F : B → U) (y : B) (x : F y)
  : Ξ (F y) (transp (<i> F (idp B y @ i)) 0 x) x
 := <j> transp (<i> F y) j x

def encode-decode-Pi (B : U) (F : B → U) (y : B)
    (x : fiber (Sigma B F) B (pr₁ B F) y)
  : Ξ (fiber (Sigma B F) B (pr₁ B F) y)
         ((y, encode-Pi B F y x), idp B y) x
 := <i> ( (x.2 @ i, transp (<j> F (x.2 @ i ∨ -j)) i x.1.2),
          <j> x.2 @ i ∧ j )

def Bundle=Pi (B : U) (F : B → U) (y : B)
  : PathP (<_> U) (fiber (Sigma B F) B (pr₁ B F) y) (F y)
 := iso→Path (fiber (Sigma B F) B (pr₁ B F) y) (F y)
    (encode-Pi B F y) (decode-Pi B F y)
    (decode-encode-Pi B F y) (encode-decode-Pi B F y)
```

Definition 19. (Fibration-1) Dependent fiber bundle derived from Ξ contractability.

```
def isFBundle1 (B: U) (p: B → U) (F: U): U₁
 := Σ (_: Π (b: B), isContr (PathP (<_>U) (p b) F)), (Π (x: Sigma B p), B)
```

Definition 20. (Fibration-2). Dependent fiber bundle derived from surjective function.

```
def isFBundle2 (B: U) (p: B → U) (F: U): U
 := Σ (v: U) (w: surjective v B), (Π (x: v), PathP (<_>U) (p (w.1 x)) F)
```

Definition 21. (Fibration-3). Non-dependent fiber bundle derived from fiber truncation.

```
def im₁ (A B: U) (f: A → B): U
 := Σ (b: B), ‖_‖₋₁ (Π (a : A), Path B (f a) b)

def BAut (F: U): U := im₁ 1 U (λ (x: 1), F)

def 1–Im₁ (A B: U) (f: A → B): im₁ A B f → B
 := λ (x : im₁ A B f), x.1

def 1–BAut (F: U): BAut F → U := 1–Im₁ 1 U (λ (x: 1), F)

def classify (E: U) (A' A: U) (E': A' → U) (E: A → U)
     (f: A' → A): U := Π(x: A'), Ξ U (E'(x)) (E(f(x)))

def isFBundle3 (E B: U) (p: E → B) (F: U): U₁
 := Σ (X: B → BAut F),
       classify E B (BAut F) (λ (b: B), fiber E B p b)
                (1–BAut F) X
```

Definition 22. (Fibration-4). Non-dependen fiber bundle derived as pullback square.

```
def isFBundle4 (E B: U) (p: E → B) (F: U): U₁
 := Σ (X: U) (v: surjective X B)
       (v': prod X F → E),
       pullbackSq (prod X F) E X B p v.1 v' (λ (x: prod X F), x.1)
```

2.5 Equivalence

Definition 23. (Fiberwise Equivalence). Fiberwise equivalence $\simeq$ or Equiv of function $f : A \to B$ represents internal equality of types A and B in the universe U as contractible fibers of f over base B.

$$A \simeq B : U \ \ =_{def} \mathbf{Equiv}(A,B) : U =_{def}$$

$$\sum_{f:A\to B} \prod_{y:B} \sum_{x:\Sigma_{x:A} y =_B f(x)} \sum_{w:\Sigma_{x:A} y=_B f(x)} x =_{\Sigma_{x:A} y =_B f(x)} w.$$

```
def isContr (A: U) : U
 := Σ (x: A), Π (y: A), Ξ A x y

def fiber (A B : U) (f: A → B) (y : B): U
 := Σ (x : A), Ξ B y (f x)

def isEquiv (A B : U) (f : A → B) : U
 := Π (y : B), isContr (fiber A B f y)

def equiv (A B : U) : U
 := Σ (f : A → B), isEquiv A B f
```

Definition 24. (Fiberwise Reflection). There is a fiberwise instance $\text{id}_\simeq$ of $A \simeq A$ that is derived as $(\text{id}(A), \text{isContrSingl}(A))$:

$$\text{id}_\simeq : \mathbf{Equiv}(A,A).$$

```
def singl (A: U) (a: A): U
 := Σ (x: A), Ξ A a x

def contr (A : U) (a b : A) (p : Ξ A a b)
  : Ξ (singl A a) (eta A a) (b, p)
 := <i> (p @ i , &lt;j> p @ i /\ j)

def isContrSingl (A : U) (a : A) : isContr (singl A a)
 := ((a,idp A a),(\(z:singl A a),contr A a z.1 z.2))

def idEquiv (A : U) : equiv A A
 := (\(a:A) -> a, isContrSingl A)
```

Theorem 15. (Fiberwise Induction Principle). For any $P : A \to B \to A \simeq B \to U$ and it's evidence d at $(B, B, \text{id}_\simeq(B))$ there is a function $\mathbf{Ind}_\simeq$. HoTT 5.8.5

$$\mathbf{Ind}_\simeq(P,d) : (p : A \simeq B) \to P(A,B,p).$$

```
def J-equiv (A B: U)
    (P: Π (A B: U), equiv A B → U)
    (d: P B B (idEquiv B))
  : Π (e: equiv A B), P A B e
 := λ (e: equiv A B),
    subst (single B) (\ (z: single B), P z.1 B z.2)
                     (B,idEquiv B) (A,e)
                     (contrSinglEquiv A B e) d
```

Theorem 16. (Fiberwise Computation of Induction Principle).

```
def compute−Equiv (A : U)
     (C : Π (A B: U), equiv A B → U)
     (d: C A A (idEquiv A))
   : Ξ (C A A (idEquiv A)) d
     (ind−Equiv A A C d (idEquiv A))
```

Definition 25. (Surjective).

```
isSurjective (A B: U) (f: A −> B): U
  = (b: B) * pTrunc (fiber A B f b)

surjective (A B: U): U
  = (f: A −> B)
  * isSurjective A B f
```

Definition 26. (Injective).

```
isInjective' (A B: U) (f: A −> B): U
  = (b: B) −> isProp (fiber A B f b)

injective (A B: U): U
  = (f: A −> B)
  * isInjective A B f
```

Definition 27. (Embedding).

```
isEmbedding (A B: U) (f: A −> B) : U
  = (x y: A) −> isEquiv (Ξ A x y) (Ξ B (f x) (f y)) (cong A B f x y)

embedding (A B: U): U
  = (f: A −> B)
  * isEmbedding A B f
```

Definition 28. (Half-adjoint Equivalence).

```
isHae (A B: U) (f: A −> B): U
  = (g: B −> A)
  * (eta_: Ξ (id A) (o A B A g f) (idfun A))
  * (eps_: Ξ (id B) (o B A B f g) (idfun B))
  * ((x: A) −> Ξ B (f ((eta_ @ 0) x)) ((eps_ @ 0) (f x)))

hae (A B: U): U
  = (f: A −> B)
  * isHae A B f
```

2.6 Homotopy

The first higher equality we meet in homotopy theory is a notion of homotopy, where we compare two functions or two path spaces (which is sort of dependent families). The homotopy interval I = $[0,1]$ is the perfect foundation for definition of homotopy.

Definition 29. (Interval). Compact interval.

```
def I : U := inductive { i0 | i1 | seg : i0 ≡ i1 }
```

You can think of **I** as isomorphism of equality type, disregarding carriers on the edges. By mapping $i0, i1 : \mathbf{I}$ to $x, y : A$ one can obtain identity or equality type from classic type theory.

Definition 30. (Interval Split). The convertion function from I to a type of comparison is a direct eliminator of interval. The interval is also known as one of primitive higher inductive types which will be given in the next **Issue IV: Higher Inductive Types**.

```
def pathToHtpy (A: U) (x y: A) (p: Ξ A x y) : I → A
 := split { i0 → x | i1 → y | seg @ i → p @ i }
```

Definition 31. (Homotopy). The homotopy between two function $f, g : X \to Y$ is a continuous map of cylinder $H : X \times \mathbf{I} \to Y$ such that

$$\begin{cases} H(x,0) = f(x), \\ H(x,1) = g(x). \end{cases}$$

```
homotopy (X Y: U) (f g: X -> Y)
         (p: (x: X) -> Ξ Y (f x) (g x))
         (x: X): I -> Y = pathToHtpy Y (f x) (g x) (p x)
```

Definition 32. (funExt-Formation)

```
funext_form (A B: U) (f g: A -> B): U
  = Ξ (A -> B) f g
```

Definition 33. (funExt-Introduction)

```
funext (A B: U) (f g: A -> B) (p: (x:A) -> Ξ B (f x) (g x))
  : funext_form A B f g
  = <i> \(a: A) -> p a @ i
```

Definition 34. (funExt-Elimination)

```
happly (A B: U) (f g: A -> B) (p: funext_form A B f g) (x: A)
  : Ξ B (f x) (g x)
  = cong (A -> B) B (\(h: A -> B) -> apply A B h x) f g p
```

Definition 35. (funExt-Computation)

```
funext_Beta (A B: U) (f g: A -> B) (p: (x:A) -> Ξ B (f x) (g x))
  : (x:A) -> Ξ B (f x) (g x)
  = \(x:A) -> happly A B f g (funext A B f g p) x
```

Definition 36. (funExt-Uniqueness)

```
funext_Eta (A B: U) (f g: A -> B) (p: Ξ (A -> B) f g)
  : Ξ (Ξ (A -> B) f g) (funext A B f g (happly A B f g p)) p
  = refl (Ξ (A -> B) f g) p
```

2.7 Isomorphism

Definition 37. (iso-Formation)

```
iso_Form (A B: U): U = isIso A B -> Ξ U A B
```

Definition 38. (iso-Introduction)

```
iso_Intro (A B: U): iso_Form A B
```

Definition 39. (iso-Elimination)

```
iso_Elim (A B: U): Ξ U A B -> isIso A B
```

Definition 40. (iso-Computation)

```
iso_Comp (A B : U) (p : Ξ U A B)
  : Ξ (Ξ U A B) (iso_Intro A B (iso_Elim A B p)) p
```

Definition 41. (iso-Uniqueness)

```
iso_Uniq (A B : U) (p: isIso A B)
  : Ξ (isIso A B) (iso_Elim A B (iso_Intro A B p)) p
```

2.8 Univalence

Definition 42. (uni-Formation)

```
univ_Formation (A B: U): U = equiv A B -> Ξ U A B
```

Definition 43. (uni-Introduction)

```
equivToΞ (A B: U): univ_Formation A B
  = \(p: equiv A B) -> <i> Glue B [(i=0) -> (A,p),
    (i=1) -> (B, subst U (equiv B) B B (<_>B) (idEquiv B)) ]
```

Definition 44. (uni-Elimination)

```
pathToEquiv (A B: U) (p: Ξ U A B) : equiv A B
  = subst U (equiv A) A B p (idEquiv A)
```

Definition 45. (uni-Computation)

```
eqToEq (A B : U) (p : Ξ U A B)
  : Ξ (Ξ U A B) (equivToPath A B (pathToEquiv A B p)) p
  = <j i> let Ai: U = p@i in Glue B
    [ (i=0) -> (A,pathToEquiv A B p),
      (i=1) -> (B,pathToEquiv B B (<k> B)),
      (j=1) -> (p@i,pathToEquiv Ai B (<k> p @ (i \/ k))) ]
```

Definition 46. (uni-Uniqueness)

```
transPathFun (A B : U) (w: equiv A B)
  : Ξ (A -> B) w.1 (pathToEquiv A B (equivToPath A B w)).1
```

2.9 Loop

Definition 47. (Pointed Space). A pointed type (A, a) is a type $A : U$ together with a point $a : A$, called its basepoint.

```
pointed: U = (A: U) * A
point (A: pointed): A.1 = A.2
space (A: pointed): U = A.1
```

Definition 48. (Loop Space).

$$\Omega(A, a) =_{def} ((a =_A a), refl_A(a)).$$

```
omega1 (A: pointed) : pointed
  = (Ξ (space A) (point A) (point A), refl A.1 (point A))
```

Definition 49. (n-Loop Space).

$$\begin{cases} \Omega^0(A, a) =_{def} (A, a) \\ \Omega^{n+1}(A, a) =_{def} \Omega^n(\Omega(A, a)) \end{cases}$$

```
omega : nat -> pointed -> pointed = split
  zero -> idfun pointed
  succ n -> \(A: pointed) -> omega n (omega1 A)
```

2.10 Groupoid

The first text about groupoid interpretation of type theory can be found in Francois Lamarche: A proposal about Foundations[2]. Then Martin Hofmann and Thomas Streicher wrote the initial document on groupoid interpretation of type theory[3].

Equality	**Homotopy**	**∞-Groupoid**
reflexivity	constant path	identity morphism
symmetry	inversion of path	inverse morphism
transitivity	concatenation of paths	composition of mopphisms

There is a deep connection between higher-dimentinal groupoids in category theory and spaces in homotopy theory, equipped with some topology. The category or groupoid could be built where the objects are particular spaces or types, and morphisms are path types between these types, composition operation is a path concatenation. We can write this groupoid here recalling that it should be category with inverted morphisms.

```
cat: U = (A: U) * (A -> A -> U)
groupoid: U = (X: cat) * isCatGroupoid X
PathCat (X: U): cat = (X,\(x y:X)->Path X x y)

def isCatGroupoid (C: cat): U := Σ
    (id:        Π (x: C.ob), C.hom x x)
    (c:         Π (x y z:C.ob), C.hom x y -> C.hom y z -> C.hom x z)
    (HomSet:    Π (x y: C.ob), isSet (C.hom x y))
    (inv:       Π (x y: C.ob), C.hom x y -> C.hom y x)
    (inv-left:  Π (x y: C.ob) (p: C.hom x y),
                Ξ (C.hom x x) (c x y x p (inv x y p)) (id x))
    (inv-right: Π (x y: C.ob) (p: C.hom x y),
                Ξ (C.hom y y) (c y x y (inv x y p) p) (id y))
    (left:      Π (x y: C.ob) (f: C.hom x y),
                Ξ (C.hom x y) f (c x x y (id x) f))
    (right:     Π (x y: C.ob) (f: C.hom x y),
                Ξ (C.hom x y) f (c x y y f (id y)))
    (assoc:     Π (x y z w: C.ob) (f: C.hom x y)
                  (g: C.hom y z) (h: C.hom z w),
                Ξ (C.hom x w) (c x z w (c x y z f g) h)
                              (c x y w f (c y z w g h))), ⋆
```

[2]http://www.cse.chalmers.se/~coquand/Proposal.pdf

[3]Martin Hofmann and Thomas Streicher. The Groupoid Interpretation of Type Theory. 1996.

```
def isProp (A : U) : U
 := Π (a b : A), Ξ A a b

def isSet (A : U) : U
 := Π (a b : A) (x y : Ξ A a b),
    Ξ (Ξ A a b) x y

def isGroupoid (A : U) : U
 := Π (a b : A) (x y : Ξ A a b)
    (i j : Ξ (Ξ A a b) x y),
    Ξ (Ξ (Ξ A a b) x y) i j

def CatGroupoid (X : U) (G : isGroupoid X)
  : isCatGroupoid (PathCat X)
 := ( idp X,
      comp-Path X,
      G,
      sym X,
      comp-inv-Path⁻¹ X,
      comp-inv-Path X,
      comp-Path-left X,
      comp-Path-right X,
      comp-Path-assoc X,
      ⋆
    )

def comp-Ξ (A : U) (a b c : A) (p : Ξ A a b) (q : Ξ A b c) : Ξ A a c
 := <i> hcomp A (∂ i)
    (λ (j : I), [(i = 0) → a,
                 (i = 1) → q @ j]) (p @ i)

def comp-inv-Ξ⁻¹ (A : U) (a b : A) (p : Ξ A a b)
  : Ξ (Ξ A a a) (comp-Ξ A a b a p (<i> p @ -i)) (<_> a)
 := <k j> hcomp A (∂ j ∨ k)
    (λ (i : I), [(j = 0) → a,
                 (j = 1) → p @ -i ∧ -k,
                 (k = 1) → a]) (p @ j ∧ -k)

def comp-inv-Ξ (A : U) (a b : A) (p : Ξ A a b)
  : Ξ (Ξ A b b) (comp-Ξ A b a b (<i> p @ -i) p) (<_> b)
 := <j i> hcomp A (∂ i ∨ j)
    (λ (k : I), [(i = 0) → b,
                 (j = 1) → b,
                 (i = 1) → p @ j \/ k]) (p @ -i ∨ j)

def comp-Ξ-left (A : U) (a b : A) (p: Ξ A a b)
  : Ξ (Ξ A a b) p (comp-Ξ A a a b (<_> a) p)
 := <j i> hcomp A (∂ i ∨ -j)
    (λ (k : I), [(i = 0) → a,
                 (i = 1) → p @ k,
                 (j = 0) → p @ i /\ k]) a

def comp-Ξ-right (A : U) (a b : A) (p: Ξ A a b)
  : Ξ (Ξ A a b) p (comp-Ξ A a b b p (<_> b))
 := <j i> hcomp A (∂ i ∨ -j)
    (λ (k : I), [(i = 0) → a,
                 (i = 1) → b,
                 (j = 0) → p @ i]) (p @ i)
```

```
def comp–Ξ–assoc (A : U) (a b c d : A)
     (f : Ξ A a b) (g : Ξ A b c) (h : Ξ A c d)
  : Ξ (Ξ A a d) (comp–Ξ A a c d (comp–Ξ A a b c f g) h)
                (comp–Ξ A a b d f (comp–Ξ A b c d g h))
 := J A a (λ (a : A) (b : A) (f : Ξ A a b),
     Π (c d : A) (g : Ξ A b c) (h : Ξ A c d),
     Ξ (Ξ A a d) (comp–Ξ A a c d (comp–Ξ A a b c f g) h)
                 (comp–Ξ A a b d f (comp–Ξ A b c d g h)))
        (λ (c d : A) (g : Ξ A a c) (h : Ξ A c d),
        comp–Ξ (Ξ A a d)
            (comp–Ξ A a c d (comp–Ξ A a a c (<_> a) g) h)
            (comp–Ξ A a c d g h)
            (comp–Ξ A a a d (<_> a) (comp–Ξ A a c d g h))
            (<i> comp–Ξ A a c d (comp–Ξ–left A a c g @ –i) h)
            (comp–Ξ–left A a d (comp–Ξ A a c d g h))) b f c d g h
```

2.11 Homotopy Groups

Definition 50. (n-th Homotopy Group of m-Sphere).

$$\pi_n S^m = ||\Omega^n(S^m)||_0.$$

```
piS (n: nat): (m: nat) -> U = split
   zero   -> sTrunc (space (omega n (bool,false)))
   succ x -> sTrunc (space (omega n (Sn (succ x),north)))
```

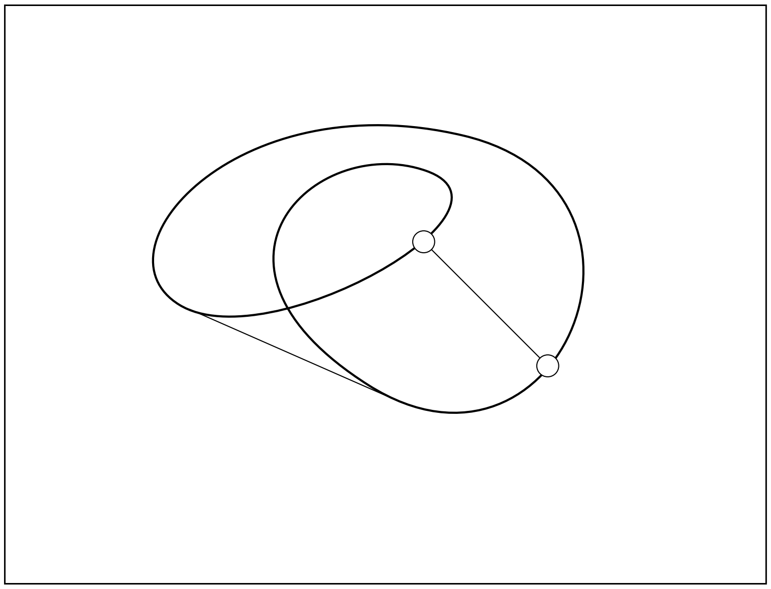
Theorem 17. ($\Omega(S^1) = \mathbb{Z}$).

```
data S1 = base
        | loop <i> [ (i=0) -> base ,
                     (i=1) -> base ]

loopS1 : U = Ξ S1 base base

encode (x:S1) (p:Ξ S1 base x)
  : helix x
  = subst S1 helix base x p zeroZ

decode : (x:S1) -> helix x -> Ξ S1 base x = split
  base -> loopIt
  loop @ i -> rem @ i where
    p : Ξ U (Z -> loopS1) (Z -> loopS1)
      = <j> helix (loop1@j) -> Ξ S1 base (loop1@j)
    rem : PathP p loopIt loopIt
      = corFib1 S1 helix (\(x:S1)->Ξ S1 base x) base
        loopIt loopIt loop1 (\(n:Z) ->
        comp (<i> Ξ loopS1 (oneTurn (loopIt n))
             (loopIt (testIsoPath Z Z sucZ predZ
                      sucpredZ predsucZ n @ i)))
             (<i>(lem1It n)@-i) [])

loopS1eqZ : Ξ U Z loopS1
  = isoPath Z loopS1 (decode base) (encode base)
    sectionZ retractZ
```

2.12 Hopf Fibrations

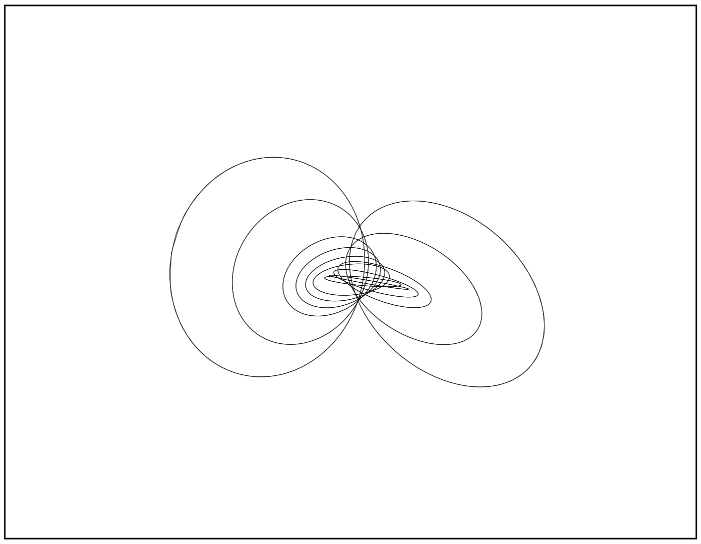
Example 2. (S^1 $\mathbb{R}$ Hopf Fiber).

```
data bool = false | true

negBool : bool -> bool
  = split { false -> true ; true -> false }

negBoolK : (b : bool) -> Ξ bool (negBool (negBool b)) b
  = split { false-><i>false;true-><i>true }

negBoolEquiv : equiv bool bool
  = (negBool,gradLemma bool bool negBool negBool negBoolK negBoolK)

S2 : U = susp S1
S3 : U = susp S2

ua (A B : U) (e : equiv A B) : Ξ U A B =
  <i> Glue B [ (i = 0) -> (A,e),
               (i = 1) -> (B,idEquiv B) ]

moebius : S1 -> U = split
  base -> bool
  loop @ i -> ua bool bool negBoolEquiv @ i

TH0 : U = (c : S1) * moebius c
```



Example 3. (S^3 $\mathbb{C}$ Hopf Fiber). S^3 Fibration was peeoneered by Guillaume Brunerie.

```
rot: (x : S1) -> Ξ S1 x x = split
     base -> loop1
     loop @ i -> constSquare S1 base loop1 @ i

mu : S1 -> equiv S1 S1 = split
     base -> idEquiv S1
     loop @ i -> equivPath S1 S1 (idEquiv S1)
                  (idEquiv S1) (<j> \(x : S1) -> rot x @ j) @ i

H : S2 -> U = split
     north -> S1
     south -> S1
     merid x @ i -> ua S1 S1 (mu x) @ i

total : U = (c : S2) * H c
```



Definition 51. (H-space). H-space over a carrier A is a tuple

$$H_A = \begin{cases} A : U \\ e : A \\ \mu : A \to A \to A \\ \beta : \Pi(a : A), \mu(e, a) = a \times \mu(a, e) = a \end{cases}$$

.

Theorem 18. (Hopf Invariant). Let $\phi : S^{2n-1} \to S^n$ a continuous map. Then homotopy pushout (cofiber) of ϕ is $cofib(\phi) = S^n \bigcup_\phi \mathbb{D}^{2n}$ has ordinary cohomology

$$H^k(\text{cofib}(\phi), \mathbb{Z}) = \begin{cases} \mathbb{Z} \; for \; k = n, 2n \\ \\ 0 \; otherwise \end{cases}$$

Theorem 19. (Four). There are fiber bundles: (S^0, S^1, p, S^1), (S^1, S^3, p, S^2), (S^3, S^7, p, S^4), (S^7, S^{15}, p, S^8).

Hence for α, β generators of the cohomology groups in degree n and $2n$, respectively, there exists an integer $h(\phi)$ that expresses the **cup product** square of α as a multiple of β — $\alpha \sqcup \alpha = h(\phi) \cdot \beta$. This integer $h(\phi)$ is called Hopf invariant of ϕ.

Theorem 20. (Adams, Atiyah). Hopf Fibrations are only maps that have Hopf invariant 1.