Volume I: Foundations

Introduction to Homotopy Type Theory

Issue I: Type Theory (Martin-Löf)
Issue II: Inductive Types (Coquand)
Issue III: Homotopy Type Theory (Awodey)
Issue IV: Higher Inductive Types (Lumsdaine)
Issue V: Modalities (Shulman)

Namdak Tonpa

2021 · Groupoid Infinity

I

Contents

Issue I: Martin-Löf Type Theory

Maksym Sokhatskyi

National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnical Institute
June 1, 2025

Abstract

Martin-Löf Type Theory (MLTT), introduced by Per Martin-Löf in 1972, is a cornerstone of constructive mathematics, providing a foundation for formalizing mathematical proofs and programming languages. Its 1973 variant, MLTT-73, incorporates dependent types (Π, Σ) and identity types (Id), with the J eliminator as a key construct for reasoning about equality. Historically, internalizing MLTT in a type checker while constructively proving the J eliminator has been challenging due to limitations in pure functional systems. This article presents a canonical formalization of MLTT-73 and its internalization (without η-rule for identity types due to gropoid interpretation) in **Per**, a dependent type theory language equipped with cubical type primitives. Using presented type theory, we constructively prove induction and computation MLTT-73 inference rules, including the J eliminator, and demonstrate suitability as a robust foundation for mathematical languages. We also provide logical, categorical, and homotopical interpretations of MLTT to contextualize its multidisciplinary significance. This work advances the mechanization of constructive mathematics and offers a blueprint for future type-theoretic explorations.

Keywords: Martin-Löf Type Theory, Cubical Type Theory.

Introduction to MLTT

For decades, type theorists have sought to fully internalize Martin-Löf Type Theory (MLTT) within a type checker, a task akin to building a self-verifying blueprint for mathematics.

Introduced by Per Martin-Löf in 1972 [2] MLTT-72 had only Π and Σ types. In 1973, a variant MLTT-73 with Id types was introduces with countable hierarchy of universes. In 1975, a variant of MLTT-75 with Π and Σ, Id, $+$, and $\mathbb{N}$ type was officially introduced [3] including infinite predicative hierarchy of universes.

Central to MLTT-73 is the J eliminator, a rule that governs how identity proofs are used, but its constructive derivation has long eluded pure functional type checkers due to the complexity of equality types. This article addresses this challenge by presenting a canonical formalization of MLTT-73 and its internalization in **Per**, a novel type theory language designed for constructive proofs.

Leveraging cubical type theory [14], this language incorporates Path types and universe polymorphism to faithfully embed MLTT-73 rules, achieving a constructive proof of the J eliminator. This internalization serves as an ultimate test of a type checker's robustness, verifying its ability to fuse and full coverage of introduction and elimination rules through beta and eta equalities.

To make MLTT accessible, we provide intuitive interpretations of its types: logical (as quantifiers), categorical (as functors), and homotopical (as spaces). These perspectives highlight MLTT's role as a bridge between mathematics and computation. Our work builds on Martin-Löf's vision of constructive mathematics, offering a minimal yet powerful framework for mechanized reasoning. We aim to inspire researchers and practitioners to explore type theory's potential in formalizing mathematics and designing reliable software.

Syntax of Per

The BNF consists of: i) telescopes (contexts) and definitions; ii) pure dependent type theory syntax; iii) identity system; iv) cubical face system; v) module system. It is slightly based on cubicaltt.

```
F = module I where L
L = ∅ | import I | def I T : O := O
T = ∅ | (I : O) T
O = I | U | (O)
  | Π (I : O), O | λ (I : O), O | O → O | O O
  | Ξ O O O O O | ⟨ O ⟩ O | O @ O | transp O O
  | 0 | 1 | -O | O ∧ O | O ∨ O | □
  | Σ (I : O), O | O .1 | O .2 | O , O
```

Here, = (definition), $\varnothing$ (empty set), | (vertical bar) — are parts of BNF language and $\langle$, $\rangle$, (,), :=, $\vee$, $\wedge$, -, $\rightarrow$, 0, 1, @, $\square$, **module**, **import**, **where**, **transp**, .**1**, .**2**, and , are terminals of the type checker language [1].

[1] `https://github.com/groupoid/per`

1 Interpretations

1.1 Type Theory

In MLTT, types are defined by five classes of rules: (1) *formation*, specifying the type's signature; (2) *introduction*, defining constructors for its elements; (3) *elimination*, providing a dependent induction principle; (4) *computation* (beta-equality), governing reduction; and (5) *uniqueness* (eta-equality), ensuring canonical forms, though the latter is absent for identity types in homotopical settings.

For MLTT-73, we focus on Π (dependent function types), Σ (dependent pair types), and Id (identity types). MLTT-72 provided the foundational Π and Σ types, lacking mechanisms for equality, which MLTT-73 introduced via Id types, originally assuming uniqueness of identity proofs (UIP) in some contexts [3]. In cubical type theory, Id types are replaced by **Path** types, defined as functions from an interval $[0, 1]$, making the J eliminator computationally effective and supporting constructive proofs [1]. The identity type, introduced in MLTT-73 and refined in [3], is significant for enabling constructive equality reasoning.

Modern homotopical interpretations, pioneered by Hofmann and Streicher [6], refute UIP, adopting **Path** types that model equality as paths in a space, aligning with cubical type theory's constructive framework [14]. This shift, integral to MLTT-75, facilitates the internalization of MLTT-73 rules.

Type checkers operate within contexts, binding variables to indexed universes, built-in types, or user-defined types via de Bruijn indices (to avoid variable capture) or names (for user-friendly proof assistants). These contexts, central to MLTT implementations, enable queries about type derivability and code extraction, forming the core of type checkers. As shown in Table 1 MLTT-75 unifies these constructs across multiple domains.

1.2 Logic

The logical interpretation casts MLTT-75 as a system for intuitionistic higher-order logic, where types correspond to propositions and terms to proofs, embodying the Curry-Howard correspondence. In this view, a type A represents a proposition, and a term $a : A$ is a proof of A. The Π-type, $\prod_{x:A} B(x)$, encodes universal quantification ($\forall x : A, B(x)$), while the Σ-type, $\sum_{x:A} B(x)$, represents existential quantification ($\exists x : A, B(x)$). The identity type, $\mathrm{Id}_A(a, b)$, captures propositional equality ($a =_A b$), with the J eliminator providing a constructive means to reason about equalities.

Each type's five rules (formation, introduction, elimination, computation, and uniqueness, except for Id in cubical settings) mirror the structure of logical inference rules. For instance, the introduction rule for Π constructs a lambda term (proof of a universal statement), while its elimination rule applies the term to an argument (using the universal statement).

MLTT-73 is not standalone framework for constructive mathematics but rather the extended foundational core on top of MLTT-72. Adding **0** (Empty),

Type Theory	Logic	Category Theory	Homotopy Theory
A type	class	object	space
isProp A	proposition	(-1)-truncated object	space
a:A program	proof	generalized element	point
$B(x)$	predicate	indexed object	fibration
$b(x) : B(x)$	conditional proof	indexed elements	section
$\mathbf{0}$	$\perp$ false	initial object	empty space
$\mathbf{1}$	$\top$ true	terminal object	singleton
$\mathbf{2}$	boolean	subobject classifier	$\mathbb{S}^0$
$A + B$	$A \vee B$ disjunction	coproduct	coproduct space
$A \times B$	$A \wedge B$ conjunction	product	product space
$A \to B$	$A \Rightarrow B$	internal hom	function space
$\sum x : A, B(x)$	$\exists_{x:A} B(x)$	dependent sum	total space
$\prod x : A, B(x)$	$\forall_{x:A} B(x)$	dependent product	space of sections
$\mathbf{Path}_A$	equivalence $=_A$	path space object	path space A^I
quotient	equivalence class	quotient	quotient
W-type	induction	colimit	complex
type of types	universe	object classifier	universe
quantum circuit	proof net	string diagram	

1 (Unit) types allows resulting type system to internalize intuitionistic propositional logic (IPL), extending further with **2** (Bool) it can encode classical logic with the rule of excluded middle (CPL) [10].

1.3 Category Theory

The categorical interpretation models MLTT-75 within category theory, where types are objects, terms are morphisms, and type constructions are functors. This perspective, formalized by Cartmell and Seely [13], views MLTT-75 with **0**, **1**, **2** types as a locally cartesian closed category (LCCC) with boolean as subobject classifiers forming boolean topoi. Here, Π-types correspond to dependent products (right adjoints to base change functors), and Σ-types to dependent sums (left adjoints). The identity type, Id_A, is modeled as a path space object, reflecting equality as a morphism.

For example, given a morphism $f : A \to B$ in a category, the Π_f functor maps a dependent type over B to one over A, generalizing function spaces, while Σ_f constructs the total space of a fibration.

1.4 Homotopy Theory

The homotopical interpretation, a breakthrough in modern type theory, views MLTT-73 types as spaces and terms as points, with identity types as paths. Introduced by Hofmann and Streicher's groupoid model [6], this perspective refutes the uniqueness of identity proofs (UIP) in classical MLTT-73, replacing Id with Path types that model equality as continuous paths in a space. In cubical type theory, Path types are functions from an interval $[0, 1]$ to a type, enabling constructive proofs of MLTT-73 rules, including the J eliminator.

Here, Π-types represent spaces of sections, Σ-types denote total spaces of fibrations, and Path types form path spaces (A^I). This interpretation connects MLTT-73 to homotopy theory, where types are ∞-groupoids, and fibrations (dependent types) are studied geometrically. For instance, a Π-type can be seen as a trivial fiber bundle, with its introduction rule constructing a section [1].

Set Theory

The set-theoretical interpretation models MLTT-75's types as sets and terms as elements, aligning with classical first-order logic. In this view, a type A is a set, and a term $a : A$ is an element. The Π-type represents a set of functions, Σ-type a disjoint union of sets, and $\mathrm{Id}_A(a, b)$ an equality relation. However, this interpretation is limited, as it cannot capture higher equalities (e.g., paths between paths) or inductive types directly, due to its 0-truncated nature [1].

2 Dependent Type Theory

2.1 Dependent Product (Π)

Π is a dependent product type, the generalization of functions. As a function it can serve the wide range of mathematical constructions as its domain and codomain, which are in general: objects, types, or spaces; and could have as its instance: sets, functions, polynomial functors, infinitesimals, ∞-groupoids, topological ∞-groupoid, CW-complexes, categories, languages, etc.

At this light there could be many interpretation of Π types from different areas of mathematics. We give here three: i) logical interpretation of Π as ∀ quantifier from higher order logic that forms a ground of type theory; ii) geomeric intepretation of Π as fiber bundle; iii) categorical interpretation of functions as functors.

Type-theoretical interpretation

As a logical system dependent type theory could correspond to higher order logic. However here only type-theoretical model is given completely.

Definition 1 (Π-Formation)**.** Π-types represents the way we create the spaces of dependent functions $f : \Pi(x : A), B(x)$ with domain in A and codomain in type family $B : A \to U$ over A.

$$\Pi(A, B) : U =_{def} \prod_{A:U} \prod_{B:A\to U} \prod_{x:A} B(x).$$

```
def Pi (A : U) (B : A → U) : U := Π (x : A), B x
```

Definition 2 (Π-Introduction)**.** Lambda constructor defines a new lambda function in the space of dependent functions. It is called lambda abstraction and displayed as $\lambda x.b(x)$ or $x \mapsto b(x)$.

$$\lambda(x : A), b(x) : \Pi(A, B) =_{def}$$

$$\prod_{A:U} \prod_{B:A\to U} \prod_{b:\Pi(A,B)} \lambda x, b_x.$$

```
def lambda (A: U) (B: A → U) (b: Pi A B) : Pi A B := λ (x : A), b x
def lam (A B: U) (f: A → B) : A → B := λ (x : A), f x
```

When codomain is not dependent on valude from domain the function $f : A \to B$ is studied in System F_ω, dependent case in studied in Systen P_ω or Calculus of Construction (CoC).

Definition 3 (Π-Induction Principle)**.** States that if predicate holds for lambda function then there is a function from function space to the space of predicate.

```
def Π-ind (A : U) (B : A -> U) (C : Pi A B → U) (g: Π (x: Pi A B), C x)
  : Π (p: Pi A B), C p := λ (p: Pi A B), g p
```

Definition 4 (Π-Elimination)**.** Application reduces the term by using recursive substitution.

$$f\ a : B(a) =_{def} \prod_{A:U} \prod_{B:A\to U} \prod_{a:A} \prod_{f:\prod_{x:A} B(a)} f(a).$$

```
def apply (A: U) (B: A → U) (f: Pi A B) (a: A) : B a := f a
def app (A B: U) (f: A → B) (x: A) : B := f x
```

Theorem 1 (Π-Composition)**.** Composition is using application of appropriate singnatures.

$$f(a) =_{B(a)} (\lambda(x : A) \to f(a))(a).$$

```
def ∘ᵀ (α β γ: U) : U
 := (β → γ) → (α → β) → (α → γ)

def ∘ (α β γ : U) : ∘ᵀ α β γ
 := λ (g: β → γ) (f: α → β) (x: α), g (f x)
```

Theorem 2 (Π-Computation)**.** β-rule shows that composition lam ∘ app could be fused.

$$f(a) =_{B(a)} (\lambda(x : A) \to f(a))(a).$$

```
def Π-β (A : U) (B : A → U) (a : A) (f : Pi A B)
  : Path (B a) (apply A B (lambda A B f) a) (f a)
 := idp (B a) (f a)
```

Theorem 3. (Π-Uniqueness). η-rule shows that composition app ∘ lam could be fused.

$$f =_{(x:A)\to B(a)} (\lambda(y : A) \to f(y)).$$

```
def Π-η (A : U) (B : A → U) (a : A) (f : Pi A B)
  : Path (Pi A B) f (λ (x : A), f x)
 := idp (Pi A B) f
```

Categorical interpretation

The adjoints Π and Σ is not the only adjoints could be presented in type system. Axiomatic cohesions could contain a set of adjoint pairs as a core type checker operations.

Definition 5 (Dependent Product)**.** The dependent product along morphism $g : B \to A$ in category C is the right adjoint $\Pi_g : C_{/B} \to C_{/A}$ of the base change functor.

Definition 6 (Space of Sections)**.** Let $\mathbf{H}$ be a $(\infty, 1)$-topos, and let $E \to B : \mathbf{H}_{/B}$ a bundle in $\mathbf{H}$, object in the slice topos. Then the space of sections $\Gamma_\Sigma(E)$ of this bundle is the Dependent Product:

$$\Gamma_\Sigma(E) = \Pi_\Sigma(E) \in \mathbf{H}.$$

Theorem 4 (Homotopy Equivalence)**.** If fiber space is set for all base, and there are two functions $f, g : (x : A) \to B(x)$ and two homotopies between them, then these homotopies are equal.

```
def setPi (A: U) (B: A -> U)
    (h: Π (x: A), isSet (B x)) (f g: Pi A B)
    (p q: Path (Pi A B) f g)
  : Path (Path (Pi A B) f g) p q
```

Theorem 5 (Contractability)**.** If domain and codomain is contractible then the space of sections is contractible.

```
def piIsContr (A: U) (B: A -> U) (u: isContr A)
    (q: Π (x: A), isContr (B x))
  : isContr (Pi A B)
```

Definition 7 (Section)**.** A section of morphism $f : A \to B$ in some category is the morphism $g : B \to A$ such that $f \circ g : B \xrightarrow{g} A \xrightarrow{f} B$ equals the identity morphism on B.

Homotopical interpretation

Geometrically, Π type is a space of sections, while the dependent codomain is a space of fibrations. Lambda functions are sections or points in these spaces, while the function result is a fibration. Π type also represents the cartesian family of sets, generalizing the cartesian product of sets.

Definition 8. (Fiber). The fiber of the map $p : E \to B$ in a point $y : B$ is all points $x : E$ such that $p(x) = y$.

Definition 9. (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 following property: for any point $y : B$ exists a neighborhood U_b for which 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 10. (Cartesian Product of Family over B). Is a set F of sections of the bundle with elimination map $app : F \times B \to E$ such that

$$F \times B \xrightarrow{app} E \xrightarrow{pr_1} B \tag{1}$$

pr_1 is a product projection, so pr_1, app are morphisms of slice category $Set_{/B}$. The universal mapping property of F: for all A and morphism $A \times B \to E$ in $Set_{/B}$ exists unique map $A \to F$ such that everything commute. So a category with all dependent products is necessarily a category with all pullbacks.

Definition 11 (Trivial Fiber Bundle)**.** When total space E is cartesian product $\Sigma(B, F)$ and $p = pr_1$ then such bundle is called trivial $(F, \Sigma(B, F), pr_1, B)$.

Theorem 6 (Functions Preserve Paths)**.** For a function $f : (x : A) \to B(x)$ there is an $ap_f : x =_A y \to f(x) =_{B(x)} f(y)$. This is called application of f to path or congruence property (for non-dependent case — *cong* function). This property behaves functoriality as if paths are groupoid morphisms and types are objects.

Theorem 7 (Trivial Fiber Bundle equals Family of Sets)**.** Inverse image (fiber) of fiber bundle $(F, B * F, pr_1, B)$ in point $y : B$ equals $F(y)$.

```
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
 := λ (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
 := λ (x : F y), ((y, x), idp B y)

def decode-encode-Pi (B : U) (F : B → U) (y : B) (x : F y)
```

```
  : Path (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)
  : Path (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)
```

2.2 Dependent Sum (Σ)

Σ-type is a space that contains dependent pairs where type of the second element depends on the value of the first element. As only one point of fiber domain present in every defined pair, Σ-type is also a dependent sum, where fiber base is a disjoint union.

Σ is a dependent sum type, the generalization of products. Σ type is a total space of fibration. Element of total space is formed as a pair of basepoint and fibration.

Spaces of dependent pairs are using in type theory to model cartesian products, disjoint sums, fiber bundles, vector spaces, telescopes, lenses, contexts, objects, algebras, $\exists$-type, etc.

Type-theoretical interpretation

Definition 12 (Σ-Formation)**.** The dependent sum type is indexed over type A in the sense of coproduct or disjoint union, where only one fiber codomain $B(x)$ is present in pair.

$$\Sigma(A,B) : U =_{def} \prod_{A:U} \prod_{B:A\to U} \sum_{x:A} B(x).$$

```
def Sigma (A: U) (B: A → U) : U := Σ (x: A), B(x)
```

Definition 13 (Σ-Introduction)**.** The dependent pair constructor is a way to create indexed pair over type A in the sense of coproduct or disjoint union.

$$\textbf{pair} : \Sigma(A,B) =_{def} \prod_{A:U} \prod_{B:A\to U} \prod_{a:A} \prod_{b:B(a)} (a,b).$$

```
def pair (A: U) (B: A → U) (a: A) (b: B a) : Sigma A B := (a, b)
```

Definition 14 (Σ-Elimination)**.** The dependent projections $pr_1 : \Sigma(A,B) \to A$ and $pr_2 : \Pi_{x:\Sigma(A,B)} B(pr_1(x))$ are pair deconstructors.

$$\mathbf{pr}_1 : \prod_{A:U} \prod_{B:A\to U} \prod_{x:\Sigma(A,B)} A =_{def} .1 =_{def} (a,b) \mapsto a.$$

$$\mathbf{pr}_2 : \prod_{A:U} \prod_{B:A\to U} \prod_{x:\Sigma(A,B)} B(x.1) =_{def} .2 =_{def} (a,b) \mapsto b.$$

```
def pr₁ (A: U) (B: A → U) (x: Sigma A B) : A := x.1
def pr₂ (A: U) (B: A → U) (x: Sigma A B) : B (pr₁ A B x) := x.2
```

Definition 15 (Σ-Induction)**.** States that if predicate holds for two projections then predicate holds for total space.

```
def Σ-ind (A : U) (B : A -> U)
      (C : Π (s: Σ (x: A), B x), U)
      (g: Π (x: A) (y: B x), C (x,y))
      (p: Σ (x: A), B x)
   : C p := g p.1 p.2
```

Theorem 8 (Σ-Computation)**.**
```
def Σ-β₁ (A : U) (B : A → U) (a : A) (b : B a)
   : Path A a (pr₁ A B (a ,b)) := idp A a

def Σ-β₂ (A : U) (B : A → U) (a : A) (b : B a)
   : Path (B a) b (pr₂ A B (a, b)) := idp (B a) b
```

Theorem 9 (Σ-Uniqueness)**.**
```
def Σ-η (A : U) (B : A → U) (p : Sigma A B)
   : Path (Sigma A B) p (pr₁ A B p, pr₂ A B p)
  := idp  (Sigma A B) p
```

Categorical interpretation

Definition 16. (Dependent Sum). The dependent sum along the morphism $f : A \to B$ in category C is the left adjoint $\Sigma_f : C_{/A} \to C_{/B}$ of the base change functor.

Set-theoretical interpretation

Theorem 10. (Axiom of Choice). If for all $x : A$ there is $y : B$ such that $R(x,y)$, then there is a function $f : A \to B$ such that for all $x : A$ there is a witness of $R(x, f(x))$.

```
def ac (A B: U) (R: A -> B -> U)
      (g: Π (x: A), Σ (y: B), R x y)
   : Σ (f: A -> B), Π (x: A), R x (f x)
  := (\(i:A),(g i).1,\(j:A),(g j).2)
```

Theorem 11. (Total). If fiber over base implies another fiber over the same base then we can construct total space of section over that base with another fiber.

```
def total (A:U) (B C : A -> U)
     (f : Π (x:A), B x -> C x)
     (w: Σ(x: A), B x)
  : Σ (x: A), C x := (w.1,f (w.1) (w.2))
```

2.3 Path Space (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.

This ctt file reflects [2]CCHM cubicaltt model with connections. For [3]ABCFHL yacctt model with variables please refer to ytt file. You may also want to read [4]BCH, [5]AFH. There is a [6]PO paper about CCHM axiomatic in a topos.

Chosing flavour of normal forms for identity system

Here we give brief description of structure inside path spaces:

Bounded Distributive Lattice: A bounded distributive lattice in HoTT is a type $L : \mathcal{U}$ equipped with binary operations $\wedge : L \to L \to L$, $\vee : L \to L \to L$, and constants $0 : L$, $1 : L$, satisfying associativity ($a \wedge (b \wedge c) \equiv (a \wedge b) \wedge c$, $a \vee (b \vee c) \equiv (a \vee b) \vee c$), commutativity ($a \wedge b \equiv b \wedge a$, $a \vee b \equiv b \vee a$), idempotence ($a \wedge a \equiv a$, $a \vee a \equiv a$), absorption ($a \wedge (a \vee b) \equiv a$, $a \vee (a \wedge b) \equiv a$), distributivity ($a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c)$, $a \vee (b \wedge c) \equiv (a \vee b) \wedge (a \vee c)$), and bounds ($a \wedge 0 \equiv 0$, $a \vee 1 \equiv 1$). In a Boolean topos, L corresponds to the type of subobjects with $\wedge \equiv \times$, $\vee \equiv +$, $0 \equiv \bot$, $1 \equiv \top$.

De Morgan Algebra: A De Morgan Algebra in HoTT is a bounded distributive lattice $(L, \wedge, \vee, 0, 1) : \mathcal{U}$ equipped with a unary operation $\neg : L \to L$ satisfying De Morgan's Laws ($\neg(a \wedge b) \equiv \neg a \vee \neg b$, $\neg(a \vee b) \equiv \neg a \wedge \neg b$) and involution ($\neg\neg a \equiv a$). The type L models propositions with a negation operation preserving these equivalences, and in a Boolean topos, $L \cong 2 = \{\mathrm{true}, \mathrm{false}\}$ forms a Boolean algebra, satisfying De Morgan's Laws as isomorphisms.

Heyting Algebra: A Heyting Algebra in HoTT is a bounded distributive lattice $(L, \wedge, \vee, 0, 1) : \mathcal{U}$ equipped with an implication operation $\to : L \to L \to L$

[2]Cyril Cohen, Thierry Coquand, Simon Huber, Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom. 2015. `https://5ht.co/cubicaltt.pdf`

[3]Carlo Angiuli, Brunerie, Coquand, Kuen-Bang Hou (Favonia), Robert Harper, Dan Licata. Cartesian Cubical Type Theory. 2017. `https://5ht.co/cctt.pdf`

[4]Marc Bezem, Thierry Coquand, Simon Huber. A model of type theory in cubical sets. 2014. `http://www.cse.chalmers.se/~coquand/mod1.pdf`

[5]Carlo Angiuli, Kuen-Bang Hou (Favonia), Robert Harper. Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. 2018. `https://www.cs.cmu.edu/~cangiuli/papers/ccctt.pdf`

[6]Andrew Pitts, Ian Orton. Axioms for Modelling Cubical Type Theory in a Topos. 2016. `https://arxiv.org/pdf/1712.04864.pdf`

such that, for all $a, b, c : L$, there is an equivalence $a \leq b \to c \iff a \wedge b \leq c$, where $\leq$ is the partial order defined by $a \leq b \iff a \wedge b \equiv a$. Negation is defined as $\neg a \equiv a \to 0$, and modus ponens holds: given $a : A$ and $f : A \to B$, there exists $f a : B$. In a Boolean topos, the Heyting algebra becomes a Boolean algebra, with $\to$ corresponding to the exponential B^A.

Boolean Algebra: A Boolean Algebra in HoTT is a De Morgan Algebra $(L, \wedge, \vee, \neg, 0, 1) : \mathcal{U}$ satisfying the law of excluded middle ($a \vee \neg a \equiv 1$) and non-contradiction ($a \wedge \neg a \equiv 0$). The type $L \cong 2 = \{\text{true}, \text{false}\}$ models classical propositions, with a mandatory Boolean type in a Boolean topos, where L is the subobject classifier $\Omega \cong 2$, and all operations correspond to classical logical connectives.

In **Per** De Morgan algebra is used (CCHM flavour).

Type-theoretical interpretation

Definition 17 (Path Formation)**.**

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

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

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

Definition 18 (Path Introduction)**.**

$$\mathbf{idp} : x \equiv_A x =_{def} \prod_{A:U} \prod_{x:A} [i]x.$$

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

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 $[i]x$.

Definition 19 (Path Application)**.** You can apply face to path.

```
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 20 (Path Composition)**.** Composition operation allows to build a new path by given to paths in a connected point.

$$\begin{tikzcd}
a \arrow[r, "comp"] & c \\
a \arrow[u, "\lambda(i:I) \to a"] \arrow[r, "p@i"] & b \arrow[u, "q"']
\end{tikzcd}$$

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

Theorem 12 (Path Inversion)**.**

```
def inv (A: U) (a b: A) (p: Path A a b) : Path A b a := <i> p @ −i
```

Definition 21 (Connections)**.** Connections allows you to build square with given only one element of path: i) $\lambda\ (i,j : I) \rightarrow p\ @\ min(i,j)$; ii) $\lambda\ (i,j : I) \rightarrow p\ @\ max(i,j)$.

$$
\begin{tikzcd}
a \arrow[r, "p"] & b \\
a \arrow[u, "\lambda\ (i:I) \rightarrow a"] \arrow[r, "\lambda\ (i:I) \rightarrow a"'] & a \arrow[u, "p"']
\end{tikzcd}
\qquad
\begin{tikzcd}
b \arrow[r, "\lambda\ (i:I) \rightarrow b"] & b \\
a \arrow[u, "p"] \arrow[r, "p"] & b \arrow[u, "\lambda\ (i:I) \rightarrow b"']
\end{tikzcd}
$$

```
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)

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)
```

Theorem 13 (Congruence)**.** Is a map between values of one type to path space of another type by an encode function between types. Implemented as lambda defined on $[0,1]$ that returns application of encode function to path application of the given path to lamda argument $\lambda(i : I), f(p@i)$ for both cases.

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

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

```
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)
```

Theorem 14 (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.

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

$$\prod_{A:I\rightarrow U} \prod_{r:I} \lambda x, \mathbf{transp}([i]A(i), 0, x).$$

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

def transp-U (A B: U) (p : PathP (<_>U) A B) (i: I)
 := transp (<i> (\(_:U),U) (p @ i)) i A
```

Definition 22 (Singleton). `def singl (A: U) (a: A): U := Σ (x: A), Ξ A a x`

Theorem 15 (Singleton Instance). `def eta (A: U) (a: A): singl A a := (a, idp A a)`

Theorem 16 (Singleton Contractability). `def contr (A : U) (a b : A) (p : Ξ A a b)`

```
   : Ξ (singl A a) (eta A a) (b, p)
 := <i> (p @ i, <j> p @ i /\ j)
```

Theorem 17 (Path Elimination). `def subst (A : U) (P : A -> U) (a b : A)`

```
    (p : Ξ A a b) (e : P a) : P b
 := transp (<i> P (p @ i)) 0 e

def D (A : U) : U₁
 := Π (x y : A), Path A x y → U

def J (A: U) (x: A) (C: D A) (d: C x x (idp A x))
    (y: A) (p: Ξ A x y) : C x y p
 := subst (singl A x) (\ (z: singl A x), C x (z.1) (z.2))
    (eta A x) (y, p) (contr A x y p) d
```

Theorem 18. (Path Computation).

```
def trans_comp (A : U) (a : A)
   : Ξ A a (transport A A (<i> A) a)
 := <j> transp (<_> A) -j a

def subst-comp (A: U) (P: A → U) (a: A) (e: P a)
   : Ξ (P a) e (subst A P a a (idp A a) e)
 := trans_comp (P a) e

def J-β (A : U) (a : A) (C : D A) (d: C a a (idp A a))
   : Ξ (C a a (idp A a)) d (J A a C d a (idp A a))
 := subst-comp (singl A a)
    (\ (z: singl A a), C a (z.1) (z.2)) (eta A a) d
```

Note that Path type has no Eta rule due to groupoid interpretation.

Groupoid interpretation

The groupoid interpretation of type theory is well known article by Martin Hofmann and Thomas Streicher, more specific interpretation of identity type as infinity groupoid [6].

2.4 Universes ($\mathbf{U}_i$)

In Martin-Löf Type Theory (MLTT), universes are types that classify other types, forming a cumulative hierarchy to manage type formation and avoid paradoxes like Russell's. MLTT-73 adopts a predicative hierarchy of universes, denoted U_i for $i \in \mathbb{N}$, where each universe U_i is a type in the next universe U_{i+1}.

This section defines the universe hierarchy constructively, specifying formation, introduction, and computation rules, and illustrates their encoding in **Per**.

Definition 23 (Universe Formation)**.** For each natural number $i \in \mathbb{N}$, there exists a universe U_i, which is a type classifying small types at level i. The formation rule is: $\Gamma \vdash U_i : U_{i+1}$. Universes are introduced as constructors, with each U_i inhabiting U_{i+1}.

```
def U (i : Nat) : U (suc i)
```

Definition 24 (Universe Introduction)**.** A type A belongs to a universe U_i if it can be derived as a type at level i. For MLTT-73, this includes base types (e.g., Π, Σ, Path), user-defined types, and universes U_j for $j < i$. The introduction rule is: $\Gamma; A \vdash A : U_i$, where i is the minimal level such that $A \in U_i$. Types like $\Pi(A, B)$, $\Sigma(A, B)$, and $\Xi(A, x, y)$ are explicitly landed in a universe:

```
def Pi (A : U i) (B : A -> U i) : U i := Π (x : A), B x
def Sigma (A : U i) (B : A -> U i) : U i := Σ (x : A), B x
def Path (A : U i) (x y : A) : U i := PathP (<_> A) x y
```

Definition 25 (Cumulative Hierarchy)**.** The universe hierarchy is cumulative, meaning if $A : U_i$, then $A : U_j$ for all $j > i$. This ensures flexibility in type checking, as types can be lifted to higher universes. This is implicit in the type checker's ability to assign types to higher universes when needed.

Definition 26 (Predicative Rules)**.** The formation of dependent types (e.g., Π, Σ) lands in the maximum of the universe levels of its constituents. For example, for Π-types: $\Gamma \vdash A : U_i$ and $\Gamma, x : A \vdash B(x) : U_j$ we can derive $\Gamma \vdash \Pi(x : A), B(x) : U_{\max(i,j)}$ This predicative rule ensures that the universe level reflects the highest level of the domain or codomain.

```
def Level (i j : N) (A : U i) (B : A -> U j)
  : U (max i j) := Π (x : A), B x
```

Similar rules apply to Σ and Path types, ensuring all MLTT-73 types are predicatively landed.

Definition 27 (Definitional Equality)**.** Universes support definitional equality, where two types $A, B : U_i$ are equal if their normalized forms are identical. This is crucial for type checking in MLTT-73.

Contexts

In Martin-Löf Type Theory (MLTT), contexts define the typing environment for judgments, consisting of a sequence of typed variable declarations that enable the derivation of types and terms.

Context as metatheoretical entity couldn't be internalized but could be imagined as telescopes, ensuring well-formedness and supporting constructive type checking. Explicit context rendering could be seen in categorical interpretation of dependent type theory

Definition 28 (Empty Context)**.** The empty context contains no variable declarations and serves as the base case for context formation. It is represented as the unit type, indicating an empty telescope:

$$\gamma_0 : \Gamma =_{def} \star.$$

Definition 29 (Context Comprehension)**.** A context is extended by adding a variable declaration for a type dependent on the existing context. For a context Γ and a type A over Γ, the extended context is:

$$\Gamma; A =_{def} \sum_{\gamma:\Gamma} A(\gamma).$$

This is encoded as a dependent pair, binding a variable to a type in the context.

Definition 30 (Context Derivability)**.** A type A is derivable in a context Γ if it can be assigned to a universe given the variables in Γ:

$$\Gamma \vdash A =_{def} \prod_{\gamma:\Gamma} A(\gamma).$$

This corresponds to a dependent function type, ensuring A is well-typed across all context elements: For terms, a term $t : A$ in Γ, written $\Gamma \vdash t : A$, is derivable if it respects the context's bindings.

Definition 31 (Terms)**.** A term is an element of a type within a context. Given $\Gamma \vdash A : U_i$, a term t satisfies $\Gamma \vdash t : A$. Terms include variables, constructors (e.g., λ for Π, pairs for Σ), and applications, defined by MLTT-73's syntax.

Contexts provide a structured environment for deriving judgments. They integrate with the any reasoning framework, supporting and ensuring sequential constructive verification.

MLTT-73

Here is given formal model of type-theoretical interpretation of Martin-Löf Type Theory. It combines 4 Path rules (no eta), 5 Π rules, and 6 Σ rules (two elims). The proof is provided by direct embedding (internalizing) the model intro the model of type checker which is even more powerful.

Definition 32 (MLTT-73 Reality Check)**.** The MLTT as a Type is defined by taking all rules for Π, Σ and Path types into one Σ telescope or context.

```
def MLTT–73 (A : U) : U₁ :=
  Σ (Π–form   : Π (B : A → U), U)
    (Π–ctor₁  : Π (B : A → U), Pi A B → Pi A B)
    (Π–elim₁  : Π (B : A → U), Pi A B → Pi A B)
    (Π–comp₁  : Π (B : A → U) (a : A) (f : Pi A B),
                Ξ (Ξ (B a) (Π–elim₁ B (Π–ctor₁ B f) a) (f a))
    (Π–comp₂  : Π (B : A → U) (a : A) (f : Pi A B),
                Ξ (Pi A B) f (λ (x : A), f x))
    (Σ–form   : Π (B : A → U), U)
    (Σ–ctor₁  : Π (B : A → U) (a : A) (b : B a) , Sigma A B)
    (Σ–elim₁  : Π (B : A → U) (p : Sigma A B), A)
    (Σ–elim₂  : Π (B : A → U) (p : Sigma A B), B (pr₁ A B p))
    (Σ–comp₁  : Π (B : A → U) (a : A) (b: B a),
                Ξ A a (Σ–elim₁ B (Σ–ctor₁ B a b)))
    (Σ–comp₂  : Π (B : A → U) (a : A) (b: B a),
                Ξ (B a) b (Σ–elim₂ B (a, b)))
    (Σ–comp₃  : Π (B : A → U) (p : Sigma A B),
                Ξ (Sigma A B) p (pr₁ A B p, pr₂ A B p))
    (=–form   : Π (a : A), A → U)
    (=–ctor₁  : Π (a : A), Path A a a)
    (=–elim₁  : Π (a : A) (C: D A) (d: C a a (=–ctor₁ a))
                  (y: A) (p: Path A a y), C a y p)
    (=–comp₁  : Π (a : A) (C: D A) (d: C a a (=–ctor₁ a)),
                Ξ (C a a (=–ctor₁ a)) d
                  (=–elim₁ a C d a (=–ctor₁ a))), 1
```

Theorem 19. (Model Check). There is an instance of MLTT.

```
def internalizing (A : U) : MLTT A
 := ( Pi A, Π–lambda A, Π–apply A, Π–β A, Π–η A,
      Sigma A, pair A, pr₁ A, pr₂ A, Σ–β₁ A, Σ–β₂ A, Σ–η A,
      Path A, idp A, J A, J–β A, ⋆ )
```

The result of the work is a `mltt.ctt` file which can be runned using `cubicaltt`. Note that MLTT-73 internalization includes only eliminator and computational rule for identity system (without uniqness rule), as cubical Path spaces refute uniqueness of identity proofs.

Conclusions

This article presents a landmark achievement in type theory: the constructive internalization of Martin-Löf Type Theory (MLTT-73) computational rules within the **Per** language, a minimal type system equipped with cubical type theory primitives.

This internalization, formalized also in the `mltt.ctt` for double checking, validates MLTT-73 in `cubicaltt`, providing a rigorous test of a type checker's ability to fuse introduction and elimination rules through computational and uniqueness equations.

The significance of this work lies in its constructive approach to the J eliminator, a cornerstone of MLTT-73 identity type, which previous internalization attempts failed to derive constructively [3, 10]. By leveraging cubical type

Language	U^n	Π	Σ	Id	Ξ	$\mathbb{N}$	0/1/2	W	Ind
Systen P_ω (CoC-88)		x							
MLTT-72		x	x						
Henk (ECC)	x	x							
Errett (LCCC/IPL)	x	x	x				x		
MLTT-73	x	x	x	x					
Per	x	x	x		x				
MLTT-75	x	x	x	x		x	x		
MLTT-80	x	x	x	x			x	x	
Anders (HTS)	x	x	x	x	x		x	x	
Frank (CoC+CIC)	x	x							x
Christine (Coq)	x	x	x	x					x
cubicaltt		x	x		x				x
Agda	x	x	x	x	x				x
Lean	x	x	x	x					x
NuPRL		x	x	x					x

theory's Path types and operations (e.g., connections, compositions), the type checker achieves a compact foundational core for verifying mathematics.

The article also elucidates MLTT-73 versatility through logical, categorical, homotopical, and set-theoretical interpretations, offering a comprehensive landscape for researchers and newcomers to type theory.

References

[1] Vladimir Voevodsky et al., *Homotopy Type Theory*, in *Univalent Foundations of Mathematics*, 2013.

[2] Per Martin-Löf and Giovanni Sambin, *The Theory of Types*, in *Studies in Proof Theory*, 1972.

[3] Per Martin-Löf, *An Intuitionistic Theory of Types: Predicative Part*, in *Studies in Logic and the Foundations of Mathematics*, vol. 80, pp. 73–118, 1975. `doi:10.1016/S0049-237X(08)71945-1`

[4] Per Martin-Löf and Giovanni Sambin, *Intuitionistic Type Theory*, in *Studies in Proof Theory*, 1984.

[5] Thierry Coquand and Gérard Huet, *The Calculus of Constructions*, in *Information and Computation*, pp. 95–120, 1988. `doi:10.1016/0890-5401(88)90005-3`

[6] Martin Hofmann and Thomas Streicher, *The Groupoid Interpretation of Type Theory*, in *Venice Festschrift*, Oxford University Press, pp. 83–111, 1996.

[7] Claudio Hermida and Bart Jacobs, *Fibrations with Indeterminates: Contextual and Functional Completeness for Polymorphic Lambda Calculi*, in *Mathematical Structures in Computer Science*, vol. 5, pp. 501–531, 1995.

[8] Alexandre Buisse and Peter Dybjer, *The Interpretation of Intuitionistic Type Theory in Locally Cartesian Closed Categories – an Intuitionistic Perspective*,in *Electronic Notes in Theoretical Computer Science*, pp. 21–32, 2008. `doi:10.1016/j.entcs.2008.10.003`

[9] Errett Bishop, *Foundations of Constructive Analysis*, 1967.

[10] Bengt Nordström, Kent Petersson, and Jan M. Smith, *Programming in Martin-Löf's Type Theory*, Oxford University Press, 1990.

[11] Matthieu Sozeau and Nicolas Tabareau, *Internalizing Intensional Type Theory*, unpublished.

[12] Martin Hofmann and Thomas Streicher, *The Groupoid Model Refutes Uniqueness of Identity Proofs*, in *Logic in Computer Science (LICS'94)*, IEEE, pp. 208–212, 1994.

[13] Bart Jacobs, *Categorical Logic and Type Theory*, vol. 141, 1999.

[14] Anders Mörtberg et al., *Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom*, arXiv:1611.02108, 2017.

[15] Simon Huber, *Cubical Interpretations of Type Theory*, Ph.D. thesis, Dept. of Computer Science and Engineering, University of Gothenburg, 2016.

[16] Maksym Sokhatskyi and Pavlo Maslianko, *The Systems Engineering of Consistent Pure Language with Effect Type System for Certified Applications and Higher Languages*, in *Proc. 4th Int. Conf. Mathematical Models and Computational Techniques in Science and Engineering*, 2018. `doi:10.1063/1.5045439`

Issue II: Inductive Types

Maksym Sokhatskyi [1]

[1] National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnical Institute
June 1, 2025

Abstract

Impredicative Encoding of Inductive Types in HoTT.

Keywords: Formal Methods, Type Theory, Programming Languages, Theoretical Computer Science, Applied Mathematics, Cubical Type Theory, Martin-Löf Type Theory

3 Inductive Encodings

3.1 Church Encoding

You know Church encoding which also has its dependent alanolgue in CoC, however in Coq it is imposible to detive Inductive Principle as type system lacks fixpoint and functional extensionality. The example of working compiler of PTS languages are Om and Morte. Assume we have Church encoded NAT:

```
nat = (X:U) -> (X -> X) -> X -> X
```

where first parameter $(X- > X)$ is a *succ*, the second parameter X is *zero*, and the result of encoding is landed in X. Even if we encode the parameter

```
list (A: U) = (X:U) -> X -> (A -> X) -> X
```

and paremeter A let's say live in 42 universe and X live in 2 universe, then by the signature of encoding the term will be landed in X, thus 2 universe. In other words such dependency is called impredicative displaying that landed term is not a predicate over parameters. This means that Church encoding is incompatible with predicative type checkers with predicative of predicative-cumulative hierarchies.

3.2 Scott Encoding

3.3 Parigot Encoding

3.4 CPS Encoding

3.5 Interaction Networks Encoding

3.6 Impredicative Encoding

In HoTT n-types is encoded as n-groupoids, thus we need to add a predicate in which n-type we would like to land the encoding:

```
NAT (A: U) = (X:U) -> isSet X -> X -> (A -> X) -> X
```

Here we added isSet predicate. With this motto we can implement propositional truncation by landing term in isProp or even HIT by langing in isGroupoid:

```
TRUN (A:U) type = (X: U) -> isProp X -> (A -> X) -> X
S1 = (X:U) -> isGroupoid X -> ((x:X) -> Path X x x) -> X
MONOPLE (A:U) = (X:U) -> isSet X -> (A -> X) -> X
NAT = (X:U) -> isSet X -> X -> (A -> X) -> X
```

The main publication on this topic could be found at [11] and [10].

The Unit Example

Here we have the implementation of Unit impredicative encoding in HoTT.

```
upPath      (X Y:U)(f:X->Y)(a:X->X): X -> Y = o X X Y f a
downPath    (X Y:U)(f:X->Y)(b:Y->Y): X -> Y = o X Y Y b f
naturality  (X Y:U)(f:X->Y)(a:X->X)(b:Y->Y): U
  = Path (X->Y)(upPath X Y f a)(downPath X Y f b)

unitEnc': U = (X: U) -> isSet X -> X -> X
isUnitEnc (one: unitEnc'): U
  = (X Y:U)(x:isSet X)(y:isSet Y)(f:X->Y) ->
    naturality X Y f (one X x)(one Y y)

unitEnc: U = (x: unitEnc') * isUnitEnc x
unitEncStar: unitEnc = (\(X:U)(_:isSet X) ->
  idfun X,\(X Y: U)(_:isSet X)(_:isSet Y)->refl(X->Y))
unitEncRec  (C: U) (s: isSet C) (c: C): unitEnc -> C
  = \(z: unitEnc) -> z.1 C s c
unitEncBeta (C: U) (s: isSet C) (c: C)
  : Path C (unitEncRec C s c unitEncStar) c = refl C c
unitEncEta (z: unitEnc): Path unitEnc unitEncStar z = undefined
unitEncInd (P: unitEnc -> U) (a: unitEnc): P unitEncStar -> P a
  = subst unitEnc P unitEncStar a (unitEncEta a)
unitEncCondition (n: unitEnc'): isProp (isUnitEnc n)
  =  \ (f g: isUnitEnc n) ->
     <h> \ (x y: U) -> \ (X: isSet x) -> \ (Y: isSet y)
  -> \ (F: x -> y) -> <i> \ (R: x) -> Y (F (n x X R)) (n y Y (F R))
       (<j> f x y X Y F @ j R) (<j> g x y X Y F @ j R) @ h @ i
```

3.7 Lambek Encoding: Homotopy Initial Algebras

4 Inductive Types

4.1 W

Well-founded trees without mutual recursion represented as W-types.

Definition 33. (W-Formation). For $A : \mathcal{U}$ and $B : A \to \mathcal{U}$, type W is defined as $\mathrm{W}(A, B) : \mathcal{U}$ or

$$\mathrm{W}_{(x:A)} B(x) : \mathcal{U}.$$

```
def W' (A : U) (B : A →U) : U := W (x : A), B x
```

Definition 34. (W-Introduction). Elements of $\mathrm{W}_{(x:A)} B(x)$ are called well-founded trees and created with single sup constructor:

$$\sup : \mathrm{W}_{(x:A)} B(x).$$

```
def sup$'$ (A: U) (B: A → U) (x: A) (f: B x → W' A B)
  : W' A B
 := sup A B x f
```

Theorem 20. (Induction Principle $\mathrm{ind}_{\mathrm{W}}$). The induction principle states that for any types $A : \mathcal{U}$ and $B : A \to \mathcal{U}$ and type family C over $\mathrm{W}(A, B)$ and the function $g : G$, where

$$G = \prod_{x:A} \prod_{f:B(x) \text{ß} \mathrm{W}(A,B)} \prod_{b:B(x)} C(f(b)) \text{ß} C(\sup(x, f))$$

there is a dependent function:

$$\mathrm{ind}_{\mathrm{W}} : \prod_{C:\mathrm{W}(A,B) \text{ß} \mathcal{U}} \prod_{g:G} \prod_{a:A} \prod_{f:B(a) \text{ß} \mathrm{W}(A,B)} \prod_{b:B(a)} C(f(b)).$$

```
def W-ind (A : U) (B : A → U)
    (C : (W (x : A), B x) → U)
    (g : Π (x : A) (f : B x → (W (x : A), B x)),
           (Π (b : B x), C (f b)) → C (sup A B x f))
    (a : A) (f : B a → (W (x : A), B x)) (b : B a)
  : C (f b) := ind^W A B C g (f b)
```

Theorem 21. ($\mathrm{ind}_{\mathrm{W}}$ Computes). The induction principle $\mathrm{ind}^{\mathrm{W}}$ satisfies the equation:

$$\mathrm{ind}_{\mathrm{W}}\text{-}\beta : g(a, f, \lambda b.\mathrm{ind}^{\mathrm{W}}(g, f(b)))$$
$$=_{def} \mathrm{ind}_{\mathrm{W}}(g, \sup(a, f)).$$

```
def ind^W-β (A : U) (B : A → U)
    (C : (W (x : A), B x) → U) (g : Π (x : A)
    (f : B x → (W (x : A), B x)), (Π (b : B x), C (f b)) → C (sup A B x f))
    (a : A) (f : B a → (W (x : A), B x))
   : PathP (<_> C (sup A B a f))
     (ind^W A B C g (sup A B a f))
     (g a f (λ (b : B a), ind^W A B C g (f b)))
  := <_> g a f (λ (b : B a), ind^W A B C g (f b))
```

4.2 M

4.3 Empty

The Empty type represents False-type logical $\mathbf{0}$, type without inhabitants, void or $\perp$ (Bottom). As it has not inhabitants it lacks both constructors and eliminators, however, it has induction.

Definition 35. (Formation). Empty-type is defined as built-in $\mathbf{0}$-type:

$$\mathbf{0} : \mathcal{U}.$$

Theorem 22. (Induction Principle ind_0). $\mathbf{0}$-type is satisfying the induction principle:

$$\text{ind}_0 : \prod_{C\,:\,\mathbf{0} \to \mathcal{U}} \prod_{z\,:\,\mathbf{0}} C(z).$$

```
def Empty-ind (C: 0 → U) (z: 0) : C z := ind₀ (C z) z
```

Definition 36. (Negation or isEmpty). For any type A negation of A is defined as arrow from A to $\mathbf{0}$:

$$\neg A := A \to \mathbf{0}.$$

```
def isEmpty (A: U): U := A → 0
```

The witness of $\neg A$ is obtained by assuming A and deriving a contradiction. This techniques is called proof of negation and is applicable to any types in constrast to proof by contradiction which implies $\neg\neg A \to A$ (double negation elimination) and is applicable only to decidable types with $\neg A + A$ property.

4.4 Unit

Unit type is the simplest type equipped with full set of MLTT inference rules. It contains single inhabitant $\star$ (star).

4.5 Bool

4.6 Maybe

4.7 Either

4.8 Nat

4.9 List

4.10 Vector

4.11 Stream

4.12 Interpreter

References

[1] Frank Pfenning and Christine Paulin-Mohring, *Inductively Defined Types in the Calculus of Constructions*, in *Proc. 5th Int. Conf. Mathematical Foundations of Programming Semantics*, 1989, pp. 209–228. *doi:10.1007/BFb0040259*

[2] Christine Paulin-Mohring, *Inductive Definitions in the System Coq: Rules and Properties*, in *Typed Lambda Calculi and Applications (TLCA)*, 1993, pp. 328–345. *doi:10.1007/BFb0037116*

[3] Christine Paulin-Mohring, *Defining Inductive Sets in Type Theory*, in: G. Huet and G. Plotkin (eds), *Logical Environments*, Cambridge University Press, 1994, pp. 249–272.

[4] Peter Dybjer, *Inductive Sets and Families in Martin-Löf's Type Theory and Their Set-Theoretic Semantics*, *Lecture Notes in Computer Science*, 530, 1991, pp. 280–306. *doi:10.1007/BFb0014059*

[5] Peter Dybjer, *Inductive Families*, *Formal Aspects of Computing*, 6(4), 1994, pp. 440–465. *doi:10.1007/BF01211308*

[6] Peter Dybjer, *Representing inductively defined sets by wellorderings in Martin-Löf's type theory*, *Theoretical Computer Science*, 176(1–2), 1997, pp. 329–335. *doi:10.1016/S0304-3975(96)00145-4*

[7] Martin Hofmann, *Extensional Constructs in Intensional Type Theory*, PhD thesis, University of Edinburgh, 1995. `https://www2.informatik.uni-freiburg.de/~mhofmann/phdthesis.pdf`

[8] Martin Hofmann, *Syntax and Semantics of Dependent Types*, in: *Semantics and Logics of Computation*, 1995, pp. 79–130.

[9] Newstead, C. (2018). *Algebraic Models of Dependent Type Theory*. PhD thesis, Carnegie Mellon University. Available at `https://arxiv.org/abs/2103.06155`.

[10] Sam Speight, *Impredicative Encoding of Inductive Types in HoTT*, 2017. `https://github.com/sspeight93/Papers/`

[11] Steve Awodey, *Impredicative Encodings in HoTT*, 2017. `https://www.newton.ac.uk/files/seminar/20170711090010001-1009680.pdf`

[12] Steve Awodey. *Type theory and homotopy*, 2010. `https://arxiv.org/abs/1010.1810`

[13] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. *Indexed Containers. Logical Methods in Computer Science*, 18(2), 2022, pp. 15:1–15:37. `https://lmcs.episciences.org/`

[14] Marcelo P. Fiore, Andrew M. Pitts, and S. C. Steenkamp. *Quotients, Inductive Types, & Quotient Inductive Types.* University of Cambridge, 2022. `https://arxiv.org/pdf/1705.07088`

[15] Thorsten Altenkirch, Neil Ghani, and Peter Morris. *Containers—Constructively*, 2012. `https://arxiv.org/pdf/1201.3898`

[16] Thorsten Altenkirch, Conor McBride, and James Chapman. *Towards Observational Type Theory*, 2013. `https://arxiv.org/pdf/1307.2765`

[17] Peter Dybjer, *Representing inductively defined sets by wellorderings in Martin-Löf's type theory*, *Theoretical Computer Science*, 176(1–2), 1997, pp. 329–335. *doi:10.1016/S0304-3975(96)00145-4*

[18] Ieke Moerdijk and Erik Palmgren, *Wellfounded trees in categories*, *Annals of Pure and Applied Logic*, 104(1–3), 2000, pp. 189–218. *doi:10.1016/S0168-0072(00)00012-9*

[19] Michael Abbott, Thorsten Altenkirch, and Neil Ghani, *Containers: Constructing strictly positive types*, *Theoretical Computer Science*, 342(1), 2005, pp. 3–27. *doi:10.1016/j.tcs.2005.06.002*

[20] Benno van den Berg and Ieke Moerdijk, *W-types in sheaves*, 2008. `https://arxiv.org/abs/0810.2398`

[21] Nicola Gambino and Martin Hyland, *Wellfounded Trees and Dependent Polynomial Functors*, in *TYPES 2003*, LNCS 3085, Springer, 2004, pp. 210–225. *doi:10.1007/978-3-540-24849-1_14*

[22] Michael Abbott, Thorsten Altenkirch, and Neil Ghani, *Representing Nested Inductive Types using W-types*, in *ICALP 2004*, LNCS 3142, Springer, 2004, pp. 124–135. *doi:10.1007/978-3-540-27836-8_8*

[23] Steve Awodey, Nicola Gambino, and Kristina Sojakova, *Inductive types in homotopy type theory*, *LICS 2012*, pp. 95–104. *doi:10.1109/LICS.2012.21*, `https://arxiv.org/abs/1201.3898`

[24] Benno van den Berg and Ieke Moerdijk, *W-types in Homotopy Type Theory*, *Mathematical Structures in Computer Science*, 25(5), 2015, pp. 1100–1115. *doi:10.1017/S0960129514000516*, `https://arxiv.org/abs/1307.2765`

[25] Kristina Sojakova, *Higher Inductive Types as Homotopy-Initial Algebras*, *ACM SIGPLAN Notices*, 50(1), 2015, pp. 31–42. *doi:10.1145/2775051.2676983*, `https://arxiv.org/abs/1402.0761`

[26] Steve Awodey, Nicola Gambino, and Kristina Sojakova, *Homotopy-initial algebras in type theory*, *Journal of the ACM*, 63(6), 2017, Article 45. *doi:10.1145/3006383*, `https://arxiv.org/abs/1504.05531`

[27] Christian Sattler, *On relating indexed W-types with ordinary ones*, in *TYPES 2015*, pp. 71–72. `https://types2015.inria.fr/slides/sattler.pdf`

[28] Per Martin-Löf, *Constructive Mathematics and Computer Programming*, in: Proc. 6th Int. Congress of Logic, Methodology and Philosophy of Science, 1979. *Studies in Logic and the Foundations of Mathematics* 104 (1982), pp. 153–175. *doi:10.1016/S0049-237X(09)70189-2*

[29] Per Martin-Löf (notes by Giovanni Sambin), *Intuitionistic type theory*, Lecture notes Padua 1984, Bibliopolis, Napoli (1984).

[30] Jasper Hugunin, *Why Not W?*, *LIPIcs*, 188 (TYPES 2020), 2021. *doi:10.4230/LIPIcs.TYPES.2020.8*

[31] Nils Anders Danielsson, *Positive h-levels are closed under W*, 2012. `https://www.cse.chalmers.se/~nad/listings/w-level/WLevel.html`

[32] Jasper Hugunin, *IWTypes Repository*. `https://github.com/jashug/IWTypes`

Issue III: Homotopy Type Theory

Maksym Sokhatskyi [1]

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

5 Groupoid Interpretation

5.1 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):

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

5.2 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,
      ⋆
    )
```

5.3 Metatheory: Adjunction Triples

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

5.3.1 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.

5.3.2 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.

5.3.3 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 $\mathbb{R}$.

6 Homotopy Type Theory

6.1 Identity Systems

Definition 37. (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 23. (Fundamental Theorem of Identity System).

Definition 38. (Strict Identity System). An identity system over type A and universe of pretypes V_i is called strict identity system ($=$), which respects UIP.

Definition 39. (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.

6.2 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 40. (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 41. (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 42. (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 43. (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 44. (Path Inversion).

Theorem 24. (Congruence).

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

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

```
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 45. (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 46. (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 47. (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 48. (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 25. (Heterogeneous Composition Kan Operation).

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

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

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

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

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

```
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 26. (Homogeneous Composition Kan Operation).

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

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

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

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

Theorem 27. (Substitution).

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

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

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

```
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 28. (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 29. (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 30. (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 31. (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 32. (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 33. (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.

6.3 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 49. (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 50. (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 51. (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 34. (Glue Computation).

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

Theorem 35. (Glue Uniqueness).

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

6.4 Fibration

Definition 52 (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 53 (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 54 (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 36 (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 55. (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 56. (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 57. (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 58. (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)
```

6.5 Equivalence

Definition 59. (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 60. (Fiberwise Reflection). There is a fiberwise instance $\mathrm{id}_\simeq$ of $A \simeq A$ that is derived as $(\mathrm{id}(A), \mathrm{isContrSingl}(A))$:

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

```
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 37. (Fiberwise Induction Principle). For any $P : A \to B \to A \simeq B \to U$ and it's evidence d at $(B, B, \mathrm{id}_\simeq(B))$ 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 38. (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 61. (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 62. (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 63. (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 64. (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
```

6.6 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 $\mathrm{I} = [0, 1]$ is the perfect foundation for definition of homotopy.

Definition 65. (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 66. (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 67. (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 68. (funExt-Formation)

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

Definition 69. (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 70. (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 71. (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 72. (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
```

6.7 Isomorphism

Definition 73. (iso-Formation)

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

Definition 74. (iso-Introduction)

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

Definition 75. (iso-Elimination)

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

Definition 76. (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 77. (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
```

6.8 Univalence

Definition 78. (uni-Formation)

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

Definition 79. (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 80. (uni-Elimination)

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

Definition 81. (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 82. (uni-Uniqueness)

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

6.9 Loop

Definition 83. (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 84. (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 85. (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)
```

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

6.10 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].

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
```

6.11 Homotopy Groups

Definition 86. (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)))
```

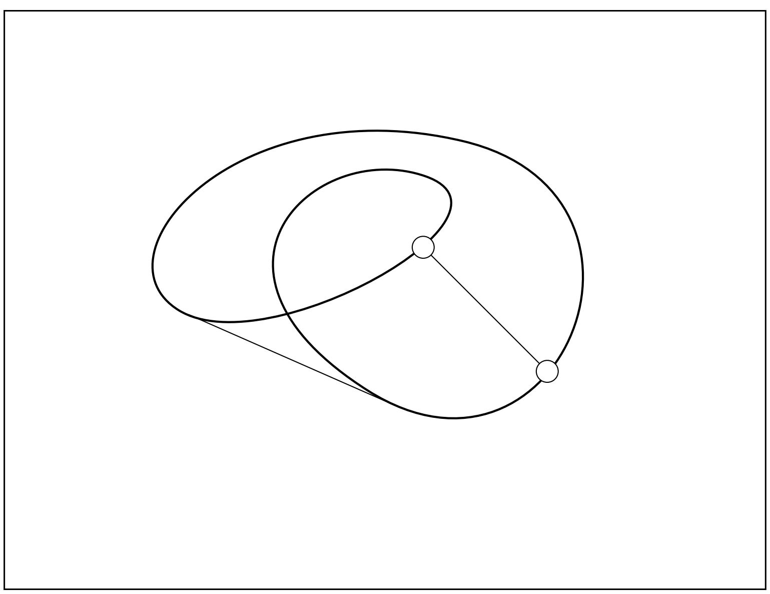
Theorem 39. $(\Omega(S^1) = \mathbb{Z})$.

```
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
```

6.12 Hopf Fibrations

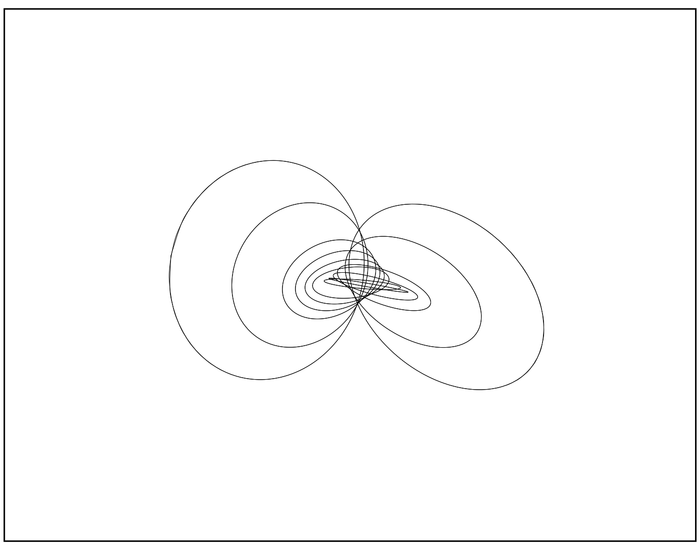
Example 2. (S^1 $\mathbb{R}$ Hopf Fiber).

```
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 87. (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 40. (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 41. (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 42. (Adams, Atiyah). Hopf Fibrations are only maps that have Hopf invariant 1.

Issue IV: Higher Inductive Types

Maksym Sokhatskyi [1]

[1] National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnic Institute
May 4, 2019

Abstract

CW-complexes are central to both homotopy theory and homotopy type theory (HoTT) and are encoded in cubical theorem-proving systems as higher inductive types (HIT), similar to recursive trees for (co)inductive types. We explore the basic primitives of homotopy theory, which are considered as a foundational basis in theorem-proving systems.
Keywords: Homotopy Theory, Type Theory

7 CW-Complexes

CW-complexes are spaces constructed by attaching cells of various dimensions. In HoTT, they are encoded as higher inductive types (HIT), where cells are constructors for points and paths.

Definition 88. (Cell Attachment). The attachment of an n-cell to a space X along $f : S^{n-1} \to X$ is a pushout:

$$
\begin{tikzcd}
S^{n-1} \arrow[r, "f"] \arrow[d, "\iota"] & X \arrow[d, "j"] \\
D^n \arrow[r, "g"] & X \cup_f D^n
\end{tikzcd}
$$

Here, $\iota : S^{n-1} \hookrightarrow D^n$ is the boundary inclusion, and $X \cup_f D^n$ is the pushout that attaches an n-cell to X via f. The result depends on the homotopy class of f.

Definition 89. (CW-Complex). A CW-complex is a space X, constructed inductively by attaching cells, with a skeletal filtration:

- (-1)-skeleton: $X_{-1} = \varnothing$.

- For $n \geq 0$, the n-skeleton X_n is obtained by attaching n-cells to X_{n-1}. For indices J_n and maps $\{f_j : S^{n-1} \to X_{n-1}\}_{j \in J_n}$, X_n is the pushout:

$$\begin{tikzcd} \coprod_{j \in J_n} S^{n-1} \arrow[r, "\coprod f_j"] \arrow[d, "\coprod \iota_j"] & X_{n-1} \arrow[d, "i_n"] \\ \coprod_{j \in J_n} D^n \arrow[r, "\coprod g_j"] & X_n \end{tikzcd}$$

  where $\coprod_{j \in J_n} S^{n-1}$, $\coprod_{j \in J_n} D^n$ are disjoint unions, and $i_n : X_{n-1} \hookrightarrow X_n$ is the inclusion.

- X is the colimit:

$$\varnothing = X_{-1} \hookrightarrow X_0 \hookrightarrow X_1 \hookrightarrow \ldots \hookrightarrow X,$$

  where X_n is the n-skeleton, and $X = \mathrm{colim}_{n \to \infty} X_n$. The sequence is the skeletal filtration.

In HoTT, CW-complexes are higher inductive types (HIT) with constructors for cells and paths for attachment.

7.1 Introduction: Countable Constructors

Some HITs require an infinite number of constructors for spaces, such as Eilenberg-MacLane spaces or the infinite sphere S^∞.

```
def S∞ : U
 := inductive { base
              | loop (n: ℕ) : base ≡ base
              }
```

Challenges include type checking, computation, and expressiveness.

Agda Cubical uses cubical primitives to handle HITs, supporting infinite constructors via HITs indexed by natural numbers, as colimits.

7.2 Motivation: Higher Inductive Types

HITs in HoTT enable direct encoding of topological spaces, such as CW-complexes. In homotopy theory, spaces are constructed by attaching cells via attaching maps. HoTT views types as spaces, elements as points, and equalities as paths, making HITs a natural choice. Standard inductive types cannot capture higher homotopies, but HITs allow constructors for points and paths. For example, the circle S^1 (Definition 2) has a base point and a loop, encoding its fundamental group $\mathbb{Z}$. HITs avoid the use of multiple quotient spaces, preserving the synthetic nature of HoTT. In cubical type theory, paths are intervals (e.g., $< i >$) with computational content, unlike propositional equalities, enabling efficient type checking in tools such as Agda Cubical.

7.3 Metatheory: Cohesive Topoi

7.3.1 Geometric Proofs

$$\Re \dashv \Im \dashv \&$$

For differential geometry, type theory incorporates primitive axioms of categorical meta-theoretical models of three Schreiber-Shulman functors: infinitesimal neighborhood ($\Im$), reduced modality ($\Re$), and infinitesimal discrete neighborhood ($\&$).

7.3.2 Flat Proofs

7.3.3 Sharp Proofs

7.3.4 Bose Proofs

7.3.5 Fermi Proofs

7.3.6 Linear Proofs

$$\otimes \dashv x \dashv \multimap$$

For engineering applications (e.g., Milner's π-calculus, quantum computing) and linear type theory, type theory embeds linear proofs based on the adjunction of the tensor and linear function spaces: $(A \otimes B) \multimap A \simeq A \multimap (B \multimap C)$, represented in a symmetric monoidal category $\mathbf{D}$ for a functor $[A, B]$ as: $\mathbf{D}(A \otimes B, C) \simeq \mathbf{D}(A, [B, C])$.

8 Higher Inductive Types

CW-complexes are central to HoTT and appear in cubical type checkers as HITs. Unlike inductive types (recursive trees), HITs encode CW-complexes, capturing points (0-cells) and higher paths (n-cells). The definition of an HIT specifies a CW-complex through cubical composition, an initial algebra in the cubical model.

8.1 Suspension

The suspension ΣA of a type A is a higher inductive type that constructs a new type by adding two points, called poles, and paths connecting each point of A to these poles. It is a fundamental construction in homotopy theory, often used to shift homotopy groups, e.g., obtaining S^{n+1} from S^n.

Definition 90. (Formation). For any type $A : \mathcal{U}$, there exists a suspension type $\Sigma A : \mathcal{U}$.

Definition 91. (Constructors). For a type $A : \mathcal{U}$, the suspension $\Sigma A : \mathcal{U}$ is generated by the following higher inductive compositional structure:

$$\Sigma := \begin{cases} \text{north} \\ \text{south} \\ \text{merid} : (a : A) \to \text{north} \equiv \text{south} \end{cases}$$

```
def Σ (A: U) : U
 := inductive { north
              | south
              | merid (a: A) : north ≡ south
              }
```

Theorem 43. (Elimination). For a family of types $B : \Sigma A \to \mathcal{U}$, points $n : B(\text{north})$, $s : B(\text{south})$, and a family of dependent paths

$$m : \Pi(a : A), \text{PathOver}(B, \text{merid}(a), n, s),$$

there exists a dependent map $\text{Ind}_{\Sigma A} : (x : \Sigma A) \to B(x)$, such that:

$$\begin{cases} \text{Ind}_{\Sigma A}(\text{north}) = n \\ \text{Ind}_{\Sigma A}(\text{south}) = s \\ \text{Ind}_{\Sigma A}(\text{merid}(a, i)) = m(a, i) \end{cases}$$

```
def PathOver (B: Σ A -> U) (a: A) (n: B north) (s: B south) : U
  := PathP (λ i , B (merid a @ i)) n s

def Ind_ΣA (A: U) (B: Σ A -> U) (n: B north) (s: B south)
    (m: (a: A) -> PathOver B (merid a) n s) : (x: Σ A) -> B x
 := split { north -> n | south -> s | merid a @ i -> m a @ i }
```

Theorem 44. (Computation).

$$\text{Ind}_{\Sigma} A(\text{north}) = n \text{Ind}_{\Sigma} A(\text{south}) = s \text{Ind}_{\Sigma} A(\text{merid}(a, i)) = m(a, i)$$

```
def Σ-β (A: U) (B: ΣA -> U) (n: B north) (s: B south)
    (m: (a: A) -> PathOver B (merid a) n s) (x: Σ A)
  : Path (B x) (Σ-I A B n s m x)
    split { north -> n | south -> s | merid a @ i -> m a @ i }
```

Theorem 45. (Uniqueness). Any two maps $h_1, h_2 : (x : \Sigma A) \to B(x)$ are homotopic if they agree on north, south, and merid, i.e., if $h_1(\text{north}) = h_2(\text{north})$, $h_1(\text{south}) = h_2(\text{south})$, and $h_1(\text{merid } a) = h_2(\text{merid } a)$ for all $a : A$.

8.2 Pushout

The pushout (amalgamation) is a higher inductive type that constructs a type by gluing two types A and B along a common type C via maps $f : C \to A$ and $g : C \to B$. It is a fundamental construction in homotopy theory, used to model cell attachment and cofibrant objects, generalizing the topological notion of a pushout.

Definition 92. (Formation). For types $A, B, C : \mathcal{U}$ and maps $f : C \to A$, $g : C \to B$, there exists a pushout $\sqcup(A, B, C, f, g) : \mathcal{U}$.

Definition 93. (Constructors). The pushout is generated by the following higher inductive compositional structure:

$$\sqcup := \begin{cases} \mathrm{po}_1 : A \to \sqcup(A, B, C, f, g) \\ \mathrm{po}_2 : B \to \sqcup(A, B, C, f, g) \\ \mathrm{po}_3 : (c : C) \to \mathrm{po}_1(f(c)) \equiv \mathrm{po}_2(g(c)) \end{cases}$$

```
def ⊔ (A B C: U) (f: C -> A) (g: C -> B) : U
 := inductive { po₁ (a: A)
              | po₂ (b: B)
              | po₃ (c: C) : po₁(f(c)) ≡ po₂(g(c))
              }
```

Theorem 46. (Elimination). For a type $D : \mathcal{U}$, maps $u : A \to D$, $v : B \to D$, and a family of paths $p : (c : C) \to u(f(c)) \equiv v(g(c))$, there exists a map $\mathrm{Ind}_\sqcup : \sqcup(A, B, C, f, g) \to D$, such that:

$$\begin{cases} \mathrm{Ind}_\sqcup(\mathrm{po}_1(a)) = u(a) \\ \mathrm{Ind}_\sqcup(\mathrm{po}_2(b)) = v(b) \\ \mathrm{Ind}_\sqcup(\mathrm{po}_3(c, i)) = p(c, i) \end{cases}$$

```
def PathOver (A B C : U) (f : C → A) (g : C → B)
     (D : ⊔ A B C f g → U)
     (c : C) (u : D (po₁ (f c))) (v : D (po₂ (g c))) : U
 := PathP (λ i, D (po₃ c i)) u v

def Ind⊔ : (A B C : U) (f : C → A) (g : C → B)
     (D : ⊔ A B C f g → U)
     (u : (a : A) → D (po₁ a))
     (v : (b : B) → D (po₂ b))
     (p : (c : C) → PathOver D c (u (f c)) (v (g c)))
   : (x : ⊔ A B C f g) → D x
 := split { po₁ a → u a | po₂ b → v b | po₃ c @ i → p c @ i }
```

Theorem 47. (Computation). For $x : \sqcup(A, B, C, f, g)$,

$$\begin{cases} \mathrm{Ind}_\sqcup(\mathrm{po}_1(a)) \equiv u(a) \\ \mathrm{Ind}_\sqcup(\mathrm{po}_2(b)) \equiv v(b) \\ \mathrm{Ind}_\sqcup(\mathrm{po}_3(c, i)) \equiv p(c, i) \end{cases}$$

Theorem 48. (Uniqueness). Any two maps $u, v : \sqcup(A, B, C, f, g) \to D$ are homotopic if they agree on po_1, po_2, and po_3, i.e., if $u(\mathrm{po}_1(a)) = v(\mathrm{po}_1(a))$ for all $a : A$, $u(\mathrm{po}_2(b)) = v(\mathrm{po}_2(b))$ for all $b : B$, and $u(\mathrm{po}_3(c)) = v(\mathrm{po}_3(c))$ for all $c : C$.

Example 4. (Cell Attachment) The pushout models the attachment of an n-cell to a space X. Given $f : S^{n-1} \to X$ and inclusion $g : S^{n-1} \to D^n$, the pushout $\sqcup(X, D^n, S^{n-1}, f, g)$ is the space $X \cup_f D^n$, attaching an n-disk to X along f.

$$\begin{tikzcd} S^{n-1} \arrow[r, "f"] \arrow[d, "g"] & X \arrow[d] \\ D^n \arrow[r] & X \cup_f D^n \end{tikzcd}$$

8.3 Spheres

Spheres are higher inductive types with higher-dimensional paths, representing fundamental topological spaces.

Definition 94. (Pointed n-Spheres) The n-sphere S^n is defined recursively as a type in the universe $\mathcal{U}$ using general recursion over dimensions:

$$\mathbb{S}^n := \begin{cases} \text{point} : \mathbb{S}^n, \\ \text{surface} : < i_1, ... i_n > [\ (i_1 = 0) \to \text{point}, (i_1 = 1) \to \text{point},\ ... \\ \qquad\qquad\qquad\qquad (i_n = 0) \to \text{point}, (i_n = 1) \to \text{point}\ ] \end{cases}$$

Definition 95. (n-Spheres via Suspension) The n-sphere S^n is defined recursively as a type in the universe $\mathcal{U}$ using general recursion over natural numbers $\mathbb{N}$. For each $n \in \mathbb{N}$, the type $S^n : \mathcal{U}$ is defined as:

$$\mathbb{S}^n := \begin{cases} S^0 = \mathbf{2}, \\ S^{n+1} = \Sigma(S^n). \end{cases}$$

```
def sphere : ℕ → U := ℕ-iter U 2 Σ
```

This iterative definition applies the suspension functor Σ to the base type $\mathbf{2}$ (0-sphere) n times to obtain S^n.

Example 5. (Sphere as CW-Complex) The n-sphere S^n can be constructed as a CW-complex with one 0-cell and one n-cell:

$$\begin{cases} X_0 = \{\text{base}\},\ \text{one point} \\ X_k = X_0 \text{ for } 0 < k < n,\ \text{no additional cells} \\ X_n : \text{Attachment of an } n\text{-cell to } X_{n-1} = \{\text{base}\} \text{ along } f : S^{n-1} \to \{\text{base}\} \end{cases}$$

The constructor cell attaches the boundary of the n-cell to the base point, yielding the type S^n.

8.4 Hub and Spokes

The hub and spokes construction $\odot$ defines an n-truncation, ensuring that the type has no non-trivial homotopy groups above dimension n. It models the type as a CW-complex with a hub (central point) and spokes (paths to points).

Definition 96. (Formation). For types $S, A : \mathcal{U}$, there exists a hub and spokes type $\odot\ (S, A) : \mathcal{U}$.

Definition 97. (Constructors). The hub and spokes type is freely generated by the following higher inductive compositional structure:

$$\odot := \begin{cases} \text{base} : A \to \odot\ (S, A) \\ \text{hub} : (S \to \odot\ (S, A)) \to \odot\ (S, A) \\ \text{spoke} : (f : S \to \odot\ (S, A)) \to (s : S) \to \text{hub}(f) \equiv f(s) \end{cases}$$

```
def ⊙ (S A: U) : U
 := inductive { base (x: A)
              | hub (f: S -> ⊙ S A)
              | spoke (f: S -> ⊙ S A) (s:S) : hub f ≡ f s
              }
```

Theorem 49. (Elimination). For a family of types $P : \text{HubSpokes}\, S\, A \to \mathcal{U}$, maps pbase : $(x : A) \to P(\text{base}\, x)$, phub : $(f : S \to \text{HubSpokes}\, S\, A) \to P(\text{hub}\, f)$, and a family of paths pspoke : $(f : S \to \text{HubSpokes}\, S\, A) \to (s : S) \to \text{PathP}\, (< i > P(\text{spoke}\, f\, s\, @\, i))\, (\text{phub}\, f)\, (P(f\, s))$, there exists a map hubSpokesInd : $(z : \text{HubSpokes}\, S\, A) \to P(z)$, such that:

$$\begin{cases} \text{Ind}_{\odot}\, (\text{base}\, x) = \text{pbase}\, x \\ \text{Ind}_{\odot}\, (\text{hub}\, f) = \text{phub}\, f \\ \text{Ind}_{\odot}\, (\text{spoke}\, f\, s\, @\, i) = \text{pspoke}\, f\, s\, @\, i \end{cases}$$

8.5 Truncation

Set Truncation

Definition 98. (Formation). Set truncation (0-truncation), denoted $\|A\|_0$, ensures that the type is a set, with homotopy groups vanishing above dimension 0.

Definition 99. (Constructors). For $A : \mathcal{U}$, $\|A\|_0 : \mathcal{U}$ is defined by the following higher inductive compositional structure:

$$\|\_\|_0 := \begin{cases} \text{inc} : A \to \|A\|_0 \\ \text{squash} : (a, b : \|A\|_0) \to (p, q : a \equiv b) \to p \equiv q \end{cases}$$

```
def ||_||0 (A: U) : U
 := inductive { inc (a: A)
              | squash (a b: ||A||0) (p q: Path (||A||0) a b)
                <i j> [ (i = 0) -> p @ j, (i = 1) -> q @ j,
                        (j = 0) -> a,     (j = 1) -> b ]
              }
```

Theorem 50. (Elimination $\|A\|_0$) For a set $B : \mathcal{U}$ (i.e., isSet(B)), and a map $f : A \to B$, there exists setTruncRec : $\|A\|_0 \to B$, such that $\text{Ind}_{\|A\|_0}(\text{inc}(a)) = f(a)$.

Groupoid Truncation

Definition 100. (Formation). Groupoid truncation (1-truncation), denoted $\|A\|_1$, ensures that the type is a 1-groupoid, with homotopy groups vanishing above dimension 1.

Definition 101. (Constructors). For $A : \mathcal{U}$, $\|A\|_1 : \mathcal{U}$ is defined by the following higher inductive compositional structure:

$$\|\_\|_1 := \begin{cases} \text{inc} : A \to \|A\|_1 \\ \text{squash} : (a, b : \|A\|_1) \to (p, q : a \equiv b) \to (r, s : p \equiv q) \to r \equiv s \end{cases}$$

```
def ||_||1 (A: U) : U
 := inductive { inc (a: A)
              | squash (a b: ||A||1) (p q: Path (||A||1) a b)
                (r s: Path (Path (||A||1) a b) p q) <i j k>
                [ (i = 0) -> r @ j @ k, (i = 1) -> s @ j @ k,
                  (j = 0) -> p @ k,     (j = 1) -> q @ k,
                  (k = 0) -> a,         (k = 1) -> b ]
              }
```

Theorem 51. (Elimination $\|A\|_1$) For a 1-groupoid $B : \mathcal{U}$ (i.e., isGroupoid(B)), and a map $f : A \to B$, there exists $\text{Ind}_{\|A\|_1} : \|A\|_1 \to B$, such that $\text{Ind}_{\|A\|_1}(\text{inc}(a)) = f(a)$.

8.6 Quotients

Set Quotient Spaces

Quotient spaces are a powerful computational tool in type theory, embedded in the core of Lean.

Definition 102. (Formation). Set quotient spaces construct a type A, quotiented by a relation $R : A \to A \to \mathcal{U}$, ensuring that the result is a set.

Definition 103. (Constructors). For a type $A : \mathcal{U}$ and a relation $R : A \to A \to \mathcal{U}$, the set quotient space $A/R : \mathcal{U}$ is freely generated by the following higher inductive compositional structure:

$$A/R := \begin{cases} \text{quot} : A \to A/R \\ \text{ident} : (a, b : A) \to R(a, b) \to \text{quot}(a) \equiv \text{quot}(b) \\ \text{trunc} : (a, b : A/R) \to (p, q : a \equiv b) \to p \equiv q \end{cases}$$

```
def / (A: U) (R: A -> A -> U) : U
  := inductive { quot (a: A)
               | ident (a b: A) (r: R a b) : quot(a) ≡ quot(b)
               | trunc (a b : / A R) (p q : Path (/ A R) a b)
                 <i j> [ (i = 0) -> p @ j , (i = 1) -> q @ j ,
                         (j = 0) -> a ,      (j = 1) -> b ]
               }
```

Theorem 52. (Elimination). For a family of types $B : A/R \to \mathcal{U}$ with $\text{isSet}(Bx)$, and maps $f : (x : A) \to B(\text{quot}(x))$, $g : (a, b : A) \to (r : R(a, b)) \to \text{PathP}(< i > B(\text{ident}(a, b, r) @ i))(f(a))(f(b))$, there exists $\text{Ind}_{A/R} : \Pi(x : A/R), B(x)$, such that $\text{Ind}_{A/R}(\text{quot}(a)) = f(a)$.

Groupoid Quotient Spaces

Definition 104. (Formation). Groupoid quotient spaces extend set quotient spaces to produce a 1-groupoid, including constructors for higher paths. Groupoid quotient spaces construct a type A, quotiented by a relation $R : A \to A \to \mathcal{U}$, ensuring that the result is a groupoid.

Definition 105. (Constructors). For a type $A : \mathcal{U}$ and a relation $R : A \to A \to \mathcal{U}$, the groupoid quotient space $A//R : \mathcal{U}$ includes constructors for points, paths, and higher paths, ensuring a 1-groupoid structure.

8.7 Wedge

The wedge of two pointed types A and B, denoted $A \vee B$, is a higher inductive type representing the union of A and B with identified base points. Topologically, it corresponds to $A \times \{y_0\} \cup \{x_0\} \times B$, where x_0 and y_0 are the base points of A and B, respectively.

Definition 106. (Formation). For pointed types A, B : pointed, the wedge $A \vee B : \mathcal{U}$.

Definition 107. (Constructors). The wedge is generated by the following higher inductive compositional structure:

$$\vee := \begin{cases} \text{winl} : A.1 \to A \vee B \\ \text{winr} : B.1 \to A \vee B \\ \text{wglue} : \text{winl}(A.2) \equiv \text{winr}(B.2) \end{cases}$$

```
def ∨ (A : pointed) (B : pointed) : U
 := inductive { winl (a : A.1)
              | winr (b : B.1)
              | wglue : winl(A.2) ≡ winr(B.2)
              }
```

Theorem 53. (Elimination). For a type $P : A \vee B\mathcal{U}$, maps $f : A.1 \to C$, $g : B.1 \to C$, and a path $p : \text{PathOverlue}(P, f(A.2), g(B.2))$, there exists a map $\text{Ind}_\vee : A \vee B \to C$, such that:

$$\begin{cases} \text{Ind}(\text{winl}(a)) = f(a) \\ \text{Ind}(\text{winr}(b)) = g(b) \\ \text{Ind}(\text{wglue}(x)) = p(x) \end{cases}$$

```
def PathOverGlue : (P : A ∨ B → U)
     (p : P (inl (A.2))) (q : P (inr (B.2))) : U
 := PathP (λ i → P (wglue i)) p q

def Ind∨ (A B : pointed) (C : U) (f : A.1 -> C) (g : B.1 -> C)
     (p : Path C (f A.2) (g B.2))
  : A ∨ B -> C
 := split { winl a -> f a | winr b -> g b | wglue @ x -> p @ x }
```

Theorem 54. (Computation). For $z : \text{Wedge } AB$,

$$\begin{cases} \text{Ind}_\vee(\text{winl } a) \equiv f(a) \\ \text{Ind}_\vee(\text{winr } b) \equiv g(b) \\ \text{Ind}_\vee(\text{wglue } @\, x) \equiv p\,@\,x \end{cases}$$

Theorem 55. (Uniqueness). Any two maps $h_1, h_2 : \text{Wedge } AB \to C$ are homotopic if they agree on winl, winr, and wglue, i.e., if $h_1(\text{winl } a) = h_2(\text{winl } a)$ for all $a : A.1$, $h_1(\text{winr } b) = h_2(\text{winr } b)$ for all $b : B.1$, and $h_1(\text{wglue}) = h_2(\text{wglue})$.

8.8 Smash Product

The smash product of two pointed types A and B, denoted $A \wedge B$, is a higher inductive type that quotients the product $A \times B$ by the pushout $A \sqcup B$. It represents the space $A \times B/(A \times \{y_0\} \cup \{x_0\} \times B)$, collapsing the wedge to a single point.

Definition 108. (Formation). For pointed types A, B : pointed, the smash product $A \wedge B : \mathcal{U}$.

Definition 109. (Constructors). The smash product is generated by the following higher inductive compositional structure:

$$A \wedge B := \begin{cases} \text{basel} : A \wedge B \\ \text{baser} : A \wedge B \\ \text{proj}(x : A.1)(y : B.1) : A \wedge B \\ \text{gluel}(a : A.2) : \text{proj}(a, B.2) \equiv \text{basel} \\ \text{gluer}(b : B.2) : \text{proj}(A.2, b) \equiv \text{baser} \end{cases}$$

```
def ∧ (A : pointed) (B : pointed) : U
 := inductive { basel
              | baser
              | proj (a : A.1) (b : B.1)
              | gluel (a : A.2) : proj(a,B.2) ≡ basel
              | gluer (a : B.2) : proj(A.2,b) ≡ baser
              }
```

Theorem 56. (Elimination). For a family of types P : Smash $A\,B \to \mathcal{U}$, points pbasel : $P(\text{basel})$, pbaser : $P(\text{baser})$, maps pproj : $(x : A.1) \to (y : B.1) \to P(\text{proj}\,x\,y)$, and a family of paths pgluel : $(a : A.1) \to \text{pproj}(a, B.2) \equiv \text{pbasel}$, pgluer : $(b : B.1) \to \text{pproj}(A.2, b) \equiv \text{pbaser}$, there exists a map $\text{Ind}_\wedge : (z : A \wedge B) \to P(z)$, such that:

$$\begin{cases} \text{Ind}_\wedge\,(\text{basel}) = \text{pbasel} \\ \text{Ind}_\wedge\,(\text{baser}) = \text{pbaser} \\ \text{Ind}_\wedge\,(\text{proj}\,x\,y) = \text{pproj}\,x\,y \\ \text{Ind}_\wedge\,(\text{gluel}\,a\,@\,i) = \text{pgluel}\,a\,@\,i \\ \text{Ind}_\wedge\,(\text{gluer}\,b\,@\,i) = \text{pgluer}\,b\,@\,i \end{cases}$$

```
def Ind∧ (A B : pointed) (P : A ∧ B -> U)
    (pbasel : P basel) (pbaser : P baser)
    (pproj : (x : A.1) -> (y : B.1) -> P (proj x y))
    (pgluel : (a : A.1) -> PathP (<i> P (gluel a @ i)) (pproj a B.2) pbasel)
    (pgluer : (b : B.1) -> PathP (<i> P (gluer b @ i)) (pproj A.2 b) pbaser)
  : (z : A ∧ B) -> P z
 := split { basel -> pbasel | baser -> pbaser | proj x y -> pproj x y
          | gluel a @ i -> pgluel a @ i | gluer b @ i -> pgluer b @ i }
```

Theorem 57. (Computation). For a family of types $P : A \wedge B \to \mathcal{U}$, points $\mathrm{pbasel} : P(\mathrm{basel})$, $\mathrm{pbaser} : P(\mathrm{baser})$, map $\mathrm{pproj} : (x : A.1) \to (y : B.1) \to P(\mathrm{proj}\, x\, y)$, and families of paths $\mathrm{pgluel} : (a : A.1) \to \mathrm{PathP}\,(< i > P(\mathrm{gluel}\, a \,@\, i))\,(\mathrm{pproj}\, a\, B.2)\,\mathrm{pbasel}$, $\mathrm{pgluer} : (b : B.1) \to \mathrm{PathP}\,(< i > P(\mathrm{gluer}\, b \,@\, i))\,(\mathrm{pproj}\, A.2\, b)\,\mathrm{pbaser}$, the map $\mathrm{Ind}_{\wedge} : (z : A \wedge B) \to P(z)$ satisfies all equations for all variants of the predicate P:

$$\begin{cases} \mathrm{Ind}_{\wedge}\,(\mathrm{basel}) \equiv \mathrm{pbasel} \\ \mathrm{Ind}_{\wedge}\,(\mathrm{baser}) \equiv \mathrm{pbaser} \\ \mathrm{Ind}_{\wedge}\,(\mathrm{proj}\, x\, y) \equiv \mathrm{pproj}\, x\, y \\ \mathrm{Ind}_{\wedge}\,(\mathrm{gluel}\, a \,@\, i) \equiv \mathrm{pgluel}\, a \,@\, i \\ \mathrm{Ind}_{\wedge}\,(\mathrm{gluer}\, b \,@\, i) \equiv \mathrm{pgluer}\, b \,@\, i \end{cases}$$

Theorem 58. (Uniqueness). For a family of types $P : A \wedge B \to \mathcal{U}$, and maps $h_1, h_2 : (z : A \wedge B) \to P(z)$, if there exist paths $e_{\mathrm{basel}} : h_1(\mathrm{basel}) \equiv h_2(\mathrm{basel})$, $e_{\mathrm{baser}} : h_1(\mathrm{baser}) \equiv h_2(\mathrm{baser})$, $e_{\mathrm{proj}} : (x : A.1) \to (y : B.1) \to h_1(\mathrm{proj}\, x\, y) \equiv h_2(\mathrm{proj}\, x\, y)$, $e_{\mathrm{gluel}} : (a : A.1) \to \mathrm{PathP}\,(< i > h_1(\mathrm{gluel}\, a \,@\, i) \equiv h_2(\mathrm{gluel}\, a \,@\, i))\,(e_{\mathrm{proj}}\, a\, B.2)\, e_{\mathrm{basel}}$, $e_{\mathrm{gluer}} : (b : B.1) \to \mathrm{PathP}\,(< i > h_1(\mathrm{gluer}\, b \,@\, i) \equiv h_2(\mathrm{gluer}\, b \,@\, i))\,(e_{\mathrm{proj}}\, A.2\, b)\, e_{\mathrm{baser}}$, then $h_1 \equiv h_2$, i.e., there exists a path $(z : A \wedge B) \to h_1(z) \equiv h_2(z)$.

8.9 Join

The join of two types A and B, denoted $A \vee B$, is a higher inductive type that constructs a type by joining each point of A to each point of B via a path. Topologically, it corresponds to the join of spaces, forming a space that interpolates between A and B.

Definition 110. (Formation). For types $A, B : \mathcal{U}$, the join $A * B : \mathcal{U}$.

Definition 111. (Constructors). The join is generated by the following higher inductive compositional structure:

$$A \vee B := \begin{cases} \text{joinl} : A \to A \vee B \\ \text{joinr} : B \to A \vee B \\ \text{join}(a : A)(b : B) : \text{joinl}(a) \equiv \text{joinr}(b) \end{cases}$$

```
def ∨ (A : U) (B : U) : U
 := inductive { joinl (a: A)
              | joinr (b: B)
              | join (a: A) (b: B) : joinl(a) ≡ joinr(b)
              }
```

Theorem 59. (Elimination). For a type $C : \mathcal{U}$, maps $f : A \to C$, $g : B \to C$, and a family of paths $h : (a : A) \to (b : B) \to f(a) \equiv g(b)$, there exists a map $\text{Ind}_{\vee} : A \vee B \to C$, such that:

$$\begin{cases} \text{Ind}_{\vee}(\text{joinl}(a)) = f(a) \\ \text{Ind}_{\vee}(\text{joinr}(b)) = g(b) \\ \text{Ind}_{\vee}(\text{join}(a, b, i)) = h(a, b, i) \end{cases}$$

```
def Ind∨ (A B C : U) (f : A -> C) (g : B -> C)
    (h : (a : A) -> (b : B) -> Path C (f a) (g b))
  : A ∨ B -> C
 := split { joinl a -> f a
          | joinr b -> g b
          | join a b @ i -> h a b @ i
          }
```

Theorem 60. (Computation). For all $z : A \vee B$, and predicate P, the rules of $\text{Ind}_{\vee}$ hold for all parameters of the predicate P.

Theorem 61. (Uniqueness). Any two maps $h_1, h_2 : A \vee B \to C$ are homotopic if they agree on joinl, joinr, and join.

8.10 Colimit

Colimits construct the limit of a sequence of types, connected by maps, e.g., propositional truncations.

Definition 112. (Colimit) For a sequence of types $A : \text{nat} \to \mathcal{U}$ and maps $f : (n : \mathbb{N}) \to An \to A(\text{succ}(n))$, the colimit type $\text{colimit}(A, f) : \mathcal{U}$.

$$\text{colim} := \begin{cases} \text{ix} : (n : \text{nat}) \to An \to \text{colimit}(A, f) \\ \text{gx} : (n : \text{nat}) \to (a : A(n)) \to \text{ix}(\text{succ}(n), f(n, a)) \equiv \text{ix}(n, a) \end{cases}$$

```
def colimit (A : nat -> U) (f : (n : nat) -> A n -> A (succ n)) : U
 := inductive { ix (n : nat) (x: A n)
              | gx (n : nat) (a: A n)
                <i> [ (i=0) -> ix (succ n) (f n a),
                      (i=1) -> ix n a ]
              }
```

Theorem 62. (Elimination colimit) For a type P : colimit $Af \to \mathcal{U}$, with $p : (n : \text{nat}) \to (x : An) \to P(\text{ix}(n, x))$ and $q : (n : \text{nat}) \to (a : An) \to \text{PathP}(\langle i \rangle P(\text{gx}(n, a)@i))(p(\text{succ } n)(fna))(pna)$, there exists $i : \Pi_{x:\text{colimit } Af} P(x)$, such that $i(\text{ix}(n, x)) = pnx$.

8.11 Coequalizers

Coequalizer

The coequalizer of two maps $f, g : A \to B$ is a higher inductive type (HIT) that constructs a type consisting of elements in B, where f and g agree, along with paths ensuring this equality. It is a fundamental construction in homotopy theory, capturing the subspace of B where $f(a) = g(a)$ for $a : A$.

Definition 113. (Formation). For types $A, B : \mathcal{U}$ and maps $f, g : A \to B$, the coequalizer coeq $ABfg : \mathcal{U}$.

Definition 114. (Constructors). The coequalizer is generated by the following higher inductive compositional structure:

$$\text{Coeq} := \begin{cases} \text{inC} : B \to \text{Coeq}(A, B, f, g) \\ \text{glueC} : (a : A) \to \text{inC}(f(a)) \equiv \text{inC}(g(a)) \end{cases}$$

```
def Coeq (A B: U) (f g: A -> B) : U
 := inductive { inC (b: B)
              | glueC (a: A) : inC (f a) ≡ inC (g a)
              }
```

Theorem 63. (Elimination). For a type $C : \mathcal{U}$, map $h : B \to C$, and a family of paths $y : (x : A) \to \text{Path}_C(h(fx), h(gx))$, there exists a map coequRec : coeq $ABfg \to C$, such that:

$$\begin{cases} \text{coequRec}(\text{inC}(x)) = h(x) \\ \text{coequRec}(\text{glueC}(x, i)) = y(x, i) \end{cases}$$

```
def coequRec (A B C : U) (f g : A -> B) (h: B -> C)
     (y: (x : A) -> Path C (h (f x)) (h (g x)))
   : (z : coeq A B f g) -> C
  := split { inC x -> h x | glueC x @ i -> y x @ i }
```

Theorem 64. (Computation). For z : coeq $ABfg$,

$$\begin{cases} \text{coequRec}(\text{inC } x) \equiv h(x) \\ \text{coequRec}(\text{glueC } x \mathbin{@} i) \equiv y(x) \mathbin{@} i \end{cases}$$

Theorem 65. (Uniqueness). Any two maps h_1, h_2 : coeq $ABfg \to C$ are homotopic if they agree on inC and glueC, i.e., if $h_1(\text{inC } x) = h_2(\text{inC } x)$ for all $x : B$ and $h_1(\text{glueC } a) = h_2(\text{glueC } a)$ for all $a : A$.

Example 6. (Coequalizer as Subspace) The coequalizer coeq $ABfg$ represents the subspace of B, where $f(a) = g(a)$. For example, if $A = B = \mathbb{R}$ and $f(x) = x^2$, $g(x) = x$, the coequalizer captures the points where $x^2 = x$, i.e., $\{0, 1\}$.

Path Coequalizer

The path coequalizer is a higher inductive type that generalizes the coequalizer to handle pairs of paths in B. Given a map $p : A \to (b_1, b_2 : B) \times (\mathrm{Path}_B(b_1, b_2)) \times (\mathrm{Path}_B(b_1, b_2))$, it constructs a type where elements of A generate pairs of paths between points in B, with paths connecting the endpoints of these paths.

Definition 115. (Formation). For types $A, B : \mathcal{U}$ and a map $p : A \to (b_1, b_2 : B) \times (b_1 \equiv b_2) \times (b_1 \equiv b_2)$, there exists a path coequalizer $\mathrm{Coeq}_{\equiv}(A, B, p) : \mathcal{U}$.

Definition 116. (Constructors). The path coequalizer is generated by the following higher inductive compositional structure:

$$\mathrm{Coequ}_{\equiv} := \begin{cases} \mathrm{inP} : B \to \mathrm{Coeq}_{\equiv}(A, B, p) \\ \mathrm{glueP} : (a : A) \to \mathrm{inP}(p(a).2.2.1@0) \equiv \mathrm{inP}(p(a).2.2.2@1) \end{cases}$$

```
data Coeq-≡ (A B: U) (p : A -> Σ (b1 b2: B), b1 ≡ b2 × b1 ≡ b2)
   = inP (b: B)
   | glueP (a:A) <i> [(i=0) -> inP ((p a).2.2.1 @ 0),
                      (i=1) -> inP ((p a).2.2.2 @ 1) ]
```

Theorem 66. (Elimination). For a type $C : \mathcal{U}$, map $h : B \to C$, and a family of paths $y : (a : A) \to h(p(a).2.2.1@0) \equiv h(p(a).2.2.2@1)$, there exists a map $\text{Ind-Coequ}_{\equiv} : \mathrm{Coeq}_{\equiv}(A, B, p) \to C$, such that:

$$\begin{cases} \mathrm{coequPRec}(\mathrm{inP}(b)) = h(b) \\ \mathrm{coequPRec}(\mathrm{glueP}(a, i)) = y(a, i) \end{cases}$$

```
def Ind-Coequ≡ (A B C : U)
     (p : A -> Σ (b1 b2: B) (x: Path B b1 b2), Path B b1 b2)
     (h: B -> C) (y: (a : A) -> Path C (h (((p a).2.2.1) @ 0)) (h (((p a).2.2.2) @ 1)))
   : (z : coeqP A B p) -> C
  := split { inP b -> h b | glueP a @ i -> y a @ i }
```

Theorem 67. (Computation). For $z : \mathrm{coeqP}\ ABp$,

$$\begin{cases} \mathrm{coequPRec}(\mathrm{inP}\ b) \equiv h(b) \\ \mathrm{coequPRec}(\mathrm{glueP}\ a\,@\,i) \equiv y(a)\,@\,i \end{cases}$$

Theorem 68. (Uniqueness). Any two maps $h_1, h_2 : \mathrm{coeqP}\ ABp \to C$ are homotopic if they agree on inP and glueP, i.e., if $h_1(\mathrm{inP}\ b) = h_2(\mathrm{inP}\ b)$ for all $b : B$ and $h_1(\mathrm{glueP}\ a) = h_2(\mathrm{glueP}\ a)$ for all $a : A$.

8.12 K(G,n)

Eilenberg-MacLane spaces $K(G,n)$ have a single non-trivial homotopy group $\pi_n(K(G,n)) = G$. They are defined using truncations and suspensions.

Definition 117. (K(G,n)) For an abelian group G : abgroup, the type $KGn(G)$: nat $\to \mathcal{U}$.

$$K(G,n) := \begin{cases} n = 0 \rightsquigarrow \text{discreteTopology}(G) \\ n \geq 1 \rightsquigarrow \|\Sigma^{n-1}(K1'(G.1, G.2.1))\|_n \end{cases}$$

```
def KGn (G: abgroup) : N -> U
 := split { zero -> discreteTopology G
          | succ n -> nTrunc (Σ (K1' (G.1,G.2.1)) n) (succ n)
          }
```

Theorem 69. (Elimination KGn) For $n \geq 1$, a type $B : \mathcal{U}$ with isNGroupoid(B, succ n), and a map f : suspension($K1'G$) $\to B$, there exists $\text{rec}_{KGn} : KGnG(\text{succ } n) \to B$, defined via nTruncRec.

8.13 Localization

Localization constructs an F-local type from a type X, with respect to a family of maps $F_A : S(a) \to T(a)$.

Definition 118. (Localization Modality) For a family of maps $F_A : S(a) \to T(a)$, the F-localization $L_F^{AST}(X) : \mathcal{U}$.

$$L_F^A(X) := \begin{cases} \text{center} : X \to L_{F_A}(X) \\ \text{ext}(a : A) \to (S(a) \to L_{F_A}(X)) : T(a) \to L_{F_A}(X) \\ \text{isExt}(a : A)(f : S(a) \to L_{F_A}(X)) \to (s : S(a)) : \text{ext}(a, f, F(a,s)) \equiv f(s) \\ \text{extEq}(a : A)(g, h : T(a) \to L_{F_A}(X)) \\ \quad (p : (s : S(a)) \to g(F(a,s)) \equiv h(F(a,s))) \\ \quad (t : T(a)) : g(t) \equiv h(t) \\ \text{isExtEq} : (a : A)(g, h : T(a) \to L_{F_A}(X)) \\ \quad (p : (s : S(a)) \to g(F(a,s)) \equiv h(F(a,s))) \\ \quad (s : S(a)) : \text{extEq}(a, g, h, p, F(a,s) \equiv p(s) \end{cases}$$

```
data Localize (A X: U) (S T: A -> U) (F : (x:A) -> S x -> T x)
   = center (x: X)
   | ext (a: A) (f: S a -> Localize A X S T F) (t: T a)
   | isExt (a: A) (f: S a -> Localize A X S T F) (s: S a) <i>
     [ (i=0) -> ext a f (F a s) , (i=1) -> f s ]
   | extEq (a: A) (g h: T a -> Localize A X S T F)
     (p: (s : S a) -> Path (Localize A X S T F) (g (F a s)) (h (F a s)))
     (t : T a) <i> [ (i=0) -> g t , (i=1) -> h t ]
   | isExtEq (a: A) (g h : T a -> Localize A X S T F)
     (p: (s : S a) -> Path (T a -> Localize A X S T F) (g (F a s)) (h (F a s)))
     (s : S a) <i> [ (i=0) -> extEq a g h p (F a s) , (i=1) -> p s ]
```

Theorem 70. (Localization Induction) For any $P : \Pi_{X:U} L_{F_A}(X) \to U$ with $\{n, r, s\}$, satisfying coherence conditions, there exists $i : \Pi_{x:L_{F_A}(X)} P(x)$, such that $i \cdot \text{center}_X = n$.

Conclusion

HITs directly encode CW-complexes in HoTT, bridging topology and type theory. They enable the analysis and manipulation of homotopical types.

References

[1] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, IAS, 2013.

[2] C. Cohen, T. Coquand, S. Huber, A. Mörtberg, *Cubical Type Theory*, Journal of Automated Reasoning, 2018.

[3] A. Mörtberg et al., *Agda Cubical Library*, `https://github.com/agda/cubical`, 2023.

[4] M. Shulman, *Higher Inductive Types in HoTT*, `https://arxiv.org/abs/1705.07088`, 2017.

[5] J. D. Christensen, M. Opie, E. Rijke, L. Scoccola, *Localization in Homotopy Type Theory*, `https://arxiv.org/pdf/1807.04155.pdf`, 2018.

[6] E. Rijke, M. Shulman, B. Spitters, *Modalities in Homotopy Type Theory*, `https://arxiv.org/pdf/1706.07526v6.pdf`, 2017.

[7] M. Riley, E. Finster, D. R. Licata, *Synthetic Spectra via a Monadic and Comonadic Modality*, `https://arxiv.org/pdf/2102.04099.pdf`, 2021.

Issue V: Modalities and Identity Systems

Namdak Tonpa

June 1, 2025

Abstract

This article explores the interplay between modalities, identity systems, and homologies in the framework of Homotopy Type Theory (HoTT). We formalize modalities and identity systems as structures within $(\infty,1)$-categories and investigate the homological properties arising when their functor compositions are treated as groups. Special attention is given to topological structures, such as the Möbius strip, that emerge from non-trivial compositions, and their role in generating non-trivial fundamental groups. A classification of generators is provided, highlighting their categorical and homotopical properties.

9 Modalities and Identity Systems

Homotopy Type Theory (HoTT) provides a powerful framework for studying categorical structures through the lens of types, paths, and higher homotopies. In this context, *modalities* and *identity systems* serve as fundamental constructs that encode localization and identification properties, respectively. When compositions of their associated functors are interpreted as groups, they give rise to homological structures, such as fundamental groups, that can model complex topological spaces like the Möbius strip. This article formalizes these concepts and explores their implications in $(\infty,1)$-toposes, with a focus on the emergence of CW-complexes and homologies.

9.1 Modality

Definition 119 (Modality)**.** A modality in HoTT is a structure comprising:

```
def Modality :=
  Σ (modality: U → U)
    (isModal  : U → U)
    (eta:       Π (A : U), A → modality A)
    (elim:      Π (A : U) (B : modality A → U)
                  (B–Modal : Π (x : modality A), isModal (B x))
                  (f: Π (x : A), (B (eta A x))),
                  (Π (x : modality A), B x))
    (elim–β :   Π (A : U) (B : modality A → U)
                  (B–Modal : Π (x : modality A), isModal (B x))
                  (f : Π (x : A), (B (eta A x))) (a : A),
                  PathP (<_>B (eta A a)) (elim A B B–Modal f (eta A a)) (f a))
    (modalityIsModal : Π (A : U), isModal (modality A))
    (propIsModal : Π (A : U), Π (a b : isModal A),
                      PathP (<_>isModal A) a b)
    (=–Modal : Π (A : U) (x y : modality A),
                  isModal (PathP (<_>modality A) x y)), 1
```

where $\mathcal{U}$ is a universe of types, η is a natural inclusion, and elim provides a universal property for modal types (see [1] for details).

Modalities act as localization functors, projecting types onto subcategories of modal types. For instance, the *discrete modality* ($\flat$) trivializes higher homotopies, while the *codiscrete modality* ($\sharp$) makes types contractible.

9.2 Identity Systems

Definition 120 (Identity System)**.** For a type $A : \mathcal{U}$, an identity system is defined as:

```
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)),
                 Path (C a a (=-ctor a)) d
                      (=-elim a C d a (=-ctor a))), 1
```

where = -form generalizes the identity type, and = -ctor ensures reflexivity.

Identity systems generalize paths in HoTT, allowing the construction of types with non-trivial fundamental groups, such as the Möbius strip, where identifications generate $\mathbb{Z}$.

9.3 Classification of Generators

The following table classifies key generators, including modalities and identity systems, based on their categorical and homotopical properties.

Table 1: Classification of Generators in Homotopy Type Theory

Generator	**Notation**	**Type**	**Adjunction**
Discrete	$\flat$	Modality	$\flat \dashv \sharp$
Codiscrete	$\sharp$	Comodality	$\flat \dashv \sharp$
Bosonic	$\bigcirc$	Modality	$\bigcirc \dashv \bigcirc^{+}$
Fermionic/Infinitesimal	$\Im$	Modality	$\Im \dashv \Im^{+}$
Rheonomic	Rh	Modality	—
Reduced	$\Re$	Modality	—
Polynomial	W	Inductive	—
Polynomial	M	Coinductive	—
Higher Inductive	HIT	Inductive	HIT $\dashv$ Path
Higher Coinductive	CoHIT	Coinductive	Path $\dashv$ CoHIT
Path Spaces	**Path**	Identification	HIT $\dashv \Im$
Identity	$=$	Identification	—
Isomporphism	$\cong$	Identification	—
Equality	$\simeq$	Identification	—

9.4 Homologies from Functor Compositions

When functor compositions of modalities and identity systems are treated as groups, they generate homological structures, such as fundamental groups or homology groups. For example, consider the composition $\flat \circ \sharp \circ \flat$. In a topological context, this may correspond to a localization that preserves certain homotopical features, potentially yielding a CW-complex like the Möbius strip.

Theorem 71. Let $\mathcal{C}$ be an $(\infty,1)$-topos, and let $F = \flat \circ \sharp \circ \flat$ be a functor composition treated as a group action. The resulting structure induces a fundamental group isomorphic to $\mathbb{Z}$ for types modeling the Möbius strip.

Sketch. The Möbius strip can be constructed as a higher inductive type (HIT) with an identity system generating $\mathbb{Z}$. The functor $\flat$ discretizes the type, $\sharp$ contracts it, and the second $\flat$ reintroduces discrete structure, preserving the non-trivial loop in the identification system. The resulting type has a fundamental group $\pi_1 \cong \mathbb{Z}$. □

9.5 Topological Interpretation

The Möbius strip, as a CW-complex, arises naturally in this framework. Its non-trivial fundamental group is generated by an identity system, while modalities like $\Im$ or $\bigcirc$ introduce twisting or orientation properties. This connects to topological quantum field theories (TQFTs), where surfaces like the Möbius strip encode non-trivial symmetries.

9.6 Conclusion

Modalities and identity systems in HoTT provide a rich framework for modeling categorical and topological phenomena. By treating functor compositions as groups, we uncover homological structures that bridge type theory and topology. Future work may explore applications in TQFT and synthetic differential geometry.

References

[1] M. Shulman, *Brouwer's fixed-point theorem in real-cohesive homotopy type theory*, Mathematical Structures in Computer Science, 2018.

[2] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, 2013.