# Зміст


1 Volume I: Introduction 1
1.1 Martin-Löf Type Theory . . . 4
1.1.1 Interpretations . . . 4
1.1.2 Types . . . 6
1.1.3 Contexts . . . 13
1.1.4 Universes . . . 13
1.1.5 MLTT-75 . . . 15
1.2 Inductive Types . . . 22
1.2.1 W . . . 22
1.2.2 Empty . . . 23
1.2.3 Unit . . . 24
1.2.4 Bool . . . 25
1.2.5 Maybe . . . 25
1.2.6 Either . . . 25
1.2.7 Nat . . . 25
1.2.8 List . . . 25
1.3 Inductive Encodings . . . 26
1.3.1 Church Encoding . . . 26
1.3.2 Impredicative Encoding . . . 26
1.3.3 The Unit Example . . . 26
1.4 Introduction . . . 29
1.4.1 Introduction: Type Theory . . . 29
1.4.2 Motivation: Homotopy Type Theory . . . 30
1.4.3 Metatheory: Adjunction Triples . . . 31
1.4.4 Historical Notes . . . 32
1.4.5 Acknowledgement . . . 32
1.5 Homotopy Type Theory . . . 33
1.5.1 Groupoid Interpretation . . . 33
1.5.2 Identity Systems . . . 36
1.5.3 Path (Ξ) . . . 37
1.5.4 Glue . . . 43
1.5.5 Fibration . . . 44
1.5.6 Equivalence . . . 47
1.5.7 Homotopy . . . 49
1.5.8 Isomorphism . . . 51
1.5.9 Univalence . . . 52
1.5.10 Loop Spaces . . . 53
1.5.11 Homotopy Groups . . . 54

1.5.12 Hopf Fibrations . . . . . 55
1.6 CW-Complexes . . . . . 57
1.6.1 Motivation for Higher Inductive Types . . . . . 59
1.6.2 HITs with Countable Constructors . . . . . 59
1.7 Higher Inductive Types . . . . . 59
1.7.1 Suspension . . . . . 60
1.7.2 Pushout . . . . . 61
1.7.3 Spheres . . . . . 62
1.7.4 Hub and Spokes . . . . . 63
1.7.5 Truncation . . . . . 64
1.7.6 Quotient Spaces . . . . . 65
1.7.7 Wedge . . . . . 66
1.7.8 Smash Product . . . . . 67
1.7.9 Join . . . . . 69
1.7.10 Colimit . . . . . 70
1.7.11 Coequalizers . . . . . 71
1.7.12 K(G,n) . . . . . 73
1.7.13 Localization . . . . . 74
1.8 Conclusion . . . . . 74

Розділ 1

# Volume I: Introduction

# Issue I: Type Theory

Максим Сохацький [1]

[1] Національний технічний університет України
Київський політехнічний інститут імені Ігоря Сікорського
26 квітня 2025 р.

Abstract:

Background. The long road from pure type systems of AUTOMATH by de Bruijn to type checkers with homotopical core was made. This article touches only the formal Martin-Löf Type Theory core type system with Π and Σ types (that correspond to ∀ and ∃ quantifiers for mathematical reasoning) and identity type (MLTT-75). Expressing the MLTT embedding in a host type checker for a long time was inaccessible due to the non-derivability of the J eliminator in pure functions. This was recently made possible by cubical type theory and cubical type checker.
Objective. Select the type system as a part of conceptual model of theorem proving system that is able to derive the J eliminator and its theorems based on the latest research in cubical type systems. The goal of this article is to demonstrate the formal embedding of MLTT-75 into **Per** with constructive proofs of the complete set of inference rules including J eliminator.
Methods. As types are formulated using 5 types of rules (formation, intro, elimination, computation, uniqueness) according to MLTT we constructed aliases for the host language primitives and used the cubical type checker to prove that it has the realization of MLTT-75.
Results. This work leads to several results: 1) **Per** — a special embedded version of type theory with infinite number of universes and Path type suitable for HoTT purposes without uniqueness rule of equality type; 2) The actual embedding of MLTT with syntax implying universe polymorphism and cubical primitives in **Per**; 3) The different interpretations of types were given: set-theoretical, groupoid, homotopical;
4) Internalization could be seen as an ultimate test sample for type checker as intro-elimination fusion resides in beta-eta rules, so by proving them, we prove properties of the host type checker.
Conclusion. We should note that this is an entrance to the internalization technique, and after formal MLTT embedding, we could go through inductive types up to embedding of CW-complexes as the indexed gluing of the higher inductive types. This means the implementation of a wide spectrum of math theories inside HoTT up to algebraic topology. Keywords: Martin-Löf Type Theory, Cubical Type Theory.

# Зміст

## Introduction

Each language implementation needs to be checked. The one of possible test cases for type checkers is the direct embedding of type theory model into the language of type checker. As types in Martin-Löf Type Theory [?, ?] are formulated using 5 types of rules (formation, introduction, elimination, computation, uniqueness), we construct aliases for host language primitives and use type checker to prove that it is MLTT-75. This could be seen as ultimate test sample for type checker as intro-elimination fusion resides in beta-eta rules, so by proving them we prove properties of the host type checker.

Also this issue opens a series of articles dedicated to formalization in cubical type theory the foundations of mathematics. This issue is dedicated to MLTT modeling and its verification. Also

as many may not be familiar with Π and Σ types, this issue presents different interpretation of MLTT types.

This test is fully made possible only after 2017 when new constructive HoTT [?] prover cubicaltt[1] prover was presented [?].

### Problem Statement

The formal initial problem was to create a full self-contained MLTT internalization in the host typechecker, where all theorems are being checked constructively. This task involves a modern techniques in type theory, namely cubical type theories. By following most advaced theories apply this results for building minimal type checker that is able to derive J and the whole MLTT theorems constructively. This leads us to the compact MLTT core yet compatible with future possible homotopy extensions.

### **Per** Language Syntax

The BNF notation of type checker language used in code samples consists of: i) telescopes (contexts or sigma chains) and definitions; ii) pure dependent type theory syntax; iii) inductive data definitions (sum chains) and split eliminator; iv) cubical face system; v) module system. It is slightly based on cubicaltt.

```
   sys := [ sides ]
  side := (id=0)→exp +(id=1)→exp
    f1 := f1 /\ f2
    f2 := −f2 + id + 0 + 1
  form := form \/ f1 + f1 + f2
 sides := #empty + cos + side
   cos := side , side + side , cos
    id := #list #nat
   ids := #list id
   mod := module id where imps dec
  imps := #list imp
   imp := import id
   brs := #empty + cobrs
 cobrs := | br brs
    br := ids → exp +ids @ ids → exp
   tel := #empty + cotel
   dec := #empty + codec
 cotel := (exp:exp) tel
 codec := def dec
   sum := #empty + id tel + id tel | sum
   def := data id tel=sum +id tel:exp=exp
        + id tel : exp where def
   app := exp exp
   exp := cotel * exp + cotel → exp
        + exp → exp +(exp) +id
        + (exp,exp) + \cotele → exp
        + split cobrs +exp .1
        + exp .2 +⟨ ids ⟩ exp
        + exp @ form + app + comp exp sys
```

[1] http://github.com/mortberg/cubicaltt

Here := (definition), + (disjoint sum), #empty, #nat, #list are parts of BNF language and |, :, *, ⟨, ⟩, (, ), =, \, /, -, →, 0, 1, @, [, ], **module**, **import**, **data**, **split**, **where**, **comp**, **.1**, **.2**, and , are terminals of type checker language. This language includes inductive types, higher inductive types and gluening operations needed for both, the constructive homotopy type theory and univalence. All these concepts as a part of the languages will be described in the upcoming Issues II — V.

## 1.1 Martin-Löf Type Theory

Martin-Löf Type Theory MLTT-80 contains Π, Σ, Id, W, 0, 1, 2 types.

Any new type in MLTT presented with set of 5 rules: i) formation rules, the signature of type; ii) the set of constructors which produce the elements of formation rule signature; iii) the dependent eliminator or induction principle for this type; iv) the beta-equality or computational rule; v) the eta-equality or uniquness principle. Π, Σ, and Path types will be given shortly. This interpretation or rather way of modeling is MLTT specific.

The most interesting are Id types. Id types were added in MLTT-75 [?] while original MLTT-72 with only Π and Σ was introduced in [?]. Predicative Universe Hierarchy was added in [?]. While original MLTT-75 contains Id types that preserve uniquness of identity proofs (UIP) or eta-rule of Id type, HoTT refutes UIP (eta rule desn't hold) and introduces univalent heterogeneous Path equality [?].

Path types are essential to prove computation and uniquness rules for all types (needed for building signature and terms), so we will be able to prove all the MLTT rules as a whole.

### 1.1.1 Interpretations

In contexts you can bind to variables (through de Brujin indexes or string names): i) indexed universes; ii) built-in types; iii) user constructed types, and ask questions about type derivability, type checking and code extraction. This system defines the core type checker within its language.

By using this languages it is possible to encode different interpretations of type theory itself and its syntax by construction. Usually the issues will refer to following interpretations: i) type-theoretical; ii) categorical; iii) set-theoretical; iv) homotopical; v) fibrational or geometrical.

#### Logical or Type-theoretical interpretation

According to type theoretical interpretation of MLTT for any type should be provided 5 formal inference rules: i) formation; ii) introduction; iii) dependent elimination principle; iv) beta rule or computational rule; v) eta rule or uniqueness rule. The last one could be exceptional for Path types. The formal representation of all rules of MLTT are given according to type-theoretical interpretation as a final result in this Issue I. It was proven that classical Logic could be embedded into intuitionistic propositional logic (IPL) which is directly embedded into MLTT.

Logical and type-theoretical interpretations could be distincted. Also set-theoretical interpretation is not presented in the Table.

Табл. 1.1: *
Table. Interpretations correspond to mathematical theories

| 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 |
| $\emptyset$ | $\bot$ false | initial object | empty space |
| **1** | $\top$ true | terminal object | singleton |
| $A + B$ | $A \vee B$ 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 | |

Categorical or Topos-theoretical interpretation

Categorical interpretation [?] is a modeling through categories and functors. First category is defined as objects, morphisms and their properties, then we define functors, etc. In particular, as an example, according to categorical interpretation Π and Σ types of MLTT are presented as adjoint functors, and forms itself a locally closed cartesian category, which will be given an intermediate result in future issues. In some sense we include here topos-theoretical interpretations, with presheaf model of type theory as example (in this case fibrations are constructes as functors, categorically).

Homotopical interpretation

In classical MLTT uniquness rule of Id type do holds strictly. In Homotopical interpretation of MLTT we need to allow a path space as Path type where uniqueness rule doesn't hold. Groupoid interpretation of Path equality that doesn't hold UIP generally was given in 1996 by Martin Hofmann and Thomas Streicher [?].

When objects are defined as fibrations, or dependent products, or indexed-objects this leds to fibrational semantics and geometric sheaf interpretation. Several definition of fiber bundles and trivial fiber bindle as direct isomorphisms of Π types is given here as theorem. As fibrations study in homotopical interpretation, geometric interpretation could be treated as homotopical.

Set-theoretical interpretation

Set-theoretical interpretations could replace first-order logic, but could not allow higher equalities, as long as inductive types to be embedded directly. Set is modelled in type theory according to homotopical interpretation as n-type.

### 1.1.2 Types

MLTT-80 could be reduced to Π, Σ, Path types (MLTT-75) omitting polynomial functors W modeled by F-algebras and their terminators: 0, 1, 2 types. In this issue Π, Σ, Path are given as a core of MLTT-75. The inductive types will be disscussed in the upcoming Issue II: Inductive Types.

Π-type

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 1.1.1. (Π-Formation).

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

```
Pi (A: U) (B: A -> U): U = (x: A) -> B x
```

Definition 1.1.2. (Π-Introduction).

$$\backslash(x : A) \to b =_{def} \prod_{A:U} \prod_{B:A\to U} \prod_{a:A} \prod_{b:B(a)} \lambda x.b : \prod_{y:A} B(a).$$

```
lambda (A B: U) (b: B): A -> B = \ (x: A) -> b
lam (A:U) (B: A -> U) (a:A) (b:B a)
    : A -> B a = \ (x: A) -> b
```

Definition 1.1.3. (Π-Elimination).

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

```
apply (A B: U) (f: A -> B) (a: A) : B = f a
app (A: U) (B: A -> U) (a: A)
    (f: A -> B a) : B a = f a
```

Theorem 1.1.1. (Π-Computation).

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

```
Beta (A: U) (B: A -> U) (a: A) (f: A -> B a)
   : Path (B a) (app A B a (lam A B a (f a)))
                (f a)
```

Theorem 1.1.2. (Π-Uniqueness).

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

```
Eta (A: U) (B: A -> U) (a: A) (f: A -> B a)
  : Path (A -> B a) f (\(x:A) -> f x)
```

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 1.1.4. (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 1.1.5. (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 1.1.3. (HomSet). If codomain is set then space of sections is a set.

```
setFun (A B : U) (_: isSet B)
  : isSet (A -> B)
```

Theorem 1.1.4. (Contractability). If domain and codomain is contractible then the space of sections is contractible.

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

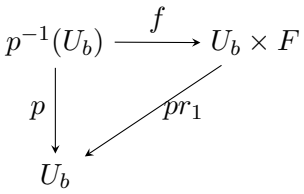
Definition 1.1.6. (Section). A section of morphism $f : A \to B$ in some category is the morphism $g : B \to A$ such that $f \circ g : B \xrightarrow{g} A \xrightarrow{f} B$ equals the identity morphism on B.

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 1.1.7. (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 1.1.8. (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.




Definition 1.1.9. (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

$$F \times B \xrightarrow{app} E \xrightarrow{pr_1} B \tag{1.1}$$

$pr_1$ 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 1.1.10. (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 1.1.5. (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 1.1.6. (Trivial Fiber equals Family of Sets). Inverse image (fiber) of fiber bundle $(F, B * F, pr_1, B)$ in point $y : B$ equals $F(y)$.

```
FiberPi (B: U) (F: B -> U) (y: B)
  : Path U (fiber (Sigma B F) B (pi1 B F) y)
           (F y)
```

Theorem 1.1.7. (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.

```
setPi (A: U) (B: A -> U)
    (h: (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
```

Note that we will not be able to prove this theorem until Issue III: Homotopy Type Theory because bi-invertible iso type will be announced there.

### Σ-type

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.

### Type-theoretical interpretation

Definition 1.1.11. (Σ-Formation).

```
Sigma (A : U) (B : A -> U)
  : U = (x : A) * B x
```

Definition 1.1.12. (Σ-Introduction).

```
dpair (A: U) (B: A -> U) (a: A) (b: B a)
  : Sigma A B = (a,b)
```

Definition 1.1.13. (Σ-Elimination).

```
pr1 (A: U) (B: A -> U)
    (x: Sigma A B): A = x.1

pr2 (A: U) (B: A -> U)
    (x: Sigma A B): B (pr1 A B x) = x.2

sigInd (A: U) (B: A -> U)
       (C: Sigma A B -> U)
       (g: (a: A) (b: B a) -> C (a, b))
       (p: Sigma A B) : C p = g p.1 p.2
```

Theorem 1.1.8. (Σ-Computation).

```
Beta1 (A: U) (B: A -> U)
      (a:A) (b: B a)
    : Equ A a (pr1 A B (a,b))

Beta2 (A: U) (B: A -> U)
      (a: A) (b: B a)
    : Equ (B a) b (pr2 A B (a,b))
```

Theorem 1.1.9. (Σ-Uniqueness).

```
Eta2 (A: U) (B: A -> U) (p: Sigma A B)
   : Equ (Sigma A B) p (pr1 A B p,pr2 A B p)
```

### Categorical interpretation

Definition 1.1.14. (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 1.1.10. (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 $R(x, f(x))$.

```
ac (A B: U) (R: A -> B -> U)
  : (p: (x:A) -> (y:B)*(R x y))
-> (f:A->B) * ((x:A)->R(x)(f x))
```

Theorem 1.1.11. (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.

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

Theorem 1.1.12. (Σ-Contractability). If the fiber is set then the Σ is set.

```
setSig (A:U) (B: A -> U) (sA: isSet A)
       (sB : (x:A) -> isSet (B x))
     : isSet (Sigma A B)
```

Theorem 1.1.13. (Path Between Sigmas). Path between two sigmas $t, u : \Sigma(A, B)$ could be decomposed to sigma of two paths $p : t_1 =_A u_1)$ and $(t_2 =_{B(p@i)} u_2)$.

```
pathSig (A:U) (B : A -> U) (t u : Sigma A B)
      : Path U (Path (Sigma A B) t u)
               ((p: Path A t.1 u.1)
              * PathP (<i>B(p@i)) t.2 u.2)
```

### Ξ-type

The Path identity type or Ξ defines a Path space with elements and values. Elements of that space are functions from interval $[0, 1]$ to a values of that path space. 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.

### Cubical interpretation

Cubical interpretation was first given by Simon Huber [?] and later was written first constructive type checker in the world by Anders Mörtberg [?].

Definition 1.1.15. (Path Formation).

```
Hetero (A B: U)(a: A)(b: B)(P: Path U A B)
  : U = PathP P a b
Path (A: U) (a b: A)
  : U = PathP (<i> A) a b
```

[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

Definition 1.1.16. (Path Reflexivity). Returns an element of 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 a. Written in syntax as |<i>a| which equals to $\lambda\ (i : I) \to a$.

```
refl (A: U) (a: A) : Path A a a
```

Definition 1.1.17. (Path Application). You can apply face to path.

```
app1 (A: U)(a b:A)(p:Path A a b):A=p@0
app2 (A: U)(a b:A)(p:Path A a b):A=p@1
```

Definition 1.1.18. (Path Composition). Composition operation allows to build a new path by given to paths in a connected point.

$$\begin{array}{ccc} & \xrightarrow{comp} & \\ a & & c \\ \uparrow & & \uparrow q \\ \lambda(i:I) \to a & & \\ \downarrow & & \\ a & \xrightarrow{p@i} & b \end{array}$$

```
composition
    (A: U) (a b c: A)
    (p: Path A a b) (q: Path A b c)
  : Path A a c
  = comp (<i>Path A a (q@i)) p []
```

Theorem 1.1.14. (Path Inversion).

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

Definition 1.1.19. (Connections). Connections allows you to build square with given only one element of path: i) $\lambda\ (i, j : I) \to p\ @\ min(i, j)$; ii) $\lambda\ (i, j : I) \to p\ @\ max(i, j)$.

$$\begin{array}{ccc} & \xrightarrow{p} & \\ a & & b \\ \uparrow & & \uparrow p \\ \lambda\ (i:I) \to a & & \\ \downarrow & & \\ a & \xrightarrow{\lambda\ (i:I) \to a} & a \end{array} \qquad \begin{array}{ccc} & \xrightarrow{\lambda\ (i:I) \to b} & \\ b & & b \\ \uparrow p & & \uparrow \\ & & \lambda\ (i:I) \to b \\ \downarrow & & \\ a & \xrightarrow{p} & b \end{array}$$

```
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)

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 1.1.15. (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 |λ (i:I) → f (p @ i)| for both cases.

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

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 1.1.16. (Transport). Transports a value of the domain type to the value of the codomain type by a given path element of the path space between domain and codomain types. Defined as path composition with |[]| of a over a path $p$ — |comp p a []|.

```
trans (A B: U) (p: Path U A B) (a: A) : B
```

Type-theoretical interpretation

Definition 1.1.20. (Singleton).

```
singl (A: U) (a: A): U = (x: A) * Path A a x
```

Theorem 1.1.17. (Singleton Instance).

```
eta (A: U) (a: A): singl A a = (a,refl A a)
```

Theorem 1.1.18. (Singleton Contractability).

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

Theorem 1.1.19. (Path Elimination, Paulin-Mohring). J is formulated in a form of Paulin-Mohring and implemented using two facts that singleton are contractible and dependent function transport.

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

Theorem 1.1.20. (Path Elimination, HoTT). J from HoTT book.

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

Theorem 1.1.21. (Path Computation).

```
trans_comp (A: U) (a: A)
  : Path A a (trans A A (<_> A) a)
  = fill (<i> A) a []
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)
  = trans_comp (P a) e
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))
  = subst_comp (singl A a) T (eta A a) d
    where T (z:singl A a)
        : U = C a (z.1) (z.2)
```

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.

### 1.1.3 Contexts

Speaking of type checker execution, we introduce context or dictionary with types and terms, from which we can derive typed variables. This chain could be implemented as nested sigma types (due to R.A.G.Seely) or list types (due to Voevodsky). Categorically dependent type theory is built upon categories of contexts.

Definition 1.1.21. (Empty Context).

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

Definition 1.1.22. (Context Comprehension).

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

Definition 1.1.23. (Context Derivability).

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

### 1.1.4 Universes

Definition 1.1.24. (Terms). Point in initial object of language AST inductive definition is called a term. If type theory or language is defined as an inductive type (AST) then the term is defined as its instance.

Definition 1.1.25. (Sorts). N-indexed set of universes $\mathrm{U}_{n\in\mathrm{N}}$. Could have any number of elements which defines different type systems. All built-in types as long as user defined types are landed usually by default in $U_0$ universe. Sorts represented in type checker as a separate constructor.

Definition 1.1.26. (Axioms). The inclusion rules $\mathrm{U_i} : \mathrm{U_j}, i, j \in \mathrm{N}$, that define which universe is element of another given universe. You may attach any rules that joins $i, j$ in some way. Axioms with sorts define universe hierarchy.

Definition 1.1.27. (Rules). The set of landings $\mathrm{U_i} \to \mathrm{U_j} : \mathrm{U}_{\lambda(\mathrm{i,j}),\mathrm{i,j}\in\mathrm{N}}$, where $\lambda : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$. These rules define term dependence or how we land (in which universe) formation rules in definitions.

Definition 1.1.28. (Predicative hierarchy). If $\lambda$ in Rules is an uncurried function $\max : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$ then such universe hierarchy is called predicative.

Definition 1.1.29. (Impredicative hierarchy). If $\lambda$ in Rules is a second projection of a tuple $\mathrm{snd} : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$ then such universe hierarchy is called impredicative.

Definition 1.1.30. (Definitional Equality). For any $\mathrm{U}_i, i \in \mathrm{N}$ there is defined an equality between its members and between its instances. For all x,y ∈ A, there is defined a x=y. Definitional equality compares normalized term instances.

Definition 1.1.31. (SAR). The universum space is configured with a triple of: i) sorts, a set of universes $\mathrm{U}_{n\in\mathrm{N}}$ indexed over set N; ii) axioms, a set of inclusions $\mathrm{U_i} : \mathrm{U_j}, i, j \in \mathrm{N}$; iii) rules of term dependence universe landing, a set of landings $\mathrm{U_i} \to \mathrm{U_j} : \mathrm{U}_{\lambda(\mathrm{i,j}),\mathrm{i,j}\in\mathrm{N}}$, where $\lambda$ could be function *max* (predicative) or *snd* (impredicative).

Example 1.1.1. (CoC). SAR = $\{\{\star, \square\}, \{\star : \square\}, \{i \to j : j; i, j \in \{\star, \square\}\}$. Terms live in universe $\star$, and types live in universe $\square$. In CoC $\lambda = \mathrm{snd}$.

Example 1.1.2. ($\mathrm{PTS}^\infty$, $\mathrm{MLTT}^\infty$).
SAR = $\{\mathrm{U}_{i\in\mathrm{N}}, \mathrm{U_i} : \mathrm{U}_{\mathrm{j;i<j;i,j}\in\mathrm{N}}, \mathrm{U_i} \to \mathrm{U_j} : \mathrm{U}_{\lambda(\mathrm{i,j});\mathrm{i,j}\in\mathrm{N}}\}$. Where $U_i$ is a universe of $i$-level or $i$-category in categorical interpretation. The working prototype of $\mathrm{PTS}^\infty$ is given in Issue XVI: Pure Type System [?].

### 1.1.5 MLTT-75

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 1.1.32. (MLTT-75). The MLTT as a Type is defined by taking all rules for Π, Σ and Path types into one Σ telescope or context.

```
MLTT (A: U): U
  = (Pi_Former: (A → U) → U)
  * (Pi_Intro: (B: A → U) (a: A) → B a → (A → B a))
  * (Pi_Elim: (B: A → U) (a: A) → (A → B a) → B a)
  * (Pi_Comp₁: (B: A → U) (a: A)
    (f: A → B a) → Path (B a)
    (Pi_Elim B a(Pi_Intro B a(f a)))(f a))
  * (Pi_Comp₂: (B: A → U) (a: A)
    (f: A → B a) → Path (A → B a) f (\(x:A) → f x))
  * (Sigma_Former: (A → U) → U)
  * (Sigma_Intro: (B: A → U) (a: A) (b: B a) → Sigma A B)
  * (Sigma_Elim1: (B: A → U)
    (_: Sigma A B) → A)
  * (Sigma_Elim2: (B: A → U)
    (x: Sigma A B) → B (pr1 A B x))
  * (Sigma_Comp1: (B: A → U) (a: A) (b: B a)
      → Path A a (Sigma_Elim1 B (Sigma_Intro B a b)))
  * (Sigma_Comp2: (B: A → U) (a: A)
    (b: B a) → Path (B a) b
    (Sigma_Elim2 B (a,b)))
  * (Sigma_Comp3: (B: A → U) (p: Sigma A B)
      → Path (Sigma A B) p (pr1 A B p,pr2 A B p))
  * (Id_Former: A → A → U)
  * (Id_Intro: (a: A) → Path A a a)
  * (Id_Elim: (x: A) (C: D A)
    (d: C x x (Id_Intro x))
    (y: A) (p: Path A x y) → C x y p)
  * (Id_Comp: (a:A)(C: D A)
    (d: C a a (Id_Intro a)) →
    Path (C a a (Id_Intro a))
       d (Id_Elim a C d a (Id_Intro a))) * U
```

Theorem 1.1.22. (Model Check). There is an instance of MLTT.

```
instance (A: U): MLTT A
  = (Pi A,      lam A, app A,
                Beta A, Eta A,
     Sigma A,   dpair A, pr1 A, pr2 A,
                Beta1 A, Beta2 A, Eta2 A,
     Path A,    refl A, J A,
                J_comp A, A)
```

### Cubical Model Check

The result of the work is a |mltt.ctt| file which can be runned using |cubicaltt|. Note that computation rules take a seconds to type check.

```
cubicaltt — 6 second.
Arend — 1 second.
Agda (cubical) — & 2 second.
```

### Conclusions

In this issue the type-theoretical model (interpretation) of MLTT was presented in cubical syntax and type checked in it. This is the first constructive proof of internalization of MLTT.

From the theoretical point of view the landspace of possible interpretation was shown corresponding different mathematical theories for those who are new to type theory. The brief description of the previous attempts to internalize MLTT could be found as canonical example in MLTT works, but none of them give the constructive J eliminator or its equality rule.

Type theoretical cubical constructions was given for the Path types along the article for other interpretations, all of them were taken from our Groupoid Infinity [7] base library.

Табл. 1.2: *
Table. Core Features

| Lang | $\Pi$ | $\Sigma$ | $\equiv$ | Path | $U^\infty$ | Co/Fix | Lazy |
|---|---|---|---|---|---|---|---|
| PTS | x | | | | | | |
| Cedile, MLTT | x | x | x | | | | |
| Henk | x | | | | x | | |
| Per | x | x | | x | x | | |
| Lean, Agda | x | | x | x | x | | |
| NuPRL | x | x | x | | | x | |
| System-D | x | x | | | | x | x |
| cubicaltt | x | x | x | x | | x | |

[7] https://groupoid.space/

The objective of complete derivability of all eliminators, computational and uniquness rules is a basic objective for constructive mathematics as mathematical reasoning implies verification and mechanization. Yes cubical type system represent most compact system that make possible derivability of all theorems for core types which make this system as a first candidate for the metacircular type checker.

Also for programming purposes we may also want to investigate Fixpoint as a useful type in coinductive and modal type theories and harmful type in theoretical foundation of type systems. Elimination the possibility of uncontrolled Fixpoint is a main objective of the correct type system for reasoning without paradoxes. By this creatiria we could filter all the fixpoint implementations being condidered harmful.

Without a doubt the core type that makes type theory more like programming is the inductive type system that allows to define type families. In the following Issue II will be shown the semantics and embedding of inductive types with several types of Inductive-Recursive encodings.

Табл. 1.3: *
Table. Inductive Type Systems

| Lang | Co/Inductive | Quot/Trunc | HITs |
|---|---|---|---|
| System-D | x | | |
| Lean | x | x | |
| NuPRL | x | x | |
| Arend | x | x | x |
| Agda, Coq | x | | x |
| cubicaltt, yacctt, RedPRL | x | | x |

Further research of the most pure type theory on a weak fibrations and pure Kan oprations without interval lattice structure (connections, de Morgan algebra, connection algebras) and diagonal coersions could be made on the way of building a minimal homotopy core [?].

Табл. 1.4: *
Table. Cubical Type Systems

| Lang | Interval | Diagonal | Kan/Coe |
|---|---|---|---|
| BCH, cubical | | | $0 \to r$, $1 \to r$ |
| CCHM, cubicaltt, Agda | $\vee$,$\wedge$ | | $0 \to 1$ |
| Dedekind | $\vee$,$\wedge$ | | $0 \to 1$, $1 \to 0$ |
| AFH/ABCFHL, yacctt | | x | $r \to s$ |
| HTS/CMS | | | $r \to s$, $weak$ |

The next language after **Henk** and **Per** will be **Anders** with homotopy type system and infinite number of universes. Along with **Joe** cartesian interpreter this evaluators form a set of languages as a part of conceptual model of theorem proving system with formalized virtual machine as

extraction target.

### Further Research

This article opens the door to a series that will unvail the different topics of homotopy type theory with practical emphasis to cubical type checkers. The Foundations volume of articles define formal programming language with geometric foundations and show how to prove properties of such constructions. The second volume of article is dedicated to cover the programming and modeling of Mathematics.

Issue I: Type Theory. The first volume of definitions gathered into one article dedicated to various $\prod$, $\sum$ and $\equiv$ properties and internalization of MLTT in the host language typechecker.

Issue II: Inductive Types. This episode tales a story of inductive types, their encodings, induction principle and its models.

Issue III: Homotopy Type Theory. This issue is try to present the Homotopy Type Theory without higher inductive types to neglect the core and principles of homotopical proofs.

Issue IV: Higher Inductive Types. The metamodel of HIT is a theory of CW-complexes. The category of HIT is a homotopy category. This volume finalizes the building of the computational theory.

Issue V: Modalities. The constructive extensions with additional context and adjoint transports between toposes (cohesive toposes). This approach serves the needs of modal logics, differential geometry, cohomology.

The main intention of Foundation volume is to show the internal language of working topos of CW-complexes, the construction of fibrational sheaf type theory.

Issue XVI: Pure Type System. Pure Type System named after **Henk** Barendregt.

Issue XVII: Inductive Type System. Inductive Type System named after **Per** Martin-Löf.

Issue XIX: Modal Homotopy Type System. Modal Homotopy Type System named after **Anders** Mörtberg.

Проблематика. Був пройдений довгий шлях від чистих типових систем AUTOMATH де Брейна до гомотопічних типових верифікаторів. Ця стаття стосується тільки формального ядра теорії типів Мартіна-Льофа: Π, Σ і Ξ типів (які відповідають квантору загальності ∀ та квантору існування ∃ у класичній логіці) та типу-рівності.
Мета дослідження. Визначити типову систему як частину моделі системи доведення теорем, у якій конструктивно виражається J елімінатор та його теореми, спираючись на більш абстрактні примітиви типа рівності. Це стало можливим завдяки кубічній теорії типів (2016) та типовому кубічному верифікатору cubicaltt[8] (2017). Ціль статті — продемонструвати формальне вбудовування теорії типів Мартіна-Льофа в виконуючу авторську кубічну типову систему **Per** з повним набором правил виводу.
Методика реалізації. Так як всі типи в теорії формулюються за допомогою п'яти правил: формації, інтро, елімінації, обчисленя, рівності) що в сутності є кодуванням ізоморфізмами ініальних об'єктів в категорія F-алгебр, ми зконструювали номінальні типи-синоніми для виконуючого верифікатора та довели, що це є реалізацією MLTT. Так як не всі можуть бути знайомі з теорією типів, це випуск також містить їх інтерпретації з точки зору різних розділів математики.
Результати дослідження. Ця робота веде до декількох результатів: 1) **Per** — спеціальна версія теорії типів Мартіна-Льофа зі зліченною кількістю всесвітів та Path типом без eta-правила для HoTT застосування у яку ми будем вбудовувати класичну MLTT; 2) Інтерналізація MLTT в **Per** з синтаксисом який дозволяє виводити поліморфні всесвіти; 3) Класифіковані різні інтерпретації цієї системи типів: теоретико-типова, категорна або топосо-теоретична, гомотопічна або кубічна; 4) Це може розглядатися як універсальний тест для імплементації типового верифікатора, позаяк компенсаця інтро правила та правила елімінатора пов'язані в правилі обчислення та рівності (бета та ета редукціях). Таким чином, доводжучи реалізацію MLTT, ми доводимо властивості виконуючого верифікатора; 6) Завдяки позитивним результатм кубічна теорія була вибрана як геометричне розширення системи індуктивних типів для математичної верифікації як частина концептуальної системи доведення теорем, яка включатиме серію мов як середовище верифікації.
Висновки. Додамо, що це тільки вхід в техніку прямого вбудовування і після MLTT моделювання, ми можем піднятися вище — до вбудовування в систему індуктивних типів, і далі, до вбудовування CW-комплексів як зклейок вищих індуктивних типів, та далі до модальних логік. Це означає широкий спектр математичних теорії всередині HoTT аж до алгебраїчної топології.
Ключові слова: Теорія типів Мартіна-Льофа, Кубічна теорія типів.

Issue II: Inductive Types Maksym Sokhatskyi [1] [1] National Technical University of Ukraine Igor Sikorsky Kyiv Polytechnical Institute
26 квітня 2025 р.

Abstract:

Impredicative Encoding of Inductive Types in HoTT.

Keywords: Formal Methods, Type Theory, Programming Languages, Theoretical Computer

[8] http://github.com/mortberg/cubicaltt

Science, Applied Mathematics, Cubical Type Theory, Martin-Löf Type Theory

# Зміст

## 1.2 Inductive Types

### 1.2.1 W

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

Definition 1.2.1. (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 1.2.2. (W-Introduction). Elements of $\mathrm{W}_{(x:A)} B(x)$ are called well-founded trees and created with single sup constructor:

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

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

Theorem 1.2.1. (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

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

there is a dependent function:

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

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

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

$$=_{def} \mathrm{ind}_{\mathrm{W}}(g, \mathrm{sup}(a, f)).$$

```
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))
```

### 1.2.2 Empty

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

Definition 1.2.3. (Formation). Empty-type is defined as built-in $\mathbf{0}$-type:

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

Theorem 1.2.3. (Induction Principle $\mathrm{ind}_0$). $\mathbf{0}$-type is satisfying the induction principle:

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

```
def Empty_ind (C: 0 → U) (z: 0) : C z := ind₀ (C z) z
```

Definition 1.2.4. (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.

### 1.2.3 Unit

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

### 1.2.4 Bool

### 1.2.5 Maybe

### 1.2.6 Either

### 1.2.7 Nat

### 1.2.8 List

## 1.3 Inductive Encodings

### 1.3.1 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.

### 1.3.2 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 [?] and [?].

### 1.3.3 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
```

Issue III: Homotopy Type Theory Maksym Sokhatskyi [1] [1] National Technical University of Ukraine

Igor Sikorsky Kyiv Polytechnical Institute

26 квітня 2025 р.

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

# Зміст

## 1.4 Introduction

### 1.4.1 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 (**Π**)
- Context (**Σ**)
- Identification (=)
- Polynomial (**W**)
- Path (Ξ)
- Gluing (**Glue**)
- Infinitesimal (ℑ)
- Complex (**HIT**)

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

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

### 1.4.2 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 $\infty$-groupoid structure for path spaces. Further definitions include:

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

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

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

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

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

### 1.4.3 Metatheory: Adjunction Triples

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

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 ($\Pi$) and pair spaces ($\Sigma$). These proof methods enable direct analysis of fibrations.

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 ($\Xi$), and contractible space (C). These methods allow direct manipulation of identification systems, strict for set theory and homotopic for homotopy theory.

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).

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 ($\&$).

One additional part recently was dropped.

Linear Proofs

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

For engineering applications (e.g., Milner's $\pi$-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: $\mathbf{D}(A \otimes B, C) \simeq \mathbf{D}(A, [B, C])$.

### 1.4.4 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)[9].

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.

### 1.4.5 Acknowledgement

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

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

## 1.5 Homotopy Type Theory

### 1.5.1 Groupoid Interpretation

The first text about groupoid interpretation of type theory can be found in Francois Lamarche: A proposal about Foundations[10]. Then Martin Hofmann and Thomas Streicher wrote the initial document on groupoid interpretation of type theory[11].

| Equality | Homotopy | $\infty$-Groupoid |
|---|---|---|
| reflexivity | constant path | identity morphism |
| symmetry | inversion of path | inverse morphism |
| transitivity | concatenation of paths | composition of mopphisms |

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

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

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

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

[11] 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
```

### 1.5.2 Identity Systems

Definition 1.5.1. (Identity System). An identity system over type $A$ in universe $X_i$ is a family $R : A \to A \to X_i$ with a function $r_0 : \Pi_{a:A} R(a, a)$ such that any type family $D : \Pi_{a,b:A} R(a, b) \to X_i$ and $d : \Pi_{a:A} D(a, a, r_0(a))$, there exists a function $f : \Pi_{a,b:A} \Pi_{r:R(a,b)} D(a, b, r)$ such that $f(a, a, r_0(a)) = d(a)$ for all $a : A$.

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

Example 1.5.1. There are number of equality signs used in this tutorial, all of them listed in the following table of identity systems:

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

Theorem 1.5.1. (Fundamental Theorem of Identity System).

Definition 1.5.2. (Strict Identity System). An identity system over type $A$ and universe of pretypes $V_i$ is called strict identity system ($=$), which respects UIP.

Definition 1.5.3. (Homotopy Identity System). An identity system over type $A$ and universe of homotopy types $U_i$ is called homotopy identity system ($\equiv$), which models discrete infinity groupoid.

### 1.5.3 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 1.5.4. (Path Formation).

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

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

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

Definition 1.5.5. (Path Introduction). Returns a reflexivity path space for a given value of the type. The inhabitant of that path space is the lambda on the homotopy interval $[0,1]$ that returns a constant value $x$. Written in syntax as $[i]x$.

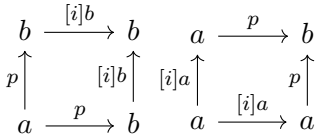
$$\mathrm{id}_{\equiv} : x \equiv_A x =_{def} \prod_{A:U} \prod_{x:A} [i]x$$

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

Definition 1.5.6. (Path Application).

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

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

Definition 1.5.7. (Path Connections). Connections allow you to build a square with only one element of path: i) $[i,j]p$ @ $min(i,j)$; ii) $[i,j]p$ @ $max(i,j)$.




```
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 1.5.8. (Path Inversion).

Theorem 1.5.2. (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 1.5.9. (Generalized Transport Kan Operation). Transports a value of the left type to the value of the right type by a given path element of the path space between left and right types.

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

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

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

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

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

Definition 1.5.10. (Partial Elements).

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

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

Definition 1.5.11. (Cubical Subtypes).

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

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

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

Definition 1.5.12. (Cubical Elements).

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

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

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

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

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

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

Theorem 1.5.3. (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 1.5.4. (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 1.5.5. (Substitution).

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

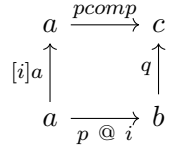
$$\prod_{A:U} \prod_{P:A\to U} \prod_{x,y:A} \prod_{p:x=y}$$

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

```
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 1.5.6. (Path Composition).




```
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 1.5.7. (J by Paulin-Mohring).

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

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

Theorem 1.5.8. (Contractability of Singleton).

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

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

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

Theorem 1.5.9. (HoTT Dependent Eliminator).

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

Theorem 1.5.10. (Diagonal Path Induction).

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

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

Theorem 1.5.11. (Path Computation).

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

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

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

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

### 1.5.4 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 1.5.13. (Glue Formation). The Glue types take a partial family of types A that are equivalent to the base type B. These types are then "glued" onto B and the equivalence data gets packaged up into a new type.

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

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

Definition 1.5.14. (Glue Introduction).

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

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

Definition 1.5.15. (Glue Elimination).

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

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

Theorem 1.5.12. (Glue Computation).

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

Theorem 1.5.13. (Glue Uniqueness).

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

### 1.5.5 Fibration

Definition 1.5.16 (Fiber). The fiber of the map $p : E \to B$ at a point $y : B$ is the set of all points $x : E$ such that $p(x) = y$.

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

Definition 1.5.17 (Fiber Bundle). The fiber bundle $F \to E \xrightarrow{p} B$ on a total space $E$ with fiber layer $F$ and base $B$ is a structure $(F, E, p, B)$, where $p : E \to B$ is a surjective map with the following property: for any point $y : B$ there exists a neighborhood $U_b$ for which there is a homeomorphism

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

making the following diagram commute:

$$\begin{array}{ccc} p^{-1}(U_b) & \xrightarrow{f} & U_b \times F \\ {\scriptstyle p}\downarrow & \swarrow_{pr_1} & \\ U_b & & \end{array}$$

Definition 1.5.18 (Trivial Fiber Bundle). When the total space $E$ is the cartesian product $\Sigma(B, F)$ and $p = pr_1$, then such a bundle is called trivial: $(F, \Sigma(B, F), pr_1, B)$.

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

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

Theorem 1.5.14 (Fiber Bundle ≡ Π). 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 1.5.19. (Fibration-1) Dependent fiber bundle derived from Ξ contractability.

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

Definition 1.5.20. (Fibration-2). Dependent fiber bundle derived from surjective function.

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

Definition 1.5.21. (Fibration-3). Non-dependent fiber bundle derived from fiber truncation.

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

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

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

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

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

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

Definition 1.5.22. (Fibration-4). Non-dependen fiber bundle derived as pullback square.

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

### 1.5.6 Equivalence

Definition 1.5.23. (Fiberwise Equivalence). Fiberwise equivalence $\simeq$ or Equiv of function $f : A \to B$ represents internal equality of types $A$ and $B$ in the universe $U$ as contractible fibers of $f$ over base $B$.

$$A \simeq B : U \ =_{def} \mathbf{Equiv}(A, B) : U =_{def}$$
$$\sum_{f:A\to B} \prod_{y:B} \sum_{x:\Sigma_{x:A} y=_B f(x)} \sum_{w:\Sigma_{x:A} y=_B f(x)} x =_{\Sigma_{x:A} y=_B f(x)} w.$$

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

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

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

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

Definition 1.5.24. (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 1.5.15. (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 1.5.16. (Fiberwise Computation of Induction Principle).

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

Definition 1.5.25. (Surjective).

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

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

Definition 1.5.26. (Injective).

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

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

Definition 1.5.27. (Embedding).

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

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

Definition 1.5.28. (Half-adjoint Equivalence).

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

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

### 1.5.7 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 1.5.29. (Interval). Compact interval.

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

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

Definition 1.5.30. (Interval Split). The convertion function from I to a type of comparison is a direct eliminator of interval. The interval is also known as one of primitive higher inductive types which will be given in the next Issue IV: Higher Inductive Types.

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

Definition 1.5.31. (Homotopy). The homotopy between two function $f, g : X \to Y$ is a continuous map of cylinder $H : X \times \mathbf{I} \to Y$ such that

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

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

Definition 1.5.32. (funExt-Formation)

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

Definition 1.5.33. (funExt-Introduction)

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

Definition 1.5.34. (funExt-Elimination)

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

Definition 1.5.35. (funExt-Computation)

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

Definition 1.5.36. (funExt-Uniqueness)

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

### 1.5.8 Isomorphism

Definition 1.5.37. (iso-Formation)

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

Definition 1.5.38. (iso-Introduction)

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

Definition 1.5.39. (iso-Elimination)

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

Definition 1.5.40. (iso-Computation)

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

Definition 1.5.41. (iso-Uniqueness)

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

### 1.5.9 Univalence

Definition 1.5.42. (uni-Formation)

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

Definition 1.5.43. (uni-Introduction)

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

Definition 1.5.44. (uni-Elimination)

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

Definition 1.5.45. (uni-Computation)

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

Definition 1.5.46. (uni-Uniqueness)

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

### 1.5.10 Loop Spaces

Definition 1.5.47. (Pointed Space). A pointed type $(A, a)$ is a type $A : U$ together with a point $a : A$, called its basepoint.

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

Definition 1.5.48. (Loop Space).

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

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

Definition 1.5.49. (n-Loop Space).

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

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

### 1.5.11 Homotopy Groups

Definition 1.5.50. (n-th Homotopy Group of m-Sphere).

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

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

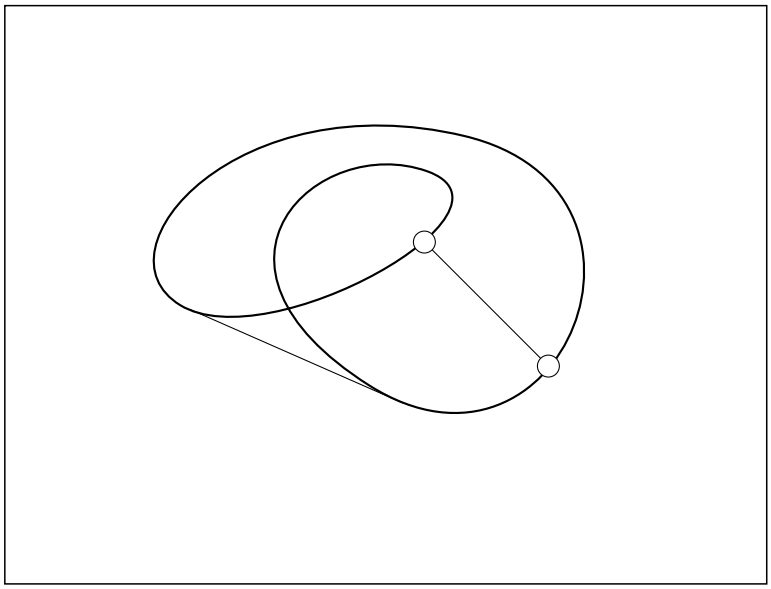
Theorem 1.5.17. ($\Omega(S^1) = \mathbb{Z}$).

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

loopS1 : U = Ξ S1 base base

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

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

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

### 1.5.12 Hopf Fibrations

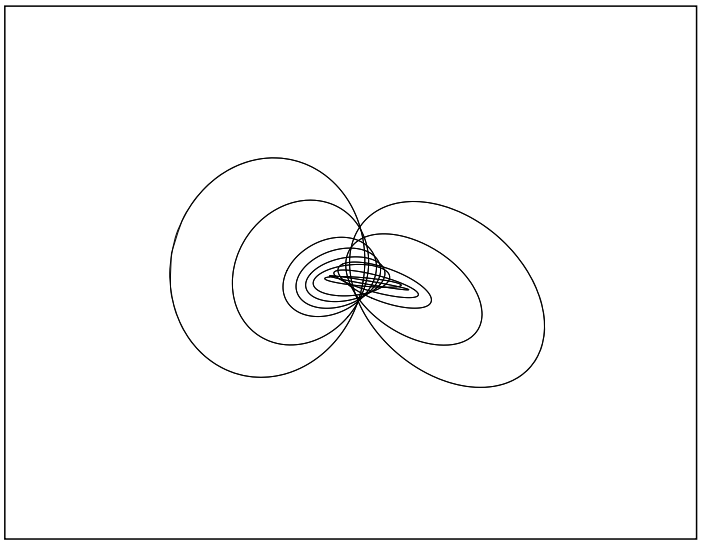
Example 1.5.2. ($S^1$ $\mathbb{R}$ Hopf Fiber).

```
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 1.5.3. ($S^3$ $\mathbb{C}$ 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 1.5.51. (H-space). H-space over a carrier $A$ is a tuple

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

.

Theorem 1.5.18. (Hopf Invariant). Let $\phi : S^{2n-1} \to S^n$ a continuous map. Then homotopy pushout (cofiber) of $\phi$ 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 1.5.19. (Four). There are fiber bundles: $(S^0, S^1, p, S^1)$, $(S^1, S^3, p, S^2)$, $(S^3, S^7, p, S^4)$, $(S^7, S^{15}, p, S^8)$.

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

Theorem 1.5.20. (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

# Зміст

## 1.6 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 1.6.1. (Cell Attachment). The attachment of an $n$-cell to a space $X$ along $f : S^{n-1} \to X$ is a pushout:

$$\begin{array}{ccc} S^{n-1} & \xrightarrow{f} & X \\ \downarrow{\scriptstyle \iota} & & \downarrow{\scriptstyle j} \\ D^n & \xrightarrow{g} & X \cup_f D^n \end{array}$$

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 1.6.2. (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:

$$\begin{array}{ccc} \coprod_{j \in J_n} S^{n-1} & \xrightarrow{\coprod f_j} & X_{n-1} \\ \downarrow{\scriptstyle \coprod \iota_j} & & \downarrow{\scriptstyle i_n} \\ \coprod_{j \in J_n} D^n & \xrightarrow{\coprod g_j} & X_n \end{array}$$

  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 $X = \operatorname{colim}_{n \to \infty} X_n$. The sequence is the skeletal filtration.

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

### 1.6.1 Motivation for 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.

### 1.6.2 HITs with Countable Constructors

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

```
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.

## 1.7 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.

### 1.7.1 Suspension

The suspension $\Sigma A$ 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 1.7.1. (Formation). For any type $A : \mathcal{U}$, there exists a suspension type $\Sigma A : \mathcal{U}$.

Definition 1.7.2. (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 1.7.1. (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 1.7.2. (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 1.7.3. (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 $a : A$.

### 1.7.2 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 1.7.3. (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 1.7.4. (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 1.7.4. (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 1.7.5. (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 1.7.6. (Uniqueness). Any two maps $u, v : \sqcup(A, B, C, f, g) \to D$ are homotopic if they agree on $\mathrm{po}_1$, $\mathrm{po}_2$, and $\mathrm{po}_3$, 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 1.7.1. (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 $f$.

$$\begin{array}{ccc} S^{n-1} & \xrightarrow{f} & X \\ \downarrow{\scriptstyle g} & & \downarrow \\ D^n & \longrightarrow & X \cup_f D^n \end{array}$$

### 1.7.3 Spheres

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

Definition 1.7.5. (Pointed n-Spheres) The $n$-sphere $S^n$ is defined recursively as a type in the universe $\mathcal{U}$ using general recursion over dimensions:

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

Definition 1.7.6. (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 $\Sigma$ to the base type $\mathbf{2}$ (0-sphere) $n$ times to obtain $S^n$.

Example 1.7.2. (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 $S^n$.

### 1.7.4 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 1.7.7. (Formation). For types $S, A : \mathcal{U}$, there exists a hub and spokes type $\odot\,(S, A) : \mathcal{U}$.

Definition 1.7.8. (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 1.7.7. (Elimination). For a family of types $P : \text{HubSpokes}\, S\, A \to \mathcal{U}$, maps $\text{pbase} : (x : A) \to P(\text{base}\, x)$, $\text{phub} : (f : S \to \text{HubSpokes}\, S\, A) \to P(\text{hub}\, f)$, and a family of paths $\text{pspoke} : (f : S \to \text{HubSpokes}\, S\, A) \to (s : S) \to \text{PathP}\,(< i > P(\text{spoke}\, f\, s\, @\, i))\,(\text{phub}\, f)\,(P(f\, s))$, there exists a map $\text{hubSpokesInd} : (z : \text{HubSpokes}\, S\, A) \to P(z)$, such that:

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

### 1.7.5 Truncation

#### Set Truncation

Definition 1.7.9. (Formation). Set truncation (0-truncation), denoted $\|A\|_0$, ensures that the type is a set, with homotopy groups vanishing above dimension 0.

Definition 1.7.10. (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 1.7.8. (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 1.7.11. (Formation). Groupoid truncation (1-truncation), denoted $\|A\|_1$, ensures that the type is a 1-groupoid, with homotopy groups vanishing above dimension 1.

Definition 1.7.12. (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 1.7.9. (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 $\text{Ind}_{\|A\|_1}(\text{inc}(a)) = f(a)$.

### 1.7.6 Quotient Spaces

#### Set Quotient Spaces

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

Definition 1.7.13. (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 1.7.14. (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 1.7.10. (Elimination). For a family of types $B : A/R \to \mathcal{U}$ with $\text{isSet}(Bx)$, and maps $f : (x : A) \to B(\text{quot}(x))$, $g : (a, b : A) \to (r : R(a, b)) \to \text{PathP}(< i > B(\text{ident}(a, b, r) \mathbin{@} i))(f(a))(f(b))$, 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 1.7.15. (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 1.7.16. (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.

### 1.7.7 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 1.7.17. (Formation). For pointed types $A, B$ : pointed, the wedge $A \vee B : \mathcal{U}$.

Definition 1.7.18. (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 1.7.11. (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 1.7.12. (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 1.7.13. (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 $h_1(\text{wglue}) = h_2(\text{wglue})$.

### 1.7.8 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 1.7.19. (Formation). For pointed types $A, B$ : pointed, the smash product $A \wedge B : \mathcal{U}$.

Definition 1.7.20. (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 1.7.14. (Elimination). For a family of types $P$ : Smash $A\,B \to \mathcal{U}$, points pbasel : $P$(basel), pbaser : $P$(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 1.7.15. (Computation). For a family of types $P : A \wedge B \to \mathcal{U}$, points pbasel : $P$(basel), pbaser : $P$(baser), map pproj : $(x : A.1) \to (y : B.1) \to P(\text{proj}\,x\,y)$, and families of paths pgluel : $(a : A.1) \to \text{PathP}\,(< i > P(\text{gluel}\,a\,@\,i))\,(\text{pproj}\,a\,B.2)\,\text{pbasel}$, 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 1.7.16. (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 $(z : A \wedge B) \to h_1(z) \equiv h_2(z)$.

### 1.7.9 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 1.7.21. (Formation). For types $A, B : \mathcal{U}$, the join $A * B : \mathcal{U}$.

Definition 1.7.22. (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 1.7.17. (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 1.7.18. (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 1.7.19. (Uniqueness). Any two maps $h_1, h_2 : A \vee B \to C$ are homotopic if they agree on joinl, joinr, and join.

### 1.7.10 Colimit

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

Definition 1.7.23. (Colimit) For a sequence of types $A : \mathrm{nat} \to \mathcal{U}$ and maps $f : (n : \mathbb{N}) \to An \to A(\mathrm{succ}(n))$, the colimit type $\mathrm{colimit}(A, f) : \mathcal{U}$.

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

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

### 1.7.11 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 1.7.24. (Formation). For types $A, B : \mathcal{U}$ and maps $f, g : A \to B$, the coequalizer coeq $ABfg : \mathcal{U}$.

Definition 1.7.25. (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 1.7.21. (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 1.7.22. (Computation). For $z :$ coeq $ABfg$,

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

Theorem 1.7.23. (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 1.7.3. (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 1.7.26. (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 1.7.27. (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 1.7.24. (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 1.7.25. (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 1.7.26. (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 $a : A$.

### 1.7.12 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 1.7.28. (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 1.7.27. (Elimination $KGn$) For $n \geq 1$, a type $B : \mathcal{U}$ with isNGroupoid$(B, \text{succ } n)$, and a map $f$ : suspension$(K1'G) \to B$, there exists $\text{rec}_{KGn} : KGnG(\text{succ } n) \to B$, defined via nTruncRec.

### 1.7.13 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 1.7.29. (Localization Modality) For a family of maps $F_A : S(a) \to T(a)$, the $F$-localization $L_F^{AST}(X) : \mathcal{U}$.

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

```
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 1.7.28. (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 $i \cdot \text{center}_X = n$.

## 1.8 Conclusion

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

# Бібліографія


[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.