Issue I: Martin-Löf Type Theory

Maksym Sokhatskyi

National Technical University of Ukraine
Igor Sikorsky Kyiv Polytechnical Institute

June 1, 2025

Abstract

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

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

Contents

Introduction to MLTT

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

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

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

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

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

Syntax of Per

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

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

Here, = (definition), ∅ (empty set), | (vertical bar) — are parts of BNF language and ⟨, ⟩, (,), :=, ∨, ∧, -, →, 0, 1, @, □, **module**, **import**, **where**, **transp**, **.1**, **.2**, and , are terminals of the type checker language [1].

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

1 Interpretations

1.1 Type Theory

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

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

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

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

1.2 Logic

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

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

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

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

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

1.3 Category Theory

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

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

1.4 Homotopy Theory

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

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

Set Theory

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

2 Dependent Type Theory

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

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

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

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

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

Definition 2 (Universe Introduction)**.** A type A belongs to a universe U_i if it can be derived as a type at level i. For MLTT-73, this includes base types (e.g., Π, Σ, Path), user-defined types, and universes U_j for $j < i$. The introduction rule is: $\Gamma; A \vdash A : U_i$, where i is the minimal level such that $A \in U_i$. Types like $\Pi(A, B)$, $\Sigma(A, B)$, and $\Xi(A, x, y)$ are explicitly landed in a universe:

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

Definition 3 (Cumulative Hierarchy)**.** The universe hierarchy is cumulative, meaning if $A : U_i$, then $A : U_j$ for all $j > i$. This ensures flexibility in type checking, as types can be lifted to higher universes. This is implicit in the type checker's ability to assign types to higher universes when needed.

Definition 4 (Predicative Rules)**.** The formation of dependent types (e.g., Π, Σ) lands in the maximum of the universe levels of its constituents. For example, for Π-types: $\Gamma \vdash A : U_i$ and $\Gamma, x : A \vdash B(x) : U_j$ we can derive $\Gamma \vdash \Pi(x : A), B(x) : U_{\max(i,j)}$ This predicative rule ensures that the universe level reflects the highest level of the domain or codomain.

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

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

Definition 5 (Definitional Equality)**.** Universes support definitional equality, where two types $A, B : U_i$ are equal if their normalized forms are identical. This is crucial for type checking in MLTT-73.

2.2 Dependent Product (Π)

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

At this light there could be many interpretation of Π types from different areas of mathematics. We give here three: i) logical interpretation of Π as $\forall$ 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 6 (Π-Formation)**.** Π-types represents the way we create the spaces of dependent functions $f : \Pi(x : A), B(x)$ with domain in A and codomain in type family $B : A \to U$ over A.

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

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

Definition 7 (Π-Introduction)**.** Lambda constructor defines a new lambda function in the space of dependent functions. It is called lambda abstraction and displayed as $\lambda x.b(x)$ or $x \mapsto b(x)$.

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

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

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

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

Definition 8 (Π-Induction Principle)**.** States that if predicate holds for lambda function then there is a function from function space to the space of predicate.

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

Definition 9 (Π-Elimination)**.** Application reduces the term by using recursive substitution.

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

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

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

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

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

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

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

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

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

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

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

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

Categorical interpretation

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

Definition 10 (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 11 (Space of Sections)**.** Let $\mathbf{H}$ be a $(\infty, 1)$-topos, and let $E \to B : \mathbf{H}_{/B}$ a bundle in $\mathbf{H}$, object in the slice topos. Then the space of sections $\Gamma_\Sigma(E)$ of this bundle is the Dependent Product:

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

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

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

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

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

Definition 12 (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 13. (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 14. (Fiber Bundle). The fiber bundle $F \to E \xrightarrow{p} B$ on a total space E with fiber layer F and base B is a structure (F, E, p, B) where $p : E \to B$ is a surjective map with following property: for any point $y : B$ exists a neighborhood U_b for which a homeomorphism $f : p^{-1}(U_b) \to U_b \times F$ making the following diagram commute.

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

Definition 15. (Cartesian Product of Family over B). Is a set F of sections of the bundle with elimination map $app : F \times B \to E$ such that

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

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

Definition 16 (Trivial Fiber Bundle)**.** When total space E is cartesian product $\Sigma(B, F)$ and $p = pr_1$ then such bundle is called trivial $(F, \Sigma(B, F), pr_1, B)$.

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

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

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

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

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

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

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

def Bundle=Pi (B : U) (F : B → U) (y : B)
```

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

2.3 Dependent Sum (Σ)

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

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

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

Type-theoretical interpretation

Definition 17 (Σ-Formation)**.** The dependent sum type is indexed over type A in the sense of coproduct or disjoint union, where only one fiber codomain $B(x)$ is present in pair.

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

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

Definition 18 (Σ-Introduction)**.** The dependent pair constructor is a way to create indexed pair over type A in the sense of coproduct or disjoint union.

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

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

Definition 19 (Σ-Elimination)**.** The dependent projections $pr_1 : \Sigma(A,B) \rightarrow A$ and $pr_2 : \Pi_{x:\Sigma(A,B)} B(pr_1(x))$ are pair deconstructors.

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

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

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

Definition 20 (Σ-Induction)**.** States that if predicate holds for two projections then predicate holds for total space.

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

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

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

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

Categorical interpretation

Definition 21. (Dependent Sum). The dependent sum along the morphism $f : A \to B$ in category C is the left adjoint $\Sigma_f : C_{/A} \to C_{/B}$ of the base change functor.

Set-theoretical interpretation

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

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

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

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

2.4 Path Space (Ξ)

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

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

Chosing flavour of normal forms for identity system

Here we give brief description of structure inside path spaces:

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

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

Heyting Algebra: A Heyting Algebra in HoTT is a bounded distributive lattice $(L, \wedge, \vee, 0, 1) : \mathcal{U}$ equipped with an implication operation $\to : L \to L \to L$ such that, for all $a, b, c : L$, there is an equivalence $a \le b \to c \iff a \wedge b \le c$, where $\le$ is the partial order defined by $a \le b \iff a \wedge b \equiv a$. Negation is defined as $\neg a \equiv a \to 0$, and modus ponens holds: given $a : A$ and $f : A \to B$, there exists $f\,a : B$. In a Boolean topos, the Heyting algebra becomes a Boolean algebra, with $\to$ corresponding to the exponential B^A.

Boolean Algebra: A Boolean Algebra in HoTT is a De Morgan Algebra $(L, \wedge, \vee, \neg, 0, 1) : \mathcal{U}$ satisfying the law of excluded middle ($a \vee \neg a \equiv 1$) and non-contradiction ($a \wedge \neg a \equiv 0$). The type $L \cong 2 = \{\text{true}, \text{false}\}$ models classical

[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`

propositions, with a mandatory Boolean type in a Boolean topos, where L is the subobject classifier $\Omega \cong 2$, and all operations correspond to classical logical connectives.

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

Type-theoretical interpretation

Definition 22 (Path Formation)**.**

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

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

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

Definition 23 (Path Introduction)**.**

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

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

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

Definition 24 (Path Application)**.** You can apply face to path.

```
def at0 (A: U) (a b: A) (p: Path A a b) : A := p @ 0
def at1 (A: U) (a b: A) (p: Path A a b) : A := p @ 1
```

Definition 25 (Path Composition)**.** Composition operation allows to build a new path by given to paths in a connected point.

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

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

Theorem 12 (Path Inversion)**.**

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

Definition 26 (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) $\lambda\ (i,j : I) \to p\ @\ max(i,j)$.

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

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

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

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

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

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

```
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)
```

Theorem 14 (Generalized Transport Kan Operation)**.** Transports a value of the left type to the value of the right type by a given path element of the path space between left and right types.

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

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

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

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

Definition 27 (Singleton). `def singl (A: U) (a: A): U := Σ (x: A), Ξ A a x`

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

Theorem 16 (Singleton Contractability).
```
def contr (A : U) (a b : A) (p : Ξ A a b)
   : Ξ (singl A a) (eta A a) (b, p)
 := <i> (p @ i, <j> p @ i /\ j)
```

Theorem 17 (Path Elimination).
```
def subst (A : U) (P : A -> U) (a b : A)
    (p : Ξ A a b) (e : P a) : P b
 := transp (<i> P (p @ i)) 0 e

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

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

Theorem 18. (Path Computation).

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

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

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

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

Groupoid interpretation

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

2.5 Natural Numbers (N)

The natural numbers, denoted $\mathbf{N}$, introduced in MLTT-75, form a fundamental type in mathematics, representing the non-negative integers (including zero) with operations for construction and reasoning. This section defines the type $\mathbf{N}$, its constructors (zero and successor), and its induction principle, along with the β- and η-rules for computation and uniqueness.

Type-theoretical interpretation

The natural numbers are defined as a type with two constructors: zero for the number 0 and succ for the successor function, which generates the next natural number. The induction principle, $\mathrm{Ind}_{\mathbf{N}}$, provides a method to reason about all natural numbers. The β- and η-rules govern the computational behavior and uniqueness of functions defined over $\mathbf{N}$.

Definition 28 ($\mathbf{N}$-Formation)**.** The type of natural numbers $\mathbf{N}$ is a type in the universe U, representing the non-negative integers.

$$\mathbf{N} : U =_{\mathrm{def}} \mathbb{N}.$$

```
def N : U := ℕ
```

Definition 29 ($\mathbf{N}$-Introduction)**.** The natural numbers are constructed using two constructors: 1) $\mathrm{zero} : \mathbf{N}$, representing the number 0; 2) $\mathrm{succ} : \mathbf{N} \to \mathbf{N}$, the successor function mapping a natural number n to $n+1$.

$$\mathrm{zero} : \mathbf{N}, \quad \mathrm{succ} : \mathbf{N} \to \mathbf{N}.$$

```
def zero : N := 0
def succ (n : N) : N := n + 1
```

Definition 30 ($\mathbf{N}$-Induction Principle)**.** The induction principle for natural numbers, $\mathrm{Ind}_{\mathbf{N}}$, states that to prove a property $C : \mathbf{N} \to U$ holds for all $n : \mathbf{N}$, it suffices to provide:

- A proof $c_0 : C(\mathrm{zero})$ for the base case.
- A function $c_s : \prod_{n:\mathbf{N}} C(n) \to C(\mathrm{succ}(n))$ for the inductive step.

Then, there exists a function that assigns to each $n : \mathbf{N}$ a proof of $C(n)$.

$$\mathrm{Ind}_{\mathbf{N}} : \prod_{C:\mathbf{N}\to U} C(\mathrm{zero}) \to \left(\prod_{n:\mathbf{N}} C(n) \to C(\mathrm{succ}(n))\right) \to \prod_{n:\mathbf{N}} C(n).$$

Definition 31 ($\mathbf{N}$-Elimination)**.** The elimination rule for $\mathbf{N}$ is given by applying the induction principle to compute over natural numbers. For a natural number

$n : \mathbf{N}$, a type family $C : \mathbf{N} \to U$, a base case $c_0 : C(\text{zero})$, and an inductive step $c_s : \prod_{n:\mathbf{N}} C(n) \to C(\text{succ}(n))$, the eliminator computes:

$$\text{ind}_{\mathbf{N}}(C, c_0, c_s, n) : C(n).$$

Specifically:

- For $n = \text{zero}$, $\text{ind}_{\mathbf{N}}(C, c_0, c_s, \text{zero}) = c_0$.
- For $n = \text{succ}(m)$, $\text{ind}_{\mathbf{N}}(C, c_0, c_s, \text{succ}(m)) = c_s(m, \text{ind}_{\mathbf{N}}(C, c_0, c_s, m))$.

```
def ind_N (C : N → U) (c0 : C zero)
    (cs : Π (n : N), C n → C (succ n))
  : Π (n : N), C n
```

Theorem 19 (**N**-Computation (β-rules))**.** The β-rules for natural numbers specify the computational behavior of the induction principle:

- For the base case:

$$\text{ind}_{\mathbf{N}}(C, c_0, c_s, \text{zero}) =_{C(\text{zero})} c_0.$$

- For the inductive step:

$$\text{ind}_{\mathbf{N}}(C, c_0, c_s, \text{succ}(n)) =_{C(\text{succ}(n))} c_s(n, \text{ind}_{\mathbf{N}}(C, c_0, c_s, n)).$$

```
def N-β-zero (C : N → U) (c0 : C zero)
    (cs : Π (n : N), C n → C (succ n))
  : Path (C zero) (ind_N C c0 cs zero) c0 := idp (C zero) c0

def N-β-succ (C : N → U) (c0 : C zero)
    (cs : Π (n : N), C n → C (succ n)) (n : N)
  : Path (C (succ n)) (ind_N C c0 cs (succ n)) (cs n (ind_N C c0 cs n))
 := idp (C (succ n)) (cs n (ind_N C c0 cs n))
```

Theorem 20 (**N**-Uniqueness (η-rule))**.** The η-rule for natural numbers ensures the uniqueness of functions defined by induction. For a function $f : \prod_{n:\mathbf{N}} C(n)$ defined over $\mathbf{N}$, it is equal to the function defined by induction using the same base and step cases:

$$f =_{\prod_{n:\mathbf{N}} C(n)} \text{ind}_{\mathbf{N}}(C, f(\text{zero}), \lambda(n : \mathbf{N}), f(\text{succ}(n))).$$

```
def N-η (C : N → U) (f : Π (n : N), C n)
  : Ξ (Π(n : N), C n) f (ind_N C (f zero) (λ (n : N), f (succ n)))
  := idp (Π (n : N), C n) f
```

These definitions and theorems provide a formal framework for the natural numbers in type theory, capturing their structure, computational behavior, and uniqueness properties.

Contexts

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

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

Definition 32 (Empty Context)**.** The empty context contains no variable declarations and serves as the base case for context formation. It is represented as the unit type, indicating an empty telescope:

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

Definition 33 (Context Comprehension)**.** A context is extended by adding a variable declaration for a type dependent on the existing context. For a context Γ and a type A over Γ, the extended context is:

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

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

Definition 34 (Context Derivability)**.** A type A is derivable in a context Γ if it can be assigned to a universe given the variables in Γ:

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

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

Definition 35 (Terms)**.** A term is an element of a type within a context. Given $\Gamma \vdash A : U_i$, a term t satisfies $\Gamma \vdash t : A$. Terms include variables, constructors (e.g., λ for Π, pairs for Σ), and applications, defined by MLTT-73's syntax.

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

MLTT-73

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

Definition 36 (MLTT-73 Reality Check)**.** The MLTT as a Type is defined by taking all rules for Π, Σ and Path types into one Σ telescope or context.

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

Theorem 21. (Model Check). There is an instance of MLTT.

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

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

Conclusions

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

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

The significance of this work lies in its constructive approach to the J eliminator, a cornerstone of MLTT-73 identity type, which previous internalization

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

attempts failed to derive constructively [3, 10]. By leveraging cubical type theory's Path types and operations (e.g., connections, compositions), the type checker achieves a compact foundational core for verifying mathematics.

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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