Direct Embedding of Martin-Löf Type Theory into the Cubical Type Checkers

Maxim Sokhatsky

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

August 13, 2018

Abstract

Each language implementation needs to be checked. The one of possible test cases for cubical type checkers is the direct embedding of type theory model intro the language of type checker. As Martin-Löf Type Theory is formulated using 5 types of rules, we construct aliases for host language primitives and use type checker to prove theorems about itself.

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.

Keywords: Cubical Type Theory, Martin-Löf Type Theory

Contents

Intro

Each language implementation needs to be checked. The one of possible test cases for cubical type checkers is the direct embedding of type theory model intro the language of type checker. As Martin-Löf Type Theory is formulated using 5 types of rules, we construct aliases for host language primitives and use type checker to prove theorems about itself.

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.

This technique of direct embedding of the model into the type checker primitives was also used by authors to prove that Category of Sets is Cartesian Closed.

Cubical Syntax

The BNF notation of `cubicaltt` consists of 1) telescopes (contexts or sigma chains); 2) inductive data definitions (sum chains); 3) split eliminator; 4) branches of split eliminators; 5) pure dependent type theory syntax. It also has where, import, module constructions.

```
def := data id tele = sum + id tele : exp = exp +
       id tele : exp where def

exp := cotele*exp + cotele→exp + exp→exp + (exp) + app + id +
       (exp,exp) + \ cotele→exp + split cobrs + exp.1 + exp.2
```

```
    0 := #empty           imp    := [ import id ]
  brs := 0 + cobrs        tele   := 0 + cotele
  app := exp exp          cotele := ( exp : exp ) tele
   id := [ #nat ]         sum    := 0 + id tele + id tele | sum
  ids := [ id ]           br     := ids → exp
codec := def dec          mod    := module id where imp dec
  dec := 0 + codec        cobrs  := | br brs
```

Note that the syntax lacks HITs as for this article we don't need ones.

1 Martin-Löf Type Theory

Martin-Löf Type Theory (MLTT) contains Π, Σ, Id, W, Nat, List types. Id types were added in 1984 while original MLTT was introduced in 1972. Predicative Universe Hierarchy was added in 1975. Despite original MLTT contains Id types that preserve uniquness of identity proofs (UIP), we introduce here homotopical univalent heterogeneous Path interval types with higher equalities (∞-Groupoid interpretation). Thus, allowing to proof all the MLTT rules as a whole by using reflection rule, connections and path applications.

1.1 Pi

Π is a dependent function type, the generalization of functions. As a function it can serve the wide range of mathematical constructions, objects, types, or spaces: sets, functions, polynomial functors, infinitesimals, ∞-groupoids, topological ∞-groupoid, CW-complexes, categories, languages, etc. We give here nearest isomorphism of Π-types, the fibrations or fiber bundles. The next isomorphism of functions are functors.

Definition 1. (Π-Formation).

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

Definition 2. (Π-Introduction).

```
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 3. (Π-Elimination).

```
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. (Π-Computation).

```
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 2. (Π-Uniqueness).

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

Examples from Mathematics

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.

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 4. (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.

Definition 5. (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 6. (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.

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

Definition 7. (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 moriphisms 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 8. (Trivial Fiber Bundle). When total space E is cartesian product $\Sigma(B, F)$ and $p = pr_1$ then such bundle is called trivial $(F, \Sigma(B, F), pr_1, B)$.

Definition 9. (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 10. (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 3. (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 4. (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 5. (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
```

Theorem 6. (HomSet). If codomain is set then space of sections is a set.

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

Theorem 7. (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)
```

1.2 Sigma

Σ is a dependent product 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.

Definition 11. (Σ-Formation).

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

Definition 12. (Σ-Introduction).

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

Definition 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 8. (Σ-Computation).

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

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

Theorem 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)
   = refl (Sigma A B) p
```

Examples from Mathematics

Definition 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.

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 $R(x, f(x))$.

```
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 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 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 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)
```

1.3 Path

The Path identity type 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 [1]CCHM cubicaltt model with connections. For [2]ABCFHL yacctt model with variables please refer to ytt file. You may also want to read [3]BCH, [4]AFH. There is a [5]PO paper about CCHM axiomatic in a topos.

Definition 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
```

Definition 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 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 18. (Path Composition). Composition operation allows to build a new path by given to paths in a connected point.

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

```
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 []
```

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

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

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

[4]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

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

Theorem 14. (Path Inversion).

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

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

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

```
connection1 (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)

connection2 (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)
```

Theorem 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 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
```

Definition 20. (Singleton).

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

Theorem 17. (Singleton Instance).

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

Theorem 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 19. (Path Elimination, Diagonal).

```
D (A: U) : U = (x y: A) -> Path A x y -> U
J (A: U) (x y: A) (C: D A)
  (d: C x x (refl A x))
  (p: Path A x y) : C x y p
= subst (singl A x) T (eta A x) (y, p) (contr A x y p) d where
  T (z: singl A x) : U = C x (z.1) (z.2)
```

Theorem 20. (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 21. (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 22. (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.

1.4 MLTT

Definition 21. (MLTT). The MLTT as a Type is defined by taking all rules for Π, Σ and Path type into one Σ-chain.

```
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 23. (Sound 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)
```

MLTT 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.

```
$ time cubical -b mltt.ctt
Checking: MLTT
Checking: instance
File loaded.

real    0m6.308s
user    0m6.278s
sys     0m0.014s
```