Introductory notes on Martin-Löf's Type Theory

Nikos Rigas*

Version 2024-11-20

1 Introduction

Martin-Löf's type theory (MLTT) may be viewed as any of the following:

- A foundation and formalization of constructive mathematics, especially those developed by E. Bishop.
- A theory of meaning of (constructive) mathematics.
- An archetypical functional programming language.
- (After Voevodsky) A synthetic language for homotopy theory and a foundation of constructive-structural mathematics.

2 Judgmentals of MLTT

We will present MLTT as a theory of definitions. This is akin to a programming language that is initially "empty", and any types, functions, and similar need to be defined from scratch before they can be used. Such a programming language would involve concrete choices regarding the form of definitions and provide the syntactical means necessary for stating such definitions. In MLTT, this rôle is played by the *judgmental layer* of the language.

Judgmental equality

A previously undefined expression e_1 may be assigned the same meaning as an already defined expression e_2; this act is denoted by

$$e_1 \quad :\equiv \quad e_2.$$

We further write $e_1 \equiv e_2$ if expressions e_1 and e_2 may be syntactically identified by expanding the definitions of sub-expressions. If this is the case, we say that e_1 and e_2 are *judgmentally equal* or *definitionally equal* or *synonymous*.

Elementhood

Types have elements (more on that later). We write $a : A$ to express the judgment that a is an element of type A.

*Email: nicolasr@di.uoa.gr

Families

Indexed families are denoted by prepending the indices parenthesized. For example,

$$(x : A)\ t(x) : B$$

denotes a family of elements of type B indexed by type A. We often refer to such a family by saying "for (arbitrary) $x : A$, $t(x) : B$". Similarly,

$$(x_1 : A_1, \ldots, x_n : A_n)\ t(x) : A$$

denotes a family with n indices. (A family with zero indices is a single element of A.) For example, for any type A there is the identity family $(x : A)\ x$, and any element a of A gives rise to constant families over any and all indices.

The indices of a family may themselves be families. This is reflected in expressions like

$$(x : A, (y : B)\ z(y) : C)\ t(x, z),$$

which signifies a family indexed by elements of A and families of elements of C indexed by elements of B.

3 Inductive types

The primary means of introducing new types into the system is *inductive definition*. For example, the natural numbers are generated by the induction scheme

- zero is a natural number, and
- the successor of a natural number is a natural number.

In type-theoretic notation, the above inductive description takes the form

- $0 : \mathrm{Nat}$, and
- $(n : \mathrm{Nat})\ \mathsf{s}(n) : \mathrm{Nat}$.

0 and s are the postulated ways of constructing natural numbers; they are the *constructors* of Nat. 0 has zero indices and constructs a single element of Nat, whereas s is a family of natural numbers with one index which is itself a natural number (hence, it is a recursive constructor).

An inductive definition of a type amounts to listing its constructors, i.e., certain families of elements of the type being defined that signify the (canonical) ways of constructing elements of that type. More general forms (e.g., mutual inductive definitions, inductive-inductive definitions, inductive-recursive definitions) are possible; we will introduce them only where necessary.

Other common examples of inductively defined types are the type $\mathrm{List}(A)$ of lists of elements of a type A, with constructors

- $\mathsf{nil}_A : \mathrm{List}(A)$,
- $(\mathit{head} : A, \mathit{tail} : \mathrm{List}(A))\ \mathsf{cons}_A(\mathit{head}, \mathit{tail}) : \mathrm{List}(A)$,

and the type Bool, with constructors

- $\mathsf{false} : \mathrm{Bool}$,
- $\mathsf{true} : \mathrm{Bool}$.

4 Recursion

There are two primary uses of the inductive description of a type: Defining functions by recursion, and proving properties of its elements by induction. We will treat recursion now, and defer induction for when we have type families at our disposal. In the case of Nat this is the familiar definition by primitive recursion: Given a type C, an element c_0 of C and a family $(x : \mathrm{Nat}, y : C)\, c_{\mathsf{s}}(x, y)$ of elements of C, the assignments

$$\begin{aligned} t(0) &:\equiv c_0, \\ t(\mathsf{s}(n)) &:\equiv c_{\mathsf{s}}(n, t(n)), \end{aligned}$$

define $t(x) : C$ for arbitrary $x : \mathrm{Nat}$. For example, we may define addition of natural numbers by recursion in the second operand:

$$\begin{aligned} m + 0 &:\equiv m, \\ m + \mathsf{s}(n) &:\equiv \mathsf{s}(m + n). \end{aligned}$$

In other words, addition (to m) is defined by the instance of Nat-recursion where $c_0 \equiv m$ and $c_{\mathsf{s}}(x, y) \equiv \mathsf{s}(y)$. Any definition by recursion over the natural numbers is determined by these two parameters, c_0 and c_{s}. By turning these into indices of the family being defined, we arrive at the *recursor*

$$(z : C, (x : \mathrm{Nat}, y : C)\, w(x, y) : C, n : \mathrm{Nat})\, \mathsf{rec}^C_{\mathrm{Nat}}(z, w, n) : C$$

of Nat, defined by the recursion

$$\begin{aligned} \mathsf{rec}^C_{\mathrm{Nat}}(z, w, 0) &:\equiv z, \\ \mathsf{rec}^C_{\mathrm{Nat}}(z, w, \mathsf{s}(n)) &:\equiv w(n, \mathsf{rec}^C_{\mathrm{Nat}}(z, w, n)). \end{aligned}$$

The superscript C is omitted when uninteresting or implied by the context.

Any recursive family may be explicitly defined with the help of the recursor; for example, addition would be defined by

$$m + n :\equiv \mathsf{rec}_{\mathrm{Nat}}(m, (x : \mathrm{Nat}, y : \mathrm{Nat})\, \mathsf{s}(y), n).$$

The recursion principle of any inductive type follows the same pattern; namely, in order to define a family over an inductive type it suffices to specify its instances on the constructors. For example, the recursion principle of Bool asserts that given two elements c_{false} and c_{true} of a type C, the assignments

$$\begin{aligned} t(\mathsf{false}) &:\equiv c_{\mathsf{false}}, \\ t(\mathsf{true}) &:\equiv c_{\mathsf{true}}, \end{aligned}$$

define a family $(x : \mathrm{Bool})\, t(x) : C$.

5 Logic

Our next task will be to add/inject logic into MLTT. The following definitions will reconstruct intuitionistic first-order logic, by means of the *propositions-as-types* paradigm. The general idea of propositions-as-types is to identify each proposition with the type whose elements are the possible pieces of evidence for that proposition; then, a proof of A from assumptions $A_1, \ldots, A_n$ yields evidence for A conditional on evidence for $A_1, \ldots, A_n$, i.e., it is a family of elements of A indexed by $A_1, \ldots, A_n$.

5.1 Product

The type-theoretic analogue of the conjunction of two propositions is the *product* $A_1 \times A_2$ of two types A_1 and A_2, defined by the ordered pair constructor:

- For $x_1 : A_1$ and $x_2 : A_2$, $\mathsf{pair}(x_1, x_2) : A_1 \times A_2$.

By erasing the elements from the formation rule

$$\frac{x_1 : A_1 \qquad x_2 : A_2}{\mathsf{pair}(x_1, x_2) : A_1 \times A_2}$$

of pair and switching to logical notation, we obtain the introduction rule

$$\frac{\phi_1 \qquad \phi_2}{\phi_1 \wedge \phi_2}\ (\wedge\text{-intro})$$

of conjunction.

Recursion principle: Given a family $(x_1 : A_1, x_2 : A_2)\, c_{\mathsf{pair}}(x_1, x_2)$ of elements of a type C indexed by A_1 and A_2, the assignment

$$t(\mathsf{pair}(x_1, x_2)) \quad :\equiv \quad c_{\mathsf{pair}}(x_1, x_2)$$

defines $t(x) : C$ for any $x : A_1 \times A_2$. Recursion over $A_1 \times A_2$ may be expressed by means of the recursor

$$\frac{(x_1 : A_1, x_2 : A_2)\, z(x_1, x_2) : C \qquad x : A_1 \times A_2}{\mathsf{rec}_{A_1 \times A_2}(z, x) : C}$$

of $A_1 \times A_2$, defined by the recursion

$$\mathsf{rec}_{A_1 \times A_2}(z, \mathsf{pair}(x_1, x_2)) \quad :\equiv \quad z(x_1, x_2).$$

Omitting the elements yields the elimination rule

$$\frac{\begin{array}{c}(\phi_1, \phi_2) \\ \vdots \\ \theta\end{array} \qquad \phi_1 \wedge \phi_2}{\theta}\ (\wedge\text{-elim})$$

of conjunction.

5.2 Function type

The *type* $A \to B$ *of functions* from A to B corresponds to logical implication; it is defined by the functional abstraction constructor:

- For a family $(x : A)\, b(x)$ of elements of B indexed by A, $\lambda(x : A)\, b(x) : A \to B$.

(Technically, we should be writing $\lambda((x : A)\, b(x))$, since the argument to λ is the entire family $(x : A)\, b(x)$, but the tradition is to omit parentheses here.) By omitting the elements and switching to logical notation, this yields the introduction rule

$$\frac{\begin{array}{c}(\phi) \\ \vdots \\ \psi\end{array}}{\phi \supset \psi}\ (\supset\text{-intro})$$

of implication.

Recursion principle: Given a family $((x:A)\, y(x):B)\, c_\lambda(y):C$, the assignment

$$t(\lambda(x:A)\, b(x)) \quad :\equiv \quad c_\lambda((x:A)\, b(x))$$

defines $t(f)$ for arbitrary $f:A\to B$. The recursor of $A\to B$ has the formation rule

$$\frac{((x:A)\, y(x):B)\, z(y):C \qquad f:A\to B}{\mathsf{rec}_{A\to B}(z,f):C}$$

and is defined by the recursion

$$\mathsf{rec}_{A\to B}(z,\lambda(x:A)\, b(x)) \quad :\equiv \quad z((x:A)\, b(x)).$$

If we erase the elements, this becomes the elimination rule

$$\frac{\begin{array}{c}\left(\dfrac{\phi}{\psi}\right)\\ \vdots\\ \theta\end{array} \qquad \phi\supset\psi}{\theta}\ (\supset\text{-elim})$$

for implication.

5.3 Sum

Disjunction is modelled by the *sum* A_1+A_2 of types A_1 and A_2, defined by the two constructors

- For $x_1:A_1$, $\mathsf{in}_1(x_1):A_1+A_2$.
- For $x_2:A_2$, $\mathsf{in}_2(x_2):A_1+A_2$.

These constructors correspond to the introduction rules

$$\frac{\phi_1}{\phi_1\vee\phi_2}\ (\vee\text{-intro1}) \qquad\qquad \frac{\phi_2}{\phi_1\vee\phi_2}\ (\vee\text{-intro2})$$

for disjunction.

Recursion principle: Given families $(x_1:A_1)\, c_{\mathsf{in}_1}(x_1):C$ and $(x_2:A_2)\, c_{\mathsf{in}_2}(x_2):C$, the assignments

$$\begin{aligned} t(\mathsf{in}_1(x_1)) &:\equiv c_{\mathsf{in}_1}(x_1),\\ t(\mathsf{in}_2(x_2)) &:\equiv c_{\mathsf{in}_2}(x_2),\end{aligned}$$

define $t(x):C$ for any $x:A_1+A_2$. The recursor has the form

$$\frac{(x_1:A_1)\, z_1(x_1):C \qquad (x_2:A_2)\, z_2(x_2):C \qquad x:A_1+A_2}{\mathsf{rec}_{A_1+A_2}(z_1,z_2,x):C}\ ,$$

is defined by the recursion

$$\begin{aligned} \mathsf{rec}_{A_1+A_2}(z_1,z_2,\mathsf{in}_1(x_1)) &:\equiv z_1(x_1),\\ \mathsf{rec}_{A_1+A_2}(z_1,z_2,\mathsf{in}_2(x_2)) &:\equiv z_2(x_2),\end{aligned}$$

and yields the elimination rule

$$\frac{\begin{array}{cc}(\phi_1) & (\phi_2)\\ \vdots & \vdots\\ \theta & \theta\end{array} \qquad \phi_1 \vee \phi_2}{\theta}\ (\vee\text{-elim})$$

(proof by cases) for disjunction.

5.4 $\mathbb{0}$

The type $\mathbb{0}$ corresponds to falsum ($\bot$); it has no constructors. Hence, its recursion principle stipulates the existence of a family $(x:\mathbb{0})\,t(x):C$ for any type C. Its recursor has the form

$$\frac{x:\mathbb{0}}{\mathsf{rec}_{\mathbb{0}}(x):C}\ ,$$

has no defining assignments (because there is nothing it can be defined on), and corresponds to the elimination rule

$$\frac{\bot}{\theta}\ (\bot\text{-elim})$$

(*ex falso*) of $\bot$.

The *negation* of a type A is defined to be the type $\neg A :\equiv A \to \mathbb{0}$.

5.5 Dependent sum

The product may be generalized by allowing the second operand to depend on the first: Given a type family $(x:A)\,B(x)$, the *dependent sum* $\sum(x:A)\,B(x)$ is defined by the constructor

- For $x:A$ and $y:B(x)$, $\mathsf{pair}(x,y):\sum(x:A)\,B(x)$.

(Note that the order now becomes important: We may not declare $y:B(x)$ before we have declared $x:A$.) We use the same name for the constructors of the product and the dependent sum to reinforce the fact that the latter is a generalization of the former. By erasing elements we obtain the introduction rule

$$\frac{\phi(a)}{\exists(x:A)\,\phi(x)}\ (\exists\text{-intro})$$

of the existential quantifier.

Recursion principle: Given an element $c_{\mathsf{pair}}(x,y):C$ for arbitrary $x:A$ and $y:B(x)$, the assignment

$$t(\mathsf{pair}(x,y)) \quad :\equiv \quad c_{\mathsf{pair}}(x,y)$$

defines $t(x,y)$ for any $x:A$ and $y:B(x)$. The form of the recursor is

$$\frac{(x:A,y:B(x))\ z(x,y):C \qquad w:\sum(x:A)\,B(x)}{\mathsf{rec}_{\sum(x:A)\,B(x)}(z,w):C}$$

and yields the elimination rule

$$\dfrac{\begin{array}{c}(x:A,\phi(x))\\ \vdots\\ \theta\end{array} \qquad \exists(x:A)\,\phi(x)}{\theta}\ (\exists\text{-elim})$$

for the existential quantifier. (Notice that displaying the cancellation of $x:A$ is necessary here to avoid any undesired dependencies.)

5.6 Dependent product

As in the case of the product, we may relax the conditions for the function type: Given a type family $(x:A)\,B(x)$, the *dependent product* $\prod(x:A)\,B(x)$ is defined by the constructor

- For a family $(x:A)\,b(x):B(x)$, $\lambda(x:A)\,b(x):\prod(x:A)\,B(x)$.

Again, using the same symbol for the constructors of the function type and the dependent product is justified by the latter being a generalization of the former. Omitting the elements yields the introduction rule

$$\dfrac{\begin{array}{c}(x:A)\\ \vdots\\ \phi(x)\end{array}}{\forall(x:A)\,\phi(x)}\ (\forall\text{-intro})$$

of the universal quantifier. (Notice, once more, the necessity of displaying the cancellation of $x:A$.)

Recursion principle: Given a family $((x:A)\,y(x):B(x))\,c_\lambda(y):C$, the assignment

$$t(\lambda(x:A)\,b(x)) \quad :\equiv \quad c_\lambda((x:A)\,b(x))$$

defines $t(f)$ for arbitrary $f:\prod(x:A)\,B(x)$. The recursor of $\prod(x:A)\,B(x)$ has the formation rule

$$\dfrac{((x:A)\,y(x):B(x))\,z(y):C \qquad f:\prod(x:A)\,B(x)}{\mathsf{rec}_{\prod(x:A)\,B(x)}(z,f):C}$$

and is defined by the recursion

$$\mathsf{rec}_{\prod(x:A)\,B(x)}(z,\lambda(x:A)\,b(x)) \quad :\equiv \quad z((x:A)\,b(x)).$$

If we erase the elements, this becomes the elimination rule

$$\dfrac{\begin{array}{c}\left(\dfrac{x:A}{\phi(x)}\right)\\ \vdots\\ \theta\end{array} \qquad \forall(x:A)\,\phi(x)}{\theta}\ (\forall\text{-elim})$$

of the universal quantifier.

5.7 Projections and function application

Of the types defined above, those that have a single constructor admit simpler (and familiar) constructs equivalent to (i.e., interdefinable with) their recursors. First, we may define, for $x : A_1 \times A_2$, the *projections*

$$\mathsf{pr}_i(x) : A_i, \quad i = 1, 2$$

by the recursion

$$\mathsf{pr}_i(\mathsf{pair}(x_1, x_2)) \quad :\equiv \quad x_i.$$

Then, the recursor of $A_1 \times A_2$ may be expressed in terms of pr_1 and pr_2 by setting

$$\mathsf{rec}_{A_1 \times A_2}(z, x) \quad :\equiv \quad z(\mathsf{pr}_1(x), \mathsf{pr}_2(x)).$$

This definition satisfies defining property of $\mathsf{rec}_{A_1 \times A_2}$, namely,

$$\begin{aligned} \mathsf{rec}_{A_1 \times A_2}(z, \mathsf{pair}(x_1, x_2)) &\equiv z(\mathsf{pr}_1(\mathsf{pair}(x_1, x_2)), \mathsf{pr}_2(\mathsf{pair}(x_1, x_2))) \\ &\equiv z(x_1, x_2). \end{aligned}$$

Similarly, we define the *application* $\mathsf{apply}_f(a)$ of $f : A \to B$ to $a : A$ by the recursion (on f)

$$\mathsf{apply}_{\lambda(x:A)\, b(x)}(a) \quad :\equiv \quad b(a).$$

The recursor of $A \to B$ may then be defined in terms of function application:

$$\mathsf{rec}_{A \to B}(z, f) \quad :\equiv \quad z((x : A)\, \mathsf{apply}_f(x)).$$

The defining property of $\mathsf{rec}_{A \to B}$ is satisfied:

$$\begin{aligned} \mathsf{rec}_{A \to B}(z, \lambda(x : A)\, b(x)) &\equiv z((x : A)\, \mathsf{apply}_{\lambda(x:A)\, b(x)}(x)) \\ &\equiv z((x : A)\, b(x)). \end{aligned}$$

We will follow common practice and write $f(x)$ instead of $\mathsf{apply}_f(x)$. We often use function application and projections in place of recursion over functions respectively pairs.

The above extend, mutatis mutandis, to dependent sums and products. Note, also, that these constructs yield the familiar elimination rules for conjunction, implication, and universal quantification (the existential quantifier does not have such special elimination rules).

Bottom line

We have reconstructed intuitionistic first-order logic within MLTT. Consequently, propositions of first-order logic may now be expressed by types (pending the definition of useful predicates, like equality, to be discussed next). What this means, in practice, is that formulating a theorem amounts to describing a type, and proving it amounts to exhibiting an element of that type. This may well take place in natural language. For example, we may show (the type-theoretic analogue of) the transitivity of implication:

Theorem. *If $A \to B$ then if $B \to C$ then $A \to C$.*

Proof. We need to exhibit a function $F : (A \to B) \to ((B \to C) \to (A \to C))$. Let $f : A \to B$ and $g : B \to C$. Then, for $x : A$, $g(f(x)) : C$. Hence,

$$g \circ f :\equiv \lambda(x : A)\, g(f(x)) : A \to C.$$

We may now abstract g and f in this order to obtain the desired function

$$\lambda(f : A \to B)\, \lambda(g : B \to C)\, g \circ f. \qquad \square$$

Exercises

Exercise. Define multiplication on Nat by recursion and/or using the recursor. Optionally, continue with exponentiation and the factorial.

Exercise. Show the following logical facts.

(i) $A \to A$.

(ii) $A \to \neg\neg A$.

(iii) $\neg\neg(A + \neg A)$.

(iv) $(A + \neg A) \to (\neg\neg A \to A)$.