M782
Special Topics in Logic: Type Theory
Lecture notes

Nikos Rigas[1]

Version 2025.10.13

[1]Email: nicolasr@di.uoa.gr

Page intentionally left blank

Chapter 1

Martin-Löf's Type Theory

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 constructs 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 $e_1 :\equiv e_2$. 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*.

Types and typing judgments Types have elements (more on this later). We use uppercase letters for types and lowercase letters for their elements. We write $a : A$ to express the judgment that a is an element of type A.

Families If we are given an element $f(x)$ of B for each element x of A, what we have is a family of elements of B indexed by A. Such an *indexed family* is denoted by

$$(x : A)\, f(x) : B,$$

or by the equivalent *formation rule*

$$\frac{x : A}{f(x) : B}\ ,$$

and we say "for (arbitrary) x in A, $f(x)$ in B" or "an element $f(x)$ of B for each element x of A." Similarly,

$$(x_1 : A_1, \dots, x_n : A_n)\, f(x) : A$$

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.

Indices may themselves be families. This is reflected in expressions like

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

which signifies a family depending on elements of A and families of elements of C depending on elements of B.

An important special case is families of types, a.k.a. *type families*. As usual, a type family $(x : A)\, B(x)$ is the same thing as one type $B(x)$ for each $x : A$.

1.1 Inductive types and recursion

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

- there is a natural number, called *zero*, and
- for each natural number there is a natural number, called its *successor*.

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

- $0 : \mathbb{N}$, and
- $(n : \mathbb{N})\, \mathsf{s}(n) : \mathbb{N}$.

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

The elements of $\mathbb{N}$ that can be formed with the help of the constructors are

$$0,\ \mathsf{s}(0),\ \mathsf{s}(\mathsf{s}(0)),\ \ldots,$$

for which we use the familiar numerals 0, 1, 2 and so on.

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; they will be employed as the need arises.

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 $\mathbb{N}$ this is the familiar definition by primitive recursion: Given a type C, an element c_0 of C and elements $c_{\mathsf{s}}(n, c)$ of C, one for each $n : \mathbb{N}$ and each $c : C$, the assignments

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

define $f(n) : C$ for any $n : \mathbb{N}$.

As a first example, let us define the predecessor $\mathsf{pred}(n)$ of a natural number n: In this case, the type C is $\mathbb{N}$ (the only type we have at the moment), and the defining assignments are

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

That is, we instantiate the general principle of recursion to the case where c_0 is 0 and $c_s(x, y)$ is x.

We may also define families with several indices by recursion with respect to one of them. For instance, we may define addition of natural numbers by recursion in the second operand:

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

In other words, addition (to m) is defined by the instance of $\mathbb{N}$-recursion where $c_0 \equiv m$ and $c_s(n, c) \equiv s(c)$. Any definition by recursion over the natural numbers is determined by these two parameters: The element c_0 and the family c_s. By turning these into indices of the family being defined, we arrive at the *recursor*

$$(c_0 : C, (x : \mathbb{N}, y : C)\, c_s(x, y) : C, n : \mathbb{N})\, \mathsf{rec}^C_{\mathbb{N}}(c_0, c_s, n) : C$$

of $\mathbb{N}$, defined by the recursion

$$\begin{aligned} \mathsf{rec}^C_{\mathbb{N}}(c_0, c_s, 0) &\coloneqq c_0, \\ \mathsf{rec}^C_{\mathbb{N}}(c_0, c_s, s(n)) &\coloneqq c_s(n, \mathsf{rec}^C_{\mathbb{N}}(c_0, c_s, n)). \end{aligned}$$

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

Any recursive family can be explicitly defined with the help of the recursor; for example, the predecessor of a natural number can be expressed as

$$\mathsf{pred}(n) \coloneqq \mathsf{rec}_{\mathbb{N}}(0, (x : \mathbb{N}, y : \mathbb{N})\, x, n),$$

and the sum of two natural numbers can be expressed as

$$m + n \coloneqq \mathsf{rec}_{\mathbb{N}}(m, (x : \mathbb{N}, y : \mathbb{N})\, s(y), n).$$

Exercise 1.1.1. Define multiplication, exponentiation and the factorial.

Another common example is the type $\mathrm{List}(A)$ of lists of elements of a type A, defined by the induction scheme

- $\mathsf{nil}_A : \mathrm{List}(A)$,
- $(a : A, l : \mathrm{List}(A))\, \mathsf{cons}_A(a, l) : \mathrm{List}(A)$.

nil_A stands for the empty list, and $\mathsf{cons}_A(a, l)$ stands for the extension of l by a. In practice, subscripts are omitted when A is understood.

The recursion principle for $\mathrm{List}(A)$ takes the following form: Given

- a type C,
- an element c_{nil} of C, and
- an element $c_{\mathsf{cons}}(x, y, z) : C$ for arbitrary $x : A$, $y : \mathrm{List}(A)$ and $z : C$,

the assignments

$$\begin{aligned} f(\mathsf{nil}) &\coloneqq c_{\mathsf{nil}}, \\ f(\mathsf{cons}(a, l)) &\coloneqq c_{\mathsf{cons}}(a, l, f(l)) \end{aligned}$$

define $f(l) : C$ for arbitrary $l : \mathrm{List}(A)$. For example, the length $\mathsf{len}(l)$ of a list l is defined by the recursion

$$\begin{aligned} \mathsf{len}(\mathsf{nil}) &:\equiv 0, \\ \mathsf{len}(\mathsf{cons}(a, l)) &:\equiv \mathsf{s}(\mathsf{len}(l)), \end{aligned}$$

and the concatenation $l + k$ of two lists l and k is defined by the recursion

$$\begin{aligned} \mathsf{nil} + k &:\equiv k, \\ \mathsf{cons}(a, l) + k &:\equiv \mathsf{cons}(a, l + k), \end{aligned}$$

while the sum $\mathsf{sum}(l)$ of the elements of a list l of natural numbers is defined by

$$\begin{aligned} \mathsf{sum}(\mathsf{nil}) &:\equiv 0, \\ \mathsf{sum}(\mathsf{cons}(a, l)) &:\equiv a + \mathsf{sum}(l). \end{aligned}$$

Like we did with $\mathbb{N}$, we may compile the induction principle of $\mathrm{List}(A)$ into a recursor

$$(c_{\mathsf{nil}} : C, (x : A, y : \mathrm{List}(A), z : C)\, c_{\mathsf{cons}}(x, y, z) : C, l : \mathrm{List}(A))\, \mathsf{rec}_{\mathrm{List}(A)}(c_{\mathsf{nil}}, c_{\mathsf{cons}}, l) : C$$

defined by the recursion

$$\begin{aligned} \mathsf{rec}_{\mathrm{List}(A)}(c_{\mathsf{nil}}, c_{\mathsf{cons}}, \mathsf{nil}) &:\equiv c_{\mathsf{nil}}, \\ \mathsf{rec}_{\mathrm{List}(A)}(c_{\mathsf{nil}}, c_{\mathsf{cons}}, \mathsf{cons}(a, l)) &:\equiv c_{\mathsf{cons}}(a, l, \mathsf{rec}_{\mathrm{List}(A)}(c_{\mathsf{nil}}, c_{\mathsf{cons}}, l)). \end{aligned}$$

The length of a list can now be defined as

$$\mathsf{len}(l) :\equiv \mathsf{rec}_{\mathrm{List}(A)}(0, (x : A, y : \mathrm{List}(A), z : \mathbb{N})\, \mathsf{s}(z), l),$$

the concatenation of two lists as

$$l + k :\equiv \mathsf{rec}_{\mathrm{List}(A)}(k, (x : A, y : \mathrm{List}(A), z : \mathrm{List}(A))\, \mathsf{cons}(x, z), l),$$

and the sum of a list of natural numbers as

$$\mathsf{sum}(l) :\equiv \mathsf{rec}_{\mathrm{List}(\mathbb{N})}(0, (x : A, y : \mathrm{List}(A), z : \mathbb{N})\, x + z, l).$$

Exercise 1.1.2. The concatenation $\mathsf{cat}(L) : \mathrm{List}(A)$ of a list $L : \mathrm{List}(\mathrm{List}(A))$ of lists of elements of a type A is defined by the recursion

$$\begin{aligned} \mathsf{cat}(\mathsf{nil}_{\mathrm{List}(A)}) &:\equiv \mathsf{nil}_A, \\ \mathsf{cat}(\mathsf{cons}_{\mathrm{List}(A)}(l, L)) &:\equiv l + \mathsf{cat}(L). \end{aligned}$$

Express $\mathsf{cat}(L)$ using the recursor of $\mathrm{List}(\mathrm{List}(A))$.

The type BinTree of binary trees is captured by the induction scheme

- $\mathsf{triv} : \mathrm{BinTree}$,
- $(t_1 : \mathrm{BinTree}, t_2 : \mathrm{BinTree})\, \mathsf{join}(t_1, t_2) : \mathrm{BinTree}$.

This inductive definition gives rise to the following induction principle: Given

- a type C,

- an element $c_{\mathsf{triv}} : C$, and
- a family $(x, y : \mathrm{BinTree}, z, w : C)\, c_{\mathsf{join}}(x, y, z, w) : C$,

the assignments

$$\begin{aligned} f(\mathsf{triv}) &\equiv c_{\mathsf{triv}}, \\ f(\mathsf{join}(t_1, t_2)) &\equiv c_{\mathsf{join}}(t_1, t_2, f(t_1), f(t_2)), \end{aligned}$$

define a family $(t : \mathrm{BinTree})\, f(t) : C$.

Exercise 1.1.3. Describe $\mathsf{rec}_{\mathrm{BinTree}}$ and write its defining assignments.

Binary trees are a special case of trees of given branching. The type $\mathrm{Tree}(A)$ of trees of branching type A is defined by the induction scheme

- $\mathsf{triv}_A : \mathrm{Tree}(A)$.
- $((x : A)\, t(x) : \mathrm{Tree}(A))\, \mathsf{join}_A(x : A)\, t(x) : \mathrm{Tree}(A)$.

More descriptively, the second constructor takes a tree $t(x)$ for each $x : A$ and connects all these trees with a new root to form the tree $\mathsf{join}_A(x : A)\, t(x)$. In order to extend a family f to $\mathsf{join}_A(x : A)\, t(x)$, assuming we already have $f(t(x)) : C$ for $x : A$, we need a family $((x : A)\, y(x) : \mathrm{Tree}(A), (x : A)\, z(x) : C)\, c_{\mathsf{join}_A}(y, z) : C$; together with an element $c_{\mathsf{triv}} : C$, we may define f by setting

$$\begin{aligned} f(\mathsf{triv}_A) &\equiv c_{\mathsf{triv}_A}, \\ f(\mathsf{join}_A(x : A)\, t(x)) &\equiv c_{\mathsf{join}_A}((x : A)\, t(x), (x : A)\, f(t(x))). \end{aligned}$$

Since the type A occurs negatively in join_A, $\mathrm{Tree}(A)$ is expected to depend contravariantly on A. Indeed, a family $(x : B)\, u(x) : A$ induces a reparametrization

$$(x : \mathrm{Tree}(A))\, \mathrm{Tree}(u)(x) : \mathrm{Tree}(B)$$

defined by

$$\begin{aligned} \mathrm{Tree}(u)(\mathsf{triv}_A) &\equiv \mathsf{triv}_B, \\ \mathrm{Tree}(u)(\mathsf{join}_A(x : A)\, t(x)) &\equiv \mathsf{join}_B(y : B)\, \mathrm{Tree}(u)(b(u(y))). \end{aligned}$$

Exercise 1.1.4. Describe the recursor of $\mathrm{Tree}(A)$, and express $\mathrm{Tree}(u)$ with its help.

Exercise 1.1.5. In the definition of $\mathrm{List}(A)$, the type A occurs positively. Given a family $(x : A)\, u(x) : B$ define, using the recursion principle or the recursor of $\mathrm{List}(A)$, the family $(x : \mathrm{List}(A))\, \mathrm{List}(u)(x) : \mathrm{List}(B)$ which transforms a list of elements of A to the list of their u-images.

The following induction scheme describes a type with exactly two elements:

- $\mathbb{f} : \mathbb{B}$,
- $\mathbb{t} : \mathbb{B}$.

The recursion principle for $\mathbb{B}$ expresses the fact that in order to define a family $(x : \mathbb{B})\, f(x)$ we merely need to specify $f(\mathbb{f})$ and $f(\mathbb{t})$. Specifically, given two elements $c_{\mathbb{f}}$ and $c_{\mathbb{t}}$ of a type C, the assignments

$$\begin{aligned} f(\mathbb{f}) &:\equiv c_{\mathbb{f}}, \\ f(\mathbb{t}) &:\equiv c_{\mathbb{t}} \end{aligned}$$

define a family $(x : \mathbb{B})\, f(x) : C$. As usual, we have a recursor

$$(c_{\mathbb{f}} : C, c_{\mathbb{t}} : C, x : \mathbb{B})\, \mathsf{rec}_{\mathbb{B}}(c_{\mathbb{f}}, c_{\mathbb{t}}, x) : C$$

with defining assignments

$$\begin{aligned} \mathsf{rec}_{\mathbb{B}}(c_{\mathbb{f}}, c_{\mathbb{t}}, \mathbb{f}) &:\equiv c_{\mathbb{f}}, \\ \mathsf{rec}_{\mathbb{B}}(c_{\mathbb{f}}, c_{\mathbb{t}}, \mathbb{t}) &:\equiv c_{\mathbb{t}}. \end{aligned}$$

For example, we can define the truth table of disjunction by the recursion

$$\begin{aligned} \mathsf{or}(x, \mathbb{f}) &:\equiv x, \\ \mathsf{or}(x, \mathbb{t}) &:\equiv \mathbb{t}, \end{aligned}$$

or, alternatively, via the recursor of $\mathbb{B}$,

$$\mathsf{or}(x, y) :\equiv \mathsf{rec}_{\mathbb{B}}(x, \mathbb{t}, y).$$

Exercise 1.1.6. Define the truth tables of conjunction, implication, and negation.

Unlike mathematical treatments of type theory, most programming languages opt for descriptive rather than uniform names for the recursors of the various types. For example, $\mathsf{rec}_{\mathbb{B}}(c_{\mathbb{f}}, c_{\mathbb{t}}, b)$ is usually written

$$\texttt{if } b \texttt{ then } c_{\mathbb{t}} \texttt{ else } c_{\mathbb{f}}$$

or a variant thereof.

Exercises

Exercise 1.1.7. Define the list $\mathsf{reverse}(l)$ that contains the entries of a list l in reverse order.

1.2 Logic

Type theory is intimately related to logic, especially proof theory. The apex of this relationship is the *propositions-as-types* interpretation, which is detailed in this section. However, here lies a significantly deeper fact, which has to do with the reading of type theory as a theory of meaning of analytic propositions. The question of the determination of analytic propositions has a rich history and is connected with various other extremely interesting philosophical questions, the treatment of which is beyond the scope of these notes. We will only try to outline the reasoning that underlies the relationship between logic and type theory.

The proof process is absolutely central to mathematics. And rightly so, since it is the way in which we acquire knowledge of mathematical propositions. But what is

the relation between proof and knowledge? According to the constructive reading of logic, to prove ϕ is to effect a construction attesting to its truth, and to know (that) ϕ is to possess such a construction.

A proof may, in addition to a conclusion, also have assumptions. In this case, the knowledge that is conveyed is conditional. In other words, a proof with assumptions provides someone who knows the assumptions with a way to know the conclusion.

In this way, we have clarified the relation between proof and knowledge. It still remains to say what the proofs of propositions are. In this connection, observe that what constitutes a proof of each proposition is precisely what one needs to know in order to be able both to formulate proofs and to understand the proofs that others formulate, i.e., recognize the proofs that others formulate as such.

Thesis. A proposition is a specification of what constitutes a proof of it.

A *sentence* is a linguistic expression whose referent is a proposition. In light of the above, to understand a sentence is to know which proposition it refers to. Then, to attribute meaning to a sentence is to associate a specification of proofs to it. On the other hand, the determination of the meaning of the sentences of an existing language such as the language of mathematics cannot be arbitrary, but must be based on an analysis of the use of that language. For the use of logical constants in constructive mathematics, the relevant analysis is summarized in the *Brouwer-Heyting-Kolmogorov (BHK) interpretation*:

- A proof of $\phi_1 \wedge \phi_2$ consists of a proof of ϕ_1 and a proof of ϕ_2.
- Proofs of $\phi_1 \vee \phi_2$ are the proofs of ϕ_1 as well as those of ϕ_2.
- A proof of $\phi \supset \psi$ is a construction that transforms an arbitrary proof of ϕ into a proof of ψ.
- There are no proofs of $\bot$.
- There is a proof of $\top$.
- A proof of $\forall (x : A)\, \phi(x)$ is a construction that transforms an arbitrary element a of A into a proof of $\phi(a)$.
- A proof of $\exists (x : A)\, phi(x)$ consists of an element a of A and a proof of $\phi(a)$.

BHK clarifies the way in which logical constants are used in constructive mathematics. It does this by describing the meaning of each complex sentence in terms of the meanings of the simpler sentences from which it is composed. In fact, each of the clauses of BHK is an induction scheme. Consequently, we can say that the meaning of each sentence is the type of its proofs. This allows us to extend our type-theoretic terminology and type-theoretic notation to logic; in particular, we write $x : \phi$ if x is a proof of ϕ. We will examine the connectives in turn.

1.2.1 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

$$\frac{\phi_1 \qquad \phi_2}{\phi_1 \wedge \phi_2}$$

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

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

of conjunction.

1.2.2 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 $(x : A)\, b(x) : B$, $\lambda(b) : A \to B$.

By omitting the elements and switching to logical notation, this yields the introduction rule

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

of implication.

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

$$t(\lambda(b)) \quad :\equiv \quad c_\lambda(b)$$

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

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

If we erase the elements, this becomes the elimination rule

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

for implication.

1.2.3 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

$$\frac{\phi_1}{\phi_1 \vee \phi_2} \qquad\qquad \frac{\phi_2}{\phi_1 \vee \phi_2}$$

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

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

(proof by cases) for disjunction.

1.2.4 $\mathbb{0}$

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

$$\frac{\bot}{\theta}$$

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

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

1.2.5 $\mathbb{1}$

The type $\mathbb{1}$ corresponds to verum ($\top$). It has the single constructor

- $\star : \mathbb{1}$

that corresponds to the introduction rule

$$\frac{}{\top}\ .$$

In order to define a family $(x : \mathbb{1})\, f(x)$ we merely need to specify $f(\star)$; hence, the recursor of $\mathbb{1}$ has the form

$$\frac{c_\star : C \qquad x : \mathbb{1}}{\mathsf{rec}_{\mathbb{1}}(c_\star, x) : C}$$

and yields the elimination rule

$$\frac{\theta \qquad \top}{\theta}$$

of $\top$.

1.2.6 Dependent sum

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

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

(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

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

of the existential quantifier.

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

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

defines $t(w)$ for any $w\colon \sum B$. The form of the recursor is

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

and yields the elimination rule

$$\frac{\begin{array}{c}(x\colon A, \phi(x))\\ \vdots \\ \theta\end{array} \qquad \exists(x\colon A)\,\phi(x)}{\theta}$$

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

1.2.7 Dependent product

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

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

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

$$\frac{\begin{array}{c}(x\colon A)\\ \vdots \\ \phi(x)\end{array}}{\forall(x\colon A)\,\phi(x)}$$

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

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

$$t(\lambda(b)) \quad :\equiv \quad c_\lambda(b)$$

defines $t(f)$ for arbitrary $f\colon \prod B$. The recursor of $\prod B$ has the formation rule

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

and is defined by the recursion

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

If we erase the elements, this becomes the elimination rule

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

of the universal quantifier.

1.2.8 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

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

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

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

This definition satisfies the 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* $\mathsf{apply}_f(a):B$ of $f:A\to B$ to $a:A$ by the recursion (on f)

$$\mathsf{apply}_{\lambda(b)}(a) \;\equiv\; b(a).$$

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

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

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

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

Finally, we note that the recursor of $\mathbb{1}$ is definable even without recursion:

$$\mathsf{rec}_{\mathbb{1}}(c_\star,x) \;\equiv\; c_\star.$$

We will follow common practice and write $f(x)$ (or fx) in place of $\mathsf{apply}_f(x)$. We mostly 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 verum (i.e., no elimination rule), conjunction, implication (i.e., modus ponens), and universal quantification (i.e., universal modus ponens); the existential quantifier does not have such special elimination rules. All natural deduction rules may be found in table 1.1.

	Introduction rules	Elimination rules	(Special)
$\wedge$	$\dfrac{\phi_1 \quad \phi_2}{\phi_1 \wedge \phi_2}$	$\dfrac{\begin{array}{c}(\phi_1, \phi_2)\\ \vdots \\ \theta \end{array} \qquad \phi_1 \wedge \phi_2}{\theta}$	$\dfrac{\phi_1 \wedge \phi_2}{\phi_1} \qquad \dfrac{\phi_1 \wedge \phi_2}{\phi_2}$
$\vee$	$\dfrac{\phi_1}{\phi_1 \vee \phi_2} \qquad \dfrac{\phi_2}{\phi_1 \vee \phi_2}$	$\dfrac{\begin{array}{cc}(\phi_1) & (\phi_2)\\ \vdots & \vdots \\ \theta & \theta \end{array} \qquad \phi_1 \vee \phi_2}{\theta}$	
$\supset$	$\dfrac{\begin{array}{c}(\phi)\\ \vdots \\ \psi \end{array}}{\phi \supset \psi}$	$\dfrac{\begin{array}{c}\left(\dfrac{\phi}{\psi}\right)\\ \vdots \\ \theta \end{array} \qquad \phi \supset \psi}{\theta}$	$\dfrac{\phi \supset \psi \qquad \phi}{\psi}$
$\bot$		$\dfrac{\bot}{\theta}$	
$\top$	$\dfrac{}{\top}$	$\dfrac{\theta \quad \top}{\theta}$	No elimination rule
$\exists$	$\dfrac{a : A \quad \phi(a)}{\exists (x : A)\, \phi(x)}$	$\dfrac{\begin{array}{c}(x : A, \phi(x))\\ \vdots \\ \theta \end{array} \qquad \exists (x : A)\, \phi(x)}{\theta}$	
$\forall$	$\dfrac{\begin{array}{c}(x : A)\\ \vdots \\ \phi(x) \end{array}}{\forall (x : A)\, \phi(x)}$	$\dfrac{\begin{array}{c}\left(\dfrac{x : A}{\phi(x)}\right)\\ \vdots \\ \theta \end{array} \qquad \forall (x : A)\, \phi(x)}{\theta}$	$\dfrac{\forall (x : A)\, \phi(x) \qquad a : A}{\phi(a)}$

Table 1.1: Natural deduction 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 analogues of) the reflexivity and transitivity of implication:

Theorem. (i) $A \to A$.

(ii) *If* $A \to B$ *then if* $B \to C$ *then* $A \to C$.

Proof. (i) We need to exhibit a function $F : A \to A$; this is served by the identity

$$\mathsf{id}_A :\equiv \lambda(x : A)\, x : A \to A.$$

(ii) We have 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 1.2.1. Show the following logical facts.

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

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

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

Exercise 1.2.2. Let E be the type defined by the single constructor

- For $x : E$, $e(x) : E$.

This type would result from $\mathbb{N}$ if we removed the constructor 0. Intuitively, E should have no elements, as there is no way to bootstrap production. Show that this is indeed the case, i.e., prove $\neg E$. [Hint: Define a function $f : E \to \mathbb{0}$ by E-recursion.]