Introductory notes on Martin-Löf's Type Theory

Nikos Rigas*

Version 2024-11-28

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 of 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.

The last view is supported by recent developments involving an extended notion of inductive type (higher inductive types) as well as an axiom (univalence) that may be understood as a precise expression of the structuralist principle (identity of isomorphs) in type theory. The extension of MLTT with these new ingredients is called homotopy type theory (HoTT) and offers a number of views of its own; the main ones are

- A synthetic language for homotopy theory: It allows the development of homotopy theory in a way that avoids concrete topological representations. For instance, paths are no longer represented as continuous functions from the interval (although they could be); rather, they are defined inductively.
- The internal language of weak ∞-topoi: Similarly, it offers an alternative to categorical diagrams, which are difficult or impossible to reason with in this context.
- A foundational theory that is at once constructivist and structuralist: It realizes the prospect of developing (at least) those branches of mathematics that study abstract structures (e.g., algebra, geometry) while avoiding the need to resort to an external theory (e.g., set theory) to construct or otherwise find instances of these structures.

This suggests that HoTT may indeed function as a unifying framework for a range of diverse aspects of mathematics.

These notes, which are meant as a prelude to HoTT, will emphasize the foundational character of type theory and the main ideas underlying it.

*Email: nicolasr@di.uoa.gr

Remarks on the presentation

It has become established practice (after the HoTTbook, presumably) to present (homotopy) type theory taking Σ-types, Π-types and universes as primitive. From the point of view of the formal system, it makes little difference, as it is purely an organizational issue, and may indeed be a shorter path to the stage of development where homotopical ideas and results can be discussed. From a conceptual point of view, deciding what to take as primitive is of essence, as it reflects the conceptual precedence between the various ingredients of the theory. For a constructive theory such as MLTT, getting the order of precedence right means, among other things, preserving the interdependence between constructions. In these notes, we have chosen to only accept constructors of inductive types as primitive (as they express elementary, indecomposable and otherwise irreducible acts of construction), and to treat everything else (including Σ-types, Π-types and universes) as defined. Constructors of inductive and inductive-recursive types (such as zero and successor for the natural numbers, pairs for products and dependent sums, lambdas for function types and dependent products, false and true for the type of truth values, type formers for universes, etc.), inductive type families (such as equality, fibers, etc.), recursive type families (such as observational equality, heterogeneous equality, truth predicates, etc.), as well as recursors and other constructs, are all expressed by families of various shapes and forms. These families may depend on elements of types, but they may also depend on families of elements of types (e.g., lambda), families of families (e.g., recursors of function types), and so on. Consequently, our language provides for an arbitrary nesting of dependencies.

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 may 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 this later). We write $a : A$ to express the judgment that a is an element of A.

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,

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

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.

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

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

which signifies a family depending on elements of A and families of elements of C depending on 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; they will be introduced as the need arises.

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

Exercises

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

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 $A_1, \dots, A_n$ yields evidence for A conditional on evidence for $A_1, \dots, A_n$, i.e., it is a family of elements of A indexed by $A_1, \dots, A_n$.

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

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

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.

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

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

of implication.

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

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

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

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

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

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

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

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

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.

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

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

(*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

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

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 $t(w)$ for any $w : \sum(x:A)\,B(x)$. 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

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

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

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

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

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

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

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

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 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)

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

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 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. Show the following logical facts.

(i) $A \to A$.

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

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

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

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

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

This type would result from Nat if we removed the constructor 0. Intuitively, E should have no elements. Prove that this is indeed the case, i.e., show that $\neg E$. [Hint: Formulate its recursion principle first.]

6 Equality

Predicate logic also involves predicates, such as $x < y$, Prime(n), and so on. Under the propositions-as-types interpretation, predicates correspond to families of types, indexed by the respective domains of the arguments.

The definition of a family by recursion applies, in particular, to type families (more on this in the section on universes). Another option is to define a type family by giving constructors for its various instances. The definition of equality exploits the latter possibility.

Equality of a type A may be defined either as a family $(x:A, y:A)\ x = y$ with respect to both sides, or as a family $(x:A)\ a = x$ for each particular element a of A; we will examine them in turn.

6.1 Based equality

Let $a:A$. *Equality-to-a* is the type family $(x:A)\ a = x$ with the single constructor

- $\mathsf{refl}_a : a = a$.

This constructor corresponds to the introduction rule

$$\frac{}{a = a}\ .$$

We often refer to elements of $a = b$ as *identifications* between a and b.

Recursion with respect to a type family is a principle of definition over all instances of the family at once. In the case of based equality, the recurion principle concerns itself with the definition of families of the form $(x:A, p:a = x)\ t(x, p):C(x)$ into an arbitrary type family $(x:A)C(x)$ of the same shape as based equality. Its name comes from homotopy type theory, and refers to the interpretation of identifications as paths.

Based-path recursion: Given a type family $(x:A)\ C(x)$ and an element c_{refl_a} of $C(a)$, the assignment

$$t(a, \mathsf{refl}_a) \quad :\equiv \quad c_{\mathsf{refl}_a}$$

defines $t(x, p):C(x)$ for any $x:A$ and $p:a = x$.

Notation. For families of the form $t(x, p)$ for $x:A$ and $p:a = x$, the first argument may be suppressed, as it is determined by the second one, and we may simply write $t(p)$.

Recursors may be interesting from a logical perspective, because they correspond to elimination rules, but we generally find it more natural and convenient to formulate definitions by recursion directly. The exception is the recursor of based equality, which is useful enough to have its own name,

$$\frac{x:C(a) \qquad p:a = b}{\mathsf{transport}^C(p, x) :\equiv \mathsf{rec}^C_{a=\_}(x, p):C(b)}\ .$$

$\mathsf{transport}^C(p, x)$ is pronounced "the transport of $x:C(a)$ to $C(b)$ along $p:a = b$".

The elimination rule corresponding to transport is the law of the *indiscernibility of identicals*:

$$\frac{\phi(a) \qquad a = b}{\phi(b)}\ .$$

The following operations testify that based equality is an equivalence relation.

Reflexivity Let $a : A$. Then,

$$\mathsf{refl}_a : a = a.$$

Transitivity Let $p : a = b$ and $q : b = c$. Then,

$$p \cdot q :\equiv \mathsf{transport}^{a=\_}(q, p) : a = c.$$

Symmetry Let $p : a = b$. Then,

$$p^{-1} :\equiv \mathsf{transport}^{\_=a}(p, \mathsf{refl}_a) : b = a.$$

Based equality is also preserved by families: Let $(x : A)\ u(x) : B$. Then, we may define $u(p) : u(a) = u(b)$ for arbitrary $p : a = b$ by means of the recursion

$$u(\mathsf{refl}_a) :\equiv \mathsf{refl}_{u(a)},$$

or by transporting $\mathsf{refl}_{u(a)} : u(a) = u(a)$ to $u(a) = u(b)$ along p:

$$u(p) :\equiv \mathsf{transport}^{u(a)=u(\_)}(p, \mathsf{refl}_{u(a)}).$$

6.2 Symmetric equality

The *symmetric equality* of a type A is the type family $(x : A, y : A)\ x = y$ having the constructor

- For $x : A$, $\mathsf{refl}_x : x = x$.

As was the case with based equality, recursion over symmetric equality is a principle for defining families of the form $(x : A, y : A, p : x = y)\ t(x, y, p) : C(x, y)$ into a type family $(x : A, y : A)\ C(x, y)$ of the same shape as symmetric equality.

Path recursion: Given types $C(x, y)$ for $x, y : A$ and elements $c_{\mathsf{refl}}(x) : C(x, x)$ for $x : A$, the assignment

$$t(x, x, \mathsf{refl}_x) \quad :\equiv \quad c_{\mathsf{refl}}(x)$$

defines $t(x, y, p) : C(x, y)$ for arbitrary $x, y : A$ and $p : x = y$.

The recursor of symmetric equality is

$$((x : A)\ z(x) : C(x, x), a, b : A, p : a = b)\ \mathsf{rec}^{C}_{=}(z, p) : C(a, b)$$

and corresponds to the alternative elimination rule

$$\frac{\begin{array}{c}(x : A)\\ \vdots\\ \phi(x, x) \qquad a = b\end{array}}{\phi(a, b)}$$

of equality, which says that equals satisfy any reflexive relation.

6.3 Equivalence between the two definitions

Based equality and symmetric equality are, essentially, two different descriptions of the same relation. This is testified by the interderivability between the respective recursion principles (equivalently, the interdefinability between $\mathsf{rec}_=$ and $\mathsf{transport}$). One direction is straightforward: Given $(x:A)\, z(x):C(x,x)$ and $p:a=b$ in A, an element of $C(a,b)$ may be obtained by transporting $z(a):C(a,a)$ along p,

$$\mathsf{rec}_=^C(z,p) \quad :\equiv \quad \mathsf{transport}^{C(a,\_)}(p,z(a)).$$

The verification of the defining property of $\mathsf{rec}_=$ is left to the reader. The other direction isn't particularly difficult either, provided we have function types at our disposal: Let $(x:A)\ C(x)$ be a family of types over A. We first define functions $f_p:C(a)\to C(b)$ for $p:a=b$ by means of the path recursion

$$f_{\mathsf{refl}_x} \quad :\equiv \quad \mathsf{id}_{C(x)}.$$

We may then define

$$\mathsf{transport}^C(p,x) \quad :\equiv \quad f_p(x).$$

It is possible, as a matter of fact, to define the entire family

$$(a,b:A,\, p:a=b,\, w:C(a))\ \mathsf{transport}^C(p,w):C(b)$$

by the path recursion

$$\mathsf{transport}^C(\mathsf{refl}_x,w) \quad :\equiv \quad w$$

without mentioning function types. This definition is essentially correct, but requires a more general form of path recursion. See the next, optional, paragraph and the exercises that follow it for some discussion.

More general forms of recursion

A definition by Nat-recursion of the form

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

may be understood as defining $(n:\mathrm{Nat})\ t(n,z)$ for each individual z. Sometimes, however, $t(\mathsf{s}(n),z)$ is defined in terms of $t(n,z')$ for (several) arbitrary values of z'. This situation arises, e.g., when we do simultaneous recursion in two arguments:

$$\begin{aligned} \mathsf{is\_equal}(0,0) &:\equiv \mathsf{true},\\ \mathsf{is\_equal}(0,\mathsf{s}(n)) &:\equiv \mathsf{false},\\ \mathsf{is\_equal}(\mathsf{s}(m),0) &:\equiv \mathsf{false},\\ \mathsf{is\_equal}(\mathsf{s}(m),\mathsf{s}(n)) &:\equiv \mathsf{is\_equal}(m,n). \end{aligned}$$

To express such a definition, we would need to supply the entire family $(z)\ t(n,z)$ as an argument to c_s:

$$\begin{aligned} t(0,z) &:\equiv c_0(z),\\ t(\mathsf{s}(n),z) &:\equiv c_\mathsf{s}(n,(z')\ t(n,z'),z). \end{aligned}$$

The above assignments determine $t(0, z)$ for all z and, once $t(n, z')$ is defined for all z', they determine $t(\mathsf{s}(n), z)$ for all z. Nevertheless, this is not equivalent to an ordinary definition by recursion over the natural numbers, despite being as legitimate a definition as any. Formulating a more general recursion principle to accommodate for this case is not particularly difficult, but we won't bother (see the exercises); instead, we will express our intent to employ this and other forms of definition as we see fit.

Exercises

Exercise. Based on the discussion of the previous paragraph, formulate a more general principle of definition by recursion over the natural numbers. Optionally, describe the corresponding recursor. Show how this principle can be reduced to (i.e., derived from) ordinary Nat-recursion in the presence of function types.

Exercise. A more general form of path recursion would be as follows: Given

- types $B(x, y)$ for $x, y : A$,
- types $C(x, y, z)$ for $x, y : A$ and $z : B(x, y)$, and
- elements $c_{\mathsf{refl}}(x, z)$ of $C(x, x, z)$ for $x : A$ and $z : B(x, x)$,

the assignment

$$t(x, x, \mathsf{refl}_x, z) \quad :\equiv \quad c_{\mathsf{refl}}(x, z)$$

defines $t(x, y, p, z) : C(x, y, z)$ for $x, y : A$, $p : x = y$ and $z : B(x, y)$. Use this principle to derive based-path recursion.

Exercise. A different generalization of Nat-recursion is necessary for expressing second-order recursive definitions such as the definition of the Fibonacci sequence. Formulate this principle. Optionally, describe the corresponding recursor. Show that this principle can be reduced to ordinary recursion in the presence of binary products.

7 Induction

8 Universes