Introductory notes on Martin-Löf's Type Theory

Nikos Rigas*

Version 2024-12-19

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 archetypal 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 can 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 constitute a shorter path to the stage of development where homotopical ideas and results can be discussed. The choice to treat univalence universes as primitive, in particular, is further justified by the non-constructibility of such universes. Nevertheless, we believe that the inductive-recursive definition of universes may offer useful additional insights on the notion of universe. Therefore, in these notes we have chosen to explicitly construct everything, i.e., to only accept constructors as primitive (as they express elementary, indecomposable and otherwise irreducible acts of construction), and to treat everything else 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 depending on families (e.g., recursors of function types), and so on. Consequently, our choice of language provides for an unlimited 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

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

$\mathsf{0}$ and s are the postulated ways of constructing natural numbers; they are the *constructors* of Nat. $\mathsf{0}$ 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 employed 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 $(x\colon \mathrm{Nat}, y\colon C)\ c_{\mathsf{s}}(x,y)$ of elements of C, the assignments

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

define $t(x)\colon C$ for arbitrary $x\colon \mathrm{Nat}$. 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*

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

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

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

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

Exercises

Exercise 4.1. Define multiplication on Nat by recursion and/or using the recursor. Optionally, do the same 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* $A \longrightarrow B$ *of functions* from A to B corresponds to logical implication; it is defined by the functional abstraction constructor:

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

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

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

of implication.

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

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

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

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

and is defined by the recursion

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

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.

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

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 $(x_1 : A_1)\, c_{\mathsf{in}_1}(x_1) \colon C$ and $(x_2 : A_2)\, c_{\mathsf{in}_2}(x_2) \colon C$, 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 $t(x) \colon C$ for any $x \colon A_1 + A_2$. The recursor has the form

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

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

$$\frac{\begin{matrix}(\phi_1) \\ \vdots \\ \theta\end{matrix} \qquad \begin{matrix}(\phi_2) \\ \vdots \\ \theta\end{matrix} \qquad \phi_1 \vee \phi_2}{\theta}$$

(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 $(x \colon \mathbb{0})\, t(x) \colon C$ for any type C. Its recursor has the form

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

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 $\neg A :\equiv A \longrightarrow \mathbb{0}$.

5.5 Dependent sum

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

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

(Note that the order now becomes important: We may not declare $y\colon B(x)$ before we have declared $x\colon A$.) 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

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

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 $w\colon \sum(x\colon A)\,B(x)$. The form of the recursor is

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

and yields the elimination rule

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

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

5.6 Dependent product

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

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

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

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

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

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

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

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

and is defined by the recursion

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

If we erase the elements, this becomes the elimination rule

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

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 $x\colon A_1 \times A_2$, the *projections*

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

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* $\mathsf{apply}_f(a)\colon B$ of $f\colon A \longrightarrow B$ to $a\colon A$ by the recursion (on f)

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

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

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

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

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

We will follow common practice and write $f(x)$ (or $f\,x$) 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 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 analogues of) the reflexivity and transitivity of implication:

Theorem. (i) *$A \longrightarrow A$.*

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

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

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

(ii) We have to exhibit a function $F\colon (A \longrightarrow B) \longrightarrow ((B \longrightarrow C) \longrightarrow (A \longrightarrow C))$. Let $f\colon A \longrightarrow B$ and $g\colon B \longrightarrow C$. Then, for $x\colon A$, $g(f(x))\colon C$. Hence,

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

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

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

Exercises

Exercise 5.1. Show the following logical facts.

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

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

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

Exercise 5.2. Let E be the type defined by the single constructor

- For $x\colon E$, $e(x)\colon E$.

This type would result from Nat 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 $f\colon E \longrightarrow \mathbb{0}$ by E-recursion.]

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 in two equivalent ways: 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 that family at once. In the case of based equality, the recursion 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 draws on 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 is often suppressed, as it is determined by the second one, and we 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,

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

and is defined by the based-path recursion

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

$\mathsf{transport}^C(p, x)$ 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 identifications testify that based equality is an equivalence relation.

Reflexivity Let $a \colon A$. Then,

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

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

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

Symmetry Let $p \colon a = b$. Then,

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

6.2 Symmetric equality

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

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

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

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

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

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

The recursor of symmetric equality is

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

is defined by the path recursion

$$\mathsf{rec}^{C}_{=}(z, \mathsf{refl}_x) \quad :\equiv \quad z(x)$$

and corresponds to the alternative elimination rule

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

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 witnessed by the interderivability between the respective recursion principles (equivalently, the interdefinability between $\mathsf{rec}_=$ and transport). One direction is straightforward: Given $(x\colon A)\ z(x)\colon C(x,x)$ and $p\colon a = b$ in A, an element of $C(a,b)$ may be obtained by transporting $z(a)\colon C(a,a)$ 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 $(x\colon A)\ C(x)$ be a family of types over A. We first define functions $f_p\colon C(a) \longrightarrow C(b)$ for $p\colon a = b$ 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

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

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 optional paragraph at the end of the section and the exercises that follow it for some discussion.

6.4 Basic properties of equality

We have already shown that equality is an equivalence relation using based-path recursion. We will now introduce the official operations on paths that constitute the lowest level of the groupoid structure of types. It is customary to use path recursion for this purpose, to avoid any unintended judgmental equalities.

Let A be a type. For any $x\colon A$, we have $\mathsf{refl}_x\colon x = x$. For $p\colon x = y$, an identification $p^{-1}\colon y = x$ is defined by the path recursion

$$\mathsf{refl}_x^{-1} \quad :\equiv \quad \mathsf{refl}_x.$$

Finally, for $p\colon x = y$ and $q\colon y = z$, we first define $t(x,y,p)\colon x = y$ by the path recursion[1]

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

[1] If we set $t(x,y,p) :\equiv p$ directly, we obtain the version of the definition using based-path recursion. This is a standard trick for weakening a judgmental equality into a propositional one; see the exercise on the uniqueness of the definiendum at the end of the next section.

An identification $p \bullet q \colon x = z$ may now be defined by the further path recursion (on q)

$$p \bullet \mathsf{refl}_y \quad :\equiv \quad t(x, y, p).$$

Equality is also respected by families: Let $(x \colon A)\ u(x) \colon B$. Then, we may define $u(p) \colon u(x) = u(y)$ for arbitrary $p \colon x = y$ by means of the path recursion

$$u(\mathsf{refl}_x) \quad :\equiv \quad \mathsf{refl}_{u(x)}.$$

Strictly speaking, this notation is ambiguous. It follows the common practice in category theory of using the same symbol for the action of a functor on objects and on morphisms. In the rare case that we need to differentiate between the two, we will write $\mathsf{ap}_u(p)$ for $u(p)$.

More general forms of recursion

A definition by Nat-recursion of the form

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

may be understood as defining $(n \colon \mathrm{Nat})\ t(n, z)$ for each individual z; indeed, such a definition can be expressed by means of the recursor:

$$t(n, z) \quad :\equiv \quad \mathsf{rec}_{\mathrm{Nat}}(c_0(z), (x \colon \mathrm{Nat}, y)\ c_{\mathsf{s}}(x, y, z), n).$$

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:

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

To formulate such a definition, we would need to supply the entire family $(z)\ t(n, z)$ as an argument to c_{s}:

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

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 definition isn't always 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 (see the exercises); for now, we will be content to state our intention to employ this and other forms of definition as we see fit.

Exercises

Exercise 6.1. 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 6.2. A more general form of path recursion would be as follows: Given

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

the assignment

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

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

Exercise 6.3. 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 cartesian products.

Exercise 6.4. A special case of recursion over the natural numbers is *iteration*:

$$\begin{aligned} t(0) \quad &:\equiv \quad d_0, \\ t(\mathsf{s}(x)) \quad &:\equiv \quad d_{\mathsf{s}}(t(x)), \end{aligned}$$

where $d_0 \colon C$ and $d_{\mathsf{s}}(y) \colon C$ for $y \colon C$. Given a type C together with elements $c_0 \colon C$ and $c_{\mathsf{s}}(x, y) \colon C$ for $x \colon \mathrm{Nat}$ and $y \colon C$ define, using iteration rather than recursion, a family $(x \colon \mathrm{Nat})\, r(x) \colon C$ that satisfies the recurrences determined by c_0 and c_{s} *propositionally*:

$$\begin{aligned} r(0) \quad &= \quad c_0, \\ r(\mathsf{s}(x)) \quad &= \quad c_{\mathsf{s}}(x, r(x)). \end{aligned}$$

7 Induction

Inductive types also support proofs by induction. In the case of Nat, this is the familiar principle

From $\phi(0)$ and $\forall x\, (\phi(x) \supset \phi(\mathsf{s}(x)))$ infer $\forall x\, \phi(x)$.

In the style of propositions-as-types, we would say that from

- evidence for $\phi(0)$ and
- evidence for $\phi(\mathsf{s}(x))$ depending on evidence for $\phi(x)$

we may assemble evidence for $\phi(n)$ for arbitrary n: Nat. Its type-theoretic expression is the following
Principle of Nat-induction: Given a type family $(x\colon \mathrm{Nat})\ C(x)$ together with

- an element c_0 of $C(0)$, and
- a family $(x\colon \mathrm{Nat}, y\colon C(x))\ c_{\mathsf{s}}(x,y)\colon C(\mathsf{s}(x))$,

the assignments

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

serve to define $t(x)\colon C(x)$ for x: Nat.

This principle generalizes the recursion principle of Nat by allowing the type of $t(x)$ to depend on x. The induction principles of the other inductive types and type families are formulated in the same fashion.

The induction principle of Nat may be seen as indirectly asserting that all natural numbers are generated by the constructors 0 and s. A direct expression of this fact, which is already implicit in the inductive definition of Nat, is hindered by the presence of the recursive constructor s. For most types, where there is no such limitation, the corresponding properties may be stated and proved, and shown to be equivalent to their induction principles. Especially for types with a single constructor, these so-called "uniqueness principles" take a particularly simple form, known from the λ-calculus as "propositional η rules". In the case of the type $\mathbb{1}$ with the single constructor $\star\colon \mathbb{1}$, for instance, we have an identification

$$\eta_{\mathbb{1}}(x)\colon \star = x$$

for any $x\colon \mathbb{1}$, obtained by the induction

$$\eta_{\mathbb{1}}(\star) \quad :\equiv \quad \mathsf{refl}_{\star},$$

testifying that $\star$ is the only element of $\mathbb{1}$. What's more, $\mathbb{1}$-induction may be recovered from $\eta_{\mathbb{1}}$ (i.e., $\eta_{\mathbb{1}}$ and the inductor of $\mathbb{1}$ are interdefinable). Namely, given a type family C over $\mathbb{1}$ and an element $c_\star$ of $C(\star)$, the family $(x\colon \mathbb{1})\ t(x)\colon C(x)$ defined by

$$t(x) \quad :\equiv \quad \mathsf{transport}^{C}(\eta_{\mathbb{1}}(x), c_\star)$$

satisfies

$$
\begin{aligned}
t(\star) &\equiv \mathsf{transport}^{C}(\eta_{\mathbb{1}}(\star), c_\star)\\
&\equiv \mathsf{transport}^{C}(\mathsf{refl}_{\star}, c_\star)\\
&\equiv c_\star.
\end{aligned}
$$

Similarly, we have an identification

$$\eta_{A_1\times A_2}(x)\colon \mathsf{pair}(\mathsf{pr}_1(x), \mathsf{pr}_2(x)) = x$$

for any $x\colon A_1 \times A_2$, defined by the recursion

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

Conversely, given

- a type family $(x\colon A_1 \times A_2)\ C(x)$, and
- a family $(x_1\colon A_1, x_2\colon A_2)\ c_{\mathsf{pair}}(x_1, x_2)\colon C(\mathsf{pair}(x_1, x_2))$,

we may define a family $(x\colon A_1 \times A_2)\ t(x)\colon C(x)$ satisfying

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

with the help of $\eta_{A_1 \times A_2}$:

$$t(x) \quad :\equiv \quad \mathsf{transport}^C(\eta_{A_1\times A_2}(x), c_{\mathsf{pair}}(\mathsf{pr}_1(x), \mathsf{pr}_2(x))).$$

Finally, we have an identification

$$\eta_{A\to B}(f)\colon \lambda(\mathsf{apply}_f) = f$$

for any $f\colon A \longrightarrow B$, defined by the recursion

$$\eta_{A\to B}(\lambda(x\colon A)\ b(x)) \quad :\equiv \quad \mathsf{refl}_{\lambda(x:A)\ b(x)}.$$

Once again, definition by induction over $A \longrightarrow B$ can be recovered as follows: Given

- a type family $(f\colon A \longrightarrow B)\ C(f)$, and
- a family $((x\colon A)\ b(x)\colon B)\ c_\lambda(b)\colon C(\lambda(b))$,

a family $(f\colon A \longrightarrow B)\ t(f)\colon C(f)$ satisfying

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

may be defined by

$$t(f) \quad :\equiv \quad \mathsf{transport}^C(\eta_{A\to B}(f), c_\lambda(\mathsf{apply}_f)).$$

Induction over equality comes in two (equivalent) variants.
Based-path induction: Let $a\colon A$. Given a type $C(x, p)$ for each $x\colon A$ and each $p\colon a = x$, and an element c_{refl_a} of $C(a, \mathsf{refl}_a)$, the assignment

$$t(a, \mathsf{refl}_a) \quad :\equiv \quad c_{\mathsf{refl}_a} \tag{1}$$

defines $t(x, p)\colon C(x, p)$ for arbitrary $x\colon A$ and $p\colon a = x$.

One may be tempted to think, judging from the inductive definition of equality, that refl_a is the only element of $a = a$. This is incorrect and is, in fact, refuted by univalence. The culprit is that it is the entire family $(x\colon A)\ a = x$ that is defined by refl_a rather than any specific instance. This is evidenced, among other things, by the family t in (1) being defined on all types $a = x$ at once rather than on any one of them. Hence, the correct way to express this is to implicate pairs: For any $x\colon A$ and $p\colon a = x$, an identification

$$\eta_{a=\_}(x, p)\colon \mathsf{pair}(a, \mathsf{refl}_a) = \mathsf{pair}(x, p)$$

may be defined by the based-path induction

$$\eta_{a=\_}(a, \mathsf{refl}_a) \quad :\equiv \quad \mathsf{refl}_{\mathsf{pair}(a, \mathsf{refl}_a)},$$

showing that every element of $\sum(x\colon A)\ a = x$ is equal to $\mathsf{pair}(a, \mathsf{refl}_a)$.

Path induction: Given types $C(x, y, p)$ for $x, y\colon A$ and $p\colon x = y$, and elements $c_{\mathsf{refl}}(x)$ of $C(x, x, \mathsf{refl}_x)$ for $x\colon A$, the assignment

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

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

Exercises

Exercise 7.1. Show that $\prod(x\colon \mathrm{Bool})\ [(\mathsf{false} = x) + (\mathsf{true} = x)]$.

Exercise 7.2. Show that

$$\prod(x\colon A_1 + A_2)\ \Big[\Big(\sum(x_1 : A_1)\ \mathsf{in}_1(x_1) = x\Big) + \Big(\sum(x_2 : A_2)\ \mathsf{in}_2(x_2) = x\Big)\Big].$$

Exercise 7.3. Formulate the induction principle of $\mathbb{0}$. Show that it is derivable from (and hence equivalent to) its recursion principle.

Exercise 7.4 (Uniqueness of the definiendum)**.** Consider the induction

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

where $c_0\colon C(0)$ and $c_{\mathsf{s}}(x, y)\colon C(\mathsf{s}(x))$ for $x\colon \mathrm{Nat}$ and $y\colon C(x)$. Given $u(x)\colon C(x)$ for $x\colon \mathrm{Nat}$ together with identifications

- $p_0\colon u(0) = c_0$, and
- $p_{\mathsf{s}}(x)\colon u(\mathsf{s}(x)) = c_{\mathsf{s}}(x, u(x))$ for $x\colon \mathrm{Nat}$,

show that $u(x) = t(x)$ for all $x\colon \mathrm{Nat}$.

8 Universes

(...)

8.1 Recursive type families

Recursion principles allow us to define families over types by exploiting the way their elements are generated by the constructors. Nothing is made use of or otherwise assumed about the members of the family being defined. Hence, we may lift the restriction that they are elements of some type (or some types) and consider more general forms of definition by recursion. In this section, we will explore the case of recursively defined type families. A very simple example occurs in the proof of the following

Theorem. $\mathsf{true} \neq \mathsf{false}$.

Proof. The statement asks for a function from $\mathsf{true} = \mathsf{false}$ to $\mathbb{0}$. Consider the type family over Bool defined by

$$\begin{aligned} C(\mathsf{false}) &:\equiv \mathbb{0}, \\ C(\mathsf{true}) &:\equiv \mathbb{1}. \end{aligned}$$

Since $\star\colon \mathbb{1} \equiv C(\mathsf{true})$, it follows that $\mathsf{transport}^C(p, \star)\colon C(\mathsf{false}) \equiv \mathbb{0}$ for any $p\colon \mathsf{true} = \mathsf{false}$. By abstracting p, we obtain the desired function

$$\lambda(p\colon \mathsf{true} = \mathsf{false})\ \mathsf{transport}^T(p, \star)\colon (\mathsf{true} = \mathsf{false}) \longrightarrow \mathbb{0}. \qquad \square$$

Another example is bounded quantification over Nat, defined by

$$\begin{aligned}
\forall(x<0)\,B(x) &\;:\equiv\; \mathbb{1},\\
\forall(x<\mathsf{s}(n))\,B(x) &\;:\equiv\; (\forall(x<n)\,B(x))\times B(n),\\
\exists(x<0)\,B(x) &\;:\equiv\; \mathbb{0},\\
\exists(x<\mathsf{s}(n))\,B(x) &\;:\equiv\; (\exists(x<n)\,B(x))+B(n).
\end{aligned}$$

As a by-product, we obtain the finite types

$$F_n \;:\equiv\; \exists(x<n)\,\mathbb{1}.$$

8.2 Truth predicates

Next, we will reconstruct the language of propositional logic inside type theory. The type Sent of (formal) sentences is defined by

- For $r, s\colon \mathrm{Sent}$, $r \wedge s\colon \mathrm{Sent}$.
- For $r, s\colon \mathrm{Sent}$, $r \supset s\colon \mathrm{Sent}$.
- For $r, s\colon \mathrm{Sent}$, $r \vee s\colon \mathrm{Sent}$.
- $\bot\colon \mathrm{Sent}$.

In order to interpret this language, we need to associate each sentence $s\colon \mathrm{Sent}$ with a proposition, i.e. (by propositions-as-types), a type $T(s)$. This is accomplished by the following recursion.

$$\begin{aligned}
T(r\wedge s) &\;:\equiv\; T(r)\times T(s),\\
T(r\supset s) &\;:\equiv\; T(r)\longrightarrow T(s),\\
T(r\vee s) &\;:\equiv\; T(r)+T(s),\\
T(\bot) &\;:\equiv\; \mathbb{0}.
\end{aligned}$$

The predicate T, which was introduced in this form by A. Tarski, attaches a notion of evidence and, by extension, a meaning to formal sentences. In more traditional philosophical terminology, we might say that $T(s)$ expresses the conditions under which s is true; for this reason, T is called the *truth predicate* of Sent.

To extend our constructions to predicate logic, we need to incorporate individual variables. In first-order languages. variables have domains, called sorts. We will assume given a type Sort of sort symbols and, for each $k\colon \mathrm{Sort}$ a type $T_{\mathrm{Sort}}(k)$ that interprets k and serves as the domain of the respective variables.

To avoid confusion, formal equality of sort k will be written eq_k. The additional constructors for sentences are

- For $k\colon \mathrm{Sort}$ and $a, b\colon T_{\mathrm{Sort}}(k)$, $\mathsf{eq}_k(a,b)\colon \mathrm{Sent}$.
- For $k\colon \mathrm{Sort}$ and family $(x\colon T_{\mathrm{Sort}}(k))\ s(x)\colon \mathrm{Sent}$, $\forall_k s\colon \mathrm{Sent}$.
- For $k\colon \mathrm{Sort}$ and family $(x\colon T_{\mathrm{Sort}}(k))\ s(x)\colon \mathrm{Sent}$, $\exists_k s\colon \mathrm{Sent}$.

The corresponding clauses in the definition of T are

$$\begin{aligned}
T(\mathsf{eq}_k(a,b)) &\;:\equiv\; a =_{T_{\mathrm{Sort}}(k)} b,\\
T(\forall_k s) &\;:\equiv\; \prod(x\colon T_{\mathrm{Sort}}(k))\ T(s(x)),\\
T(\exists_k s) &\;:\equiv\; \sum(x\colon T_{\mathrm{Sort}}(k))\ T(s(x)).
\end{aligned}$$

8.3 Tarski universes

When attempting to extend the constructions presented above to define a type U that reflects a non-trivial fragment of type theory, we run into the following obstacle: This time, there is no distinction between Sort and Sent, which means that T_{Sort} and T merge into one predicate. As a consequence, the constructors of U need to refer back to T. The clause for the dependent product, e.g., now looks like this:

- For any element $a\colon U$ and any family $(x\colon T(a))\ b(x)\colon U$, $\mathsf{dp}_a b\colon U$.

The bottom line is that we have to define the type U and the type family T simultaneously and, along the way, allow the constructors of U to refer to $T(u)$ for previously constructed elements u of U (this is not dissimilar to the constructor s in the definition of Nat being able to refer to previously constructed natural numbers). Such definitions are called *inductive-recursive*, because they combine the inductive definition of a type with the recursive definition of a predicate over that type.

The complete mutual definition of U and T for the fragment $\{+, \mathbb{0}, \mathbb{1}, =, \prod, \sum\}$ has the form

$$
\begin{aligned}
(a, b\colon U)\ \mathsf{sum}(a,b)\colon U &\qquad T(\mathsf{sum}(a,b)) :\equiv T(a) + T(b),\\
\mathsf{zero}\colon U &\qquad T(\mathsf{zero}) :\equiv \mathbb{0},\\
\mathsf{one}\colon U &\qquad T(\mathsf{one}) :\equiv \mathbb{1},\\
(a\colon U, b, c\colon T(a))\ \mathsf{eq}_a(b,c)\colon U &\qquad T(\mathsf{eq}_a(b,c)) :\equiv b =_{T(a)} c,\\
(a\colon U, (x\colon T(a))\ b(x)\colon U)\ \mathsf{dp}_a b\colon U &\qquad T(\mathsf{dp}_a b) :\equiv \textstyle\prod(x\colon T(a))\ T(b(x)),\\
(a\colon U, (x\colon T(a))\ b(x)\colon U)\ \mathsf{ds}_a b\colon U &\qquad T(\mathsf{ds}_a b) :\equiv \textstyle\sum(x\colon T(a))\ T(b(x)).
\end{aligned}
$$

A *universe* (for lack of a better definition) is a pair (U, T) where U is a type (the underlying type, or carrier, of the universe) and T is a type family over U (the truth predicate of the universe).

(...)