M782
Special Topics in Logic: Type Theory
Lecture notes

Nikos Rigas[1]

`Version 2025.11.11`

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

Contents

1 Martin-Löf's Type Theory **2**
1.1 Inductive types and recursion . . . 3
1.2 Logic . . . 9
1.2.1 Product . . . 10
1.2.2 Function type . . . 11
1.2.3 Sum . . . 11
1.2.4 $\mathbb{0}$ . . . 12
1.2.5 $\mathbb{1}$ . . . 13
1.2.6 Dependent sum . . . 13
1.2.7 Dependent product . . . 14
1.2.8 Projections and function application . . . 14
1.3 Equality . . . 17
1.3.1 Based equality . . . 18
1.3.2 Equality . . . 19
1.3.3 Equivalence of the two definitions . . . 19
1.3.4 Basic properties of equality . . . 20
1.3.5 More general forms of recursion . . . 21
1.4 Induction . . . 23
1.5 Universes . . . 29
1.5.1 Recursive type families . . . 29
1.5.2 Course-of-values induction . . . 30
1.5.3 Truth predicates . . . 30
1.5.4 Tarski universes . . . 31

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 write $a : A$ to express the judgment that a is an element of type A. We use uppercase letters for variables ranging over types and lowercase letters for variables ranging over elements of a type.

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. Occasionally, we adopt the notation from category theory of denoting a single index whose range is understood by an underscore, i.e., we write $f(\_)$ for $(()\,x : A)f(x)$.

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, and which has the formation rule

$$\frac{x : A \qquad (y : B)\,z(y) : C}{f(x, z) : D},$$

or the more logically inclined

$$\frac{x : A \qquad \begin{matrix}(y : B)\\ z(y) : C\end{matrix}}{f(x, z) : D}.$$

An important special case is indexed 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

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

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

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

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

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

That is, we instantiate the general principle of recursion to the case where c_0 is 0 and $c_\mathsf{s}(x, y)$ 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:

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

In other words, addition (to m) is defined by the instance of $\mathbb{N}$-recursion where $c_0 \equiv m$ and $c_\mathsf{s}(n, c) \equiv \mathsf{s}(c)$. 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*

$$\frac{c_0 : C \qquad \begin{array}{c}(x : \mathbb{N}, y : C) \\ c_\mathsf{s}(x, y) : C\end{array} \qquad n : \mathbb{N}}{\mathsf{rec}_\mathbb{N}^C(c_0, c_\mathsf{s}, n) : C}$$

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

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

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

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

Exercise 1.1. Define multiplication, exponentiation and the factorial.

Solution. Using the recursor:

$$\begin{aligned} mn &:\equiv \mathsf{rec}_{\mathbb{N}}(0, (x : \mathbb{N}, y : \mathbb{N})\, y + m, n), \\ m^n &:\equiv \mathsf{rec}_{\mathbb{N}}(\mathsf{s}(0), (x : \mathbb{N}, y : \mathbb{N})\, ym, n), \\ n! &:\equiv \mathsf{rec}_{\mathbb{N}}(\mathsf{s}(0), (x : \mathbb{N}, y : \mathbb{N})\, y\,\mathsf{s}(x), n). \end{aligned}$$ □

Another common example is the type List(A) of lists of elements of a type A, defined by the induction scheme

- nil_A : 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 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

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

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 List(A) into a recursor

$$\frac{c_{\mathsf{nil}} : C \qquad \begin{array}{c}(x : A, y : \mathrm{List}(A), z : C) \\ c_{\mathsf{cons}}(x, y, z) : C\end{array} \qquad l : \mathrm{List}(A)}{\mathsf{rec}_{\mathrm{List}(A)}(c_{\mathsf{nil}}, c_{\mathsf{cons}}, l) : C}$$

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

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

the concatenation of two lists as

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

and the sum of a list of natural numbers as

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

Exercise 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 $\mathrm{List}(\mathrm{List}(A))$.

Solution.

$$\mathsf{cat}(L) :\equiv \mathsf{rec}_{\mathrm{List}(\mathrm{List}(A))}(\mathsf{nil}_A, (x : \mathrm{List}(A), y : \mathrm{List}(\mathrm{List}(A)), z : \mathrm{List}(A))\, x + z, L). \quad \square$$

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

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

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

Exercise 1.3. Describe $\mathsf{rec}_{\mathrm{BinTree}}$ and write its defining assignments.

Solution. The recursor of BinTree is the family

$$\frac{c_{\mathsf{triv}} : C \qquad \begin{array}{c}(x_1\mathrm{BinTree}, x_2 : \mathrm{BinTree}, y_1 : C, y_2 : C) \\ c_{\mathsf{join}}(x_1, x_2, y_1, y_2) : C\end{array} \qquad t : \mathrm{BinTree}}{\mathsf{rec}_{\mathrm{BinTree}}(c_{\mathsf{triv}}, c_{\mathsf{join}}, t) : C}$$

defined by

$$\begin{aligned} &\mathsf{rec}_{\mathrm{BinTree}}(c_{\mathsf{triv}}, c_{\mathsf{join}}, \mathsf{triv}) &&:\equiv c_{\mathsf{triv}}, \\ &\mathsf{rec}_{\mathrm{BinTree}}(c_{\mathsf{triv}}, c_{\mathsf{join}}, \mathsf{join}(t_1, t_2)) &&:\equiv \\ &\qquad c_{\mathsf{join}}(t_1, t_2, \mathsf{rec}_{\mathrm{BinTree}}(c_{\mathsf{triv}}, c_{\mathsf{join}}, t_1), \mathsf{rec}_{\mathrm{BinTree}}(c_{\mathsf{triv}}, c_{\mathsf{join}}, t_2)). \end{aligned}$$

$\square$

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

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

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 $\mathsf{join}_A(t)$. In order to extend a family f to $\mathsf{join}_A(t)$, 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

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

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

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

defined by

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

Exercise 1.4. Describe the recursor of Tree(A), and express Tree(g) with its help.

Solution. The recursor of Tree(A) is a family of the form

$$\frac{x : C \qquad \begin{array}{c} ((a : A)\, u(a) : \mathrm{Tree}(A), (a : A)\, v(a) : C) \\ y(u, v) : C \end{array} \qquad z : \mathrm{Tree}(A)}{\mathsf{rec}_{\mathrm{Tree}(A)}(x, y, z) : C}$$

and is defined by the recursion

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

Tree(g) can be expressed as

$$\mathrm{Tree}(g)(t) :\equiv$$
$$\mathsf{rec}_{\mathrm{Tree}(A)}(\mathsf{triv}_B, ((x : A)\, u(x) : \mathrm{Tree}(A), (x : A)\, v(x) : \mathrm{Tree}(B))\ \mathsf{join}_B\,(y : B)\, u(g(y)), t).$$

□

Exercise 1.5. In the definition of List(A), the type A occurs positively. Given a family $(x : A)\, f(x) : B$ define, using the recursion principle or the recursor of List(A), the family $(x : \mathrm{List}(A))\ \mathrm{List}(f)(x) : \mathrm{List}(B)$ which transforms a list of elements of A to the list of their f-images.

Solution. By recursion:

$$\begin{aligned} \mathsf{List}(f)(\mathsf{nil}_A) &:\equiv \mathsf{nil}_B, \\ \mathsf{List}(f)(\mathsf{cons}_A(a,l)) &:\equiv \mathsf{cons}_B(f(a), \mathsf{List}(f)(l)). \end{aligned}$$

By the recursor:

$$\mathsf{List}(f)(l) :\equiv \mathsf{rec}_{\mathsf{List}(A)}(\mathsf{nil}_B, (x : \mathsf{List}(A), y : A, z : \mathsf{List}(B))\ \mathsf{cons}_B(f(y), z), l). \quad \square$$

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 1.6. Define the truth tables of conjunction, implication, and negation.

Solution. There are several possibilities; e.g., by recursion in the first operand:

$$\begin{aligned} \mathsf{and}(x,y) &:\equiv \mathsf{rec}_\mathbb{B}(\mathbb{f}, y, x), \\ \mathsf{ifthen}(x,y) &:\equiv \mathsf{rec}_\mathbb{B}(\mathbb{t}, y, x). \\ \mathsf{not}(x) &:\equiv \mathsf{rec}_\mathbb{B}(\mathbb{t}, \mathbb{f}, x) \quad (\equiv \mathsf{ifthen}(x, \mathbb{f}))\,. \end{aligned} \quad \square$$

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

Solution. One possibility is to define

$$\mathsf{reverse}(l) :\equiv \mathsf{swap}(\mathsf{nil}, l),$$

where

$$\begin{aligned} \mathsf{swap}(k, \mathsf{nil}) &:\equiv k, \\ \mathsf{swap}(k, \mathsf{cons}(a, l)) &:\equiv \mathsf{swap}(\mathsf{cons}(a, k), l). \end{aligned} \qquad \square$$

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

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

of conjunction.

1.2.2 Function type

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

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

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

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

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

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

and is defined by the recursion

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

If we erase the elements, this becomes the elimination rule

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

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

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

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

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

and yields the elimination rule

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

(proof by cases) for disjunction.

1.2.4 $\mathbb{0}$

The type $\mathbb{0}$ corresponds to falsum ($\perp$); 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

$$\frac{\perp}{\theta}$$

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

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

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 $c_{\mathsf{pair}}(x, y) : C$ 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 $w : \sum B$. The form of the recursor is

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

and yields the elimination rule

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

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

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

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

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

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

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

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

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

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

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

of the universal quantifier.

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

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

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 $f : A \longrightarrow B$ to $a : A$ by the recursion (on f)

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

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

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

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:

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

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 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.

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 1.2.1. (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 $F : A \longrightarrow A$; this is served by the identity

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

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

Table 1.1: Natural deduction rules.

(ii) We have to exhibit a function $F : (A \longrightarrow B) \longrightarrow ((B \longrightarrow C) \longrightarrow (A \longrightarrow C))$. Let $f : A \longrightarrow B$ and $g : B \longrightarrow C$. Then, for $x : A$, $g(f(x)) : C$. 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 1.8. Show the following logical facts.

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

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

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

Solution. (i) A function from A to $\neg\neg A$ is

$$\lambda(x : A)\, \lambda(f : \neg A)\, f(x).$$

(ii) Let $f : \neg(A + \neg A) \equiv (A + \neg A) \longrightarrow \mathbb{0}$. Then, $f \circ \mathsf{in}_1 : A \longrightarrow \mathbb{0} \equiv \neg A$ and $f \circ \mathsf{in}_2 : \neg A \longrightarrow \mathbb{0}$. Hence,

$$(f \circ \mathsf{in}_2)(f \circ \mathsf{in}_1) : \mathbb{0}.$$

(iii) A function $F : (A + \neg A) \longrightarrow (\neg\neg A \longrightarrow A)$ can be defined by the $+$-recursion

$$\begin{aligned} F(\mathsf{in}_1(x)) &:\equiv \lambda(g : \neg\neg A)\, x, \\ F(\mathsf{in}_2(f)) &:\equiv \lambda(g : \neg\neg A)\ \mathsf{rec}_{\mathbb{0}}^{A}(g(f)). \end{aligned} \qquad \square$$

Exercise 1.9. Let E be the type defined by the single constructor

- For $x : E$, $\mathsf{e}(x) : E$.

This type would result from $\mathbb{N}$ if we removed 0. 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 $f : E \longrightarrow \mathbb{0}$ by E-recursion.]

Solution. A function $f : E \longrightarrow \mathbb{0}$ may be defined by the recursion

$$f(\mathsf{e}(x)) :\equiv f(x). \qquad \square$$

1.3 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.

1.3.1 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 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 the definition of families of the form $(x : A, p : a = x)\, f(x, p) : C(x)$ 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

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

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

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

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,

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

and is defined by the based-path recursion

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

$\mathsf{transport}^C(p, x)$ 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 $a : A$. Then,

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

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

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

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

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

1.3.2 Equality

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

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

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

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

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

The recursor of equality is

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

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

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

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

1.3.3 Equivalence of the two definitions

Based equality and equality are, essentially, two different descriptions of the same relation. This is witnessed by the interderivability between the respective recursion principles or, equivalently, the interdefinability between $\mathsf{rec}_=$ and 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 $p_* : C(a) \longrightarrow C(b)$ for $p : a = b$ by means of the path recursion

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

We may then define

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

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

1.3.4 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 $x{:}A$, we have $\mathsf{refl}_x{:}x=x$. For $p{:}x=y$, an identification $p^{-1}:y=x$ is defined by the path recursion

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

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

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

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

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

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

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

Strictly speaking, this notation is ambiguous. It follows the established 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 $\mathsf{ap}_f(p)$ for $f(p)$.

Exercise 1.10 (Equal functions are poinwise equal)**.** Let $f,g:A\longrightarrow B$. Show that if $f=g$ then $f(x)=g(x)$ for all $x:A$.

Solution. For $x{:}A$ and $f{:}A\longrightarrow B$ let $t_x(f)=f(x)$. Given an identification $p{:}f=g$ we obtain, by the previous remark, an identification $t_x(p):f(x)=g(x)$. Alternatively, $t_x(p)$ can be defined directly by the path recursion

$$t_x(\mathsf{refl}_f):\equiv\mathsf{refl}_{f(x)}. \qquad \square$$

[1]If we set $f(x,y,p):\equiv p$ 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 theorem 1.4.1.

1.3.5 More general forms of recursion

A definition by $\mathbb{N}$-recursion of the form

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

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

$$f(n,z) \quad :\equiv \quad \mathsf{rec}_\mathbb{N}(c_0(z), (x : \mathbb{N}, y)\, c_\mathsf{s}(x,y,z), n).$$

Sometimes, however, $f(\mathsf{s}(n),z)$ is defined in terms of $f(n,z')$ for (several) arbitrary values of z'. This situation arises, e.g., when we do simultaneous recursion in two arguments:

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

To formulate such a definition, we would need to supply the entire family $f(n,\_)$ as an argument to c_s:

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

The above assignments determine $f(0,z)$ for all z and, once $f(n,z')$ is defined for all z', they determine $f(\mathsf{s}(n),z)$ for all z. Such a definition need not be equivalent to an ordinary definition by recursion over the natural numbers. 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.

Exercise 1.11. Based on the discussion of this 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 $\mathbb{N}$-recursion in the presence of function types.

Exercise 1.12. 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

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

defines $f(x,y,p,z) : C(x,y,z)$ for $x,y : A$, $p : x = y$ and $z : B(x,y)$. Derive based-path recursion from this principle.

Exercise 1.13. A different generalization of $\mathbb{N}$-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.

Exercises

Equality is the basis for defining several other type families. For instance, we can express the *fiber* of a family $(x : A)\, f(x) : B$ over $b : B$ as the type

$$\mathrm{Fib}_f(b) \quad :\equiv \quad \textstyle\sum(x : A)\, b = f(x). \tag{1.1}$$

Conversely, equality can be defined from fibers (this means that we could, in principle, use fibers rather than equality as our basic type family). The purpose of the following exercises is to guide you through the necessary steps.

Exercise 1.14. The type family $(y : B)$ $\mathrm{Fib}_f(y)$ can alternatively be defined as the inductive type family with constructor

- $(x : A)$ $\mathsf{ec}(x) : \mathrm{Fib}_f(f(x))$.

Formulate the recursion principle of Fib_f and describe its recursor.

Solution. The recursion principle of Fib_f reads as follows: Given a type family $(y : B)\, C(y)$ and a family $(x : A)\, c_{\mathsf{ec}}(x) : C(f(x))$, the assignments

$$t(f(x), \mathsf{ec}(x)) :\equiv c_{\mathsf{ec}}(x)$$

for $x : A$ define a family $(y : B, z : \mathrm{Fib}_f(y))\, t(y, z) : C(y)$.

The recursor of Fib_f is formed according to the rule

$$\frac{\begin{array}{c}(x : A)\\ \vdots \\ c_{\mathsf{ec}}(x) : C(f(x))\end{array} \qquad y : B \qquad z : \mathrm{Fib}_f(y)}{\mathsf{rec}^{C}_{\mathrm{Fib}_f}(c_{\mathsf{ec}}, y, z) : C(y)}\ ,$$

and is defined by the recursion

$$\mathsf{rec}^{C}_{\mathrm{Fib}_f}(c_{\mathsf{ec}}, f(x), \mathsf{ec}(x)) :\equiv c_{\mathsf{ec}}(x)$$

for $x : A$. □

For $a : A$, let $(w : \mathbb{1})\, a^{\star}(w) : A$ be the constant family over $\mathbb{1}$ with $a^{\star}(w) \equiv a$.

Exercise 1.15. Consider the following alternative definition of based equality on a type A:

- For $a, b : A$, let
$$a = b :\equiv \mathrm{Fib}_{a^{\star}}(b).$$

- For $a : A$, let
$$\mathsf{refl}_a :\equiv \mathsf{ec}(\star).$$

Show that the recursion principle of based equality is satisfied.

Solution. It is probably easiest to introduce an appropriate family transport and show that it satisfies the defining property of the recursor of based equality: For a type family $(x : A)\, C(x)$, an element $c_{\mathsf{refl}_a} : C(a)$, and an element $p : a = b$, first define the constant family $(w : \mathbb{1})\, c_{\mathsf{ec}}(w) : C(a^\star(w))$ with $c_{\mathsf{ec}}(w) \equiv c_{\mathsf{refl}_a}$, and then let

$$\mathsf{transport}^C(p, c_{\mathsf{refl}_a}) :\equiv \mathsf{rec}^C_{a=\_}(c_{\mathsf{ec}}, b, p).$$

Then, for $a : A$,

$$\begin{aligned}
\mathsf{transport}^C(\mathsf{refl}_a, c_{\mathsf{refl}_a}) &\equiv \mathsf{rec}^C_{a=\_}(c_{\mathsf{ec}}, a, \mathsf{refl}_a)\\
&\equiv \mathsf{rec}^C_{\mathrm{Fib}_{a^\star}}(c_{\mathsf{ec}}, a^\star(\star), \mathsf{ec}(\star))\\
&\equiv c_{\mathsf{ec}}(\star)\\
&\equiv c_{\mathsf{refl}_a}.
\end{aligned}$$

□

1.4 Induction

The presence of type families in the theory allows us to strengthen the recursion principles of the various types. We will take the natural numbers as an example: The sense of a recursive definition

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

is that f can be computed stepwise:

$$\begin{aligned}
f(0) &\equiv c,\\
f(1) &\equiv c_0(f(0)),\\
f(2) &\equiv c_1(f(1)),
\end{aligned}$$

and so on. Put differently, the values of f are obtained by chasing c along the diagram

$$C \xrightarrow{c_0} C \xrightarrow{c_1} C \xrightarrow{c_2} \cdots .$$

The same procedure, however, applies to the more general diagram

$$C_0 \xrightarrow{c_0} C_1 \xrightarrow{c_1} C_2 \xrightarrow{c_2} \cdots ,$$

where the single type C has been replaced by a sequence $C_0, C_1, C_2, \ldots$ of types. We are thus led to the *induction principle* of $\mathbb{N}$: Given

- a type family $(n : \mathbb{N})\, C(n)$,
- an element $c_0 : C(0)$, and
- a family $(n : \mathbb{N}, c : C(n))\, c_{\mathsf{s}}(n, c) : C(\mathsf{s}(n))$,

the assignments

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

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

The generalization of the recursor corresponding to induction is the *inductor*

$$\frac{c_0 : C(0) \qquad \begin{array}{c}(n : \mathbb{N}, c : C(n)) \\ c_{\mathsf{s}}(n, c) : C(\mathsf{s}(n))\end{array} \qquad n : \mathbb{N}}{\mathsf{ind}_{\mathbb{N}}^{C}(c_0, c_{\mathsf{s}}, n) : C(n)}$$

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

$$\begin{array}{lcl}\mathsf{ind}_{\mathbb{N}}^{C}(c_0, c_{\mathsf{s}}, 0) & :\equiv & c_0, \\ \mathsf{ind}_{\mathbb{N}}^{C}(c_0, c_{\mathsf{s}}, \mathsf{s}(n)) & :\equiv & c_{\mathsf{s}}(n, \mathsf{ind}_{\mathbb{N}}^{C}(c_0, c_{\mathsf{s}}, n)).\end{array}$$

The induction principle of $\mathbb{N}$ generalizes the recursion principle of $\mathbb{N}$ by allowing the type of $f(x)$ 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 $\mathrm{List}(A)$, for instance, is formulated as follows: Given

- a type family $(l : \mathrm{List}(A))\, C(l)$,
- an element $c_{\mathsf{nil}} : C(\mathsf{nil})$, and
- a type family $(a : A, l : \mathrm{List}(A), c : C(l))\, c_{\mathsf{cons}}(a, l, c) : C(\mathsf{cons}(a, l))$,

the assignments

$$\begin{array}{lcl} f(\mathsf{nil}) & :\equiv & c_{\mathsf{nil}}, \\ f(\mathsf{cons}(a, l)) & :\equiv & c_{\mathsf{cons}}(a, l, f(l)),\end{array}$$

define a family $(l : \mathrm{List}(A))\, f(l) : C(l)$.

The inductor of $\mathrm{List}(A)$ has the form

$$\frac{c_{\mathsf{nil}} : C(\mathsf{nil}) \qquad \begin{array}{c}(a : A, l : \mathrm{List}(A), c : C(l)) \\ c_{\mathsf{cons}}(a, l, c) : C(\mathsf{cons}(a, l))\end{array} \qquad l : \mathrm{List}(A)}{\mathsf{ind}_{\mathrm{List}(A)}^{C}(c_{\mathsf{nil}}, c_{\mathsf{cons}}, l) : C(l)}$$

and is defined by the recursion

$$\begin{array}{lcl}\mathsf{ind}_{\mathrm{List}(A)}^{C}(c_{\mathsf{nil}}, c_{\mathsf{cons}}, \mathsf{nil}) & :\equiv & c_{\mathsf{nil}}, \\ \mathsf{ind}_{\mathrm{List}(A)}^{C}(c_{\mathsf{nil}}, c_{\mathsf{cons}}, \mathsf{cons}(a, l)) & :\equiv & c_{\mathsf{cons}}(a, l, \mathsf{ind}_{\mathrm{List}(A)}^{C}(c_{\mathsf{nil}}, c_{\mathsf{cons}}, l)).\end{array}$$

Exercise 1.16. Formulate the induction principle and the inductor of $\mathbb{B}$.

Solution. Induction principle: Given

- a type family $(b : \mathbb{B})\, C(b)$,
- an element $c_{\mathbb{f}} : C(\mathbb{f})$, and
- an element $c_{\mathbb{t}} : C(\mathbb{t})$,

the assignments

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

define a family $(b : \mathbb{B})\, f(b) : C(b)$.

Inductor:

$$\frac{c_{\mathbb{f}} : C(\mathbb{f}) \qquad c_{\mathbb{t}} : C(\mathbb{t}) \qquad b : \mathbb{B}}{\mathsf{ind}_{\mathbb{B}}^{C}(c_{\mathbb{f}}, c_{\mathbb{t}}, b) : C(b)},$$

with defining assignments

$$\mathsf{ind}_{\mathbb{B}}^{C}(c_{\mathbb{f}}, c_{\mathbb{t}}, \mathbb{f}) :\equiv c_{\mathbb{f}},$$
$$\mathsf{ind}_{\mathbb{B}}^{C}(c_{\mathbb{f}}, c_{\mathbb{t}}, \mathbb{t}) :\equiv c_{\mathbb{t}}. \qquad \square$$

The induction principle of $\mathbb{N}$ expresses the familiar method of proof by induction: If $\phi(x)$ is a property of natural numbers then, from a proof c_0 of $\phi(0)$ and a proof $c_{\mathsf{s}}(n, c)$ of $\phi(\mathsf{s}(n))$ depending on an arbitrary proof c of $\phi(n)$, we obtain, via the family f defined by induction, a proof $f(n)$ of $\phi(n)$ for an arbitrary natural number n.

As an example of using the induction principle, assume that we want to show that addition of natural numbers is commutative,

$$m + n = n + m,$$

by induction in n. We need to show that $m + 0 = 0 + m$ which, given that $m + 0 \equiv m$, may be written

$$0 + m = m \tag{1.2}$$

and $m + \mathsf{s}(n) = \mathsf{s}(n) + m$ assuming $m + n = n + m$ which, given that $m + \mathsf{s}(n) \equiv \mathsf{s}(m + n)$ and using the induction hypothesis, may be written

$$\mathsf{s}(n) + m = \mathsf{s}(n + m). \tag{1.3}$$

Note that (1.2) and (1.3) are the defining equations of addition inverted; this becomes more evident if we let $m +' n :\equiv n + m$:

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

We just stumbled upon a special case of the following

Theorem 1.4.1 (Uniqueness of the definiendum). *Consider the inductive definition*

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

and assume that we are given a family g *over* $\mathbb{N}$ *such that*

- $\mathrm{g}(0) = c_0$, *and*
- $\mathrm{g}(\mathsf{s}(n)) = c_{\mathsf{s}}(n, \mathrm{g}(n))$ *for* $n : \mathbb{N}$.

Then, $\mathrm{g}(n) = f(n)$ *for all* $n : \mathbb{N}$.

Proof. We are given identifications $p_0 : \mathrm{g}(0) = c_0$ and $p_{\mathsf{s}}(n) : \mathrm{g}(\mathsf{s}(n)) = c_{\mathsf{s}}(n, \mathrm{g}(n))$ for $n : \mathbb{N}$. We will define $p(n) : \mathrm{g}(n) = f(n)$ by induction. The first assignment is obvious,

$$p(0) :\equiv p_0 : \mathrm{g}(0) = c_0 \equiv f(0).$$

For the other, if $p(n) : g(n) = f(n)$ then

$$p_{\mathsf{s}}(n) : g(\mathsf{s}(n)) = c_{\mathsf{s}}(n, g(n))$$

and

$$c_{\mathsf{s}}(n, p(n)) : c_{\mathsf{s}}(n, g(n)) = c_{\mathsf{s}}(n, f(n)) \equiv f(\mathsf{s}(n)),$$

whence we can use the transitivity of equality to obtain an identification

$$p(\mathsf{s}(n)) :\equiv p_{\mathsf{s}}(n) \bullet c_{\mathsf{s}}(n, p(n)) : g(\mathsf{s}(n)) = f(\mathsf{s}(n)).$$

By the induction principle, $p(n) : g(n) = f(n)$ has been defined for all $n : \mathbb{N}$. $\square$

Induction and uniqueness The induction principle of $\mathbb{N}$ may be seen as indirectly asserting that all natural numbers are generated by the constructors $\mathsf{0}$ and s. A direct expression of this fact, which is already implicit in the inductive definition of $\mathbb{N}$, 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 $\star : \mathbb{1}$, for instance, we have an identification

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

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

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

ensuring 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 $(x : \mathbb{1})\, t(x) : C(x)$ defined by

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

satisfies

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

Similarly, we have an identification

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

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

we may define a family $(x : A_1 \times A_2)\, t(x) : C(x)$ 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

$$\eta_{A \longrightarrow B}(f) : \lambda(\mathsf{apply}_f) = f$$

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

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

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

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

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

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

may be defined by

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

Induction over equality comes in two (equivalent) variants.

Based-path induction Let $a : A$. Given a type $C(x, p)$ for each $x : A$ and each $p : a = x$, and an element c_{refl_a} of $C(a, \mathsf{refl}_a)$, the assignment

$$t(a, \mathsf{refl}_a) \quad :\equiv \quad c_{\mathsf{refl}_a} \tag{1.4}$$

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

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 $(x : A)\, a = x$ that is defined by refl_a rather than any specific instance. This is evidenced, among other things, by the family t in (1.4) 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 $x : A$ and $p : a = x$, an identification

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

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 $\sum (x : A)\, a = x$ is equal to $\mathsf{pair}(a, \mathsf{refl}_a)$.

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

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

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

Exercises

Exercise 1.17. Show that $\prod (x : \mathbb{B})\ [(\mathbb{f} = x) + (\mathbb{t} = x)]$.

Solution. Let $(x : \mathbb{B})\, C(x)$ be the type family with $C(x) \equiv (\mathbb{f} = x) + (\mathbb{t} = x)$. We have injections of reflexivities

$$\begin{gathered} \mathsf{in}_1(\mathsf{refl}_{\mathbb{f}}) : C(\mathbb{f}), \\ \mathsf{in}_2(\mathsf{refl}_{\mathbb{t}}) : C(\mathbb{t}). \end{gathered}$$

Hence, we can define a family $(x : \mathbb{B})\, f(x) : C(x)$ by the induction

$$\begin{gathered} f(\mathbb{f}) :\equiv \mathsf{in}_1(\mathsf{refl}_{\mathbb{f}}), \\ f(\mathbb{t}) :\equiv \mathsf{in}_2(\mathsf{refl}_{\mathbb{t}}). \end{gathered}$$

Then, $\lambda(f) : \prod (x : \mathbb{B})\ [(\mathbb{f} = x) + (\mathbb{t} = x)]$. □

Exercise 1.18. Formulate the induction principle of $\mathbb{0}$. Show that it is derivable from (and hence equivalent to) its recursion principle.

Solution. $\mathbb{0}$-induction: For any type family C over $\mathbb{0}$ there is a family $(x : \mathbb{0})\, t(x) : C(x)$.

Such a family can be defined by $\mathbb{0}$-recursion: For $x : \mathbb{0}$, there is a family $(y : \mathbb{0})\, t_x(y) : C(x)$. In particular, $t_x(x) : C(x)$. □

Exercise 1.19. Define inequality $m \leq n$ on $\mathbb{N}$ as the inductive type family with constructors

- For $n : \mathbb{N}$, $\mathsf{le}_0(n) : 0 \leq n$.
- For $e : m \leq n$, $\mathsf{le}_{\mathsf{s}}(m, n, e) : \mathsf{s}(m) \leq \mathsf{s}(n)$.

(i) Formulate the recursion principle of $\leq$.

(ii) Show that $m \leq n$ is logically equivalent to $\sum (x : \mathbb{N})\, x + m = n$.

Solution. (i) Given

- a type family $(m : \mathbb{N}, n : \mathbb{N})\, C(m, n)$,
- a family $(n : \mathbb{N})\, c_0(n) : C(0, n)$, and
- a family $(m : \mathbb{N}, n : \mathbb{N}, e : m \leq n, c : C(m, n))\, c_{\mathsf{s}}(m, n, e, c) : C(\mathsf{s}(m), \mathsf{s}(n))$,

the assignments

$$\begin{gathered} f(0, n, \mathsf{le}_0(n)) :\equiv c_0(n), \\ f(\mathsf{s}(m), \mathsf{s}(n), \mathsf{le}_{\mathsf{s}}(m, n, e)) :\equiv c_{\mathsf{s}}(m, n, e, f(m, n, e)) \end{gathered}$$

define a family $(m : \mathbb{N}, n; \mathbb{N}, e : m \leq n)\, f(m, n, e) : C(m, n)$.

(ii) For the left-to-right implication, define $f_{m,n} : m \leq n \longrightarrow \sum (x : \mathbb{N})\, x + m = n$ by the recursion

$$\begin{aligned} f_{0,n}(\mathsf{le}_0(n)) &:\equiv \mathsf{pair}(n, \mathsf{refl}_n), \\ f_{\mathsf{s}(m),\mathsf{s}(n)}(\mathsf{le}_\mathsf{s}(e)) &:\equiv \mathsf{pair}(\mathsf{pr}_1(f(m,n,e)), \mathsf{s}(\mathsf{pr}_2(f(m,n,e)))). \end{aligned}$$

For the converse, first define $e(x, m) : m \leq x + m$ by the induction

$$\begin{aligned} e(x, 0) &:\equiv \mathsf{le}_0(x), \\ e(x, \mathsf{s}(m)) &:\equiv \mathsf{le}_\mathsf{s}(e(x, m)), \end{aligned}$$

and then define $g_{m,n} : (\sum (x : \mathbb{N})\, x + m = n) \longrightarrow m \leq n$ by the recursion (over the dependent sum)

$$g_{m,n}(\mathsf{pair}(x, p)) :\equiv \mathsf{transport}^{m \leq -}(p, e(x, m)).$$ □

1.5 Universes

At first glance, universes are for types what function types are for families. The elements of a function type encode families of a certain shape, and decoding is effected by means of the apply family; universes encode certain types and decoding is effected by means of a predicate. We will start by examining type families defined by recursion.

1.5.1 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 simple example occurs in the proof of the following

Theorem 1.5.1. $\mathbb{t} \neq \mathbb{f}$.

Proof. The statement asks for a function from $\mathbb{t} = \mathbb{f}$ to $\mathbb{0}$. Consider the type family over $\mathbb{B}$ defined by

$$\begin{aligned} C(\mathbb{f}) &:\equiv \mathbb{0}, \\ C(\mathbb{t}) &:\equiv \mathbb{1}. \end{aligned}$$

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

$$\lambda(p : \mathbb{t} = \mathbb{f})\, \mathsf{transport}^T(p, \star) : (\mathbb{t} = \mathbb{f}) \longrightarrow \mathbb{0}.$$ □

Another example is bounded quantification over $\mathbb{N}$, defined by

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

As a by-product, we obtain the finite types

$$F_n \;\coloneqq\; \sum(x < n)\,\mathbb{1}$$

and strict order

$$m < n \;\coloneqq\; \sum(x < n)\, m = x.$$

1.5.2 Course-of-values induction

In mathematics, we also use the principle of *strong induction* or *course-of-values* induction. This principle is usually derived from the non-constructive *minimum principle*, namely, that a non-empty subset of $\mathbb{N}$ has a minimum. This, however, is completely unnecessary, as course-of-values induction is constructively valid; indeed, it is derivable from ordinary $\mathbb{N}$-induction. The proof is outlined below.

The *course of values* $(n : \mathbb{N})\, \hat{f}(n) : \prod(x < n)\, B(x)$ of a family $(n : \mathbb{N})\, f(n) : B(n)$ is defined by the induction

$$\begin{array}{rcl} \hat{f}(0) & \coloneqq & \star, \\ \hat{f}(\mathsf{s}(n)) & \coloneqq & \mathsf{pair}(\hat{f}(n), f(n)). \end{array}$$

Theorem 1.5.2 (Course-of-values induction). *Let B be a type family over $\mathbb{N}$. Given a family $(n : \mathbb{N}, y : \prod(x < n)\, B(x))\, c(n, y) : B(n)$ there is a family $(n : \mathbb{N})\, f(n) : B(n)$ such that $f(n) = c(n)(\hat{f}(n))$ for all $n : \mathbb{N}$.*

Proof. Define $g(n) : \prod(x < n)\, B(x)$ for $n : \mathbb{N}$ by the induction

$$\begin{array}{c} g(0) \coloneqq \star, \\ g(\mathsf{s}(n)) \coloneqq \mathsf{pair}(g(n), c(n)(g(n))). \end{array}$$

Then, let $f(n) \coloneqq c(n)(g(n))$. $\square$

Course-of-values induction can be formulated as the following proof principle: If, for arbitrary $n : \mathbb{N}$, $\phi(n)$ follows from $\forall(x < n)\, \phi(x)$, then $\phi(n)$ holds for all $n : \mathbb{N}$.

1.5.3 Truth predicates

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

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

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

$$\begin{array}{rcl} T(r \wedge s) & \coloneqq & T(r) \times T(s), \\ T(r \supset s) & \coloneqq & T(r) \longrightarrow T(s), \\ T(r \vee s) & \coloneqq & T(r) + T(s), \\ T(\bot) & \coloneqq & \mathbb{0}. \end{array}$$

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

The corresponding clauses in the definition of T are

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

1.5.4 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 an element $a : U$ and a family $(x : T(a))\, b(x) : U$, $\pi_a(b) : U$.

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 $\mathbb{N}$ 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

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

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).

A popular (and rare) example where induction-recursion can be used outside the context of universe construction is lists (and similar data types) without repetitions. Given a relation of apartness $x \mathrel{\#} y$ for $x, y : A$, the type $\mathrm{DList}(A)$ of lists of distinct elements of A can be defined by the induction-recursion

$$\begin{aligned}
&\mathsf{nil} : \mathrm{DList}(A) \\
&\qquad \mathrm{Fresh}(b, \mathsf{nil}) :\equiv \mathbb{1}, \\
(a : A, l : \mathrm{DList}(A), e : \mathrm{Fresh}(a, l))\ &\mathsf{cons}(a, l, e) : \mathrm{DList}(A) \\
&\mathrm{Fresh}(b, \mathsf{cons}(a, l, e)) :\equiv \mathrm{Fresh}(b, l) \times b \mathrel{\#} a.
\end{aligned}$$

Bibliography

[1] Errett Bishop, *Foundations of Constructive Analysis*, Academic Press, New York, 1967.

[2] Per Martin-Löf, *Intuitionistic Type Theory*, Studies in proof theory, vol. 1, Bibliopolis, Napoli, 1984.

[3] ———, *On the meanings of the logical constants and the justifications of the logical laws*, Nordic journal of philosophical logic **1** (1996), no. 1, 11–60.

[4] Emily Riehl, *Category Theory in Context*, Aurora: Dover Modern Math Originals, Courier Dover Publications, 2017.

[5] Egbert Rijke, *Introduction to Homotopy Type Theory* (2022), https://arxiv.org/abs/2212.11082.

[6] Alfred Tarski, *The Concept of Truth in Formalized Languages*, Logic, Semantics, Metamathematics, 2nd ed. (J. Corcoran, ed.), Hackett, Indianapolis, 1983, pp. 152–278.

[7] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, https://homotopytypetheory.org/book/. Institute for Advanced Study, 2013.