M782 – Special Topics in Logic: Type Theory
Notes on Martin-Löf's type theory

Nikos Rigas

Version 2026.11.12

These notes are based on, and in turn used for, the course on Type Theory taught at the graduate program ALMA. They are at an early stage of development and there is currently no stable link; the latest version can be found on the course's eClass.

This work is licensed under a Creative Commons "Attribution 4.0 International" license.



Contents

1 Inductive types and recursion **3**
- 1.1 Judgmentals of MLTT . . . 3
- 1.2 Inductive types . . . 4
- 1.3 Mutual inductive definitions . . . 10
- 1.4 Inductive type families . . . 11
- 1.5 Exercises . . . 12
- 1.6 Bonus material . . . 13

2 Predicate logic with equality **15**
- 2.1 Propositions as types . . . 15
- 2.2 Predicate logic . . . 17
- 2.3 Equality . . . 25
- 2.4 Exercises . . . 27

3 Induction **28**
- 3.1 Induction . . . 28
- 3.2 Course-of-values induction . . . 31
- 3.3 Induction and uniqueness . . . 31
- 3.4 Exercises . . . 34

4 Universes **35**
- 4.1 Recursive type families . . . 35
- 4.2 Truth predicates . . . 36
- 4.3 Tarski universes . . . 37

Bibliography **38**

Lecture 1

Inductive types and recursion

One feature that differentiates one type theory from another is what kind of types each provides for. MLTT mainly studies inductive types and various generalizations of them. In this lecture, we will discuss inductive types by example and, along the way, introduce and develop the core vocabulary of the language.

1.1 Judgmentals of MLTT

We will present MLTT as a theory of definitions. This is akin to a programming language that is initially empty, so to say, 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. We denote such an indexed family 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,

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

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.

We do not distinguish between the curried and the un-curried notation for a family; e.g., $(x : A)\ (y : B)\, f(x)(y) : C$ and $(x : A, y : B)\, f(x, y) : C$ are alternative notations for the same family.

Indices may themselves be families. For instance, given two types A and B, we have a family

$$((x : A)\, f(x) : B, a : A)\ \mathsf{eval}(f, a) : B$$

where

$$\mathsf{eval}(f, a) :\equiv f(a),$$

whose formation rule is

$$\frac{(x : A)\, f(x) : B \qquad a : A}{\mathsf{eval}(f, a) : B}$$

or the more logically inclined

$$\frac{\begin{array}{l}(x : A)\\ f(x) : B \qquad a : A\end{array}}{\mathsf{eval}(f, a) : B} \ .$$

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 $x : A$. The prensence of type families enables the expression of *dependent* families such as $(x : A)\, f(x) : B(x)$.

1.2 Inductive types

The natural numbers

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

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

$\mathsf{0}$ and s are the postulated ways of constructing natural numbers; they are the *constructors* of $\mathbb{N}$. $\mathsf{0}$ 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

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

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

An inductive definition of a type amounts to listing its constructors, i.e., certain families of elements of the type being defined that signify the (canonical) ways of constructing elements of that type. Each constructor signifies an atomic/irreducible act of construction, and the definition, as a whole, is to be understood as a specification of what the elements of the type look like.

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

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

define $f(n) : C$ for arbitrary $n : \mathbb{N}$; in other words, they define a family $(n : \mathbb{N})\ f(n) : C$.

The rationale is that the definition of $f(n)$ by $\mathbb{N}$-recursion follows the construction of n; namely, $f(\mathsf{0})$ is defined first, then $f(\mathsf{s}(\mathsf{0}))$, then $f(\mathsf{s}(\mathsf{s}(\mathsf{0}))))$, and so on; in particular, the dependence of $f(\mathsf{s}(n))$ on $f(n)$ in the second clause mirrors the dependence of $\mathsf{s}(n)$ on n. The (validity of the) principle follows from and, in a sense, enforces/dictates, the intented reading of the inductive definition as $\mathbb{N}$ being freely generated by its constructors $\mathsf{0}$ and s (ref).

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

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

That is, we instantiate the general principle of recursion to the case where c_{0} is $\mathsf{0}$ 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:

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

In other words, addition (to m) is defined by the instance of $\mathbb{N}$-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}.

We occasionally would rather not give a name to the family being defined by recursion; this happens, e.g., if we want to instantiate it immediately or when it is used to instantiate some other family. This is achieved by turning c_0 and c_{s} into indices of the family being defined. We thus arrive at the *recursor*

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

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

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

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

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

and the sum of two natural numbers can be expressed as

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

Exercise 1.1. Define multiplication, exponentiation, and the factorial. □

Lists

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

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

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

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

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

the assignments

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

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 recursion principle of $\mathrm{List}(A)$ into a recursor

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

defined by the recursion

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

The length of a list can now be defined as

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

the concatenation of two lists as

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

and the sum of a list of natural numbers as

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

Exercise 1.2. 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 $\mathrm{List}(\mathrm{List}(A))$. □

Trees

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 recursion principle: Given

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

the assignments

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

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

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

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

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

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 $\mathsf{join}(t)$. In order to extend a family f to $\mathsf{join}(t)$, assuming we already have $f(t(x)) : C$ for $x : A$, we need a family $((x : A)\, w(x) : \mathrm{Tree}(A), (x : A)\, z(x) : C)\, c_{\mathsf{join}}(w, z) : C$; together with an element $c_{\mathsf{triv}} : C$, we may define f by setting

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

Notation. For families f that, like join, have a single argument that is itself a family, it is customary to omit the outer parentheses and write $f(x : A)\, t(x)$ in place of $f((x : A)\, t(x))$. Furthermore, we will systematically abbreviate named families $(x : A)\, t(x)$ to t. E.g., if we let

$$\mathrm{id}_A(x) :\equiv x$$

for $x : A$, the three expressions

$$\begin{gathered} f(x : A)\, x \\ f(x : A)\ \mathrm{id}_A(x) \\ f(\mathrm{id}_A) \end{gathered}$$

are synonymous. It will also be convenient to use the categorical notation of abbreviating the composite family $(x : A)\, g(f(x))$ by $g \circ f$, i.e., for two families $(x : A)\, f(x) : B$ and $(y : B)\, g(y) : C$ we let

$$(g \circ f)(x) :\equiv g(f(x)).$$

Then, definition of $(y : \mathrm{Tree}(A))\, f(y) : C$ by recursion may be written in the form

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

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

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

defined by

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

The recursor of $\mathrm{Tree}(A)$ has the form

$$\frac{u : C \qquad \left(\dfrac{x : A}{w(x) : \mathrm{Tree}(A)}, \dfrac{x : A}{z(x) : C}\right) \qquad v(w, z) : C \qquad t : \mathrm{Tree}(A)}{\mathsf{rec}_{\mathrm{Tree}(A)}(u, v, t) : C}$$

and is defined by the recursion

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

E.g., $\mathrm{Tree}(g)(t)$ can be obtained as the instance

$$\mathsf{rec}_{\mathrm{Tree}(A)}(\mathsf{triv}, ((x : A)\, w(x) : \mathrm{Tree}(A), (x : A)\, z(x) : \mathrm{Tree}(B))\ \mathsf{join}(w \circ g), t).$$

Exercise 1.4. In the definition of $\mathrm{List}(A)$, the type A occurs positively. Given a family $(x : A)\, g(x) : B$ define, using the recursion principle or the recursor of $\mathrm{List}(A)$, the family $(l : \mathrm{List}(A))\ \mathrm{List}(g)(l) : \mathrm{List}(B)$ that maps the list $[a_1, \ldots, a_n]$ to the list $[g(a_1), \ldots, g(a_n)]$. □

Truth values

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

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

The recursion principle for Bool expresses the fact that a family $(x : \mathrm{Bool})\, f(x)$ is determined by the instances $f(\mathsf{false})$ and $f(\mathsf{true})$. Namely, given two elements c_{false} and c_{true} of a type C, the assignments

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

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

$$(c_{\mathsf{false}} : C, c_{\mathsf{true}} : C, x : \mathrm{Bool})\ \mathsf{rec}_{\mathrm{Bool}}(c_{\mathsf{false}}, c_{\mathsf{true}}, x) : C$$

with defining assignments

$$\begin{aligned} \mathsf{rec}_{\mathrm{Bool}}(c_{\mathsf{false}}, c_{\mathsf{true}}, \mathsf{false}) &:\equiv c_{\mathsf{false}}, \\ \mathsf{rec}_{\mathrm{Bool}}(c_{\mathsf{false}}, c_{\mathsf{true}}, \mathsf{true}) &:\equiv c_{\mathsf{true}}. \end{aligned}$$

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

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

or, alternatively, via the recursor of Bool,

$$\mathsf{or}(x, y) :\equiv \mathsf{rec}_{\mathrm{Bool}}(x, \mathsf{true}, y).$$

Exercise 1.5. Define truth tables for conjunction, implication, and negation. □

Note that, 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, $\mathsf{rec}_{\mathrm{Bool}}(c_{\mathsf{false}}, c_{\mathsf{true}}, b)$ is usually written

$$\texttt{if } b \texttt{ then } c_{\mathsf{true}} \texttt{ else } c_{\mathsf{false}}$$

or a variant thereof.

1.3 Mutual inductive definitions

It is also possible to define several types at once. The difference is that now the constructors of each type can take arguments from the other type(s) as well. This form of inductive definition is called *mutual inductive definition*. For example, we can separate the even and the odd natural numbers:

- $0 : \mathbb{E}$,
- $(x : \mathbb{E})\, \mathsf{o}(x) : \mathbb{O}$,
- $(y : \mathbb{O})\, \mathsf{e}(y) : \mathbb{E}$.

As the two types are defined simultaneously, $(\mathbb{E}, \mathbb{O})$-recursion is a principle for defining families over both of them. Specifically, given two types $C_{\mathbb{E}}$ and $C_{\mathbb{O}}$ together with

- an element $c_0 : C_{\mathbb{E}}$,
- a family $(x : \mathbb{E}, z : C_{\mathbb{E}})\, c_{\mathsf{o}}(x, z) : C_{\mathbb{O}}$, and
- a family $(y : \mathbb{O}, z : C_{\mathbb{O}})\, c_{\mathsf{e}}(y, z) : C_{\mathbb{E}}$,

the assignments

$$\begin{aligned} f_{\mathbb{E}}(0) &:\equiv c_0, \\ f_{\mathbb{O}}(\mathsf{o}(x)) &:\equiv c_{\mathsf{o}}(x, f_{\mathbb{E}}(x)), \\ f_{\mathbb{E}}(\mathsf{e}(y)) &:\equiv c_{\mathsf{e}}(y, f_{\mathbb{O}}(y)) \end{aligned}$$

define two families $(x : \mathbb{E})\, f_{\mathbb{E}}(x) : C_{\mathbb{E}}$ and $(y : \mathbb{O})\, f_{\mathbb{O}}(y) : C_{\mathbb{O}}$.

Exercise 1.6. Formulate the recursor of $(\mathbb{E}, \mathbb{O})$. [Hint: It is two families.] □

As a trivial application of the recursion principle, we can inject both $\mathbb{E}$ and $\mathbb{O}$ into $\mathbb{N}$ as the usual subsets of even resp. odd natural numbers via

$$\begin{aligned} \mathsf{as_nat}(0) &:\equiv 0, \\ \mathsf{as_nat}(\mathsf{o}(x)) &:\equiv \mathsf{s}(\mathsf{as_nat}(x)), \\ \mathsf{as_nat}(\mathsf{e}(y)) &:\equiv \mathsf{s}(\mathsf{as_nat}(y)). \end{aligned}$$

Exercise 1.7. $\mathbb{E}$ is clearly isomorphic to $\mathbb{N}$ (and so is $\mathbb{O}$). We can formalize this by emulating the constructors of $\mathbb{N}$ in $\mathbb{E}$ as follows:

$$\begin{aligned} 0_{\mathbb{E}} &:\equiv 0, \\ \mathsf{s}_{\mathbb{E}}(x) &:\equiv \mathsf{e}(\mathsf{o}(x)). \end{aligned}$$

Show that the type $\mathbb{E}$, with zero and successor defined in this way, satisfies the recursion principle of $\mathbb{N}$: Given a type C together with an element $c_0 : C$ and a family $(x : \mathbb{E}, z : C)\, c_{\mathsf{s}}(x, z) : C$, define a family $(x : \mathbb{E})\, f(x) : C$ such that

$$\begin{aligned} f(0_{\mathbb{E}}) &\equiv c_0, \\ f(\mathsf{s}_{\mathbb{E}}(x)) &\equiv c_{\mathsf{s}}(x, f(x)). \end{aligned}$$

□

1.4 Inductive type families

It is also possible to define an entire type family $(x : A)\, B(x)$ by formulating constructors for its various instances. The point is that now the constructors can change the index, i.e., they can take input from a different instance, so that we generally cannot single out an inductive definition of $B(a)$ for particular $a : A$. As a consequence, we cannot say that the instances $B(a)$ are inductively defined, only that the entire family $(x : A)\, B(x)$ is inductively defined. This has far-reaching consequences that will concern us later.

The standard example is the type of lists of specified length, a.k.a. *vectors*. In the case of lists, we had family len that assigned each list its length. We can, instead, make the length part of the type of each element. The type family $(n : \mathbb{N})\ \mathrm{Vect}_n(A)$ is generated by the constructors

- $\mathsf{null} : \mathrm{Vect}_0(A)$,
- $(n : \mathbb{N}, a : A, v : \mathrm{Vect}_n(A))\ \mathsf{extend}(n, a, v) : \mathrm{Vect}_{\mathsf{s}(n)}(A)$.

An instance such as $\mathrm{Vect}_3(A)$ is not inductively defined: The constructor which builds elements of $\mathrm{Vect}_3(A)$ takes input from a different instance of the family, namely, $\mathrm{Vect}_2(A)$.

In particular, the recursion principle of an inductive type family must refer to the entire type family as well. In the case of vectors, $\mathrm{Vect}(A)$-recursion is a principle for defining families over all instances $\mathrm{Vect}_n(A)$ at once.

Recursion principle of $\mathrm{Vect}(A)$**.** Given a type family $(n : \mathbb{N})\, C(n)$ together with

- an element $c_{\mathsf{null}} : C(0)$, and
- a family $(n : \mathbb{N}, a : A, v : \mathrm{Vect}_n(A), c : C(n))\, c_{\mathsf{extend}}(n, a, v, c) : C(\mathsf{s}(n))$,

the assignments

$$\begin{aligned} f(0, \mathsf{null}) &:\equiv c_{\mathsf{null}}, \\ f(\mathsf{s}(n), \mathsf{extend}(n, a, v)) &:\equiv c_{\mathsf{extend}}(n, a, v, f(v)) \end{aligned}$$

define a family $(n : \mathbb{N}, v : \mathrm{Vect}_n(A))\, f(n, v) : C(n)$. □

The recursor of $\mathrm{Vect}(A)$ has the form

$$\frac{c_{\mathsf{null}} : C(0) \qquad (n : \mathbb{N}, a : A, v : \mathrm{Vect}_n(A), c : C(n)) \\ c_{\mathsf{extend}}(n, a, v, c) : C(\mathsf{s}(n)) \qquad n : \mathbb{N} \qquad v : \mathrm{Vect}_n(A)}{\mathsf{rec}^C_{\mathrm{Vect}(A)}(c_{\mathsf{null}}, c_{\mathsf{extend}}, n, v) : C(n)}$$

and is defined by the recursion

$$\begin{aligned} &\mathsf{rec}^C_{\mathrm{Vect}(A)}(c_{\mathsf{null}}, c_{\mathsf{extend}}, 0, \mathsf{null}) :\equiv c_{\mathsf{null}}, \\ &\mathsf{rec}^C_{\mathrm{Vect}(A)}(c_{\mathsf{null}}, c_{\mathsf{extend}}, \mathsf{s}(n), \mathsf{extend}(n, a, v)) \\ &\qquad :\equiv c_{\mathsf{extend}}(n, a, v, \mathsf{rec}^C_{\mathrm{Vect}(A)}(c_{\mathsf{null}}, c_{\mathsf{extend}}, n, v)). \end{aligned}$$

As a straightforward example, we have an inclusion of each $\mathrm{Vect}_n(A)$ into $\mathrm{List}(A)$, defined by the recursion

$$\begin{aligned} \mathsf{as_list}(\mathsf{null}) &:\equiv \mathsf{nil}, \\ \mathsf{as_list}(\mathsf{extend}(n, a, v)) &:\equiv \mathsf{cons}(a, \mathsf{as_list}(v)), \end{aligned}$$

or via the recursor,

$$\mathsf{as_list}(v) :\equiv \mathsf{rec}_{\mathrm{Vect}(A)}(\mathsf{nil}, (n : \mathbb{N}, a : A, w : \mathrm{Vect}_n(A), l : \mathrm{List}(A))\ \mathsf{cons}(a, l), v).$$

Note that the inclusion *forgets* the lenght of its argument.

Vectors are more suitable than lists when we need to define operations that are stratified according to length. For example, there isn't a single addition between vectors, but one for each length $n : \mathbb{N}$; assuming that A is equipped with a binary operation $+$, vector addition may be defined by the double recursion

$$\begin{aligned} \mathsf{null} + \mathsf{null} &:\equiv \mathsf{null}, \\ \mathsf{extend}(n, a, v) + \mathsf{extend}(n, b, w) &:\equiv \mathsf{extend}(n, a + b, v + w). \end{aligned}$$

While $\mathrm{List}(g)$ can be defined on the whole of $\mathrm{List}(A)$ at once (exercise 1.4), the additional information that $\mathrm{List}(g)(l)$ has the same length as l is welcome:

$$\begin{aligned} \mathrm{Vect}_0(g)(\mathsf{null}) &:\equiv \mathsf{null}, \\ \mathrm{Vect}_{\mathsf{s}(n)}(g)(\mathsf{extend}(n, a, v)) &:\equiv \mathsf{extend}(n, g(a), \mathrm{Vect}_n(g)(v)). \end{aligned}$$

It is also possible to define the entire family $\mathrm{Vect}(A)$ by recursion (ref).

1.5 Exercises

Exercise 1.8. Define the list $\mathsf{reverse}(l)$ that contains the entries of a list l in reverse order.

1.6 Bonus material

More general forms of recursion

The recursion principles we have formulated so far do not account for/accommodate/cover all cases of recursive definitions that are encountered in practice.

Mutual recursion

We occasionally define several sequences by specifying the value at $\mathsf{s}(n)$ of each in terms of the value at n not only of itself but also of the others. This situation occurs with higher-order recursive definitions such as the definition of the Fibonacci sequence,

$$\begin{aligned} a_0 &:\equiv 0, \\ a_1 &:\equiv 1, \\ a_{n+2} &:\equiv a_n + a_{n+1}. \end{aligned} \tag{1.1}$$

The standard treatment amounts to replacing the second-order recursion by a system of first-order recursions[1] with the help of a second sequence b_n,

$$\begin{aligned} a_0 &:\equiv 0, & b_0 &:\equiv 1, \\ a_{n+1} &:\equiv b_n, & b_{n+1} &:\equiv a_n + b_n, \end{aligned}$$

or, using a notation reminiscent of the programming language Python[2],

$$\begin{aligned} a_0, b_0 &:\equiv 0, 1, \\ a_{n+1}, b_{n+1} &:\equiv b_n, a_n + b_n. \end{aligned} \tag{1.2}$$

The idea now is to allow expressions to refer to more than one value (or fewer). If we do this for $\mathsf{rec}_{\mathbb{N}}$ and its first two arguments, we may rewrite (1.2) in the form

$$a_n, b_n :\equiv \mathsf{rec}_{\mathbb{N}}(c_0, c_{\mathsf{s}}, n),$$

[1] This is analogous to rewriting a second-order differential equation

$$H(x, f, f', f'') = 0$$

as a system of first-order differential equations

$$\begin{aligned} H(x, f, g, g') &= 0, \\ g &= f'. \end{aligned}$$

The analogy doesn't end here: We can imitate the methodology of linear differential equations to solve (1.1) and obtain the closed expression

$$a_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}.$$

[2] Compare with the following function that calculates a_n:

```
def fib(n):
    a,b = 0,1
    while n > 0:
        a,b = b,a+b
        n = n-1
    return a
```

where

$$c_0 :\equiv 0, 1,$$
$$c_{\mathsf{s}}(x, y, z) :\equiv z, y + z.$$

Multiple/simultaneous recursion

Earlier we saw various examples of defining families with several arguments by recursion in one of them. However, there are situations where we need to recurse in two or more arguments at the same time. This need arises, for instance, in the following "definition" of the truth function of the equality of the natural numbers:

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

The idea here is to unwind the double/simultaneous recursion into two consecutive recursions. To this effect, we write $\mathsf{is_equal}(m, n)$ as $\mathsf{is_equal}_m(n)$. Using this notation, the first two assignments

$$\begin{aligned}
\mathsf{is_equal}_0(0) &:\equiv \mathsf{true},\\
\mathsf{is_equal}_0(\mathsf{s}(n)) &:\equiv \mathsf{false}
\end{aligned}$$

constitute a recursive definition of $\mathsf{is_equal}_0$. Similarly, once we have defined $\mathsf{is_equal}_m$, the other two assignments

$$\begin{aligned}
\mathsf{is_equal}_{\mathsf{s}(m)}(0) &:\equiv \mathsf{false},\\
\mathsf{is_equal}_{\mathsf{s}(m)}(\mathsf{s}(n)) &:\equiv \mathsf{is_equal}_m(n)
\end{aligned}$$

define $\mathsf{is_equal}_{\mathsf{s}(m)}$. It only remains to put these two together into a recursive definition of $\mathsf{is_equal}_m$ or, alternatively, to invoke the recursor of the natural numbers and let

$$\mathsf{is_equal}_m :\equiv \mathsf{rec}_{\mathbb{N}}(c_0, c_{\mathsf{s}}, m),$$

where

$$\begin{aligned}
c_0 &:\equiv (n : \mathbb{N})\ \mathsf{rec}_{\mathbb{N}}(\mathsf{true}, (x : \mathbb{N}, y : \mathrm{Bool})\, \mathsf{false}, n),\\
c_{\mathsf{s}}(v, w) &:\equiv (n : \mathbb{N})\ \mathsf{rec}_{\mathbb{N}}(\mathsf{false}, (x : \mathbb{N}, y : \mathrm{Bool})\, w(x), n).
\end{aligned}$$

Note that we use here a different form of the recursor than the one we have presented, as both the second argument of c_{s} and $\mathsf{rec}_{\mathbb{N}}(c_0, c_{\mathsf{s}}, m)$ itself are families rather than elements.

Lecture 2

Predicate logic with equality

Type theory is intimately related to logic, especially syllogistic. At the core of this relationship lies the *propositions-as-types* conception/paradigm (a.k.a. the Curry-Howard correspondence), which allows the faithful reconstruction of logic inside type theory, to be detailed in this lecture. The reconstruction enables us to formulate and prove mathematical statements in type theory directly, without having to resort to some external notion of logical consequence.

Propositions-as-types is usually presented as a formal analogy, a correspondence, or an interpretation. It goes much deeper, however; namely, it is rooted on the common meaning-theoretic background shared between logic and type theory. It is thus connected with other questions on analyticity and the theory of meaning. There is no room here for a full discussion; we will rather have to content ourselves with a rough sketch of the main ideas. The interested reader may want to consult [5] for a more detailed exposition.

2.1 Propositions as types

Syllogistic is the study of inference, i.e., the process of drawing conclusions from assumptions. Its goal is to describe, explain and justify this process. The main observation is that inference proceeds in steps, by instantiating and combining *rules of inference* into larger structures called *deductions*. The framework that is relevant to MLTT is natural deduction, where rules consist of any number of premises and one conclusion. A premise can be a sentence, but it can also be an entire deduction, i.e., it can contain interim assumptions to be discharged by the application of the rule of inference as, e.g., in the disjunction elimination rule

$$\frac{\begin{array}{ccc}(\phi_1) & (\phi_2) & \\ \theta & \theta & \phi_1 \vee \phi_2\end{array}}{\theta}\ .$$

This rule expresses case distinction: In order to derive θ from $\phi_1 \vee \phi_2$, we may distinguish cases as to whether ϕ_1 or ϕ_2. Here, ϕ_1 and ϕ_2 are assumptions local to the respective cases; they get discharged (i.e., out of scope) as soon as the

two cases are combined into a single deduction by the disjunction elimination rule.

The problem of justifying arbitrary deductions reduces to that of justifying the rules of inference. With respect to their epistemic status, rules fall into four main classes.

- A rule may be obtainable as a composite of other rules; e.g., the rule

$$\frac{(\phi \wedge \psi) \wedge \theta}{\phi}$$

  can be obtained as

$$\dfrac{\dfrac{(\phi \wedge \psi) \wedge \theta}{\phi \wedge \psi}}{\phi}\ .$$

  Such rules, which are called *derived rules*, are, effectively, abbreviated deductions; the problem of justifying a derived rule reduces to that of justifying the rules used to derive it.

- A rule may be used in a deduction without prior justification. Such rules are *hypothetical*, or *assumed*, rules. If a hypothetical rule later receives a justification, it will cease to be hypothetical, e.g., if it becomes a derived rule, its derivation can then be plugged into the deduction and produce another deduction that does not depend on this rule. In this regard, hypothetical rules function much like assumptions.

- A rule may constitute part of the definition of a sentence. Such rules are in no need of justification, as they (like definitions in general) are devoid of factual content. In natural deduction, these are the *introduction* rules.

- A rule may be a reflection on the definition of a sentence. Whether such a rule is justified by virtue of being such a reflection, or is in no need of justification due to being an implicit part of that definition, is a subtle conceptual point that we will not attempt to tackle here; either way, such rules can be used without further justification. In natural deduction, these are the *elimination* rules.

The last two points need further elaboration. To make sense of them, observe that the main function of inference is that it conveys knowledge of a statement. In light of this observation, a deduction is to be construed as a textual description of a reason (or piece of evidence) for its conclusion; in our context, these semantic counterparts of deductions are customarily called *proofs*. Hence, to know a statement is to possess a proof of that statement.

A deduction may, in addition to a conclusion, also involve assumptions. The knowledge conveyed by such a deduction is *conditional*, i.e., it provides a way for someone who possesses proofs of the assuptions to arrive at a proof of the conclusion. In other words, a deduction describes a construction/method/process for transforming arbitrary proofs of some sentences (the assumptions) to a proof of a sentence (the conclusion).

Then, justifying a rule of inference (and, by extension, formulating any deduction) amounts to relating proofs of its premises, if any, to proofs of its conclusion (if a premise is a deduction rather than a sentence, then a proof of that premise is just a method of the appropriate form once again, recursively). To accomplish this, we need to know the form of the proofs of the various sentences. This is where the rules of the third class come in: They serve to fix the meaning of their conclusion, i.e., to describe the form of the proofs of the conclusion in terms of the proofs of the premises. Hence, these rules function as constructors for the type of proofs of their conclusion. It follows that this type, like all inductive types, satisfies a recursion principle, which also has the form of a rule of inference, in which the sentence features as a premise rather than a conclusion. These are the rules of the fourth class.

2.2 Predicate logic

In the case of predicate logic, reflection and observation have led to a wide agreement regarding the form of proofs of the various compound sentences, known as the *Brouwer-Heyting-Kolmogorov (BHK) interpretation*.

- $\top$ has a proof.
- $\bot$ has no proofs.
- A proof of $\phi_1 \wedge \phi_2$ is a pair consisting of a proof of ϕ_1 and one 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 any proof of ϕ to a proof of ψ.
- A proof of $\exists\,(x : A)\,\phi(x)$ is a pair consisting of an element a of A and a proof of $\phi(a)$.
- A proof of $\forall\,(x : A)\,\phi(x)$ is a construction that transforms any element a of A to a proof of $\phi(a)$.

BHK clarifies the way in which the connectives are used in constructive mathematics. It does this by describing the proofs of each sentence in terms of the proofs of its constituent sentences; i.e., it is recursive (more on this in the lecture on universes). We will examine each logical constant in turn.

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

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

of conjunction.

Recursion principle of $A_1 \times A_2$. Given elements $c_{\mathsf{pair}}(x_1, x_2) : C$ for $x_1 : A_1$ and $x_2 : A_2$, 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

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

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

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

of conjunction.

Sum

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

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

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 of $A_1 + A_2$. Given $c_{\mathsf{inj}_1}(x_1) : C$ for $x_1 : A_1$ and $c_{\mathsf{inj}_2}(x_2) : C$ for $x_2 : A_2$, the assignments

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

define $t(x) : C$ for any $x : A_1 + A_2$. □

The recursor has the form

$$\frac{\begin{array}{ccc}(x_1 : A_1) & & (x_2 : A_2)\\ z_1(x_1) : C & z_2(x_2) : C & x : A_1 + A_2\end{array}}{\mathsf{rec}_{A_1+A_2}(z_1, z_2, x) : C},$$

is defined by the recursion

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

and yields the elimination rule

$$\frac{\begin{array}{ccc}(\phi_1) & (\phi_2) & \\ \theta & \theta & \phi_1 \vee \phi_2\end{array}}{\theta}$$

(proof by cases) for disjunction.

Function type

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

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

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

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

of implication.

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

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

defines $t(f) : C$ for arbitrary $f : A \to B$. □

The recursor of $A \to B$ has the formation rule

$$\frac{\begin{array}{cc}\left(\dfrac{x : A}{y(x) : B}\right) & \\ z(y) : C & f : A \to B\end{array}}{\mathsf{rec}_{A\to B}(z, f) : C}$$

and is defined by the recursion

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

If we erase the elements, this becomes the elimination rule

$$\frac{\begin{array}{cc}\left(\dfrac{\phi}{\psi}\right) & \\ \theta & \phi \supset \psi\end{array}}{\theta}$$

for implication.

0

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

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

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

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

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

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

1

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

- $\star : \mathbf{1}$

that corresponds to the introduction rule

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

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

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

and yields the elimination rule

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

of $\top$.

Predicate logic also involves predicates, such as $x < y$, $\mathrm{Prime}(n)$, and so on. Under propositions-as-types, predicates correspond to families of types, indexed by the respective domains of the arguments. With predicates, we also have quantifiers; these are modelled by the following constructs.

Dependent sum

The product may be generalized by allowing the type of 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

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

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

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

and yields the elimination rule

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

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

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

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

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

$$\frac{\begin{array}{c}\left(\dfrac{x : A}{y(x) : B(x)}\right) \\ z(y) : C \qquad\qquad f : \prod(B)\end{array}}{\mathsf{rec}_{\prod(B)}(z, f) : C}$$

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

$$\frac{\left(\dfrac{x : A}{\phi(x)}\right) \qquad \forall\,(x : A)\,\phi(x)}{\theta \qquad\qquad\qquad\qquad \theta}$$

of the universal quantifier.

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

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

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

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

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

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

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 $\mathbf{1}$ is definable even without recursion:

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

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

Table 2.1: Natural deduction rules.

We will follow common practice and write $f(x)$ in place of $\mathsf{apply}_f(x)$. We mostly use function application and projections in place of recursion over types of 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 2.1.

Bottom line

We have reconstructed intuitionistic first-order logic in 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. (i) $A \to A$.

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

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

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

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

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

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

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

Decidability

The principle that any sentence is either true or false, a.k.a. the *law of excluded middle (LEM)*, is not valid constructively. While it cannot be refuted for any individual sentence, the universal statement that it holds for an entire class of sentences can; such refutations, known as *strong counterexamples*, were put forward for the first time by Brouwer, who noticed that LEM is contradicted by various continuity principles. Surprisingly, univalence also refutes LEM for arbitrary types. In this lecture, we will restrict ourselves to the most rudimentary type-theoretic discussion of the main concepts as an opportunity to practice handling statements and proofs in type theory; they will become useful later.

A type that has elements is said to be *inhabited*. Type A is *decidable* if it either is or is not inhabited; this may be expressed by the type

$$\mathrm{IsDec}(A) :\equiv A + \neg A.$$

The elements of the definiens come either from elements of A (i.e., witnesses that A is inhabited), or functions $A \to \mathbf{0}$ (i.e., witnesses that A is not inhabited). We also let

$$\mathrm{IsStable}(A) :\equiv \neg\neg A \to A$$

and say that A is *stable*.

Exercise 2.1. Show the following for any type A.

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

(ii) $\neg\neg\, \mathrm{IsDec}(A)$.

(iii) $\mathrm{IsStable}(\mathrm{IsDec}(A)) \to \mathrm{IsDec}(A)$.

(iv) $\mathrm{IsDec}(A) \to \mathrm{IsStable}(A)$. $\square$

It follows that any decidable type is stable. Conversely, if all types in a class of types that is closed under IsDec are stable, then all types in that class are decidable. In particular, LEM is equivalent to the principle of stability or double-negation elimination.

Exercise 2.2. Let $A \veebar B :\equiv (A + B) \times \neg(A \times B)$ ("A *xor* B"). Show that if $A \veebar B$ then $\mathrm{IsDec}(A)$. $\square$

2.3 Equality

Equality of a type A is a binary predicate on A. It can be expressed either in the un-carried form $(x, y : A)\, x = y$, as a type family with two indices that are treated symmetrically, or in the curried form $(a : A)\ (x : A)\, a = x$ with the two indices treated asymmetrically, i.e., as a family of type families $(x : A)\, a = x$, one for each $a : A$. The inductive definition of equality is independent of the form, but we obtain two different recursion (and induction) principles that are both useful. We therefore will examine both forms, starting with the latter.

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. When we need to specify the type whose equality we are referring to, we will write $a =^A b$.

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 it is generally more natural and convenient to formulate definitions by recursion directly. One of the few exceptions 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 $\mathsf{transport}$ is the law of the *indiscernibility of identicals*:

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

The following identifications attest to based equality being 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,

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

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

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

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

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: Given a family $(x : A)\, f(x) : B$, 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 $f(p)$.

2.4 Exercises

Exercise 2.3. Let E be the type with the single constructor

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

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

[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 2.

Lecture 3

Induction

3.1 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

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

is that f can be computed stepwise:

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

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 $(n : \mathbb{N})\, C(n)$ together with

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

the assignments

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

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

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

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

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

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

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 other inductive types and type families are formulated in the same fashion. The induction principle of $\mathrm{List}(A)$, for instance, reads 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

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

and is defined by the induction

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

Exercise 3.1. Formulate the induction principle and the inductor of Bool. □

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 $\phi(\mathsf{0})$ 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

$$0 + m = m \tag{3.1}$$

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

$$\mathsf{s}(n) + m = \mathsf{s}(n + m). \tag{3.2}$$

Note that (3.1) and (3.2) 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 2 (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 (dependent) family g over $\mathbb{N}$ such that

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

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

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

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

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 $n : \mathbb{N}$. □

Example 3. Consider the sequence defined by the recursion

$$\begin{aligned} a_0 &:\equiv 0, \\ a_{n+1} &:\equiv 2a_n + 1. \end{aligned}$$

Showing that $a_n = 2^n - 1$ by induction in n is straightforward, but it is perhaps even simpler to apply the previous theorem: Let $b_n = 2^n - 1$ for $n : \mathbb{N}$. Then,

$$\begin{aligned} b_0 &= 0, \\ b_{n+1} &= 2^{n+1} - 1 = 2(\cdot 2^n - 1) + 1 = 2b_n + 1. \end{aligned}$$

By uniqueness of the definiendum we conclude that $a_n = b_n$ for all n. □

Exercise 3.2. Complete the proof of the commutativity of addition (i.e., derive it from theorem 2). □

3.2 Course-of-values induction

In mathematics, we also use the principle of *course-of-values* induction (a.k.a. strong induction): If $\phi(n)$ follows from $\forall\,(x < n)\,\phi(x)$ for all $n : \mathbb{N}$, then $\phi(n)$ holds for all $n : \mathbb{N}$. This principle is usually derived from the non-constructive *minimum principle*, namely, that a non-empty set of natural numbers has a least element. This, however, is completely unnecessary, as course-of-values induction is constructively valid; indeed, it is derivable from ordinary induction. The proof is outlined below.

The *bounded product* $(n : \mathbb{N}) \prod(x < n)\,P(x)$ of a type family $(n : \mathbb{N})\,P(n)$ is the inductive type family with constructors

- $\mathsf{bu}_0 : \prod(x < 0)\,P(x)$,
- $(n : \mathbb{N}, z : \prod(x < n)\,P(x), w : P(n))\ \mathsf{bu}_\mathsf{s}(n, z, w) : \prod(x < \mathsf{s}(n))\,P(x)$.

The *course of values* $(n : \mathbb{N})\ \mathsf{cov}_f(n) : \prod(x < n)\,P(x)$ of a dependent family $(n : \mathbb{N})\,f(n) : P(n)$ is defined by the induction

$$\begin{aligned}\mathsf{cov}_f(0) &:\equiv \mathsf{bu}_0,\\ \mathsf{cov}_f(\mathsf{s}(n)) &:\equiv \mathsf{bu}_\mathsf{s}(n, \mathsf{cov}_f(n), f(n)).\end{aligned}$$

Theorem 4 (Course-of-values induction)**.** *Let P be a type family over $\mathbb{N}$. Given elements $c(n, ind_hyp)$ of $P(n)$ for $n : \mathbb{N}$ and $ind_hyp : \prod(x < n)\,P(x)$, there is a family $(n : \mathbb{N})\,f(n) : P(n)$ satisfying*

$$f(n) = c(n, \mathsf{cov}_f(n)) \tag{3.3}$$

for all $n : \mathbb{N}$.

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

$$\begin{aligned}g(0) &:\equiv \mathsf{bu}_0,\\ g(\mathsf{s}(n)) &:\equiv \mathsf{bu}_\mathsf{s}(n, g(n), c(n, g(n))).\end{aligned}$$

Then, let $f(n) :\equiv c(n, g(n))$. □

Note that equation (3.3) is mainly relevant to course-of-values *recursion*.

Exercise 3.3. Complete the proof of theorem 4. [Hint: It suffices to show that $g(n) = \mathsf{cov}_f(n)$; proceed by induction in n (or use theorem 2).] □

3.3 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 $\mathbf{1}$ with the single constructor $\star : \mathbf{1}$, for instance, we have an identification

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

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

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

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

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

satisfies

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

Similarly, we have an identification

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

for any $x : A_1 \times A_2$, defined by the induction

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

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

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

for any $f : A \to B$, defined by the induction

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

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

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

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

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

may be defined by

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

Exercise 3.4. Formulate the uniqueness principles of $\sum$ and $\prod$ and show that they are equivalent to their induction principles. □

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

$$t(a, \mathsf{refl}_a) \quad :\equiv \quad c_{\mathsf{refl}_a} \tag{3.4}$$

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 conclusion 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 the specific instance $a = a$. This is witnessed, among other things, by the family t in (3.4) 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 $p : x = y$. □

To show, for example, that $(p^{-1})^{-1} = p$ for any $p : x = y$ it suffices, by the principle of path induction, to examine identifications of the form refl_x, for which verification is trivial; we say that properties of (arbitrary) identifications "compute on reflexivity".

3.4 Exercises

Exercise 3.5. Show that $\prod(x : \mathrm{Bool})\ [(\mathsf{false} = x) + (\mathsf{true} = x)]$.

Exercise 3.6. Formulate the induction principle of $A_1 + A_2$.

Exercise 3.7. Formulate the induction principle of $\mathbf{0}$. Show that it is derivable from (and hence equivalent to) its recursion principle.

Lecture 4

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 apply; universes encode certain types and decoding is effected by means of a predicate. These so-called "truth predicates" need to be defined in tandem with the universe whose elements they decode, in a way that combines the inductive definition of a type and the recursive definition of a family over that type into a single act called "induction-recursion". an "inductive-recursive definition".

4.1 Recursive type families

The recursion principle of a type allows us to define families over that type by exploiting the way its elements are generated by the constructors. Nothing is made use of or otherwise assumed about the members of the family being defined. So far, we have only encountered families of elements of some type (or some types), but other cases of recursive definition are possible. In this lecture, we will explore the case of type families defined by recursion, a.k.a. *recursive type families*. A simple example occurs in the proof of the following

Theorem 5. $\mathsf{true} \neq \mathsf{false}$.

Proof. The statement asks for a function $f : \mathsf{true} = \mathsf{false} \to \mathbf{0}$. Consider the type family C over Bool defined by the recursion

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

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

$$\lambda(p : \mathsf{true} = \mathsf{false})\ \mathsf{transport}^C(p, \star) : \mathsf{true} = \mathsf{false} \to \mathbf{0}. \qquad \square$$

Sometimes, an inductive type family can be equivalently described as a recursive type family. For example, bounded quantification over $\mathbb{N}$, which was defined as an inductive type family in section 3.2, may also be defined by the

recursion

$$\begin{aligned}
\textstyle\prod(x < 0)\, B(x) &:\equiv \mathbf{1},\\
\textstyle\prod(x < \mathsf{s}(n))\, B(x) &:\equiv \left[\textstyle\prod(x < n)\, B(x)\right] \times B(n),\\
\textstyle\sum(x < 0)\, B(x) &:\equiv \mathbf{0},\\
\textstyle\sum(x < \mathsf{s}(n))\, B(x) &:\equiv \left[\textstyle\sum(x < n)\, B(x)\right] + B(n).
\end{aligned}$$

As a by-product of bounded existential quantification we obtain the finite types

$$\mathrm{Fin}_n :\equiv \textstyle\sum(x < n)\, \mathbf{1}$$

and strict order

$$m < n :\equiv \textstyle\sum(x < n)\, m = x.$$

The recursive description of a type family is generally easier to work with than the equivalent inductive description (this, however, depends on what one wants to do with that type family).

4.2 Truth predicates

We will start by reconstructing the language of propositional logic inside type theory. The type Sent of (formal) sentences is defined by

- $\top$: Sent.
- $\bot$: Sent.
- $r \wedge s$: Sent for r, s : Sent.
- $r \vee s$: Sent for r, s : Sent.
- $r \supset s$: Sent for r, s : Sent.

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

$$\begin{aligned}
T(\top) &:\equiv \mathbf{1},\\
T(\bot) &:\equiv \mathbf{0},\\
T(r \wedge s) &:\equiv T(r) \times T(s),\\
T(r \vee s) &:\equiv T(r) + T(s),\\
T(r \supset s) &:\equiv T(r) \to T(s).
\end{aligned}$$

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

The corresponding clauses in the definition of T are

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

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

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 $\{\mathbf{0}, \mathbf{1}, +, \sum, \prod, =\}$ has the form

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

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 of induction-recursion outside the context of universe construction is lists (and similar data types) without repetition. Given

a relation of apartness $x \# y$ for $x, y : A$, the type $\mathrm{DList}(A)$ of lists of distinct elements of A can be defined by the induction-recursion

$$\begin{aligned} \mathsf{nil} &: \mathrm{DList}(A) \\ \mathrm{Fresh}(b, \mathsf{nil}) &:\equiv \mathbf{1}, \end{aligned}$$

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

Induction-recursion also has significant applications in metamathematics; specifically, proofs of normalization for (various fragments of) type theory.

Bibliography

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

[2] Martin Hofmann and Thomas Streicher, *The groupoid interpretation of type theory*, Twnety-Five Years of Constructive Type Theory, 1998, pp. 83–111.

[3] Per Martin-Löf, *Constructive Mathematics and Computer Programming*, Logic, Methodology, and Philosophy of Science VI: Proceedings of the Sixth International Congress of Logic, Methodology, and Philosophy of Science, Elsevier North-Holland, 1982, pp. 153–175.

[4] ———, *Intuitionistic Type Theory*, Studies in proof theory, vol. 1, Bibliopolis, Napoli, 1984.

[5] ———, *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.

[6] ———, *100 Years of Zermelo's Axiom of Choice: What was the Problem with It?*, Logicism, Intuitionism, and Formalism (Sten Lindström, Erik Palmgren, Krister Segerberg, and Viggo Stoltenberg-Hansen, eds.), Springer, 2009, pp. 209–219.

[7] Erik Palmgren, *On universes in type theory*, Twnety-Five Years of Constructive Type Theory, 1998, pp. 191–204.

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

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

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

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