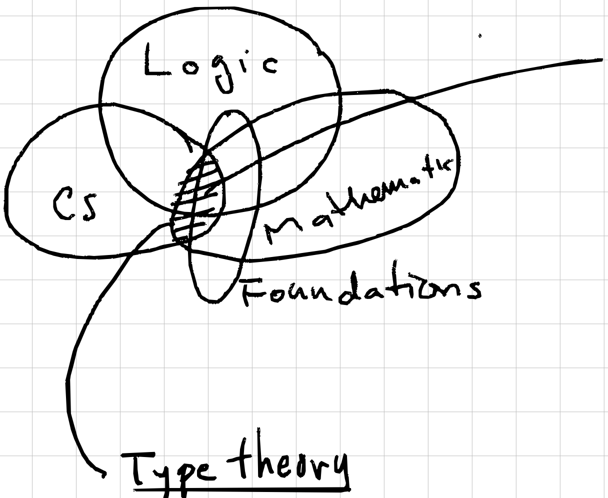
Lectures on Type Theory BICMR May 2025



Proof assistants

Agda
Coq
Lean

} all use a **dependent type theory**

I. Martin-Löf Type Theory (MLTT)

Per Martin-Löf

- Formulate mathematical objects, e.g. $\mathbb{N}$
- Curry-Howard correspondence : Logic internal to MLTT
- Identity Types

II. Homotopy Type Theory (HoTT)

Vladimir Voevodsky

- Types as Spaces (Homotopy Types)
- Univalence Axiom
- Higher Inductive Types

Types which are not sets

$\pi_1(S^1) \cong \mathbb{Z}$

References

- HoTT Book
- HoTTEST Summer School
- https://scripta.io/g/jxxcarlson:mltt

1. Sets and Types

Sets

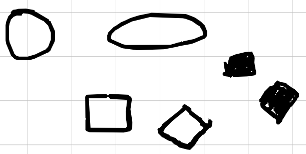
S = collection of elements

$a \in S$, $a \notin S$ ← these are propositions: capable of being proved true or false

element

is a member

set



A = {□, ◇}

B = {■, ◆}

C = {■, □}

We gather a bunch of elements to form a set. We can do this in many ways.

□ ∈ A, □ ∉ B (true)

◆ ∈ A (false)

The Set of Natural Numbers

Von Neumann's natural numbers

$$0_S \in 1_S \in 2_S \in \cdots$$

$$0_S \notin 0_S$$

$$\begin{array}{cccc} \{\ \} & \{\{\ \}\} & \{\{\ \}, \{\{\ \}\}\}, & \ldots \\ \equiv & \equiv & \equiv & \\ 0_S, & 1_S, & 2_S, & , \text{etc.} \end{array}$$

We collect the numbers $0_S, 1_S, \ldots$ to form the set $\mathbb{N}$, Evens, Primes, etc.

$$\mathbb{N} = \{\{\ \}, \{\{\ \}\}, \{\{\ \}, \{\{\ \}\}\}, \ldots\}$$
$$= \{0_S, 1_S, 2_S, \ldots\}$$

$$\text{Evens} = \{0_S, 2_S, 4_S, \ldots\} \qquad \text{Odds} = \{1_S, 3_S, 5_S, \ldots\}$$

$$\text{Primes} = \{2_S, 3_S, 5_S, \ldots\}$$

$$\boxed{2_S \in \mathbb{N}, \quad 2_S \in \text{Evens}}$$

2. MLTT as a FORMAL SYSTEM

Parts of the Formal System

Judgments : things we assert — the facts known to the system

Rules of Inference : rules to derive new judgments from existing ones.

Goncharov's Cherry-Banana Calculus

Symbols: ○, □

Grammar: X ::= ○ | □ | ○X | □X

Terminal Symbols: ○, □ **Nonterminals** X

Production rules: X → ○ , X → □ , X → ○X , X → □X

Productions X → ○X → ○○X → ○○□

Language: Not too interesting: all nonempty strings in ○ and □.

Inference Rules

axiom

$$\frac{}{\bigcirc}$$

prepend

$$\frac{X}{\square\ X}$$

collapse

$$\frac{X\square \quad \square Y}{X\ Y}$$

enclose.

$$\frac{X}{\bigcirc\ X\ \square}$$

Derivation:

$$\dfrac{\dfrac{\dfrac{}{\bigcirc}\,a}{\bigcirc\ \bigcirc\ \square}\,c \qquad \dfrac{\dfrac{}{\bigcirc}\,a}{\square\ \bigcirc}\,p}{\bigcirc\ \bigcirc\ \bigcirc}\,c$$

$$\boxed{\dfrac{P_1\ P_2 \cdots\ P_n}{\text{Conclusion}}}$$

$\bigcirc$ is an axiom. In this case, the initial judgment

$\bigcirc\ \bigcirc\ \bigcirc$ is derivable: it is a new judgment

$\square$ is not derivable

Natural Numbers

$$\frac{}{\mathbb{N}\ \text{type}}\,f \qquad \underbrace{\frac{}{\text{zero} : \mathbb{N}}\,i_1 \qquad \frac{k : \mathbb{N}}{\text{succ}\ k : \mathbb{N}}\,i_2}_{\text{introduction rules}}$$

formation rule

Derivation

$$\cfrac{\cfrac{\cfrac{}{\text{zero} : \mathbb{N}}\,i_1}{\text{succ zero} : \mathbb{N}}\,i_2}{\text{succ}\,(\text{succ zero}) : \mathbb{N}}\,i_2$$

Terminology

zero and succ are constructors for $\mathbb{N}$

Judgments so far: $\text{zero} : \mathbb{N}$,
$\text{succ zero} : \mathbb{N}$,
$\text{succ}\,(\text{succ zero}) : \mathbb{N}$

Aliases $0 :\equiv \text{zero}$, $1 :\equiv \text{succ}\,(\text{zero})$, …

Lambda Calculus

Syntax for λ-terms

1. Variable rule. There is an infinite list of variables $x_1, x_2, \ldots$. These are λ-terms.

2. Abstraction rule. If x is a variable and b is a λ-term, then $\lambda x.b$ is a λ-term, the abstraction of b relative to x.

3. Application rule. If a and b are λ-terms, then so is ab.

Semantics : β-reduction

replace all occurrences of x by a

$$(\lambda x.b)\,a = \cancel{\lambda x}.b[a/x] = b[a/x]$$

body of λ-abstraction

Examples

- $(\lambda x.x)\,a = a$ — $\lambda x.x$ is the identity function
- $\underbrace{(\lambda x.\lambda y.x)}_{}\,1\;2 \xrightarrow{\beta} (\lambda y.1)\,2 \xrightarrow{\beta} 1$ FIRST
- $(\underbrace{\lambda x.\lambda y.y}_{})\,1\;2 \xrightarrow{\beta} (\lambda y.y)\,2 \xrightarrow{\beta} 2$ SECOND

Function Types

$$\frac{A\ \text{type} \qquad B\ \text{type}}{(A \to B)\ \text{type}}\ \text{formation}$$

constructor = →

$$\frac{x : A \vdash b : B}{\lambda x . b : A \to B}\ \text{intro}$$

$$\frac{f : A \to B \qquad a : A}{f(a) : B}\ \text{elim}$$

$$\frac{x : A \vdash b : B \qquad a : A}{(\lambda x . b)\, a = b[a/x] : B}\ \text{comp.}$$

Functions of Several Variables

given types A, B, C:

(1)
$$\frac{B\ \text{type} \qquad C\ \text{type}}{(B \to C)\ \text{type}}$$

(2)
$$\frac{A\ \text{type} \qquad (B \to C)\ \text{type}}{A \to (B \to C)\ \text{type}}$$

Convention: Formation of Types

is right-associative

$$A \to B \to C$$

$$\boxed{\dfrac{A\ \text{type} \qquad \dfrac{B\ \text{type} \qquad C\ \text{type}}{(B \to C)\ \text{type}}}{A \to (B \to C)\ \text{type}}}$$

Evaluation

$$\begin{array}{l} f : A \to B \to C \\ a : A \\ \hline * \quad f(a) : B \to C \quad \leftarrow \\ \hdashline \quad b : B \\ \hline ** \quad f(a)(b) : C \end{array}$$

Alternate notation

$f\,a$ instead of $f(a)$

$f\,a : B \to C$

$(f\,a)\,b : C$ write $f\,a\,b$

Convention

function application is left-associative

$f\,a\,b : C$

The Boolean Type

$$\frac{}{\mathbb{B}\ \text{type}}\ \text{formation}$$

$$\frac{}{\text{true} : \mathbb{B}} \qquad \frac{}{\text{false} : \mathbb{B}}\ \text{intro.}$$

The constructors of $\mathbb{B}$ are <u>true</u> and <u>false</u>

To define $g : \mathbb{B} \to C$:

$$\frac{C\ \text{type} \quad t : C \quad f : C \quad b : \mathbb{B}}{\text{elim}_{\mathbb{B}}(t, f, b) : C}\ \text{elim}$$

$$\left.\begin{array}{c} \dfrac{t : C \quad f : C}{\text{elim}_{\mathbb{B}}(t, f, \text{true}) \equiv t : C} \\[2em] \dfrac{t : C \quad f : C}{\text{elim}_{\mathbb{B}}(t, f, \text{false}) \equiv f : C} \end{array}\right\}\ \text{comp.}$$

$\text{not} : \mathbb{B} \to \mathbb{B}$

$\text{not}\ b = \text{elim}_{\mathbb{B}}(\text{false}, \text{true}, b)$

$$\begin{aligned} \text{not true} &= \text{elim}_{\mathbb{B}}(\text{false}, \text{true}, \text{true}) \\ &= \text{false} \end{aligned}$$

— — — — — — — — — —

$\text{not} = \lambda b.\, \text{elim}_{\mathbb{B}}(\text{false}, \text{true}, b)$

$\text{not false} =$

$(\cancel{\lambda b}.\, \text{elim}_{\mathbb{B}}(\text{false}, \text{true}, b))\ \text{false} =$

$\text{elim}_{\mathbb{B}}(\text{false}, \text{true}, \text{false}) =$

true

— — — — — — — — — —

<u>Pattern-matching defn.</u>

$\text{not} : \mathbb{B} \to \mathbb{B}$
$\text{not true} = \text{false}$
$\text{not false} = \text{true}$

Boolean type — elimination rule

$\text{and} : \mathbb{B} \to \mathbb{B} \to \mathbb{B}$

(1)

$$\left.\begin{array}{llll} \text{and} & \text{true} & \text{true} & = \text{true} \\ \text{and} & \text{true} & \text{false} & = \text{false} \\ \text{and} & \text{false} & \text{true} & = \text{false} \\ \text{and} & \text{false} & \text{false} & = \text{false} \end{array}\right\} \text{definition by pattern-matching}$$

(2)

$$\left.\begin{array}{llll} \text{and} & \_ & \text{false} & = \text{false} \\ \text{and} & x & \text{true} & = x \end{array}\right\} \text{better def by pattern-matching}$$

Definition via the eliminator:

$$\frac{t : \qquad f : C}{\text{elim}_{\mathbb{B}}(t, f, \text{true}) \equiv t : \mathbb{B}}$$

$$\frac{t : C \quad f : C}{\text{elim}_{\mathbb{B}}(t, f, \text{false}) \equiv f : \mathbb{B}}$$

$$\text{and} = \lambda b_1 . \lambda b_2 . \text{elim}_{\mathbb{B}}(b_2, \text{false}, b_1)$$

$$\text{and } x \text{ false} = \text{elim}_{\mathbb{B}}(\text{false}, \text{false}, x) \equiv \text{false}$$

$$\text{and } x \text{ true} = \text{elim}_{\mathbb{B}}(\text{true}, \text{false}, x) \equiv x$$

Note Nondependent pattern-matching definitions can always be translated into eliminator-style definitions. This done automatically by Agda, Coq, and Lean.

Pattern-matching defs are better for humans, eliminator-style Ts better for machines

3. Comparing Sets and Types

Sets vs Types : Comparison

- The numbers $0_S, 1_S, 2_S, \ldots$ exist independently of any set into which they may be gathered.

- Let $\mathcal{P}$ be the set of prime numbers. Then $7_S \in \mathbb{N}$ and $7_S \in \mathcal{P}$.

  But a term cannot be a term of more than one type:

> zero is a term of $\mathbb{N}$ and no other type

or (and (or true Fale) true | false

→ or (and true true) false

→ or true false

→ true

Russell's Paradox

1902, Letter to Frege

$R = \{x \mid x \notin x\}$

$0 \in R, \ 1 \in R, \ \ldots, \ \mathbb{N} \in R, \ \ldots$

Question $R \in R$? Def $\Rightarrow R \notin R$ Contradiction!

$R \notin R$? Def $\Rightarrow R \in R$ Contradiction!

The problem: unrestricted set comprehensions
self-referentiality

Russell's solution ... a theory of types
that restricts the way sets are formed.

Universes

universe of small types

Postulate a hierarchy of universes: $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 \cdots$

Level

Henceforth: write the rules to take into account universes:

$$\frac{}{\mathbb{N} : \mathcal{U}_0}$$

$$\frac{A : \mathcal{U}_n \qquad B : \mathcal{U}_n}{A \to B : \mathcal{U}_n}$$

Universes are cumulative

if $A : \mathcal{U}_i$, then $A : \mathcal{U}_j$ for $j > i$

Populating a universe

$$\frac{}{\mathbb{N} : \mathcal{U}_0}, \qquad \frac{\mathbb{N} : \mathcal{U}_0}{\mathbb{N} \to \mathbb{N} : \mathcal{U}_0}$$

Now $\mathcal{U}_0$ has two terms, $\mathbb{N}$ and $\mathbb{N} \to \mathbb{N}$

Large Types

$f : A \to \mathcal{U}_0$ — a type family where $A : \mathcal{U}_0$

main point

Since $\boxed{\mathcal{U}_0 : \mathcal{U}_1}$, A is a term of $\mathcal{U}_1$ also

$$\frac{A : \mathcal{U}_1 \qquad \mathcal{U}_0 : \mathcal{U}_1}{(A \to \mathcal{U}_0) : \mathcal{U}_1}$$

Formation rule for function type

$$\frac{A : \mathcal{U}_n \qquad B : \mathcal{U}_n}{(A \to B) : \mathcal{U}_n}$$

Conclude: the type of f, $A \to \mathcal{U}_0$ is $\mathcal{U}_1$

Example: $0 =_{\mathbb{N}} n : \mathcal{U}_0$

$f(n) = (0 =_{\mathbb{N}} n)$.

$f : \mathbb{N} \to \mathcal{U}_0$

$\underbrace{\mathbb{N} \to \mathcal{U}_0}_{\text{large type}} : \mathcal{U}_1$

The type of such families is in $\mathcal{U}_1$

4. Functions out of $\mathbb{N}$

(towards the eliminator
for $\mathbb{N}$)

Functions out of ℕ

Official method: Elimination Rule (later)

But we will use pattern-matching for now

Examples (Definition by pattern-matching)

```
double : ℕ → ℕ
double zero = zero
double (succ n) = succ (succ (double n))
```

```
fact : ℕ → ℕ
fact zero = succ zero
fact (succ n) = (succ n) * (fact n)
```

```
add : ℕ → ℕ → ℕ
add zero n = n
add (succ m) n = succ (add m n)
```

General form of function definition:

- Type signature
- One equation for each constructor

All these definitions use recursion.

All computations terminate

The recursion is primitive

Sample computation

double 2

double (succ (succ (zero)))
succ (succ (double (succ (zero))))
succ (succ (succ (succ (double zero))))
succ (succ (succ (succ (zero))))

4

- The number of constructors in the argument of double decreases at each step
- This implies termination
- Computation = Evaluation = Reduction

Bad Definitions

$f : \mathbb{N} \to \mathbb{N}$

f zero = zero --→ **Error**: incomplete pattern matching for f. Missing cases: f(suc x)

$f : \mathbb{N} \to \mathbb{N}$

f zero = zero

f (suc n) = f(suc n) -- **Error**: Termination checking failed for the following functions: f
Problematic calls: f(suc n)

The Addition Operator

$\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$

$\mathrm{zero} + n = n$

$(\mathrm{succ}\ m) + n = \mathrm{succ}\ (m+n)$

An operator is a function of two arguments, where one argument comes before the function symbol, the other comes after.

Exercises:

1) Compute add 2 2
2) Compute 2+2
3) Compute fact 3

5. Primitive Recursion and the Eliminator

ble

PRIMITIVE RECURSION ...

<u>A scheme for setting up recursive computations that always terminate</u>

$\mathbb{N}$

$\text{fact} : \mathbb{N} \to \mathbb{N}$

$\text{fact zero} = \text{succ zero}$ ← base case

$\text{fact} (\text{succ } n) = (\text{succ } n) * (\text{fact } n)$

$= \underbrace{\text{step } n \ (\text{fact } n)}$ — where

$\text{step } n \ v = (\text{succ } n) * v$

Recursion scheme

$$\boxed{\begin{array}{l} f \ \text{zero} = \text{base} \\ f \ (\underline{\text{succ } n}) = \text{step } n \ (f \ \underline{n}) \\ \\ \text{base} : C \\ \text{step} : \mathbb{N} \to C \to C \end{array}}$$

The non-dependent eliminator for $\mathbb{N}$

Data needed to construct $f : \mathbb{N} \to C$

(i) $C : \mathcal{U}$
(ii) $\text{base} : C$
(iii) $\text{step} : \mathbb{N} \to C \to C$

Make up a type that consumes these data and produces $f : \mathbb{N} \to C$.

$$\text{rec}_{\mathbb{N}} : (C : \mathcal{U}) \to C \to (N \to C \to C) \to (\mathbb{N} \to C)$$

the recursor

$$f\ n = \text{rec}_{\mathbb{N}}\ C\ \text{base}\ \text{step}$$

where

$$\text{rec}_{\mathbb{N}}\ C\ \text{base}\ \text{step}\ 0 = \text{base}$$
$$\text{rec}_{\mathbb{N}}\ C\ \text{base}\ \text{step}\ (\text{succ}\ n) = \text{step}\ n\ (\text{rec}_{\mathbb{N}}\ C\ \text{base}\ \text{step}\ n)$$

Example

$[\text{step}\ n\ v = (\text{succ}\ n) * v]$

$$\text{fact} = \text{rec}_{\mathbb{N}}\ N\ \text{zero}\ (\lambda\ n\ v \to (\text{succ}\ n) * v)$$

$\mathbb{N} \to C$

$\text{add} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$
$\text{add zero } n = n$
$\text{add (succ } m)\ n = \text{succ (add } m\ n)$

Then

$C = \mathbb{N} \to \mathbb{N}$
$\text{base} = \text{add zero} = \lambda n.n$
step : maps add m as a function of n to add (succ m) as a function of n

$$\begin{cases} \text{add(succ } m)\ n = \text{succ (add } m\ n) \\ \lambda n.(\text{add (succ } m)\ n) = \end{cases}$$

Want:

$\text{step } n'\ (\text{add } m) = \text{add(succ } m)$

$\text{step } n'\ \underbrace{\lambda n.(\underbrace{(\text{add } m)}_{g}}_{g} = \lambda n.(\text{add (succ } m)\ n)$

$= \lambda n.(\text{succ}((\text{add } m)\ n)) \to \lambda n.(\text{succ } (g\ n))$

$\text{step } n'\ g = \lambda n.(\text{succ}(g\ n))$

Recursion scheme

$f \text{ zero} = \text{base}$
$f\ (\underline{\text{succ } n}) = \text{step } n\ (\underline{f\ n})$

$\text{base} : C$
$\text{step} : \mathbb{N} \to C \to C$

$\lambda n.(\underbrace{(\text{add } m)}_{g}\ n)$
$\to \lambda n.(g\ n)\ n$
$\to g\ n$

⟨ AGDA ⟩

*4

Pattern-matching vs the Eliminator

Pattern-matching definition

$\mathrm{fact} : \mathbb{N} \to \mathbb{N}$ (1a)

$\mathrm{fact}\ \mathrm{zero} = \boxed{\mathrm{succ}\ \mathrm{zero}}$

$\mathrm{fact}\ (\mathrm{succ}\ n) = \underline{(\mathrm{succ}\ n) * (\mathrm{fact}\ n)}$ (2a)

Eliminator Definition

(1b)

$\mathrm{fact} = \mathrm{rec}_{\mathbb{N}}\ \mathbb{N}\ \boxed{(\mathrm{succ}\ \mathrm{zero})}\ (\lambda\, n\, v \to \underline{(\mathrm{succ}\ n) * v}\ )$

(2b)

$$\begin{tikzcd}
\text{Pattern-Matching} \arrow[r, bend left, "(a)"] & \text{Elimination Rule} \arrow[l, bend left, "(b)"]
\end{tikzcd}$$

6. Type Theory $\Rightarrow$ Logic

Propositions as Types

Type theory provides its own logic via the dictionary below (Curry-Howard)

Propositions P $\longleftrightarrow$ Types P

Proofs $\longleftrightarrow$ Terms $p : P$ $\longleftrightarrow$ $\left\{ \begin{array}{l} P \text{ is inhabited} \\ P \text{ has a witness } p \end{array} \right\}$

Implication $P \Rightarrow Q$ $\longleftrightarrow$ Function type $P \to Q$

False $\longleftrightarrow$ Empty type $\bot$

Negation $\neg P$ $\longleftrightarrow$ Function type $P \to \bot$

... ...

A theorem, lemma, etc. is a proposition + proof: $x : P$

Propositional Logic

Example: Modus Ponens

Given $p : P$, $f : P \to Q$

Construct $f\,p : Q$

Example: Contrapositive.

Given: $f : P \to Q$ and $nq : Q \to \bot$

$$P \xrightarrow{f} Q \xrightarrow{nq} \bot$$

Construct: $nq \circ f : P \to \bot$

Curry-Howard Correspondence

	Logic	Type Theory	
1 ✓	Proposition P	Type P	
2 ✓	Proof of P	Term x of P	
3 ✓	$P \Rightarrow Q$	$P \to Q$	Function type
4	$P \land Q$	$P \times Q$	Product Type
5	$P \lor Q$	$P + Q$	Sum type
6 ✓	$\neg P$	$P \to \bot$	
7	$\forall x . P(x)$	$(x:P) \to Q(x)$	Dependent function
8	$\exists x . P(x)$	$\Sigma (x:P) . Q(x)$	Dependant sum

also

$\Pi_{(x:P)} Q(x)$

$\Sigma (x:P), Q(x)$

Curry-Howard Correspondence

Logic	Type Theory	
1 ✓ Proposition P	Type P	
2 ✓ Proof of P	Term x of P	
3 ✓ $P \Rightarrow Q$	$P \to Q$	Function type
4 $P \land Q$	$P \times Q$	Product Type
5 $P \lor Q$	$P + Q$	Sum type
6 ✓ $\neg P$	$P \to \bot$	
7 $\forall x . P(x)$	$(x : P) \to Q(x)$	Dependent function
8 $\exists x . P(x)$	$\Sigma (x:P) . Q(x)$	Dependant sum

also

$\Pi_{(x:P)} Q(x)$

$\Sigma (x:P), Q(x)$

Conjunction (Products)

$$\frac{A : \mathcal{U} \qquad B : \mathcal{U}}{A \times B : \mathcal{U}}$$ formation

$$\frac{a : A \qquad b : B}{(a, b) : A \times B}$$ introduction

$$\frac{p : A \times B}{\mathrm{fst}\ p : A} \qquad \frac{p : A \times B}{\mathrm{snd}(p) : A \times B}$$ elimination

$$\mathrm{fst}(a, b) = a \qquad \mathrm{snd}(a, b) = b$$ computation

Lemma: $A \wedge B \Leftrightarrow B \wedge A$

Lemma: $A \times B \to B \times A$ and $B \times A \to A \times B$

proof $(a, b) = (b, a)$ pattern matching

Dependent Sums: These generalize products

Recall product type $A \times B$
terms are pairs (a,b),
with $a:A$ and $b:B$.

In dependent sums,
pairs (a,b), where
$a:A$, $b:B(a)$, where
$B(a)$ is a type depending on a

Example

$$\text{divides } d\ n = \sum_{(q:\mathbb{N})} (q, \underbrace{d*q=n}_{\text{proof that } d*q=n})$$

[these will be mostly empty]

Model existential quantification

$$\exists\, d, \text{divides } n$$

Disjunction (Coproducts, Sum type)

$$\frac{A:\mathcal{U} \quad B:\mathcal{U}}{A+B:\mathcal{U}}$$ formation

$$\frac{a:A}{\mathrm{inl}\ a : A+B}$$ intro (left-injection)

$$\frac{b:B}{\mathrm{inr}\ b : A+B}$$ intro (right-injection)

Sum Type (Coproduct, Disjunction)

Lemma $A \vee B \Leftrightarrow B \vee A$

Lemma $A+B \to B+A$ and $B+A \to A+$

proof : $A+B \to B+A$

proof e =

$\mathrm{inl}_{A+B}\ a \to \mathrm{inr}_{B+A}\ a$

$\mathrm{inr}_{A+B}\ b \to \mathrm{inl}_{B+A}\ b$

Elimination Rule for Coproducts

To define $f : A + B \to C$

you need $f_L : A \to C$, $f_B : B \to C$.

$$\mathrm{elim}_+(f_L, f_R, \mathrm{inl}\ a) = f_L(a)$$

$$\mathrm{elim}_+(f_L, f_R, \mathrm{inr}\ b) = f_R(b)$$

$$\boxed{\text{Then} \quad f\,x = \mathrm{elim}_+(f_L, f_R, x)}$$

$$\begin{tikzcd}
A \arrow[dr, "\mathrm{inl}\ a"'] \arrow[ddr, bend right=60, "f_L"'] & & B \arrow[dl, "\mathrm{inr}\ b"] \arrow[ddl, bend left=60, "f_R"] \\
& A+B \arrow[d, "f"] & \\
& C &
\end{tikzcd}$$

$$\mathrm{rec}_{A+B} : (A \to C) \to (B \to C) \to (A+B \to C)$$

Examples

- $\begin{cases} f : A+B \to B+A \\ f\ x = \mathrm{elim}_+(\mathrm{inr}_{B+A}, \mathrm{inl}_{B+A}, x) \end{cases}$

- $\begin{cases} f : \mathbb{N}+\mathbb{N} \to \mathbb{N} \\ f\ x = \mathrm{elim}_+(\lambda x. x+1, \lambda x. 2*x, x) \end{cases}$

Elimination Rule for the Empty Type The Principle of Explosion

"From a falsehood, everything."

Empty type: $$\frac{}{\bot : \mathcal{U}}$$ formation

- No introduction rules, no constructors
- No way to construct terms

Elimination rule

$$\frac{t:\bot \qquad C:\mathcal{U}}{\bot\text{-elim}(t,C):C}$$

$$\bot\text{-elim} : (C:\mathcal{U}) \to (\bot \to C)$$

$$\left\{ \begin{array}{l} \text{If there is a term of } \bot, \\ \text{there is a function} \\ \quad f : \bot \to C \end{array} \right\}$$

Corollary If there is a witness to $\bot$, there is a witness to C

William of Soissons, 14th c; Philo the Dialectician "material conditional" c. 300 BCE

Law of the Excluded Middle (LEM) Aristotle 4C BCE

lem: $(P : \mathcal{U}) \to P + (P \to \bot)$

For all $P : \mathcal{U}$, there is a term $t : P + (P \to \bot)$

Case (1): $\mathrm{inl}\ t : P$ -- P is proved

Case (2) $\mathrm{inr}\ t : P \to \bot$ -- $\neg P$ is proved

Constructive logic + LEM ⇒ universal mechanism for either proving or refuting P for all P (!!)

Notes

BHK: Brouwer, Heyting, Kolmogorov (Intuitionistic Logic)

Heyting: LEM is not derivable in Intuitionistic Logic

Extended Example: All natural numbers are either even or odd.

Plan

(1) Define a type family Even n

(2) Define a type family Odd n

(3) The proposition is

$$\boxed{(n : \mathbb{N}) \to \text{Even } n + \text{Odd } n}$$

$$\boxed{\forall n, n \text{ is even or odd}}$$

(4) Construct a witness to the proposition

Note: universal quantification is encoded by a dependent function

$$\forall n, P(n) \rightsquigarrow (n : \mathbb{N}) \to P\, n$$

$f : (n : \mathbb{N}) \to P\, n$ ... For every $n : \mathbb{N}$ $f n : P n$

(1a) Define Even as a type

Even is a type family with constructors evenZero and evenSS

form. $\text{Even} : \mathbb{N} \to \mathcal{U}$

intro. $\text{evenZero} : \text{Even } 0$

intro. $\text{evenSS} : (n : \mathbb{N}) \to \text{Even}(n) \to \text{Even}(\text{succ}(\text{succ } n))$

Terms are constructed recursively

$\text{even evenZero} : \text{Even } 0$

$\text{evenSS}(\text{even evenZero}) : \text{Even } 2$

$\text{evenSS}(\text{evenSS}(\text{even evenZero})) : \text{Even } 4$

...

The type Even n exists for all $n : \mathbb{N}$. The types Even 0, Even 2, Even 4,... are inhabited. The types Even 1, Even 3, Even 5,... are empty

---

(1b) $\text{Odd} : \mathbb{N} \to \mathcal{U}$

$\text{oddOne} : \text{Odd } 1$

$\text{oddSS} : (n : \mathbb{N}) \to \text{Odd } n \to \text{Odd}(\text{succ}(\text{succ } n))$

...

Witness to (proof of) the theorem

$\text{evenOrOdd} : (n : \mathbb{N}) \to \text{Even}\ n + \text{Odd}\ n$

$\text{evenOrOdd}\ 0 = \text{inl evenZero}$

$\text{evenOrOdd}\ 1 = \text{inr oddOne}$

Now construct the remaining terms recursively:

- if we have a witness $e : \text{Even}\ n$, we produce $\text{inl}\ (\text{evenSS}\ e) : \text{Even}\ (\text{succ}\ (\text{succ}\ n))$
- if we have a witness $o : \text{Odd}\ n$, we produce $\text{inr}\ (\text{oddSS}\ o) : \text{Odd}\ (\text{succ}\ (\text{succ}\ n))$

Let motive = constant family $C(x) = \text{Even}(\text{succ}\ (\text{succ}\ n)) + \text{Odd}(\text{succ}(\text{succ}\ n))$

$$
\begin{aligned}
&\text{evenOrOdd}\ (\text{succ}\ (\text{succ}\ n)) = \\
&\quad \text{elim}_+\ (\lambda x.\ \text{Even}(\text{succ}(\text{succ}\ n))) + \text{Odd}\ (\text{succ}(\text{succ}\ n))), \\
&\qquad \lambda e.\ \text{inl}\ (\text{evenSS}\ e), \\
&\qquad \lambda o.\ \text{inr}\ (\text{oddSS}\ o), \\
&\qquad \text{evenOrOdd}\ n\ )
\end{aligned}
$$

⟨SLIDE⟩

Non-Dependent Eliminator

$$\mathrm{elim}_+ : (C : \mathcal{U}) \to (a : A \to C) \to (b : B \to C) \to A+B \to C$$

$$\mathrm{elim}_+(C, f_L, f_R, \mathrm{inl}\, a) = f_L\, a$$

$$\mathrm{elim}_+(C, f_L, f_R, \mathrm{inr}\, b) = f_R\, b$$

$$f\, x = \mathrm{elim}_+(C, f_L, f_R, x)$$

Dependent Eliminator

$$\mathrm{elim}_+ : (C : A+B \to \mathcal{U}) \to ((a : A) \to C(\mathrm{inl}\, a)) \to ((b : B) \to C(\mathrm{inr}\, b)) \to ((x : A+B) \to C(x))$$

$$\mathrm{elim}_+(C, f_L, f_R, \mathrm{inl}\, a) = f_L(\mathrm{inl}\, a)$$

$$\mathrm{elim}_+(C, f_L, f_R, \mathrm{inr}\, b) = f_R(\mathrm{inr}\, a)$$

$$f\, x = \mathrm{elim}_+(C, f_L, f_R, x)$$

Universal Quantification

$\forall n, P(n)$ – universally quantified proposition

How can we formulate this in MLTT?

$$f : n \mapsto \text{proof of } \underbrace{P(n)}_{\text{a type that depends on } n}$$

Therefore

f = <u>dependent function</u> from $\mathbb{N}$ to $P(n)$

$$f : (n : \mathbb{N}) \to P(n)$$

To prove $\forall n : \mathbb{N}, P(n) = Q(n)$ we need <u>identity types</u>.