Martin-Löf Type Theory

Part 1: Simple types

Andreas Abel

DAT350 / DIT235 *Types for programs and proofs*
Chalmers University of Technology and University of Gothenburg

September 2026

Per Martin-Löf

- Born 1942. Swedish logician and philosopher.
- *Intuitionistic Type Theory* (1972–1984): a foundation for constructive mathematics.
- A logic, a computation model and a programming language in one formalism.

A type theory consists of judgment forms and inference rules.
Both parts of this course are instances of that.

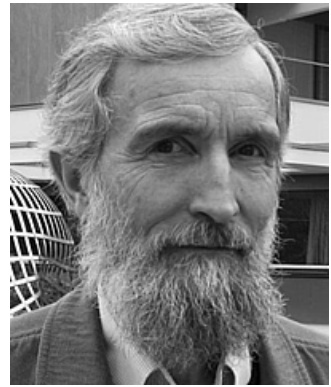

Photo: Institute for Advanced Study

What is a type theory?

1. A set of judgment forms — what can be asserted.
2. A set of inference rules — how assertions are derived.

Martin-Löf justifies the rules by *meaning explanations*.
They can be made formal by constructing a *model*.

Part 1 covers four judgment forms and rules for $\to$, $\times$, $\top$, Bool, $+$, $\bot$, $\mathbb{N}$.

Roadmap

Part 1 — simple types

The four judgments

The five principles

Function space $A \to B$

Product $A \times B$

Unit $\top$

Booleans Bool

Disjoint sum $A_1 + A_2$

Empty type $\bot$

Natural numbers $\mathbb{N}$

Part 2 — dependent types

The conversion rule

Dependent function Π

Dependent sum Σ

Dependent eliminators

Identity type Id

subst and J

Axiom K

Universes

Type : Type and System F

The four judgments

The four judgment forms

	Judgment	Reading
1	$\Gamma \vdash A$ type	A is a well-formed type
2	$\Gamma \vdash A = B$ type	A and B are equal types
3	$\Gamma \vdash t : A$	t is a well-formed term of type A
4	$\Gamma \vdash t = u : A$	t and u are equal terms at type A

- *Intensional* Type Theory: judgements are decidable.
- Judgments 2 and 4 are judgmental (definitional) equality (decided by the type checker, not proven by the user).
- A *proposition* $a = b$ is a different thing. That is the Id type, in Part 2.

Contexts

A fifth, auxiliary judgment: $\vdash \Gamma$ ctx.

$$\frac{}{\vdash \cdot \text{ ctx}} \qquad \frac{\vdash \Gamma \text{ ctx} \qquad \Gamma \vdash A \text{ type}}{\vdash \Gamma, x{:}A \text{ ctx}} \qquad \frac{\Gamma \vdash A \text{ type}}{\Gamma, x{:}A \vdash x : A}$$

Each judgment presupposes the ones before it:

$\Gamma \vdash A$ type	presupposes $\vdash \Gamma$ ctx
$\Gamma \vdash t : A$	presupposes $\Gamma \vdash A$ type
$\Gamma \vdash A = B$ type	presupposes $\Gamma \vdash A$ type and $\Gamma \vdash B$ type
$\Gamma \vdash t = u : A$	presupposes $\Gamma \vdash t : A$ and $\Gamma \vdash u : A$

Presupposition is what makes the judgments interdependent.

Structural rules

Weakening and substitution, for any judgment $\mathcal{J}$:

$$\frac{\Gamma, \Gamma' \vdash \mathcal{J} \qquad \Gamma \vdash A \text{ type}}{\Gamma, x{:}A, \Gamma' \vdash \mathcal{J}} \qquad \frac{\Gamma, x{:}A, \Gamma' \vdash \mathcal{J} \qquad \Gamma \vdash u : A}{\Gamma, \Gamma'[u/x] \vdash \mathcal{J}[u/x]}$$

Both equality judgments are equivalence relations:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A} \qquad \frac{\Gamma \vdash t = u : A}{\Gamma \vdash u = t : A} \qquad \frac{\Gamma \vdash t = u : A \qquad \Gamma \vdash u = v : A}{\Gamma \vdash t = v : A}$$

and likewise for $\Gamma \vdash A = B$ type.

Congruence

Every term former respects equality. For application:

$$\frac{\Gamma \vdash t = t' : A \to B \qquad \Gamma \vdash u = u' : A}{\Gamma \vdash t\,u = t'\,u' : B}$$

Every formation and introduction rule comes with a congruence rule of this shape. They are omitted from the following slides.

The conversion rule

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = B \text{ type}}{\Gamma \vdash t : B}\ \text{CONV}$$

This is the only rule connecting type equality (2) to typing (3).

In Part 1 it has no effect: type equality will be syntactic identity, so no two distinct types are ever considered equal.

In Part 2, types may contain terms, and the rule becomes essential.

The five principles

Types à la Martin-Löf

Every type former is given by five kinds of rules — the five principles:

1. **Formation** — when is the type well-formed?
2. **Introduction** — how are its elements constructed?
3. **Elimination** — how is the information in them extracted?
4. **Computation** (β) — how does elimination act on introduction?
5. **Extensionality** (η) — elimination is complete.

A single principle may give rise to several rules. Principles 4 and 5 are the two halves of a balance condition on principles 2 and 3.

Harmony: local soundness and local completeness

Introduction and elimination rules must be in harmony (Pfenning and Davies).

Principle	Meaning	Rule
Local soundness	Introducing a connective and immediately eliminating it yields no new information; the detour can be removed.	β: $\mathsf{fst}\langle a, b\rangle = a$
Local completeness	Decomposing an element and rebuilding it loses no information.	η: $p = \langle \mathsf{fst}\, p, \mathsf{snd}\, p\rangle$

If local soundness fails, the elimination rules are *too strong*.
If local completeness fails, the elimination rules are *too weak*.

Two flavours

Negative $\to$, $\times$, $\top$

- Elimination projects.
- Several eliminations, one introduction.
- η is a plain equation

$$t = \langle \mathsf{fst}\ t, \mathsf{snd}\ t \rangle$$

- Rules indexed by fields.

Positive Bool, $+$, $\bot$, $\mathbb{N}$

- Elimination is case analysis.
- Several introductions, one elimination.
- η is stated for a term e with a free variable z.
- Rules indexed by constructors.

Part 1 covers the negative types first, then the positive ones.

Agda

Agda

Agda is an implementation of rules of this kind.

```
module lecture3 where
```

- Set is the type of (small) types — the universe. It is explained in Part 2.
- data declares a *positive* type: a list of constructors.
- record declares a *negative* type: a list of fields.
- Definitions are given by pattern matching (f (suc n) = ...) or by copattern (f .fst = ...), which says what each elimination returns.

Agda checks typing (judgments 1 and 3), and decides equality (judgments 2 and 4).

A probe for definitional equality

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

The rules of this type are derived in Part 2. Here one fact is used:

refl typechecks at $x \equiv y$ if and only if $\Gamma \vdash x = y : A$

So refl can be used to ask Agda whether two terms are judgmentally equal.

Function space $A \to B$

$A \to B$: formation and introduction

1. Formation

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma \vdash A \to B \text{ type}}$$

2. Introduction

$$\frac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

The premise extends the context: this is a *hypothetical* judgment.

Curry–Howard: implication introduction. To prove $A \Rightarrow B$, assume A and prove B.

$A \to B$: elimination and computation

3. Elimination

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}$$

4. Computation (β, local soundness)

$$\frac{\Gamma, x{:}A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x.t)\,u = t[u/x] : B}$$

$t[u/x]$ is *substitution*: a meta-level operation defined by recursion on t, renaming bound variables to avoid capture.

$A \to B$: extensionality

5. Extensionality (η, local completeness)

$$\frac{\Gamma \vdash t : A \to B}{\Gamma \vdash t = \lambda x.\, t\, x : A \to B}$$

Nothing about t is invisible to application: applying t to a fresh variable and abstracting again recovers t.

An equivalent hypothetical formulation:

$$\frac{\Gamma, x{:}A \vdash t\, x = u\, x : B}{\Gamma \vdash t = u : A \to B}$$

This is *judgmental* equality. The function extensionality axiom is a statement about the Id type and is discussed in Part 2.

$A \to B$ in Agda

Function types are primitive.

```
id : {A : Set} → A → A
id = λ x → x

apply : {A B : Set} → (A → B) → A → B
apply f a = f a
```

$A \to B$ in Agda

Function types are primitive.

```
id : {A : Set} → A → A
id = λ x → x

apply : {A B : Set} → (A → B) → A → B
apply f a = f a
```

η holds judgmentally, and the probe confirms it:

```
eta→ : {A B : Set} (f : A → B) → f ≡ (λ x → f x)
eta→ f = refl
```

Cartesian product $A \times B$

$A \times B$: all five principles

1. $$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma \vdash A \times B \text{ type}}$$

2. $$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \times B}$$

3. $$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathsf{fst}\, t : A} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathsf{snd}\, t : B}$$

4. $$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\begin{array}{l}\Gamma \vdash \mathsf{fst}\langle t, u \rangle = t : A \\ \Gamma \vdash \mathsf{snd}\langle t, u \rangle = u : B\end{array}}$$

5. $$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash t = \langle \mathsf{fst}\, t, \mathsf{snd}\, t \rangle : A \times B}$$ (surjective pairing)

$A \times B$ in Agda, first alternative: data

```
module DataProduct where

  data _×_ (A B : Set) : Set where
    _,_ : A → B → A × B

  fst : {A B : Set} → A × B → A
  fst (a , b) = a

  snd : {A B : Set} → A × B → B
  snd (a , b) = b
```

$A \times B$ in Agda, first alternative: data

```
module DataProduct where

  data _×_ (A B : Set) : Set where
    _,_ : A → B → A × B

  fst : {A B : Set} → A × B → A
  fst (a , b) = a

  snd : {A B : Set} → A × B → B
  snd (a , b) = b
```

β holds. η does not:

```
-- eta× : {A B : Set} (p : A × B) → p ≡ (fst p , snd p)
-- eta× p = refl                    -- rejected
```

$A \times B$ in Agda, second alternative: record

```
module RecordProduct where

  record _×_ (A B : Set) : Set where
    constructor _,_
    field
      fst : A
      snd : B
  open _×_

  eta× : {A B : Set} (p : A × B) → p ≡ (fst p , snd p)
  eta× p = refl -- accepted
```

Introduction by copattern

Principle 2 says how to build a pair. For a negative type there is a second way: state what each elimination returns.

$$\begin{aligned}&\mathsf{swap} : \{A\ B : \mathsf{Set}\} \to A \times B \to B \times A\\&\mathsf{swap}\ p\ .\mathsf{fst} = \mathsf{snd}\ p\\&\mathsf{swap}\ p\ .\mathsf{snd} = \mathsf{fst}\ p\end{aligned}$$

One equation per field.

This is sound because η holds: the two equations determine swap p completely.

Comparing the two declarations

The two declarations have the same elements and the same β-rule.

They differ in one respect:

whether the extensionality principle holds judgmentally.

record is the negative type of the preceding slides; data is a positive type with the same constructors.

$\top$ — the nullary product

Counting the rules

For a *negative* type, the rules are indexed by its fields

one argument of the introduction per field; one elimination per field; one computation rule per field. Extensionality is a single equation.

For a *positive* type, the rules are indexed by its constructors

one introduction per constructor; one branch of the eliminator per constructor; one computation rule per constructor. Extensionality is a single equation.

$A \times B$ is the instance with two fields. The unit type is the instance with zero fields.

$\top$ by instantiation

1. **Formation:** $\dfrac{}{\Gamma \vdash \top \text{ type}}$
2. **Introduction:** zero arguments, so $\dfrac{}{\Gamma \vdash \langle\rangle : \top}$
3. **Elimination:** zero fields, so there is no elimination rule.
4. **Computation:** zero rules.
5. **Extensionality:** $\dfrac{\Gamma \vdash t : \top}{\Gamma \vdash t = \langle\rangle : \top}$

All of the content of $\top$ is in principle 5.
Without it, the type has no rules that say anything about its elements.

⊤ in Agda

```
module RecordUnit where
  record ⊤ : Set where
    constructor tt

  eta⊤ : (t : ⊤) → t ≡ tt
  eta⊤ t = refl -- accepted
```

$\top$ in Agda

```
module RecordUnit where
  record ⊤ : Set where
    constructor tt

  eta⊤ : (t : ⊤) → t ≡ tt
  eta⊤ t = refl -- accepted


module DataUnit where
  data ⊤ : Set where
    tt : ⊤

  -- eta⊤ : (t : ⊤) → t ≡ tt
  -- eta⊤ t = refl                    -- rejected
```

With η, every element is judgmentally tt; without it, a variable of type $\top$ is opaque.

Booleans — the first positive type

Bool: **principles 1–4**

1. **Formation** $\dfrac{}{\Gamma \vdash \mathsf{Bool}\ \mathsf{type}}$

2. **Introduction** $\dfrac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}} \qquad \dfrac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}$

3. **Elimination**
$$\frac{\Gamma \vdash b : \mathsf{Bool} \qquad \Gamma \vdash t : C \qquad \Gamma \vdash f : C}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ t\ \mathsf{else}\ f : C}$$

4. **Computation**

$$\Gamma \vdash \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ t\ \mathsf{else}\ f = t : C \qquad \Gamma \vdash \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ t\ \mathsf{else}\ f = f : C$$

The result type C is arbitrary, but fixed. In Part 2 it is allowed to depend on b.

Bool: **principle 5**

For any term e with a free variable z of type Bool:

$$\frac{\Gamma, z{:}\mathsf{Bool} \vdash e : C \qquad \Gamma \vdash b : \mathsf{Bool}}{\Gamma \vdash e[b/z] = \mathsf{if}\ b\ \mathsf{then}\ e[\mathsf{true}/z]\ \mathsf{else}\ e[\mathsf{false}/z] : C}$$

A term depending on b is determined by its two instances.

negative $(\times)$ $\quad t = \langle \mathsf{fst}\, t, \mathsf{snd}\, t \rangle$ an equation about t alone

positive (Bool) $\quad e[b/z] = \mathsf{if}\ b\ \mathsf{then}\ \ldots\ \mathsf{else}\ \ldots$ an equation about how b is used

Bool **in Agda**

```
data Bool : Set where
  true false : Bool

if_then_else_ : {C : Set} → Bool → C → C → C
if true then t else f = t
if false then t else f = f

not : Bool → Bool
not b = if b then false else true
```

Agda does not implement principle 5 for data types.

Disjoint sum $A + B$

$A + B$: all five principles

1. $$\frac{\Gamma \vdash A_1 \text{ type} \qquad \Gamma \vdash A_2 \text{ type}}{\Gamma \vdash A_1 + A_2 \text{ type}}$$

2. $$\frac{\Gamma \vdash a : A_i}{\Gamma \vdash \mathsf{in}_i\, a : A_1 + A_2}$$

3. $$\frac{\Gamma \vdash t : A_1 + A_2 \qquad \Gamma, x_1{:}A_1 \vdash t_1 : C \qquad \Gamma, x_2{:}A_2 \vdash t_2 : C}{\Gamma \vdash \mathsf{case}\ t\ \mathsf{of}\ \lambda\{x_1 \to t_1;\ x_2 \to t_2\} : C}$$

4. $\Gamma \vdash \mathsf{case}\ \mathsf{in}_i\, a_i\ \mathsf{of}\ \lambda\{x_1 \to t_1;\ x_2 \to t_2\} = t_i[a_i/x_i] : C$

5. $\Gamma \vdash e[t/z] = \mathsf{case}\ t\ \mathsf{of}\ \lambda\{x_1 \to e[\mathsf{in}_1 x_1/z];\ x_2 \to e[\mathsf{in}_2 x_2/z]\} : C$

Two constructors: two introductions, two branches, two computation rules.

$\times$ compared with $+$

	$A \times B$	$A + B$
polarity	negative	positive
indexed by	2 fields	2 constructors
introduction	one, taking two arguments	two, taking one each
elimination	two projections	one case, two branches
η	$t = \langle \mathsf{fst}\ t, \mathsf{snd}\ t \rangle$	$e[t/z] = \mathsf{case}\ t\ \ldots$

The same five principles, with the roles of introduction and elimination exchanged.

$A + B$ in Agda

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B

case⊎ : {A B C : Set} → A ⊎ B → (A → C) → (B → C) → C
case⊎ (inj₁ a) f g = f a
case⊎ (inj₂ b) f g = g b
```

There is no useful record version: a negative presentation of sums requires further machinery.

$\bot$ — the nullary sum

$\bot$ by instantiation

The same counting principle, on the positive side, with zero constructors.

1. **Formation:** $\dfrac{}{\Gamma \vdash \bot \text{ type}}$
2. **Introduction:** zero constructors, so there is no introduction rule.
3. **Elimination:** case with zero branches, so the premises reduce to the scrutinee and the result type:

$$\frac{\Gamma \vdash t : \bot \qquad \Gamma \vdash C \text{ type}}{\Gamma \vdash \text{case } t \text{ of } \lambda\{\,\} : C} \qquad \text{written absurd } t$$

4. **Computation:** zero rules.
5. **Extensionality:** $\dfrac{\Gamma, z{:}\bot \vdash e : C \qquad \Gamma \vdash t : \bot}{\Gamma \vdash e[t/z] = \text{absurd } t : C}$

Ex falso quodlibet is not a separate axiom: it is the case rule with an empty list of branches.

$\top$ and $\bot$

	$\top$	$\bot$
introduction	one, nullary	none
elimination	none	one, no branches
computation	none	none
content in	principle 5	principle 3

Principle 5 for $\bot$ states that in a context containing $t : \bot$, any two terms of the same type are equal.

Curry–Howard: $\top$ is truth, $\bot$ is falsity, absurd is *ex falso*, and negation is $\neg A := A \to \bot$.

⊥ in Agda

```
data ⊥ : Set where -- a data type with no constructors

⊥-elim : {C : Set} → ⊥ → C
⊥-elim ()

¬_ : Set → Set
¬ A = A → ⊥
```

The absurd pattern () is the empty list of branches of principle 3.

Natural numbers

$\mathbb{N}$: formation and introduction

$$\frac{}{\Gamma \vdash \mathbb{N}\ \mathsf{type}} \qquad \frac{}{\Gamma \vdash \mathsf{zero} : \mathbb{N}} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{suc}\, n : \mathbb{N}}$$

The first inductive type: a constructor whose argument has the type being defined.

Two constructors, so by the counting principle: two branches in the eliminator, two computation rules.

$\mathbb{N}$: the recursor

$$\frac{\Gamma \vdash C \text{ type} \qquad \Gamma \vdash c_z : C \qquad \Gamma, x{:}\mathbb{N} \qquad \vdash c_s : C \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \text{rec } C\ c_z\ (x.y.c_s)\ n : C}\ \text{rec}$$

One branch per constructor. Without the marked hypothesis this is case analysis with a name for the predecessor.

$\mathbb{N}$: the recursor

$$\frac{\Gamma \vdash C \text{ type} \qquad \Gamma \vdash c_z : C \qquad \Gamma, x{:}\mathbb{N},\ y{:}C \vdash c_s : C \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \text{rec } C\ c_z\ (x.y.c_s)\ n : C}\ \text{rec}$$

The additional hypothesis $y : C$ is the result of the recursive call on the predecessor.

Recursion is built into the rule, so every use of rec terminates.

$\mathbb{N}$: computation and extensionality

4. Computation

$$\begin{aligned} \text{rec } C\ c_z\ (x.y.c_s)\ \text{zero} &= c_z : C \\ \text{rec } C\ c_z\ (x.y.c_s)\ (\text{suc } n) &= c_s[n/x][\text{rec } C\ c_z\ (x.y.c_s)\ n/y] : C \end{aligned}$$

This is the first computation rule whose right-hand side mentions the eliminator again.

5. Extensionality, for a term e with a free variable z of type $\mathbb{N}$:

$$\frac{\Gamma, z{:}\mathbb{N} \vdash e : C \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash e[n/z] = \text{rec } C\ e[\text{zero}/z]\ (x.y.\ e[\text{suc } x/z])\ n : C}$$

Unlike the previous principles 5, this one is not derivable from finitely many instances. In Part 2 it turns out to hold only *propositionally*.

Gödel's System T

Simply typed λ-calculus with $\mathbb{N}$ and the recursor, at all types.

- Gödel (1958), *Dialectica*: a consistency proof for arithmetic.
- T is strongly normalising; Tait's computability method (1967) is the basis of later normalisation proofs for type theory.
- T computes exactly the provably total functions of Peano arithmetic.

The recursor may be used at a function type. Ackermann's function is definable in T, although it is not primitive recursive in the first-order sense.

$\mathbb{N}$ in Agda

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ

{-# BUILTIN NATURAL ℕ #-}

rec : (C : Set) → C → (ℕ → C → C) → ℕ → C
rec C cz cs zero = cz
rec C cz cs (suc n) = cs n (rec C cz cs n)

_+_ : ℕ → ℕ → ℕ
m + n = rec ℕ n (λ _ r → suc r) m
```

Addition recurses on its *first* argument. Part 2 returns to this.

Ackermann, by recursion at a function type

$A(0, n) = n + 1, \quad A(m{+}1, 0) = A(m, 1), \quad A(m{+}1, n{+}1) = A(m, A(m{+}1, n)).$

Writing $f_m = A(m, -)$: $\quad f_0 = \mathsf{suc} \quad$ and $\quad f_{m+1} = \mathsf{rec}\ \mathbb{N}\ (f_m\, 1)\ (\lambda n\, r.\ f_m\, r).$

```
ack : ℕ → (ℕ → ℕ)
ack = rec (ℕ → ℕ) suc (λ m f → rec ℕ (f 1) (λ n r → f r))
```

The recursor is used at $C = \mathbb{N} \to \mathbb{N}$.

Summary

Simple type theory

	pol.	form.	intro	elim	comp	ext
$A \to B$	$-$	✓	λ	app	β	$t = \lambda x.\, t\, x$
$A \times B$	$-$	✓	$\langle\,,\,\rangle$	fst, snd	2	$t = \langle \mathsf{fst}\ t, \mathsf{snd}\ t \rangle$
$\top$	$-$	✓	$\langle\rangle$	—	—	$t = \langle\rangle$
Bool	$+$	✓	true, false	if	2	$e[b/z] = \ldots$
$A + B$	$+$	✓	$\mathsf{in}_1, \mathsf{in}_2$	case	2	$e[t/z] = \ldots$
$\bot$	$+$	✓	—	absurd	—	$e[t/z] = \mathsf{absurd}\ t$
$\mathbb{N}$	$+$	✓	zero, suc	rec	2	schema

The empty entries follow from the counting principle: $\top$ has no fields, so no eliminations; $\bot$ has no constructors, so no introductions.

Digression: meta-logic

Three principles

The connectives $\to$, $\times$ and $+$ implement three principles:

connective	**principle**	
$A \to B$	entailment	$A \to B$ iff A entails B
$A \times B$	composition	$A \times B$ iff A and B
$A + B$	choice	$A + B$ iff A or B

None of these principles is created by the connective. Each is already present in the meta-logic — the language in which the inference rules themselves are written. The connectives *reify* them: they turn a structure of the meta-level into a type of the object level.

Where the principles already occur

principle	**in the meta-logic**	**reified as**
entailment	hypothetical judgment $\Gamma, x{:}A \vdash t : B$, together with substitution	$A \to B$
composition	a rule with *two premises*	$A \times B$
choice	*two rules* with the same conclusion	$A + B$

Each row is visible in the shape of the rules of Part 1.

1. Entailment

The hypothetical judgment $\Gamma, x{:}A \vdash t : B$ says that A entails B. It is a notion of the meta-logic: contexts and the structural rules.

Introduction internalises it; elimination is an instance of substitution:

$$\frac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}$$

The computation principle $\Gamma \vdash (\lambda x.t)\,u = t[u/x] : B$ is exactly the substitution principle of the meta-logic, made into an equation.

2. Composition

Composition is the aggregation of information: to have $A \times B$ is to have both A and B.

In the meta-logic this is a rule with *two premises*. The introduction rule for $\times$ has the same shape, and there are correspondingly *two* elimination rules, one recovering each premise:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \times B} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathsf{fst}\, t : A} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathsf{snd}\, t : B}$$

The nullary case is a rule with *no* premises, that is, an axiom: $\top$, with no elimination rules.

3. Choice

Choice carries one bit of information: which of the two alternatives holds.

In the meta-logic this is the presence of *two rules* with the same conclusion. Correspondingly $+$ has two introduction rules, and its eliminator must supply a branch for each:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{in}_1 a : A + B} \qquad \frac{\Gamma \vdash b : B}{\Gamma \vdash \mathsf{in}_2 b : A + B}$$

The nullary case is a type given by *no* rules: $\bot$, with no introduction rules, and an eliminator with no branches.

The counting principle of §5 and §7 is this observation applied to arities.

The object language reflects its meta-language

The type formers do not add reasoning principles. They make principles of the meta-logic available as objects that can themselves be quantified over, computed with, and reasoned about.

This is why the same five principles keep reappearing, and why judgments and propositions must be kept apart — a distinction Part 2 returns to with the identity type.

Ouroboros, Codex Parisinus graecus 2327 (1478),

copied by Theodoros Pelecanos. Public domain.

Part 2

Type equality has so far been syntactic identity, and the CONV rule has not been used.

In Part 2, types may depend on terms:

- Π and Σ;
- the same positive types with *dependent* eliminators, so that recursion becomes induction;
- the identity type, with subst, J and axiom K;
- universes, the inconsistency of $\mathtt{Set} : \mathtt{Set}$, and impredicative quantification.

References

- Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984. Notes by Giovanni Sambin.
- Per Martin-Löf. Constructive mathematics and computer programming. *Logic, Methodology and Philosophy of Science VI*, 1982.
- Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic J. Philos. Logic* 1(1), 1996. (judgments versus propositions)
- Bengt Nordström, Kent Petersson, Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford, 1990.
- Frank Pfenning, Rowan Davies. A judgmental reconstruction of modal logic. *Math. Struct. in Comp. Sci.* 11(4), 2001. (harmony, local soundness and completeness)
- Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica* 12, 1958.
- William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symbolic Logic* 32(2), 1967.
- William A. Howard. The formulae-as-types notion of construction. In *To H.B. Curry*, Academic Press, 1980.
- The Agda documentation. `https://agda.readthedocs.io`