

Lecture 22 – Dependent Types

AAA551: Programming Language Theory

Jihyeok Park



2026 Spring

- Curry-Howard Isomorphism
 - Propositions as Types
 - The Curry-Howard Dictionary
 - Connectives via Scala (\implies , \wedge , \vee , \top , \perp , \neg)
 - Quantifiers (\forall as polymorphism, \exists as existential types)
 - Proofs as Programs (De Morgan, Contraposition, ...)
 - Evaluation as Proof Normalization
- Limits of Intuitionistic Logic
- Need for More Power: types that depend on **terms**

1. Why Dependent Types?
2. Dependent Function Types (Π)
 - Syntax and Intuition
 - Typing Rules
3. Dependent Pair Types (Σ)
 - Syntax and Intuition
 - Typing Rules
4. The Calculus of Constructions (λC)
 - One Syntax for Terms and Types
 - Type Equivalence via Computation
5. Curry-Howard, Reloaded
 - Quantifiers over Values
 - Propositional Equality
6. Dependent Types in Practice

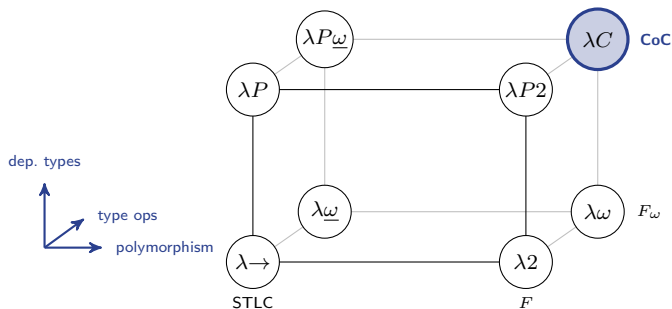
1. Why Dependent Types?
2. Dependent Function Types (Π)
 - Syntax and Intuition
 - Typing Rules
3. Dependent Pair Types (Σ)
 - Syntax and Intuition
 - Typing Rules
4. The Calculus of Constructions (λC)
 - One Syntax for Terms and Types
 - Type Equivalence via Computation
5. Curry-Howard, Reloaded
 - Quantifiers over Values
 - Propositional Equality
6. Dependent Types in Practice

Every abstraction we have studied lets a **term** or a **type** depend on something:

system	form	depends on
STLC	$\lambda x:\tau. e$	term on term
F	$\Lambda \alpha. e$	term on type
F_ω	$\lambda \alpha :: K. \tau$	type on type
?	?	type on term

The missing corner is a **type that depends on a term**. Such a type system is called **dependently typed**.

Eight type systems, three independent axes. Each axis turns on one **new form of dependence** beyond plain λ -calculus:



Bottom-front-left $\lambda \rightarrow$ is STLC; combining all three axes gives the **Calculus of Constructions λC** .

1. Why Dependent Types?
2. Dependent Function Types (Π)
 - Syntax and Intuition
 - Typing Rules
3. Dependent Pair Types (Σ)
 - Syntax and Intuition
 - Typing Rules
4. The Calculus of Constructions (λC)
 - One Syntax for Terms and Types
 - Type Equivalence via Computation
5. Curry-Howard, Reloaded
 - Quantifiers over Values
 - Propositional Equality
6. Dependent Types in Practice

A **dependent function type** (or **Π -type**) is written

$$\Pi(x : \tau_1). \tau_2$$

where τ_2 **may mention** the bound variable x .

Read it as: “a function that, given an $x : \tau_1$, returns a value whose type is τ_2 , possibly depending on x ”.

Two important cases:

$$\begin{aligned} \Pi(x : \tau_1). \tau_2 &= \tau_1 \rightarrow \tau_2 && \text{if } x \notin \text{fvs}(\tau_2) \\ \Pi(\alpha : \text{Type}). \tau &= \forall \alpha. \tau && \text{(type argument case)} \end{aligned}$$

`replicate` : $\Pi(n : \text{Nat}). \Pi(A : \text{Type}). A \rightarrow \text{Vec}[A, n]$
`append` : $\Pi(A : \text{Type}). \Pi(m, n : \text{Nat}).$
 $\text{Vec}[A, m] \rightarrow \text{Vec}[A, n] \rightarrow \text{Vec}[A, m + n]$
`zeroId` : $\Pi(n : \text{Nat}). (n + 0 =_{\text{Nat}} n)$

Each return type **depends on the earlier arguments**:

- `replicate`'s return type mentions the argument n ;
- `append`'s return type uses $m + n$;
- `zeroId`'s return type is a **proposition** about n .

Well-formedness, introduction, elimination:

$$\frac{\Gamma \vdash \tau_1 : \mathbf{Type} \quad \Gamma[x : \tau_1] \vdash \tau_2 : \mathbf{Type}}{\Gamma \vdash \Pi(x : \tau_1). \tau_2 : \mathbf{Type}} \quad \Pi\text{-F}$$

$$\frac{\Gamma[x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1. e : \Pi(x : \tau_1). \tau_2} \quad \Pi\text{-I}$$

$$\frac{\Gamma \vdash e_1 : \Pi(x : \tau_1). \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2 [x \mapsto e_2]} \quad \Pi\text{-E}$$

Two new ingredients beyond STLC:

- $\Pi\text{-F}$ extends Γ with the term-level x to check τ_2 ;
- $\Pi\text{-E}$ **substitutes a term into a type**. Once $x \notin \text{fvs}(\tau_2)$ this is a no-op, recovering plain $\tau_1 \rightarrow \tau_2$.

1. Why Dependent Types?
2. Dependent Function Types (Π)
 - Syntax and Intuition
 - Typing Rules
3. Dependent Pair Types (Σ)
 - Syntax and Intuition
 - Typing Rules
4. The Calculus of Constructions (λC)
 - One Syntax for Terms and Types
 - Type Equivalence via Computation
5. Curry-Howard, Reloaded
 - Quantifiers over Values
 - Propositional Equality
6. Dependent Types in Practice

A **dependent pair type** (or Σ -**type**) is written

$$\Sigma(x : \tau_1). \tau_2$$

Its inhabitants are pairs (a, b) where $a : \tau_1$ and $b : \tau_2[x \mapsto a]$.

The **type of the second component depends on the value of the first**.

Two important cases:

$$\begin{aligned}\Sigma(x : \tau_1). \tau_2 &= \tau_1 \times \tau_2 && \text{if } x \notin \text{fvs}(\tau_2) \\ \Sigma(\alpha : \text{Type}). \tau &= \exists \alpha. \tau && \text{(type witness case)}\end{aligned}$$

A list paired with its length – a **vector of unknown length**:

$$\text{SomeVec}[A] \triangleq \Sigma(n : \text{Nat}). \text{Vec}[A, n]$$

An element together with a **proof** about it:

$$\text{Even} \triangleq \Sigma(n : \text{Nat}). (n \bmod 2 =_{\text{Nat}} 0)$$

– a value of `Even` is a number **bundled with** a proof of its evenness. This is the dependent-types version of a **refinement type**.

Introduction pairs a witness with evidence:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 [x \mapsto e_1]}{\Gamma \vdash (e_1, e_2) : \Sigma(x : \tau_1). \tau_2} \Sigma\text{-I}$$

Elimination projects the witness; the **second projection's type depends on the first**:

$$\frac{\Gamma \vdash e : \Sigma(x : \tau_1). \tau_2}{\Gamma \vdash \text{fst } e : \tau_1} \Sigma\text{-E}_1 \quad \frac{\Gamma \vdash e : \Sigma(x : \tau_1). \tau_2}{\Gamma \vdash \text{snd } e : \tau_2 [x \mapsto \text{fst } e]} \Sigma\text{-E}_2$$

Well-formedness mirrors Π -F:

$$\frac{\Gamma \vdash \tau_1 : \mathbf{Type} \quad \Gamma[x : \tau_1] \vdash \tau_2 : \mathbf{Type}}{\Gamma \vdash \Sigma(x : \tau_1). \tau_2 : \mathbf{Type}} \Sigma\text{-F}$$

1. Why Dependent Types?
2. Dependent Function Types (Π)
 - Syntax and Intuition
 - Typing Rules
3. Dependent Pair Types (Σ)
 - Syntax and Intuition
 - Typing Rules
4. The Calculus of Constructions (λC)
 - One Syntax for Terms and Types
 - Type Equivalence via Computation
5. Curry-Howard, Reloaded
 - Quantifiers over Values
 - Propositional Equality
6. Dependent Types in Practice

In F_ω , terms and types lived in **separate grammars**. λC **merges them** into one grammar with a single Π :

Expressions $e ::= x \mid \lambda x:e. e \mid e e \mid \Pi(x : e). e \mid \text{Type}$

Arrows are sugar: $\tau_1 \rightarrow \tau_2 \triangleq \Pi(- : \tau_1). \tau_2$ when the binder is unused.

One Π **subsumes all three axes** of the λ -cube:

- polymorphism: $\Pi(\alpha : \text{Type}). \alpha \rightarrow \alpha$
- type operators: $\Pi(F : \text{Type} \rightarrow \text{Type}). F \text{ Nat}$
- dependent types: $\Pi(n : \text{Nat}). \text{Vec}[\text{Nat}, n]$

Σ -types, Nat , lists, ... are not primitive in λC . They can be **encoded** via Π (much like \exists via \forall in Lecture 21), but in practice are **added natively** in CIC via inductive types (next section).

Since types **contain terms**, deciding whether two types are equal requires **reducing terms inside them**:

$$\text{Vec}[A, 2 + 3] \equiv \text{Vec}[A, 5]$$

Type equivalence is closed under

- term-level β : $(\lambda x:\tau. e) v \equiv e[x \mapsto v]$
- type-level β (inherited from F_ω)
- congruence under every constructor (Π , application, ...)

The application rule becomes:

$$\frac{\Gamma \vdash e_1 : \Pi(x : \tau_1). \tau_2 \quad \Gamma \vdash e_2 : \tau_3 \quad \tau_1 \equiv \tau_3}{\Gamma \vdash e_1 e_2 : \tau_2[x \mapsto e_2]}$$

Type checking is decidable **only if** the equivalence relation \equiv on terms is decidable.

But in a Turing-complete language, term equality is **undecidable!** Two options:

- **Restrict to total functions** (Coq, Agda, Idris, Lean) – enforced by a **termination checker** and a **positivity check** on inductive types. Evaluation is strongly normalizing, so type checking terminates.
- **Restrict where dependence may occur** (Scala 3, DOT) – types may depend only on a **limited fragment** of terms (paths, singletons), not arbitrary computation:

```
trait Container { type T; def value: T }  
def get(c: Container): c.T = c.value    // c.T : type depends on c
```

1. Why Dependent Types?
2. Dependent Function Types (Π)
 - Syntax and Intuition
 - Typing Rules
3. Dependent Pair Types (Σ)
 - Syntax and Intuition
 - Typing Rules
4. The Calculus of Constructions (λC)
 - One Syntax for Terms and Types
 - Type Equivalence via Computation
5. Curry-Howard, Reloaded
 - Quantifiers over Values
 - Propositional Equality
6. Dependent Types in Practice

In the previous lecture, \forall and \exists ranged over **types**. With Π and Σ , they range over **values** – giving us **first-order predicate logic**:

Logic	Dependent Type
$\forall n : \text{Nat}. P(n)$	$\Pi(n : \text{Nat}). P(n)$
$\exists n : \text{Nat}. P(n)$	$\Sigma(n : \text{Nat}). P(n)$
$a = b$	$a =_A b$

The following are recovered as **special cases** of Π and Σ :

$$\begin{aligned} \tau_1 \rightarrow \tau_2 &= \Pi(x : \tau_1). \tau_2 && (x \notin \text{fvs}(\tau_2)) \\ \tau_1 \times \tau_2 &= \Sigma(x : \tau_1). \tau_2 && (x \notin \text{fvs}(\tau_2)) \\ \forall \alpha. \tau &= \Pi(\alpha : \text{Type}). \tau && (\text{universal}) \\ \exists \alpha. \tau &= \Sigma(\alpha : \text{Type}). \tau && (\text{existential}) \end{aligned}$$

For any type A and any $a, b : A$, the **equality type** $a =_A b$ is a type whose inhabitants are **proofs that a and b are equal**.

It has one introduction rule – **reflexivity**:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \text{refl } a : (a =_A a)} \text{REFL}$$

By the type-equivalence rule, $\text{refl } a$ also has type $a' =_A a'$ whenever $a \equiv a'$. So, for instance,

$$\text{refl } 5 : (2 + 3) =_{\text{Nat}} 5$$

type-checks via the following chain:

- 1 By REFL: $\text{refl } 5 : 5 =_{\text{Nat}} 5$
- 2 Term reduction: $2 + 3 \rightarrow^* 5$, hence $2 + 3 \equiv 5$
- 3 Type equivalence (congruence): $(2 + 3) =_{\text{Nat}} 5 \equiv 5 =_{\text{Nat}} 5$
- 4 Conversion: $\text{refl } 5 : (2 + 3) =_{\text{Nat}} 5 \checkmark$

Equality elimination is **Leibniz's law**: if $a =_A b$ and $P(a)$ holds, then $P(b)$ holds.

$$\frac{\Gamma \vdash P : A \rightarrow \text{Type} \quad \Gamma \vdash p : (a =_A b) \quad \Gamma \vdash q : P(a)}{\Gamma \vdash \text{subst } P p q : P(b)} \text{SUBST}$$

This is exactly what Rocq's `rewrite` tactic does: given a proof of $a = b$, it rewrites a to b inside the current goal.

```
Goal forall n m : nat, n = m -> n + 1 = m + 1.
```

```
Proof.
```

```
  intros n m H.
```

```
  rewrite H.          (* goal becomes m + 1 = m + 1 *)
```

```
  reflexivity.
```

```
Qed.
```

1. Why Dependent Types?
2. Dependent Function Types (Π)
 - Syntax and Intuition
 - Typing Rules
3. Dependent Pair Types (Σ)
 - Syntax and Intuition
 - Typing Rules
4. The Calculus of Constructions (λC)
 - One Syntax for Terms and Types
 - Type Equivalence via Computation
5. Curry-Howard, Reloaded
 - Quantifiers over Values
 - Propositional Equality
6. Dependent Types in Practice

Full-blown dependent types are the basis of modern **proof assistants** and **certified programming** languages:

Language	Foundation	Used For
Coq/Rocq	CIC	CompCert, Iris, math (Feit-Thompson)
Agda	MLTT + universe	teaching, HoTT
Idris	TT + QTT	practical dependent programming
Lean	CIC variant	Mathlib, Liquid Tensor, Fermat
F*	TT + refinements	verified crypto (HACL*)

In each, a **type** is a **theorem statement** and a **well-typed term** is a **machine-checked proof**. The kernel that checks types is the **trusted computing base**.

Coq/Rocq and Lean refine λC in two steps:

CoC (Coquand & Huet, 1988) – pure λC .

- Only Π -types; data types must be **encoded**.
- Works in principle, but induction is awkward.

CIC (Paulin-Mohring, 1993) – CoC + **inductive types**.

- Declare data types directly:

```
Inductive Nat : Type := 0 : Nat | S : Nat -> Nat.
```

- Induction and pattern matching come **for free**.
- The foundation of **Coq/Rocq** and **Lean**.

- Refinement Types

Jihyeok Park
jihyeok_park@korea.ac.kr
<https://plrg.korea.ac.kr>