Lecture 22 – Dependent Types

AAA551: Programming Language Theory

Jihyeok Park

2026 Spring

Recall

- Curry-Howard Isomorphism
  - Propositions as Types
  - The Curry-Howard Dictionary
  - Connectives via Scala ($\Longrightarrow$, $\wedge$, $\vee$, $\top$, $\bot$, $\neg$)
  - 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**

Contents

Contents

All Abstraction So Far

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

All Abstraction So Far

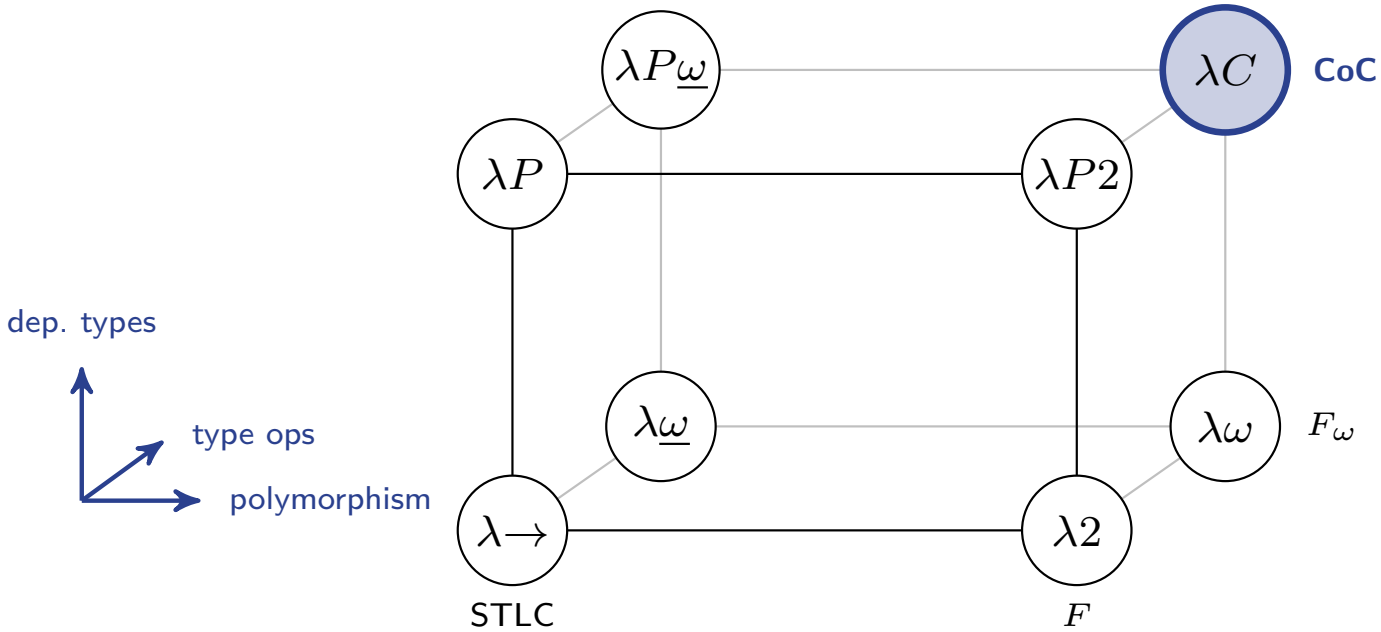
Every abstraction we have studied lets a **term** or a **type** depend on something:

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**.

Barendregt's λ-Cube

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



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

Contents

Dependent Function Type

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

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

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

Dependent Function Type

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".

Dependent Function Type

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:

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

Examples of Π-Types

$$\begin{aligned}
\texttt{replicate} &: \Pi(n : \texttt{Nat}).\ \Pi(A : \texttt{Type}).\ A \to \texttt{Vec}[A, n] \\
\texttt{append} &: \Pi(A : \texttt{Type}).\ \Pi(m, n : \texttt{Nat}). \\
&\qquad \texttt{Vec}[A, m] \to \texttt{Vec}[A, n] \to \texttt{Vec}[A, m + n] \\
\texttt{zeroId} &: \Pi(n : \texttt{Nat}).\ (n + 0 =_{\texttt{Nat}} n)
\end{aligned}$$

Examples of Π-Types

$$
\begin{aligned}
\texttt{replicate} &: \Pi(n : \texttt{Nat}).\ \Pi(A : \texttt{Type}).\ A \to \texttt{Vec}[A, n] \\
\texttt{append} &: \Pi(A : \texttt{Type}).\ \Pi(m, n : \texttt{Nat}). \\
&\qquad \texttt{Vec}[A, m] \to \texttt{Vec}[A, n] \to \texttt{Vec}[A, m + n] \\
\texttt{zeroId} &: \Pi(n : \texttt{Nat}).\ (n + 0 =_{\texttt{Nat}} n)
\end{aligned}
$$

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.

Typing Rules for Π

Well-formedness, introduction, elimination:

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

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

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

Two new ingredients beyond STLC:

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

Contents

Dependent Pair Type

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 $b : \tau_2[x \mapsto a]$.

Dependent Pair Type

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**.

Dependent Pair Type

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:

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

Examples of Σ-Types

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

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

Examples of Σ-Types

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

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

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

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

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

Typing Rules for Σ

Introduction pairs a witness with evidence:

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

Typing Rules for Σ

Introduction pairs a witness with evidence:

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

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

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

Typing Rules for Σ

Introduction pairs a witness with evidence:

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

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

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

Well-formedness mirrors Π-F:

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

Contents

Syntax of λC

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

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

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

Syntax of λC

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

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

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

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

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

Syntax of λC

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

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

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

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

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

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).

Type Equivalence Includes Term Computation

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

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

Type Equivalence Includes Term Computation

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

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

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, ...)

Type Equivalence Includes Term Computation

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

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

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:

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

Decidability and Totality

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

Decidability and Totality

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
```

Contents

Predicate Logic, Now Directly

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**
$\forall n : \texttt{Nat}.\ P(n)$	$\Pi(n : \texttt{Nat}).\ P(n)$
$\exists n : \texttt{Nat}.\ P(n)$	$\Sigma(n : \texttt{Nat}).\ P(n)$
$a = b$	$a =_A b$

Predicate Logic, Now Directly

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**
$\forall n : \texttt{Nat}.\ P(n)$	$\Pi(n : \texttt{Nat}).\ P(n)$
$\exists n : \texttt{Nat}.\ P(n)$	$\Sigma(n : \texttt{Nat}).\ P(n)$
$a = b$	$a =_A b$

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

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

The Equality Type

PLRG

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**.

The Equality Type

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**:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \texttt{refl}\ a : (a =_A a)}\ \textsc{Refl}$$

The Equality Type

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**:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \texttt{refl}\ a : (a =_A a)}\ \textsc{Refl}$$

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

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

type-checks via the following chain:

1. By $\textsc{Refl}$: $\texttt{refl}\ 5\ :\ 5 =_{\texttt{Nat}} 5$
2. Term reduction: $2+3\ \to^*\ 5$, hence $2+3 \equiv 5$
3. Type equivalence (congruence): $(2+3) =_{\texttt{Nat}} 5\ \equiv\ 5 =_{\texttt{Nat}} 5$
4. Conversion: $\texttt{refl}\ 5\ :\ (2+3) =_{\texttt{Nat}} 5$ ✓

Eliminating Equality

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

$$\frac{\Gamma \vdash P : A \to \texttt{Type} \qquad \Gamma \vdash p : (a =_A b) \qquad \Gamma \vdash q : P(a)}{\Gamma \vdash \texttt{subst}\, P\, p\, q : P(b)}\ \textsc{Subst}$$

Eliminating Equality

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

$$\frac{\Gamma \vdash P : A \to \texttt{Type} \qquad \Gamma \vdash p : (a =_A b) \qquad \Gamma \vdash q : P(a)}{\Gamma \vdash \texttt{subst}\, P\, p\, q : P(b)}\ \textsc{Subst}$$

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.
```

Contents

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*)

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**.

From CoC to CIC

Coq/Rocq and Lean refine λC in two steps:

From CoC to CIC

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.

From CoC to CIC

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 := O : Nat | S : Nat -> Nat.
```

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

Next Lecture

- Refinement Types

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