Lecture 21 – Curry-Howard Isomorphism

AAA551: Programming Language Theory

Jihyeok Park

2026 Spring

Recall

- Type Operators
  - Motivation and Kinds
  - Syntax of System F_ω
  - Kinding Rules
  - Type Equivalence (type-level β)
  - Typing Rules
  - Examples (Pair, Id, Comp)
- Higher-Kinded Types in Scala (Type Lambdas, `F[_]`)
- Subtyping for Type Operators ($F^\omega_{<:}$, Variance)

Contents

Contents

Two Worlds, One Structure

Logicians and programmers spent decades doing what looked like very different work. . .

logician	**programmer**
writes a proof of $P \implies Q$	writes a function $P \to Q$

. . . until **Curry** (1934, 1958) and **Howard** (1969) noticed: these are **literally the same activity**.

A **type** is a **proposition**.

A **program** of that type is a **proof** of that proposition.

Running the program is **simplifying** the proof.

First Glimpse

Consider this Scala function:

```
def modusPonens[P, Q](p: P, f: P => Q): Q = f(p)
```

Read as a **program**: given a value of type `P` and a function from `P` to `Q`, return a value of type `Q`.

Read as a **proof**: given that P is true and that $P \implies Q$ is true, conclude that Q is true.

$$\frac{P \qquad P \implies Q}{Q}\ \text{MODUS PONENS}$$

The type signature **is** the logical claim. The function body **is** the proof. The Scala compiler **is** the proof checker.

The Curry-Howard Dictionary

Every logical connective has a **corresponding type constructor**:

Logic	**Types**	**Scala**
$P \implies Q$	$\tau_1 \to \tau_2$	`P => Q`
$P \wedge Q$	$\tau_1 \times \tau_2$	`(P, Q)`
$P \vee Q$	$\tau_1 + \tau_2$	`Either[P, Q]`
$\top$ (true)	`Unit`	`Unit`
$\bot$ (false)	`Void`	`Nothing`
$\neg P$	$\tau \to$ `Void`	`P => Nothing`
$\forall \alpha. P(\alpha)$	$\forall \alpha. \tau$	`[A] => ...`
$\exists \alpha. P(\alpha)$	$\exists \alpha. \tau$	`trait` w/ abstract type

Reading the table **left-to-right**: every logical theorem becomes a type to inhabit. Reading **right-to-left**: every well-typed program is a proof of some proposition.

Contents

Implication is a Function

Natural deduction for $\Longrightarrow$ **is** the STLC typing rule for $\to$:

$$\frac{P \vdash Q}{\vdash P \Longrightarrow Q} \Longrightarrow\text{-I} \qquad \frac{\Gamma[x:\tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1 . e : \tau_1 \to \tau_2}$$

$$\frac{\vdash P \Longrightarrow Q \qquad \vdash P}{\vdash Q} \Longrightarrow\text{-E} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

Three tautologies – the **I, K, S combinators**:

```
// P => P                                          (identity, axiom rule)
def id[P](p: P): P = p

// P => (Q => P)                                   (constant)
def k[P, Q](p: P)(q: Q): P = p

// (P => Q => R) => (P => Q) => (P => R)
def s[P, Q, R](f: P => Q => R)(g: P => Q)(p: P): R = f(p)(g(p))
```

Conjunction is a Pair

Rules for $\wedge$ match the typing rules for **product** $\tau_1 \times \tau_2$:

$$\frac{\vdash P \qquad \vdash Q}{\vdash P \wedge Q}\ \wedge\text{-I} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\vdash P_1 \wedge P_2}{\vdash P_i}\ \wedge\text{-E}_i \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\, e : \tau_1 \quad \Gamma \vdash \texttt{snd}\, e : \tau_2}$$

```
infix type /\[P, Q] = (P, Q)

// commutativity:  P /\ Q => Q /\ P
def andComm[P, Q](pq: P /\ Q): Q /\ P = (pq._2, pq._1)

// fan-out: (P => Q) /\ (P => R)  =>  P => (Q /\ R)
def andFan[P, Q, R](fs: (P => Q) /\ (P => R)): P => (Q /\ R) =
  p => (fs._1(p), fs._2(p))
```

Disjunction is `Either`

Rules for $\vee$ correspond to the **sum** type $\tau_1 + \tau_2$:

$$\frac{\vdash P}{\vdash P \vee Q}\ \vee\text{-I}_1 \qquad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{inl}\, e : \tau_1 + \tau_2}$$

$$\frac{\vdash Q}{\vdash P \vee Q}\ \vee\text{-I}_2 \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{inr}\, e : \tau_1 + \tau_2}$$

$$\frac{\vdash P \vee Q \qquad P \vdash R \qquad Q \vdash R}{\vdash R}\ \vee\text{-E} \qquad \frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma[x_i{:}\tau_i] \vdash e_i : \tau}{\Gamma \vdash \texttt{case}\, e\, \texttt{of}\, \ldots : \tau}$$

```
infix type \/[P, Q] = Either[P, Q]

// commutativity:  P \/ Q => Q \/ P
def orComm[P, Q](e: P \/ Q): Q \/ P = e match
  case Left(p)  => Right(p)
  case Right(q) => Left(q)
```

$\top$ is `Unit`, $\bot$ is `Nothing`

$\top$ has a trivial axiom; `Unit` has a trivial inhabitant:

$$\frac{}{\vdash \top}\ \top\text{-I} \qquad\qquad \frac{}{\Gamma \vdash () : \texttt{Unit}}$$

$\bot$ has **no introduction rule**; `Nothing` has **no inhabitant**. But from $\bot$ **anything follows** (*ex falso quodlibet*):

$$\frac{\vdash \bot}{\vdash P}\ \bot\text{-E} \qquad\qquad \frac{\Gamma \vdash e : \bot}{\Gamma \vdash \texttt{absurd}\, e : \tau}$$

```
type True = Unit
type False = Nothing

def truth: True = ()                    // proof of True
def absurd[P](n: False): P = n          // ex falso
```

In Scala, `Nothing` is a **subtype of every type**, so returning `n` type-checks at any `P` – this is how `absurd` is realized.

Negation

Negation is **defined**, not primitive:

$$\neg P \triangleq P \implies \bot \qquad\qquad \texttt{Not}[\tau] \triangleq \tau \to \texttt{Void}$$

A **refutation** of P takes any proof of P and produces $\bot$:

```
type ~[P] = P => Nothing
```

Provable – $P \implies \neg\neg P$.

$$\lambda p : P.\ \lambda k : P \to \bot.\ k\ p\ :\ P \to (P \to \bot) \to \bot$$

```
def doubleNeg[P](p: P): ~[~[P]] = (k: P => Nothing) => k(p)
```

Not provable – $\neg\neg P \implies P$.

No closed term of type $((P \to \bot) \to \bot) \to P$ exists for arbitrary P.

$\forall$ is Polymorphism

PLRG

In propositional logic, $\forall$ **is** System F's polymorphism:

$$\frac{\alpha \notin \mathsf{fvs}(\Gamma) \qquad \Gamma \vdash P(\alpha)}{\Gamma \vdash \forall \alpha.\, P(\alpha)}\ \forall\text{-I} \qquad \frac{\alpha \notin \mathsf{dom}(\Gamma) \qquad \Gamma[\alpha] \vdash e : \tau}{\Gamma \vdash \Lambda \alpha.\, e : \forall \alpha.\, \tau}$$

$$\frac{\Gamma \vdash \forall \alpha.\, P(\alpha)}{\Gamma \vdash P(\tau)}\ \forall\text{-E} \qquad \frac{\Gamma \vdash e : \forall \alpha.\, \tau' \qquad \Gamma \vdash \tau}{\Gamma \vdash e[\tau] : \tau'[\alpha \mapsto \tau]}$$

```
// forall A. A => A                              (polymorphic identity)
def id[A](a: A): A = a

// forall P, Q. (P /\ Q) => (Q /\ P)     (commutativity of and)
def andComm[P, Q](pq: P /\ Q): Q /\ P = (pq._2, pq._1)
```

∃ is an Existential Type

∃-rules **are** the pack/unpack rules from:

$$\frac{\Gamma \vdash P(t)}{\Gamma \vdash \exists x.\, P(x)}\ \exists\text{-I} \qquad \frac{\Gamma \vdash \tau \qquad \Gamma \vdash e : \tau'[\alpha \mapsto \tau]}{\Gamma \vdash \mathtt{pack}\,\{\tau, e\} : \exists \alpha.\, \tau'}$$

$$\frac{\exists x.\, P(x) \qquad x, P(x) \vdash Q}{\vdash Q}\ \exists\text{-E} \qquad \frac{\Gamma \vdash e_1 : \exists \alpha.\, \tau' \qquad \Gamma[\alpha][x{:}\tau'] \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{let}\{\alpha, x\} = \mathtt{unpack}\, e_1\, \mathtt{in}\, e_2 : \tau_2}$$

A proof of $\exists \alpha.\, \alpha$ **exhibits a witness** and a value:

```
// "There exists a type T and a value of type T."
trait SomeValue:
  type T
  val value: T

val obj: SomeValue = new SomeValue:    // witness T = Int, value = 42
  type T = Int
  val value: T = 42
```

Contents

Example 1: De Morgan (one direction)

Claim: $(\neg P \wedge \neg Q) \implies \neg(P \vee Q)$.

```
def deMorgan[P, Q](nb: ~[P] /\ ~[Q]): ~[P \/ Q] =
  // nb : ~[P] /\ ~[Q]
  //    = (P => False) /\ (Q => False)
  // e  : P \/ Q
  e => e match
    case Left(p) =>
      // p         : P
      // nb._1     : P => False
      nb._1(p)
      // nb._1(p) : False
    case Right(q) =>
      // q         : Q
      // nb._2     : Q => False
      nb._2(q)
      // nb._2(q) : False
```

Example 2: Contraposition

Claim: $(P \implies Q) \implies (\neg Q \implies \neg P)$.

```
def contra[P, Q](f: P => Q): ~[Q] => ~[P] =
  // f : P => Q
  g =>
    // g : ~[Q]
    //   = Q => False
    p =>
      // p       : P
      // f(p)    : Q
      g(f(p))
      // g(f(p)) : False
```

Example 3: Distribution of $\wedge$ over $\vee$

Claim: $P \wedge (Q \vee R) \implies (P \wedge Q) \vee (P \wedge R)$.

```
def distAndOr[P, Q, R](x: P /\ (Q \/ R)): (P /\ Q) \/ (P /\ R) =
  // x._1 : P
  // x._2 : Q \/ R
  x._2 match
    case Left(q) =>
      // q                  : Q
      // (x._1, q)          : P /\ Q
      Left((x._1, q))
      // Left((x._1, q))    : (P /\ Q) \/ (P /\ R)
    case Right(r) =>
      // r                  : R
      // (x._1, r)          : P /\ R
      Right((x._1, r))
      // Right((x._1, r)) : (P /\ Q) \/ (P /\ R)
```

Example 4: Using deMorgan and contra

PLRG

Claim: $(R \implies P \vee Q) \implies (\neg P \wedge \neg Q) \implies \neg R$.

```
// deMorgan : (~[P] /\ ~[Q]) => ~(P \/ Q)
def deMorgan[P, Q](nb: ~[P] /\ ~[Q]): ~[P \/ Q] = ...

// contra    : (P => Q) => (~[Q] => ~[P])
def contra[P, Q](f: P => Q): ~[Q] => ~[P] = ...

def elimOr[P, Q, R](f: R => (P \/ Q))(nb: ~[P] /\ ~[Q]): ~[R] =
  // f             : R => P \/ Q
  // nb            : ~[P] /\ ~[Q]
  // deMorgan(nb) : ~[P \/ Q]                    -- Example 1
  // contra(f)    : ~[P \/ Q] => ~[R]            -- Example 2
  contra(f)(deMorgan(nb))
```

Evaluation is Proof Normalization

The correspondence extends from **statics** to **dynamics**:

$$\underbrace{(\lambda x{:}\tau_1 \,.\, e)\, v \;\to\; e[x \mapsto v]}_{\beta\text{-reduction}}$$

$$\underbrace{\dfrac{\dfrac{\cdots}{\vdash P \implies Q} \qquad \dfrac{\cdots}{\vdash P}}{\vdash Q} \;\rightsquigarrow\; \dfrac{\cdots}{\vdash Q}}_{\text{cut elimination}}$$

In Scala terms, every reducible expression

```
val q: Q = ((p: P) => body)(myP)          // beta-redex
```

is a logical **detour**: "$P \implies Q$" was introduced and immediately eliminated against a proof of P. Reducing it removes the detour and proves Q directly.

Slogan: *running your program = simplifying your proof.* Moreover, STLC's strong normalization = cut-elimination theorem.

Contents

What Cannot Be Proved (in Scala)

Some classical tautologies have **no honest implementation**:

```scala
// excluded middle
def lem[P]: Either[P, P => Nothing] = ???

// double-negation elimination
def dne[P](f: (P => Nothing) => Nothing): P = ???

// Peirce's law
def peirce[P, Q](f: (P => Q) => P): P = ???
```

None of these can be filled in **without cheating** (`null`, exceptions, infinite loops). They are exactly the **classical tautologies that intuitionistic logic does not accept**.

This is a **feature**: intuitionistic proofs are **constructive**, so they correspond to **executable** programs. Classical proofs can be "proofs by impossibility" – nothing to run.

Why We Need More Power

Curry-Howard with F_ω already encodes **propositional and higher-order propositional logic**. But many real theorems quantify over **values**:

$$\forall n : \texttt{Nat}.\ \ n + 0 = n$$

Such a proposition mentions a **term** (n), so its proof must produce evidence that **varies with** n – a **type that depends on a term**:

```
// hypothetical -- not valid Scala
def addZero(n: Nat): (n + 0 =:= n) = ???
```

Allowing types to depend on terms gives us a **dependent type system**, rich enough to state and prove such theorems.

Next Lecture

- Dependent Types

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