The Curry-Howard Correspondence and Dependent Types in Idris 2

Outline

1. **Introduction & Historical Context**
   - Intuitionistic Logic & Theoretical Foundations
2. **The Core Isomorphism (Propositions-as-Types)**
   - Propositions $\leftrightarrow$ Types
   - Proofs $\leftrightarrow$ Programs
   - Simplification $\leftrightarrow$ Evaluation
3. **Mapping Intuitionistic Logic to Simple Types**
   - Implication, Conjunction, Disjunction, and Truth/Falsity
4. **Extending the Correspondence: Predicate Logic**
   - Universal ($\forall$) and Existential ($\exists$) Quantifiers
   - Dependent Types: Dependent Pair (Σ) and Dependent Function (Π)
5. **Idris 2 in Action**
   - Vectors indexed by length (`Vect n a`)
   - Equality proofs as types (`a = b`)
   - Erasure and Quantitative Type Theory ($0, 1, \omega$)
6. **Summary & Key Takeaways**

---

1. Theoretical Foundations

Intuitionistic vs. Classical Logic

- **Classical Logic (Hilbert, Frege):**
  - Truth is absolute (Law of Excluded Middle: $A \vee \neg A$ holds unconditionally).
  - Proof by contradiction is always valid ($\neg \neg A \implies A$).
- **Intuitionistic Logic (Brouwer, Heyting, Kolmogorov):**
  - A proposition is true **only if we can construct a proof for it**.

- Rejects the Law of Excluded Middle as an axiom.
- **Constructive:** To prove $A \vee B$, we must explicitly prove A or prove B

> *"To prove a statement means to construct a mathematical object verifying it."*

---

2. The Core Isomorphism

The Rosetta Stone of Computing and Logic

Logic / Proof Theory	Type Theory / Programming
Proposition	**Type**
Proof	**Program / Term**
Formula Proof Checking	**Type Checking**
Normalization / Proof Reduction	**Evaluation / Computation (β-reduction)**
Consistent System	**Non-empty Type / Habitability**
Inconsistent System (Contradiction)	**Empty Type (`Void`)**

---

3. Propositional Logic $\leftrightarrow$ Simple Types (Part 1)

Implication & Conjunction

1. Implication ($A \Longrightarrow B$) $\leftrightarrow$ Functions (`A -> B`)

- **Logic:** A proof of $A \Longrightarrow B$ is a procedure converting a proof of A into a proof of B.
- **Types:** A function taking an argument of type `A` and producing a result of type `B` .

```
-- Logic: A implies A
-- Type: A -> A
identity : a -> a
identity x = x
```

2. Conjunction ($A \wedge B$) $\leftrightarrow$ Pair / Product Types (`(A, B)`)

- **Logic:** A proof of $A \wedge B$ consists of a proof of A **and** a proof of B.
- **Types:** A tuple containing a value of type `A` and a value of type `B`.

```
-- Logic: (A and B) implies A
proj1 : (a, b) -> a
proj1 (x, y) = x
```

---

4. Propositional Logic $\leftrightarrow$ Simple Types (Part 2)

Disjunction, Truth, and Falsity

3. Disjunction ($A \vee B$) $\leftrightarrow$ Sum Types (`Either A B`)

- **Logic:** A proof of $A \vee B$ is either a proof of A or a proof of B.
- **Types:** `Either A B` instantiated via `Left a` or `Right b`.

4. Falsity ($\bot$) $\leftrightarrow$ Empty Type (`Void`)

- **Logic:** An impossible proposition; no proof exists.
- **Types:** A type with zero constructors (`Void`).
- **Principle of Explosion (EFS):** $\bot \implies A$ becomes `Void -> a`.

```
absurdity : Void -> a
-- Idris checks that Void has no clauses to cover!
```

5. Truth ($\top$) $\leftrightarrow$ Unit Type (`()`)

- **Logic:** A trivially true proposition requiring no non-trivial proof.
- **Types:** A type with exactly one element (`() : ()`).

---

5. Moving Beyond Propositional Logic

Why Do We Need Dependent Types?

- Propositional logic only deals with static propositional variables (A, B).

- **First-Order Predicate Logic** introduces quantifiers:
  - $\forall x \in S, P(x)$ — Universal Quantification ("For all x, $P(x)$ holds")
  - $\exists x \in S, P(x)$ — Existential Quantification ("There exists x such that $P(x)$ holds")
- **Question:** How do we express terms/types where types *depend on values*?
- **Answer: Dependent Types** (Π-types and Σ-types).

---

6. Universal Quantifier ($\forall$) as Π-Types

Dependent Function Types (Π-Types)

- **Mathematical Logic:** $\forall (x : A), P(x)$
- **Type Theory:** A function where the *return type* varies depending on the *argument value*.
- **Idris Syntax:** `(x : A) -> P x`

Example in Idris 2:

```
import Data.Vect

-- A function returning a vector filled with n copies of x.
-- The return type (Vect n a) depends directly on the value of n!
replicate : (n : Nat) -> a -> Vect n a
replicate Z     x = []
replicate (S k) x = x :: replicate k x
```

---

7. Existential Quantifier ($\exists$) as Σ-Types

Dependent Pair Types (Σ-Types)

- **Mathematical Logic:** $\exists (x : A), P(x)$
- **Type Theory:** A pair where the type of the second element depends on the *value* of the first element.
- **Idris Syntax:** `(x : A ** P x)` or `Subset A (\x => P x)`

Example in Idris 2:

```
-- "There exists some natural number n, and a vector of length n containing Ints"
AnyVect : Type
```

```
AnyVect = (n : Nat ** Vect n Int)

exampleVector : AnyVect
exampleVector = (3 ** [1, 2, 3])
```

---

9. Proofs as First-Class Values in Idris 2

Equality as a Type (Equality Proposition)

- Equality in dependent type theory is represented via the **Identity Type** (=).
- `x = y` is a **Type**.
- The single constructor is `Refl` (Reflexivity: `x = x`).

```
-- Theorem: For all n, 0 + n = n
plusZeroL : (n : Nat) -> 0 + n = n
plusZeroL n = Refl  -- Proved trivially by evaluation / beta-reduction

-- Theorem: For all n, n + 0 = n
plusZeroR : (n : Nat) -> n + 0 = n
plusZeroR Z     = Refl
plusZeroR (S k) =
  -- We use equality rewriting/induction
  let prf = plusZeroR k in
  rewrite prf in Refl
```

---

10. Correctness by Construction

Type-Safe Vector Indexing

By encoding invariants into the type, out-of-bounds errors become *logical impossibilities* (type errors at compile time).

```
import Data.Vect

-- Fin n is the type of natural numbers strictly less than n
-- Fin 0 has no elements.
-- Fin 3 has elements: 0, 1, 2.

lookup : Fin n -> Vect n a -> a
```

```
lookup FZ     (x :: xs) = x
lookup (FS k) (x :: xs) = lookup k xs

-- Trying to lookup an index out of bounds will NOT compile!
```

---

11. Modern Idris 2: Quantitative Type Theory (QTT)

Connecting Proofs and Performance

In traditional intuitionistic logic, proofs carry no runtime cost once verified. Idris 2 implements **QTT**, where every variable has a usage count (multiplicity):

- `0` **(Erased):** Used only at compile time for types/proofs. Erased at runtime.
- `1` **(Linear):** Used exactly once at runtime (useful for resource management).
- ω **(Unrestricted):** Used zero or more times at runtime (standard behavior).

```
-- 'n' has quantity 0: It exists purely as a logical proof/type-index.
-- It will be completely removed from compiled binaries!
length : {0 n : Nat} -> {0 a : Type} -> Vect n a -> Nat
length {n} _ = n
```

---

12. Summary Matrix

Complete Correspondence Table

Logic Concept	Curry-Howard Mapping	Idris 2 Representation
Proposition A	Type A	`a : Type`
Proof of A	Term $x : A$	`val : a`
Implication $A \implies B$	Function Type $A \to B$	`a -> b`
Conjunction $A \wedge B$	Product Type $A \times B$	`(a, b)`
Disjunction $A \vee B$	Sum Type $A + B$	`Either a b`
Truth $\top$	Unit Type $\mathbf{1}$	`()`

Falsity $\bot$	Empty Type $\mathbf{0}$	`Void`
Universal Quantifier $\forall x.P(x)$	Dependent Product Type (Π-type)	`(x : a) -> p x`
Existential Quantifier $\exists x.P(x)$	Dependent Sum Type (Σ-type)	`(x : a ** p x)`
Theorem Verification	Type-checking	`idris2 --check`

---

13. Conclusion & Q&A

Key Takeaways

1. **Writing a type is stating a theorem.**
2. **Writing code is constructing a proof.**
3. **Dependent types** extend this power to predicate logic, enabling exact specification of software behavior.
4. **Idris 2** brings Curry-Howard into practical software engineering using **Quantitative Type Theory**, ensuring proofs do not pollute runtime performance.

Questions?

- **References & Recommended Reading:**
  - Wadler, Philip (2015). *"Propositions as Types"*. Communications of the ACM.
  - Brady, Edwin (2017). *"Type-Driven Development with Idris"*. Manning Publications.