

Lecture 3 on Interactive Proof Assistants :

Why Do They Work?

Assaf Kfoury¹

kfoury@bu.edu

Boston University

20 March 2026

¹ <https://www.cs.bu.edu/~kfoury/>

- 1 Reminders About Interactive Proof Assistants
- 2 Two Distinct Foundational Traditions
- 3 Curry-Howard Isomorphism: Core Idea & How It Works
- 4 Historical Roots of Curry-Howard Isomorphism
- 5 Examples in Lean 4
- 6 More Advanced Examples in Lean 4
- 7 Final Comments: Where Does Homotopy Type Theory Fit?

I will go over **Sections 1, 2, 3,** and **4**, relatively fast, and will try to spend more time on **Sections 5** and **6**. I leave my comments in **Section 7** to the reader. Whatever I omit from my presentation, I will make sure that all my slides are available for download later.

Reminder: What Is an Interactive Proof Assistant?

- ▶ An **Interactive Proof Assistant (IPA)** is software that helps humans to construct and verify formalized mathematical proofs.
- ▶ Well-known IPAs include **Lean 4** and **Agda**, in addition to the 'veteran' **Coq/Rocq** and **Isabelle/HOL** which have been constantly updated and extended since they started some 40+ years ago.
- ▶ A proof accepted by an IPA is guaranteed correct, modulo the correctness of what is called *a small auditable kernel*.
- ▶ IPAs are used in mathematics (e.g. the Feit–Thompson theorem in **Coq**), and in software verification (e.g. CompCert with **Coq**, seL4 with **Isabelle**).

▶ SKIP THE NOTES

NOTES for slide 3:

- ▶ In the world of automated reasoning (including IPAs like Lean, Agda, Coq/Rocq, and Isabelle), the “kernel” is the tiny piece of code that actually checks if a mathematical proof is valid. Modern IPAs are massive pieces of software with millions of lines of code. They have complex user interfaces, automated tactics, and AI-driven search tools. If there is a bug in any of that code, the system might accidentally say a false theorem is true. To solve this, these systems use what is called an **auditable kernel** (also known as the **de Bruijn criterion**), according to which a system emits only proofs that can be checked by a tiny trusted kernel, so that soundness depends only on a minimal auditable trusted base. Some of the issues involved:
 - ▶ **Separation of Concerns:** All the messy, complex, and experimental code is used to find the proof.
 - ▶ **The Choke Point:** Once the proof is found, it is compiled down into extremely basic, fundamental logical steps and passed to the kernel.
 - ▶ **The Audit:** The kernel is kept incredibly small (sometimes just a few thousand lines of code). Because it is so small, humans can read every line to ensure the logic rules are programmed flawlessly. Furthermore, you can write other independent, small kernels in different programming languages to double-check the work of the main kernel.

Two Distinct Foundational Traditions

- One important foundational tradition of several IPAs:
Type Theory and, more specifically, *Dependent Type Theory*.
- ***Lean 4***, ***Coq/Rocq***, and ***Agda*** – among others – are all built on *dependent type-theoretic* foundations.
- Understanding *type theory* and *dependent type theory* gives us the reasons for:
 - ***why*** proofs are programs (and vice versa),
 - ***how*** the kernel verifies proofs via type-checking,
 - ***what*** “trust” means in a proof assistant.
- It is the *architectural principle* on which these IPAs are built, not just a technical detail.

Two Distinct Foundational Traditions

However, not all IPAs share the same foundations. There is a meaningful divide between:

- **Type-theoretic systems:**

- ▶ **Coq/Rocq** — *Calculus of Inductive Constructions* (CiC)
- ▶ **Lean 4** — a variant of CiC with extensions
- ▶ **Agda** — *Martin-Löf Intuitionistic Type Theory* (MLTT)

- **Set-theoretic / HOL-based systems:**

- ▶ **Isabelle/HOL, HOL4, HOL Light**
— based on *Higher-Order Logic* (HOL), classical and set-theoretic in spirit

- These represent two different foundational philosophies.

Two Distinct Foundational Traditions

Isabelle/HOL: A Closer Look

- **Isabelle** is a *generic* proof assistant; its meta-logic is called **Pure**.²
- **Isabelle/HOL** is its most widely used instantiation, built on Higher-Order Logic.
- HOL is related to Church's simple type theory; it does use types, but these are *simple* types, not *dependent* types.
- The semantics is **set-theoretic**: the basic objects are sets, not types in the dependent sense.
- The *Curry–Howard Isomorphism* (also called *Curry–Howard correspondence*) does **not** apply to Isabelle/HOL in the direct way it does to the type-theoretic systems.

► SKIP THE NOTES

² The name **Pure** borrows from the programming and type-theory concept of “purity” (having no external state or assumptions). It represents the logic of deduction in its most stripped-down, structural form—free from the “impurity” of actual mathematical axioms.

NOTES for slide 6:

- ▶ Because Isabelle/HOL does not use the Curry-Howard correspondence, it lacks dependent types (types that depend on values, like "a list of length n "). In Lean or Coq/Rocq, you can write a sorting algorithm where the proof that it sorts correctly is woven directly into the type signature of the function itself. You cannot do this natively in Isabelle/HOL.
- ▶ However, abandoning Curry-Howard gives Isabelle a massive advantage in automation: Because Isabelle/HOL uses a simpler, classical higher-order logic without the heavy baggage of dependent types, it is vastly easier to connect it to external automated theorem provers and SMT solvers (like Z3 or cvc5). This is the secret behind Isabelle's famous Sledgehammer tool, which can automatically find proofs for users in ways that are currently much harder to achieve in Lean or Coq/Rocq.

Two Distinct Foundational Traditions

Classification of Major Proof Assistants – other than Lean, Rocq, Agda

System	Foundation	Curry-Howard
NuPRL	Type Theory (Martin-Löf)	Yes Full Curry-Howard; proofs are programs
HOL Light	Higher-Order Logic (set-theoretic)	No Does not support the Curry-Howard correspondence
Metamath	Minimal logic (meta-logical)	Partial Foundation-agnostic; can encode type-theoretic systems but not native
PVS	Higher-Order Logic (set-theoretic)	No Classical logic basis; proofs \neq programs
Mizar	First-Order Logic + Tarski-Grothendieck Set Theory	No Set theory foundation; no computational content

Two Distinct Foundational Traditions

Type-Theoretic vs. Set-Theoretic: Summary of Key Distinctions

Type-Theoretic Systems (Curry-Howard Applicable)

Examples: Lean, Rocq (Coq), Agda, NuPRL

- Foundation: Dependent type theory (e.g., CIC, Martin-Löf Type Theory)
- Proofs are programs; propositions are types
- Full Curry-Howard isomorphism: proofs \leftrightarrow programs
- Constructive logic; computational content extractable

Set-Theoretic/Classical Systems (No Curry-Howard)

Examples: Isabelle/HOL, HOL Light, PVS, Mizar

- Foundation: Higher-order logic or set theory
- Proofs establish truth; no direct computational content
- Classical reasoning (e.g., law of excluded middle)
- Proof objects exist but are not executable programs

Special case: Metamath is foundation-agnostic and can encode various logical systems.

Curry–Howard Isomorphism (CHI)

Core idea

- The **Curry–Howard Isomorphism**
(also called **Curry–Howard Correspondence** and **Propositions-as-Types** principle):
the conceptual core of all IPAs based on *dependent-type theory*.
- The central insight:
 - A **proposition** corresponds to a **type**.
 - A **proof** of a proposition corresponds to a **term** (program) inhabiting that type.
- Consequence: *proof verification = type checking*.
- This is why the kernel of an IPA can be small: it only needs to implement a correct type-checker.

Curry–Howard Isomorphism (CHI)

How CHI works

- ▶ Any mathematical assertion can be formalized as a **well-formed formula** Φ in a suitable formal logic.
- ▶ Any such Φ can be interpreted as a **type** in a dependent type theory \mathcal{TT} (for Lean 4: the *Calculus of Inductive Constructions*).³
- ▶ Φ has a formal proof **if and only if** the type Φ is **inhabited** — i.e., there exists a term M such that $M : \Phi$ in \mathcal{TT} .
- ▶ Such a term M is simultaneously a **proof** of Φ and a **program** in the functional programming language induced by \mathcal{TT} — this is the **Curry–Howard Isomorphism**.
- ▶ Crucially, **if type-checking is decidable** in \mathcal{TT} , then verifying that $M : \Phi$ becomes a purely mechanical computation — and this is what makes interactive proof assistants possible.⁴ Not every type theory meets this condition; the *Calculus of Inductive Constructions* does, which is one reason it was chosen as the foundation for Lean 4 and Coq.

³ Because CIC is a higher-order logic which features function types (implication) and inductive types (which handle AND, OR, True, and False constructively), it naturally and perfectly embeds the entirety of Intuitionistic Propositional Logic (IPL). Just as CIC completely absorbs (IPL) by turning logical connectives into basic types, it absorbs Intuitionistic First-Order Logic (IFOL) by introducing Dependent Types. First-Order Logic adds two major concepts to propositional logic: Universal Quantification (\forall) and Existential Quantification (\exists).

⁴ This condition is non-trivial: in implicitly – but not explicitly – typed lambda calculi, such as **System F**, both type inference and type checking are undecidable, despite the **Curry–Howard Isomorphism** remaining fully intact.

Curry–Howard Isomorphism (CHI)

How CHI works

- ▶ Any mathematical assertion can be formalized as a **well-formed formula** Φ in a suitable formal logic.
- ▶ Any such Φ can be interpreted as a **type** in a dependent type theory \mathcal{TT} (for Lean 4: the *Calculus of Inductive Constructions*).³
- ▶ Φ has a formal proof **if and only if** the type Φ is **inhabited** — i.e., there exists a term M such that $M : \Phi$ in \mathcal{TT} .
- ▶ Such a term M is simultaneously a **proof** of Φ and a **program** in the functional programming language induced by \mathcal{TT} — this is the **Curry–Howard Isomorphism**.
- ▶ Crucially, **if type-checking is decidable** in \mathcal{TT} , then verifying that $M : \Phi$ becomes a purely mechanical computation — and this is what makes interactive proof assistants possible.⁴ Not every type theory meets this condition; the *Calculus of Inductive Constructions* does, which is one reason it was chosen as the foundation for Lean 4 and Coq.

³ Because CIC is a higher-order logic which features function types (implication) and inductive types (which handle AND, OR, True, and False constructively), it naturally and perfectly embeds the entirety of Intuitionistic Propositional Logic (IPL). Just as CIC completely absorbs (IPL) by turning logical connectives into basic types, it absorbs Intuitionistic First-Order Logic (IFOL) by introducing Dependent Types. First-Order Logic adds two major concepts to propositional logic: Universal Quantification (\forall) and Existential Quantification (\exists).

⁴ This condition is non-trivial: in implicitly – but not explicitly – typed lambda calculi, such as *System F*, both type inference and type checking are undecidable, despite the *Curry–Howard Isomorphism* remaining fully intact.

Curry–Howard Isomorphism (CHI)

How CHI works

- ▶ Any mathematical assertion can be formalized as a **well-formed formula** Φ in a suitable formal logic.
- ▶ Any such Φ can be interpreted as a **type** in a dependent type theory \mathcal{TT} (for Lean 4: the *Calculus of Inductive Constructions*).³
- ▶ Φ has a formal proof **if and only if** the type Φ is **inhabited** — i.e., there exists a term M such that $M : \Phi$ in \mathcal{TT} .
- ▶ Such a term M is simultaneously a **proof** of Φ and a **program** in the functional programming language induced by \mathcal{TT} — this is the **Curry–Howard Isomorphism**.
- ▶ Crucially, **if type-checking is decidable** in \mathcal{TT} , then verifying that $M : \Phi$ becomes a purely mechanical computation — and this is what makes interactive proof assistants possible.⁴ Not every type theory meets this condition; the *Calculus of Inductive Constructions* does, which is one reason it was chosen as the foundation for Lean 4 and Coq.

³ Because CIC is a higher-order logic which features function types (implication) and inductive types (which handle AND, OR, True, and False constructively), it naturally and perfectly embeds the entirety of Intuitionistic Propositional Logic (IPL). Just as CIC completely absorbs (IPL) by turning logical connectives into basic types, it absorbs Intuitionistic First-Order Logic (IFOL) by introducing Dependent Types. First-Order Logic adds two major concepts to propositional logic: Universal Quantification (\forall) and Existential Quantification (\exists).

⁴ This condition is non-trivial: in implicitly – but not explicitly – typed lambda calculi, such as *System F*, both type inference and type checking are undecidable, despite the *Curry–Howard Isomorphism* remaining fully intact.

Curry–Howard Isomorphism (CHI)

How CHI works

- ▶ Any mathematical assertion can be formalized as a **well-formed formula** Φ in a suitable formal logic.
- ▶ Any such Φ can be interpreted as a **type** in a dependent type theory \mathcal{TT} (for Lean 4: the *Calculus of Inductive Constructions*).³
- ▶ Φ has a formal proof **if and only if** the type Φ is **inhabited** — i.e., there exists a term M such that $M : \Phi$ in \mathcal{TT} .
- ▶ Such a term M is simultaneously a **proof** of Φ and a **program** in the functional programming language induced by \mathcal{TT} — this is the **Curry–Howard Isomorphism**.
- ▶ Crucially, **if type-checking is decidable** in \mathcal{TT} , then verifying that $M : \Phi$ becomes a purely mechanical computation — and this is what makes interactive proof assistants possible.⁴ Not every type theory meets this condition; the *Calculus of Inductive Constructions* does, which is one reason it was chosen as the foundation for Lean 4 and Coq.

³ Because CIC is a higher-order logic which features function types (implication) and inductive types (which handle AND, OR, True, and False constructively), it naturally and perfectly embeds the entirety of Intuitionistic Propositional Logic (IPL). Just as CIC completely absorbs (IPL) by turning logical connectives into basic types, it absorbs Intuitionistic First-Order Logic (IFOL) by introducing Dependent Types. First-Order Logic adds two major concepts to propositional logic: Universal Quantification (\forall) and Existential Quantification (\exists).

⁴ This condition is non-trivial: in implicitly – but not explicitly – typed lambda calculi, such as *System F*, both type inference and type checking are undecidable, despite the *Curry–Howard Isomorphism* remaining fully intact.

Curry–Howard Isomorphism (CHI)

How CHI works

- ▶ Any mathematical assertion can be formalized as a **well-formed formula** Φ in a suitable formal logic.
- ▶ Any such Φ can be interpreted as a **type** in a dependent type theory \mathcal{TT} (for Lean 4: the *Calculus of Inductive Constructions*).³
- ▶ Φ has a formal proof **if and only if** the type Φ is **inhabited** — i.e., there exists a term M such that $M : \Phi$ in \mathcal{TT} .
- ▶ Such a term M is simultaneously a **proof** of Φ and a **program** in the functional programming language induced by \mathcal{TT} — this is the **Curry–Howard Isomorphism**.
- ▶ Crucially, **if type-checking is decidable** in \mathcal{TT} , then verifying that $M : \Phi$ becomes a purely mechanical computation — and this is what makes interactive proof assistants possible.⁴ Not every type theory meets this condition; the *Calculus of Inductive Constructions* does, which is one reason it was chosen as the foundation for Lean 4 and Coq.

³ Because CIC is a higher-order logic which features function types (implication) and inductive types (which handle AND, OR, True, and False constructively), it naturally and perfectly embeds the entirety of Intuitionistic Propositional Logic (IPL). Just as CIC completely absorbs (IPL) by turning logical connectives into basic types, it absorbs Intuitionistic First-Order Logic (IFOL) by introducing Dependent Types. First-Order Logic adds two major concepts to propositional logic: Universal Quantification (\forall) and Existential Quantification (\exists).

⁴ This condition is non-trivial: in implicitly – but not explicitly – typed lambda calculi, such as **System F**, both type inference and type checking are undecidable, despite the **Curry–Howard Isomorphism** remaining fully intact.

Curry–Howard Isomorphism (CHI)

How CHI works

- ▶ The type theory \mathcal{TT} and its term language are **defined simultaneously**: a type theory requires, at minimum,
 - (i) a language of *types*,
 - (ii) a language of *terms*,
 - (iii) a *typing relation* $M : \Phi$ connecting the two, and
 - (iv) *reduction rules* defining when two terms are definitionally equal (needed to decide $M : \Phi$).

Components (iii) and (iv) already commit us to a term language with computation — namely, the **lambda calculus**, with abstraction $\lambda x. M$, application $M N$, and substitution. The lambda calculus is thus not an add-on to \mathcal{TT} : it is *constitutive* of it.

▶ SKIP THE NOTES

NOTES for slide 15:

The ***Lambda Calculus*** provides the functional core; the ***Calculus of Constructions*** adds dependent and polymorphic types to model quantified logic; the ***Calculus of Inductive Constructions*** adds primitive inductive types to model mathematical structures and structural induction — Lean 4 is based on this final layer.

- ***Layer 1: The Lambda Calculus.*** The foundation is Church's untyped lambda calculus (1930s), which gives you anonymous functions, application, and substitution. The key move toward proof assistants is passing to the simply typed lambda calculus (Church 1940), where every term has a type and the type-checking problem becomes decidable. Under CHI, this already models propositional logic — but only the implicational fragment, with no quantifiers, no induction, and no universe of types.

NOTES for slide 15:

The *Lambda Calculus* provides the functional core; the *Calculus of Constructions* adds dependent and polymorphic types to model quantified logic; the *Calculus of Inductive Constructions* adds primitive inductive types to model mathematical structures and structural induction — Lean 4 is based on this final layer.

- ▶ *Layer 1: The Lambda Calculus.*
- ▶ *Layer 2: The Calculus of Constructions (CoC).* Coquand and Huet (1988) extended the simply typed lambda calculus in two simultaneous directions:
 - **Dependent types:** types may depend on terms, — this is what models first-order and higher-order quantification.
 - **Polymorphism (types depending on types):** following Girard's System F, you get quantification over types themselves.

The result is the **Pure Type System** known as CoC, which sits at the top corner of Barendregt's lambda cube. CoC is already expressive enough to encode many mathematical proofs, but it has a critical limitation: it has **no primitive notion of inductive types**. Natural numbers, lists, trees, and so on must be encoded via Church encodings, which are awkward and lose computational properties you want (*e.g.*, dependent elimination is not always derivable).

NOTES for slide 15:

The ***Lambda Calculus*** provides the functional core; the ***Calculus of Constructions*** adds dependent and polymorphic types to model quantified logic; the ***Calculus of Inductive Constructions*** adds primitive inductive types to model mathematical structures and structural induction — Lean 4 is based on this final layer.

- ▶ ***Layer 1: The Lambda Calculus.***
- ▶ ***Layer 2: The Calculus of Constructions (CoC).***
- ▶ ***Layer 3: The Calculus of Inductive Constructions (CIC).*** Coquand and Paulin-Mohring added **primitive inductive types** to CoC, yielding CIC. This is a substantial extension, not a minor convenience, which adds:
 - A general scheme for declaring inductive type families (covering Nat, List, Eq, etc.)
 - **Primitive recursors and eliminators** generated automatically for each inductive type, giving you structural induction as a first-class feature rather than an encoding.
 - The type Prop as a separate impredicative universe for proof-irrelevant propositions, sitting alongside Type for data.

Lean 4 (like Coq) is based on CIC, further extended with a **hierarchy of universe levels** (Type 0, Type 1, ...) to avoid Russell-style paradoxes, and with **quotient types** and other refinements.

NOTES for slide 15:

- ▶ In **Lean 4** (and similar dependent-type-based proof assistants) verifying a formalized proof is essentially the same as type checking: the kernel/type checker verifies that a proof term has the asserted proposition as its type. But there are important practical and conceptual nuances to understand.
- ▶ Concise elaboration:
 - ▶ CHI: **Lean 4** treats propositions as types and proofs as terms (programs). A formal proof is represented by a term whose type is the proposition you want to prove. So “proof verification” reduces to checking that term.
 - ▶ *What the type checker does*: The kernel (type checker) checks that all constructions are correctly typed, that definitions and applications respect types, and that dependent types and universe levels are consistent. If the kernel accepts the term, the proof is correct (relative to **Lean**’s kernel implementation).
 - ▶ *Tactics vs proof terms*: Users usually write proofs using tactics (e.g., tactic scripts, apply, simp, automation). Those build proof terms. Verification is checking the resulting proof term. Tactics are convenience/meta-level tools; the only thing ultimately trusted is the kernel/type checker checking the term.
 - ▶ *Trusted computing base*: Verification = type checking only if you trust the kernel and the code that produces the proof term that the kernel checks. In **Lean 4** the trusted part is the kernel/type checker plus any primitives implemented in the kernel. Meta-level code (tactics, elaborator, automation, macros, plugins) is not trusted; bugs there can produce wrong proof terms, but the kernel should reject ill-typed terms. If a kernel bug exists, correctness guarantee fails.

NOTES for slide 15:

- ▶ Other checks beyond pure type checking:
 - Universe consistency and level checking are part of type checking.
 - Some systems perform additional checks (*e.g.*, definitional equality normalization, trusted primitives, kernel axioms). In practice, the kernel also verifies that axioms/constants used are as declared.
 - Nonlogical properties (like checking type class inference or automation traces) are meta-level; final verification rests on the kernel.
- ▶ Practical consequences:
 - A verified proof = a proof term that the kernel accepts; thus verification is type checking.
 - When you use tactics, assume the kernel is the ultimate judge: if it type checks the final term, the proof is valid.
 - To minimize the trusted base, rely on the kernel and minimize use of unsafe primitives or external trust (`unsafeCast`, `FFI`, trusting outputs of unverified tools).
- ▶ Summary sentence: In **Lean 4**, verifying a formalized proof is performed by the kernel's type checker: it checks that the constructed proof term has the claimed proposition as its type. However, this statement presumes trust in the kernel and acknowledges that tactics and automation are helper layers that must ultimately yield a well-typed term.

Historical Roots of Curry-Howard Isomorphism

- The correspondence was noticed independently by **Haskell Curry** (1934) and **William Howard** (1969).⁵
- In its original form it links:
 - **Simply-typed lambda calculus** \longleftrightarrow **propositional logic**
 - **System F** (polymorphic lambda calculus) \longleftrightarrow **second-order logic**
- Philip Wadler's essay "**Propositions as Types**" (*CACM*, 2015) gives a very readable account of why this correspondence "keeps arising again and again in different guises."
- CIC and MLTT are powerful *generalizations* of this foundational insight.

⁵ Although the simply-typed lambda calculus (STLC) – first published by Church in 1940 – is the most transparent setting in which the Curry-Howard isomorphism (CHI) becomes fully visible, the CHI is not tied to the STLC. Curry observed the corresponding patterns with a typed version of what is called **Combinatory Logic** in 1934. Howard articulated the full structural correspondence with the STLC in 1969.

Historical Roots of Curry-Howard Isomorphism

Martin-Löf Intuitionistic Type Theory (MLTT)

- Developed by **Per Martin-Löf** (1970s–1980s) as an alternative *foundation for constructive mathematics*.
- Key type formers: **Π -types** (dependent function types), **Σ -types** (dependent pair types), **identity types**, and **universe types**.
- MLTT is **predicative**: it does not allow impredicative definitions (a type cannot quantify over all types including itself).
- The logical reading: Π is universal quantification, Σ is existential quantification, identity types express equality.
- **Agda** is the primary IPA built directly on MLTT.

Historical Roots of Curry-Howard Isomorphism

The Calculus of Inductive Constructions (CIC)

- CiC was developed by **Thierry Coquand** and collaborators, building on the *Calculus of Constructions* (CoC).
- Key additions over MLTT:
 - **Impredicative** sort `Prop`: quantification over all propositions is allowed.
 - **Inductive types**: a general mechanism for defining data types and recursive structures (natural numbers, lists, trees, ...).
 - **Universe hierarchy**: to avoid paradoxes (analogous to Russell's paradox in set theory).
- **Coq/Rocq** is built directly on CIC.
- **Lean 4** uses a variant of CIC with additional extensions (e.g. quotient types, proof irrelevance).

Historical Roots of Curry-Howard Isomorphism

CIC vs. MLTT: Key Differences

- **Predicativity:** MLTT is fully predicative; CIC has an impredicative sort (Prop in Coq/Rocq).
- **Inductive types:** CIC has a built-in, general schema for inductive definitions; MLTT handles these more sparingly.
- **Classical logic:** Both are constructive by default; classical axioms can be added to either, but this is more common in CIC-based systems.
- **Proof relevance:** In MLTT all proofs are potentially relevant (as terms); Lean 4 adds *proof irrelevance* for propositions.
- **Homotopy Type Theory (HoTT):** A major modern extension of MLTT (adding the Univalence Axiom), pursued in Agda and a separate version of Lean.

Historical Roots of Curry-Howard Isomorphism

Dependent Types: Why They Matter

- In simple type theory, types are fixed (e.g. `List Nat`).
- In **dependent type theory**, a type can *depend on a value*:
 - Example: $\text{Vec } A \ n$ — the type of lists of type A with *exactly* n elements (where n is a value).
- This allows types to *express specifications*: a function with type $\Pi (n : \mathbb{N}). \text{Vec } A \ n \rightarrow \text{Vec } A \ n$ is guaranteed to return a list of the same length.
- Dependent types extend the Curry–Howard correspondence to full **predicate logic** and beyond, which is what makes IPAs so expressive.

Historical Roots of Curry-Howard Isomorphism

Summary: The Conceptual Hierarchy

- **Curry-Howard Isomorphism** is the *seed*: propositions are types, proofs are programs.
- **Dependent type theory** is the *generalization* that makes the correspondence powerful enough for real mathematics.
- **MLTT** (Agda) and **CiC** (Coq/Rocq, Lean 4) are two mature instantiations of dependent type theory, differing mainly in predicativity and inductive definitions.
- **HOL-based systems** (Isabelle/HOL) follow a different tradition — set-theoretic and classical — where Curry-Howard does not apply directly.
- Understanding this hierarchy is the key to understanding *why* proof assistants work, not just *how* to use them.

► SKIP THE NOTES

NOTES for slide 21:

Recommended References:

- ▶ **Curry–Howard correspondence (Wikipedia):**
https://en.wikipedia.org/wiki/Curry–Howard_correspondence
- ▶ **Philip Wadler, “Propositions as Types” (CACM 2015):**
<https://dl.acm.org/doi/fullHtml/10.1145/2699407>
- ▶ **Intuitionistic Type Theory (Wikipedia):**
https://en.wikipedia.org/wiki/Intuitionistic_type_theory
- ▶ **Type Theory (Wikipedia):**
https://en.wikipedia.org/wiki/Type_theory
- ▶ **Intuitionistic Type Theory (Stanford Encyclopedia of Philosophy):**
<https://plato.stanford.edu/entries/type-theory-intuitionistic/>

Five Examples With *Lean* 4

Five examples written in *Lean* 4 follow.

- All five examples are structured in the same way:
 - ▶ a brief statement of the logical reading,
 - ▶ the *Lean* 4 code in a verbatim block,
 - ▶ and bullet points explaining what the code illustrates.
- The five examples are:
 - ▶ Logical connective: Implication $P \Rightarrow P$.
Type-theoretic counterpart: Function type / identity function.
 - ▶ Logical connective: Conjunction $P \wedge Q$.
Type-theoretic counterpart: Product type / pair.
 - ▶ Logical connective: Disjunction $P \vee Q$.
Type-theoretic counterpart: Sum type / injection.
 - ▶ Logical connective: Universal quantification $\forall n, P(n)$.
Type-theoretic counterpart: Π -type / dependent function.
 - ▶ Logical connective: Negation $\neg P$ and \perp .
Type-theoretic counterpart: Function to the empty type.

Example 1: Implication as a Function Type

Logical reading: $P \Rightarrow Q$ is the type of functions from P to Q .

A proof of $P \Rightarrow P$ is the identity function.

Lean 4 code

```
-- The proof of the proposition  $P \rightarrow P$  is the identity function

theorem identity_1 (P : Prop) (h : P) : P := h

theorem identity_2 {P : Prop} : P → P := fun x => x -- term mode

theorem identity_3 (P : Prop) : P → P := by          -- tactic mode
  intro z
  exact z
```

Read `identity_1` above, the first implementation of the identity function:

- '`P : Prop`' declares P as a proposition.
- '`h : P`' is a hypothesis, *i.e.*, a *term of type* P .
- The proof term is just `h` — returning what we assumed.
- **Curry–Howard:** the proof of $P \Rightarrow P$ is the program `fun h => h`.

Example 1: Implication as a Function Type

Logical reading: $P \Rightarrow Q$ is the type of functions from P to Q .

A proof of $P \Rightarrow P$ is the identity function.

Lean 4 code

```
-- The proof of the proposition  $P \rightarrow P$  is the identity function

theorem identity_1 (P : Prop) (h : P) : P := h

theorem identity_2 {P : Prop} : P  $\rightarrow$  P := fun x => x -- term mode

theorem identity_3 (P : Prop) : P  $\rightarrow$  P := by           -- tactic mode
  intro z
  exact z
```

Read `identity_1` above, the first implementation of the identity function:

- '`P : Prop`' declares P as a proposition.
- '`h : P`' is a hypothesis, *i.e.*, a *term of type* P .
- The proof term is just h — returning what we assumed.
- **Curry–Howard:** the proof of $P \Rightarrow P$ is the program `fun h => h`.

Example 2: Conjunction as a Product Type

Logical reading: $P \wedge Q$ corresponds to the *product type* $P \times Q$.

A proof of $P \wedge Q$ is a *pair* (proof of P , proof of Q).

Lean 4 code

```
-- From  $P \wedge Q$  we can extract  $P$  (conjunction elimination)
theorem and_left (P Q : Prop) (h : P ∧ Q) : P := h.left

-- A proof of  $P \wedge Q$  is a pair of proofs

-- in term mode:
theorem and_intro_1 (P Q : Prop) (hp : P) (hq : Q) : P ∧ Q := ⟨hp, hq⟩

-- in tactic mode:
theorem and_intro_2 (P Q : Prop) (hp : P) (hq : Q) : P ∧ Q := by
  constructor
  exact hp
  exact hq
```

- `h.left` projects the first component of the pair — exactly as with a product type.
- `⟨hp, hq⟩` constructs the pair — the proof of $P \wedge Q$.
- **Curry–Howard:** conjunction introduction = pair construction; conjunction elimination = projection.

Example 3: Disjunction as a Sum Type

Logical reading: $P \vee Q$ corresponds to the *sum type* (disjoint union) $P + Q$.

A proof of $P \vee Q$ is *either* a proof of P or a proof of Q .

Lean 4 code

```
-- From P we can prove P ∨ Q (left injection)
theorem or_left (P Q : Prop) (hp : P) : P ∨ Q := Or.inl hp

-- From Q we can prove P ∨ Q (right injection)
theorem or_right (P Q : Prop) (hq : Q) : P ∨ Q := Or.inr hq
```

- `Or.inl` and `Or.inr` are the two *constructors* of the sum type — left and right injection.
- **Curry–Howard:** disjunction introduction = injection into a sum type; disjunction elimination = case analysis (pattern matching).

Example 4: Universal Quantification as a Π -type

Logical reading: $\forall n : \mathbb{N}, P(n)$ is the dependent function type $\Pi (n : \mathbb{N}), P n$.

A proof is a function that, given any n , returns a proof of $P(n)$.

Lean 4 code

```
-- Every natural number equals itself
theorem nat_eq_self_1 (n : Nat) : n = n := rfl
theorem nat_eq_self_2 :  $\forall (n : \text{Nat}), n = n := \text{fun } \_ => \text{rfl}$ 

-- Adding zero on the right yields the same number
theorem add_zero_right (n : Nat) : n + 0 = n := rfl
```

- The type of `nat_eq_self_1` and `nat_eq_self_2` is $(n : \text{Nat}) \rightarrow n = n$, which is Π -type because the return type *depends* on the input n .
- `rfl` is the canonical proof term for reflexivity of equality (the identity type).
- **Curry–Howard:** a universally quantified proof *is* a dependent function.

► SKIP THE NOTES

NOTES for slide 27:

- Why does `refl` not need `n` in the implementation of `theorem nat_eq_self_2`?
This is subtle and worth explaining. When Lean 4 elaborates `refl`, it needs to unify the two sides of the equality `n = n`. It does this symbolically at the type level — it sees that both sides are the same term `n` and closes the goal without ever inspecting what value `n` actually is. The variable `n` is present in the type of the proof term but not in its body, which is why the linter flags it as unused.

Example 5: Negation and Absurdity

Logical reading: $\neg P$ is defined as $P \Rightarrow \perp$, where \perp (False) is the *empty type* (no inhabitants).

A proof of $\neg P$ is a function that maps any proof of P to a proof of \perp .

Lean 4 code

```
--  $\neg P$  is notation for  $P \rightarrow \text{False}$ 
-- From a contradiction ( $P$  and  $\neg P$ ) we can prove anything
theorem ex_falso (P Q : Prop) (hp : P) (hnp :  $\neg P$ ) : Q :=
  absurd hp hnp

-- Double negation introduction:  $P \rightarrow \neg\neg P$ 
theorem double_neg_1 (P : Prop) (hp : P) :  $\neg\neg P$  :=      -- term mode
  fun (hmp :  $\neg P$ ) => hmp hp
theorem double_neg_2 (P : Prop) (hp : P) :  $\neg\neg P$  := by    -- tactic mode
  intro hnp ; apply hnp ; exact hp
```

- $\neg P$ unfolds to $P \rightarrow \text{False}$;
absurd applies this function to derive anything.
- **Curry–Howard:** negation = function to the empty type;
ex falso quodlibet = elimination of the empty type.
- Note: $\neg\neg P \Rightarrow P$ (double negation *elimination*) requires classical logic
and is *not* provable in pure Lean 4 without adding the `Classical` axiom.

Seven More Advanced Examples With *Lean* 4

Seven more advanced examples written in *Lean* 4 follow.

- All seven examples are structured in the same way:
 - ▶ a brief statement of the logical reading,
 - ▶ the *Lean* 4 code in a verbatim block,
 - ▶ and bullet points explaining what the code illustrates.
- The seven examples are:
 - ▶ Topic: Existential quantification $\exists n, P(n)$.
Key Curry-Howard point: \exists is a Σ -type; proof = dependent pair with witness.
 - ▶ Topic: Modus ponens & syllogism.
Key Curry-Howard point: Logical inference rules = function application and composition.
 - ▶ Topic: Proof by induction.
Key Curry-Howard point: Induction principle = recursor for the inductive type `Nat`.
 - ▶ Topic: `False` and *ex falso*
Key Curry-Howard point: `False` = empty type; its eliminator = the unique function out of \emptyset .
 - ▶ Topic: Equality as the identity type.
Key Curry-Howard point: `a = b` is a type; `refl` is its constructor; rewriting is its eliminator.
 - ▶ Topic: Commutativity of Addition.
Key Curry-Howard point: proof by induction is structural recursion on an inductive type.
 - ▶ Topic: Distributivity of Multiplication over Addition.
Key Curry-Howard point: a chain of equational reasoning ('`calc`' block) is a chain of compositions of identity-type terms.

Example 6: Existential Quantification as a Σ -type

Logical reading: $\exists n : \mathbb{N}, P(n)$ is the dependent *pair type* $\Sigma (n : \mathbb{N}), P n$.

A proof is a *pair* (w, p) : a witness w and a proof p that $P(w)$ holds.

Lean 4 code

```
-- There exists a natural number that equals zero
theorem exists_zero :  $\exists n : \text{Nat}, n = 0 := \langle 0, \text{rfl} \rangle$ 

-- If every n satisfies P, then there exists an n satisfying P
theorem forall_to_exists (P : Nat  $\rightarrow$  Prop)
  (h :  $\forall n, P n$ ) :  $\exists n, P n := \langle 0, h 0 \rangle$ 
```

- $\langle 0, \text{rfl} \rangle$ is the Σ -type pair: the witness is 0, and `rfl` proves $0 = 0$.
- **Curry–Howard:** existential introduction = pair construction in a Σ -type; existential elimination = dependent pattern matching.
- This generalizes Example 2 (conjunction) to the *dependent* setting.

Example 7: Modus Ponens as Function Application

Logical reading: Modus ponens — from P and $P \Rightarrow Q$, deduce Q — is *exactly* function application in the lambda calculus.

Lean 4 code

```
-- Modus ponens: function application IS the inference rule
theorem modus_ponens (P Q : Prop)
  (hp : P) (hpq : P → Q) : Q := hpq hp

-- Hypothetical syllogism: function composition IS transitivity
theorem hyp_syllogism (P Q R : Prop)
  (hpq : P → Q) (hqr : Q → R) : P → R := fun hp => hqr (hpq hp)
```

- $hpq\ hp$ is simply applying the function hpq to the argument hp .
- $\text{fun } hp \Rightarrow hqr\ (hpq\ hp)$ is function *composition* — the proof of transitivity of implication *is* $hqr \circ hpq$.
- **Curry–Howard:** logical inference rules = term-formation rules of the typed lambda calculus.

Example 8: Proof by Induction as Structural Recursion

Logical reading: Math induction over \mathbb{N} corresponds to *structural recursion* on the inductive type `Nat`.
The recursor for `Nat` is the induction principle.

Lean 4 code

```
-- Proof by induction that 0 is a right identity for addition
theorem add_zero (n : Nat) : n + 0 = n := by
  induction n with
  | zero      => rfl
  | succ n ih => simp [Nat.succ_add, ih]

-- The proof term produced is a recursive function on Nat:
-- add_zero 0      := rfl
-- add_zero (succ n) := ... (add_zero n) ...
```

- The `induction` tactic constructs a *recursive term* over the two constructors of `Nat`, which are `zero` and `succ`.
- `ih` (induction hypothesis) is the *recursive call* in the proof term.
- **Curry–Howard:** induction principle = recursor for an inductive type;
induction step = recursive case of the function.

Example 9: The Proposition `False` as the Empty Type

Logical reading: `False` has *no* proof terms (no inhabitants). From `False` anything follows — this is the *elimination* of the empty type, written `False.elim` in Lean 4.

Lean 4 code

```
-- False.elim : False → C   for any C   (ex falso quodlibet)
theorem from_false (C : Prop) (h : False) : C := False.elim h

-- A concrete use: 0 ≠ 1 lets us prove anything from 0 = 1
theorem zero_ne_one_absurd (h : 0 = 1) (C : Prop) : C :=
  absurd h (by decide)
```

- `False.elim` is precisely the *recursor* for the empty type: since there are no constructors, the case split has no cases.
- **Curry–Howard:** `False` = the empty type \emptyset ; *ex falso* = the unique function out of \emptyset into any type.
- This makes the *consistency* of the type theory concrete: if `False` were inhabited, *every* type would be, collapsing the system.

Example 10: Equality as the Identity Type

Logical reading: The proposition $a = b$ is the *identity type* $\text{Id}(A, a, b)$ from MLTT. Its unique constructor `rfl` witnesses $a = a$. Reasoning about equality is *pattern matching* on this type.

Lean 4 code

```
-- Symmetry of equality: swap the identity type witness
theorem eq_symm A : Type a b : A (h : a = b) : b = a :=
  h ► rfl

-- Transitivity of equality: compose two identity witnesses
theorem eq_trans A : Type a b c : A
  (h1 : a = b) (h2 : b = c) : a = c :=
  h1 ► h2
```

- `rfl` is the sole constructor of the identity type: it proves $a = a$.
- The `►` (substitution) operator *rewrites* along an equality proof — this is the *eliminator* of the identity type.
- **Curry–Howard:** equality proofs *are* terms of the identity type; rewriting *is* the recursor for that type. In Homotopy Type Theory (HoTT), identity types can have rich higher-dimensional structure.

Example 11: Commutativity of Addition

Goal: Prove $\forall m n : \mathbb{N}, m + n = n + m$ from scratch, using only the definitions of `Nat.zero` and `Nat.succ`, and the two recursion equations $0 + n = n$ and $(\text{succ } k) + n = \text{succ } (k + n)$.⁶

Lean 4 code

```
-- Helper 1: succ n + m = succ (n + m)
theorem add_succ_left :
  ∀ (n m : Nat) , Nat.succ n + m = Nat.succ (n + m)
  | _ , 0      => rfl
  | n , m + 1 => congrArg Nat.succ (add_succ_left n m)
-- Helper 2: n + succ m = succ (n + m)
theorem add_succ_right :
  ∀ (n m : Nat) , n + Nat.succ m = Nat.succ (n + m)
  | _ , 0      => rfl
  | n , m + 1 => congrArg Nat.succ (add_succ_right n m)
-- Main theorem: commutativity
theorem add_comm : ∀ (m n : Nat) , m + n = n + m
  | 0,      n => (Nat.zero_add n).trans (Nat.add_zero n).symm
  | m + 1,  n => (add_succ_left m n).trans
                  ((congrArg Nat.succ (add_comm m n)).trans
                   (add_succ_right n m).symm)
```

⁶ In our coding of the proofs in this example and in the next Example 12, we invoke several pre-defined functions in the namespace `Nat` of Lean 4. Thus, we invoke the successor function by writing `Nat.succ`.

NOTES for slide 35:

Recall: the Curry-Howard Isomorphism (CHI) states that *propositions* are *types* and *proofs* are *programs*. Our example is a perfect three-layer demonstration of this:

- Layer 1: Propositions as Types. Each theorem statement is a type:

```
add_succ_left : ∀ (n m : Nat), Nat.succ n + m = Nat.succ (n + m)
add_succ_right : ∀ (n m : Nat), n + Nat.succ m = Nat.succ (n + m)
add_comm : ∀ (m n : Nat), m + n = n + m
```

Under CHI, the type “ $\forall (m\ n : \text{Nat}), m + n = n + m$ ” is not merely a statement about numbers – it is a *type whose inhabitants are proofs*. To assert the theorem is to claim this type is inhabited.

- Layer 2: Proofs as Terms. `add_succ_left` and `add_succ_right` are proved by pattern matching and recursion — exactly the same constructs used to write programs. The proof term:

```
| n , m + 1 => congrArg Nat.succ (add_succ_left n m)
```

is a *recursive function call*. Under CHI: structural induction on `Nat` is primitive recursion, and the type of `congrArg`. $\{u, v\}$ is

$$\{\alpha: \text{Sort } u\} \{\beta: \text{Sort } v\} \{a1\ a2: \alpha\} \\ (f: \alpha \rightarrow \beta) \ (h: a1 = a2) : f\ a1 = f\ a2$$

which is a term-level function.

NOTES for slide 35:

- Layer 3: Proof Composition as Function Composition. The theorem `my_add_comm` is proved by composing the helper lemmas using `.trans` (transitivity of equality) and `.symm` (symmetry of equality). Under CHI, these are not “logical inference rules” — they are *functions that transform proof terms*:

- `.trans`: $\{\alpha : \text{Sort } u\} \rightarrow \{a\ b\ c : \alpha\} \rightarrow (a = b) \rightarrow (b = c) \rightarrow (a = c)$
takes two equality proofs and produces a new one,
- `.symm`: $\{\alpha : \text{Sort } u\} \rightarrow \{a\ b : \alpha\} \rightarrow (a = b) \rightarrow (b = a)$
transforms one proof into another,
- `congrArg` applies a function to both sides of an equality.

The proof of `add_comm` is literally a *program* that computes a proof term by calling helper functions (`add_succ_left`, `add_succ_right`) and composing their results. The entire proof is *executable*: it constructs a witness (an inhabitant) of the type $\forall (m\ n : \text{Nat}),\ m + n = n + m$.

NOTES for slide 35:

There is an additional correspondence which is important to stress, besides the fact that *proofs are programs* as explained in the notes of the preceding slide: ***what we call 'proof by induction' in mathematics is exactly structural recursion in programming.*** In relation to the preceding example:

Commutativity of Addition: Under CHI, proof by induction is structural recursion on an inductive type. More precisely:

- ▶ The proposition $\forall m\ n : \mathbb{N}, m + n = n + m$ is a Π -type over $\mathbb{N} \times \mathbb{N}$.
- ▶ The proof term is a recursive function defined by pattern matching on the constructors of `Nat` (`.zero` and `.succ`).
- ▶ The induction hypothesis is the recursive call of that function.
- ▶ The base case corresponds to the `.zero` constructor clause, closed by `rfl`.
- ▶ The inductive step corresponds to the `.succ` constructor clause, which calls the proof function recursively.

The recursor for `Nat` is the induction principle, and structural recursion is proof by induction. There is no distinction between the two — they are the same thing viewed through different lenses.

Example 12: Distributivity of “ \times ” over “ $+$ ” (Slide 1)

Goal: Prove $\forall k m n : \mathbb{N}, k \times (m + n) = k \times m + k \times n$ from scratch, using only the recursion equations for `Nat.mul`: $0 \times n = 0$ and $(\text{succ } k) \times n = n + k \times n$.

We present the code for this example over three slides. The lemmas (helper theorems) and the main theorem in Example 11 are re-used in this example.

Lean 4 code

```
-- Helper 3: Associativity of addition
theorem add_assoc :  $\forall (a b c : \text{Nat}), (a + b) + c = a + (b + c)$ 
  | 0, b, c =>
    calc (0 + b) + c
      = b + c           := congrArg ( $\cdot + c$ ) (Nat.zero_add b)
    _ = 0 + (b + c)     := (Nat.zero_add (b + c)).symm
  | Nat.succ a, b, c =>
    calc (Nat.succ a + b) + c
      = (Nat.succ (a + b)) + c := add_succ_left a b ► rfl
    _ = Nat.succ ((a + b) + c) := add_succ_left (a + b) c
    _ = Nat.succ (a + (b + c)) := congrArg Nat.succ (add_assoc a b c)
    _ = Nat.succ a + (b + c)   := (add_succ_left a (b + c)).symm
```


Example 12: Distributivity of “ \times ” over “ $+$ ” (Slide 2)

Lean 4 code

```
-- Helper 4 : a + (b + (c + d)) = (a + c) + (b + d)
theorem add_comm_assoc (a b c d : Nat) :
  a + (b + (c + d)) = (a + c) + (b + d) :=
  Eq.trans
    (Nat.add_left_comm a b (c + d))
      -- a + (b + (c + d)) = b + (a + (c + d))
    (Eq.trans
      (congrArg (b + ·) (Eq.symm (add_assoc a c d)))
        -- b + (a + (c + d)) = b + ((a + c) + d)
      (Nat.add_left_comm b (a + c) d))
        -- b + ((a + c) + d) = (a + c) + (b + d)
```

Example 12: Distributivity of “ \times ” over “ $+$ ” (Slide 3)

Lean 4 code

```
-- Main theorem: left distributivity
theorem mul_distrib :  $\forall$  (k m n : Nat), k * (m + n) = k * m + k * n
| 0, m, n      =>
  (Nat.zero_mul (m + n)).trans
  ((Nat.zero_mul m).symm  $\triangleright$  (Nat.zero_mul n).symm  $\triangleright$  rfl)
| k + 1, m, n =>
  let ih := mul_distrib k m n
  calc
    (k + 1) * (m + n)
      = (m + n) + k * (m + n)      :=
        Nat.succ_mul k (m + n)  $\triangleright$  add_comm (k * (m + n)) (m + n)
    _ = (m + n) + (k * m + k * n)  := congrArg ((m + n) +  $\cdot$ ) ih
    _ = m + (n + (k * m + k * n))  := add_assoc m n (k * m + k * n)
    _ = (m + k * m) + (n + k * n)  := add_comm_assoc m n (k * m) (k * n)
    _ = (k + 1) * m + (k + 1) * n :=
      congrArg_2 ( $\cdot$  +  $\cdot$ )
      ((add_comm m (k * m)).trans (Nat.succ_mul k m).symm)
      ((add_comm n (k * n)).trans (Nat.succ_mul k n).symm)
```

[▶ SKIP THE NOTES](#)

NOTES for slide 38:

Recall: the Curry-Howard Isomorphism (CHI) states that *propositions* are *types* and *proofs* are *programs*. As with Example 11, this example admits a clean three-layer reading:

- Layer 1: Propositions as Types. Each theorem statement is a type:

```
add_succ_left  :  $\forall (n\ m : \text{Nat}), \text{Nat.succ } n + m = \text{Nat.succ } (n + m)$   
add_succ_right :  $\forall (n\ m : \text{Nat}), n + \text{Nat.succ } m = \text{Nat.succ } (n + m)$   
add_comm      :  $\forall (m\ n : \text{Nat}), m + n = n + m$   
add_assoc     :  $\forall (a\ b\ c : \text{Nat}), (a + b) + c = a + (b + c)$   
add_comm_assoc :  $\forall (a\ b\ c\ d : \text{Nat}), a + (b + (c + d)) = (a + c) + (b + d)$   
mul_distrib   :  $\forall (k\ m\ n : \text{Nat}), k * (m + n) = k * m + k * n$ 
```

Under CHI, “ $\forall (k\ m\ n : \text{Nat}), k * (m + n) = k * m + k * n$ ” is not merely a statement about numbers — it is a *type whose inhabitants are proofs*. Notice the layered dependency: `mul_distrib` calls `add_comm_assoc`, which calls `add_assoc`, which calls `add_succ_left`, which calls ... , which calls ... etc. The proposition hierarchy is also a ***type dependency hierarchy***.

NOTES for slide 38:

- Layer 2: Proofs as Terms, and Induction as Structural Recursion. The proof of `mul_distrib` is a recursive function defined by pattern matching on the two constructors of `Nat`, with recursion on `k`, the multiplier.
 - The **base case** $k = 0$ constructs a term of type $0 * (m + n) = 0 * m + 0 * n$ by invoking `Nat.zero_mul` three times and composing the resulting equality witnesses with `.trans` and the rewrite operator `►`.
 - The **inductive step** $k+1$ uses `let ih := mul_distrib k m n`—this is the recursive call. Under CHI, the ***induction hypothesis is the recursive call*** of the proof function being defined; there is no separate logical rule for “assume the hypothesis for k ”.

The `calc` block is a chain of equalities each justified by a previously constructed term. Under CHI, this is ***proof composition by transitivity***: each step assembles a longer equality proof from shorter ones via `.trans`, and the entire `calc` is a single term of the required equality type.

NOTES for slide 38:

► Layer 3: Proof Composition as Function Composition, and Interleaved Recursions.

The proof of `mul_distrib` is not merely recursive — it is **mutually layered**: the outer recursion is on k (the multiplier), but each step of that recursion delegates to proofs that themselves recurse on the addition arguments. This layering corresponds exactly to the mathematical fact that distributivity must rest on the properties of addition. Under CHI, the key compositions in the inductive step are:

- `congrArg ((m + n) + ·) ih` applies the function $((m + n) + \cdot)$ to both sides of induction hypothesis $ih : k * (m+n) = k * m + k * n$. The type of `congrArg` is $(f : \alpha \rightarrow \beta) \rightarrow (a_1 = a_2) \rightarrow f a_1 = f a_2$ — a higher-order function that **lifts an equality proof through any function**.
- `add_comm_assoc m n (k * m) (k * n)` is a single term of type $m + (n + (k * m + k * n)) = (m + k * m) + (n + k * n)$. It does the essential arithmetic rearrangement in one step, because it was *abstracted* precisely for this purpose. Under CHI, **lemma abstraction is ordinary functional abstraction**: defining a helper to avoid duplicating a subterm is indistinguishable from defining a helper function in a program.
- The final step uses `congrArg2 (· + ·)`, which lifts two equality proofs simultaneously through a binary function, to combine $m + k * m = (k+1) * m$ and $n + k * n = (k+1) * n$ into the desired conclusion. Its type is $(f : \alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (a_1 = a_2) \rightarrow (b_1 = b_2) \rightarrow f a_1 b_1 = f a_2 b_2$, and its application here is pure **function application to proof terms**.

NOTES for slide 38:

The proof of `mul_distrib` is a *program* that, given any $k\ m\ n : \text{Nat}$, computes — by structural recursion on k — a witness of the type $k * (m + n) = k * m + k * n$. ***The distinction between “proving a theorem” and “writing a program” has fully dissolved.***

Distributivity of Multiplication over Addition: Under CHI, proof by induction is structural recursion on an inductive type. More precisely:

- ▶ The proposition $\forall k\ m\ n : \mathbb{N},\ k \times (m + n) = k \times m + k \times n$ is a Π -type over $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$.
- ▶ The proof term is a recursive function defined by pattern matching on the constructors of `Nat` (`.zero` and `.succ`), with k as the induction variable.
- ▶ The induction hypothesis is the recursive call `mul_distrib k m n`.
- ▶ The base case ($k = 0$) is closed by explicit equality witnesses for `Nat.zero_mul`, composed with `.trans` and ▶.
- ▶ The inductive step ($k + 1$) calls the proof function recursively and composes the result into the `calc` chain using `congrArg`, `add_comm_assoc`, and `congrArg2`.

The recursor for `Nat` is the induction principle, and structural recursion is proof by induction. There is no distinction between the two.

While Example 11 illustrates that *induction* is *recursion*, Example 12 goes further and illustrates that equational reasoning is term-level composition in the identity type — which is the key idea that, when taken to its logical conclusion, leads to ***Homotopy Type Theory*** (HoTT), where identity types can have rich higher-dimensional structure.

Example 12: Type-Checking as Proof Verification

What Lean 4's kernel actually checks (Curry–Howard in action):

- `mul_distrib` is a **term** of type $\Pi (k\ m\ n : \text{Nat}), k \times (m + n) = k \times m + k \times n$ — a three-fold dependent function type (Π^3).
- The **base case** `.zero` reduces by *definitional equality* (`rfl`): Lean 4's kernel computes $0 \times x = 0$ by the recursor and checks both sides are the same normal form.
- The **inductive case** builds a *chain of identity-type terms* (`calc` block): each step is a proof of an equality, and the chain is composed by **transitivity of the identity type**.
- “`let ih`” binds the **induction hypothesis** as a *local term*: it is the recursive call to the proof function, exactly as in a structurally recursive program.
- **Curry–Howard summary for both Example 11 and Example 12:**
 - Propositions = types (Π -types, identity types)
 - Proofs = terms (recursive functions, `calc` chains)
 - Proof verification = type-checking by the kernel
 - Induction = structural recursion on `Nat`

Final Comments

Is Homotopy Type Theory Part of the Foundation of IPAs?

A natural question at this point is whether **Homotopy Type Theory (HoTT)** forms part of the foundation of proof assistants such as *Lean 4* or *Coq/Rocq*.

The short answer is: **no** — and the true situation is essentially the *opposite* of what one might expect:

- HoTT is more closely associated with *Coq/Rocq* than with *Lean 4*.
- The landmark *HoTT Book* (2013) was developed using *Coq* as its primary proof assistant.
- A substantial *Coq* library for HoTT exists, and *Coq*'s type theory accommodates the **Univalence Axiom** (as an optional axiom).
- *Lean 4* is not based on HoTT — its kernel and mainstream library choices follow different design priorities.

Final Comments

What Is Homotopy Type Theory?

HoTT is an extension of **Martin-Löf Type Theory** (a family of dependent-type theories that influences both *Coq/Rocq* and *Lean*) with two key additions:

- **The Univalence Axiom** (Voevodsky):

Equivalence of types is equivalent to equality of types.

This gives a precise foundation for the informal mathematical practice of treating isomorphic structures as identical – *i.e.*, transport along equivalences behaves like genuine substitution, formalizing the informal practice of treating isomorphic structures as interchangeable.

- **Higher Inductive Types (HITs):**

Inductive types that have not only *point* constructors but also *path* constructors (equalities between points) and higher path constructors.

The term “homotopy” arises from interpreting: types as *spaces*, terms as *points*, proofs of equality as *paths*, and proofs of equality of proofs as *homotopies*.

Final Comments

Why Lean 4 and HoTT Pull in Opposite Directions

Lean 4's foundational choices diverge from HoTT in ways that create tensions between the two approaches:

Proof Irrelevance (**Lean 4** default / practical mode)

In **Lean 4**'s standard mode and in much of mathlib's usage, proofs of propositions are often treated as irrelevant for computation and used up to uniqueness. This practical stance reduces the homotopical richness of identity types in typical developments. Any two proofs of the same proposition are *definitionally equal*: the identity type $a = b$ is collapsed to at most one point.

This is **incompatible with HoTT**, where the whole point is that the identity type $a = b$ can have *rich structure* — multiple distinct proofs, higher homotopies, and so on.

Lean 4 also takes **quotient types** as a primitive, providing extensionality without requiring the full machinery of HoTT.

Note

A **Lean4 HoTT** project exists, but it requires disabling proof irrelevance and working in a non-standard mode — it is not part of **Lean 4**'s default workflow or kernel.

Final Comments

Which Proof Assistants Are Closest to HoTT?

	ML TT	CIC	Univalence	Proof Irrel.
Coq	✓	✓	axiom (opt.)	optional
Lean 4	✓	✓	×	✓ (default)
Agda	✓	partial	opt. (<code>--cubical</code>)	optional
Cubical Agda	✓	partial	✓ (built-in)	×

Cubical Agda is the proof assistant where HoTT ideas are most deeply embedded: **Cubical Type Theory** gives a *constructive (computational)* interpretation of Univalence rather than merely postulating it as an axiom.

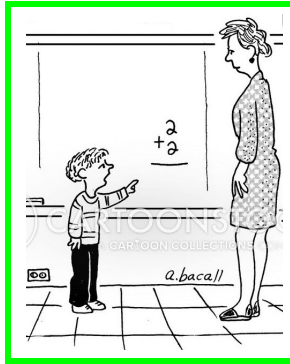
Coq occupies a middle ground: HoTT work is supported (via libraries and axioms) but typically requires adding Univalence and HITs as axioms or working in a specialized mode; consistency relative to CIC is established via model constructions rather than derived internally in standard Coq.

Final Comments

Summary

- HoTT is **not** part of the foundation of Lean 4; the two are in certain respects *incompatible*.
- HoTT is more at home in **Coq** (as an optional axiom and via libraries) and especially in **Cubical Agda** (as a built-in constructive feature).
- The philosophical motivation of HoTT — treating equivalent structures as truly identical — has nonetheless influenced the *design* and design of Lean 4's mathematics library **Mathlib**, even if not its kernel and foundational type theory.
- The proof assistant in which HoTT is most faithfully realized, constructively and without additional axioms, is **Cubical Agda**.
- For the purposes of this lecture: Lean 4's practical stack emphasizes CIC-like dependent typing together with proof-irrelevance conventions and primitive quotient support; HoTT remains a separate and partly rival foundational programme.

Thank you!



Rather than learn how to solve that problem,
is it not better that we learn how to operate software that can solve it?