# Working Principles Of Proof Assistant

And Formalization Of Some Proofs In Agda


**Ashwot Acharya, Bishesh Bohora, Supreme Chaudhary**

**Supervisor:** Mr K.B Manandhar

Kathmandu University

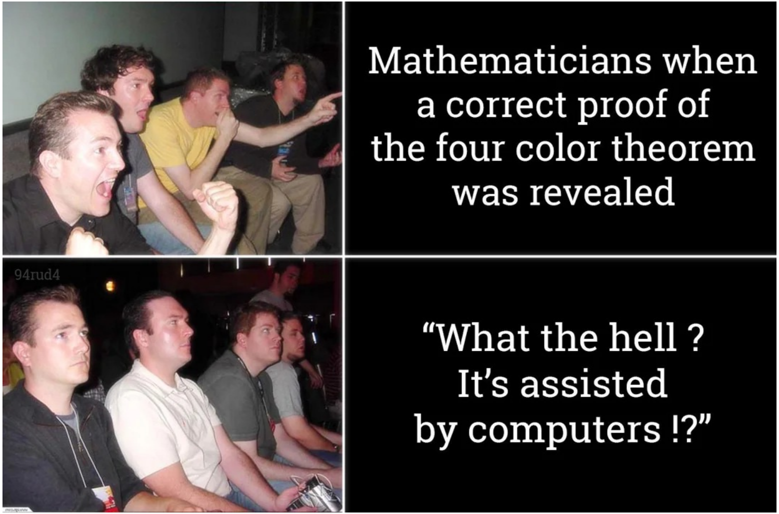
# Proof Assistants

# What are proof assistant

Proof assistant, are software more specifically a type of programming language thats allows us to formalize mathematical proofs in computer for digital verification.

# Need of digital verification

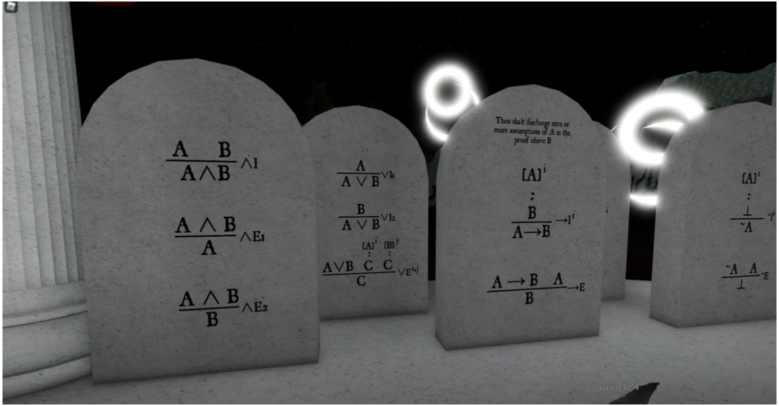
- Fast and Efficient
- Many cases can be explored which would take mathematicians long time
  ex: The Kepler Conjecture's proof , which was so complex that verifying it manually would take 20 person-years, but proof assistants made this verification feasible and fast.
- What if you don't use proof assistants? ABC conjecture


Mathematicians when a correct proof of the four color theorem was revealed
94rud4
"What the hell ? It's assisted by computers !?"

# Foundations

# Natural Deduction

- **Natural Deduction** is a rule-based system for deriving conclusions from assumptions in logic.
- Instead of using exhaustive truth tables, proofs are built step-by-step using inference rules.
- Example: Proving from $A \wedge (A \rightarrow \perp)$ that $\perp$ (contradiction) can be derived.



$\dfrac{A \quad B}{A \wedge B} \wedge I$
$\dfrac{A \wedge B}{A} \wedge E_1$
$\dfrac{A \wedge B}{B} \wedge E_2$
$\dfrac{A}{A \vee B} \vee I_1$
$\dfrac{B}{A \vee B} \vee I_2$
$\dfrac{A \vee B \quad \begin{matrix}[A]^i \\ \vdots \\ C\end{matrix} \quad \begin{matrix}[B]^j \\ \vdots \\ C\end{matrix}}{C} \vee E^{i,j}$
Thou shalt discharge zero or more assumptions of A in the proof above B
$\dfrac{\begin{matrix}[A]^i \\ \vdots \\ B\end{matrix}}{A \to B} \to I^i$
$\dfrac{A \to B \quad A}{B} \to E$
quang1094

# Intuitionistic Logic

- **Intuitionistic Logic** Also called Constructive Logic, reflects principles of constructive mathematics, where a statement is only true if a proof can be constructed.
- Omits some classical logic rules, such as the Law of Excluded Middle.
- Stronger requirement: to prove existence, a method or algorithm must be given.
- Proof assistants leverage this constructive approach for digital verification.

| Introduction Rules | Elimination Rules |
|---|---|
| $\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,\wedge\mathrm{I}$ | $\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,\wedge\mathrm{E_L} \qquad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,\wedge\mathrm{E_R}$ |
| $\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,\vee\mathrm{I_L} \qquad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,\vee\mathrm{I_R}$ | $\dfrac{\Gamma \vdash A \vee B \qquad \Gamma, u{:}A \vdash C \qquad \Gamma, w{:}B \vdash C}{\Gamma \vdash C}\,\vee\mathrm{E}^{u,w}$ |
| $\dfrac{\Gamma, u{:}A \vdash B}{\Gamma \vdash A \supset B}\,\supset\mathrm{I}^u$ | $\dfrac{\Gamma \vdash A \supset B \qquad \Gamma \vdash A}{\Gamma \vdash B}\,\supset\mathrm{E}$ |
| $\dfrac{\Gamma, u{:}A \vdash p}{\Gamma \vdash \neg A}\,\neg\mathrm{I}^{p,u}$ | $\dfrac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash C}\,\neg\mathrm{E}$ |
| $\dfrac{}{\Gamma \vdash \top}\,\top\mathrm{I}$ | *no $\top$ elimination* |
| *no $\bot$ introduction* | $\dfrac{\Gamma \vdash \bot}{\Gamma \vdash C}\,\bot\mathrm{E}$ |
| $\dfrac{\Gamma \vdash [a/x]A}{\Gamma \vdash \forall x.\ A}\,\forall\mathrm{I}^a$ | $\dfrac{\Gamma \vdash \forall x.\ A}{\Gamma \vdash [t/x]A}\,\forall\mathrm{E}$ |
| $\dfrac{\Gamma \vdash [t/x]A}{\Gamma \vdash \exists x.\ A}\,\exists\mathrm{I}$ | $\dfrac{\Gamma \vdash \exists x.\ A \qquad \Gamma, u{:}[a/x]A \vdash C}{\Gamma \vdash C}\,\exists\mathrm{E}^{a,u}$ |

# Inference Rules for Intuitionistic Logic

# Lambda Calculus

## What is Lambda Calculus?

- A formal model of computation by Alonzo Church
- Lambda Terms:

Variables: $x$, $y$, $z$

Abstraction: $\lambda x.E$

Application: $(\lambda x.E)\, F$

## Examples

- $\lambda x.x^2$ is a function
- $(\lambda x.x^2)\,(3) \to 9$

# Type Theory & Simply Typed Lambda Calculus

## Type Theory Basics

- Assigns types to terms: $1 : \mathbb{N}$, $+ : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$
- Typing is decidable

## Type Categories

- Base Types: $\mathbb{N}$, Bool, $\bot$
- Arrow Types: $f : A \longrightarrow B$
- Product Types: $\langle a, b \rangle : A \times B$
- Sum Types: $a : A + B$

## Typed Lambda Calculus

- $(\lambda x : \mathbb{N}.x^2) : \mathbb{N} \longrightarrow \mathbb{N}$

## Dependent Types

- Types depend on values: $Vec\,(n)$
- Indexed types, predicate representation
- $Vec : \mathbb{N} \longrightarrow Type$

# Curry–Howard Correspondence

- sometimes referred as Curry Howard Isomorphism
- A connection between logic, computation, and type theory
- Also known as the *proofs-as-programs* and *propositions-as-types* principle

## Core Idea

- A **proposition** corresponds to a **type**
- A **proof** of the proposition is a **program** (term) of that type
- Proof checking is equivalent to type checking
- Proof normalization corresponds to program evaluation

## Curry–Howard Correspondence

| Logic | Type Theory | Programming |
|---|---|---|
| Proposition | Type | - |
| Proof | Term | Program |
| Implication $A \longrightarrow B$ | $A \longrightarrow B$ | $\lambda$-abstraction |
| Conjunction $A \wedge B$ | $A \times B$ | Pair |
| Disjunction $A \vee B$ | $A + B$ | Tagged union |
| Falsehood $\bot$ | Empty Type | No term |
| Universal $\forall x.A(x)$ | $\Pi x : A.B(x)$ | Function over types |
| Existential $\exists x.A(x)$ | $\Sigma x : A.B(x)$ | Dependent pair |

# Martin-Löf Type Theory

## What is it?

- A formal system for constructive mathematics
- Also known as Intuitionistic Type Theory
- Backbone of modern Proof Assistants

## Core Types

- ($\Pi$-type): Dependent function type, $\forall x : A.B\,(x)$
- ($\Sigma$-type): Dependent sum type, $\exists x : A.B\,(x)$
- **Identity Type**: Internal equality between terms

## Type Universes

- Types have types: $1 : \mathbb{N}, \mathbb{N} : \mathit{Type}, \mathit{Type} : \cdots$
- Not like sets within sets — avoids paradoxes
- Enables reasoning and abstraction over types themselves

# Architecture of proof assistant

# Architecture of a Proof Assistant

- **Kernel**: Minimal, trustworthy codebase enforcing logical rules and validating proofs.
- **Tactic Engine**: Helps build and automate proofs step by step.
- **Formal Proof Language**: Rigorously expresses definitions, statements, and proofs.
- **Libraries**: Collections of verified mathematical foundations for reuse.
- **User Interface**: IDEs and plugins for interactive, efficient proof development.

# Kernel: The Trusted Core

- The **kernel** is the minimal and most critical part of a proof assistant.
- It enforces the logical rules of the underlying formal system (e.g., type theory).
- Responsible for **validating every proof step** to guarantee correctness.
- Ensures **soundness and trustworthiness**; the rest of the system depends on its integrity.
- Typically very small and rigorously tested or formally verified to avoid bugs.
- Example: Agda's kernel is written in Haskell and integrates normalization to check definitional equality.

# Tactic Engine: Proof Construction Assistant

- The **tactic engine** supports users in constructing proofs interactively.
- It breaks complex proof goals into simpler subgoals using **proof strategies** called tactics.
- Provides **automation** for common proof patterns, speeding up proof development.
- Enables both **forward** and **backward** reasoning approaches.
- Even fully automated tactics rely on the kernel for final verification.
- Varies among assistants (Agda has minimal/no tactics, Coq and Lean have powerful tactic systems).

## Formal Proof Language: Expressing Proofs Precisely

- This language allows expressing **definitions, propositions, and proofs** rigorously.
- Typically a **dependently typed language** so logical properties can be encoded as types.
- Provides **syntax and semantics** suitable for formal reasoning and machine checking.
- Enables users to write **human-readable yet unambiguous** formal proofs.
- Integrates smoothly with tactics and type checker to maintain correctness.
- Example languages: Agda's core language, Coq's Gallina, Lean's dependent type language.

# Libraries: Reusable Verified Foundations

- Extensive collections of **formalized mathematics and algorithms** supporting new developments.
- Include **basic theories** such as arithmetic, algebra, logic, and set theory.
- Enable users to **build on existing verified results** without re-proving foundations.
- Libraries evolve and grow, fostering **collaboration and community sharing**.
- Well-maintained libraries reduce duplication and improve proof assistant adoption.
- Examples include Coq's Standard Library, Agda Standard Library, Lean's mathlib.

# User Interface: Proof Development Environment

- Provides **interactive tools** like IDEs, editor plugins, or command line interfaces.
- Features include **syntax highlighting, error reporting, real-time proof state visualization,** and **auto-completion**.
- Enhances **usability and productivity** for proof authors.
- Supports **integration with tactics and proof language** for seamless workflow.
- Examples: CoqIDE, Proof General, Emacs-mode for Agda, VS Code extensions.
- A good interface lowers the learning curve and makes formalization more accessible.

# Comparative Study

# Comparative Table: Agda, Rocq (Coq), and Lean

| Component | Agda | Rocq (Coq) | Lean |
|---|---|---|---|
| **Proof Style** | Explicit term-based, manual proof writing | Tactic-based, automated backward reasoning | Both tactic-based and term-style |
| **Kernel** | Minimal, written in Haskell, tight integration with normalization | Based on Calculus of Inductive Constructions (CIC), written in Coq (extracted to OCaml) | CIC-based, written in C++/C |
| **Type Checking** | Bidirectional, transparent, normalization by evaluation | Bidirectional, heavy conversion, strong automation | Bidirectional, smart elaboration (coercion, backtracking, overloading) |
| **Automation** | Limited (no tactics, minimal automation) | Extensive tactic engine and proof search | Advanced, seamless tactic/term mixing, smart elaborator |
| **Use Cases** | Foundations, education, dependently typed programming | Large/complex formalizations, industrial-scale proofs | Research, education, combinatorial/mathematical formalizations |

# Formalization Of Some proofs

# Eg: Defining Natural Numbers

```
data N : Set where
     Zero : N
     suc : N -> N
```

Thank you!