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

- faster computation for complex problems
- many exceptional 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.

Logical foundation

- Based on logic (natural deduction, intuitionistic logic), λ-calculus, and type theory.
- Curry–Howard Correspondence:

$$\text{Propositions} \leftrightarrow \text{Types}$$
$$\text{Proofs} \leftrightarrow \text{Programs}$$

- Dependently Typed Languages: Types can depend on values, enabling encoding of properties and proofs.

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.
- Basis for how proof assistants check the logical structure of proofs.

Intuitionistic Logic

- **Intuitionistic Logic** formalizes constructive mathematics, where a statement is only true if a proof can be constructed.
- Omits some classical logic principles, 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.

λ-Calculus and Type Theory

- λ-**Calculus**: A foundational system for defining and applying functions using abstraction and application.
- **Type Theory**: Assigns types to every term; ensures correctness of operations.
- **Curry–Howard Correspondence**:

$$\text{Propositions} \leftrightarrow \text{Types}$$
$$\text{Proofs} \leftrightarrow \text{Programs}$$

- *Dependent types* allow types to depend on values, expressing complex logical properties.

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!