Working Principles of Proof Assitants and Formalization of Some Proofs in Agda

Ashwot Acharya, Bishesh Bohora, Supreme Chaudary

Kathmandu University

June 3, 2025

Outline

Introduction

As the title suggests, our project will revolve around exploration of theoritical fondations behind Proof Assitants and practice them.

Proof Assistants

In computer science and mathematical logic, a proof assistant or interactive theorem prover is a software tool to assist with the development of formal proofs by human–machine collaboration. [Wikipedia, 2025]

Examples:

- Coq
- LEAN
- Agda

History

- **Gödel's Incompleteness Theorems (1930s)**: Revealed limitations of formal systems; sparked interest in formal logic and verification.
- **Computability Theory (1940s–50s)**: Turing machines and λ-calculus laid the groundwork for mechanized reasoning.
- **Logic Theorist (1954)**: First automated theorem prover by Newell and Simon, capable of proving theorems in propositional logic.
- **LISP (1960)**: A symbolic programming language created by John McCarthy; became essential for early theorem proving systems.
- **Automath (1967)**: First system to check mathematical proofs using dependent types.

- **LCF & ML (1970s)**: Introduced tactic-based proofs and the ML programming language; foundational to later systems.
- **Coq (1986)**: A proof assistant based on constructive type theory, supporting verified programming and formal proofs.
- **Isabelle (1989)**: Generic theorem prover with support for multiple logics and strong automation tools.
- **Four-Color Theorem (1996)**: First major mathematical theorem re-verified by proof assistants (Coq and HOL).
- **Feit-Thompson Theorem (2012)**: Large-scale group theory proof formalized in Coq, showcasing proof assistant capability.
- **Lean (2015-2023)**: Modern proof assistant combining type theory with performance and usability; popular in formal math via `mathlib`.

Motivation

- Strong interest in mathematics and formal reasoning.
- Discovered type theory through internet memes on category theory.
- Fascinated by the Four Colour Theorem and its computer-assisted proof.
- The rise of AI raised the question: *"How do computers understand reasoning?"*
- Drawn to functional programming, which closely mirrors mathematical logic and structure.

Type Theory

- Type theory is a formal system that classifies expressions by their "types."
- Originally developed as an alternative to set theory for foundations of mathematics.
- Predecessor to Dependent Type Theory, Martin Löf Type Theory which form basis for various proof assitants.
- Types prevent logical paradoxes and provide a basis for constructive reasoning.

Curry–Howard Correspondence

- A deep analogy between **logic and computation**:
  - Propositions $\leftrightarrow$ Types
  - Proofs $\leftrightarrow$ Programs
- A proof of a proposition is a program of a corresponding type.
- Enables writing code that is **correct-by-construction**.
- Fundamental to systems like Coq, where proving a theorem is like writing a program.

λ-Calculus and Functional Programming

- **Lambda Calculus**: A minimal formal system for function definition and application; the foundation of computation theory.
- **Functional Programming**: Directly inspired by lambda calculus; treats computation as evaluation of mathematical functions.
- In proof assitants, Core logic is based on typed lambda calculus.
- Tools like Coq and Agda embed functional languages with type theory.

Methodology

- Investigating the Coq and the agda proof assistant
- Assessing the logic behind each of these proof assistants
- Investigating How to formalized proofs in Agda

Agda

Agda is a functional programming language with dependent types. It is based on Martin Löf Type Theory. And most importantly it is a proof assitant. [Bove et al., 2009]

Why Agda?

- Dependently Typed programming language
- Fully embraces the Curry–Howard isomorphism
- Minimal where users typically define the core logical construct from small set of primitives
- Active Community

Work Plan

Week	Work plan
1	Understanding (Dependent) Type Theory and Proof Theory
2	Understanding the implementation of type theory models in digital proof assistant
3	Understanding Purely functional Programming paradigm and λ-calculus
4	Working Principles of Agda and its core implementation
5	Formalization of some proofs in Agda

Significance

- Writing proofs in a formal language adds rigor and reduces ambiguity, allowing proof assistants to treat proofs as executable code that can be mechanically checked, shifting trust from human readers to computers and minimizing errors in mathematical verification.
- Using the same theoretical foundation, treating programs as proofs and verifying their correctness increases reliability, ensuring that code—especially for sensitive applications—is secure and functions exactly as intended.

Expected Outcomes

- Understanding of dependent type theory and logics
- Understanding of λ-calculus and purely functional programming
- Formalization of some selected proofs in agda

References

Bove, A., Dybjer, P., and Norell, U. (2009).
A brief overview of agda – a functional language with dependent types.

Wikipedia (2025).
Proof assistant — wikipedia, the free encyclopedia.
https://en.wikipedia.org/wiki/Proof_assistant.