Working Principles of Proof Assistants and Formalization of some proofs in Agda

Bishesh Bohora, Supreme Chaudhary, Ashwot Acharya
Kathmandu University

July 26, 2025

Abstract

Contents

1 Introduction

2 Foundations

This section sheds light upon some fundamental concept on which Proof Assistants work along with how a proof should be written so that it can be mechanically verified.

2.1 Logic Foundations

This works assumes prior knowledge of Propositional and Predicate Logic. And we refer to both together as **Classical**.

2.1.1 Natural Deduction

The propositions or formulas in Propositional Logic can be verified or proved simply by constructing their truth tables. But for logically complex propositions or propositions with many atomic statements, it becomes difficult to construct a truth table. With predicates, this becomes impossible. Therefore, to mitigate this we adhere to a basic set of inference rules with which we derive conclusions from assumptions in step by step, structured manner. The rule based system which allows us to reason about logical structure of propositions is known as **Natural Deduction**.
With the rules in Section 2.3 of [**?**] we now present proofs of some propositions carried out with Natural Deduction for Classical Logic.

Proposition 2.1 $(A \wedge \neg A) \to \bot$

By negation introduction we can write $\neg A$ *as* $A \to \bot$
$(A \wedge (A \to \bot)) \to \bot$

$$\frac{[A] \qquad [A \to \bot]}{\bot} \to E$$

Proposition 2.2 *Proof for Law of Excluded Middle* $(P \vee \neg P)$
Note that Proposition 2.1 is used here.

$$\dfrac{\dfrac{\dfrac{[P]}{P \vee \neg P} \vee I \qquad [\neg(P \vee \neg P)]}{\bot} \bot}{\neg P} \neg I$$
$$\dfrac{\dfrac{P \vee \neg P}{\bot} \vee I \qquad [\neg(P \vee \neg P)]}{\neg\neg(P \vee \neg P)} \bot \neg I$$
$$P \vee \neg P \; \neg\neg E$$

Proposition 2.3

$$\forall x(A(x) \to B(x)), A(s) \vdash \exists x B(x)$$

$$\dfrac{\dfrac{[\forall x(A(x) \to B(x))] \qquad [A(s)]}{A(s) \to B(s)} \forall E}{B(s)} \to E$$
$$\exists x B(x) \; \exists I$$

The **soundness** and **completeness** of this system are discussed in Section 3.1 and 3.2 [**?**].

2.1.2 Intuitionstic Logic

Intuitionstic Logic was introduced to formalize the constructive method to do mathematics. Unlike in Classical Logic a statement is True if we can construct a proof for it and to claim a statement is False, again a proof of its falsity is required. This allows the case that some statements are not provable. To show something exists one must provide an method or algorithm to construct it. Proof Assistants leverage this fact. The constructive view of doing mathematics gives a stricter criteria. Intuitionstic Logic can be obtained by restricting certain parts of Classical Logic, like the Law of Excluded Middle.

For inference rules for Natural Deduction in Intuitionstic Logic See 2.1 [**?**], here the language is slightly different from above, we have

Terms $t ::= x \mid a \mid f(t_1, \ldots, t_n)$

Propositions $A ::= P(t_1, \ldots, t_n) \mid A_1 \wedge A_2 \mid A_1 \supset A_2 \mid A_1 \vee A_2 \mid \neg A \mid \bot \mid \top \mid \forall x.A \mid \exists x.A$

The main focus is that this new set of rules does not contain the double negation rule which is present in what we introduced in Section 2.1.1. **Example 2.2** uses the Double negation rule in its proof, which we don't have now. This agrees to that Law of Excluded Middle does not work in Intuitionstic Logic. The method of proof by contradiction also relies on this rule, so constructivists omit it.

These rules are revised with localized hypotheses i.e we use the above rules under a set of premises, and refer the derivation as "derived under a context". With introduction of contexts

$$Contexts\ \Gamma ::= .|\Gamma, u : A$$

[See 2.3 [**?**]]

2.2 λ - Calculus and Type Theory

2.2.1 λ-Calculus

It is a model of computation introduced by Alzano Church. It consists of construction and operations on lambda terms. Among the lambda terms we have,

1. Variables : $x, y, z...$ are lambda terms
2. Application : If E,F are lambda terms EF is a lambda term
3. Abstraction : If E is a lambda term $\lambda x.E$ is a lambda term

Therefore lambda calculus is a formal system involving Abstraction and Application of functions along with some reduction rules. Abstraction is the definition of a anonymous function , for example

$$\lambda x.x^2$$

Now Application is calling that given function by applying it.

Example 2.1

$$\begin{aligned}(\lambda x.x^2)3\\ 3^2\\ 9\end{aligned}$$

The functions can be constructed such that they can take multiple values too, say a function to compute sum of squares can be implemented with lambda calculus as

Example 2.2

$$\lambda x \lambda y.x^2 + y^2$$

Its application is as

$$(\lambda x \lambda y.x^2 + y^2)(3)(4)$$
$$(\lambda y.3^2 + y^2)(4)$$
$$3^2 + 4^2$$
$$9 + 16$$
$$25$$

This is called **currying**.
The reduction rules are :

1. α-Conversion
   The function remains the same if all bound variable is renamed. In above example $\lambda y.y^2$ does the same thing.

   $\lambda x.t$ is same as $\lambda y.t[x := y]$

2. β-Reduction
   This is substitution. During application all the instances of bound variable in the expression is replaced by the argument. Again referring above example x^2 becomes 3^2.
   $(\lambda x.E)F$ is $E[x := F]$, where $E[x := F]$

For comprehensive treatment of this topic, see [**?**]. And for the original formulation by Church []

2.2.2 Type Theory

Type Theory is a formal system that associates every object or **term** a **Type**. The earliest version of Type Theory was introduced by Russell to avoid paradoxes in Set Theory (like **Russell's Paradox**). Later with development of Simply Typed Lambda Calculus, this became important as a foundation of Programming Languages.

Unlike Sets, Types do not talk about the semantics but rather the structure or syntax only. For example, in Type theory, $1 + 2 : \mathbb{N}$ we carry out the judgement that 1+2 is a Natural Number whereas while talking bout sets $1 + 2 \in \mathbb{N}$ means that 1+2 is inside $\mathbb{N}$. With types, we do not care what 1+2 means, the information we have is the operator '+' has a type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and $1 : \mathbb{N}, 2 : \mathbb{N}$ therefore the judgement $1 + 2 : \mathbb{N}$ follows. Whereas with sets, we proceed as $1 + 2 = 3 \in \mathbb{N}$.

Note that verifying if a term a is of type A is decidable, i.e there is an algorithm for it. This process is called Typing or Type Checking. Proof Assistants leverage this for proof checking. Proof checking is decidable; proof finding not. [**?**]

In simple type theory there are,

- **Base Types** : Primitive types which are not built from anything else, Eg: Boolean Type, Nat Type, $\perp$ or Empty Type etc. These are generally denoted by capital letters M,N,T etc. and we can carry out the judgement as
  a:A, a is term of Type A.

- **Arrow Types** : Also called as function type which take input of a type and return another. As we have discussed '+' has type $\mathbb{N} \to \mathbb{N}$. More generally if a function f takes input of type A and returns output of type B then, $f : A \to B$. In logic these are equivalent to implication. We will discuss about this correspondence later.

Which can be further extended as,

- **Product Types** Product types are similar to Cartesian Products from Set Theory, the terms inhabiting the product type is a pair of terms from the involving types, $a : A, b : B$ then $\langle a, b \rangle : A \times B$.

- **Sum Type** This is similar to disjoint union with sets, therefore the term has one of the participating types.

$$a : A, b : B, a : A + B, b : A + B$$

2.2.3 Simply Typed Lambda Calculus

We have already discussed about the Lambda Calculus as well as Types, now we assign a type to each lambda term. The functions in Untyped lambda Calculus are too general. In Example 2.5 we did not mention what squaring means, can we apply it to some arbitary type of data, what will the function return, these concerns remain unanswered. But when we assign every term a type, it can be assigned what can the function take and return.

- Variables x:T , a variable which has a type T

- Abstraction $\lambda x : M.t : T$, the function takes an input of type M and returns t of type T.

- Application $(\lambda x : M.t : T)(a : M)$

We present a typed version of Example 2.5

Example 2.3

$$\lambda x : \mathbb{N}.x^2 : \mathbb{N}$$

Note that now this function can only take natural numbers,
Application

$$(\lambda x : \mathbb{N}.x^2 : \mathbb{N})(3)$$

which gives, along with the judgement

$$3^2 : \mathbb{N}$$

2.2.4 Dependent Type Theory and Dependently Typed λ-Calculus

In this extension of theory of types, we allow types to depend upon some value. For example, modeling vectors with types, we require it to depend on Natural number for its dimension. u:Vec(n) means u is a vector type with n dimension and n:$\mathbb{N}$. This can be taken as a function type with $Vec : \mathbb{N} \to Type$, which takes a type of natural number as input and returns a type. This allows indexing of types and is useful for representing predicates. And similarly as with simply typed λ-Calculus we can have lambda terms with dependent types.

2.3 Proof Terms and The Curry Howard Correspondence

Let us treat every proposition or Logical Formula as a Type, and their proofs as the term inhabiting it. Intuitionstically, we interpret a proposition being true if we present a proof for it. Here, it will be true if there is a term (proof) inhabiting the type (proposition). p:P is now interpreted as p is a proof of P. In [**?**] the strong correspondence between intuitionstic derivation and lambda terms was mentioned. In essence the corrrespondence is as follows,

Logic	**Computation**	**Type Theory**
Proposition	Type	Type
Proof	Program	Term
Proof Normalization	Program Evaluation	Reduction
Implication ($A \to B$)	Function Type ($A \to B$)	λ-abstraction
Conjunction ($A \wedge B$)	Product Type ($A \times B$)	Pairing
Disjunction ($A \vee B$)	Sum Type ($A + B$)	Tagged union
Falsehood ($\bot$)	Empty Type	No term
Truth ($\top$)	Unit Type	Singleton term
Negation ($\neg A$)	$A \to \bot$	Function to empty type
Universal Quantifier ($\forall x.A(x)$)	Dependent Product ($\Pi x : A.B(x)$)	Function over types
Existential Quantifier ($\exists x.A(x)$)	Dependent Sum ($\Sigma x : A.B(x)$)	Pair with dependent type

Table 1: Curry–Howard Correspondence: Logic, Computation, and Type Theory

The use of dependent product and sum for the Quantifier highlights the necessity of Dependent Type Theory. The rules (Proof Terms only), equivalent to Deduction rules of Intuitionstic Logic are in Section 2.4 of [**?**]. Note that this does not contain the rule for predicates. While writing a proof in a proof assistant our reasoning follows deduction rules of this kind. Then to write a program (equivalent to proof) of the proposition $A \to B \to A$, we need to construct a function that takes term of type A , term of type B and returns term of type A, that is $\lambda a{:}A.\lambda b{:}B.t{:}A$ which proceeds as

$$\dfrac{\dfrac{\dfrac{}{a{:}A,\ b{:}B \vdash a : A}\ \text{var}}{a{:}A \vdash \lambda b{:}B.\ a : B \to A}\ \text{lam}}{\vdash \lambda a{:}A.\lambda b{:}B.\ a : A \to B \to A}\ \text{lam}$$