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

Bishesh Bohora, Supreme Chaudhary, Ashwot Acharya
Kathmandu University

July 30, 2025

Abstract

This report explores the theoretical foundations and working principles of proof assistants, with a focus on Agda. It introduces key concepts from logic, type theory, and the Curry–Howard correspondence that underlie formal verification. We discuss how proofs are treated as programs and propositions as types in dependently typed systems. A comparative study of proof assistants such as Agda, Coq (Rocq), and Lean is included, highlighting their kernel architectures and type-checking algorithms. Finally, we demonstrate formalization of selected logical proofs in Agda, reflecting on challenges and limitations encountered during implementation.

Contents

1 Introduction

In 1998 Thomas C. Hales announced that he had proved the kepler conjecture (Solane,1998) However the peer review took over 4 years especially due to the proof being incredibely difficult to check, with over a dozen of mathematician to refree the proof and although they were 99formalized that they were able to completely validate the correctness of the proof, which according to Thomas Hales, would have taken 20 person-year of manual work. [Szpiro, G, 2003]

Proof assistants helps with formalization of mathematical proofs in computer which enables for their verification digitally. Some exhaustive proofs may contain large number of cases which is not feasible to write and verify manually and requires computer assitance. Such Proof assistants (systems) work by treating proofs as computational objects and checking them using strict logical rules. To enable this, proofs must be written in a formal language that removes ambiguity and allows machines to interpret them accurately. Underlying this process is a theoretical framework—often type theory—that defines how propositions and proofs relate to each other. This method bridges programming and logic: writing a proof resembles writing a program, and verifying it is like type-checking. By interpreting logic computation- ally, proof assistants ensure that every proof is exact, complete, and verifiable by a machine. This project invloves investigating this process.

1.1 Objectives

1. To explore the correspondence between Proofs and Programs, Type Theory and Intuitionstic Logic.
2. To investigate a current exsisting proof assistants (Agda) and identify the mathematics behind it.
3. To formalize some selected proofs in Agda and demonstrate verification process.

1.2 Limitations

1. This study fails to rigorously present the Type Checking Algorithm.
2. Avoids formalization of complicated proofs.

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* $A \to \bot$
$(A \wedge (A \to \bot)) \to \bot$

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

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

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

Proposition 2.3

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

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

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 ??** 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

$$\begin{aligned}(\lambda x \lambda y.x^2 + y^2)(3)(4)\\ (\lambda y.3^2 + y^2)(4)\end{aligned}$$

$$3^2 + 4^2$$

$$9 + 16$$

$$25$$

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

Proposition 2.4 $A \to B \to A$

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

2.4 Martin-Löf Type Theory

Martin-Löf Type Theory is the formal logic system that was developed to support constructive mathematics. It's often referred to as Intuitionstic Type Theory. It carries the use of dependent types and is the backbone of Proof Assistants.
The following types are also valid in this system.

- $\prod$-Type : The dependent function type, as mentioned in Table 1. it supports the interpretation of Universal Quantifier.
- $\sum$-Type : The dependent sum type, equivalent to the Existential Quantifier.

- Identity Type : Equality itself is internalized as a type in this system. Hence, we can judge if two terms are the same. By the correspondence, proofs are terms then this allows proof assistants to match proofs and declare their Equality.

Furthermore, We also have **Type Universes**, Every Type itself can be assigned a Type. This creates a hierarchy of types. For example, $1 : \mathbb{N}, \mathbb{N} : Type : ...$, The number 1 has type Natural, then Natural has a type called Type, which itself can have another type. Not to be interpreted as , a set being contained in another set, this gives rise to paradoxes. But rather take it as hierarchy of Universes. From this we can talk about Types themselves within our system, and allow proof assistants to safely handle abstraction over types.
The results from [**?**] where this system was first introduced by Martin Löf, will be mentioned as necessary.

.

3 General Architecture of Proof Assistants

The proof assistants follow a general architecture that adopts a design that seperates tiny,trusted kernel from progressively less trusted outer layer[]

User Interfaces (VS Code, Proof General, CoqIDE, etc.)
Elaborator / Front-End
Tactic Engine & Automation
Kernel (Type Checker + Inference Rules)
Logical Foundations (CIC, HOL, MLTT, . . .)

Figure 1: Standard proof-assistant layer cake.

3.1 The kernel

At the heart of any proof assistant lies the kernel, which is a minimal and formally verified component responsible for enforcing the programming logical rules. kernel implements the foundational calculus or logic and validates every proof step by checking derivations against these rules. The kernel does't trust the computing base and checks for every proof , which ensures that the entire system depends on this critical component [Gordon, 1993][Paulson, 1987][The Coq Development Team, 2025].

3.2 Tactic Engine

Above the kernel layer is the tactic engine that assists users in constructing proofs interactively or automatically. Tactics provide programmable strategies to decompose proof goals into simpler subgoals which can be easier to prove. Tactic implementations can become complex, but since any output they generate must be verified through the kernel before acceptance,system integrity is maintained [Nipkow et al., 2002][Delahaye, 2000].

3.3 Formal Proof language

Proof assistants offer a dedicated programming language designed to express definitions, statements and proofs. These languages are typically Dependently typed, and allows

implicit argument . they are highly readable and rigorous. Integration with tactic scripting enables a smooth blend of automated and manual proof development [Barras et al., 1997][Agda Team, 2023][Lean Development Group, 2024].

3.4 Type checker and inference engine

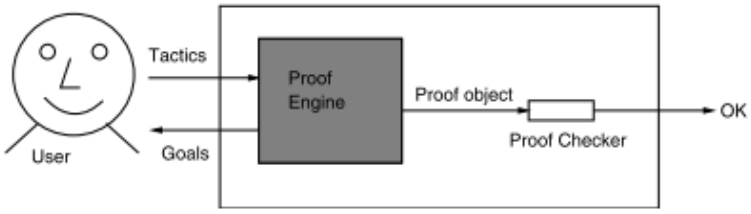
The type checker enforces logical consistency by validating the written expression , checking if it meets the system's typing rules ,importantly preventing paradoxical constructs. Proof assistant often use bi-directional type checking and other complex inference algorithms.

3.5 User Interface

Effective user interfaces (UIs) enable seamless interaction with the proof assistant. Modern UIs include integrated development environments (IDEs) or editor plugins that provide syntax highlighting, error diagnostics, real-time proof state visualization, and automated tactic suggestion. Such interfaces significantly enhance productivity and make formal methods more accessible to a wider audience [CoqIDE, 2025][Isabelle/jEdit, 2023].

3.6 Formal Libraries

Proof assistants come with extensive, rigorously verified libraries that encapsulate fundamental theories (e.g., arithmetic, set theory, algebra), supporting rapid development of complex formalizations without reinventing foundational results. These libraries grow continuously and provide a shared repository of reusable components vital for large-scale verification projects [Coq Standard Library, 2025][Mathlib, 2025][Isabelle Archive of Formal Proofs].



[?]

4 Upon Some Proof Assistants and Comparative Study

4.1 Agda

Agda is a Dependently typed functional programming language and proof assistant. Extension of Martin-Lof's Type theory. (agda documentation) In Dependently typed programming language there is no clear distinction between types and values , and types can depend on arbitary values and may appear as results of ordinary functions. This feature of Dependently typed programs allows us to encode properties of values as type of proofs of that property, allowing us to use Dependent type programming as logic. (Dependently Typed Programming in Agda)

Every agda file has a top level module which corresponds to name of the file and the main programs goes inside the top level module.[] Another key component of agda is pattern matching. Pattern matching is a mechanism for analysing the structure of values. When inductively defined datatypes are introduced the pattern matching becomes even more powerful as patten matching on one value can give the information about another value (thesis)

4.1.1 Kernel

Agda's Kernel is written in Haskell. With no seperate tactic layer, the kernel is directly involved in all proof construction making it minimalistic yet more rigid. It tightly integrates normalization, which is a direct application of the Normalization Theorem in [**?**].

4.1.2 Type Checking algorithm

Agda uses a bidirectional type checking algorithm grounded in Martin-Löf Type Theory, emphasizing manual proof construction and transparency. The checker alternates between inferring types for expressions and checking expressions against known types, supported by normalization by evaluation (NbE) to verify definitional equality. Unlike Lean or Coq, Agda offers minimal automation—there are no tactics, typeclass resolution, or automatic proof search. This makes it more predictable but also more labor-intensive. In contrast, Lean and Coq automate many aspects of type inference and proof construction via elaborators and tactics, with Coq relying on a trusted kernel and Lean integrating extensible unification and metaprogramming. Agda prioritizes explicitness and formal clarity, making it ideal for foundational studies in type theory. [**?**]

4.2 Rocq

Rocq formerly known as coq is another proof assistant or interactive theorem prover and lets you formalize mathematical proofs. becoming famous for the Formalization of four color theorem . Rocq is vastly different from agda in terms of proofs, While Agda is type theory based direct construction proof assistant Rocq on the other hand is tactic based. In Rocq users generate proofs by entering a series of tactics, which constitute steps of proofs.

4.2.1 Tactics

Mainly there are two ways of generating proofs forward reasoning and backward reasoning
In forward reasoning the simpler and smaller statements are proved and constructed to bigger statements and theorem, combining to prove the theorem at the end
In Backward reasoning proofs begins with theorem statement and gradually transformed into sub-statement or goals which are proven. Rcoq and Tactics heavily relies on Backward reasoning models. A tactic by itself may fully prove a goal or divide into sub goals to prove and most tactic in rocq require certain condition or elements to reduce a goal. some tactic are automatic and can solve complex goals
(doc website)

4.2.2 Kernel

Rocq's kernel is based on Predicative Calculus of Cumulative Inductive Constructions(PCUIC) , a Dependently typed lambda calculus extended with various other rules and features. Rocq's core kernel converts all the notations and implicit arguments into core language which is the calculus of inductive construction. the kernel wil verify the proof term built by the tactics . This type of seperation is called the "de Bruijn criterion" and makes it so that only the smaller components like the kernel is trusted and the entire program is checked by the kernel. Furthermore, its written in Coq itself (extracted to OCaml) i.e a Kernel being a program itself, its correctness is verified within it.

4.2.3 Type checking Algorithm

In Rocq the syntaxes and typing rules are specified so that they remain mathematicaly precise and machine verifiable. For that the type system is first described declaratively, then refined to algorithmic specification. Rocq introduces a bi-directional infer and check type system that guides implementation.The checker is implemented in coq itsel. Much of type checking relies on deciding if two tpes are equivalent by Conversion or if one is the subtype of other by cumulatively which is very challenging because these relations depend on reduction. The type checker is constructed as bi-directional algorithm and to ensure soundess the checker carries proof carrying code and returns the proof along with the type checking decision. Every critical theoretic properity need for type checking is proved or assumed axiomatically to be true. The bidirectional type checking employed by Rocq is weak compared to Agda but it relies heavily on conversion. (Correct and Complete Type Checking and Certified Erasure for Coq, in Coq)

4.3 Lean

Lean theorem prover is a proof assistant developed by Leonardo Di-Moura at Microsoft in 2013. one way to prove theorem in lean is to build from assertion which follows the calculus of construction. Lean treats any two element say $t1\ t2 : p$ to be equal, which is known as proof irrelevance. And allows for proposition as types logic. calculational proofs can also be written in lean and starts with the word calc.

Another way Like Rocq Lean can also relie on tactics. Tactics are useful for construction and destruction of data and there are many built in tactics avaible in lean. One of the major advantage of lean is that you can mix term-style and tactic-style proof writing and pass freely between the two. The ease of working with Lean is responsible for its popularity against other Proof Assistants.

4.3.1 Kernel

Lean's kernel is written in C++ (Lean 3) and C (Lean 4). The kernel much like Coq is based of Calculus of Inductive Constructions and employs conversion checking. Though its elaboration and tactic system is large, it is relatively less rigid but improves usability. Its performance based and offers flexibility. Lean's kernel is implemented in two layers. The First layer contains the type checker and APIs for creating and manipulating terms, declarations, and the environment and the second layer provides additional components such as inductive families and quotient types. When the kernel is instantiated, one selects which of these components should be used. [**?**]

4.3.2 Type Checking Algorithm

Likewise,the Bidirectional algorithm is applied here too. But the strength of Lean lies in its smart elaboration. Lean offers coercion, backtracking and overloading.

4.4 The Bidirectional Type Checking Algorithm

At its core, this algorithm works on two modes

- Checking Mode ($\Gamma \vdash M \leftarrow A$) Given the term M and an expectied type A , the task is to verify that M has type A. Its done when the type is known or can be known then propagated from the context.
- Synthesis (Inference) Mode ($\Gamma \vdash M \rightarrow A$) When the type is known, the job is to infer it. That is synthesize unique type A for the term M. Since the type is not available from the context Γ , it needs to be determined upwards.

The checking and infering in two directions (from or to context Γ) led it to be called Bidirectional .

5 Formalization of Some Proofs

5.1 Defining Natural Numbers, Equality and Proof of Assocaitive and Commutative Properties

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

{-# BUILTIN NATURAL N #-}


--Equality
data _==_ { A : Set } ( x : A ) : A -> Set where
    refl : x == x

cong : {A B : Set } ( f : A -> B ) { x y : A}
    -> x == y
    -> f x == f y
cong f refl = refl
sym : forall { A : Set } { x y : A}
    -> x == y
    -> y == x
sym refl = refl

trans : { A : Set } { x y z : A}
    -> x == y
    -> y == z
    -> x == z

trans refl refl = refl

{-# BUILTIN EQUALITY _==_ #-}

infix 4 _==_


--ADDITION

_+_ : N -> N -> N
zero + x = x
suc x + y = suc(x + y)


--ZERO
zero-+ : ( a : N ) -> ( ( zero + a ) == a )
```

```
zero-+ a = refl

+-zero : ( a : N ) -> ( ( a + zero ) == a )
+-zero zero = refl
+-zero (suc a) = cong suc (+-zero a)

--LEMMA FOR COMMUTATIVITY
+-suc : forall x y -> x + suc y == suc (x + y)
+-suc zero y = refl
+-suc (suc x) y = cong suc (+-suc x y)


--ASSOCIATIVITY
assoc-+ : forall x y z -> (x + y) + z == x + (y + z)
assoc-+ zero y z = refl
assoc-+ (suc x) y z = cong suc (assoc-+ x y z)

--COMMUTATIVITY
+-comm : ( a b : N) -> ( ( a + b ) == ( b + a ) )
+-comm a zero = +-zero a
+-comm a (suc b) = trans (+-suc a b) (cong suc (+-comm a b))
```

5.2 Formalizing Proposition 2.3 and 2.4

```
module props where

open import Agda.Primitive using (Level; lzero)

open import Data.Product using (Sigma; _,_)

variable
  l : Level
  X : Set l
  A B : X → Set l
  s : X


prop23 : (forall x → A x → B x) → (s : X) → A s → Sigma X B
prop23 allImp s aS = s , allImp s aS
--Replace sigma with greek alphabet while loading.


prop24 : forall {A B : Set} -> A -> B -> A
prop24 a b = a
```

5.3 DeMorgan's Law

```
    open import Agda.Primitive using (Level; lzero)
open import Data.Product using (_×_; _,_)
open import Data.Sum using (__; inj; inj)
open import Relation.Nullary using (Dec; yes; no)
open import Data.Empty using (; -elim)
open import Relation.Nullary.Negation using (¬_)


--One Direction
deMorganOneWay :  {} {P Q : Set } → (¬ P)  (¬ Q) → ¬ (P × Q)
deMorganOneWay (inj np) (p , q) = np p
deMorganOneWay (inj nq) (p , q) = nq q

--Converse, Requires Non Constructive Assumptions
deMorganOtherWay :
   {} {P Q : Set }
  → Dec P
  → Dec Q
  → ¬ (P × Q)
  → (¬ P)  (¬ Q)
deMorganOtherWay (yes p) _ notPQ = inj ( q → notPQ (p , q))
deMorganOtherWay (no np) (yes q) _ = inj np
deMorganOtherWay (no np) (no nq) _ = inj np  -- or inj nq
```

5.4 Euclid's Theorem of Primes

5.5 Euclid's Algorithm's Proof

5.6 Attempt to Constructive Reals

6 Challenges and Workarounds

References

[Alrubyli and Yazeed, 2021] Alrubyli and Yazeed (2021). Natural deduction calculus for first-order logic.

[Geuvers, 2009a] Geuvers, H. (2009a). Introduction to type theory. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5520 LNCS, pages 1–56.

[Geuvers, 2009b] Geuvers, H. (2009b). Proof assistants: History, ideas and future.

[Howard, 1969] Howard, W. H. (1969). The formulae-as-types notion of construction. Technical report.

[Martin-Löf, 1972] Martin-Löf, P. (1972). An intuitionistic theory of types. Unpublished preprint; included in *Twenty-Five Years of Constructive Type Theory*, OUP 1998. https://archive-pml.github.io/martin-lof/pdfs/An-Intuitionistic-Theory-of-Types-1972.pdf.

[Moura et al.,] Moura, L. D., Kong, S., Avigad, J., Doorn, F. V., and Raumer, J. V. The lean theorem prover (system description). Technical report.

[Norell and Chapman,] Norell, U. and Chapman, J. Dependently typed programming in agda. Technical report.

[Pfenning, 2004] Pfenning, F. (2004). This includes revised excerpts from the course notes on linear logic (spring 1998) and computation and deduction. Technical report.

[Rojas, 2015] Rojas, R. (2015). A tutorial introduction to the lambda calculus.