Interactive Theorem Proving in Coq and the Curry-Howard Isomorphism

Abhishek Kr Singh

January 7, 2015

Abstract

There seems to be a general consensus among mathematicians about the notion of a correct proof. Still, in mathematical literature, many invalid proofs remain accepted over a long period of time. It happens mostly because proofs are incomplete, and it is easy to make mistake while verifying an incomplete proof. However, writing a complete proof on paper consumes a lot of time and effort. *Proof Assistants*, such as *Coq* [8, 2], can minimize these overheads. It provides the user with lots of tools and tactics to interactively develop a proof. Most of the routine jobs, such as creation of proof-terms and type verification, are done internally by the proof assistant. Realization of such a computer based tool for representing proofs and propositions, is made possible because of the *Curry-Howard Isomorphism* [6]. It says that, proving and programming are essentially equivalent tasks. More precisely, we can use a programming paradigm, such as *typed lambda calculus* [1], to encode propositions and their proofs. This report tries to discuss curry-howard isomorphism at different levels. In particular, it discusses different features of *Calculus of Inductive Constructions* [3, 9], that makes it possible to encode different logics in Coq.

Contents

Chapter 1

Introduction

1.1 Proof Assistants

In the discipline of Mathematics, we often come across assertions, of which we are mainly interested in knowing the truth value. To be sure that a given statement is true, we rely on a proof. Normally, proofs are given in natural languages, which are full of ambiguities.

A proof of a proposition is an argument, that can convince anyone of the correctness of the proposition. To be of any use, a proof should always be finite. We often omit many small steps of reasoning, to reduce the size of a proof. In doing so, some steps of invalid reasoning might be introduced. Thus, verifying the correctness of a proof becomes, a very important task. However, the correctness of a proof can only be verified, if the proof is complete and unambiguous. These requirements are hardly satisfied, even in the scientific literature, due to the mentioned limitation of natural languages and the huge size of a complete proof.

A possible solution to this problem, is to define a formal language for writing proofs, which reflects the precise rules of proof theory [10]. This will ensure, that every proof can be verified step by step. However, the complete proof of a theorem, in such formal languages, quickly becomes a huge text. It is almost impractical, to verify them manually. It becomes necessary, to mechanize their verification. In this sense, a *Proof Assistant* like *Coq* [8], becomes very meaningful. It is a system in which, mathematicians can develop proofs, in a very expressive formal language. These proofs are built in an interactive manner, where given a proposition that one wants to prove, system proposes tools to construct the proof. These tools are called *tactics*, and they provide ways to prove propositions under some assumptions. In many cases, tactics may also help, in constructing proofs automatically. However, this is not always true.

1.2 The Coq Proof assistant

The Coq system, belongs to a large family of computer based tools such as Automath, Nqthm, Mizar etc, whose purpose is to help in proving theorems. Coq implements Gallina [8], a program specification and mathematical higher level language. Gallina is based on Calculus of Inductive Constructions [8], which itself is a very expressive variation on typed λ -calculus [1]. Through a language of commands, called Vernacular, Coq allows us to define terms, types, predicates, and state mathematical theorems. It also helps user, to interactively develop formal proofs of these theorems. In Coq, one can also define functions, that can be evaluated efficiently. Computation in Coq, is performed by successive reductions of terms to an irreducible form. An important feature of Coq is that, computation always terminates (strong normalization). However, classical results in computability theory show that, a programming language that makes it possible to describe all computable functions, must contain functions whose computations do not terminate. For this reason, there are computable functions that can be described in Coq, but for which computation can not be performed by the reduction mechanism.

1.3 An overview of the report

This report is divided into five chapters, including the present chapter. In the second chapter, we introduce basic notions, such as *terms* and their *types* of Coq. We also glance at the proof theoretic view of the logical connectives in Coq, and briefly talk about the *natural deduction* [10] style of proofs. Chapter 3, discusses in some detail, the type theory underlying Coq. It tries to explain the *Curry-Howard isomorphism* [6], between different levels of logic and type theory. This chapter ends with identifying the *Pure Type System* (PTS), which is close to the underlying type theory of Coq. We discuss the idea of *Inductive types*, seperately in Chapter 4. This chapter avoids the detailed discussion of different tactics, used while reasoing with inductive type. Instead it discusses, in some detail, the *induction principle* which is automatically generated by an inductive definition. The last chapter concludes the report, with some observations and suggestions.

Chapter 2

Terms, Computations and Proofs in Coq

This chapter introduces the notions like *terms*, *environment*, *context*, *theorems* and *proofs* in Coq. It also discusses very briefly the notion of computation which is more popularly known as *reduction* in Coq. The discussion here is at very elementary level and avoids the precise definitions of these notions. For a precise definition of these notions reader should refer to [8].

2.1 Terms and Types

Types and Expressions

Expressions in the specification language of Coq, Gallina, are formed with constants and identifiers using a few construction rules. Every expression has a type. There is no syntactic distinction made between Expressions and types in Coq. Expressions and types are together referred as terms. The notion of term covers a very general syntactic category that corresponds to the intuitive notion of a well-formed expression. The type for an identifier is usually given by a declaration. The type of a combined expression is determined from the type of its part by using typing rules.

A very important feature of Calculus of Inductive Construction (CIC) is that every type is also a term and also has a type. The type of a type is called a *sort*. `Set` and `Prop` are predefined sorts of CIC. The `Set` and `Prop` sorts are terms in the CIC and must in turn have a type, but this type, itself a term, must have another type. The CIC considers an infinite hierarchy of sorts called *universes*. This family is formed with types `Type(i)` for every $i \in N$ such that `Set:Type(i), Prop:Type(i)` and `Type(j):Type(k)` if `j<k`.

Identifiers, Environments and Context

In Coq, type-checking is done with respect to an environment, determined by the declaration and definitions that were executed earlier. A *declaration* is used to attach a type to an identifier, without giving the value. For example, the declaration of an identifier x with type A is written $(x : A)$. On the other hand, a *definition* gives a value to an identifier by associating a well-formed term. Since this term should have a type, a definition also gives a type to the identifier. Defining an identifier x with a value t is written $(x := t)$.

The scope of a definition or a declaration can be either *global* or *local*. In the former case, the scope is the rest of development; in the later case, the scope is restricted to the current *section*. The role of a section is to limit the scope of parameters, definitions and declarations. In Coq, sections define a block mechanism, similar to the one found in programming languages. It also gives us the convenience to reset a part of the development by closing the current section. At any point in a development, we have a current *environment* E and a current *context* Γ. The environment contains the global declarations and definitions, while the context is formed with the local declarations and definitions. This chapter uses the name *variable* to describe locally defined or declared identifier, *global variable* for a globally declared identifier and *constant* for a globally defined identifier.

A context Γ usually appears as a sequence of declarations, presented as $[v_1 : A_1; v_2 : A_2; \ldots; v_n : A_n]$. The notation $(v : A) \in \Gamma$ is used to express that the variable v is declared with type A in the context Γ. Adding a declaration $(v : A)$ to a context Γ is denoted as $\Gamma :: (v : A)$. We use notation $E, \Gamma \vdash t : A$ for *typing judgment* and read it as " in the environment E and the context Γ , the term t has type A."

A type, A, is said to be *inhabited* in an environment E and a context Γ if there exists a term t such that the judgment $E, \Gamma \vdash t : A$ holds. The notation $? : A$ corresponds to the question " Given A, does there exists a t such that $\vdash t : A$?". This is referred as *inhabitation problem*. The question "Given t and A, does one have $\vdash t : A$?" is known as *type checking* and expressed as $t : A?$. The notation $t :?$ is used to ask "Given t, does there exist an A such that $\vdash t : A$?". This is known as *typability* problem.

Typing Rules

This section briefly discusses the rules to construct a subset of the well formed terms in Coq which corresponds to the *simply typed* $\lambda-$ *calculus*. Here, types have two forms: 1) *Atomic types*, made of single identifier, like `nat` and `bool`. These are also known as *type variables* in simply typed λ- calculus. 2) Types of the form $(A \to B)$, where A and B are themselves types, is known as *arrow types*. These are special case of a more general construct called *dependent product*, which is explained in detail in the next chapter.

- **Identifiers:** The simplest form of an expression is an identifier x. If $(x : A) \in E \cup \Gamma$ then x has a type A. This typing rule is usually presented

as an *inference rule*: **Var** $\dfrac{(x : A) \in E \cup \Gamma}{E \cup \Gamma \vdash x : A}$.

- **Abstraction:** In simply typed $\lambda-$ calculus notation $\lambda v : A.e$ is used to represent a function which associates the expression e to the variable v. The type inference rule: **Lam** $\dfrac{E, \Gamma :: (v : A) \vdash e : B}{E, \Gamma \vdash \lambda v : A.e : A \to B}$.
- **Function Application**: The expression "$e_1 e_2$" represents function application, where e_1 is said to be in the function position while e_2 is considered as argument. Typing rule is: **App** $\dfrac{E, \Gamma \vdash e_1 : A \to B \quad E, \Gamma \vdash e_2 : A}{E, \Gamma \vdash e_1 e_2 : B}$.
- **Local Bindings**: A local binding is written `let` $v := t_1$`in` t_2 , where v is an identifier and t_1 and t_2 are expressions. It avoids repeated code and computation, by using local variables to store intermediate results. Typing rule for this construct is: **Let-in** $\dfrac{E, \Gamma \vdash t_1 : A \quad E, \Gamma :: (v := t_1 : A) \vdash t_2 : B}{E, \Gamma \vdash let\, v := t_1\, in\, t_2 :\ B}$.

2.2 Computations

Computation in Coq is performed by a series of elementary term transformation, which is known as reductions in $\lambda-$calculus. It identifies reduction patterns, known as *redex*, and reduces it until they are no longer available. The result term is called *normal form*. In Coq, with `Eval` command, one can see the results of *normalizing*. Different options can be used with this command to decide what strategy to follow. One can use `cbv` strategy "call-by-value", to indicate that arguments of functions should be reduced before the function, on the other hand `lazy` strategy can be used to mean that arguments of functions should be reduced as late as possible.

If t and u are two terms and v is a variable we represent by $t[v := u]$ the term obtained by replacing all free occurrences of v by u in t, with the right amount of variable renaming to avoid binding any free variables of u in the resulting expression. This operation is known as *substitution* and is very necessary to describe different kinds of reductions in Coq.

There are four kinds of reductions used in Coq.

- δ- **reduction** is used to replace an identifier with its definition. If t is a term and v an identifier with value $t^{'}$ in the current context, then δ-reducing v in t will transform t into $t[v := t^{'}]$ .Notation $t \to_\delta t[v := t^{'}]$ is used for the same.
- β- **reduction** makes it possible to transform a term of the form "$(\lambda v : T.e_1)e_2$" into the term $e_1[v := e_2]$. It is represented as $(\lambda v : T.e_1)e_2 \to_\beta e_1[v := e_2]$.
- ζ- **reduction** is concerned with transforming local bindings into equivalent forms that do not use the local binding construct. More precisely it is represented as `(let` $v := e_1$ `in` $e_2) \to_\zeta e_2[v := e_1]$.

- ι- **reduction** is related to inductive objects and is responsible for computation in recursive programs.

Combination of the above mentioned reductions enjoy very important properties. Every sequence of reductions from a given term is finite. This property is called *strong normalization*. It assures that every computation on a term always terminates. Moreover, if t can be transformed into t_1 and t_2, probably using two different sequence of reductions, then there exists a term t_3 such that t_1 and t_2 can both be reduced to t_3. This is called the *confluence property*. An important consequence of these two properties is that any term t has a unique normal form with respect to each of the reductions.

Two terms are called to be *convertible* if they can be reduced to the same term. In CIC this property of *convertibility* is decidable. To decide whether t_1 and t_2 are convertible, it is enough to compute their normal forms and then to compare these normal forms modulo bound variables renaming.

2.3 Propositions and Proofs

The type of a proposition in Coq is called Prop. Moreover we have predicates to build parametric propositions. These propositions can be considered as functions while determining their type. For example, the predicate "to be a sorted list" has type (list Z)→Prop. Even more complex predicates can be formed in Coq because arguments may themselves be predicates. For example, the property of being a reflexive relation on Z is a predicate of type (Z→Z→Prop)→Prop.

In above example we have already seen that propositions can refer to data. The Coq type system also makes the converse possible. The type of a program can contain constraints expressed as propositions that must be satisfied by data. This is called a dependent type in Coq.

Most of these issues related to dependent types are discussed in detail in the next chapter. The details of how the proofs are represented using the underlying type theory of Coq is the subject of next chapter. This section discusses a proof style, called natural deduction which is the underlying reasoning technique used in Coq.

Intuitionistic Logic and Natural deduction

The understanding of propositions and their validity in Coq is not the same as in classical logic. The classical approach, proposed by Tarski [11], assigns to every variable a denotation, a truth value $\boldsymbol{t}$ (true) or $\boldsymbol{f}$ (false). One considers all possible assignments, and the formula is true if it is true for all of them. Statements are either true or false. Here "false" means the same as "not true" and this is expressed by the *principle of excluded middle* that "$p \vee \neg p$" must hold no matter what the meaning of p is. For example, consider the following statement:

"There are two irrational number x and y, such that x^y is rational."

The proof of this fact is very simple: if $\sqrt{2}^{\sqrt{2}}$ is a rational then we can take $x = y = \sqrt{2}$; otherwise take $x = \sqrt{2}^{\sqrt{2}}$ and $y = \sqrt{2}$.

However, the problem with this proof is that we do not know which of the two possibilities is the right one. There is very little information in this proof, because it is not *constructive*.

Coq's understanding of proposition and its validity is based on the Intuitionistic logic which believes in finding a proof or "construction" of the given proposition. This approach was advocated by Brouwer, Heyting and Kolmogorov and hence commonly known as *BHK-interpretation.*

The following rules explain the informal constructive semantics of propositional connectives in BHK-interpretation:

- There is no possible construction of $\perp$(where $\perp$ denotes falsity).
- A construction of $\varphi_1 \wedge \varphi_2$ consists of a construction of φ_1 and a construction of φ_2;
- A construction of $\varphi_1 \vee \varphi_2$ consists of a number $i \in \{1, 2\}$ and a construction of φ_i;
- A construction of $\varphi_1 \rightarrow \varphi_2$ is a method (function) transforming every construction of φ_1 into a construction of φ_2 ;
- A construction of $\neg\varphi$is a construction of $\varphi \rightarrow \perp$.

Coq implements a proof style, called *natural deduction*, which tries to formalize the above interpretations by defining introduction and elimination rules for each logical connectives. We will use the notation $\Delta \vdash \varphi$ to mean that φ is a consequence of the assumptions in the context Δ.

Consider the inference rules: $\dfrac{\Delta \vdash \varphi \quad \Delta \vdash \psi}{\Delta \vdash \varphi \wedge \psi}\,(\wedge I)$, which is the *introduction rule* for Conjunction. This tells us that given $\Delta \vdash \varphi$ and $\Delta \vdash \psi$ we can deduce $\Delta \vdash \varphi \wedge \psi$. Introduction rules explain how to conclude a given proposition. For a propositional connective it tells us how to prove a goal whose main operator is the given propositional connective. On the other hand *elimination rules* for a propositional connective tells us how to use a hypothesis whose main operator is the given connective. It justifies how we can use a given proposition and what consequences we can derive from it. For example, knowing that $\Delta \vdash \varphi \wedge \psi$ one can deduce $\Delta \vdash \varphi$ and $\Delta \vdash \psi$ as well. This is concisely written as $\dfrac{\Delta \vdash \varphi \wedge \psi}{\Delta \vdash \varphi}(\wedge E)\dfrac{\Delta \vdash \varphi \wedge \psi}{\Delta \vdash \psi}$, which is the elimination rule for conjunction. The table 2.1 lists introduction and elimination rules for other logical connectives.

When the context is clear we can avoid writing Δ and the premises and conclusions of these inference rules can equivalently be represented in a simplified form. The corresponding simplified inference rules looks as in table 2.2.

These inference rules may help us build our intuition about the interactive theorem proving environment of Coq. However, these are not the exact way in which Coq assigns meaning to these connectives. The underlying type theory of

Introduction rule	Elimination rule
$\dfrac{\Delta \vdash \varphi \quad \Delta \vdash \psi}{\Delta \vdash \varphi \wedge \psi}\ (\wedge\mathrm{I})$	$\dfrac{\Delta \vdash \varphi \wedge \psi}{\Delta \vdash \varphi}(\wedge\mathrm{E})\dfrac{\Delta \vdash \varphi \wedge \psi}{\Delta \vdash \psi}$
$\dfrac{\Delta \vdash \varphi}{\Delta \vdash \varphi \vee \psi}\ (\vee\mathrm{I})\dfrac{\Delta \vdash \psi}{\Delta \vdash \varphi \vee \psi}$	$\dfrac{\Delta, \varphi \vdash \rho \quad \Delta, \psi \vdash \rho \quad \Delta \vdash \varphi \vee \psi}{\Delta \vdash \rho}(\vee\mathrm{E})$
$\dfrac{\Delta, \varphi \vdash \psi}{\Delta \vdash \varphi \to \psi}(\to\mathrm{I})$	$\dfrac{\Delta \vdash \varphi \to \psi \quad \Delta \vdash \varphi}{\Delta \vdash \psi}\ (\to\mathrm{E})$
	$\dfrac{\Delta \vdash \bot}{\Delta \vdash \varphi}(\bot\mathrm{E})$

Table 2.1: Introduction and Elimination rules

Introduction rule	Elimination rule
$\dfrac{\varphi \quad \psi}{\varphi \wedge \psi}\ (\wedge\mathrm{I})$	$\dfrac{\varphi \wedge \psi}{\varphi}(\wedge\mathrm{E})\dfrac{\varphi \wedge \psi}{\psi}$
$\dfrac{\varphi}{\varphi \vee \psi}\ (\vee\mathrm{I})\dfrac{\psi}{\varphi \vee \psi}$	$\dfrac{\overset{[\varphi]^u}{\rho} \quad \overset{[\psi]^v}{\rho} \quad \varphi \vee \psi}{\rho}(\vee\mathrm{E})^{u,v}$
$\dfrac{\overset{[\varphi]^u}{\psi}}{\varphi \to \psi}(\to\mathrm{I})^u$	$\dfrac{\varphi \to \psi \quad \varphi}{\psi}\ (\to\mathrm{E})$
	$\dfrac{\bot}{\varphi}\ (\bot\mathrm{E})$

Table 2.2: Simplified Introduction and Elimination rules

Logical Connectives	Hypothesis H	Conclusion
$\rightarrow$	`apply H`	`intros`
$\forall$	`apply H`	`intros`
$\wedge$	`elim H` , `destruct H as [H1 H2]`	`split`
$\neg$	`elim H`	`intros`
$\exists$	`destruct H as [x H1]`	`exists` v
$\vee$	`elim H`, `destruct H as [H1 | H2]`	`left` or `right`
`False`	`elim H`	

Table 2.3: Tactics for logical connectives

Coq first defines $\rightarrow$ and $\forall$. It then turns out that other connectives are *second order definable* with respect to $\rightarrow$and $\forall$. This will be discussed in more detail in coming chapters.

In Coq, a user can enter a statement that he wants to prove, using the command `Theorem` or `Lemma`, at the same time giving it a name for later reference. In most cases a proposition is proved under the assumption of other propositions, which are entered as `Hypothesis`. Any proof in Coq is *goal directed*. User can enter a command to decompose the goal into simpler goals. These commands are called *tactics* and they realize the introduction and elimination rules for different logical connectives in Coq. A detailed discussion on these tactics and their use can be found in [2]. Table 2.3 summarises the tactics, corresponding to different logical connectives, which are commonly used in Coq. These tactics are used either to introduce a connective present in conclusion or to eliminate it when present in `Hypothesis H`.

Local soundness and completeness

Once we have seen the introduction and elimination rules for different connectives, one must verify whether they make sense, since the intended meaning for a connective can not be just defined by any pair of these rules. The rules should fit together, in a sense that they must meet certain conditions. In particular, they should not allow us to deduce new truths (soundness) and they should be strong enough to obtain all the information contained in a connective (completeness). More precisely one can define:

- Local Soundness: Elimination rules should not be too strong with respect to the corresponding introduction rule, i.e. they should not allow us to derive more information than we should. If we introduce a connective and then immediately eliminate it, we should be able to *erase* this detour and find a more direct derivation ending in the conclusion.

- Local Completeness: Elimination rules should not be too weak with respect to the corresponding introduction rule, i.e. they should allow us to

conclude everything we should be able to. A connective can be eliminated in such a way that it retains sufficient information to reconstitute it by an introduction rule.

For example consider the following *local reductions*, in proof size for A and B:

$$\dfrac{\dfrac{\overset{D_1}{A} \quad \overset{D_2}{B}}{A \wedge B}(\wedge I)}{A}(\wedge E) \implies \overset{D_1}{A} \text{ and symmetrically } \dfrac{\dfrac{\overset{D_1}{A} \quad \overset{D_2}{B}}{A \wedge B}(\wedge I)}{B}(\wedge E) \implies \overset{D_2}{B}.$$

Here, in the first case, we are using the information $A \wedge B$ to conclude A using elimination rule for conjunction. However, due to the introduction rule for conjunction, A and B must already have constructions for them in order to introduce $A \wedge B$. Let D_1 and D_2 be constructions for A and B respectively. Therefore the conclusion, in this case A, has a shorter proof D_1 which avoids the detour via $A \wedge B$. A similar reduction in the proof size of B can be achieved as suggested. These reductions in proof size demonstrate the fact, that everything that can be deduced using elimination rule for conjunction already has a shorter proof, and hence proves the soundness of introduction-elimination pair for conjunction.

On the other hand the following *local expansion* shows that after eliminating $A \wedge B$ we have enough information to rebuild a proof for $A \wedge B$. This shows completeness of the elimination rule with respect to introduction rule for conjunction.

$$\overset{D}{A \wedge B} \implies \dfrac{\dfrac{\overset{D}{A \wedge B}}{A}(\wedge E) \quad \dfrac{\overset{D}{A \wedge B}}{B}(\wedge E)}{A \wedge B}(\wedge I)$$

Consider the following local reduction which proves the soundness for implication elimination rule:

$$\dfrac{\dfrac{\overset{[A]^u}{\overset{D}{B}}}{A \to B}(\to I^u) \quad \overset{E}{A}}{B}(\to E) \implies \overset{D[u:=E]}{B}$$

Now we can check whether the elimination rules for implication are strong enough to get all the information out they contain. Following local expansion shows this:

$$\overset{D}{A \to B} \implies \dfrac{\overset{D}{A \to B} \quad [A]^u}{B}(\to E)}{A \to B}(\to I^u)$$

Continuing in the same way we can show soundness and completeness for other connectives as well by giving local reductions and expansions.

Chapter 3

Propositions as types correspondence in Coq

In the previous chapter we have seen a subset of typing rules in Coq, which corresponds to the simply typed lambda calculus. We also looked at the natural deduction style of defining logical connectives using introduction and elimination rules. This chapter starts by showing the connection between the two, which is known as *Curry-Howard isomorphism*. This isomorphism is also known as *propositions-as-types* interpretation. It amounts to two readings of typing judgments $M : A$

- M is a term of the type A
- M is a proof of the formula A

In Coq, terms and their types are the only objects of interest. This isomorphism is the key notion in understanding how the introduction and elimination rules for different logical connectives can be interpreted as typing rules in the underlying type theory of Coq. Thus, one can also see it as encoding different logics using type theory.

This chapter introduces several systems of typed lambda calculus and explains how to encode different logics using them. Encoding of minimal propositional logic (propositional logic with only $\rightarrow$) using Simply typed lambda calculus ($\lambda \rightarrow$) is presented first. Then a collection of eight lambda calculi, known as $\lambda \rightarrow$ *cube*, is introduced by describing three of its axes. These axes represents extension of $\lambda \rightarrow$ in the direction of $\lambda 2$, $\lambda \underline{\omega}$ and λP. We also see how these extended versions and their combination can be used to encode more enriched logics. For some of the above lambda calculi equivalent Curry variants are possible. However, this chapter avoids discussions of Curry version and always considers the Church versions without explicit mention. Finally the description method of the systems in the $\lambda -$ *cube* is generalized, introducing the so called "pure type systems" (PTSs). Then we locate $\lambda C'$ as the PTS closest to the type theory underlying Coq.

3.1 Simply Typed Lambda Calculus ($\lambda \rightarrow$)

The system $\lambda \rightarrow$ Church consists of a set of *types* T, a set of *pseudoterms* Λ_E, a set of bases, a *reduction relation* on Λ_E and a type assignment relation $\vdash$. These notions are defined as follows:

1.	Types	$T = V_T \mid T \rightarrow T$;
2.	Pseudoterms	$\Lambda_E = V \mid \Lambda_E \Lambda_E \mid \lambda V : T.\Lambda_E$
3.	Bases	$\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$, with all x_i distinct and all $A_i \in T$;
4.	Contraction rule	$(\lambda x : A.M)N \rightarrow_\beta M[x := N]$;

It should be noted that character V_T denotes the syntactic category of *type variables* while V represents the category of *term variables*. In types we let brackets associate to the right and omit outside brackets. However, in term applications we let brackets associate to the left and omit outside brackets. For example, $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$ denotes $((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))$,while MNP denotes $((MN)P)$. A lambda term of the form $(\lambda x : A.M)N$ is called a $\beta-redex$ and can be contracted using the one step reduction relation $\rightarrow_\beta$. The reflexive and transitive closure of this relation is denoted by $\twoheadrightarrow_\beta$. The reflexive, transitive and symmetric closure of $\rightarrow_\beta$ is denoted by $=_\beta$.

Type assignment $\Gamma \vdash M : A$ is defined by a *deduction system* as follows.

$$\text{(start-rule)} \quad \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A};$$

$$(\rightarrow\text{elimination}) \quad \frac{\Gamma \vdash M : (A \rightarrow B)\ \Gamma \vdash N : A}{\Gamma \vdash (MN) : B};$$

$$(\rightarrow\text{introduction}) \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M) : (A \rightarrow B)}.$$

A pseudoterm M is called *legal* or *typable* if for some Γ and A one has $\Gamma \vdash M : A$. Following are some important properties of the system $\lambda \rightarrow Church$.

- *Subject Reduction:*
  If $M : \sigma$ and $M \rightarrow_\beta P$, then $P : \sigma$.
- *Church-Rosser:*
  If M is a well-typed term in $\lambda \rightarrow$and $M \twoheadrightarrow_\beta P$ *and* $M \twoheadrightarrow_\beta N$, then there is a well typed term Q such that $P \twoheadrightarrow_\beta Q$ *and* $N \twoheadrightarrow_\beta Q$.
- *Strong Normalization:*
  If M is well-typed in $\lambda \rightarrow$, then there is no infinite $\beta-$reduction path starting from M.

Proofs can be found in [1].

Minimal propositional logic vs the system $\lambda \rightarrow$ -Church

Minimal propositional logic $PROP$, corresponds to the *implication fragment* of intuitionistic propositional logic, and consists of

- *implicational propositions*, generated by the following abstract syntax: $prop ::= PropVar \mid (prop \rightarrow prop)$.
- *derivation rules*

  $$\frac{\overset{[\varphi]^u}{\psi}}{\varphi \rightarrow \psi}(\rightarrow\mathrm{I})^u \qquad \frac{\varphi \rightarrow \psi \;\; \varphi}{\psi}\ (\rightarrow\mathrm{E})$$

  Assuming Δ is a set of propositions, we write $\Delta \vdash_{PROP} \sigma$ if there is a derivation using these rules with conclusion σ and all the non-discharged assumptions in Δ.

In the present context, Curry-Howard formulas-as-types correspondence states that there is a natural one-to-one correspondence between derivations in minimal propositional logic and typable terms in $\lambda \rightarrow$. In other words, the typing judgement $x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n \vdash M : \sigma$ can also be read as M is a proof of σ from the assumptions $\tau_1, \tau_2, \ldots, \tau_n$. The formulas-as-type part of the correspondence in this case is trivial: a proposition (formula) in $PROP$ is just a type in $\lambda \rightarrow$, but the most interesting part of the correspondence is the *proofs-as-terms* embedding. In Coq, this corresponds to the idea of *proof terms.* After successful completion of a proof the system generates a proof-term which in some way encodes the whole construction for the proposition declared as `Goal`. For $PROP$, even the proofs-as-terms encoding is quite straight- forward.

The proofs-as-terms embedding from derivation of $PROP$ to term of $\lambda \rightarrow$ is defined inductively as follows. We associate to a list of propositions Δ a context Γ_Δ in the obvious way by replacing σ_i in Δ with $x_i : \sigma_i \in \Gamma_\Delta$. On the left we give the inductive clause for the derivation and on the right we describe on top of the line the terms we have (by induction) and below the line the term that the derivation gets mapped to.

$$\frac{\sigma \in \Delta}{\Delta \vdash \sigma} \qquad \rightsquigarrow \qquad \frac{x : \sigma \in \Gamma_\Delta}{\Gamma_\Delta \vdash x : \sigma}$$

$$\frac{\varphi \rightarrow \psi \;\; \varphi}{\psi}\ (\rightarrow\mathrm{E}) \quad \rightsquigarrow \quad \frac{\Gamma_1 \vdash M : \varphi \rightarrow \psi \;\; \Gamma_2 \vdash N : \varphi}{\Gamma_1 \cup \Gamma_2 \vdash MN : \psi}$$

$$\frac{\overset{[\varphi]^u}{\psi}}{\varphi \rightarrow \psi}(\rightarrow\mathrm{I})^u \qquad \rightsquigarrow \qquad \frac{\Gamma, x : \varphi \vdash M : \psi}{\Gamma \vdash \lambda x : \varphi.M : \varphi \rightarrow \psi}$$

We denote this embedding by $\overline{-}$, so if D is a derivation from $PROP$, $\overline{D}$ is a $\lambda \rightarrow$-term representing it. This way of encoding derivations (constructions) seems to be very natural. Here, we use $\lambda x : \varphi.M$ to encode a proof of $\varphi \rightarrow \psi$, which means that a proof of $\varphi \rightarrow \psi$ is a function which takes a proof x (of

φ) and returns a proof M (of ψ). This idea matches exactly with the BHK-interpretation, for intuitionistic propositional logic, where a construction for $\varphi \to \psi$ is a function transforming every construction of φ into a construction of ψ.

For example, consider the following natural deduction derivation and the term in $\lambda \to$ it gets mapped to.

$$\dfrac{\dfrac{\dfrac{\dfrac{[\alpha \to \beta \to \gamma]^3 \quad [\alpha]^1}{\beta \to \gamma} \quad \dfrac{[\alpha \to \beta]^2 \quad [\alpha]^1}{\beta}}{\gamma}1}{\alpha \to \gamma}2}{(\alpha \to \beta) \to \alpha \to \gamma}3}{(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma} \quad \mapsto^{\substack{\lambda x:\alpha\to\beta\to\gamma.\lambda y:\alpha\to\beta.\lambda z:\alpha.xz(yz) \\ :(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma}}$$

To see how the term on the right is created, it is best to decorate the deduction tree with terms, starting from the leaves and working downwards, finally creating a term that corresponds to the whole deduction. It is shown below:

$$\dfrac{\dfrac{\dfrac{\dfrac{[x : \alpha \to \beta \to \gamma]^3 \quad [z : \alpha]^1}{xz : \beta \to \gamma} \quad \dfrac{[y : \alpha \to \beta]^2 \quad [z : \alpha]^1}{yz : \beta}}{xz(yz) : \gamma}1}{\lambda z : \alpha.xz(yz) : \alpha \to \gamma}2}{\lambda y : \alpha \to \beta.\lambda z : \alpha.xz(yz) : (\alpha \to \beta) \to \alpha \to \gamma}3}{\lambda x : \alpha \to \beta \to \gamma.\lambda y : \alpha \to \beta.\lambda z : \alpha.xz(yz) : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma}$$

The above way of embedding derivations into $\lambda \to$-terms can be shown to be sound and complete in the following sense:

- Soundness:
  If D is a natural deduction in $PROP$ with conclusion σ and all the non-discharged assumption in Δ, then $\Gamma_\Delta \vdash \overline{D} : \sigma$ in $\lambda \to$.
- Completeness:
  If $\Gamma \vdash M : \sigma$ in $\lambda \to$, then there is a derivation of σ from the Δ in $PROP$, where Δ is Γ without the variable-assignments.

The proof of the above claim can be given by a straight forward induction on the derivation and hence we will not discuss it here. We can extend the above encoding scheme to cover second order propositional logic and predicate logic as well. However, to do that we need to enrich the simply typed lambda calculus $\lambda \to$ in a variety of ways to accommodate Dependent types, which we discuss in the next section.

3.2 Dependent Types

Simple type theory is not very expressive: we do not have a clear notion of data types. Therefore, we can not reuse a function. For example, $\lambda x : \alpha.x : \alpha \to \alpha$ and $\lambda x : \beta.x : \beta \to \beta$ are essentially the same function but written two times for different data types. In Church version, the untyped $\lambda-$term $(\lambda y.y)(\lambda x.x)$ can be written as $(\lambda y : \sigma \to \sigma.y)(\lambda x : \sigma.x)$, which shows that we can type this term with type $\sigma \to \sigma$. If we wish to force ourselves to type the above two identities with same type at the same time, we would want to type the $\lambda-$term $(\lambda f.ff)(\lambda x.x)$ (since it reduces in one step to the term above). However, this is not possible since the term f should be of type $\sigma \to \sigma$ and of type $(\sigma \to \sigma) \to \sigma \to \sigma$ at the same time, which we can't achieve in $\lambda \to$.

In short, we wish to define functions that can treat types polymorphically. Therefore, we add types of the form $\forall\alpha.\sigma$. This would enable us to assign a type $\forall\alpha.\alpha \to \alpha$ to f . Now, f can be of type $\sigma \to \sigma$ and of type $(\sigma \to \sigma) \to \sigma \to \sigma$ at the same time, hence $(\lambda f.ff)(\lambda x.x)$ can be typed as well. This extension of $\lambda \to$ is called $\lambda 2$ (or system F) and was suggested by Girard in [5].

The system $\lambda 2$

The system $\lambda 2-$Church is defined as follows:

1. Types $T = V_T \mid T \to T \mid \forall V_T T$;
2. Pseudoterms $\Lambda_E = V \mid \Lambda_E \Lambda_E \mid \lambda V : T.\Lambda_E \mid \Lambda V_T.\Lambda_E \mid \Lambda_E T$
3. Bases $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$, with all x_i distinct and all $A_i \in T$;
4. Contraction rule $(\lambda x : A.M)N \to_\beta M[x := N]$; $(\Lambda\alpha.M)A \to_\beta M[\alpha := A]$
5. Type assignment $\Gamma \vdash M : A$ is defined as follows.

(start-rule)	$\dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$;
($\to$elimination)	$\dfrac{\Gamma \vdash M : (A \to B)\ \Gamma \vdash N : A}{\Gamma \vdash (MN) : B}$;
($\to$introduction)	$\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M) : (A \to B)}$;
($\forall-$elimination)	$\dfrac{\Gamma \vdash M : (\forall\alpha.A)}{\Gamma \vdash MB : A[\alpha := B]}$, $B \in T$;
($\forall-$introduction)	$\dfrac{\Gamma \vdash M : A}{\Gamma \vdash (\Lambda\alpha.M) : (\forall\alpha.A)}$, $\alpha \notin FV(\Gamma)$.

The condition that $\alpha \notin FV(\Gamma)$ is very important in the $\forall-$introduction rule. It means that variable α is not free in the type of a free variable of M. More precisely, if $x \in dom(\Gamma) \cap FV(M)$ then $\alpha \notin FV(\Gamma(x))$. It should be noted that application of a pseudoterm to a type is possible now and it yields a pseudoterm. We also have $(\Lambda\alpha.M)$ as an additional pattern for pseudoterm whose type is $(\forall\alpha.A)$, provided $M : A$ and α satisfies the necessary conditions.

From the proof theory point of view, it means that $(\Lambda\alpha.M)$ represents the proof of proposition $(\forall\alpha.A)$, provided M is a proof of A and α is not free in any of the formula which appear in the list of assumptions. This is the key observation that can explain the Curry-Howard isomorphism at a deeper level between the system $\lambda 2$ and second order propositional logic, $PROP2$. The *second order propositional logic*, $PROP2$ has the following additional derivation rules, which can easily be seen to correspond to the $\forall-$elimination and introduction typing rules of the system $\lambda 2$.

$$\frac{\begin{array}{c}\tau_1,\ldots\tau_n\\ \vdots\\ \sigma\end{array}}{\forall\alpha.\sigma}\ \forall - I,\ if\ \alpha \notin FV(\tau_1,\ldots,\tau_n) \qquad \frac{\forall\alpha.\sigma}{\sigma[\alpha := \tau]}$$

We can easily extend the formulas-as-type embedding, that we defined earlier for $PROP$, to cover deductions in $PROP2$ and map it to terms in $\lambda 2$.

Although, the logic of $PROP2$ only has implication and universal quantification, all other intutionistic connectives can now be defined as follows:

$$\begin{array}{rcl}\bot & := & \forall\alpha.\alpha\\ \sigma \wedge \tau & := & \forall\alpha.(\sigma \to \tau \to \alpha) \to \alpha\\ \sigma \vee \tau & := & \forall\alpha.(\sigma \to \alpha) \to (\tau \to \alpha) \to \alpha\\ \exists\alpha.\sigma & := & \forall\beta.(\forall\alpha.\sigma \to \beta) \to \beta\end{array}$$

This shows that $\lambda 2$ has enough expressive power to encode a proof of any statement in the intutionistic propositional logic. However, before we can say so confidently we must verify that the above second order definitions for connectives make sense. More precisely, all the standard constructive deduction rules (elimination and introduction rules) presented in the second chapter, should be derivable using the above definitions. As it turns out, it is indeed the case.

For example, consider the following derivation which shows how to derive σ from $\sigma \wedge \tau$ using the above second order definition of conjunction.

$$\frac{\dfrac{\forall\alpha.(\sigma \to \tau \to \alpha) \to \alpha}{(\sigma \to \tau \to \sigma) \to \sigma} \quad \dfrac{\dfrac{[\sigma]^1}{\tau \to \sigma}}{\sigma \to \tau \to \sigma}1}{\sigma}$$

This shows that $\wedge-$elimination rule is derivable from the above second order definition of conjunction. We can also embed the above derivation into a $\lambda-$term as follows:

$$\dfrac{\dfrac{M : \forall\alpha.(\sigma \to \tau \to \alpha) \to \alpha}{M\sigma : (\sigma \to \tau \to \sigma) \to \sigma} \quad \dfrac{\dfrac{[x : \sigma]^1}{\lambda y : \tau.x \ : \ \tau \to \sigma}}{\lambda x : \sigma.\lambda y : \tau.x \ : \ \sigma \to \tau \to \sigma}1}{M\sigma(\lambda x : \sigma.\lambda y : \tau.x) \ : \ \sigma}$$

Thus the following term is a "witness" in the system $\lambda 2$ for the $\wedge$-elimination.

$$\lambda M : \sigma \wedge \tau.M\sigma(\lambda x : \sigma.\lambda y : \tau.x) \ : \ (\sigma \wedge \tau) \to \sigma$$

In a similar way we can generate a witness in $\lambda 2$, corresponding to every elimination and introduction rule in the intutionistic propositional logic.

The system $\lambda\underline{\omega}$

It is very natural to think of functions which takes types as input. For example, we may want to define $f \equiv \lambda\alpha \in T.\alpha \to \alpha$, such that $f(\alpha) = \alpha \to \alpha$. It is not possible to define such a function in $\lambda 2$, even with all the expressive power it has. This will be possible, however, in the system $\lambda\underline{\omega}$. Moreover, in $\lambda\underline{\omega}$, types are generated by the system itself and not using the informal metalanguage. There is a constant $*$ such that $\sigma : *$ corresponds to $\sigma \in T$. Thus, the statement $\alpha, \beta \in T \implies (\alpha \to \beta) \in T$ now becomes $\alpha : *, \beta : * \vdash (\alpha \to \beta) : *$. In this new notation we can write $f \equiv \lambda\alpha : *.\alpha \to \alpha$.

We introduce a new category $\mathbb{K}$ (of kinds) as a parking place for f, since it neither falls in the level of terms nor among the types. The abstract syntax for kind is: $\mathbb{K} = *|\mathbb{K} \to \mathbb{K}$. In other words, $\mathbb{K} = \{*, * \to *, \dots\}$. We use the notation $k : \Box$ instead of $k \in \mathbb{K}$, where $\Box$ is introduced as a constant to represent $\mathbb{K}$. If $\vdash F : k$ and $\vdash k : \Box$, then F is called a constructor of kind k.

The system $\lambda\underline{\omega}$ is defined as follows:

1. The notion of types and terms are merged in the same syntactic category called pseudo-expressions $\mathcal{T}$ and is defined as follows
   $\mathcal{T} = V \,|\, C \,|\, \mathcal{T}\mathcal{T} \,|\, \lambda V : \mathcal{T}.\mathcal{T} \,|\, \mathcal{T} \to \mathcal{T}$
   Where V is infinite collection of variables and C of constants. Among these constants two elements are selected and given the names $*$ and $\Box$. These are called *sorts*.

2. A statement of $\lambda\underline{\omega}$ is of the form $M : A$ with $M, A \in \mathcal{T}$. A context is a finite linearly ordered set of statements with distinct variables as subjects. An empty context is denoted by $<>$.

3. Typing rules: The notation $\Gamma \vdash M : A$ is defined by the following axiom and rules, where the letter s ranges over $\{*, \Box\}$.

(axiom)	$<>\vdash * : \Box$;
(start-rule)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$, $x \notin \Gamma$;
(weakening rule)	$\dfrac{\Gamma \vdash A : B \;\; \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$, $x \notin \Gamma$;
(type/kind formation)	$\dfrac{\Gamma \vdash A : s \;\; \Gamma \vdash B : s}{\Gamma \vdash (A \to B) : s}$;
(application rule)	$\dfrac{\Gamma \vdash F : (A \to B) \;\; \Gamma \vdash a : A}{\Gamma \vdash Fa : B}$;
(abstraction rule)	$\dfrac{\Gamma, x : A \vdash b : B \;\; \Gamma \vdash (A \to B) : s}{\Gamma \vdash (\lambda x : A.b) : (A \to B)}$;
(conversion rule)	$\dfrac{\Gamma \vdash A : B \;\; \Gamma \vdash B' : s \;\; B =_\beta B'}{\Gamma \vdash A : B'}$.

The system λP

In the system $\lambda\underline{\omega}$, the notion of kind $\mathbb{K}$ has already helped us to define a function on types. However, its real benefit can be seen in system λP to realise the idea of predicates. Predicates in logic can also be viewed as functions f , whose input is variables (terms) and returns a proposition as output. If $A : *$ and $f : A \to *$, then f can be thought of acting as a predicate, since $fa : *$(i.e. fa is a proposition) whenever $a : A$ (i.e. a is a term of type A). Again, the question is where to place $A \to *$(i.e. what is its type ?). In the system λP, the notion of kind is extended such that if A is a type and k is a kind, then $A \to k$ is a kind. Thus, in particular, $A \to *$ is a kind in λP and hence f can be declared with type $A \to *$.

The system λP is formally defined as follows:

1. The set of pseudo-expression of λP , notation, $\mathcal{T}$ is defined as follows
   $\mathcal{T} = V \mid C \mid \mathcal{T}\mathcal{T} \mid \lambda V : \mathcal{T}.\mathcal{T} \mid \prod V : \mathcal{T}.\mathcal{T}$
   where V is the collection of variables and C that of constants. No distinction is made between type and term variables. Two of the constants, called sorts, are represented as $*$ and $\Box$.

2. Typing rules: the notation $\vdash$ is defined by the following axiom and rules. The letter s ranges over $\{*, \Box\}$.

(axiom)	$<>\vdash * : \Box$;
(start-rule)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$, $x \notin \Gamma$;
(weakening rule)	$\dfrac{\Gamma \vdash A : B \;\; \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$, $x \notin \Gamma$;
(type/kind formation)	$\dfrac{\Gamma \vdash A : * \;\; \Gamma, x : A \vdash B : s}{\Gamma \vdash (\prod x : A.B) : s}$;
(application rule)	$\dfrac{\Gamma \vdash F : (\prod x : A.B) \;\; \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]}$;
(abstraction rule)	$\dfrac{\Gamma, x : A \vdash b : B \;\; \Gamma \vdash (\prod x : A.B) : s}{\Gamma \vdash (\lambda x : A.b) : (\prod x : A.B)}$;
(conversion rule)	$\dfrac{\Gamma \vdash A : B \;\; \Gamma \vdash B' : s \;\; B =_\beta B'}{\Gamma \vdash A : B'}$.

In this type theory, we have a new $\prod$−type. This is a generalization of the well-known function type: if $x \notin FV(B)$, then $(\prod x : A.B)$ is just $A \to B$. Following the methods for $\lambda \to$ and $\lambda 2$ we can embed an even richer logic in λP. In the system λP, we can embed minimal first order predicate logic, the predicate logic with just implication and universal quantification and the intutionistic rules for these connectives. Here, the key idea is to represent both the domains and the formulas of the logic as types. For example, consider the following many sorted structure of first order predicate logic

$$\mathcal{A} =< A,\, B,\, f,\, g,\, P,\, Q,\, c >$$

where,

A, B are non-empty sets, the sorts of $\mathcal{A}$
$g : (A \to B)$ is a function;
$P \subset A$ and $Q \subset A \times B$ are relations;
$c \in A$ is a constant.

The above structure can easily be embedded into λP by considering the following context

$\Gamma := A : *,\, B : *,\, g : A \to B,\; P : A \to *,\; Q : A \to B \to *,\; c : A.$

It is easy to see that the introduction and elimination rules for $\forall$ corresponds to the abstraction and application rule of the $\prod$−type.

3.3 The $\lambda - cube$

The systems $\lambda \rightarrow$, $\lambda 2$, $\lambda \underline{\omega}$ and λP provide us with different expressive powers in order to encode different features of logic and proofs. However, all of these type theories differ in the way expressions and typing rules are defined. The notion of $\lambda - cube$, gives a uniform framework for defining all these different type theories. This uniform framework, also makes it possible to integrate all the essential features of different type theories into one to get the strongest possible system in the cube, called *calculus of construction*. The systems $\lambda \rightarrow$ and $\lambda 2$ in the $\lambda - cube$ are not given in their original version, but in an equivalent variant.

The System of $\lambda - cube$ can be defined as follows:

1. The set of psedu-expressions $\mathcal{T}$ is defined by the following abstract syntax,

   $$\mathcal{T} = V \mid C \mid \mathcal{T}\mathcal{T} \mid \lambda V : \mathcal{T}.\mathcal{T} \mid \prod V : \mathcal{T}.\mathcal{T}$$

   where V and C are infinite collections of variables and constants respectively. No distinction between type and term variable is made.

2. The notion of $\beta-$ conversion is defined by the following rule:

   $$(\lambda x : A.B)C \rightarrow B[x := C]$$

3. The notion of statement and pseudo-context is same as in λP. The notion, $\Gamma \vdash A : B$, is defined using typing rules. These rules are given in two groups:
   (a) the general axiom and rules, valid for all systems of the $\lambda - cube$;
   (b) the specific rules, differentiating between the eight systems; these are parametrized $\prod -$introduction rules.

   Two constants are selected and given names $*$ and $\Box$, these are called sorts. The notation $\vdash$ is defined by the following axiom and rules. Let $\mathcal{S} = \{*, \Box\}$ and s, s_1, s_2 range over $\mathcal{S}$. We use $A, B, C, a, b, \ldots$ for arbitrary pseudo-terms and $x, y, z, \ldots$ for arbitrary variables.

1. General axiom and rules.

(axiom) $<>\vdash * : \Box$;

(start-rule) $\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x \ : \ A}$, $x \notin \Gamma$;

(weakening rule) $\dfrac{\Gamma \vdash A : B \ \ \Gamma \vdash C : s}{\Gamma, x : C \vdash A \ : \ B}$, $x \notin \Gamma$;

(application rule) $\dfrac{\Gamma \vdash F : (\prod x : A.B) \ \ \Gamma \vdash a : A}{\Gamma \vdash Fa \ : \ B[x := a]}$;

(abstraction rule) $\dfrac{\Gamma, x : A \vdash b : B \ \ \Gamma \vdash (\prod x : A.B) : s}{\Gamma \vdash (\lambda x : A.b) \ : \ (\prod x : A.B)}$;

(conversion rule) $\dfrac{\Gamma \vdash A : B \ \ \Gamma \vdash B' : s \ \ B =_\beta B'}{\Gamma \vdash A \ : \ B'}$.

2. The specific rules

(s_1, s_2) rule $\dfrac{\Gamma \vdash A : s_1, \ \ \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\prod x : A.B) \ : \ s_2}$.

4. The eight systems of the $\lambda - cube$ are defined by taking the general rules plus a specific subset of the set of rules $\{(*,*), (*,\Box), (\Box,*), (\Box,\Box)\}$.

System	Set of specific rules			
$\lambda \rightarrow$	$(*,*)$			
$\lambda 2$	$(*,*)$	$(\Box,*)$		
$\lambda\underline{\omega}$	$(*,*)$			$(\Box,\Box)$
λP	$(*,*)$		$(*,\Box)$	
$\lambda P2$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	
$\lambda\omega$	$(*,*)$	$(\Box,*)$		$(\Box,\Box)$
$\lambda P\underline{\omega}$	$(*,*)$		$(*,\Box)$	$(\Box,\Box)$
$\lambda P\omega = \lambda C$	$(*,*)$	$(\Box,*)$	$(*,\Box)$	$(\Box,\Box)$

The versions of $\lambda \rightarrow$ and $\lambda 2$ in the $\lambda - cube$ appears, at face, to be different from the variants that we discussed earlier in this chapter. However, both the versions are equivalent. The equivalence is clearly seen, once we define the notions of $A \rightarrow B$, $\forall\alpha.A$ and $\Lambda\alpha.M$ in $\lambda - cube$ as follows:

$$\begin{array}{lcl} A \rightarrow B & \equiv & \prod x : A.B, \text{ where } x \notin FV(B) \\ \forall\alpha.A & \equiv & \prod \alpha : *.A, \\ \Lambda\alpha.M & \equiv & \lambda\alpha : *.M. \end{array}$$

With these definitions one can deduce elimination and introduction rules of systems $\lambda \rightarrow$ and $\lambda 2$ from the application and abstraction rules of the $\lambda - cube$.

For example, consider the following transformations which yields the elimination rule for implication from the application rule of $\lambda - cube$:

$$\frac{\Gamma \vdash F : (\prod x : A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa \ : \ B[x := a]}, \ x \notin FV(B) \Longrightarrow \frac{\Gamma \vdash F : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash Fa \ : \ B}$$

In a similar way using $(\Box, *)$ rule, the following transformation shows that $(\forall x.B)$ is a valid type:

$$(\Box, *) \text{ rule } \Longrightarrow \frac{\Gamma \vdash * : \Box, \quad \Gamma, x : * \vdash B : *}{\Gamma \vdash (\prod x : *.B) \ : *} \Longrightarrow \frac{\Gamma \vdash * : \Box, \quad \Gamma, x : * \vdash B : *}{\Gamma \vdash (\forall x.B) \ : *}$$

Again, consider the following transformation which shows how to get $\forall-$ introduction rule from the abstraction rule in the $\lambda - cube$:

$$\frac{\Gamma, x : * \vdash b : B \quad \Gamma \vdash (\prod x : *.B) : s}{\Gamma \vdash (\lambda x : *.b) \ : \ (\prod x : *.B)} \Longrightarrow \frac{\Gamma, x : * \vdash b : B \quad \Gamma \vdash (\prod x : *.B) : s}{\Gamma \vdash (\Lambda x.b) \ : \ (\forall x.B)}$$

Thus, continuing in the same way we can show, that all other typing rules of the systems $\lambda 2$ can be derived by using general rules of $\lambda - cube$ together with the specific rules $(*, *)$ and $(\Box, *)$. Hence the two descriptions, although they might look different, are essentially equivalent.

3.4 Pure type systems

The method of generating the systems in the $\lambda - cube$, can be further generalized, resulting in the notion of pure type system (PTS). The general setting of the PTSs makes it easier to give proofs of several propositions about the systems in the $\lambda - cube$. The PTSs are based on the set of pseudo-terms $\mathcal{T}$ for the $\lambda - cube$. Abstract syntax for $\mathcal{T}$ is :

$$\mathcal{T} = V \,|\, C \,|\, \mathcal{T}\mathcal{T} \,|\, \lambda V : \mathcal{T}.\mathcal{T} \,|\, \prod V : \mathcal{T}.\mathcal{T}$$

The specification of a PTS consists of a triple $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, where $\mathcal{S}$ is a subset of C, called the sorts. $\mathcal{A}$ is a set of axioms of the form $c : s$, with $c \in C$ and $s \in \mathcal{S}$. $\mathcal{R}$ is a set of rules of the form (s_1, s_2, s_3) with $s_1, s_2, s_3 \in \mathcal{S}$. We use notation $\lambda S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ for the PTS determined by the specification $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$. The set V of variables is divided into disjoint infinite subsets V_s for each sort $s \in \mathcal{S}$.Thus, $V = \cup\{V_s | s \in \mathcal{S}\}$. The members of V_s are denoted by ${}^s x$, ${}^s y$, ${}^s z, \ldots$.The notion of type derivation is defined by the following axioms and rules. Let s ranges over $\mathcal{S}$, the set of sorts; and x ranges over variables.

$\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$

(axioms)	$<>\vdash c : s$;	if $(c : s) \in \mathcal{A}$
(start-rule)	$\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$;	if $x \equiv {}^s x \notin \Gamma$;
(weakening rule)	$\dfrac{\Gamma \vdash A : B \;\; \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$;	if $x \equiv {}^s x \notin \Gamma$;
(application rule)	$\dfrac{\Gamma \vdash F : (\prod x : A.B) \;\; \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]}$;	
(abstraction rule)	$\dfrac{\Gamma, x : A \vdash b : B \;\; \Gamma \vdash (\prod x : A.B) : s}{\Gamma \vdash (\lambda x : A.b) : (\prod x : A.B)}$;	
(conversion rule)	$\dfrac{\Gamma \vdash A : B \;\; \Gamma \vdash B' : s \;\; B =_\beta B'}{\Gamma \vdash A : B'}$.	
(product)	$\dfrac{\Gamma \vdash A : s_1, \;\; \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\prod x : A.B) : s_3}$.	if $(s_1, s_2, s_3) \in \mathcal{R}$.

Sometimes, we write (s_1, s_2) as an abbreviation for the rule (s_1, s_2, s_2). The PTS $\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$ is called *full* if $\mathcal{R} = \{(s_1, s_2) | \, s_1, s_2 \in \mathcal{S}\}$.

Following are some example of PTSs:

1. $\lambda \rightarrow$ is the PTS determined by

$\mathcal{S}$	$*, \Box$
$\mathcal{A}$	$* : \Box$
$\mathcal{R}$	$(*, *)$

2. $\lambda 2$ is the PTS determined by

$\mathcal{S}$	$*, \Box$
$\mathcal{A}$	$* : \Box$
$\mathcal{R}$	$(*, *), (\Box, *)$

3. λC is the full PTS determined by

$\mathcal{S}$	$*, \Box$
$\mathcal{A}$	$* : \Box$
$\mathcal{R}$	$(*, *), (\Box, *), (*, \Box), (\Box, \Box)$

4. A variant $\lambda C'$ of λC is the full PTS with

$\mathcal{S}$	$*^t, *^p, \Box$
$\mathcal{A}$	$*^t : \Box, *^p : \Box$
$\mathcal{R}$	$\mathcal{S}^2$, i.e all pairs

The PTS $\lambda C'$ is very similar to calculus of inductive construction, the underlying type theory in the previous versions of Coq, where we have two sorts, named `Set` and `Prop`. However, in the recent version of Coq (pCIC), sort `Set` is predicative. This means that the rule $(\Box, *^t)$ is not present in $\mathcal{R}$, and hence it is not a full PTS.

Chapter 4

Inductive types in Coq

Inductive constructions are almost unavoidable, as they occur very frequently in mathematics and logic. We may want to construct a set by starting with some *initial elements* and applying certain *operations* to them over and over again. More precisely, we want the smallest set containing the initial elements and *closed under* the operations. We call such a set an *inductive* set. We may also wish to define some properties of elements of such a set and in certain cases we can also prove their validity using principle of mathematical induction. Inductive sets becomes even more interesting because we can specify and compute recursive functions on them. *Inductive types* in Coq, help us realise all of these notions. Inductive types are closed with respect to their defining (*introduction*) rules. Moreover, one can also use these types (*elimination rule*) to define predicates as well as recursive functions over them. This chapter very briefly discusses the inductive types in Coq. A more detailed and precise explanation can be found in [8, 7, 4].

4.1 Defining Inductive Types

Like any other type, that we have discussed so far, Inductive types are also defined using introduction rules. The introduction rule for inductive types explain the most basic ways of constructing an element of the type. Different rules can be considered as introducing different objects. For example, consider the following inductive definition which introduces the inductive type, `nat`, for natural numbers:

```
Inductive nat :  Set:=
| 0 :  nat
| S : nat→nat.
```

In Coq, any inductive definition has two main parts. First, we declare what kind of inductive object we wish to characterize (a set, in present case). Then

we give the introduction rules that define the type (`0` and `S` in this case). These rules are also known as *constructors*. It should be noted that inductive types are *closed* with respect to their introduction rules. For example in the present case, if n:`nat`, then n must have been introduced either by the rule `0` or by an application of the rule `S` to a previously constructed natural number. In this sense `nat` is closed.

Any inductive definition adds new constants to the present context. For example, in the present case the constants `nat`, `0` and `S` becomes available in the current context with following types:

```
nat:  Set
0:  nat
S:nat→nat
```

However, these are not the only constants added to the context. Coq adds three more constants named `nat_ind`, `nat_rec` and `nat_rect`, which corresponds to different principles of structural induction on natural numbers that Coq automatically infers from the inductive definition of `nat`.

In the previous chapter we have seen that there are systems in $\lambda - cube$, which has enough expressive power to represent `true`, `false` and other logical connectives like `conjunction` and `disjunction,` which could not be expressed using simply typed lambda calculus ($\lambda \rightarrow$). In spite of this fact, Coq implements most of these types as inductive types. We can use Print command to know how these connectives are actually defined in Coq. Following is the list of outputs, displayed in *italic* style, in Coq using Print command against these connectives:

```
Print False.
```
Inductive False: Prop:=

```
Print True.
```
Inductive True: Prop:= I :True

```
Print and.
```
Inductive and (A B : Prop) : Prop := conj : A → B→ A∧B

```
Print or.
```
Inductive or (A B : Prop) : Prop :=
| or_introl : A → A ∨ B
| or_intror : B → A ∨ B.

It is worth noting that no introduction rule is given in the inductive definition of False. This resembles with its intuitionistic interpretation that there should not be any proof for False. On the other hand, the inductive definition of True assures that every context will have I as a proof for True. In a similar way one can also derive and confirm the intutionistic behaviours of other connectives from their inductive definitions.

4.2 Elimination rule for Inductive types

From a proof theoretic view, elimination of a hypothesis, $h : A$ is a way to use the proof h of A in order to define a proof of some proposition B. Similarly in type theory, an elimination rule for the type A is some way to use an object $a : A$ in order to define an object in some type B.

An obvious elimination for inductive type is defining functions by case analysis. For example, a systematic way of building a value of type B from any value of type `nat` is to associate to `0` a constant $t_0 : B$ and to every term of the form "`S p`" a term $t_S : B$. Consider the following function defined using case analysis, which computes the predecessor:

```
Definition pred (n:nat):=  match n with
                             | 0  ⟹ 0
                             | S m  ⟹ m
                           end.
```

The construct "`match n with ... end`" in the above case has a type `nat`, since each branch returns a value of type `nat`. Therefore, the type of output is also `nat` which is a constant and independent of n. However, for the pattern matching construct of form "`match n with ... end`" a more general typing rule can be obtained in which the type of the whole expression may also depend on n. For instance let Q : `nat` →`Set`, and n : `nat` and we wish to build a term of type $Q\,n$. In order to build such a term, we can associate to the constructor `0` a term t_O : Q `0` and to the pattern "`S p`" some term t_S : $Q\,(S\,p)$. This is an example of *dependent case analysis*. The syntax of the dependent case analysis [8] gives us the following typing rule:

$$\frac{Q : nat \to Set \quad t_O : Q\,0 \quad p : nat \vdash t_S : Q(S\,p) \quad n : nat}{match\ n\ as\ n_0\ and\ return\ Q\,n_0\ with \mid 0 \Rightarrow t_O \mid S\,p \Rightarrow t_S\ end\ :\ Q\,n}$$

The above typing rule can also be read as the following logical principle (replacing `set` by `Prop` in the type of Q): in order to prove that a property Q holds for all n, it is sufficient to prove that Q holds for `0` and that for all p:`nat`, Q holds for $(S\ p)$. For example, if we do this case analysis for False we get the following typing rule:

$$\frac{Q : Prop \quad p : False}{match\ p\ return\ Q\ with\ end : Q}$$

The above rule says that if we could construct a proof p of `False` then the term " *match p return Q with end*" can act as proof for any arbitrary proposition Q. In other words, it is possible to prove any arbitrary proposition in a context, in which `False` is provable.

Structural Induction

Elements of inductive types are well-founded with respect to the structural order induced by the constructors of the type. This extra hypothesis about

well-foundedness justifies a stronger elimination rule called structural induction. Definition using structural induction are expressed through the `Fixpoint` command. This form of elimination consists in defining a value "$f\ x$" from some element x of the inductive type I, assuming that values have been already associated in the same way to the sub-parts of x of type I. For example, consider the following recursive definition,

```
Fixpoint plus (n p:  nat){struct n}:  nat :=
match n with
              | 0⇒p
              | S m ⇒S (plus m p)
end.
```

When describing a recursive function in this way we have to respect a constraint called structural recursion. There should be at least one argument such that the recursive call can only be made on a subterm of that argument. More precisely, the argument of the recursive call must be a variable, and this variable must have been obtained from the initial argument by pattern matching. This constraint ensures that every computation involving recursive function terminates.

The principal of structural induction can also be used to define proofs, that is, to prove theorem. For example consider the following Coq session:

```
Variable P                   :  nat→Prop.
Hypothesis base_case         :  P 0.
Hypothesis ind_step          :∀n:nat, P n →P (S n).
Fixpoint nat_ind (n:  nat)   :  (P n) :=
  match n return P n with
   | 0 ⇒base_case
   | S m⇒ind_step m (nat_ind m)
end
```

It is easy to see that, in the above context, function `nat_ind` defines a term of type `P n` by structural induction on `n`. Such a function is called *elimination combinator*. From the proof theoretic point of view, an elimination combinator is any function that given a predicate `P`, defines a proof of "`P x`" by structural induction on `x`. In a context, where the constants `base_case` and `ind_step` has the type specified as above, the elimination combinator `nat_ind` will be considered as a proof term for `P n` whenever `n:nat`.

In Coq, the principle of proof by induction on natural number is a special case of an elimination combinator. Since this proof principle is used very often, Coq automatically generates it when an inductive type is introduced. The tactic `apply` can be used to apply this proof principle during a proof development. The tactic `elim` is a refinement of apply specially suited for the application of elimination combinators. The detailed description on how to use `elim` and some other useful tactics like `discriminate`, `injection` and `inversion` can be found in [8].

4.3 The Co-Inductive types

Inductive proof techniques, that we have seen so far, make it possible to prove statements for infinite collections of objects, such as `nat`. Although, the collections were infinite, the objects from those collections were finite in their structures. Each object of an Inductive type is well founded with respect to the constructors of the type, because it is built in a finite number of steps from the initial elements. Indeed, this was the intuitive justification for the principle of induction. Thus, Inductive types can only take care of finite objects.

If we wish to generate infinite objects and reason with them we need something more. This can be achieved with the help of *Co-inductive types*, which is obtained by relaxing the condition of well-foundedness, and hence may contain non-well-founded objects. For example, consider the following definition of co-inductive type `Stream`, which is the type of infinite sequences formed with the elements of type `A`,

```
CoInductive Stream (A: Set) :  Set :=
| Cons :  A→Stream A →Stream A.
```

The principle of structural induction is no more available for the co-inductive type `Stream`, due to the absence of well foundedness. Hence, the only elimination rule left for streams is definition by case analysis. For, example consider the following definition of `head` and `tail` of a stream :

```
Definition head (A: Set)(s:  Stream A) :=
match s with Cons a s' ⇒a end.

Definition tail (A: Set)(s:  Stream A) :=
match s with Cons a s' ⇒s' end.
```

Both of the above definitions define functions, which return finite objects. It is also possible to define infinite objects by using the `CoFixpoint` command. It is similar to the `Definition` command except that, it makes recursive calls possible, and therefore infinite recursions can result in the creation of, infinite objects. For example, consider the following definition, which introduces the infinite list $[a, a, \dots]$:

```
CoFixpoint repeat (A: Set)(a:A) : Stream A:=
Cons a (repeat a).
```

However, not all recursive definitions are allowed using the `Cofixpoint` command. A function (method), defined using `Cofixpoint` is only accepted if all recursive calls occur inside one of the arguments of a constructor of the co-inductive type. This is called the *guard-condition*. Another important feature of these functions is that they build values in Co-inductive types. This is in contrast to the recursive functions defined using Fixpoint command, which con-

sumes values in inductive types. The name Co-inductive type originates from this duality: Co-inductive types are the co-domains of co-recursive functions while inductive types are the domains of the recursive functions.

Although, case analysis is a valid proof principle, it may not be sufficient to prove extensional properties concerning the whole infinite objects. In a similar way, in which we defined elimination combinator for inductive type, one can also define an *introduction combinator* through `CoFixpoint`. This gives us a proof principle to reason with infinite objects of Co-inductive types. The details can be found in [2, 4]. Since the Co-inductive types are also closed with respect to their constructors, the tactics `discriminate` and `injection` can be used in the same way [2].

Chapter 5

Conclusion

In this report, we discussed the expressive power and different features of the Coq proof assistant, which helps in formalizing a mathematical theory. From the discussions in Chapter 3, it is clear that Coq is capable of encoding almost every kind of propositions and proofs that we usually come across. However, the process of formalizing a theory in Coq, consumes more time as compared to proving it on paper. Most of the time this happens because Coq doesn't accept incomplete proofs. On the other hand, while writing a proof on paper, we often accept omission of small steps of reasoning. Hence, once we commit to writing complete proofs, consumption of more time can't be counted as a drawback of proof assistants.

On the completion of a proof, Coq system generates a proof-term. This proof-term contains all the information, necessary to regenerate the whole proof. In Coq, users are not allowed to directly edit proof-terms. As a result even similar looking proofs has to be done again starting from the scratch. Allowing users to edit proof-terms, and accepting those edited proof-terms for type checking, can solve the problem of redoing similar proofs.

Many a time, while proving a proposition users end up spending more time, simply because they don't know the exact way by which Coq understands those propositions (i.e. the introduction rules). This happens because users' notion of proofs and propositions, evolved independent of any proof assistant, over a long period of doing mathematical proofs. This gap in understanding can be bridged by creating easily accessible tutorials. These tutorials should very quickly introduce a new user to the most important tactics, as well as, to the basic type theory behind Coq. Documents of both the extremes are available. However, a single and concise document with both the features will be of great help in popularizing the use of Coq among mathematicians. Building up of new and specialized libraries will also attract more mathematicians to use proof assistants, as they can use basic mathematical results from the library while proving a theorem. One can also make proof assistant more usable by making the interface as comfortable as possible. The more the system is used for formalizing, its practical limitations becomes explicit, and it gets even better.

Bibliography

[1] Henk Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[3] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, February 1988.

[4] Eduardo Giménez. A tutorial on recursive types in coq. 1998.

[5] J.Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge tracts in theoretical computer science. Cambridge University Press, 1989.

[6] WA Howard. To h. b. curry: Essay on combinatory logic. *Lambda Calculus and Formalisms, chapter The formulae-as-type notion of construction*, pages 479–490.

[7] Gérard Huet. Induction principles formalized in the calculus of constructions. In *TAPSOFT'87*, pages 276–286. Springer, 1987.

[8] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[9] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. pages 209–228. Springer-Verlag, 1990.

[10] D Prawitz. Natural deduction: A proof theoretical study'. *Almqvist˜ Wiksell, Stockholm*, 1965.

[11] Alfred Tarski. The semantic conception of truth: and the foundations of semantics. *Philosophy and phenomenological research*, 4(3):341–376, 1944.