上海交通大学学位论文

逻辑系统与语义的形式化

姓　　名：陶浥尘

学　　号：519030910035

导　　师：汪宇霆

学　　院：电子信息与电气工程学院

学科/专业名称：计算机科学与技术（IEEE 试点班）

申请学位层次：学士

2023 年 5 月

A Dissertation Submitted to

Shanghai Jiao Tong University for Bachelor Degree

FORMALIZATION OF LOGIC SYSTEMS AND SEMANTICS

Author: Yichen Tao

Supervisor: Yuting Wang

School of Electronic Information and Electrical Engineering

Shanghai Jiao Tong University

Shanghai, P.R. China

May 3$^{\text{rd}}$, 2023

摘 要

逻辑系统和语义的形式化是一个研究课题，它具有内在的理论意义，并有重要的实际应用。在理论上，通过使用定理证明器对逻辑理论进行形式化，可以增强对理论可靠性的信心，因为定理证明器对定义和证明提供了严格的机器检查。在应用上，逻辑系统和语义的形式化为更高层次的项目奠定了重要的基础，如形式化验证和程序分析。这些都是重要的领域，在现实世界中有许多应用。在本论文中，各种逻辑的模块化形式化是由 LOGIC 库实现的。此外，该库的实用性和可扩展性通过命题逻辑完备性的形式化构造证明和浅层嵌入量词逻辑的形式化来表现。通过展示这些例子，我们旨在强调我们的逻辑系统和语义形式化方法的通用性和潜力。

关键词： 逻辑，定理证明，Coq，形式化方法

ABSTRACT

Formalization of logic systems and semantics is a research topic that boasts inherent theoretical interest, and has significant practical applications. Theoretically, through formalizing a logical theory using a theorem prover, there is enhanced confidence in reliability of the theory, in that the theorem prover provides a rigours machine check of the definitions and proofs. Practically, formalization of logic systems and semantics lays essential foundations for higher-level projects, such as formal verification and program analysis. These are vital fields that have numerous real-world applications. In this thesis, a modular formalization of various logics is presented by the LOGIC library. Furthermore, the utility and extensibility of the library is demonstrated by means of a formal constructive proof of propositional logic completeness and a formalization of shallowly embedded quantifier logic. By showcasing these examples, we aim to highlight the versatility and potential of our approach to formalizing logic systems and semantics.

Key words: Logic, Theorem proving, Coq, Formal methods

Contents

Chapter 1 Introduction **1**

Chapter 2 Related Works **3**

2.1 Theorem Prover 3

2.2 Logic Formalization with Theorem Prover 5

2.3 Application of Formalized Logic 6

Chapter 3 Background: Coq Proof Assistant **8**

3.1 Inductive Definition and Pattern Matching 8

3.2 Type Class 9

3.3 Module System 11

Chapter 4 The LOGIC Library **12**

4.1 Overview 12

4.2 Pararmeterized Definitions and Proofs 13

4.2.1 Connectives and Judgements 14

4.2.2 Proof Rules 16

4.3 Logic Generator 19

4.3.1 Features of Logic Generator 19

4.3.2 Implementation of Logic Generator 24

Chapter 5 Completeness of Propositional Logic **25**

5.1 Syntax and Inference Rules 25

5.2 Semantics 27

5.3 The Completeness Theorem 28

5.3.1 Construction 28

5.3.2 Main Lemma 29

5.3.3 Proof of Completeness Theorem 32

5.4 Formalization in Coq 33

5.4.1 Formalization of the Propositional Logic 34

5.4.2 Formalization of the Construction 36
5.4.3 Formal Proof of Completeness 38

Chapter 6 Formalized Quantifier Logic........ 40
6.1 Syntax and Proof Rules........ 40
6.2 Integration to Logic Generator 41

Chapter 7 Conclusion 43

Bibliography 44

Acknowledgements........ 50

Chapter 1 Introduction

Researchers from different fields, including mathematics and computer science, have utilized theorem provers to formalize intricate mathematical proofs, thereby ensuring the correctness of their theories. Some of the most widely used theorem provers are Isabelle[1], Agda[2], Coq[3], and Lean[4]. Among various formalization efforts, the formalization of logics has emerged as a prominent research topic. There are numerous scenarios that requires a formalized logic. For instance, when verifying programs with address manipulations, we will need a separation logic whose soundness is formally proved; when studying the meta-theories of logics, the formalized syntax, semantics, and inference rules are demanded. Therefore, this thesis presents a Coq formalization of logics, concerning both logic applications and logic meta-theories. More specifically,

- a foundational Coq-based logic library LOGIC is developed so that there can be higher-level results on logics based on it;
- a logic generator is developed, which employs the definitions and proofs in LOGIC to automatically generate an exportable logic formalization;
- a completeness proof of a propositional logic is formalized based on the LOGIC library;
- the syntax, rules, and derivation between rules of a shallowly-embedded quantifer logic are formalized that act as an extension to the LOGIC library.

The first two points above are part of an already published project[5], while the other two are incremental to that.

The primary objective of the LOGIC library is to give a *uniform* formalization of different logics, namely with the same set of classes to be instantiated differently so as to satisfy distinct user requirements. This would enable optimal reuse of existing definitions and proofs, providing considerable benefits in proof engineering. However, this design choice also poses great technical challenge, since different use cases may necessitate different constructions of logics. For example, the Hilbert system select "provability" ($\vdash \varphi$) as the primitive judgement, defined by axioms, and define "derivability" ($\Phi \vdash \varphi$) as follows

$$\Phi \vdash \varphi \text{ iff. exists finite } \Psi \subseteq \Phi \text{ s.t. } \vdash \left(\bigwedge_{\psi \in \Psi} \psi\right) \to \varphi.$$

Nonetheless, in the case of sequent calculi, derivability ($\Phi \vdash \varphi$) is chosen as the primitive judgement, and provability is defined as

$$\vdash \varphi \text{ iff. } \emptyset \vdash \varphi.$$

Hence, we need to figure out how to develop a method so that both of these logic systems can be formalized in a uniform manner.

To illustrate the use of LOGIC, a formalized constructive completeness proof of the propositional logic is given. Completeness is a crucial notion in the field of mathematical logic. Intuitively, completeness refers to the property of a logic system such that all statements that are valid under a certain semantics are also provable in the logic. This notion is important because it ensures that the logic system is sufficiently powerful to prove any semantically true statement. Furthermore, completeness establishes a fundamental connection between the syntax and semantics.

In the rest of the thesis, we will first discuss related works in §2. Then we give some background about the Coq proof assistant in §3. After that, we introduce the LOGIC library in §4, demonstrate the completeness proof in §5, and describe the formalization of quantifier logic in §6. Finally, we conclude the thesis in §7.

Chapter 2 Related Works

2.1 Theorem Prover

In the field of mathematics and computer science, theorem provers refer to software systems that aid in theorem proving by mechanically checking the correctness of proofs. This approach provides greater reliability to the proofs being checked compared to those done by pen and paper.

The emergence of theorem provers dates back to the 1950s. In 1956, Logic Theorist (LT)[6], written by Newell and Simon is the first program aiming at automated reasoning, and sometimes is even cited as “the first artificial intelligence program”. Its impressive proof capability is demonstrated by proving 38 of 52 theorems in Whitehead and Russell’s renowned book Principia Mathematica[7]. Notably, for some of the theorems, LT even found a proof that is more concise and elegant than originally presented in the book. One year late, General Problem Solver (GPS)[8] also created by Newell et al, was introduced. It differs from its predecessor technically, and is capable of solving any problem expressible as well-formed formulae. The typical examples of this kind or problems include predicate logic problems and Euclidean geometry problems.

The 1970s witnessed the first generation of *interactive* theorem provers (also known as proof assistants). Logic of Computable Functions (LCF)[9], developed by Milner et al., is one of the most significant among them. Its theoretical foundation, also called Logic of Computable Functions[10], was previously introduced by Scott. LCF has led to many future theorem prover projects, which will be detailed later. Automath[11], devised by De Brujin, is another interactive theorem prover that emerged in this decade. Many notions introduced in Automath, including typed lambda calculus, explicit substitution, dependent typing, were of great importance in future research. Boyer-Moore theorem prover (also known as Nqthm)[12-14], was yet another theorem prover in that period. It is designed to be a Lisp-based fully-automatic theorem prover.

During the 1980s and 1990s, several other important theorem provers emerged, including Coq[3], which is employed as the proof assistant for implementing the formalization in this thesis. Coq is based on the Calculus of Inductive Constructions (CIC), a variant of lambda calculus, implemented in OCaml, and provides extensive features, including higher-order logic, dependent typing, proof automation, code generation, etc. Since its origination by the French Institute for Research in Computer Science and Automation (INRIA), Coq has become one of the most popular proof assistants in the world, particularly in the field of formal verification. An extensive corpus of works in formalized software systems have been carried out using Coq, such as operating systems[15-16], compilers[17], verification tools[18], file systems[19], etc. Its continued development and evolution, along with its growing ecosystem, make Coq a valuable and important tool in the quest for rigor and correctness in software systems. Nevertheless, its continued development and evolution, along with its growing ecosystem of libraries and tools, make Coq a valuable and important tool in the quest for rigor and correctness in software systems and beyond.

Isabelle[1], initially released in 1986, is an LCF-style interactive theorem prover, implemented in Standard ML and Scala. It is generic in that it provides a meta-logic that supports numerous object logics, including first-order logic, higher-order logic (HOL)[20], and Zermelo–Fraenkel set theory, among which Isabelle/HOL is the most prevalently used. Isabelle also boasts a vibrant user community, having done numerous formalization projects, including Hoare logic verification[21], category theory formalization[22], number theory development[23], cryptographic protocol verification[24], and so on. The Archive of Formal Proofs documents many of the formalization projects in Isabelle, containing over 2 million lines of proofs.

Other important theorem provers that appeared in this period include NuPrl (1980s)[25], ACL2 (1990) (A Computational Logic for Applicative Common Lisp)[26], PVS (1992) (Prototype Verification System)[27].

The new century came with other modern and advanced theorem provers. Agda[2], whose current version (Agda 2, with great difference from the original Agda 1) was released in 2007, is an interactive theorem prover based on Curry-Howard correspondence (viewing propositions as types). However, unlike other trending proof assistants, Agda does not has a separation tactic language. Proofs in Agda are written in a functional style, i.e. applying proof constructs on proof terms of sub-goals. It is based on the Unifying Theory of Dependent Types (UTT)[28], which is similar to the classic Martin-Löf Type Theory (MLTT)[29].

Lean[4] was developed and released by Microsoft Research in 2013. It is open source with a user-friendly interface in Visual Studio Code supporting LaTeX-like notations, and has a robust tactic language enabling automated proof search and simplification. There is a library *mathlib*[30] for mathematics formalization in Lean, that is developed and maintained by the Lean users. It contains over 100,000 theorems and 1,000,000 lines of code.

The theorem provers have provided researchers and engineers powerful tools for formal reasoning, software verification and programming language theory research. With the increasing popularity and expanding features, these theorem provers are expected to play a more significant role in the future of both the academia and the industry.

2.2 Logic Formalization with Theorem Prover

Formal reasoning of logics has been a topic of interest for researchers in various fields, including computer science, mathematics, and philosophy. In recent years, the use of interactive theorem provers has provided new opportunities for formalizing logic systems, leading to a surge in research in this area.

One of the most prominent theorem provers used in logic formalizations is Coq. As early as in 1999, Power and Webster[31] have utilized Coq to formalize linear logic, which is a substructual logic acting as a refinement of classical and intuitionistic logic. Forster et al.[32][33] have proved the completeness and undecidability property of first-order logic in Coq. Jensen[34] devoted their Ph.D. studies to formalization of a wide range of separation logics using Coq, and Andrade[35] did a formalization of justification logic. Besides, a lot of works have been done regarding modal logic. Tews[36] formalized cut elimination for propositional multi-modal logic in Coq. Benzmüller and Paleo[37] formalized modal logic using Coq and developed a formal proof of Godel's ontological argument based on that. De Alemida Borges[38] formalized a quantified modal logic, its Kripke semantics, and provided a formal proof of its soundness theorem.

Furthermore, Coq has been extensively used in computer science education, particularly in logic related courses. Hendriks, KalisZyk, Raamsdonk, and Wiedijk[39] have developed ProofWeb, a Coq-based system for teaching logic courses to undergraduate computer science students. This system is made available to students through a web interface, and provides logic problems and hold solutions to them. Henz and Hobor[40] also taught courses on formal methods using a formalization in Coq at the National University of Singapore. The famous fully formalized textbook, *Software Foundations*[41], currently comprises six volumes, focusing on various topics like logical foundations, programming languages foundations, C program verification using VST, separation logics, etc. By the way, there are also important mathematical theorems that get formalized in Coq. Gonthier[42] has formalized the four color theorem, which is one of the most significant results in the field of combinatorics. Gonthier and his collaborators[43] have also formalized the odd order theorem in Coq.

Besides Coq, researchers often use Isabelle when formalizing logics. In 2002, Nipkow[44] formalized Hoare Logic for some constructs of imperative programming languages In Isabelle/HOL, and formally proved the soundness and completeness of the logic. Benzüller and Claus[45] formalized higher-order multi-modal logic in Isabelle, and provided a proof automation library for it. Blanchette, Pspescu, and Traytel[46] employed codatatypes in Isabelle/HOL to produce a formal proof of soundness and completeness properties of variations of the first-order logic. Schlichtkrull[47] has his Ph.D. dissertation focusing on the first-order logic formalization using Isabelle. Specificaly, the followings regarding logic formalization have been done in the dissertation: a formalization of resolution calculus and a formal proof of its soundness and completeness; a formalization of the ordered resolution calculus; a verified automatic theorem prover for first-order logic.

Agda is yet another proof assistant used in logic formalization. Bove et al.[48] formalized Aczel's Logical Theory of Constructions (LTC) with Agda, and embedded LTC's inductive notions and totality with Agda's inductive notions. Kokke[49] embedded the Lambek-Grishin calculus, a grammar logic, in Agda, and presented and verified a cut elimination procedure in the calculus. Pope[50] has formally proved how to generalize the elimination of a single existential quantifier to full elimination of quantifiers using Agda, based only on a theory of natural numbers. Due to its lack of a tactic language, Agda is not so frequently used in logic formalization as other theorem provers mentioned before.

2.3 Application of Formalized Logic

Formal verification of programs is one of the most important applications of formalized logics. VST[18,51] employs Verifiable C[52], a concurrent separation program logic designated for verification, to allow users to verify C programs modularly and foundationally, while providing a rich tactic library for proof automation. Bedrock[53] is designed for low level program verification. VeriFast[54] is a separation-logic based verification tool for C and Java programs, annotated with pre- and post-conditions written in separation logic. It allows rich, user-defined specifications. KeY[55] is a Java verification tool, providing various functionanlities, including type checking, deductive verification, and symbolic execution. It is based on the Java Modeling Language (JML), an extension of Java that enables annotations as program specifications. The logical foundation of KeY is Hoare Logic and its extensions.

Formalized logics have also been used in other research topics including program analysis and program synthesis. Frama-C[56] provides a platform for analyzing C programs, integrating several static and dynamic analysis techniques. It supports value analysis, effect analysis, dependency analysis, and more. ESC/Java[57] (Extended Static Checker for Java) is a static analysis tool that checks for run-time errors that are commonly seen in Java programs. Synquid[58] is a program synthesis tool that is capable of synthesizing a program according to the given specification. Cypress[59] is another program systhesis tool based on synthetic separation logic, and performs improved synthesis capability compared to its predecessors.

Chapter 3 Background: Coq Proof Assistant

The Coq proof assistant is employed to implement all the formalizations discussed in this thesis. This chapter introduces some fundamental aspects of Coq that are relevant to the works presented in this thesis.

3.1 Inductive Definition and Pattern Matching

Inductive definition is one of the fundamental features of Coq. It allows creating a type from the functions that act of the constructors of that type. Notice that the type created in this way is *minimal*, i.e., only the terms created by the constructors are elements of the type. A typical example of inductive type definition is definining natural numbers with Peano's encoding.

```
Inductive nat : Type :=
  | O : nat
  | S : nat -> nat .
```

With `O` representing zero, and `S` representing the successor of a natural number, this definition says the followings.

- `O` is an element of `nat`.
- If `n` is an element of `nat`, then `S n` is an element of `nat`.
- All elements of `nat` are either `O` or created by applying `S` to another `nat` element.

This encodes the natural numbers in a "unary" manner. `O` stands for 0, `S O` stands for 1, `S (S O)` stands for 2, etc. All natural numbers are covered in this definition.

(Recursive) pattern matching is the most common way of handling inductive definitions. For example, addition of two natural numbers `n` and `m` can be defined as follows.

```
Fixpoint add (n : nat) (m : nat) : nat :=
  match n with
  | O => n
  | S n' => S (add n' m)
  end.
```

This definition pattern matches the first argument, and consideres the two cases corresponding to the two constructors of `nat` definition.

- If `n` is `O`, then `add n m` evaluates to `m`.
- If `n` is `S n'`, then `add n m` evaluates to `S (add n' m)`, and `add n' m` will be recursively evaluated.

Since the inductive definition of `nat` type is minimal, the above two cases suffice to cover all possible situations. To give a more intuitive sense of this definition, the procedure of evaluating `add (S (S O)) (S O)` is demonstrated.

```
   add (S (S O)) (S O)
⟶ S (add (S O) (S O))
⟶ S (S (add O (S O)))
⟶ S (S (S O))
```

3.2 Type Class

Type class is a kind of higher order object in Coq, usually used to formalize abstract structures. It allows the same parameter naming of different instances of the same class, due to which it acts as the workhorse for formalization in this thesis.

To give an example, a formalization of the notion of categories using type classes is given. First, we state the definition of categories in textbook of Awodey[60].

Definition 3.1 (Category) A *category* consists of the following data:

- *Objects*: $A, B, C, \ldots$
- *Morphisms*: $f, g, h, \ldots$
- For each morphism $f : A \longrightarrow B$, there are given objects $\mathrm{dom}(f) = A$ and $\mathrm{cod}(f) = B$, called the *domain and codomain* of f.
- Given morphisms $f : A \longrightarrow B$ and $g : B \longrightarrow C$, there is given arrow

$$g \circ f : A \longrightarrow C$$

  called the *composite* of f and g.
- For each object A, there is given an morphism

$$1_A : A \longrightarrow A$$

called the *identity morphism* of A.

- Associativity:

$$h \circ (g \circ f) = (h \circ g) \circ f$$

  for all $f : A \to B$, $g : B \to C$, $h : C \to D$.

- Unit:

$$f \circ 1_A = f = 1_B \circ f$$

  for all $f : A \to B$.

To formalize the above notion of categories, we define two type classes - the first accommodating the operations required in a category, and the second specifying the property for the operations to satisfy to actually be a category.

```
Class CatOp : Type := {
  obj : Type;
  mph : obj -> obj -> Type;
  dom {x y : obj} : mph x y -> obj;
  cod {x y : obj} : mph x y -> obj;
  idt (x : obj) : mph x x;
  cps {x y z : obj} : mph y z -> mph x y -> mph x z }.
Class CatProp (C : CatOp) : Type := {
  dom_id : forall x, dom (idt x) = x;
  cod_id : forall x, cod (idt x) = x;
  cp_id1 : forall x y (f : mph x y), cps f (idt x) = f;
  cp_id2 : forall x y (f : mph x y), cps (idt y) f = f;
  dom_cp : forall x y z (f : mph x y) (g : mph y z),
    dom (cps g f) = dom f;
  cod_cp : forall x y z (f : mph x y) (g : mph y z),
    cod (cps g f) = cod g;
  assoc : forall w x y z (f : mph w x) (g : mph x y) (h : mph y z),
    cps h (cps g f) = cps (cps h g) f }.
```

An instance is an "implementation" of a class. To instantiate a class, one needs to fill all the parameters with concrete definitions. For example, we can formalize the category of sets as follows.

```
Instance Sets : CatOp := {
  obj := Set;
  mph := fun x y => x -> y;
  dom := fun x y f => x;
  cod := fun x y f => y;
  idt := fun x => (fun a : x => a);
  cps := fun x y z g f a => g (f a) }.
```

The objects are defined as sets, and morphisms are defined to as funcions between sets. Identity is defined to be the identity function, and composite as function composition. Then we prove that this instance satisfies the specification of a category.

```
Instance SetsCat : CatProp Sets.
Proof. ... Qed.
```

Similarly, the classes `CatOp` and `CatProp` can be instantiated to other concrete categories, such as category of types, monoids, etc.

It can be observed from the above example that type class definitions describe the common characteristics of instances of a class. This closely aligns with the goal of formalizing different logics in a uniform manner.

3.3 Module System

A *module system* is provided by Coq so that it is more convenient to structure large formalization developements. In short, a *module* consists of a collection of definitions and proofs. *Module types* serve as signatures of module, including unspecified "parameters" only whose type is indicated, and unproved "axioms" only whose proposition is stated. Module types can act as the input to a *functor* that takes the assumed existing definitions and proofs in a module type, and derives further definitions and proofs based on them.

The use of module systems helps managing large-scale, complex formalization projects, and promotes modularity of the whole development. It also enhances reusability of codes by allowing modules to be combined and reused in different contexts.

Chapter 4 The LOGIC Library

4.1 Overview

The principal goal of LOGIC and logic generator is to reuse the existing definitions and proofs to generate exportable logic formalizations, while enabling flexibility in the construction of the logic. To illustrate the underlying design principle of LOGIC, we consider a propositional logic. Assume the logic comprises three connectives: implication ($\rightarrow$), negation ($\neg$), and disjunction ($\vee$), and we want two distinct formalzations of this logic - the first one follows the approach described in Mendelson's textbook[61], which considers implication and negation as primitive connectives and derives disjunction as

$$\varphi \vee \psi \triangleq \neg\varphi \rightarrow \psi;$$

while the second formalization follows the method in the book of Ebbinghaus, Flum and Thomas[62], which views negation and disjunction as primitive connectives and define implication as

$$\varphi \rightarrow \psi \triangleq \neg\varphi \vee \psi.$$

In addition, we hope that a variety of proof system constructions are enabled. Consider the following three judgements:

- "Provability": write $\vdash \varphi$ is φ is provable;
- "Derivability from a set of propositions": write $\Phi \vdash \varphi$ if φ is derivable from a finite set of propositions Φ;
- "Derivability from a single proposition": write $\psi \vdash \varphi$ if φ is derivable from a singleton proposition ψ.

We wish to enable formalizing these judgements by selecting any one of them as primitive judgements, and deriving the others from this foundation.

It seems a possible solution to employ various modules and type classes that incoporate paramterized definitions and proofs of different logics. This has indeed been utilized in certain projects, such as VST-MSL[52] and Iris[63-64]. Despite that, there are two major shortcomings of this approach:

- It would require expertise in the entire framework to make use of the LOGIC library.

The user would have to possess familiarity with all the design details to be able to select the proper type classes and fill in the place-holder of the arguments.

- Constructing proof terms via proof term combination would incur an unacceptably large overhead, which grows exponentially with the number of implicit arguments to be filled.

To address the aforementioned problems, it becomes indispensable to implement a *logic generator*, which accepts a *configuration* designated by the user, integrates the corresponding type classes, and outputs an *interface* for constructing the logic. This obviates the users' effort to be acquainted with the entire framework. Rather, with the help of the interface, users can use the system compositionally and derive the demanded logic formalizations. The logic generator alleviates the users' burden of searching for the correct class and constructing the proof terms themselves. Their only task, is writing the configuration, and implementing the primitive definitions and proofs.

4.2 Pararmeterized Definitions and Proofs

As previously mentioned, a type class based parameterized formalization of logic definitions and proofs is given the LOGIC. However, due to the versatility of logic application scenarios, traditional applications of type classes do not suffice to solve the problems. Thus, the logics are divided into multiple layers, and different design choices are made based on their applications. We talk about formalization of connectives and judgements in §4.2.1, and discuss how proof rules are formalized in §4.2.2.

Notations in Coq and in this paper

We use "`orp x y`" to represent the disjunction of `x` and `y` in LOGIC's proposition, where "p" stands for "proposition". Additionally, Coq's infrastructure allows us to define a notation for the connectives and judgements. For example, we use "`x||y`" as an object logics' notation to represent "`orp x y`" in LOGIC, which is distinguished from Coq's notation for its own meta-logic. In this paper, for conciseness, we choose to use standard logic notations, such as $\vee, \wedge$ for object languages, and English words "or", "and" for the meta language. For consideration of readability, we still adopt this convention (within a box container) when we present Coq code.

4.2.1 Connectives and Judgements

The example in Section 4.1 is used here to illustrate our design choice for formalizing connectives and judgements. Suppose we want to formalize a logic with connectives $\to$, $\neg$, $\vee$, and the following two schemes are considered:

- (Mendelson) treat $\neg$, $\to$ as primitive connectives, and derive $\vee$ from them as shown below
$$\varphi \vee \psi \triangleq \neg\varphi \to \psi;$$
- (Ebbinghaus) treat $\neg$, $\vee$ as primitive connectives, and derive $\to$ from them as shown below
$$\varphi \to \psi \triangleq \neg\varphi \vee \psi.$$

A straight-forward way to formalize these two constructions would be defining a type class for each of the schemes' primitive connectives, and an extra definition for the derived connective.

```
Class Mendelson_Language := {
  M_expr : Type;
  M_negp : expr -> expr;
  M_impp : expr -> expr -> expr; }.
Definition M_orp := fun p q => $\boxed{\neg p \to q}$.
Class Ebbinghaus_Language := {
  E_expr : Type;
  E_negp : expr -> expr;
  E_orp  : expr -> expr -> expr; }.
Definition E_impp := fun p q => $\boxed{\neg p \vee q}$.
```

However, the method above is not sufficient to accommodate more sophiticated and versatile logic constructions, and neither does it provide a uniform formalization. Therefore, an alternative approach is taken. Languages and connectives are defined respectively with different type classes, i.e., there is one type class for the language, and one type class for each of the connectives.

```
Class Language := { expr : Type }.
Class NegLanguage (L : Language) :=
  { negp : expr -> expr }.           (* $\boxed{\neg\varphi}$ *)
Class ImpLanguage (L : Language) :=
  { impp : expr -> expr -> expr }.   (* $\boxed{\varphi \to \psi}$ *)
```

```
Class OrLanguage  (L : Language) :=
  { orp  : expr -> expr -> expr }.  (* $\boxed{\varphi \vee \psi}$ *)
```

Here, the class "`Language`" provides a means of defining the language once the set of expressions (`expr`) has been defined. Besides, the "`NegLanguage`" type class only signifies that the negation connective is a construct in language `L`, without any indication whether it is primitive or derived. Additional type class, dubbed *refl classes*, are used in LOGIC to formalize the derivation between connectives. For example, the following refl class suggests how disjuction is derived by implication and negation.

```
Class OrDef_Imp_Neg
  (L : Language)
  {_ : ImpLanguage L}
  {_ : NegLanguage L}
  {_ : OrLanguage L} :=
  { impp_negp2orp : $\boxed{\text{for any } \varphi\ \psi, \varphi \vee \psi = \neg\varphi \to \psi}$ }.
```

Seven propositional connectives and constants ($\wedge$, $\vee$, $\to$, $\leftrightarrow$, $\neg$, $\top$, $\bot$) and two separation logic connectives ($*$, $-\!\!*$) are supported in LOGIC. For brevity, only a part of them have their formalizations demonstrated here.

As for judgements, a similar approach is adopted - a type class is defined for each of the judgements in the LOGIC library.

```
Class Provable (L : Language) :=
  { provable : expr -> Prop }.                (* $\boxed{\vdash \varphi}$ *)
Class Derivable1 (L : Language) :=
  { derivable1 : expr -> expr -> Prop }.      (* $\boxed{\psi \vdash \varphi}$ *)
Class Derivable (L : Language) :=
  { derivable : set_of_expr -> expr -> Prop}. (* $\boxed{\Phi \vdash \varphi}$ *)
```

Again, the type class does not assume the corresponding judgement to be primitive or derived. The derivations are supported with addtional type classes defining the transformations. The following is an example.

```
Class DerivableProvable
  (L : Language)
  (GammaP : Provable L)
  (GammaD : Derivable L)
  {_ : ImpLanguage L} :=
  { derivable_provable: $\boxed{\text{for any } \Phi\ \varphi,\ \Phi \vdash \varphi \text{ iff.}}$
    $\boxed{\text{there exists } \varphi_1, \varphi_2, \ldots, \varphi_n \in \Phi, \text{ s.t. } \vdash \varphi_1 \to \varphi_2 \to \cdots \to \varphi_n \to \varphi}$ }.
```

Logicians may select either $\vdash \varphi$ or $\Phi \vdash \varphi$ as the primitive definition, and derive the other. Proof engineers typically use $\psi \vdash \varphi$ in program verification. All these different choices are supported by LOGIC.

4.2.2 Proof Rules

The proof rules in LOGIC are divided into primary proof rules and derived ones. The rules are also categorized into Coq type classes. The difference from how we handle connectives and judgements is that one type class may include multiple rules. This design choice is made because certain collection of rules rarely appear separately in logics. For example, in a logic with disjunction as one of its connectives, and sequent calculus as its primitive proof system, the following three inference rules are almost always presented together.

$$\frac{\Phi;\varphi \vdash \chi \quad \Phi;\psi \vdash \chi}{\Phi;\varphi \vee \psi \vdash \chi}(\textsc{OrElim}) \qquad \frac{\Phi \vdash \varphi}{\Phi \vdash \varphi \vee \psi}(\textsc{OrIntro1}) \qquad \frac{\Phi \vdash \psi}{\Phi \vdash \varphi \vee \psi}(\textsc{OrIntro2})$$

The following type class is designed to accommodate these rules.

```
Class OrDeduction
  (L : Language)
  {_ : OrLanguage L}
  (GammaD : Derivable L} :=
  { orp_elim : $\boxed{\text{for any } \Phi\ \varphi\ \psi\ \chi, \text{ if } \Phi;\varphi \vdash \chi \text{ and } \Phi;\psi \vdash \chi \text{ then } \Phi;\varphi \vee \psi \vdash \chi}$;
    orp_intro1 : $\boxed{\text{for any } \Phi\ \varphi\ \psi, \text{ if } \Phi \vdash \varphi \text{ then } \Phi \vdash \varphi \vee \psi}$;
    orp_intro2 : $\boxed{\text{for any } \Phi\ \varphi\ \psi, \text{ if } \Phi \vdash \psi \text{ then } \Phi \vdash \varphi \vee \psi}$; }.
```

Note that there are such classes for all combinations of connectives and judgements in LOGIC.

For sake of automatic derivation of rules regarding the derived connectives/judgements, there are Coq lemmas are portray how these derivations are done. For example, the following Coq lemma says that if disjunction is derived from implication and negation, then the three rules above hold.

```
Lemma OrFromNegImp
  (L : Language)
  {_ : OrLanguage L}
  {_ : NegLanguage L}
  {_ : ImpLanguage L}
  (GammaD : Derivable L)
  {_ : NegDeduction L GammaD}
    (* Negation satisfies the primary rules *)
  {_ : ImpDeduction L GammaD}
    (* Implication satisfies the primary rules *)
  {_ : OrDef_Imp_Neg L} :
    (* Disjunction is derived from negation and implication *)
  OrDeduction L GammaD.
```

When dealing with separation logic, things get more tricky. The following three inference rules regarding the separating conjunction connective ($*$) are often necessary when formalizing a separation logic.

$$\frac{\Phi \vdash \varphi * \psi}{\Phi \vdash \psi * \varphi}(\textsc{SepconComm}) \qquad \frac{\Phi \vdash \varphi * (\psi * \chi)}{\Phi \vdash (\varphi * \psi) * \chi}(\textsc{SepconAssoc})$$

$$\frac{\Phi; \varphi \vdash \varphi' \quad \Phi; \psi \vdash \psi'}{\Phi; \varphi * \psi \vdash \varphi' * \psi'}(\textsc{SepconMono})$$

In some cases, there is another connective, separating implication ($-\!\!*$, also know as "wand") in the separation logic. We may have the following rule on the adjointness of separating conjunction and separating implication.

$$\frac{\Phi; \varphi * \psi \vdash \chi}{\Phi; \varphi \vdash \psi -\!\!* \chi}(\textsc{WandSepconAdjoint})$$

It is known that SepconMono can be derived from SepconComm, SepconAssoc and WandSepconAdjoint. Therefore, a more intricate design is necessary to accommodate the potential for various constructions. The rule classes here are divided into two categories, as described below.

- **Rule classes for internal use**. These classes serve the internal use of the LOGIC library, e.g. reasoning about derivations between rule class.

```
  Class SepconSequentCalculus
    (L : Language)
    (GammaD : Derivable L)
    {_ : SepconLanguage L} :=
    { sepcon_comm : for any Φ φ ψ, if Φ ⊢ φ*ψ, then Φ ⊢ ψ*φ;
    sepcon_assoc : for any Φ φ ψ χ, if Φ ⊢ φ ∗ (ψ ∗ χ),
      then Φ ⊢ (φ ∗ ψ) ∗ χ;
    sepcon_mono : for any Φ φ ψ φ′ ψ, if Φ;φ ⊢ φ′ and Φ;ψ ⊢ ψ′,
      then Φ;φ ∗ ψ ⊢ φ′ ∗ ψ′; }.
Class WandSequentCalculus
  (L : Language)
  (GammaD : Derivable L)
  {_ : WandLanguage L} :=
  { wand_sepcon_adjoint : for any Φ φ ψ χ, if Φ;φ ∗ ψ ⊢ χ,
    then Φ;φ ⊢ ψ −∗ χ; }.
```

There is some redundancy among the rule classes for internal use, e.g. SepconMono can be derived from the other rules. Such redundancy is allowed so as to exhibit a clearer hierarchical structure of the rule classes.

- **Rule classes for users' constructions**. These classes are for the use of users. Specifically, the logic generator (described in §4.3) would use these rule classes to construct the logic demanded by the users.

```
Class SepconSC_Weak
  (L : Language)
  (GammaD : Derivable L)
  {_ : SepconLanguage L} :=
  { __sepcon_comm : for any Φ φ ψ, if Φ ⊢ φ ∗ ψ, then Φ ⊢ ψ ∗ φ;
    __sepcon_assoc : for any Φ φ ψ χ, if Φ ⊢ φ ∗ (ψ ∗ χ),
      then Φ ⊢ (φ ∗ ψ) ∗ χ; }.
Class SepconSC_Mono
  (L : Language)
  (GammaD : Derivable L)
  {_ : SepconLanguage L} :=
  { __sepcon_mono : for any Φ φ ψ φ′ ψ, if Φ;φ ⊢ φ′ and Φ;ψ ⊢ ψ′,
      then Φ;φ ∗ ψ ⊢ φ′ ∗ ψ′; }.
```

The rules are divided differently from the classes for internal use, in order to eliminate redundancy. If the user wants a separation logic without the separating implication, then they can choose to include rule cases `SepconSC_Weak` and `SepconSC_Mono` in their logic. Alternatively, if the separating implication is present, it would be advisable to include the rule classes `SepconSC_Weak` and `WandSequentCalculus`, and use an additional Coq lemma in LOGIC to derive the rule SEPCONMONO. Since there is no redundancy, the user if free to select the rule classes without concerning about the possibility of repetitive proofs.

4.3 Logic Generator

As described in §4.1, there is a logic generator in the LOGIC library. The features of the logic generator is illustrated with a running example in §4.3.1. The design and implementation of the logic generator is briefly discussed in §4.3.2.

4.3.1 Features of Logic Generator

Suppose one wants to build a logic with four primitive connectives (and logical constants, which are regarded as connectives in LOGIC's framework): implication ($\rightarrow$), conjunction ($\wedge$), disjunction ($\vee$) and falsehood ($\perp$). There are three derived connectives:

$$\varphi \leftrightarrow \psi \triangleq (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi),$$

$$\neg\varphi \triangleq \varphi \rightarrow \perp,$$

$$\top \triangleq \perp \rightarrow \perp.$$

The proof system for the logic is based on sequent calculus, i.e. the primitive judgement is "derivability" ($\Phi \vdash \varphi$), and there are primitive rules for each of the primitive connectives regarding derivability.

To generate an exportable logic library formalizing the above logic, three steps need to be done. The first step is writing a *configuration* to indicate how the logic is to be constructed. The critical definitions in the configuration file for the aforementioned logic is shown as follows.

```
Definition how_connectives :=
  [ primitive_connective impp;
    primitive_connective andp;
    primitive_connective orp;
    primitive_connective falsep;
    FROM_andp_impp_to_iffp;
    FROM_falsep_impp_TO_negp;
    FROM_falsep_impp_TO_truep ].
Definition how_judgements :=
  [ primitive_judgement derivable ].
Definition primitive_rule_classes :=
  [ derivitive_OF_basics;
    derivitive_OF_impp;
    derivitive_OF_andp;
    derivitive_OF_orp;
    derivitive_OF_falsep ].
```

The configuration specifies the followings about the desired logic.

- The list `how_connectives` indicates that the primitive connectives are $\to, \wedge, \vee, \perp$, and derived connectives are $\leftrightarrow, \neg, \top$. How the derived connectives are derived is also specified.
- The list `how_judgements` indicates that the primitive judgement is $\Phi \vdash \varphi$, and there is no derived judgement.
- The list `primitive_rule_classes` specifies the primary rules of the logic. Each entry in the list corresponds to a rule class. For example, `derivitive_OF_basics` includes the following three inference rules on the basic setting of the logic.

$$\frac{\varphi \in \Phi}{\Phi \vdash \varphi}(\textsc{Assu}) \qquad \frac{\Phi \subseteq \Psi \quad \Phi \vdash \varphi}{\Psi \vdash \varphi}(\textsc{Weaken}) \qquad \frac{\Phi \vdash \Psi \quad \Phi \cup \Psi \vdash \varphi}{\Phi \vdash \varphi}(\textsc{Subst})$$

  The entry `derivitive_OF_andp` includes the introduction and elimination rules for the conjunction connective.

$$\frac{\Phi \vdash \varphi \wedge \psi}{\Phi \vdash \varphi}(\textsc{AndElim1}) \qquad \frac{\Phi \vdash \varphi \wedge \psi}{\Phi \vdash \psi}(\textsc{AndElim2}) \qquad \frac{\Phi \vdash \varphi \quad \Phi \vdash \psi}{\Phi \vdash \varphi \wedge \psi}(\textsc{AndIntro})$$

When the configuration is fed into the logic generator, it outputs an *interface*. The interface includes Coq module types for primitives of the logic that are to be implemented by the users, and Coq modules for derived's of the logic that utilizes the type class system to implement the derivations automatically. There is a module type `LanguageSig` for primitive connectives and judgements.

```
Module Type LanguageSig.
  Parameter expr : Type .
  Definition context := (expr -> Prop) .
  Parameter derivable : (context -> expr -> Prop) .
  Parameter impp : (expr -> expr -> expr) .
  Parameter andp : (expr -> expr -> expr) .
  Parameter orp : (expr -> expr -> expr) .
  Parameter falsep : expr .
End LanguageSig.
(* automatically generated *)
```

There is also a module type `PrimitiveRulesSig` for primary rules.

```
Module Type PrimitiveRuleSig (Names: LanguageSig).
Include DerivedNames (Names).
  Axiom deduction_falsep_elim : ...
  Axiom deduction_orp_intros1 : ...
  Axiom deduction_orp_intros2 : ...
  Axiom deduction_orp_elim : ...
  Axiom deduction_andp_intros : ...
  Axiom deduction_andp_elim1 : ...
  Axiom deduction_andp_elim2 : ...
  Axiom deduction_modus_ponens : ...
  Axiom deduction_impp_intros : ...
  Axiom deduction_weaken : ...
  Axiom derivable_assum : ...
  Axiom deduction_subst : ...
End PrimitiveRuleSig.
(* automatically generated, types of axioms omitted for brevity *)
```

The module `DerivedNames` specifies the derivation of derived connectives and derived judgements, and how they are derived. The derived judgements are not shown here since there is no derived judgement in this logic.

```
Module DerivedNames (Names: LanguageSig).
Include Names.
  Definition iffp := (fun x y : expr => andp (impp x y) (impp y x)) .
  Definition negp := (fun x : expr => impp x falsep) .
  Definition truep := (impp falsep falsep) .
End DerivedNames.
```

```
(* automatically generated *)
```

Then there are the module type `LogicTheoremSig` for axiomatized derivable rules from the primitive ones, and module `LogicTheorem`, which acts as an instantiation of the module type `LogicTheoremSig`. The code of these two modules are omitted, and we list some of the rules included in them to give a taste of what can be derived automatically with the help of the logic generator.

$$\frac{\Phi \vdash \varphi \to \psi \quad \Phi \vdash \psi \to \chi}{\Phi \vdash \varphi \to \chi}(\textsc{ImpTrans}) \quad \frac{\Phi \vdash \varphi \to (\psi \to \chi)}{\Phi \vdash \psi \to (\varphi \to \chi)}(\textsc{ImpArgSwitch})$$

It is worth mentioning that the derivable rules are found automatically based on what primitive definitions are provided.

Guided by the interface, the users need to implement the definitions of primitive connectives and primitive judgements, and prove the primary proof rules. That is to say, they have to fill the parameterized and axiomatized terms in the module types of the interface. The implementation can be done in either shallow embeddings (where propositions are defined as sets of models satisfying the propositions) or deep embeddings (where propositions are defined using abstract syntax trees). If one employs shallow embeddings, the implementation can be written as follows.

```
Module NaiveLang.
  Definition expr : Type := model -> Prop.
  Definition context : Type := expr -> Prop.
  Definition impp (x y : expr) : expr := fun m => x m -> y m.
  Definition andp (x y : expr) : expr := fun m => x m /\ y m.
  Definition orp  (x y : expr) : expr := fun m => x m \/ y m.
  Definition falsep : expr := fun m => False.
  Definition derivable (Phi : context) (x : expr) : Prop :=
    forall st, (forall x0, Phi x0 -> x0 st) -> x st.
End NaiveLang.
```

Then there is another module that provides the proofs of all the primary rules listed in `PrimitiveRulesSig`. The proof are commonly simple and straight forward, taking only a few lines to be done.

Alternatively, the user may choose to apply a deep embedding. A typical way of doing this would be defining the propositions (`expr`) inductively, with each of the primitive connectives as a constructor. An extra constructor `varp` is here to “lift” atom variables to propositions.

```
Inductive expr : Type :=
  | impp : expr -> expr -> expr
  | andp : expr -> expr -> expr
  | orp  : expr -> expr -> expr
  | falsep : expr
  | varp : var -> expr .
Definition context : Type := expr -> Prop.
```

The primitive judgement is also defined inductively, with all the primary rules as its constructors.

```
Inductive derivable : context -> expr -> Prop :=
  | deduction_falsep_elim : ...
  | deduction_orp_intros1 : ...
  | deduction_orp_intros2 : ...
  | deduction_orp_elim : ...
  | deduction_andp_intros : ...
  | deduction_andp_elim1 : ...
  | deduction_andp_elim2 : ...
  | deduction_modus_ponens : ...
  | deduction_impp_intros : ...
  | deduction_weaken : ...
  | derivable_assum : ...
  | deduction_subst : ... .
```

Then the constructors in the above inductive defintions are directly filled into the corresponding entries in the module types specified by interface in order to allow for further derivations of derived connectives, judgements, and inference rules.

```
Module NaiveLang.
  Definition expr := expr.
  Definition impp := impp.
  Definition andp := andp.
  Definition orp  := orp.
  Definition falsep := falsep.
  Definition context := context.
End NaiveLang.
```

4.3.2 Implementation of Logic Generator

A brief sketch on the implementation of logic generator is given here. First, there are the lists that document all supported connectives, judgements, rule classes, and derivations among them.

```
Definition connectives := [impp; andp; orp; ...].
Definition judgements := [provable; derivable; derivable1; ...].
Definition how_connectives := [FROM_andp_impp_TO_iffp; ...].
Definition how_judgements := [FROM_provable_TO_derivable; ...].
Definition rule_classes := [derivitive_OF_impp; ...].
```

There are also lists that indicate how these are defined, and how their corresponding type classes should be instantiated.

```
Definition connective_instances_build :=
  [ (minL, Build_MinimumLanguage L impp);
    (andpL, Build_AndLanguage L andp);
    (orpL, Build_OrLanguage L orp);
    ... ].
Definition judgement_instances_build :=
  [ (GammaP, Build_Provable L provable);
    (GammaD, Build_Derivable L derivable);
    (GammaD1, Build_Derivable1 L derivable1);
    ... ].
```

The dependency among these is recorded with a dependency graph. The dependency graph is not typed manually; instead, it is computed using Coq tactics designed to analyze the dependent types of Coq terms. After given the configuration as input, the logic generator computes all the connectives, judgements, and rules that are derivable from the primitives, using an algorithm similar to topological sort on the dependency graph. Then the result is printed in the interface file using the `idtac` tactic provided by Coq, which is able to print both given strings and Coq terms.

Chapter 5 Completeness of Propositional Logic

5.1 Syntax and Inference Rules

We adopt the common way of defining the syntax and semantics of the propositional logic system. The syntax of propositions is inductively defined as follows:

$$\varphi ::= x \mid \bot \mid \top \mid \neg\psi \mid \psi \wedge \chi \mid \psi \vee \chi \mid \psi \to \chi.$$

Here, φ, ψ, χ stand for propositions, and x stands for logical variable. The above says that a proposition can be a) a logical variable; b) a logical constant; c) negation of a proposition; d) conjunction, disjunction, or implication of two propositions.

The sequent calculus is employed to be the proof system of the propositional logic. The primitive judgement for the logic is "derivability" ($\Phi \vdash \varphi$), where Φ is a set of propositions, and φ is a proposition. Informally, it says that φ can be derived from a finite subset of propositions of Φ, based on the inference rules of the proof system. This proof system is built upon a set of inference rules. The universal quantification of propositions and contexts (sets of propositions) in the inference rules is omitted for brevity.

$$\frac{\varphi \in \Phi}{\Phi \vdash \varphi}(\textsc{Assu}) \quad \frac{\Phi \subseteq \Psi \quad \Phi \vdash \varphi}{\Psi \vdash \varphi}(\textsc{Weaken}) \quad \frac{\Phi \vdash \Psi \quad \Phi \cup \Psi \vdash \varphi}{\Phi \vdash \varphi}(\textsc{Subst})$$

Here $\Phi \vdash \Psi$ is the short-hand notation for "for all $\psi \in \Psi$, $\Phi \vdash \psi$". The first three inference rules are the structural rules of the sequent calculus. These rules state the obvious, trivial fact about the propositional logic.

Then there are the connective rules. For truth and falsehood, we only have introduction rule and elimination rule respectively.

$$\frac{}{\Phi \vdash \top}(\textsc{TrueIntro}) \quad \frac{\Phi \vdash \bot}{\Phi \vdash \varphi}(\textsc{FalseElim})$$

There are introduction rules and elimination rules for conjunction and disjunction.

$$\frac{\Phi \vdash \varphi \wedge \psi}{\Phi \vdash \varphi}(\textsc{AndElim1}) \quad \frac{\Phi \vdash \varphi \wedge \psi}{\Phi \vdash \psi}(\textsc{AndElim2})$$

$$\frac{\Phi \vdash \varphi \quad \Phi \vdash \psi}{\Phi \vdash \varphi \wedge \psi}(\textsc{AndIntro}) \quad \frac{\Phi; \varphi \vdash \chi \quad \Phi; \psi \vdash \chi}{\Phi; \varphi \vee \psi \vdash \chi}(\textsc{OrElim})$$

$$\frac{\Phi \vdash \varphi}{\Phi \vdash \varphi \vee \psi}(\textsc{OrIntro1}) \qquad \frac{\Phi \vdash \psi}{\Phi \vdash \varphi \vee \psi}(\textsc{OrIntro2})$$

For implication, we have the following inference rules. The ImpElim rule is sometimes called "Modus Ponens Rule".

$$\frac{\Phi \vdash \varphi \quad \Phi \vdash \varphi \to \psi}{\Phi \vdash \psi}(\textsc{ImpElim}) \qquad \frac{\Phi; \varphi \vdash \psi}{\Phi \vdash \varphi \to \psi}(\textsc{ImpIntro})$$

We choose to reason about the *classical* propositional logic. Therefore, the "Contradiction Rule" (Contra) is included in the proof system.

$$\frac{\Phi; \neg\varphi \vdash \psi \quad \Phi; \neg\varphi \vdash \neg\psi}{\Phi \vdash \varphi}(\textsc{Contra})$$

Intuitively, this rules says that if Φ and $\neg\varphi$ together would imply "contradictory" conclusions, then φ should be derivable from Φ.

Up to this point, all primary rules have been presented. There are some additional inference rules that are derivable from the primary rules, and they would be helpful in the succeeding proof of the completeness theorem. These rules relates negation to other connectives.

$$\frac{}{\Phi; \neg\neg\varphi \vdash \varphi}(\textsc{DoubleNot1}) \qquad \frac{}{\Phi; \varphi \vdash \neg\neg\varphi}(\textsc{DoubleNot2})$$

$$\frac{}{\Phi \vdash \neg\bot}(\textsc{NotFalse}) \qquad \frac{\Phi \vdash \neg\varphi \quad \Phi \vdash \neg\psi}{\Phi \vdash \neg(\varphi \vee \psi)}(\textsc{NotOr})$$

$$\frac{\Phi \vdash \neg\varphi}{\Phi \vdash \neg(\varphi \wedge \psi)}(\textsc{NotAnd1}) \qquad \frac{\Phi \vdash \neg\psi}{\Phi \vdash \neg(\varphi \wedge \psi)}(\textsc{NotAnd2})$$

$$\frac{\Phi \vdash \neg\varphi}{\Phi \vdash \varphi \to \psi}(\textsc{NotImp1}) \qquad \frac{\Phi \vdash \varphi \quad \Phi \vdash \neg\psi}{\Phi \vdash \neg(\varphi \to \psi)}(\textsc{NotImp2})$$

$$\frac{\Phi \vdash \psi \to \varphi \quad \Phi \vdash \neg\psi \to \varphi}{\Phi \vdash \varphi}(\textsc{CaseAna})$$

It will be shown below how to derive DoubleNot1 and NotAnd1 via a derivation tree. The other auxiliary rules can be derived in a similar manner.

$$\frac{\dfrac{\neg\varphi \in \Phi; \neg\neg\varphi; \neg\varphi}{\Phi; \neg\neg\varphi; \neg\varphi \vdash \neg\varphi} \quad \dfrac{\neg\neg\varphi \in \Phi; \neg\neg\varphi; \neg\varphi}{\Phi; \neg\neg\varphi; \neg\varphi \vdash \neg\neg\varphi}}{\Phi; \neg\neg\varphi \vdash \varphi}$$

$$\frac{\dfrac{\dfrac{\Phi; \neg\neg(\varphi \wedge \psi) \vdash \varphi \wedge \psi \quad \Phi \subseteq \Phi; \neg\neg(\varphi \wedge \psi)}{\Phi; \neg\neg(\varphi \wedge \psi) \vdash \Phi; \varphi \wedge \psi} \quad \Phi; \varphi \wedge \psi \vdash \varphi}{\Phi; \neg\neg(\varphi \wedge \psi) \vdash \varphi} \quad \dfrac{\Phi \subseteq \Phi; \neg\neg(\varphi \wedge \psi) \quad \Phi \vdash \neg\varphi}{\Phi; \neg\neg(\varphi \wedge \psi) \vdash \neg\varphi}}{\Phi \vdash \neg(\varphi \wedge \psi)}$$

To conclude the section, we give the definition of "provability" in terms of derivability.

Definition 5.1 (Provability) We say that a proposition φ is "provable", denoted by $\vdash \varphi$, if and only if it can be derived from an emptyset of propositions, i.e.,

$$\vdash \varphi \quad \text{iff.} \quad \emptyset \vdash \varphi.$$

5.2 Semantics

The definition of semantics of the propositional logic is straight forward. First, we give the definition of assignments.

Definition 5.2 (Assignments) Given a set V of variables, an assignment $J : V \to \{T, F\}$ maps each of the variables to a binary truth value.

Then we extend the definition of variable assignment J to the truth value of propositions.

Definition 5.3 The truth value $J^*(\varphi)$ of a proposition φ under assignment J is defined inductively as follows:

$$\begin{aligned}
J^*(x) = T &\quad \text{iff.} \quad J(x) = T \\
J^*(\bot) = T &\quad \text{iff.} \quad \text{never} \\
J^*(\top) = T &\quad \text{iff.} \quad \text{always} \\
J^*(\neg\varphi) = T &\quad \text{iff.} \quad J^*(\varphi) = F \\
J^*(\varphi \wedge \psi) = T &\quad \text{iff.} \quad J^*(\varphi) = T \text{ and } J^*(\psi) = T \\
J^*(\varphi \vee \psi) = T &\quad \text{iff.} \quad J^*(\varphi) = T \text{ or } J^*(\psi) = T \\
J^*(\varphi \to \psi) = T &\quad \text{iff.} \quad \text{if } J^*(\varphi) = T \text{ then } J^*(\psi) = T
\end{aligned}$$

With the definitions above, we are ready to give the definitions of the satisfaction relation and validity of a proposition.

Definition 5.4 (Satisfaction) Given an assignment J and a proposition φ, J *satisfies* φ, denoted by $J \vDash \varphi$ if and only if $J^*(\varphi) = T$.

Definition 5.5 (Validity) We say that a propotision φ is valid, denoted by $\vDash \varphi$, if and only if for any J, $J \vDash \varphi$.

5.3 The Completeness Theorem

The completeness theorem of propositional logic says that all valid propositions are provable. It is formally stated as follows.

Theorem 5.1 (Completeness) For any proposition φ, $\vDash \varphi$ implies $\vdash \varphi$.

Our approach to proving the completeness theorem adopts a constructive approach, whose idea is borrowed from the book written by Wasilewska[65]. The proof shall commence by a construction (§5.3.1), and then we state and prove the main lemma (§5.3.2). Finally, we will prove the completeness theorem based on the main lemma (§5.3.3).

5.3.1 Construction

Consider a proposition φ with n free variables $x_1, x_2, \ldots, x_n$. Given an assignment $v : \{x_1, \ldots, x_n\} \to \{T, F\}$, the construction is performed as follows. Define φ'_v by

$$\varphi'_v = \begin{cases} \varphi, & \text{if } v^*(\varphi) = T \\ \neg\varphi, & \text{if } v^*(\varphi) = F \end{cases}.$$

Besides an additional proposition is constructed corresponding to each of the variables and their truth values.

$$\psi_{i,v} = \begin{cases} x_i, & \text{if } v(x_i) = T \\ \neg x_i, & \text{if } v(x_i) = F \end{cases} \quad \text{for } i = 1, 2, \ldots, n.$$

The following example gives a clearer sense of how the construction is done. Let

$$\varphi = (x_1 \to \neg x_2) \land x_3,$$

and let v be an assignment such that

$$v(x_1) = F, \quad v(x_2) = T, \quad v(x_3) = F.$$

In this case, $v^*(\varphi) = F$. Therefore, the corresponding $\varphi'_v, \psi_{1,v}, \psi_{2,v}, \psi_{3,v}$ are

$$\psi'_v = \neg\varphi = \neg((x_1 \to \neg x_2) \land x_3),$$

$$\psi_{1,v} = \neg x_1, \quad \psi_{2,v} = x_2, \quad \psi_{3,v} = \neg x_3.$$

These constructions would play crucial role in the upcoming proof, becuase they establish a connection between the syntactic aspect of propositions to the semantic aspect, namely assignments and truth values.

5.3.2 Main Lemma

The main lemma specifies how the semantic notion of truth values is transformed to the syntactic notion of derivability.

Lemma 5.1 (Main Lemma) For any proposition φ and assignment v, if the propositions φ'_v and $\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v}$ are defined as above, then

$$\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_v.$$

Proof The proof is conducted through structural induction on the proposition φ. The first three cases are rather trivial, while the subsequent cases are of greater interest.

Case 1. $\varphi = x_i$ for some variable x_i. If $v(x_i) = T$, then we would also have $v^*(\varphi) = T$, and thus $\psi_{i,v} = x_i$, $\varphi'_v = x_i$. The conclusion can be easily observed. If $v(x_i) = F$, the proof is similar.

Case 2. $\varphi = \bot$. Then $v^*(\varphi) = F$, and $\psi'_v = \neg\varphi = \neg\bot$. The conclusion is derived with the inference rule NotFalse.

Case 3. $\varphi = \top$. Then $v^*(\varphi) = T$, and $\psi'_v = \top$. The corrresponding conclusion follows directly from the primary inference rule TrueIntro.

Case 4. $\varphi = \neg\varphi_1$. By induction hypothesis, we should have that

$$\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_{1,v},$$

where $\varphi'_{1,v}$ is defined by applying the same construction of φ'_v to φ_1. Two subcases are to be considered here.

Subcase 4.1. $v^*(\varphi_1) = T$. Thus, $v^*(\varphi) = F$, and $\varphi'_v = \neg\varphi = \neg\neg\varphi_1 = \neg\neg\varphi'_{1,v}$. By the induction hypothesis and the rule DoubleNot2, the conclusion can be derived by the following derivation tree.

$$\frac{\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_{1,v} \quad \psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v}, \varphi'_{1,v} \vdash \neg\neg\varphi'_{1,v}}{\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_v}.$$

Subcase 4.2. $v^*(\varphi_1) = F$. Thus $v^*(\varphi) = T$, and $\varphi'_v = \varphi = \neg\varphi_1 = \varphi'_{1,v}$. The conclusion is obvious.

Case 5. $\varphi = \varphi_1 \wedge \varphi_2$. The following two are the induction hypotheses.

$$\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_{1,v},$$

$$\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_{2,v}.$$

Since there are two sub-propositions here, we would have to consider four subcases.

Subcase 5.1. $v^*(\varphi_1) = T, v^*(\varphi_2) = T$. In this case, we have that

$$\varphi'_v = \varphi = \varphi_1 \wedge \varphi_2,$$

$$\varphi'_{1,v} = \varphi_1, \quad \varphi'_{2,v} = \varphi_2.$$

Thus, the conclusion follows from the induction hypotheses and the AndIntro rule.

Subcase 5.2. $v^*(\varphi_1) = T, v^*(\varphi_2) = F$. In this case,

$$\varphi'_v = \neg\varphi = \neg(\varphi_1 \wedge \varphi_2),$$

$$\varphi'_{1,v} = \varphi_1, \quad \varphi'_{2,v} = \neg\varphi_2.$$

By the induction hypotheses and NotAnd2, we could derive the conclusion as follows.

$$\frac{\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \neg\varphi_2}{\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_v}$$

Subcase 5.3. $v^*(\varphi_1) = F, v^*(\varphi_2) = T$. Similar to subcase 5.2.

Subcase 5.4. $v^*(\varphi_1) = F, v^*(\varphi_2) = F$. In this case,

$$\varphi'_v = \neg\varphi = \neg(\varphi_1 \wedge \varphi_2),$$

$$\varphi'_{1,v} = \neg\varphi_1, \quad \varphi'_{2,v} = \neg\varphi_2.$$

A derivation similar to that in subcase 5.2 would yield the desired conclusion.

Case 6. $\varphi = \varphi_1 \vee \varphi_2$. We have the same induction hypotheses as in case 5.

$$\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_{1,v},$$

$$\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_{2,v}.$$

Subcase 6.1. $v^*(\varphi_1) = T, v^*(\varphi_2) = T$. In this case,

$$\varphi'_v = \varphi = \varphi_1 \vee \varphi_2,$$

$$\varphi'_{1,v} = \varphi_1, \quad \varphi'_{2,v} = \varphi_2.$$

Thus, the conclusion follows from the induction hypotheses and OrIntro1.

Subcase 6.2. $v^*(\varphi_1) = T, v^*(\varphi_2) = F$. The conclusion can be derived using the induction hypotheses and OrIntro1.

Subcase 6.3. $v^*(\varphi_1) = F, v^*(\varphi_2) = T$. The conclusion can be derived using the induction hypotheses and OrIntro2.

Subcase 6.4. $v^*(\varphi_1) = F, v^*(\varphi_2) = F$. In this case,

$$\varphi'_v = \neg\varphi = \neg(\varphi_1 \vee \varphi_2),$$

$$\varphi'_{1,v} = \neg\varphi_1, \quad \varphi'_{2,v} = \neg\varphi_2.$$

The conclusion can be derived using the induction hypotheses and NotOr as follows.

$$\frac{\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \neg\varphi_1 \quad \psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \neg\varphi_2}{\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_v}$$

Case 7. $\varphi = \varphi_1 \rightarrow \varphi_2$. We have the same induction hypotheses as in case 5.

$$\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_{1,v},$$

$$\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_{2,v}.$$

Subcase 7.1. $v^*(\varphi_2) = T$. We does not care the truth value of $v^*(\varphi_1)$ here. By the assumption, we have

$$\varphi'_v = \varphi = \varphi_1 \rightarrow \varphi_2,$$

$$\varphi'_{2,v} = \varphi_2.$$

Thus, the conclusion can be derived using the induction hypotheses, Weaken and ImpIntro.

$$\frac{\dfrac{\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \subseteq \psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v}, \varphi_1 \quad \psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi_2}{\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v}, \varphi_1 \vdash \varphi_2}}{\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_v}$$

Subcase 7.2. $v^*(\varphi_1) = T, v^*(\varphi_2) = F$. In this case,

$$\varphi'_v = \neg\varphi = \neg(\varphi_1 \rightarrow \varphi_2),$$

$$\varphi'_{1,v} = \varphi_1, \quad \varphi'_{2,v} = \neg\varphi_2.$$

The conclusion can be derived using the induction hypotheses and NotImp2.

$$\frac{\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi_1 \quad \psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \neg\varphi_2}{\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_v}$$

Subcase 7.3. $v^*(\varphi_1) = F, v^*(\varphi_2) = F$. In this case,

$$\varphi'_v = \varphi = (\varphi_1 \to \varphi_2),$$

$$\varphi'_{1,v} = \neg\varphi_1, \quad \varphi'_{2,v} = \neg\varphi_2.$$

The conclusion can be derived using the induction hypotheses and NotImp1.

$$\frac{\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \neg\varphi_1}{\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi'_v}$$

Till now, all cases of the inductive proof have been examined. Hence, the proof can be concluded.

5.3.3 Proof of Completeness Theorem

Now we are ready to prove the completeness theorem (Theorem 5.1) with the main lemma (Lemma 5.1). As a reminder, the completeness theorem says that for any proposition φ,

$$\vDash \varphi \text{ implies } \vdash \varphi.$$

Proof Consider a arbitrary proposition φ satisfying $\vDash \varphi$. Let $x_1, x_2, \ldots, x_n$ be all logical variables in φ. Let V be the set of all assignments to the variables in φ, i.e.,

$$V = \{v \mid v : \{x_1, x_2, \ldots, x_n\} \to \{T, F\}\}.$$

Fix an arbitrary assignement $v \in V$. Since $\vDash \varphi$, we should have that $v \vDash \varphi$, and thus $v^*(\varphi) = T$. Hence, by the main lemma, the following derivability holds,

$$\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v} \vdash \varphi.$$

To prove $\vdash \varphi$, we need to eliminate all the hypotheses $(\psi_{1,v}, \psi_{2,v}, \ldots, \psi_{n,v})$. This is done via induction, namely eliminating the hypotheses one after another. As the base step is trivial, it suffices to prove the induction step, i.e., if

$$\psi_{1,v}, \ldots, \psi_{i+1,v} \vdash \varphi$$

then

$$\psi_{1,v}, \ldots, \psi_{i,v} \vdash \varphi.$$

Now we prove the induction step. Suppose we already have

$$\psi_{1,v}, \ldots, \psi_{i+1,v} \vdash \varphi.$$

We construct two assignements $v_1, v_2 : \{x_1, x_2, \ldots, x_n\} \longrightarrow \{T, F\}$ as follows

$$v_1(x_j) = \begin{cases} v(x_j), & \text{if } j \neq i+1 \\ T, & \text{if } j = i+1 \end{cases}, \quad v_2(x_j) = \begin{cases} v(x_j), & \text{if } j \neq i+1 \\ F, & \text{if } j = i+1 \end{cases}.$$

Thus, $\psi_{i+1,v_1} = x_{i+1}$ and $\psi_{i+1,v_2} = \neg x_{i+1}$. Since φ is valid, we have that $v_1^*(\varphi) = T$ and $v_2^*(\varphi) = T$. Besides, for all $j \neq i+1$, $\psi_{j,v_1} = \psi_{j,v_2} = \psi_j$. Therefore, by the main lemma,

$$\psi_{1,v}, \ldots, \psi_{i,v}, x_{i+1} \vdash \varphi,$$

$$\psi_{1,v}, \ldots, \psi_{i,v}, \neg x_{i+1} \vdash \varphi.$$

Applying the ImpIntro rule to the above formulae would yield that

$$\psi_{1,v}, \ldots, \psi_{i,v} \vdash x_{i+1} \longrightarrow \varphi,$$

$$\psi_{1,v}, \ldots, \psi_{i,v} \vdash \neg x_{i+1} \longrightarrow \varphi.$$

By the CaseAna rule, the conclusion can be derived as follows.

$$\frac{\psi_{1,v}, \ldots, \psi_{i,v} \vdash x_{i+1} \longrightarrow \varphi \quad \psi_{1,v}, \ldots, \psi_{i,v} \vdash \neg x_{i+1} \longrightarrow \varphi}{\psi_{1,v}, \ldots, \psi_{i,v} \vdash \varphi}$$

Therefore, the induction step has been proved, and the hypotheses in

$$\psi_{1,v} \ldots, \psi_{n,v} \vdash \varphi$$

can be eliminated one by one. The proof is concluded.

5.4 Formalization in Coq

Formalization would give us more confidence in the reliabity of the theory. Therefore, a formalization of the completeness of the proposition logic is carried out with the help of the LOGIC library and its logic generator.

5.4.1 Formalization of the Propositional Logic

We apply a deep embeddings of the propositional logic. The proposition is defined inductively with logical variables, logical constants, and the connectives as its constructors. `V.t` is the type of variables.

```
Inductive sprop: Type :=
  | SId (x: V.t)
  | SFalse
  | STrue
  | SNot (P: sprop)
  | SAnd (P Q: sprop)
  | SOr (P Q: sprop)
  | SImpl (P Q: sprop).
```

Then there are some auxiliary definitions which are used in further definitions and proofs.

```
Definition scontext : Type := sprop -> Prop.
Definition empty_scontext : scontext := fun (_ : sprop) => False.
Definition scontext_add (Phi : scontext) (x : sprop) :=
  (Union _ Phi (Singleton _ x)).
```

As mentioned before, the sequent calculus proof system is employed for our propositional logic. Thus, the primitive judgement should be $\Phi \vdash \varphi$, which is also defined inductively, with each of the primary rules as a constructor. The provability judgement is derived from the derivability judgement.

```
Inductive sderivable : scontext -> sprop -> Prop :=
  | SAssu : forall (Phi : scontext) P,
    Phi P -> sderivable Phi P
  | SWeaken : forall (Phi Phi' : scontext) P,
    (forall phi, Phi phi -> Phi' phi) -> sderivable Phi P ->
    sderivable Phi' P
  | SSubst : forall (Phi Psi : scontext) Q,
    (forall P, Psi P -> sderivable Phi P) ->
    sderivable (Union sprop Phi Psi) Q -> sderivable Phi Q
  ...
Definition sprovable (P : sprop) : Prop := s
  derivable empty_scontext P.
```

Then there are the semantic definitions. The satisfaction relation is defined recursively, by pattern matching on the abstract syntax tree of the proposition. Validity is defined as aforementioned.

```
Fixpoint ssat (sasgn : V.t -> bool) (P : sprop) : bool :=
  match P with
  | SId x => if (sasgn x) then true else false
  | SFalse => false
  | STrue => true
  | SNot P => negb (ssat sasgn P)
  | SAnd P Q => (ssat sasgn P) && (ssat sasgn Q)
  | SOr P Q => (ssat sasgn P) || (ssat sasgn Q)
  | SImpl P Q => implb (ssat sasgn P) (ssat sasgn Q)
  end.
Definition svalid (P : sprop) : Prop :=
  forall sasgn, ssat sasgn P = true.
```

The derivation of the derived rules is done leveraging the LOGIC library and its logic generator. The configuration indicates the connectives, judgements, and primary rule classes of the logic. All connectives and judgements in this propositional logic are primitive, and we only show the primary rules list in the configuration file.

```
Definition primitive_rule_classes :=
  [ dervitive_OF_basic_setting;
    derivitive_OF_falsep;
    derivitive_OF_truep;
    derivitive_OF_classical_logic;
    derivitive_OF_andp;
    derivitive_OF_orp;
    derivitive_OF_impp ].
```

The class `derivitive_OF_classical_logic` corresponds to the rules for the negation connective in a classical logic.

Then we have the interface generated by the generator. The derivation of all the desired derived rules (mentioned in §5.1) are included in the interface file. For the implementation of primitive definitions and proofs, we simply fill in the parameterized and axiomatized definitions in the module types with the constructors used in the inductive definition of `sprop` and sderivable.

```
Module SpropLanguage.
  Definition expr := sprop.
  Definition context := scontext.
  Definition derivable := sderivable.
  Definition falsep := SFalse.
  Definition truep := STrue.
  Definition negp := SNot.
  Definition andp := SAnd.
  Definition orp := SOr.
  Definition impp := SImpl.
End SpropLanguage.
Module PrimaryRules.
  Include DerivedNames(SpropLanguage).
  Definition deduction_weaken := SWeaken.
  Definition derivable_assum := SAssu.
  Definition deduction_subst := SSubst.
  ...
End PrimaryRules.
```

5.4.2 Formalization of the Construction

With the logic system and its semantics already formalized, and we are ready to embark on the formal proof of the completeness theorem. The first step is to formalize the constructions described in §5.3.1, i.e., φ'_v and $\psi_{i,v}$. We begin with the relatively straight forward definition of φ'_v.

```
Definition prime_construct (sasgn : V.t -> bool) (P : sprop) :=
  if (ssat sasgn P) then P else SNot P.
```

To define $\psi_{i,v}$, two auxiliary functions are needed: the first one that takes in a proposition, and produces a list of all variables in it; the second one converts a Coq list to an ensemble.

```
Fixpoint all_var_ (P : sprop) : list V.t :=
  match P with
  | SId x => [x]
  | SFalse | STrue => []
  | SNot Q => all_var_ Q
  | SAnd Q R | SOr Q R
  | SImpl Q R => (all_var_ Q) ++ (all_var_ R)
  end.
Definition all_var (P : sprop) : list V.t := rm_dup (all_var_ P).
Fixpoint list_to_ensemble {A : Type} (l : list A) : A -> Prop :=
  match l with
  | [] => fun _ => False
  | x :: l' => Union _ (Singleton _ x) (list_to_ensemble l')
  end.
```

The function `rm_dup` removes the duplicate elements in a list. Without duplicate elimination, problems would arise in the last step of the completeness proof where premises are eliminated inductively. Then we can define $\psi_{i,v}$'s as an ensemble of propositions.

```
Definition psi_construct_one (sasgn : V.t -> bool) (x : V.t) :=
  if (sasgn x) then SId x else SNot (SId x).
Definition psi_construct_list_ (sasgn : V.t -> bool) (P : sprop) :=
  let av := all_var P in
    @map (V.t) (sprop) (psi_construct_one sasgn) av.
Definition psi_construct_list (sasgn : V.t -> bool) (P : sprop) :=
  list_to_ensemble (psi_construct_list_ sasgn P).
```

With all the constructions prepared, the main lemma, which specifies the properties of the construction, can be formally stated with the following Coq code.

```
Lemma main_construct : forall (sasgn : V.t -> bool) (P : sprop),
  sderivable (psi_construct_list sasgn P) (prime_construct sasgn P).
```

The principal idea of the proof is the same as is in the mathematical proof, which is to prove by induction on the syntax tree of the proposition being considered, denoted by `P` in this case. Performing induction on `P` would yield seven subgoals, each corresponding to one construct of `sprop`. To tackle these subgoals, one Coq lemma is dedicated to each of them. As an example, the following lemma is associated with the `SAnd` case.

```
Lemma main_caseand : forall (sasgn : V.t -> bool) (P1 P2 : sprop),
  sderivable (psi_construct_list sasgn P1)
    (prime_construct sasgn P1) ->
  sderivable (psi_construct_list sasgn P2)
```

```
    (prime_construct sasgn P2) ->
  sderivable (psi_construct_list sasgn (SAnd P1 P2))
    (prime_construct sasgn (SAnd P1 P2)).
```

There are some technical issues in the proof of these subgoals, and the main lemma, but they are not detailed into here. After the seven subgoals are proved, directly applying them would accomplish the proof of the main lemma.

5.4.3 Formal Proof of Completeness

The formal proof of the completeness theorem is more tricky than the theoretical proof. To be specific, two major technical challenges are posed:

- When eliminating hypotheses on the left-hand side of “⊢”, duplicate propositions might cause problems.
- The hypotheses are formalized as an “ensemble” of propositions, which brings hardships when doing induction.

The first problem is addressed by eliminating duplicates when computing the list of variables (`all_var`) in the construction. The elimination of duplicates is done via the following function.

```
Fixpoint rm_dup (l: list V.t): list V.t :=
  match l with
  | nil => nil
  | x :: xs => if (in_dec V.eq_dec x xs)
        then (rm_dup xs) else (x :: rm_dup xs)
  end.
```

This is implemented via pattern matching on the list `l`. If `l` is an empty list, then an empty list is returned. Otherwise, `l` should consist of a head element `x` and a subsequent sublist `xs`. We do case analysis on this. If `x` is not in `xs`, then the return value should be a list consisting of `x` and `rm_dup xs`, which is the recursive computation of `xs` with its duplicates removed. If `x` is in `xs`, then only `rm_dup xs` is returned. Correspondingly, there is a function that determines whether duplicates exist in a given list.

```
Inductive no_dup {A : Type} : list A -> Prop :=
  | ND_nil : no_dup []
  | ND_cons : forall x l, no_dup l -> ~ In x l -> no_dup (x :: l).
```

Although this function is inductively defined, it is essentially akin to the definition of `rm_dup`. A property regarding `rm_dup` and `no_dup` can be proved easily.

```
Lemma all_var_no_dup : forall (l : list V.t), no_dup (rm_dup l).
```

Therefore, by removing duplicates in the list of variables beforehand, and adding a check for duplicates when proving the completeness theorem inductively, the first problem is solved.

To handle the second problem, we need to dig into the definition of $\psi_{i,v}$'s construction, and divide the proof of completeness theorem into multiple phases. Each of these phase is represented by a Coq lemma, as shown below.

```
Lemma complete_lemma_1 : forall (sasgn : V.t -> bool) (P : sprop),
  svalid P ->
  sderivable
    (list_to_ensemble (map (psi_construct_one sasgn) (all_var P))) P.
Lemma complete_lemma_2 : forall (l : list V.t) (P : sprop),
  no_dup l ->
  (forall (sasgn : V.t -> bool),
    sderivable
      (list_to_ensemble (map (psi_construct_one sasgn) l)) P) ->
  sderivable empty_scontext P.
Theorem sprop_complete : forall P, svalid P -> sprovable P.
```

The proof of `complete_lemma_2` is the critical step. It says that if a proposition `P` is derivable from the $\psi_{i,v}$ construction applied to any list of variables, then `P` is valid. The basic idea of proving this lemma is to apply induction on the list `l`, and for the inductive step with `l = x :: xs`, construct two assignments that disagree only at the variable `x` to eliminate the premise related to `x`. After the two lemmas are proved, directly applying them would produce a proof of the completeness theorem.

Chapter 6 Formalized Quantifier Logic

The logic with shallowly-embedded existential quantifier (∃) as a “connective” is formalized based on the LOGIC library. In §6.1, we discuss how this quantifier logic is formalized, including its syntax and relevent inference rules. In §6.2, an example is used to illustrate how the quantifier logic extension is integrated with the logic generator in the original version of the LOGIC library.

6.1 Syntax and Proof Rules

In the shallow embeddings of a quantifier logic, the existential quantifier (∃) can be considered as a “connective”. Therefore, there is a type classes designated to the formalization of this connective.

```
Class ShallowExistsLanguage (L : Language) : Type :=
  { exp {A : Type} : (A -> expr) -> expr }.
```

Notice that the type of the existentially quantified variable is taken as an implicit argument, since it can almost always be inferred from the type of the predicate following the quantifier. For example, the proposition

$$\exists (a : A), P(a)$$

can be represented as `exp P`, and Coq will automatically inferred the implicit argument `A` from the type of `P`.

We adopt $\varphi \vdash \psi$ as the primitive judgement for the quantifier logic. Following the approach taken in VST[52], the primary inference rules regarding the existential quantifier can be defined as follows.

$$\frac{\varphi \vdash P(x_0)}{\varphi \vdash \exists x, P(x)}(\textsc{ExRight}) \qquad \frac{\text{for any } x_0,\ P(x_0) \vdash \varphi}{\exists x, P(x) \vdash \varphi}(\textsc{ExLeft})$$

Here P stands for a predicate. These two rules are categorized into one Coq type class.

```
Class ShallowExistsDeduction
  (L : Language)
  (GammaD1 : Derivable1 L)
  {_ : ShallowExistsLanguage L} :=
  { shallow_exp_right :
      for any P φ x₀, if φ ⊢ P(x₀), then φ ⊢ ∃x, P(x) ;
    shallow_exp_left :
      for any P φ, if for any x₀, P(x₀) ⊢ φ, then ∃x, P(x) ⊢ φ  }.
```

The following two rules that take the existential quantifier "out of" a conjunction are commonly used in program verification.

$$\frac{}{(\exists x, P(x)) \land \varphi \vdash \exists x, (P(x) \land \varphi)}(\textsc{ExAnd}1) \quad \frac{}{\varphi \land (\exists x, P(x)) \vdash \exists x, (\varphi \land P(x))}(\textsc{ExAnd}2)$$

These can be proved with the primary rules of existential quantifier and the following adjoint rule of implication and disjunction.

$$\frac{\varphi \vdash \psi \to \chi}{\varphi \land \psi \vdash \chi}(\textsc{Adj}1) \quad \frac{\varphi \land \psi \vdash \chi}{\varphi \vdash \psi \to \chi}(\textsc{Adj}2)$$

Thus, we have Coq lemmas for the derivation of ExAnd1 and ExAnd2 in the formalization.

6.2 Integration to Logic Generator

All the formalizations concerning the existential quantifier are implemented as an extension to the LOGIC library, and are therefore integrated into the logic generator. A demo is used here to exemplify this integration.

Our demo is minimal for illustrating the quantifier logic, and includes three connectives - conjunction ($\land$), existential quantifier ($\exists$), and implication ($\to$). There is one primitive judgement - $\varphi \vdash \psi$, indicating the derivability of a proposition from a singleton proposition. The primary rules include the rules for basic setting of the proof system, the rules for conjunction, existential quantifier, and the rules indicating the adjoint property of conjunction and implication. Therefore, the configuration of the logic can be written as follows.

```
Definition how_connectives :=
  [ primitive_connective impp;
    primitive_connective andp;
    primitive_connective exp ].
Definition how_judgements :=
  [ primitive_judgement derivable1 ].
Definition primitive_rule_classes :=
  [ derivitive1_OF_andp;
```

```
    derivitive1_OF_impp_andp_adjoint;
    derivitive1_OF_exp;
    derivitive1_OF_basic_setting ].
```

If we input the above configuration into the logic generator, it would output an interface file. The module type `LanguageSig` for primitive definitions is as follows.

```
Module Type LanguageSig.
  Parameter Inline expr : Type .
  Parameter derivable1 : (expr -> expr -> Prop) .
  Parameter impp : (expr -> expr -> expr) .
  Parameter andp : (expr -> expr -> expr) .
  Parameter exp : (forall A : Type, (A -> expr) -> expr) .
End LanguageSig.
```

Most importantly, the module type for derived rules includes the following two desired derived inference rules.

```
Axiom ex_and1 : (forall (A : Type) (P : A -> expr) (Q : expr),
  derivable1 (andp (exp A P) Q) (exp A (fun x : A => andp (P x) Q))) .
Axiom ex_and2 : (forall (A : Type) (P : expr) (Q : A -> expr),
  derivable1 (andp P (exp A Q)) (exp A (fun x : A => andp P (Q x)))) .
```

These two axioms are instantiated in a later module with the proof we provide when formalizing the quantifier logic.

Chapter 7 Conclusion

In this thesis, we present a formalization of different logics in LOGIC, a formal proof of propositional logic completeness based on LOGIC, and a formalization of shallowly embedded quantifier logic as an extension of LOGIC. In short, the LOGIC library presents a way of formalizing different logics uniformly, and allows for flexible and versatile constructions all based on the same collection of Coq type classes. The logic generator aids the users to generate the demanded exportable logic easily, only requiring the specification of configuration and implementation of primitive definitions and proof. In all, The formalization of logics could act as a foundation of many higher level research projects, such as program verification, logic and formal methods education, etc.

There is more to be explored in the field of logic formalization. Firstly, more inference rules that are derivable can be added to the quantifier logic formalization - the rules regarding separating conjunction and existential quantifier can be formalized similarly to ExAnd1 and ExAnd2; the rules concerning the iterative version of conjunction and separation conjunction can also be derived from the set of primary rules. Besides, a deep embeddings of the quantifier logic can be formalized to support more flexible formalization choices, though maybe in a different way. A different approaches of the completeness proof may also be adopted. These may become future extensions of the work presented in this thesis.

Bibliography

[1] PAULSON L C. Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow): vol. 828[M]. Springer, 1994.

[2] NORELL U. Towards a practical programming language based on dependent type theory: vol. 32[M]. Chalmers University of Technology, 2007.

[3] BARRAS B, BOUTIN S, CORNES C, et al. The Coq proof assistant reference manual [J]. INRIA, version, 1999, 6(11).

[4] De MOURA L, KONG S, AVIGAD J, et al. The Lean theorem prover (system description)[C]//Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25. 2015: 378-388.

[5] TAO Y, CAO Q. LOGIC: A Coq Library for Logics[C]//Dependable Software Engineering. Theories, Tools, and Applications: 8th International Symposium, SETTA 2022, Beijing, China, October 27-29, 2022, Proceedings. 2022: 205-226.

[6] NEWELL A, SIMON H. The logic theory machine–A complex information processing system[J]. IRE Transactions on information theory, 1956, 2(3): 61-79.

[7] WHITEHEAD A N, RUSSELL B. Principia Mathematica[J]., 1910.

[8] NEWELL A, SHAW J C, SIMON H A. Report on a general problem solving program [C]//IFIP congress: vol. 256. 1959: 64.

[9] MILNER R. Logic for computable functions description of a machine implementation [R]. STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1972.

[10] SCOTT D S. A type-theoretical alternative to ISWIM, CUCH, OWHY[J]. Theoretical Computer Science, 1993, 121(1-2): 411-440.

[11] De BRUIJN N G. AUTOMATH, a language for mathematics[M]. Springer, 1983.

[12] BOYER R S, MOORE J S. Proving theorems about LISP functions[J]. Journal of the ACM (JACM), 1975, 22(1): 129-144.

[13] BOYER R S, MOORE J S. A Lemma Driven Automatic Theorem Prover for Recursive Function Theory.[R]. SRI INTERNATIONAL MENLO PARK CALIF, 1977.

[14] BOYER R S, MOORE J S. A computational logic[M]. Academic press, 2014.

[15] KLEIN G, ELPHINSTONE K, HEISER G, et al. seL4: Formal verification of an OS

kernel[C]//Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009: 207-220.

[16] GU R, SHAO Z, CHEN H, et al. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels.[C]//OSDI: vol. 16. 2016: 653-669.

[17] LEROY X, BLAZY S, KÄSTNER D, et al. CompCert-a formally verified optimizing compiler[C]//ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress. 2016.

[18] CAO Q, BERINGER L, GRUETTER S, et al. VST-Floyd: A separation logic tool to verify correctness of C programs[J]. Journal of Automated Reasoning, 2018, 61: 367-422.

[19] CHEN H, ZIEGLER D, CHAJED T, et al. Using Crash Hoare logic for certifying the FSCQ file system[C]//Proceedings of the 25th Symposium on Operating Systems Principles. 2015: 18-37.

[20] NIPKOW T, WENZEL M, PAULSON L C. Isabelle/HOL: a proof assistant for higher-order logic[M]. Springer, 2002.

[21] WEBER T. Towards mechanized program verification with separation logic[C]// Computer Science Logic: 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004. Proceedings 18. 2004: 250-264.

[22] O'KEEFE G. Towards a readable formalisation of category theory[J]. Electronic Notes in Theoretical Computer Science, 2004, 91: 212-228.

[23] AVIGAD J, DONNELLY K, GRAY D, et al. Number Theory[J]., 2004.

[24] PAULSON L C. The inductive approach to verifying cryptographic protocols[J]. Journal of computer security, 1998, 6(1-2): 85-128.

[25] CONSTABLE R, ALLEN S, BROMLEY H, et al. Implementing mathematics[M]. Prentice-Hall, 1986.

[26] KAUFMANN M, MANOLIOS P, MOORE J S. Computer-aided reasoning: ACL2 case studies: vol. 4[M]. Springer Science & Business Media, 2013.

[27] OWRE S, RUSHBY J M, SHANKAR N. PVS: A prototype verification system[C] //Automated Deduction—CADE-11: 11th International Conference on Automated Deduction Saratoga Springs, NY, USA, June 15–18, 1992 Proceedings 11. 1992: 748-

752.

[28] LUO Z. A unifying theory of dependent types: the schematic approach[C]//Logical Foundations of Computer Science—Tver'92: Second International Symposium Tver, Russia, July 20–24, 1992 Proceedings 2. 1992: 293-304.

[29] MARTIN-LÖF P, SAMBIN G. Intuitionistic type theory: vol. 9[M]. Bibliopolis Naples, 1984.

[30] Mathlib COMMUNITY T. The lean mathematical library[C]//Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. ACM, 2020.

[31] POWER J F, WEBSTER C. Working with linear logic in coq[J]., 1999.

[32] FORSTER Y, LARCHEY-WENDLING D. Certified undecidability of intuitionistic linear logic via binary stack machines and Minsky machines[C]//Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs. 2019: 104-117.

[33] FORSTER Y, KIRST D, WEHR D. Completeness theorems for first-order logic analysed in constructive type theory[C]//Logical Foundations of Computer Science: International Symposium, LFCS 2020, Deerfield Beach, FL, USA, January 4–7, 2020, Proceedings. 2020: 47-74.

[34] JENSEN J B. Techniques for model construction in separation logic[J]. Enabling Concise and Modular Specifications in Separation Logic, 2013: 117.

[35] ANDRADE GUZMÁN J M, HERNÁNDEZ QUIROZ F. Natural deduction and semantic models of justification logic in the proof assistant COQ[J]. Logic Journal of the IGPL, 2020, 28(6): 1077-1092.

[36] TEWS H. Formalizing cut elimination of coalgebraic logics in Coq[C]//Automated Reasoning with Analytic Tableaux and Related Methods: 22nd International Conference, TABLEAUX 2013, Nancy, France, September 16-19, 2013, Proceedings 22. 2013: 257-272.

[37] BENZMÜLLER C, WOLTZENLOGEL PALEO B. Interacting with modal logics in the coq proof assistant[C]//Computer Science–Theory and Applications: 10th International Computer Science Symposium in Russia, CSR 2015, Listvyanka, Russia, July 13-17, 2015, Proceedings 10. 2015: 398-411.

[38] De ALMEIDA BORGES A. Towards a Coq formalization of a quantified modal logic [J]. arXiv e-prints, 2022: arXiv-2206.

[39] HENDRIKS M, KALISZYK C, RAAMSDONK F V, et al. Teaching logic using a state-of-the-art proof assistant[J]., 2010.

[40] HENZ M, HOBOR A. Teaching experience: Logic and formal methods with Coq[C] //Certified Programs and Proofs: First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings 1. 2011: 199-215.

[41] PIERCE B C, CASINGHINO C, GABOARDI M, et al. Software foundations[J]. Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html, 2010.

[42] GONTHIER G. The four colour theorem: Engineering of a formal proof[C]// Computer Mathematics: 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers. 2008: 333-333.

[43] GONTHIER G, ASPERTI A, AVIGAD J, et al. A machine-checked proof of the odd order theorem[C]//Interactive Theorem Proving: 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings 4. 2013: 163-179.

[44] NIPKOW T. Hoare logics in Isabelle/HOL[J]. Proof and system-reliability, 2002: 341-367.

[45] BENZMÜLLER C, CLAUS M, SULTANA N. Systematic verification of the modal logic cube in Isabelle/HOL[J]. arXiv preprint arXiv:1507.08717, 2015.

[46] BLANCHETTE J C, POPESCU A, TRAYTEL D. Soundness and completeness proofs by coinductive methods[J]. Journal of Automated Reasoning, 2017, 58: 149-179.

[47] SCHLICHTKRULL A. Formalization of logic in the Isabelle proof assistant[D]. Technical University of Denmark Lyngby, Denmark, 2018.

[48] BOVE A, DYBJER P, SICARD-RAMÍREZ A. Embedding a logical theory of constructions in agda[C]//Proceedings of the 3rd workshop on Programming languages meets program verification. 2009: 59-66.

[49] KOKKE W. Formalising type-logical grammars in Agda[J]. arXiv preprint arXiv:1709.00728, 2017.

[50] POPE J. Formalizing constructive quantifier elimination in Agda[J]. arXiv preprint arXiv:1807.04083, 2018.

[51] APPEL A W. Verified Software Toolchain: (Invited Talk)[C]//Programming Languages and Systems: 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 20. 2011: 1-17.

[52] APPEL A W. Verifiable C[M]//. 2016.

[53] CHLIPALA A. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier[C]// Proceedings of the 18th ACM SIGPLAN international conference on Functional programming. 2013: 391-402.

[54] JACOBS B, SMANS J, PHILIPPAERTS P, et al. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java.[J]. NASA Formal Methods, 2011, 6617: 41-55.

[55] AHRENDT W, BECKERT B, BUBEL R, et al. Deductive software verification–the key book[J]. Lecture notes in computer science, 2016, 10001.

[56] CORRENSON L, CUOQ P, KIRCHNER F, et al. Frama-C User Manual[A/OL]. http://frama-c.com/download/frama-c-user-manual.pdf.

[57] LEINO K R M, NELSON G, SAXE J B. ESC/Java user's manual[J]. ESC, 2000, 2000: 002.

[58] POLIKARPOVA N, KURAJ I, SOLAR-LEZAMA A. Program synthesis from polymorphic refinement types[J]. ACM SIGPLAN Notices, 2016, 51(6): 522-538.

[59] ITZHAKY S, PELEG H, POLIKARPOVA N, et al. Cyclic program synthesis[C]// Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 2021: 944-959.

[60] AWODEY S. Category theory[M]. Oxford university press, 2010.

[61] MENDELSON E. Introduction to mathematical logic[M]. CRC press, 2009.

[62] EBBINGHAUS H D, FLUM J, THOMAS W, et al. Mathematical logic: vol. 1910[M]. Springer, 1994.

[63] SIECZKOWSKI F, BIZJAK A, BIRKEDAL L. ModuRes: A Coq library for modular reasoning about concurrent higher-order imperative programming languages[C]// Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings 6. 2015: 375-390.

[64] JUNG R, JOURDAN J H, KREBBERS R, et al. RustBelt: Securing the foundations of

the Rust programming language[J]. Proceedings of the ACM on Programming Languages, 2017, 2(POPL): 1-34.

[65] WASILEWSKA A. Logics for Computer Science: Classical and Non-Classical[M]. Springer, 2018.

Acknowledgements

I would like to express my deepest gratitude to Professor Qinxiang Cao. My interest in programming languages and formal methods started when I took Prof. Cao's course <Programming Languages> in 2021 Spring semester, and I have been doing formalization projects since then. With his knowledge and expertise, Prof. Cao provided me invaluable guidance, encouragement, and feedback during my undergraduate studies. This endeavor would not have been possible if without the support of Prof. Cao. I am also deeply grateful to Professor Yuting Wang for his patience and willingness to provide constructive feedback in the completion of this work. In addition, I am grateful to my parents for their unwavering, unconditional love, support, and encouragement throughout my academic journey. Finally, I would like to thank all my friends who have provided me with emotional support and encouragement.