LOGIC: A Coq Library for Logics

Yichen Tao and Qinxiang Cao(✉)

Shanghai Jiao Tong University, Shanghai, China
taoyc0904@sjtu.edu.cn, caoqinxiang@gmail.com

Abstract. LOGIC is a Coq library for formalizing logic studies, concerning both logics' applications and logics themselves (meta-theories). For applications, users can port derived rules and efficient proof automation tactics from LOGIC to their own program-logic-based verification projects. For meta-theories, users can easily formalize a standard soundness proof or a Henkin-style completeness proof for logics like classical/intuitionistic propositional logic, separation logic and modal logic with LOGIC's help. In this paper, we present how compositional and portable proof engineering is possible in LOGIC.

Keywords: Logic · Coq · Theorem proving

1 Introduction

Theorem provers like Coq [3] and Isabelle [19] have been used for formalizing sophisticated math proofs, including many important logic theorems [11,12]. Besides its own interests, formalized logics (especially program logics) have been widely used in program verification tools to guarantee big software systems' safety [7,9,15]. But for now, there was not yet a systematic, foundational and formalized library for general logic studies. In this paper, we present LOGIC[1] a Coq library for logic applications and logics' meta-theories. Specifically,

– we want to provide a foundational library so that more advanced results can be formalized on its basis, and proofs can be reused for similar conclusions;
– we want to export useful proof rules and proof automation tactics for proving assertion entailments in different program verification projects;
– we want to use one single "eco-system" to achieve both targets above.

One challenge of LOGIC is to formalize different logics and their meta-theories in a *uniform* way (for the purpose of maximum proof reuse). This seems straightforward since different logic studies share many technical notations and definitions. For example, "$\vdash \varphi$" usually means φ is provable and "$m \models \varphi$" describes a satisfaction relation between m and φ. Also, different logics and different semantics usually share some common parts. However, differences in subtle settings bring about many proof-engineering problems. For instance, different logic studies may choose different primitive connectives; thus it is even nontrivial to unify

[1] A link to the repository of LOGIC: https://github.com/QinxiangCao/LOGIC.

different semantic definitions, which are usually defined by recursion over syntax trees. Also, different logics may choose different primitive judgements. Hilbert systems define *provability* ($\vdash \varphi$) by axioms and primary proof rules and define

$$\Phi \vdash \varphi \triangleq \text{exists a finite set } \Psi \subseteq \Phi, \text{ s.t. } \vdash \left(\bigwedge_{\psi \in \Psi} \psi \right) \to \varphi. \tag{1}$$

But sequent calculi use $\Phi \vdash \varphi$ (φ is derivable from Φ) as their primitive judgements and let $\vdash \varphi$ represent $\emptyset \vdash \varphi$. How to formalize a theory for both of them is not obvious.

Formalizing a uniform Henkin-style completeness proof is especially difficult. Different proofs may construct their canonical models differently, e.g. some proofs choose "maximal consistent sets" to be the canonical model's possible worlds while other proofs choose "derivable-closed sets". Moreover, different semantics have different additional structures in their models. For example, modal logics' Kripke model contains a binary relation for defining box modality's semantics. Intuitionistic logics' Kripke model has a preorder for defining implication's semantics. Henkin-style completeness proofs are different in detail due to the differences in these basic constructions and premises.

Another challenge of LOGIC is to generate *efficient* proof automation tactics. For generality, proofs in LOGIC should be parameterized over different languages and different proof systems. However, this parameterized setting will usually cause significant overhead for proof construction.

In the rest of this paper, we will explain how LOGIC addresses these formalization challenges. We will first introduce some background information about the Coq theorem prover in Sect. 2. We then give an overview of the whole framework of LOGIC in Sect. 3. After that, we describe our formalization of connectives, judgements, and proof rules, along with the meta-logic properties - soundness and completeness - in Sect. 4. In Sect. 5, we demonstrate how the logic generator in LOGIC can help us with exportable logic libraries. In the end, we discuss related work in Sect. 6, and conclude in Sect. 7.

2 Background: Coq Proof Assistant

Coq is a theorem prover whose logical objects are written in a Calculus of Inductive Constructions [6]. This underlying formal language enables Coq's users to define higher order functions and state high order propositions, i.e. one can quantify over a function, a predicate, or even a higher order function/predicate.

Type classes are a special kind of higher order objects in Coq, usually used to formalize abstract structures, like group, partial order, etc. For example, a common definition of group $\mathbf{G}$ is a tuple $(X_{\mathbf{G}}, +_{\mathbf{G}}, -_{\mathbf{G}}, 0_{\mathbf{G}})$ where $X_{\mathbf{G}}$ is the underlying set, $+_{\mathbf{G}}$ is an associative binary function, $-_{\mathbf{G}}$ is an inverse function and $0_{\mathbf{G}}$ is a unit element. In order to describe associativity, one usually writes: $\forall\ x\ y\ z \in X_{\mathbf{G}}, (x +_{\mathbf{G}} y) +_{\mathbf{G}} z = x +_{\mathbf{G}} (y +_{\mathbf{G}} z)$. Defining groups as a type class allows Coq's users to omit that $\mathbf{G}$ for conciseness if it is not ambiguous.

Besides higher order objects, Coq provides a *module system* for users to easily structure large developments and a means of massive abstraction. Specifically, a module is a collection of definitions and proofs; a module signature is a set of theorem statements; and a functor is a derivation from assumptions (represented by a module signature) to conclusions (represented by another signature).

The higher order logic system and the module system are two critical Coq infrastructure for modular development. In comparison, derivations using type classes are formal Coq proofs for parameterized instances, e.g. a theorem $\hat{P}$ for groups usually has a form of: for any group $\mathbf{G}$, some property P holds for $\mathbf{G}$. Functors are not Coq proofs but proof generators, e.g. given a *concrete* tuple $\mathbf{G}$, a functor $\hat{P}$ for group theory will generate a proof of $P(\mathbf{G})$ from a proof that $\mathbf{G}$ is a group.

3 Overview

The major aim of LOGIC is to reuse definitions and proofs to automatically generate logic libraries based on users' demands, as well as providing flexible options on how the logic is constructed. For example, a logic language and its proof rules can be constructed in multiple ways. Specifically, if we want to formalize a propositional logic with the following connectives:

$$\rightarrow, \neg, \vee,$$

we can either follow Elliott Mendelson's approach [18], i.e. treating $\rightarrow$ and $\neg$ as primitive connectives and defining $p \vee q \triangleq \neg p \rightarrow q$, or adopt the method mentioned in Ebbinghaus *et al.*'s book [10], i.e. treating $\neg, \vee$ as primitive connectives and define $p \rightarrow q \triangleq \neg p \vee q$.

Besides, the proof system can be constructed in various ways. We often write $\vdash \varphi$ if φ is provable, and $\Phi \vdash \varphi$ if φ is derivable from a set of propositions Φ. When φ is derivable from a singleton proposition ψ, which is a rather common case, we write $\psi \vdash \varphi$. We wish to formalize these judgements by choosing any one of them as primitive, and deriving the others.

It seems a possible solution to construct with different modules and type classes including the parameterized definitions and proofs of different logics. There are indeed previous works done in this way, *e.g.*, VST-MSL [2], Iris [16,21], and Math Classes [22]. However, there are drawbacks of such an approach:

- The users need to be familiar with the entire framework to accomplish the construction. They would have to dig into our design details so that they can find the proper place-holder of each argument, and choose the classes that meet their requirements.
- Constructive proofs of proof rules would bring a large overhead. Suppose A, B and C are types, and we have a model defined as follows.

  ```
  Definition model : Type := A * B * C.
  ```

  We aim to build a separation logic on model, and need to instantiate the following classes:

```
Class Join (worlds : Type) := ...
Class Unit (worlds : Type) := ...
Class SepAlg (worlds : Type) {J : Join worlds} := ...
Class UnitJoin (worlds : Type) {U : Unit worlds}
    {J : Join worlds} {SA : SepAlg worlds} := ...
```

We can build instances on product types with the followings.

```
Instance prod_J (A B : Type) :
    Join A -> Join B -> Join (A * B) := ...
Instance prod_U (A B : Type) :
    Unit A -> Unit B -> Unit (A * B) := ...
Instance prod_SA (A B : Type) (J_A : Join A) (J_B : Join_B) :
    SepAlg A -> SepAlg B -> SepAlg (A * B) := ...
Instance prod_UJR (A B : Type) (J_A : Join A) (J_B : Join B)
    (U_A : Unit A) (U_B : Unit B)
    (SA_A : SepAlg A) (SA_B : SepAlg B) :
    UnitJoin A -> UnitJoin B -> UnitJoin (A * B) := ...
```

It is worth mentioning that there are implicit arguments in the instances. For example, the true Coq type of prod_SA A B J_A J_B is (@SepAlg A J_A) −> (@SepAlg B J_B) −> (@SepAlg (A ∗ B) (prod_J A B J_A J_B)).
The class UnitJoin on A ∗ B ∗ C can be instantiated as follows:

```
Definition UJR_ABC : UnitJoin (A * B * C) :=
    prod_UJR (A * B) C _ _ _ _ _ _
    (prod_UJR A B _ _ _ _ _ _ UJR_A UJR_B) UJR_C.
                  ⇑
```

The placeholder for the underscore pointed by the arrow should have type Join A, and we call it J_A. We can observe that the instance J_A may appear repeatedly during the process of the entire construction (*e.g.*, it may appear in the construction of SepAlg on A, A ∗ B, A ∗ B ∗ C, *etc.*.). This phenomenon causes the size of the construction to be unacceptably large. If more classes need to be instantiated, the memory taken could be exponentially large.

In order to address the problems above, an automatized logic generator becomes indispensable. We designed a logic generator that accepts a configuration designated by the user, integrates the required classes of items, and presents an interface for the construction of logic. The users are not required to know how the entire class system works. Instead, with the help of the interface, the users can use the system compositionally and derive the demanded definitions and proofs. The logic generator frees them from the tedious work of searching for the correct class and constructing the proof terms themselves. All they need to do, is writing the configuration, and implementing the primitive items.

Furthermore, we observe that in most applications of program verification, we use a single logic, whose syntax and proof rules remain unchanged throughout the process. Since the composition of type classes has already been constructed by the generator in the whole procedure of use, there is no need to reconstruct the logic, and thus spares time and memory, which addresses the overhead problem.

4 Parameterized Definitions and Proofs

As previously mentioned, we will first build a type class based system, and then establish an auxiliary system on top of it to automatically build instances for users. However, due to the versatility of logic application scenarios, traditional applications of type classes do not suffice to solve the problems. Thus, we divide the type classes into four layers, and make different design choices based on their applications: (a) languages; (b) connectives and judgements (Sect. 4.1); (c) proof rules (Sect. 4.2); (d) soundness (Sect. 4.3) and completeness (Sect. 4.4).

Notations in Coq and in this paper. We use "andp x y" to represent the conjunction of x and y in the logic's proposition, where "p" stands for "proposition". Additionally, Coq's infrastructure allows us to define a notation for the connectives, judgements. For example, we use "x && y" as an object logics' notation to represent "andp x y" in LOGIC, which is distinguished from Coq's notation for its own logic, the meta-logic. In this paper, we choose not to use these notations for conciseness. We will use standard logic notations like $\wedge$, $\vee$, *etc.*. for object languages and use informal English words "and", "or", *etc.*. in the meta language. For consideration of readability, we will adopt this convention (but within a box container) when we present Coq code.

4.1 Connectives and Judgements

As mentioned before, we wish to enable various constructions of the same logic language. We illustrate our design choice using the previously mentioned example - formalizing a logic with connectives $\rightarrow, \neg, \vee$. We can either select $\neg, \rightarrow$ as primitive connectives and adopt Mendelson's approach, or select $\neg, \vee$ as primitive connectives and follow Ebbinghaus *et al.*'s approach.

```
Class Mendelson_Language := {
    expr : Type;
    negp : expr -> expr;
    impp : expr -> expr -> expr; }.
Definition Mendelson_orp := fun p q => $\boxed{\neg p \rightarrow q}$.

Class Ebbinghaus_Language := {
    expr : Type;
    negp : expr -> expr;
    orp  : expr -> expr -> expr; }.
Definition Ebbinghaus_impp := fun p q => $\boxed{\neg p \vee q}$.
```

However, since we wish to support both constructions (or even more sophisticated ones) simultaneously, neither of the above methods works. Instead, we define languages and connectives respectively with different type classes (see below), i.e., one type class for the language and one type class for each of the connectives. The type class does not assume if the connective is primitive or derived. They just indicate the existence and type of the corresponding connective.

The languages are defined by the following Coq type class. It says that, once the set of *expressions* (expr) is defined, the language is defined.

```
Class Language := {expr : Type}.
```

Some of the type classes for the connectives are listed as follows.

```
Class OrLanguage (L : Language) := {orp : expr -> expr -> expr}.
Class AndLanguage (L : Language) := {andp : expr -> expr -> expr}.
Class ImpLanguage (L : Language) := {impp : expr -> expr -> expr}.
```

It is worth mentioning that "OrLanguage L" does not assume disjunction to be a primitive connective. It can be either a primitive connective or a derived one. In order to reason about the derivation among connectives, we introduce *refl classes* in LOGIC, *e.g.*

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

In LOGIC, we also have algebraic structures that do not rely on expressions. Instead, they work on the "model-level", directly showing the relationships between models. These are mostly used to derive higher connectives for expressions. For example, the algebraic structure join "$\oplus$" is commonly used to define the separating conjunction "$*$".

```
Class Join (worlds : Type) : Type :=
  join : worlds -> worlds -> worlds -> Prop.
Class SepconLanguage (L: Language): Type :=
  { sepcon : expr -> expr -> expr }.
Instance worlds_L := Build_Language (worlds -> Prop).
Class SepconDef_Join {SepconL : SepconLanguage worlds_L} :=
{ join2sepcon : $\boxed{\varphi * \psi}$ = fun w =>
  $\boxed{\text{exists } w_1\ w_2,\ \oplus(w_1, w_2, w) \text{ and } \varphi\ w_1 \text{ and } \psi\ w_2}$ }.
```

In that sense, separating conjunction can be treated as a derived definition from join, which means that we do not need to distinguish logic connectives from algebraic structures of models in LOGIC. We put such derivations among connectives and algebraic definitions in one single type class system.

LOGIC supports 7 propositional connectives and constants ($\wedge, \vee, \to, \leftrightarrow, \neg, \top$, and $\bot$), separation logic connectives, and modalities in modal logics, as well as algebraic structures for defining semantics of intuitionistic propositional logic, separation logic, and modal logic. For brevity, we only demonstrate a part of them here.

Besides connectives, LOGIC also encompasses type classes formalizing judgements, built in an analogous manner. For example, provable is a meta-logic property of propositional expressions; thus, its Coq type is: expr -> Prop. We also define logic equivalence, which is useful for different applications.

```
Class Provable (L: Language) :=
  { provable: expr -> Prop }. (* |-- x *)
Class Derivable (L: Language) :=
  { derivable: set_of_expr -> expr -> Prop }. (* X |--- x *)
Class Derivable1 (L:Language) :=
  { derivable1: expr -> expr -> Prop }. (* x |-- y *)
Class LogicEquiv (L:Language) :=
  { logic_equiv: expr -> expr -> Prop }. (* x --||-- y *)
```

Again, "Provable L" does not assume $\vdash \varphi$ to be a primitive definition. Logicians may choose either $\vdash \varphi$ or $\Phi \vdash \varphi$ as a primitive definition, and derive the other. Furthermore, computer scientists usually use $\varphi \vdash \psi$ as their primitive notation in formal program verification projects. LOGIC supports all these different choices and uses additional type classes to define transformation among them. The following is an example.

```
Class DerivableProvable
      (L: Language) (GammaP: Provable L)
      (GammaD: Derivable L) {_:ImpLanguage L} :=
{ derivable_provable:
```
$\boxed{\text{for any } \Phi\ \varphi,\ \Phi \vdash \varphi \text{ iff.}}$

$\boxed{\text{there exists } \varphi_1, \varphi_2, \ldots, \varphi_n \in \Phi, \text{ s.t. } \vdash \varphi_1 \to \varphi_2 \to \cdots \to \varphi_n \to \varphi}$ }.

4.2 Proof Rules

Like connectives and judgements, we also define type classes for proof rules, and portray the derivation between them using Coq lemmas. To illustrate how these type classes and Coq lemmas are designed, we take separation logic as an example. When constructing a separation logic, the following three rules regarding the separating conjunction "$*$" are often required:

- SepconComm: for any $\varphi\ \psi$, $\varphi * \psi \vdash \psi * \varphi$;
- SepconAssoc: for any $\varphi\ \psi\ \chi$, $\varphi * (\psi * \chi) \vdash (\varphi * \psi) * \chi$;
- SepconMono: for any $\varphi\ \psi\ \varphi'\ \psi$, if $\varphi \vdash \varphi'$ and $\psi \vdash \psi'$, then $\varphi * \psi \vdash \varphi' * \psi'$.

If separating implication "$-\!\!*$" is present, we may have the following proof rule:

- WandSepconAdjoint: for any $\varphi\ \psi\ \chi$, $\varphi * \psi \vdash \chi$ iff. $\varphi \vdash \psi -\!\!* \chi$.

It is known that SepconMono can be derived from SepconComm, SepconAssoc, and WandSepconAdjoint [8]. We then introduce how the type classes and Coq lemmas are designed with regard to the above example.

Primary Rule Classes for Internal Use. The followings are the two rule classes, SepconDeduction and WandDeduction, which serve the internal use of LOGIC. Notice that there is redundancy within these type classes, i.e. the third rule in SepconDeduction can be derived by the other three rules. However, it is still listed as a primary rule since some separation logic does not have separating implication “$-\!\!*$” in its language. Furthermore, as is the case in all primary rule classes for internal use, the dependency among them exhibits a clear hierarchy.

```
Class SepconDeduction (L : Language) (GammaD1 : Derivable1 L)
  {_ : SepconLanguage} :=
{ sepcon_comm  : $\boxed{\text{for any } \varphi\ \psi,\ \varphi * \psi \vdash \psi * \varphi}$;
  sepcon_assoc  : $\boxed{\text{for any } \varphi\ \psi\ \chi,\ \varphi * (\psi * \chi) \vdash (\varphi * \psi) * \chi}$;
  sepcon_mono  : $\boxed{\text{for any } \varphi\ \psi\ \varphi'\ \psi,\ \text{if } \varphi \vdash \varphi' \text{ and } \psi \vdash \psi', \text{ then } \varphi * \psi \vdash \varphi' * \psi'}$ }.
Class WandDeduction (L : Language) (GammaD1 : Derivable L)
  {_ : SepconLanguage L} {_ : WandLanguage L} :=
{ wand_sepcon_adjoint  : $\boxed{\text{for any } \varphi\ \psi\ \chi,\ \varphi * \psi \vdash \chi \text{ iff. } \varphi \vdash \psi -\!\!* \chi}$ }.
```

With the rule classes above, we can construct parameterized proofs of other rules. For example, the monotonicity of separating implication “$-\!\!*$” is derivable from the given context, including the basic properties of judgement BasicDeduction and WandDeduction shown above. This is formalized in the following lemma.

```
Lemma derivable1_wand_mono : forall {L : Language}
  {sepconL : SepconLanguage L} {wandL : WandLanguage L}
  {GammaD1 : Derivable L} {bD : BasicDeduction L GammaD1}
  {wandD : WandDeduction L GammaD1},
  $\boxed{\text{for any } \varphi_1\ \varphi_2\ \psi_1\ \psi_2,\ \text{if } \varphi_2 \vdash \varphi_1 \text{ and } \psi_2 \vdash \psi_1,\ \text{then } \varphi_1 -\!\!* \psi_1 \vdash \varphi_2 -\!\!* \psi_2}$
```

Rule Classes for Users' Construction. There are three type classes for users' construction concerning the above separation logic example, which are listed as follows. This allows users to be flexible in constructing the logics. If the desired logic does not involve separating implication “$-\!\!*$”, the user can select SepconDeduction_Weak and SepconDeduction_Mono as the primary rule classes. Otherwise, SepconDeduction_Weak and WandDeduction can be selected as primary rule classes for the logic, so that the rule SepconMono can be automatically derived. Among these type classes, there is no redundancy so that users can use whatever type class they demand without worrying about repetitive proofs.

```
Class SepconDeduction_Weak (L : Language) (GammaD1 : Derivable1 L)
  {_ : SepconLanguage} :=
{ __sepcon_comm  : $\boxed{\text{for any } \varphi\ \psi,\ \varphi * \psi \vdash \psi * \varphi}$;
  __sepcon_assoc  : $\boxed{\text{for any } \varphi\ \psi\ \chi,\ \varphi * (\psi * \chi) \vdash (\varphi * \psi) * \chi}$ }.
Class SepconDeduction_Mono (L : Language) (GammaD1 : Derivable1 L)
  {_ : SepconLanguage} :=
{ __sepcon_mono  : $\boxed{\text{for any } \varphi\ \psi\ \varphi'\ \psi,\ \text{if } \varphi \vdash \varphi' \text{ and } \psi \vdash \psi', \text{ then } \varphi * \psi \vdash \varphi' * \psi'}$ }.
Class WandDeduction (L : Language) (GammaD1 : Derivable L)
  {_ : SepconLanguage L} {_ : WandLanguage L} :=
{ wand_sepcon_adjoint  : $\boxed{\text{for any } \varphi\ \psi\ \chi,\ \varphi * \psi \vdash \chi \text{ iff. } \varphi \vdash \psi -\!\!* \chi}$ }.
```

Constructing Primary Rules from Inputs. In LOGIC, the derivation between rule classes is depicted using Coq lemmas, which allows constructing the primary rules with the input proofs of users. For example, the following lemma says that SepconMono is derivable once SepconComm, SepconAssoc (given in type class SpeconDeduction_Weak), and WandSepconMono (given in type class WandDeduction) have been proved.

```
Lemma WeakAdjoint2Mono :
  forall {L : Language} {GammaD1 : Derivable1 L}
  {_ : SepconLanguage L} {_ : WandLanguage L}
  {_ : SepconDeduction_Weak L GammaD1}
  {_ : WandDeduction L GammaD1},
  SepconDeduction_Mono L GammaD1.
```

Rules for Derived Connectives. Another use of Coq lemmas in LOGIC lies in the syntactic sugars of connectives and judgements. As is described in Sect. 4.1, LOGIC supports using syntactic sugars to define new connectives (judgements) from primitive ones. We show that we can use these derived connectives and judgements when proving inner theorems or derived rules as if they are primitive concepts. For example, OrFromDefToAx_Imp_Neg proves that disjunction will have its introduction rule and elimination rule if it is defined as $\varphi \vee \psi \triangleq \neg\varphi \rightarrow \psi$.

4.3 Semantics and Soundness

Double turnstile "$\cdot \vDash \cdot$" usually describes a satisfaction relation, while its left side may vary in accordance with the semantics. Thus, we use Coq type classes to parameterize over different possibilities.

```
Class Model := {model : Type}.
Class Semantics (L : Language) (MD : Model) :=
  {denotation : expr -> model -> Prop}.
```

That is: a semantics is defined as long as a denotation function maps every propositional expression to a subset of "models", the set of models where it is satisfied. Here, the "model" set, which may be different in different semantics, is defined by type class Model. Based on Model and Semantics, we can define semantics of different connectives. For example, "AndSemantics L MD SM" says SM is a semantics defining on language L and models MD; this language has at least one connective, conjunction; and $\varphi \wedge \psi$ is satisfied if and only if both φ and ψ is satisfied for any φ and ψ.

```
Class AndSemantics (L: Language) {_: AndLanguage L}
  (MD: Model) (SM: Semantics L MD) :=
  {denote_andp: $\boxed{\text{for any } m\ \varphi\ \psi,\ m \vDash \varphi \wedge \psi \text{ iff. } m \vDash \varphi \text{ and } m \vDash \psi}$ }.
```

Typically, a logic's soundness is proved by induction over proof trees and it suffices to prove that all primary proof rules preserve validity. We achieve proof reuse in LOGIC by formalizing these validity preservation lemmas, under parameterized assumptions over semantic definitions. For example, the validity preservation of ModusPonens is formalized in the following lemma.

```
Lemma sound_modus_ponens:
  forall {L: Language} {MD: Model} {kMD: KripkeModel MD}
         {M: Kmodel} {SM: Semantics L MD}
         {_:ImpLanguage L} {_:IL.Relation (Kworlds M)}
         {_:KripkeIntuitionisticSemantics L MD M SM}
         {_:KripkeImpSemantics L MD M SM},
 for any φ ψ, if φ → ψ and φ are valid on M, then ψ is valid on M .
```

4.4 Completeness

Completeness proofs are usually more complicated than simple inductions on proof trees. In LOGIC, we leverage a Henkin-style completeness proof, which consists of the following steps.[2]

– Proof by contradiction: assume $\Phi \not\vdash \varphi$.
– Lindenbaum construction: find a "good" set Ψ such that $\Psi \supseteq \Phi$ and $\Psi \not\vdash \varphi$.
– Canonical model construction: define a Kripke model $\mathcal{M}^c$ whose possible worlds are all "good" sets.
– Truth lemma: prove that for any Θ and θ, $\mathcal{M}^c, \Theta \vDash \theta$ if and only if $\theta \in \Theta$.
– Achieving contradiction: $\Phi \not\vDash \varphi$, since $\mathcal{M}^c, \Psi \vDash \Phi$ but $\mathcal{M}^c, \Psi \not\vDash \varphi$.

Parameterized Lindenbaum Construction. The target of Lindenbaum construction is to find a super set $\Psi \supseteq \Phi$ (given Φ) such that $\mathcal{F}(\Psi)$. Lindenbaum constructions always follow this routine:

$$\begin{array}{ll} \Phi_0 = \Phi, \Psi = \bigcup_n \Phi_n & \\ \Phi_{n+1} = \Phi_n \cup \varphi_n & \text{if } \Phi_n \cup \varphi_n \text{ has property } \mathcal{G} \\ \Phi_{n+1} = \Phi_n & \text{if } \Phi_n \cup \varphi_n \text{ does not have property } \mathcal{G} \\ \text{where } \{\varphi_n | n \in \mathbb{N}\} \text{ are all propositions.} & \end{array} \tag{2}$$

In LOGIC, we first formalize the Lindenbaum construction process in (2) as: $\Psi = \text{LC}(\Phi, \mathcal{G})$. Then all Lindenbaum construction lemmas share the same format: for any Φ, if $\mathcal{G}(\Phi)$, then $\mathcal{F}(\text{LC}(\Phi, \mathcal{G}))$. We call it $\text{LLS}(\mathcal{F}, \mathcal{G})$, where LLS stands for *Lindenbaum Lemma Statement*. It is worth clarifying that $\text{LLS}(\mathcal{F}, \mathcal{G})$'s definition only depends on languages but does not depend on proof theories or semantics. Only when instantiating $\mathcal{F}$ and $\mathcal{G}$ with concrete sets of propositions, a proof theory would be needed in their definitions. We prove in Coq that $\text{LLS}(\mathcal{F}, \mathcal{G})$ is compositional on $\mathcal{F}$ (Lindenbaum_by_conj). Additionally, we prove some general lemmas like Lindenbaum_self_by_finiteness and Lindenbaum_derivable_closed for $\mathcal{F}$'s common conjuncts.

[2] There has been previous work formalizing completeness of first-order logic [11]. We partially base our work on [8]. We significantly improve proof reuse and support more completeness proofs.

```
Lemma Lindenbaum_by_conj {L: Language}:
  for any ℱ₁, ℱ₂, and 𝒢, if LLS(ℱ₁, 𝒢) and LLS(ℱ₂, 𝒢), then LLS(ℱ₁ and ℱ₂, 𝒢) .

Lemma Lindenbaum_self_by_finiteness {L: Language}:
  for any 𝒢, if 𝒢 is finite-captured and subset-preserved, then LLS(𝒢, 𝒢) .

Lemma Lindenbaum_derivable_closed {L: Language} {Gamma: Derivable L}:
  for any 𝒢, if 𝒢 ∘ ∂ is subset-preserved and LLS(𝒢, 𝒢), then LLS(derivable-closed, 𝒢) .

Corollary Lindenbaum_cannot_derive {L: Language} {Gamma: Derivable L}:
  for any 𝒢 φ, if 𝒢(Φ) has form Φ ⊬ φ for any Φ, then LLS(derivable-closed, 𝒢) .
```

Here, a property $\mathcal{G}$ is finite-captured means: for any Φ, if Φ's every finite subset has property $\mathcal{G}$, then Φ itself has property $\mathcal{G}$; a property $\mathcal{G}$ is subset-preserved means: for any $\Phi \supseteq \Psi$, if $\mathcal{G}(\Phi)$ then $\mathcal{G}(\Psi)$; and ∂ is a function from proposition sets to proposition sets such that $\partial(\Phi) = \{\varphi \mid \Phi \vdash \varphi\}$. Based on Coq's higher order feature, we are able to define concepts like finite-captured and use them in the lemmas above. As a result, we do not need to duplicate proofs in Coq for different Lindenbaum constructions.

Parameterized Well-Definedness. In Henkin-style proofs, we need to prove that the canonical model is indeed a legal model. For example, in separation logic's completeness proof, we should check whether the join relation is commutative in the canonical model, i.e. $\text{join}^c(\Phi, \Psi, \Theta)$ if and only if $\text{join}^c(\Psi, \Phi, \Theta)$ for any "good" sets Φ, Ψ and Θ. Here, join^c is the join relation in the canonical model, which is usually defined as:

$$\text{join}^c(\Phi, \Psi, \Theta) \text{ iff. } \Phi * \Psi \subseteq \Theta.$$

According to the definition of $\Phi * \Psi$, its proof is very straight forward since $\vdash \varphi * \psi \leftrightarrow \psi * \varphi$. However, its formalization is nontrivial since different separation logics' completeness may choose different definitions of "good". For classical separation logics, a "good" set is a maximal consistent set. For intuitionistic separation logics without disjunction ($\vee$) or false ($\perp$), a "good" set is a derivable-closed set. For intuitionistic separation logics with disjunction ($\vee$) and false ($\perp$), a "good" set is a derivable-closed, disjunction-witnessed, consistent set. In LOGIC, we prove the following general statement of join^c's commutativity.

```
Lemma canonical_comm:
  forall {L: Language} {GammaD : Derivable L} {_:SepconLanguage L}
         {_:BasicSequentCalculus L GammaD}
         {_:SepconSequentCalculus L GammaD},
  if every "good" set is derivable-closed, then joinᶜ is commutative .
```

The main idea is: we do not prove a theorem for a specific definition of "good" sets; instead, we consider a general class of "good" sets. We use this method in many places in LOGIC for proving canonical model well-formed.

Parameterized Truth Lemma. Truth lemmas are usually proved by induction over propositions' syntax trees. For proof reuse, we formalize different induction steps separately for proof reuse as we do in soundness proofs (see Sect. 4.3). For example, the induction step of conjunction is to prove:

$$\begin{array}{ll} \text{If} & \text{for any } \Theta, \mathcal{M}^c, \Theta \vDash \varphi \text{ iff. } \varphi \in \Theta \\ \text{and} & \text{for any } \Theta, \mathcal{M}^c, \Theta \vDash \psi \text{ iff. } \psi \in \Theta, \\ \text{then} & \text{for any } \Theta, \mathcal{M}^c, \Theta \vDash \varphi \wedge \psi \text{ iff. } \varphi \wedge \psi \in \Theta. \end{array}$$

Of course, the conclusion above is true only under some specific assumptions about "good" sets and canonical model's structures. We prove these lemmas based on relaxed classes of "good" sets again as we do before.

In summary, we decompose Henkin-style completeness proofs into small steps so that these intermediate conclusions can be proved in a parameterized way. We heavily use Coq's higher order logic in these generalized proofs. Readers can check our Coq development and see how we can easily combine components together and achieve formalized completeness proofs for concrete logics.

5 Logic Generator

We have in LOGIC a logic generator, which uses the parameterized definitions and proofs to support generating and exporting the desired logic libraries, so as to untangle the problems mentioned in Sect. 3. In Sect. 5.1, we introduce how users can leverage the logic generator to generate an exportable library of logic according to their requirements. In Sect. 5.2, we explain how the logic generator is implemented to perform the desired features.

5.1 Features of Logic Generator

In order to build a logic system (including its connectives, judgements, and proof rules) based on LOGIC's generator, one needs to take the following three steps. First of all, a *configuration* file is set up by the user indicating their logic's primitive connectives, judgements, and primary proof rules, as well as how the other primitives and judgements are derived. For example, if we want to formalize Mendelson's propositional logic, the configuration should be written as follows.

```
Definition how_connectives :=
[ primitive_connective impp;
  primitive_connective negp;
  FROM_impp_negp_TO_orp].
Definition how_judgements :=
[ primitive_judgement provable;
  FROM_provable_TO_derivable1].
Definition primitive_rule_classes :=
[ provability_OF_impp;
  provability_OF_classical_logic_by_contra].
```

The configuration above specifies the followings about the desired logic.

- There are three connectives in the logic's language: implication "$\to$", negation "$\neg$", and disjunction "$\vee$", where the first two are primitive, and the third one is derived by $\varphi \vee \psi \triangleq \neg\varphi \to \psi$.
- There are two judgements in the logic's proof system: provable "$\vdash \cdot$" and derivable1 "$\cdot \vdash \cdot$", where the former is primitive, and the latter is derived by $\varphi \vdash \psi \triangleq \vdash \varphi \to \psi$.
- The primitive proof rules of the logic include the followings, where the first three are basic proof rules for implication "$\to$", and the fourth is the contradiction rule.
  - ModusPones: for any $\varphi\ \psi$, if $\vdash (\varphi \to \psi)$ and $\vdash \varphi$, then $\vdash \psi$.
  - Axiom1: for any $\varphi\ \psi$, $\vdash (\varphi \to (\psi \to \varphi))$.
  - Axiom2: for any $\varphi\ \psi\ \chi$, $\vdash ((\varphi \to \psi \to \chi) \to (\varphi \to \psi) \to (\varphi \to \chi))$.
  - ByContradiction: for any $\varphi\ \psi$, $\vdash (\neg\varphi \to \psi) \to (\neg\varphi \to \neg\psi) \to \varphi$.

Then the logic generator takes the configuration as input, and outputs an *interface* file, which includes Coq module types illustrating primitive connectives, judgements and rules that users need to provide, and Coq functors that derive derived connectives, derived judgements and derived proof rules. The primitive types, connectives and judgements are included in the module type LanguageSig, which only indicates the types, and is to be implemented by the user.

```
Module Type LanguageSig.
  Parameter Inline expr : Type .
  Parameter provable : (expr -> Prop) .
  Parameter impp : (expr -> expr -> expr) .
  Parameter negp : (expr -> expr) .
End LanguageSig.
(* Automatically generated *)
```

Analogously, the primary rules are included in another module type PrimitiveRuleSig, also to be proved by the user.

```
Module Type PrimitiveRuleSig (Names: LanguageSig).
Include DerivedNames (Names).
  Axiom by_contradiction : $\boxed{\text{for any } \varphi\ \psi,\ \vdash (\neg\varphi \to \psi) \to (\neg\varphi \to \neg\psi) \to \varphi}$.
  Axiom modus_ponens : $\boxed{\text{for any } \varphi\ \psi,\ \text{if } \vdash (\varphi \to \psi) \text{ and } \vdash \varphi,\ \text{then } \vdash \psi}$.
  Axiom axiom1 : $\boxed{\text{for any } \varphi\ \psi,\ \vdash (\varphi \to (\psi \to \varphi))}$.
  Axiom axiom2 : $\boxed{\text{for any } \varphi\ \psi\ \chi,\ \vdash ((\varphi \to \psi \to \chi) \to (\varphi \to \psi) \to (\varphi \to \chi))}$.
End PrimitiveRuleSig.
(* Automatically generated *)
```

All the proof rules that can be derived using the primary ones are included in the module LogicTheorems. To give a taste of what are the rules that can be derived, we list some of the rules included in LogicTheorems.

- Derivable1Refl: for any φ, $\varphi \vdash \varphi$;
- Derivable1Trans: for any $\varphi\ \psi\ \chi$, if $\varphi \vdash \psi$ and $\psi \vdash \chi$, then $\varphi \vdash \chi$;
- Impp2Orp1: for any $\varphi\ \psi, \vdash (\varphi \to \psi) \to (\neg\varphi \vee \psi)$;

– PeirceLaw: for any $\varphi\ \psi, \vdash ((\varphi \to \psi) \to \varphi) \to \varphi$.

The first two of the above are included because the judgement derivable1 is derived by provable according to LOGIC's internal type classes. The third follows from an analogous reason. The fourth is included because we have the primary rule class provability_OF_classical_logic_by_contra making the given logic a classical logic, and there are internal lemmas that ensure such derivation is valid.

Guided by the interface file, the users need to provide concrete definitions of primitive connectives and judgements, and proofs of primary rules. These are done in an *implementation* file. LOGIC supports implementation of both deep embeddings (defining propositions by syntax trees) and shallow embeddings (defining propositions as the set of worlds where it is satisfied, without using syntax trees). If shallow embedding is employed, the implementation of primitive connectives and judgements can be written as follows.

```
Module NaiveLang.
  Definition expr := worlds -> Prop.
  Definition impp (x y : expr) : expr := fun m => [if x m, then y m].
  Definition negp (x : expr) : expr := fun m => [not x m].
  Definition provable (x : expr) : Prop := [for any m, x m].
End NaiveLang.
```

Alternatively, if the user chooses to apply a deep embedding, the proposition (expr) should be defined as a syntax tree, as shown below. Here, an expr can be constructed in three different ways, corresponding to the two primitive connectives and the atom var, and provable can be derived in three different ways, corresponding to the four primitive rules.

```
Inductive expr: Type :=
  | impp : expr -> expr -> expr
  | negp : expr -> expr
  | varp : var -> expr.

Inductive provable: expr -> Prop :=
  | modus_ponens: [for any φ ψ, if ⊢ (φ → ψ) and ⊢ φ, then ⊢ ψ]
  | axiom1: [for any φ ψ, ⊢ (φ → (ψ → φ))]
  | axiom2: [for any φ ψ χ, ⊢ ((φ → ψ → χ) → (φ → ψ) → (φ → χ))].
  | by_contradiction: [for any φ ψ, ⊢ (¬φ → ψ) → (¬φ → ¬ψ) → φ].

Module NaiveLang.
  Definition expr := expr.
  Definition impp := impp.
  Definition negp := negp.
  Definition provable := provable.
End NaiveLang.
```

No matter what kind of embedding is used, the primary rules need to be proved.

```
Module NaiveRule.
  Include DerivedNames (NaiveLang).
  Lemma by_contradiction : ...  Proof. ... Qed.
  Lemma modus_ponens : ... Proof. ... Qed.
  Lemma axiom1 : ... Proof. ... Qed.
  Lemma axiom2 : ... Proof. ... Qed.
```

Once these are done, an exported library is ready. We have many examples of using the logic generator, which we list in the appendix.

5.2 Design of Logic Generator

Here is a brief sketch of our design: we put all connectives, judgements and primary proof rules that we support in LOGIC into a built-in list; we record dependencies among them with a dependency graph; and we compute all derivable connectives, judgements and rules from user's input (the configuration file) based on this graph. The dependencies we document in the dependency graph involve the followings:

– The dependency between connectives, judgements. For example, specifying FROM_impp_negp_TO_orp in the list how_connectives depends on the connectives injunction "$\rightarrow$" and negation "$\neg$".
– The dependency between rules. This is computed internally in the logic generator. For example, the derived rules of some connectives' properties depend on the users' input rules and the derivation of the connectives.

It is worth mentioning that the dependency graph is not typed into our source code manually, we develop a Coq tactic to analyze dependent types of Coq terms and use that tactic to generate the graph automatically. With the dependency lists and computations mentioned above, we are ready to generate and print the interface according to the configuration given by the users.

6 Related Work

We are not the first one to formalize logic studies in theorem provers. Blanchette *et al.* formalized FOL completeness in Isabelle/HOL using its codata type. Foster *et al.* formalized FOL completeness [11] and undecidability [12] in Coq. Tews [23] formalized cut elimination for propositional multi-modal logics in Coq. There are many other works that we do not have space to enumerate here. Comparing to these previous work, we do not yet support first order quantifiers but we are the first one who systematically support different choices of primitive connectives (not fixing the set of connectives), logic extension (not fixing the set of primary proof rules) and compositional proof formalization (especially for completeness).

If using shallow embeddings in formalization, logics are extensible since there is no limitation on which connectives can/cannot be involved. Jensen [14] formalized soundness theorems for a wide range of separation logic and their semantics. Benzmüller and Paleo [5] formalized shallowly embedded modal logics in Coq and formalized Gödel's ontological argument based on that. Henz and Hobor [13] taught propositional modal using a formalization in Coq. The famous formalized text book *Software Foundations* [20] uses shallow embeddings to formalize assertions and Hoare logics. However, these works limit themselves in using shallow embedding, thus completeness proofs cannot be formalized in their framework.

Many research groups have developed different program verification tools based on theorem provers. Benzmuller and Claus [4] formalized higher-order multi-modal logic in Isabelle using shallow embedding and provide a proof automation library in Isabelle. VST [1,7] enables users to prove C programs correct using a shallowly embedded impredicative higher-order concurrent separation logic with a semi-automatic tactic library. Bedrock [9] are designed for low level program verification. Iris proof mode (IPM) [17] provides a tactic library for building interactive separation logic proofs. None of these verification projects can be applied to different (but similar) logics like LOGIC if not causing any overhead. For example, IPM's users should provide instances for IPM's BI type class and affined-BI type class, which causes some overhead. VST uses a rich enough memory model for C so that it can use a fixed Coq type "environ −> mpred" for C programs' assertion language. However, some of its proof rules are proved sound using a general separation logic framework VST-MSL. Thus, this generalization-instantiation process causes overhead in some of its proof automation. In comparison, LOGIC supports parameterized reasoning for internal proofs and exports proof libraries with no efficiency overhead.

7 Conclusion

In this paper, we present LOGIC which formalizes logics' meta-theories and can be used to generate exportable logic libraries. For formalized meta-theories, LOGIC is the first to support different logic settings (like primitive connectives and primary proof rules) in one uniform system. It also provides support for compositionally building completeness proofs. For logic applications, LOGIC aims to provide multi-scenario support with proof automation tactics and related proof rules. For example, users would like to have a rich language, a powerful logic and efficient proof construction commands in real program verification. But for educational purpose, a teacher may prefer to use a simple logic to explain the key ideas involved. LOGIC can export libraries for both scenarios according to users' configuration.

Acknowledgement. This research is sponsored by National Natural Science foundation of China (NSFC) Grant No. 61902240.

A Sample Use Cases of Logic Generator

A.1 Demo1: Intuitionistic Propositional Logic

Primitive connectives: $\to, \wedge, \vee, \perp$.
Syntactic sugar for connectives: $\varphi \leftrightarrow \psi \triangleq (\varphi \to \psi) \wedge (\psi \to \varphi)$, $\neg\varphi \triangleq \varphi \to \perp$, $\top \triangleq \perp \to \perp$.

Primitive judgements: provable ($\vdash \varphi$).
Syntactic sugar for judgements: for any Φ φ, $\Phi \vdash \varphi$ iff. exists $\varphi_1, \varphi_2, \ldots, \varphi_n \in \Phi$, s.t. $\vdash \varphi_1 \to \varphi_2 \to \cdots \to \varphi_n \to \varphi$.
Primary rules:

- PeirceLaw: for any φ ψ, $\vdash ((\varphi \to \psi) \to \psi) \to \psi$;
- FalsepElim: for any φ, $\vdash \bot \to \varphi$;
- OrpIntros1: for any φ ψ, $\vdash \varphi \to (\varphi \vee \psi)$;
- OrpIntros2: for any φ ψ, $\vdash \psi \to (\varphi \vee \psi)$;
- OprElim: for any φ ψ χ, $\vdash (\varphi \to \chi) \to (\psi \to \chi) \to ((\varphi \vee \psi) \to \chi)$;
- AndpIntros: for any φ ψ, $\vdash \varphi \to \psi \to (\varphi \wedge \psi)$;
- AndpElim1: for any φ ψ, $\vdash (\varphi \wedge \psi) \to \varphi$;
- AndpElim2: for any φ ψ, $\vdash (\varphi \wedge \psi) \to \psi$;
- ModusPones: for any φ ψ, if $\vdash \varphi \to \psi$ and $\vdash \varphi$, then $\vdash \psi$;
- Axiom1: for any φ ψ, $\vdash \varphi \to (\psi \to \varphi)$;
- Axiom2: for any φ ψ χ, $\vdash (\varphi \to \psi \to \chi) \to (\varphi \to \psi) \to (\varphi \to \chi)$.

A.2 Demo2: A Very Small Logic

Primitive connectives: $\to$.
Primitive judgements: provable ($\vdash \varphi$).
Primary rules:

- ModusPones: for any φ ψ, if $\vdash \varphi \to \psi$ and $\vdash \varphi$, then $\vdash \psi$;
- Axiom1: for any φ ψ, $\vdash \varphi \to (\psi \to \varphi)$;
- Axiom2: for any φ ψ χ, $\vdash (\varphi \to \psi \to \chi) \to (\varphi \to \psi) \to (\varphi \to \chi)$.

A.3 Demo3: Intuitionistic Propositional Logic

Primitive connectives: $\to, \wedge, \vee, \bot$.
Syntactic sugar for connectives: $\varphi \leftrightarrow \psi \triangleq (\varphi \to \psi) \wedge (\psi \to \varphi)$, $\neg\varphi \triangleq \varphi \to \bot$, $\top \triangleq \bot \to \bot$, $\bigwedge_{i=1}^{n} \varphi_i \triangleq \varphi_1 \wedge \ldots \wedge \varphi_n$.
Primitive judgements: derivable $\Phi \vdash \varphi$.
Syntactic sugar for judgements: for any φ, $\vdash \varphi$ iff. $\emptyset \vdash \varphi$.
Primary rules:

- DedFalsepElim: for any Φ φ, if $\Phi \vdash \bot$, then $\Phi \vdash \varphi$;
- DedOrpIntros1: for any Φ φ ψ, if $\Phi \vdash \varphi$, then $\Phi \vdash \varphi \vee \psi$;
- DedOrpIntros1: for any Φ φ ψ, if $\Phi \vdash \psi$, then $\Phi \vdash \varphi \vee \psi$;
- DedOrpElim: for any Φ φ ψ χ, if $\Phi \cup \varphi \vdash \chi$ and $\Phi \cup \psi \vdash \chi$, then $\Phi \cup (\varphi \vee \psi) \vdash \chi$;
- DedAndpIntros: for any Φ φ ψ, if $\Phi \vdash \varphi$ and $\Phi \vdash \psi$, then $\Phi \vdash \varphi \wedge \psi$;
- DedAndpElim1: for any Φ φ ψ, if $\Phi \vdash \varphi \wedge \psi$, then $\Phi \vdash \varphi$;
- DedAndpElim2: for any Φ φ ψ, if $\Phi \vdash \varphi \wedge \psi$, then $\Phi \vdash \psi$;
- DedModusPonens: for any Φ φ ψ, if $\Phi \vdash \varphi$ and $\Phi \vdash \varphi \to \psi$, then $\Phi \vdash \psi$;
- DedImppIntros: for any Φ φ ψ, if $\Phi \cup \varphi \vdash \psi$, then $\Phi \vdash \varphi \to \psi$;
- DedWeaken: for any Φ Ψ φ, if Φ is included in Ψ and $\Phi \vdash \varphi$, then $\Psi \vdash \varphi$;
- DedAssum: for any Φ φ, if φ belongs to Φ, then $\Phi \vdash \varphi$;
- DedSubst: for any Φ Ψ ψ, if (for any φ, if φ belongs to Ψ, then $\Phi \vdash \varphi$) and $\Phi \cup \Psi \vdash \psi$, then $\Phi \vdash \psi$.

A.4 Demo4: Separation Logic, Without Separation Conjunction

Primitive connectives: $\wedge, \vee, \bot, \top, *, \mathbf{emp}$.
Syntactic sugar for connectives: $\bigwedge_{i=1}^{n} \varphi_i \triangleq \varphi_1 \wedge \ldots \wedge \varphi_n$, $*_{i=1}^{n}\varphi_i \triangleq \varphi_1 * \ldots * \varphi_n$.
Primitive judgements: derivable1 $\varphi \vdash \psi$.
Primary rules:

- FalsepSepconLeft: for any φ, $\bot * \varphi \vdash \bot$;
- OrpSepconLeft: for any $\varphi\ \psi\ \chi$, $(\varphi \vee \psi) * \chi \vdash (\varphi * \chi) \vee (\psi * \chi)$;
- SepconEmpLeft: for any φ, $\varphi * \mathbf{emp} \vdash \varphi$;
- SepconEmpRight: for any φ, $\varphi \vdash \varphi * \mathbf{emp}$;
- Der1SepconComm: for any $\varphi\ \psi$, $\varphi * \psi \vdash \psi * \varphi$;
- Der1SepconAssoc1: for any $\varphi\ \psi\ \chi$, $\varphi * (\psi * \chi) \vdash (\varphi * \psi) * \chi$;
- Der1SepconMono: for any $\varphi_1\ \varphi_2\ \psi_1\ \psi_2$, if $\varphi_1 \vdash \varphi_2$ and $\psi_1 \vdash \psi_2$, then $(\varphi_1 * \psi_1) \vdash (\varphi_2 * \psi_2)$;
- Der1TruepIntros: for any φ, $\varphi \vdash \top$;
- Der1FalsepElim: for any $\varphi, \bot \vdash \varphi$;
- Der1OrpIntros1: for any $\varphi\ \psi$, $\varphi \vdash \varphi \vee \psi$;
- Der1OrpIntros2: for any $\varphi\ \psi$, $\psi \vdash \varphi \vee \psi$;
- Der1OrpElim: for any $\varphi\ \psi\ \chi$, if $\varphi \vdash \chi$ and $\psi \vdash \chi$, then $\varphi \vee \psi \vdash \chi$;
- Der1AndpIntros: for any $\varphi\ \psi\ \chi$, if $\varphi \vdash \psi$ and $\varphi \vdash \chi$, then $\varphi \vdash \psi \wedge \chi$;
- Der1AndpElim1: for any $\varphi\ \psi$, $\varphi \wedge \psi \vdash \varphi$;
- Der1AndpElim2: for any $\varphi\ \psi$, $\varphi \wedge \psi \vdash \psi$.

A.5 Demo5: Separation Logic, with Separating Conjunction

Primary connectives: $\to, \wedge, \vee, \bot, *, -\!\!*, \mathbf{emp}$.
Syntactic sugar for connectives: $\varphi \leftrightarrow \psi \triangleq (\varphi \to \psi) \wedge (\psi \to \varphi)$, $\neg\varphi \triangleq \varphi \to \bot$, $\top \triangleq \bot \to \bot$, $\bigwedge_{i=1}^{n} \varphi_i \triangleq \varphi_1 \wedge \ldots \wedge \varphi_n$, $*_{i=1}^{n}\varphi_i \triangleq \varphi_1 * \ldots * \varphi_n$.
Primitive judgements: provable $(\vdash \varphi)$.
Syntactic sugar for judgements: for any $\Phi\ \varphi$, $\Phi \vdash \varphi$ iff. exists $\varphi_1, \varphi_2, \ldots, \varphi_n \in \Phi$, s.t. $\vdash \varphi_1 \to \varphi_2 \to \cdots \to \varphi_n \to \varphi$.
Primary rules:

- SepconEmp: for any φ, $\vdash (\varphi * \mathbf{emp}) \leftrightarrow \varphi$;
- SepconComm: for any $\varphi\ \psi$, $\vdash (\varphi * \psi) \leftrightarrow (\psi * \varphi)$;
- SepconAssoc: for any $\varphi\ \psi\ \chi$, $\vdash ((\varphi * \psi) * \chi) \leftrightarrow (\varphi * (\psi * \chi))$;
- WandSepconAdjoint: for any $\varphi\ \psi\ \chi$, $\vdash ((\varphi * \psi) \to \chi) \leftrightarrow (\varphi \to (\psi -\!\!* \chi))$;
- PeirceLaw: for any $\varphi\ \psi$, $\vdash ((\varphi \to \psi) \to \psi) \to \psi$;
- FalsepElim: for any φ, $\vdash \bot \to \varphi$;
- OrpIntros1: for any $\varphi\ \psi$, $\vdash \varphi \to (\varphi \vee \psi)$;
- OrpIntros2: for any $\varphi\ \psi$, $\vdash \psi \to (\varphi \vee \psi)$;
- OprElim: for any $\varphi\ \psi\ \chi$, $\vdash (\varphi \to \chi) \to (\psi \to \chi) \to ((\varphi \vee \psi) \to \chi)$;
- AndpIntros: for any $\varphi\ \psi$, $\vdash \varphi \to \psi \to (\varphi \wedge \psi)$;
- AndpElim1: for any $\varphi\ \psi$, $\vdash (\varphi \wedge \psi) \to \varphi$;
- AndpElim2: for any $\varphi\ \psi$, $\vdash (\varphi \wedge \psi) \to \psi$;
- ModusPones: for any $\varphi\ \psi$, if $\vdash \varphi \to \psi$ and $\vdash \varphi$, then $\vdash \psi$;
- Axiom1: for any $\varphi\ \psi$, $\vdash \varphi \to (\psi \to \varphi)$;
- Axiom2: for any $\varphi\ \psi\ \chi$, $\vdash (\varphi \to \psi \to \chi) \to (\varphi \to \psi) \to (\varphi \to \chi)$.

A.6 Demo6: Separation Logic, Without Separating Implication

Primitive connectives: $\rightarrow, \wedge, *, \mathbf{emp}$.
Syntactic sugar for connectives: $\varphi \leftrightarrow \psi \triangleq (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$.
Primitive judgement: provable ($\vdash \varphi$).
Syntactic sugar for judgements: for any Φ φ, $\Phi \vdash \varphi$ iff. exists $\varphi_1, \varphi_2, \ldots, \varphi_n \in \Phi$, s.t. $\vdash \varphi_1 \rightarrow \varphi_2 \rightarrow \cdots \rightarrow \varphi_n \rightarrow \varphi$;
for any ψ φ, $\psi \dashv\vdash \varphi$ iff. $\vdash \psi \rightarrow \varphi$ and $\vdash \varphi \rightarrow \psi$.
Primary rules:

- SepconEmp: for any φ, $\vdash (\varphi * \mathbf{emp}) \leftrightarrow \varphi$;
- SepconComm: for any φ ψ, $\vdash (\varphi * \psi) \leftrightarrow (\psi * \varphi)$;
- SepconAssoc: for any φ ψ χ, $\vdash ((\varphi * \psi) * \chi) \leftrightarrow (\varphi * (\psi * \chi))$;
- SepconMono: for any φ_1 φ_2 ψ_1 ψ_2, if $\vdash \varphi_1 \rightarrow \varphi_2$ and $\vdash \psi_1 \rightarrow \psi_2$, then $\vdash (\varphi_1 * \psi_1) \rightarrow (\varphi_2 * \psi_2)$;
- AndpIntros: for any φ ψ, $\vdash \varphi \rightarrow \psi \rightarrow (\varphi \wedge \psi)$;
- AndpElim1: for any φ ψ, $\vdash (\varphi \wedge \psi) \rightarrow \varphi$;
- AndpElim2: for any φ ψ, $\vdash (\varphi \wedge \psi) \rightarrow \psi$;
- ModusPones: for any φ ψ, if $\vdash \varphi \rightarrow \psi$ and $\vdash \varphi$, then $\vdash \psi$;
- Axiom1: for any φ ψ, $\vdash \varphi \rightarrow (\psi \rightarrow \varphi)$;
- Axiom2: for any φ ψ χ, $\vdash (\varphi \rightarrow \psi \rightarrow \chi) \rightarrow (\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi)$.

A.7 Demo7: Separation Logic, Constructed from Model Level

Primitive connectives: $\oplus, \mathbf{unit}$.
Syntactic sugar for connectives: define $\rightarrow, \wedge, \vee$ directly from model level, using Coq's meta-logic; $(\varphi * \psi)\ m \triangleq$ exists m_1 m_2, $\oplus(m_1, m_2, m)$ and φ m and ψ m; $\mathbf{emp}\ m \triangleq \mathbf{unit}\ m$.
Syntactic sugar for judgements: define provable ($\vdash \varphi$) and derivable1 ($\varphi \vdash \psi$) with Coq's meta-logic.
Primary rules:

- JoinComm: for any m_1 m_2 m, if $\oplus(m_1, m_2, m)$, then $\oplus(m_2, m_1, m)$;
- JoinAssoc: for any m_1 m_2 m_3 m_{12} m_{123}, if $\oplus(m_1, m_2, m_{12})$ and $\oplus(m_{12}, m_3,$
  $m_{123})$, then (there exists m_{23}, $\oplus(m_2, m_3, m_{23})$ and $\oplus(m_1, m_{23}, m_{123})$).

A.8 Mendelson's Propositional Logic

Primitive connectives: $\rightarrow, \neg, \top$.
Syntactic sugar for connectives: $\varphi \vee \psi \triangleq \neg\varphi \rightarrow \psi$, $\perp \triangleq \neg\top$.
Primitive judgements: provable ($\vdash \varphi$).
Syntactic sugar for judgements: for any Φ φ, $\Phi \vdash \varphi$ iff. exists $\varphi_1, \varphi_2, \ldots, \varphi_n \in \Phi$, s.t. $\vdash \varphi_1 \rightarrow \varphi_2 \rightarrow \cdots \rightarrow \varphi_n \rightarrow \varphi$.
Primary rules:

- ByContradiction: for any $\varphi\ \psi$, $\vdash (\neg\varphi \to \psi) \to (\neg\varphi \to \neg\psi) \to \varphi$.
- ModusPones: for any $\varphi\ \psi$, if $\vdash \varphi \to \psi$ and $\vdash \varphi$, then $\vdash \psi$;
- Axiom1: for any $\varphi\ \psi$, $\vdash \varphi \to (\psi \to \varphi)$;
- Axiom2: for any $\varphi\ \psi\ \chi$, $\vdash (\varphi \to \psi \to \chi) \to (\varphi \to \psi) \to (\varphi \to \chi)$.

We have proved in Coq the completeness of this logic.

A.9 Minimum Separation Logic

Primitive connectives: $\to, \wedge, *$.
Primitive judgements: provable ($\vdash \varphi$).
Syntactic sugar for judgements: for any $\Phi\ \varphi$, $\Phi \vdash \varphi$ iff. exists $\varphi_1, \varphi_2, \ldots, \varphi_n \in \Phi$, s.t. $\vdash \varphi_1 \to \varphi_2 \to \cdots \to \varphi_n \to \varphi$.
Primary rules:

- SepconCommImpp: for any $\varphi\ \psi$, $\vdash (\varphi * \psi) \to (\psi * \varphi)$;
- SepconAssoc1: for any $\varphi\ \psi\ \chi$, $\vdash (\varphi * (\psi * \chi)) \to ((\varphi * \psi) * \chi)$;
- SepconMono: for any $\varphi_1\ \varphi_2\ \psi_1\ \psi_2$, if $\vdash \varphi_1 \to \varphi_2$ and $\vdash \psi_1 \to \psi_2$, then $\vdash (\varphi_1 * \psi_1) \to (\varphi_2 * \psi_2)$;
- AndpIntros: for any $\varphi\ \psi$, $\vdash \varphi \to \psi \to (\varphi \wedge \psi)$;
- AndpElim1: for any $\varphi\ \psi$, $\vdash (\varphi \wedge \psi) \to \varphi$;
- AndpElim2: for any $\varphi\ \psi$, $\vdash (\varphi \wedge \psi) \to \psi$;
- ModusPones: for any $\varphi\ \psi$, if $\vdash \varphi \to \psi$ and $\vdash \varphi$, then $\vdash \psi$;
- Axiom1: for any $\varphi\ \psi$, $\vdash \varphi \to (\psi \to \varphi)$;
- Axiom2: for any $\varphi\ \psi\ \chi$, $\vdash (\varphi \to \psi \to \chi) \to (\varphi \to \psi) \to (\varphi \to \chi)$.

We have proved in Coq the completeness of this logic.

A.10 Demo for Bedrock2's Separation Logic

Primitive connectives: $\to, \wedge, *, \mathbf{emp}$.
Syntactic sugar for connectives: $\varphi \leftrightarrow \psi \triangleq (\varphi \to \psi) \leftrightarrow (\psi \to \varphi)$.
Primitive judgements: provable ($\vdash \varphi$).
Syntactic sugar for judgements: for any $\psi\ \varphi$, $\psi \vdash \varphi$ iff. $\vdash \psi \to \varphi$;
for any $\psi\ \varphi$, $\psi \dashv\vdash \varphi$ iff. $\vdash \psi \to \varphi$ and $\vdash \varphi \to \psi$.
Primary rules:

- SepconEmp: for any φ, $\vdash (\varphi * \mathbf{emp}) \leftrightarrow \varphi$;
- SepconComm: for any $\varphi\ \psi$, $\vdash (\varphi * \psi) \leftrightarrow (\psi * \varphi)$;
- SepconAssoc: for any $\varphi\ \psi\ \chi$, $\vdash ((\varphi * \psi) * \chi) \leftrightarrow (\varphi * (\psi * \chi))$;
- SepconMono: for any $\varphi_1\ \varphi_2\ \psi_1\ \psi_2$, if $\vdash \varphi_1 \to \varphi_2$ and $\vdash \psi_1 \to \psi_2$, then $\vdash (\varphi_1 * \psi_1) \to (\varphi_2 * \psi_2)$;
- AndpIntros: for any $\varphi\ \psi$, $\vdash \varphi \to \psi \to (\varphi \wedge \psi)$;
- AndpElim1: for any $\varphi\ \psi$, $\vdash (\varphi \wedge \psi) \to \varphi$;
- AndpElim2: for any $\varphi\ \psi$, $\vdash (\varphi \wedge \psi) \to \psi$;
- ModusPones: for any $\varphi\ \psi$, if $\vdash \varphi \to \psi$ and $\vdash \varphi$, then $\vdash \psi$;
- Axiom1: for any $\varphi\ \psi$, $\vdash \varphi \to (\psi \to \varphi)$;
- Axiom2: for any $\varphi\ \psi\ \chi$, $\vdash (\varphi \to \psi \to \chi) \to (\varphi \to \psi) \to (\varphi \to \chi)$.

References

1. Appel, A.W.: Verified software toolchain. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 1–17. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19718-5_1
2. Appel, A.W.: Verifiable C, chap. 5–17, 21, 35–39 (2016)
3. Barras, B., et al.: The coq Proof Assistant reference manual. Technical report, INRIA (1998)
4. Benzmüller, C., Claus, M., Sultana, N.: Systematic verification of the modal logic cube in Isabelle/Hol. In: Kaliszyk, C., Paskevich, A. (eds.) Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, 2–3 August 2015. EPTCS, vol. 186, pp. 27–41 (2015), https://doi.org/10.4204/EPTCS.186.5
5. Benzmüller, C., Woltzenlogel Paleo, B.: Interacting with modal logics in the coq proof assistant. In: Beklemishev, L.D., Musatov, D.V. (eds.) CSR 2015. LNCS, vol. 9139, pp. 398–411. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20297-6_25
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-07964-5
7. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: VST-FLOYD: a separation logic tool to verify correctness of C programs. J. Autom. Reason. **61**(1–4), 367–422 (2018). https://doi.org/10.1007/s10817-018-9457-5
8. Cao, Q., Cuellar, S., Appel, A.W.: Bringing order to the separation logic jungle. In: Chang, B.-Y.E. (ed.) APLAS 2017. LNCS, vol. 10695, pp. 190–211. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71237-6_10
9. Chlipala, A.: The bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier. In: Morrisett, G., Uustalu, T. (eds.) ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Boston, MA, USA - 25–27 September 2013, pp. 391–402. ACM (2013). https://doi.org/10.1145/2500365.2500592
10. Ebbinghaus, H., Flum, J., Thomas, W.: Mathematical Logic. Undergraduate Texts in Mathematics, vol. 291, 2nd edn. Springer, Cham (1994). https://doi.org/10.1007/978-3-030-73839-6
11. Forster, Y., Kirst, D., Wehr, D.: Completeness theorems for first-order logic analysed in constructive type theory. J. Log. Comput. **31**(1), 112–151 (2021). https://doi.org/10.1093/logcom/exaa073
12. Forster, Y., Larchey-Wendling, D.: Certified undecidability of intuitionistic linear logic via binary stack machines and Minsky machines. In: Mahboubi, A., Myreen, M.O. (eds.) Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, 14–15 January 2019, pp. 104–117. ACM (2019). https://doi.org/10.1145/3293880.3294096
13. Henz, M., Hobor, A.: Teaching experience: logic and formal methods with coq. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 199–215. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_16
14. Jensen, J.B.: Techniques for model construction in separation logic. Ph.D. thesis, IT University of Copenhagen, March 2014. https://public.knef.dk.s3-website-us-east-1.amazonaws.com/research/sltut.pdf

15. Jung, R., Jourdan, J., Krebbers, R., Dreyer, D.: RustBelt: securing the foundations of the rust programming language. Proc. ACM Program. Lang. **2**(POPL), 66:1–66:34 (2018). https://doi.org/10.1145/3158154
16. Jung, R., et al.: Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, 15–17 January 2015, pp. 637–650. ACM (2015). https://doi.org/10.1145/2676726.2676980
17. Krebbers, R., et al.: Mosel: a general, extensible modal framework for interactive proofs in separation logic. PACMPL. **2**(ICFP), 77:1–77:30 (2018). https://doi.org/10.1145/3236772
18. Mendelson, E.: Introduction to Mathematical Logic, 3rd edn. Chapman and Hall, London (1987)
19. Paulson, L.C. (ed.): Isabelle. LNCS, vol. 828. Springer, Heidelberg (1994). https://doi.org/10.1007/BFb0030541
20. Pierce, B.C., et al.: Software foundations. Webpage: https://wwwcis.upenn.edu/bcpierce/sf/current/index.html (2010)
21. Sieczkowski, F., Bizjak, A., Birkedal, L.: ModuRes: a COQ library for modular reasoning about concurrent higher-order imperative programming languages. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 375–390. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_25
22. Sozeau, M., Oury, N.: First-class type classes. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 278–293. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_23
23. Tews, H.: Formalizing cut elimination of coalgebraic logics in COQ. In: Galmiche, D., Larchey-Wendling, D. (eds.) TABLEAUX 2013. LNCS (LNAI), vol. 8123, pp. 257–272. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40537-2_22