Proving the consistency of Logic in Lean

[1]**Luiz Carlos R. Viana**

[1]Departamento de Informática – Pontifícia Universidade Católica
do Rio de Janeiro (PUC-RJ) – Rio de Janeiro, RJ, Brazil

[1]luizcarlosrvn424@gmail.com

Abstract. *We implement classical first-order logic with equality in the Lean Interactive Theorem Prover (ITP), and prove its soundness relative to the usual semantics. As a corollary, we prove the consistency of the calculus.*

1. Introduction

The advent of intuitionistic dependent type theories allowed the development of proof assistants for the formalization and computer-aided verification of mathematical proofs. These interactive proof systems allow the experienced user to prove just about any theorem of mathematics. As explained in Martin-Löf's paper [Martin-Löf 1982], by applying the Curry-Howard isomorphism between proofs in intuitionistic logic and terms in a typed lambda calculus, we can see any flavour of intuitionistic type theory as a programming language whose programs are mathematical proofs. The Lean ITP, developed by Leonardo de Moura at Microsoft Research, is such a programming language: implementing intuitionistic type theory and the calculus of inductive constructions, Lean supports the engineering of complex proofs in all areas of mathematics. By default, Lean allows the construction of proofs in intuitionistic logic, but by the declaration of the axiom of choice as an extra assumption of the theory, it also allows classical reasoning to take place. Since intuitionistic type theory with the axiom of choice is at least as expressive as classical ZFC set theory, it is very much well suited for use as a meta-language for the development of semantics for logical calculi. In our research we implemented first-order logic as an internal embedded language of Lean and used Lean structures to give model-theoretic semantics to the logical calculus. This allowed us to derive meta-theorems about the calculus, such as its soundness, and opens up the way for future investigations.

2. Motivation of our Work

In the spirit of literate programming, we set out to investigate the usability of these provers for teaching logic to students. This investigation is justified to the extent that these tools can provide a language in which to formalize the subject and its proofs, a practical tool for the solution of mathematical exercises, and finally, a means of teaching the formal verification of computer programs and their semantics. Already some courses of logic using Lean exist, such as the online book *Logic and Proof* [Jeremy Avigad and van Doorn 2017], but we know of no logic course which teaches the soundness theorem by means of a deep embedding of the syntax of first-order logic within Lean. Our implementation gives a basis for the creation of specialized exercise lists involving meta-theorems of first-order logic. The code

can be reused as the basis of a book on meta-logic, or for the development of a general-purpose library for dealing with different logical formalisms as embedded languages of the prover; in particular Hoare logic, which is relevant for applications of Lean in software verification. It could also be used to prove the consistency of particular first-order theories via the soundness meta-theorem, by constructing models for them.

3. Implementation

We implemented the different aspects of first order logic, dividing definitions and proofs into syntactical and semantical sections. In comparison with the Flypitch project [Han and van Doorn 2019], which also implements a deep embedding of logic using dependent types within Lean, our approach was designed to be easier to read and much shorter, as it is only meant as a simple soundness proof; for this reason we also do not use De Brujin indices for variable binding. In comparison with implementations in other languages such as Isabelle/HOL, as can be seen in [From et al. 2020], our approach benefits from the usage of dependent types to provide greater generality in language definition. We also used dependent types for inductively defining a type of proofs, rather than using the approach in [From et al. 2020] of recursively defining whether a list of formulas is a proof or not. In our opinion this provides a cleaner implementation of natural deduction when dependent types are available.

We present a short summary of our implementation below. Due to limitations of space, we are not able to discuss the proofs in greater detail, but the Github repository containing the whole code can be accessed from [Viana 2020].

3.1. Syntactic Definitions

The most interesting aspect of our implementation is the usage of dependent types to provide a signature abstraction which can be reused for defining a variety of different formal languages:

```
-- The type of signatures, which defines a first-order language,
-- possibly with extra modalities, and with room for defining your
-- own preferred variable type.
@[derive inhabited]
structure signature : Type (u+1) :=
    (functional_symbol : Type u := pempty)
    (relational_symbol : Type u := pempty)
    (modality : Type u := pempty)
    (vars : Type u := ulift ℕ)
    (dec_vars : decidable_eq vars . apply_instance)
    (arity : functional_symbol → ℕ := λ_, 0)
    (rarity : relational_symbol → ℕ := λ_, 0)
    (marity : modality → ℕ := λ_, 0)
```

The type of first-order formulas will later be defined in terms of a signature instance as a dependent type. Depending on the instance, some propositional-like calculus would end up being defined; this allows us to keep all the information required to define at least the most usual kinds of logical languages packed into the

same signature type. It is also of course possible to define multiple dependent formula types for the same signature instance, so we could define (e.g.) lambda-terms for untyped lambda calculus with primitive lambda terms given by the functional symbols. Modalities are currently not used in our implementations of logical languages, but they are added to the signature to support future extensions in this direction.

We can see also that all of the fields are optional and come with sensible defaults. Variables are by default natural numbers, but any type can be chosen provided that equality between instances of that type is decidable, as this is required in many of our proofs. If no field is provided, our later definition of a first-order formula will correspond to a simple calculus in which atomic propositions consist of equality comparisons between variables. Instead if we set the type of variables to the empty type *pempty* and the relational symbols to *ulift* $\mathbb{N}$ (i.e. the type of natural numbers lifted to universe u) we get essentially classical propositional logic. Since our soundness proof is agnostic about the signature, it amounts to a proof of the soundness of all of these different logics. The basic definitions of terms and formulas are given below:

```
variable {σ : signature}
-- arity types

-- the type of functional symbols of arity n
def signature.nary (σ : signature) (n : ℕ) := subtype {f : σ.functional_symbol |
 ↪  σ.arity f = n}

-- the type of relational symbols of arity n
def signature.nrary (σ : signature) (n : ℕ) := subtype {r : σ.relational_symbol |
 ↪  σ.rarity r = n}

-- the predicate defining, and the type of, constants
def is_constant (f : σ.functional_symbol) := σ.arity f = 0
def signature.const (σ : signature) := σ.nary 0

-- terms in the language
inductive signature.term (σ : signature)
| var : σ.vars → signature.term
| app  {n : ℕ} (f : σ.nary n) (v : fin n → signature.term) :  signature.term

-- constant terms. A function that lifts a constant to a term.
def signature.cterm (σ : signature) : σ.const → σ.term
| c := term.app c fin_zero_elim

--  the type of formulas in the language
inductive  signature.formula (σ : signature)
| relational {n : ℕ} (r : σ.nrary n) (v : fin n → σ.term) :  signature.formula
| for_all :  σ.vars →  signature.formula →  signature.formula
| if_then :  signature.formula →  signature.formula →  signature.formula
```

```
| equation (t₁ t₂ : σ.term) :  signature.formula
| false :  signature.formula
```

As can be seen, we use the $\{\forall, \to, \bot\}$ functionally complete fragment of first-order logic in the definition of formulas. The other connectives were later defined in the usual way, the definition of which we omit for brevity. We further set up the symbol ⇒ as specific notation for the `if_then` constructor, as Lean allows us to specify our own notational conventions into the system. Our choice of variable binding is made due to simplicity and intuitiveness of implementation and readability of the source code. The quantifier simply binds a variable to a formula, and even though by default variables are natural numbers, they are not used as De Brujin indices.

A nice convenience that we get out of our approach can already be seen. Lean allows functions defined as `type_name.name` to be later accessed with the *dot notation* with an instance of that type. So if `σ` is a signature `σ.term` will be its dependent type of terms and `σ.formula` the type of formulas in that signature. Moving on, the proof system was implemented inductively using natural deduction rules according to the following definition:

```
-- deductive consequence of formulas: Γ ⊢ φ.
-- Type of proofs from Γ to φ.
inductive proof : set  σ.formula → σ.formula → Type u_1
| reflexivity (Γ : set  σ.formula) (φ :  σ.formula)(h : φ ∈ Γ) : proof Γ φ
| transitivity (Γ Δ : set  σ.formula) (φ :  σ.formula)
              (h₁ : ∀ ψ ∈ Δ, proof Γ ψ)
              (h₂ : proof Δ φ) :  proof Γ φ
| modus_ponens
            (φ ψ :  σ.formula) (Γ : set  σ.formula)
            (h₁ : proof Γ (φ ⇒ ψ))
            (h₂ : proof Γ φ)
             : proof Γ ψ
| intro
            (φ ψ :  σ.formula) (Γ : set  σ.formula)
            (h : proof (Γ ∪ {φ}) ψ)
             : proof Γ (φ ⇒ ψ)
| for_all_intro
            (Γ : set  σ.formula) (φ :  σ.formula)
            (x : σ.vars) (xf : x ∈ φ.free)
            (abs : abstract_in x Γ)
            (h : proof Γ φ)
             : proof Γ ( signature.formula.for_all x φ)
| for_all_elim
            (Γ : set  σ.formula) (φ :  σ.formula)
            (x : σ.vars)
            (t : σ.term) (sub : φ.substitutable x t)
            (h : proof Γ ( signature.formula.for_all x φ))
             : proof Γ (φ.rw x t)
```

```
| exfalso (Γ : set  σ.formula) (φ :  σ.formula)
          (h : proof Γ  signature.formula.false)
          : proof Γ φ
| by_contradiction (Γ : set  σ.formula) (φ :  σ.formula)
                   (h : proof Γ φ.not.not)
                   : proof Γ φ
| identity_intro
            (Γ : set  σ.formula) (t : σ.term)
             : proof Γ ( signature.formula.equation t t)
| identity_elim
            (Γ : set  σ.formula) (φ :  σ.formula)
            (x : σ.vars) (xf : x ∈ φ.free)
            (t₁ t₂: σ.term)
            (sub₁ : φ.substitutable x t₁)
            (sub₂ : φ.substitutable x t₂)
            (h : proof Γ (φ.rw x t₁))
            (eq : proof Γ ( signature.formula.equation t₁ t₂))
             : proof Γ (φ.rw x t₂)

local infixr `⊢`:55 := proof
```

Rewriting is used in the definition (`φ.rw`) as well as the notion of a variable being substitutable by a term in a formula; those are defined in the usual ways. We also require defining the notion of a variable being free in a formula, and the notion of a variable being "abstract" (or not appearing free) in a set of formulas. Originally we intended to detail the implementation of these in the following sections, but we have been constrained by the space afforded to us in this publication. It will as such have to suffice here to point the reader to the original source code in [Viana 2020], as it is well documented enough to be very readable on its own.

3.2. Semantic Definitions

For semantics, we start by defining a first-order structure for the signature σ, and the type of variable assignments:

```
-- a structure for the language defined by σ, with domain in type α.
structure signature.structure (σ : signature) (α : Type u) [nonempty α] :=
    -- functional interpretation
    (I₁ : Π {n}, σ.nary n → (fin n → α) → α)
    -- relational interpretation
    (I₂ : Π {n}, σ.nrary n → (fin n → α) → Prop)

-- type of variable assignments
def signature.vasgn (σ : signature) (α : Type u) := σ.vars → α

variables {σ : signature} {α : Type u} [nonempty α]
```

We then define references for terms in the language, and a way to bind a variable to a reference in a variable assignment:

```
-- the reference of a term in a structure relative to an assignment.
def signature.structure.reference' (M : σ.structure α) : σ.term → σ.vasgn α → α
| (signature.term.var x) asg := asg x
| (@signature.term.app _ 0 f _) _ := M.I₁ f fin_zero_elim
| (@signature.term.app _  (n+1) f v) asg := let v₂ := λ k,
 ↪  signature.structure.reference' (v k) asg
                                    in M.I₁ f v₂

-- bind the value of a variable to `val` in an assignment
-- (generates a new assignment).
def signature.vasgn.bind (ass : σ.vasgn α) (x : σ.vars) (val : α) : σ.vasgn α :=
    λy, if y = x then val else ass y
```

The reference was defined differently for constant terms and for non-constant terms arising out of a function application, that is why there are two `@signature.term.app` cases. The first maps the constant to the reference given by the interpretation, while the second first resolves the reference of every argument of the functional symbol to be applied, then interprets the functional symbol and applies the resulting function over the references of the arguments. Notice that the definition of variable assignment wouldn't be possible if equality in the type of variables were not decidable, as that is required for using an `if` statement. Further, we define the relation of satisfiability of formulas in a structure, both with respect to a variable assignment and without one:

```
-- tells whether a formula is true in a structure, relative to
-- an assignment.
def signature.structure.satisfies' : σ.structure α →  σ.formula → σ.vasgn α → Prop
| M ( signature.formula.relational r v) asg :=
          M.I₂ r $ λm,  M.reference' (v m) asg
| M ( signature.formula.for_all x ψ) ass :=
    ∀ (a : α), M.satisfies' ψ (ass.bind x a)
| M ( signature.formula.if_then ψ ψ) asg :=
    let x := M.satisfies' ψ asg,
        y := M.satisfies' ψ asg
    in x → y
| M ( signature.formula.equation t₁ t₂) asg :=
    let x := M.reference' t₁ asg,
        y := M.reference' t₂ asg
    in x = y
| M  signature.formula.false _ := false


-- tells whether a formula is true in a structure, absolutely.
def signature.structure.satisfies : σ.structure α → σ.formula → Prop
| M ψ := ∀ (ass : σ.vasgn α), M.satisfies' ψ ass

-- will reserve ⊨ without subscript for
-- semantic consequence of theories.
```

```
local infixr `⊨₁`:55 := signature.structure.satisfies
```

We have chosen to follow the convention that the functions defined with a ' depend on a choice of assignment, while the ones without it do not. Finally, we define the notion of a formula being a semantic consequence of a set of formulas:

```
-- semantic consequence.
-- Tells whether ψ is true in every model/assignment in which Γ is true.
def signature.follows (σ : signature) (Γ : set σ.formula) (ψ : σ.formula) : Prop :=
    ∀{α : Type u}[h : nonempty α]
     (M : @signature.structure σ α h) (ass : σ.vasgn α),
      (∀ ψ ∈ Γ, @signature.structure.satisfies' σ α h M ψ ass) →
       @signature.structure.satisfies' σ α h M ψ ass

local infixr `⊨`:55 := signature.follows σ
```

3.3. The Proof

Given the introduction of notation in Lean, the statement of the the proof itself can be succinctly introduced in the same format it would be on any reference textbook:

```
-- So pretty.
theorem soundness : Γ ⊢ ψ → Γ ⊨ ψ := ...
```

In order to prove it, we first needed to prove the following lemmas about binding in assignments and the semantics of rewriting variables for terms:

```
lemma bind_symm : ∀ {ass : σ.vasgn α} {x y : σ.vars} {a b}, x ≠ y → (ass.bind x
↪  a).bind y b = (ass.bind y b).bind x a := ...

lemma bind₁ : ∀ {ass : σ.vasgn α} {x : σ.vars}, ass.bind x (ass x) = ass := ...

lemma bind₂ : ∀ {ass : σ.vasgn α} {x : σ.vars} {a b}, (ass.bind x a).bind x b =
↪  ass.bind x b := ...

lemma bind₃ : ∀ {M : σ.structure α} {ψ: σ.formula}{ass : σ.vasgn α}{x : σ.vars}{a},
              x ∉ ψ.free →
              (M.satisfies' ψ (ass.bind x a) ↔
              M.satisfies' ψ ass)
              := ...

lemma fundamental : ∀ y x (M : σ.structure α) ass, abstract_in y Γ →
            (∀ ψ ∈ Γ, M.satisfies' ψ ass) →
            ( ∀ψ ∈ Γ, M.satisfies' ψ (ass.bind y x))
           := ...

lemma rw_semantics : ∀ {M : σ.structure α} {ass:σ.vasgn α} {x t} {ψ: σ.formula},
                     ψ.substitutable x t →
                     (M.satisfies' (ψ.rw x t) ass ↔
                     M.satisfies' ψ (ass.bind x (M.reference' t ass)))
                     := ...
```

We will not reproduce the proof here for lack of space. Suffices to say that the proof proceeds by induction on all possible ways that the set of formulas Γ could prove the formula φ, by showing essentially that all logical rules preserve the validity of formulas. As a corollary we can conclude the consistency of the logical calculus by proving that the empty set of formulas does not derive `formula.false`:

```
def consistent (Γ : set  σ.formula) := ¬ nonempty (Γ ⊢  signature.formula.false)
theorem consistency : consistent (∅ : set σ.formula) := ...
```

The proof works by constructing some structure M, which is then a model for the empty set. It follows by soundness that if `formula.false` could be proven from , then M would satisfy `formula.false`. Since no structure can do that, we have that does not prove `formula.false`.

4. Conclusion

We have summarized our implementation of the soundness proof, many more details could still be given about how the proofs were constructed, and the difficulties which lied therein. Our work still gives many opportunities for expansion, as one obvious yet much more laborious development would be the formalization of the completeness proof of classical predicate logic. For more practical utility, we could extend the syntax of the system to include Hoare triples and exemplify the application of Hoare logic to the formal verification of a particular algorithm in Lean. For greater theoretical gain, we can use Lean for studying quantified modal logics and their algebraizations. There is still much in store for the future.

References

From, A. H., Jensen, A. B., Schlichtkrull, A., and Villadsen, J. (2020). Teaching a formalized logical calculus. *Electronic Proceedings in Theoretical Computer Science*, 313:73–92.

Han, J. M. and van Doorn, F. (2019). A formalization of forcing and the unprovability of the continuum hypothesis.

Jeremy Avigad, R. Y. L. and van Doorn, F. (2017). *Logic and Proof*. `https://leanprover.github.io/logic_and_proof/` [accessed 4/29/2020].

Martin-Löf, P. (1982). *Constructive mathematics and computer programming*. Amsterdam: North- Holland Publishing Company.

Viana, L. C. R. (2020). Github repo: `https://github.com/maxd13/logic-soundness.git`. revision: f15dfd4.