Proving the consistency of Logic in Lean.

Luiz Carlos R. Viana

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

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.*

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 true theorem of mathematics which is at least already suspected in the literature to be true.

As explained in Martin-Löf's paper [1], by applying the Curry-Howard isomorphism [2] between proofs in intuitionistic logic and terms in a typed lambda calculus, we can see any flavour of intuitionistic type theory to be 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 in a way that resembles the more familiar forms of software development. By default, Lean allows for the construction of proofs in intuitionistic logic, but by the declaration of (the non-intuitionistic version 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 fact the community-developed standard library [3] for the Lean prover implements an internal model of ZFC within Lean:

```
-- The ZFC universe of sets consists of the type of pre-sets,
-- quotiented by extensional equivalence.
def Set : Type (u+1) := quotient pSet.setoid.{u}
```

In this paper 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.

Motivation of our Work

One general motivation for the development of interactive theorem provers is the formal verification of mathematical proofs such as Jordan's curve theorem and Fermat's last theorem, which are so large and complex that even expert mathematicians may have trouble understanding them, and for this reason may suspect them to be wrong. However, for simpler results, such as that of the soundness of first-order logic, even though a verification might provide greater security to some, there is already virtually no suspicion in the mathematical community that their canonical, more well-known, proofs might be wrong. So this reason would be very weak to motivate our research.

On the other hand, in the spirit of literate programming, we can set out to investigate the usability of these provers for teaching logic to students by providing a formalized reference of the subject and of its proofs, as a practical tool for the solution of mathematical exercises, and also as 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* [4], 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, even though there are certainly papers [5] mentioning this construction has been achieved before.

With respect to exercises, a teacher in any area of mathematics can already write a class assignment in a proof assistant as a list of theorem declarations. A solution to the assignment consists in a proof of each theorem, which is then automatically checked by the software for correctness, requiring no further correction by a human being. An assignment can even be provided in a game-like format [8] and integrated with the theorem prover. The usage of proof assistants in mathematics is sure to save many teachers much time; furthermore, such usage is sure to spread the knowledge of formal methods and of logic itself to many people who might otherwise not have learnt it.

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 [6], 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.

Implementation

We implemented the different aspects of first order logic, dividing definitions and proofs into syntactical and semantical sections. We present a short summary of our implementation below.

Syntactic Definitions

In our implementation, we assumed the existence of a type of functional symbols and a type of relational symbols as parameters, both accompanied by their respective arity functions:

```
parameter {functional_symbol : Type u}
parameter {relational_symbol : Type u}
parameter {arity : functional_symbol → ℕ}
parameter {rarity : relational_symbol → ℕ}
```

The basic definitions of terms and formulas are given below:

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

-- constant terms.
def nary.term : const → term
| c := term.app c fin_zero_elim

-- formulas
inductive formula
| relational {n : ℕ} (r : nrary n) (v : fin n → term) : formula
| false : formula
| for_all :  ℕ → formula → formula
| if_then : formula → formula → formula
| equation (t₁ t₂ : term) : formula
```

As can be seen, we use the $\{\forall, \rightarrow, \bot\}$ functionally complete fragment of first-order logic in the definition of formulas. For the `if_then` constructor we set up a specific notation, since Lean will allow us to do that:

```
reserve infixr ` ⇒ `:55
class has_exp (α : Type u) := (exp : α → α → α)
infixr ⇒ := has_exp.exp

instance formula.has_exp : has_exp formula := ⟨formula.if_then⟩
```

And now we can write φ ⇒ ψ instead of `formula.if_then φ ψ`. We also defined the other connectives:

```
def formula.not (φ : formula)   := φ ⇒ formula.false
def formula.or  (φ ψ : formula) := φ.not ⇒ ψ
def formula.and (φ ψ : formula) := (φ.not.or ψ.not).not
def formula.iff (φ ψ : formula) := (φ ⇒ ψ).and (ψ ⇒ φ)
```

The proof system was implemented inductively using natural deduction rules according to the following definition:

```
-- deductive consequence of formulas: Γ ⊢ φ
inductive entails : set formula → formula → Prop
| reflexivity (Γ : set formula) (φ : formula)(h : φ ∈ Γ) : entails Γ φ
| transitivity (Γ Δ : set formula) (φ : formula)
               (h₁ : ∀ ψ ∈ Δ, entails Γ ψ)
               (h₂ : entails Δ φ) :  entails Γ φ
| modus_ponens
            (φ ψ : formula) (Γ : set formula)
            (h₁ : entails Γ (φ ⇒ ψ))
            (h₂ : entails Γ φ)
             : entails Γ ψ
| intro
            (φ ψ : formula) (Γ : set formula)
            (h : entails (Γ ∪ {φ}) ψ)
             : entails Γ (φ ⇒ ψ)
| for_all_intro
            (Γ : set formula) (φ : formula)
            (x : ℕ) (xf : x ∈ φ.free)
            (abs : abstract_in x Γ)
            (h : entails Γ φ)
             : entails Γ (formula.for_all x φ)
| for_all_elim
            (Γ : set formula) (φ : formula)
            (x : ℕ) (t : term)
            (sub : φ.substitutable x t)
            (h : entails Γ (formula.for_all x φ))
             : entails Γ (φ.rw x t)
| exfalso (Γ : set formula) (φ : formula)
          (h : entails Γ formula.false)
          : entails Γ φ
| by_contradiction (Γ : set formula) (φ : formula)
                   (h : entails Γ φ.not.not)
                   : entails Γ φ
| identity_intro
            (Γ : set formula) (t : term)
             : entails Γ (formula.equation t t)
| identity_elim
            (Γ : set formula) (φ : formula)
            (x : ℕ) (xf : x ∈ φ.free)
            (t₁ t₂: term)
            (sub₁ : φ.substitutable x t₁)
            (sub₂ : φ.substitutable x t₂)
            (h : entails Γ (φ.rw x t₁))
            (eq : entails Γ (formula.equation t₁ t₂))
             : entails Γ (φ.rw x t₂)

local infixr `⊢`:55 := entails
```

Rewriting is used in the definition (`φ.rw`) as well as the notion of a variable being substitutable by a term in a formula. We also require defining the notion of a variable being free in a formula, and the notion of a variable being "abstract" in a set of formulas. We will detail the implementation of these in the following sections.

Rewrite system

For terms, rewriting/substituting a variable for another term is a simple procedure:

```
def term.rw : term → ℕ → term → term
| (term.var a) x t := if x = a then t else term.var a
| (term.app f v) x t :=
    let v₂ := λ m, term.rw (v m) x t in
    term.app f v₂
```

Notice that lean allows us to call this function as `t₁.rw x t₂` where `t₁` and `t₂` are terms and `x` is a variable (which we chose to represent as a natural number). And this will work the same for the rewriting of terms in formulas. For formulas, the definition is simple as well, but we must be concerned with defining also the predicate which tells whether a term is substitutable for a variable in a formula:

```
def formula.rw : formula → ℕ → term → formula
| (formula.relational r v) x t :=
    let v₂ := λ m, (v m).rw x t in
    formula.relational r v₂
| (formula.for_all y φ) x t :=
    let ψ := if y = x then φ else φ.rw x t in
    formula.for_all y ψ
| (formula.if_then φ ψ) x t := (φ.rw x t) ⇒ (ψ.rw x t)
| (formula.equation t₁ t₂) x t := formula.equation (t₁.rw x t) (t₂.rw x t)
| φ _ _ := φ

def formula.substitutable  : formula → ℕ → term → Prop
| (formula.for_all y φ) x t := x ∉ (formula.for_all y φ).free ∨
                                (y ∉ t.vars ∧ φ.substitutable x t)
| (formula.if_then φ ψ) y t := φ.substitutable y t ∧ ψ.substitutable y t
| _ _ _ := true -- atomic formulas
```

Variables

We need to concern ourselves with defining free variables as well:

```
-- the variables present in a term
def term.vars : term → set ℕ
| (term.var a) := {a}
| (term.app f v) :=
    let v₂ := λ m, term.vars (v m) in
    ⋃ m, v₂ m

-- free variables
def formula.free : formula → set ℕ
| (formula.relational r v) := ⋃ m, (v m).vars
| (formula.for_all x φ) := φ.free - {x}
| (formula.if_then φ ψ) := φ.free ∪ ψ.free
| (formula.equation t₁ t₂) := t₁.vars ∪ t₂.vars
| formula.false := ∅
```

We must also define the notion of a variable being "abstract" in a set of formulas, which is to say that the variable doesn't appear free in any formula of the set:

```
def abstract_in : ℕ → set formula → Prop
| x S := x ∉ (⋃ φ ∈ S, formula.free φ)
```

Given these definitions, all dependencies of our deductive system are defined. We can then provide some simple examples of proofs.

Proof examples

We declare sets of formulas and formulas to be referenced in the context of our proofs:

```
variables (Γ Δ : set formula) (φ : formula)
```

First we provide a simple example of how to prove that a theory proves a particular formula:

```
theorem self_entailment : Γ ⊢ (φ ⇒ φ) :=
begin
    apply entails.intro,
    apply entails.reflexivity (Γ∪{φ}) φ,
    simp
end
```

Essentially here we apply the $\to -\text{introduction}$ rule to prove $\varphi \to \varphi$. The goal then becomes to prove $\Gamma \cup \{\varphi\} \vdash \varphi$. We can prove this by noticing that $\varphi \in \Gamma \cup \{\varphi\}$, therefore by reflexivity $\Gamma \cup \{\varphi\} \vdash \varphi$, and the proof is done. Next we provide an example of how to prove a simple syntactical meta-theorem:

```
theorem monotonicity : Δ ⊆ Γ → Δ ⊢ φ → Γ ⊢ φ :=
begin
    intros H h,
    have c₁ : ∀ ψ ∈ Δ, entails Γ ψ,
        intros ψ hψ,
        apply entails.reflexivity Γ ψ,
        exact H hψ,
    apply entails.transitivity;
    assumption,
end
```

Here we introduce the hypothesis `H : Δ ⊆ Γ` and `h : Δ ⊢ φ`, and prove first the intermediate result `c₁` which says that every formula of Δ is entailed by Γ. This we prove by noticing that, by `H`, every formula of Δ is a formula of Γ, and therefore is entailed by Γ by the reflexivity rule. It then suffices to apply the transitivity rule using `c₁` to conclude that `Γ ⊢ φ`. The assumption tactic is used here to tell Lean to look into the local proof context for whatever proofs it needs to apply the transitivity rule. Since `c₁` as well as `H` and `h` are in the local context, and are needed for the rule, they are filled in for us.

Semantic Definitions

For semantics, given a type α, which has at least one element, we define 3 types:

- The type of functions mapping functional symbols of arity n to functions of type $\alpha^n \to \alpha$.
- The type of functions mapping relational symbols of arity n to n-ary relations of α.
- The type of functions mapping variables to elements of α, i.e. variable assignments.

```
parameters {α : Type u} [nonempty α]

-- functional interpretation
def fint  {n : ℕ} := nary n → (fin n → α) → α
-- relational interpretation
def rint {n : ℕ} := nrary n → (fin n → α) → Prop
-- variable assignment
def vasgn := ℕ → α
```

It is then trivial to define a structure for the language. Now since the term `structure` is a reserved word in Lean, we use a synonym for that word in our definitions, and call it simply a model:

```
structure model :=
    (I₁ : Π {n}, @fint n)
    (I₂ : Π {n}, @rint n)
```

Even though a first-order structure in logic is only considered to be a model relative to a particular formula, or set of formulas, which it satisfies, we take the two words to be synonymous; since every structure is a model of some formula or another.

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

```
def model.reference' (M : model) : term → vasgn → α
| (term.var x) asg := asg x
| (@term.app _ _  0 f _) _ := model.I₁ M f fin_zero_elim
| (@term.app _ _  (n+1) f v) asg := let v₂ := λ k, model.reference' (v k) asg
                                    in model.I₁ M f v₂

def vasgn.bind (ass : vasgn) (x : ℕ) (val : α) : vasgn :=
    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 `@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. Further, we define the relation of satisfiability of formulas in a model, both with respect to to a variable assignment and without one:

```
def model.satisfies' : model → formula → vasgn → Prop
| M (formula.relational r v) asg :=
    M.I₂ r $ λm,  M.reference' (v m) asg
| M (formula.for_all x φ) ass :=
    ∀ (a : α), M.satisfies' φ (ass.bind x a)
| M (formula.if_then φ ψ) asg :=
    let x := M.satisfies' φ asg,
        y := M.satisfies' ψ asg
    in x → y
| M (formula.equation t₁ t₂) asg :=
    let x := M.reference' t₁ asg,
        y := M.reference' t₂ asg
    in x = y
| M formula.false _ := false

def model.satisfies : model → formula → Prop
| M φ := ∀ (ass : vasgn), M.satisfies' φ ass

local infixr `⊨₁`:55 := model.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:

```
def theory.follows (Γ : set formula) (φ : formula): Prop :=
    ∀ (M : model) ass,
    (∀ ψ ∈ Γ, M.satisfies' ψ ass) →
    M.satisfies' φ ass

local infixr `⊨`:55 := theory.follows
```

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 ⊢ φ → Γ ⊨ φ := ...
```

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 : vasgn} {x y : ℕ} {a b},
                   x ≠ y →
                  (ass.bind x a).bind y b = (ass.bind y b).bind x a
                  := ...

lemma bind₁ : ∀ {ass : vasgn} {x : ℕ}, ass.bind x (ass x) = ass := ...
lemma bind₂ : ∀ {ass : vasgn} {x : ℕ} {a b},
               (ass.bind x a).bind x b = ass.bind x b := ...

lemma bind₃ : ∀ {M:model} {φ:formula}{ass : vasgn}{x : ℕ}{a},
               x ∉ φ.free →
               (M.satisfies' φ (ass.bind x a) ↔
               M.satisfies' φ ass)
               := ...

lemma fundamental : ∀ y x (M : model) ass, abstract_in y Γ →
             (∀ φ ∈ Γ, M.satisfies' φ ass) →
             ( ∀φ ∈ Γ, M.satisfies' φ (ass.bind y x))
             := ...

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

We will not reproduce the proof here for lack of space. Furthermore, many references [7] already exist for an outline of how this sort of proof works. 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 formula) := ¬ Γ ⊢ formula.false
theorem consistency : consistent ∅ := ...
```

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

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. There is still much in store for the future.

References

[1] 1982, “Constructive mathematics and computer programming”, in Logic, methodology and philosophy of science VI, Proceedings of the 1979 international congress at Hannover, Germany, L.J. Cohen, J. Los, H. Pfeiffer and K.-P. Podewski (eds). Amsterdam: North- Holland Publishing Company, pp. 153–175.

[2] 1958, "Combinatory Logic", Curry, Haskell B. and Robert Feys, Amsterdam: North-Holland.

[2] 2020, "The lean mathematical library", in Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, The mathlib Community, ACM.

[4] 2017, Logic and Proof, Jeremy Avigad, Robert Y. Lewis, and Floris van Doorn, https://leanprover.github.io/logic_and_proof/ (accessed in 4/29/2020).

[5] 2019, "A formalization of forcing and the unprovability of the continuum hypothesis", Jesse Michael Han and Floris van Doorn.

[6] 1969, "An Axiomatic Basis for Computer Programming", The Queen's University of Belfast, Departament of Computer Science Northen Ireland, C.A.R. Hoare.

[7] 1972, "A Mathematical Introduction to Logic", Herbert B. Enderton.

[8] 2020, "Natural Number Game", http://wwwf.imperial.ac.uk/~buzzard/xena/natural_number_game/ (accessed in 4/29/2020), Kevin Buzzard, Mohammad Pedramfar.