Final Report – Theorem Proving in Lean

Faculty Mentor: Philipp Hieronymi
Project Leaders: Vaibhav Karve, Scott Harman, Eion Blanchard
IGL Scholars: Nikil Ravi, Kyle Thompson, Fenglong Zhao
Tianfan Xu, Joel Schargorodsky, Noble Wulffraat

December 2020

Abstract

Theorem proving uses computers to prove or verify mathematical statements. In this project, we formalize several ideas in model theory using a theorem prover called Lean. We have successfully formalized notions such as languages, structures, terms, formulas, sentences and models, and have made some progress towards formalizing o-minimality. The code can be found at https://github.com/vaibhavkarve/igl2020.

1 Introduction

A *theorem prover* is a program that takes a statement and verifies or decides it. Given that most proof methods can be reduced to a set of axioms and associated rules, we can use theorem provers to either prove that a statement is correct, or verify the correctness of a given proof. Because of their reliability and efficiency, theorem provers can be useful in many contexts in mathematics and beyond. The benefits of digitizing known mathematics are manifold – from AI assistants for mathematicians, to proof verifiers for journal submissions, to digital archives of all mathematical knowledge – the possibilitees are many.

Lean is an open-source theorem-prover and programming language that aims to bridge the gap between interactive and automated theorem proving. It allows users to create definitions, prove theorems, and interactively perform related computations to support the construction of fully specified axiomatic proofs.[1]

As a very basic example of some Lean code, consider the following definition of an even number in Lean –

```
def even (n : ℕ) : Prop := ∃ d, n = 2 * d
```

In order to now prove that a given number, say 6, is even, we would have to convince Lean that 6 satisfies the definition of an even number.

```
lemma even_six : even 6 :=
  by {unfold even, use 3, norm_num}
```

[1] https://leanprover.github.io/

The code inside `by {...}` consists of tactics – these are similar to proof statements written by a human mathematician. Informally, `unfold even` tells Lean to read the definition of an even number, `use 3` asks Lean to use 3 as d in the definition, and `norm_num` is a tactic in Lean that can be used here to verify that $2 \times 3 = 6$.

An important fact to note here is that Lean uses *type theory* instead of *set theory*. In type theory, every expression has an associated type. For instance, we write $a : A$ (read: "a is of type A") instead of $a \in A$. We can build new types from existing ones. For example, if α and β are types, $\alpha \longrightarrow \beta$ denotes the type of functions from α to β.

The aim of this project is to use Lean to formalize ideas in first-order logic and model theory. We focus on concepts such as languages, structures, embeddings, variables, terms, sentences, models, and o-minimality.

2 Background

The team members first learned the basics of Lean, including basic syntax, recursion, inductive types, tactic-style proofs and term-style proofs. We studied the details of how Lean is designed—how computations are handled, the type structure in Lean, the overall organization of packages and mathematical areas, and more.

We have also learned about ideas in model theory, like languages, structures, embeddings, formulae, sentences, theories and models.

3 Progress

3.1 Languages

Firstly, we introduce languages. This part is important for what follows and forms the basis for the whole project.

Definition: A **language** L is a set of constant, function, and relation symbols with associated arities. Each function or relation symbol has an *arity*, which is a natural number counting how many arguments a symbol requires. Constant symbols are 0-ary function symbols.

```
structure lang : Type 1 :=
(F : ℕ → Type)    -- functions. ℕ keeps track of arity.
(R : ℕ → Type)    -- relations
```

The above code is our Lean definition of a language. Given an arity n $\in N$, the definition above keeps track of how many functions and relations of arity n there are in the language. Below, we also provide an example of a particular language.

```
/-- The language of ordered sets is the language of sets with binary relation {\
    textless}.-/
def ordered_set_lang : lang :=
  {R := λ n : ℕ, if n=2 then unit else empty,
   F := function.const ℕ empty}
```

This is our Lean definition of a language $L = \{<\}$, where $<$ is a binary relation. Syntactically, this language is characterized by having a single binary relation symbol. The function "R" returns a singleton type for input arity 2 and the empty type otherwise. Similarly, "F" always returns the empty type.

3.2 Structures

Languages rely on *interpretation* in structures to bear semantic meaning, so we now define structures.

Definition: For a language L, an **L-structure** $\mathcal{M}$ is a domain M paired with interpretations of each symbol in L.

```
structure struc (L : lang) : Type 1 :=
(univ : Type)                -- universe/domain
(F (n : ℕ) (f : L.F n) : Func univ n)
                     -- function interpretation
(R (n : ℕ) (r : L.R n) : set(vector univ n))
                     -- relation interpretation
(C : L.C → univ)     -- constant interpretation
```

The above code is our Lean definition of a structure. It tells Lean that a structure for a language L consists of a universe, along interpretations of functions, relations and constants. The following code snippet is an example of a structure in Lean. We will have to provide Lean the universe and the interpretations of the functions, relations and constants in the language, as shown below.

```
/-- Group is a structure of the group language-/
def group_is_struc_of_group_lang {A : Type} [group A] :
  struc (group_lang) :=
  {univ := A,
   F := by {intros n f,
            iterate {cases n, cases f},
              exact 1,
            iterate {cases n, cases f},
              exact group.inv,
            iterate {cases n, cases f},
              exact group.mul,
            cases f},
    R := λ _ r, empty.elim r,
    C := λ c, 1}
```

3.3 Terms

Definition: The set of L-*terms* is the smallest set T such that

i) $c \in T$ for each constant symbol $c \in C$

ii) each variable symbol $v_i \in T$ for $i = 1, 2, ...,$ and

iii) if $t_1, ..., t_{n_f} \in T$ and $f \in F$, then $f(t_1, ..., t_{n_f}) \in T$.

We then constructed the following definition in Lean.

```
inductive term : ℕ → Type
| con : L.C → term 0
| var : ℕ → term 0
| func {n : ℕ} : L.F n → term n
| app {n : ℕ} : term (n + 1) → term 0 → term n
```

We introduced the concept of ”leveled-terms” so as to effectively use recursion in our definition. A function of arity n is defined to be a term of level n. Given a term of level $n+1$ and a term of level 0, we can produce a term of level n. Thus, given a term of level n and a list $[t_1, t_2, ..., t_n]$ of terms of level 0, we can produce a term of level 0. This is analogous to the third item in Marker's definition, i.e. $f(t_1, ..., t_{n_f})$. Thus, the set of L-terms is a subset of the set we have defined. For a language L, `term L n` represents all terms of L with level n, therefore we can attain the set of L-terms with `term L 0`.

3.4 Formulas

Definition: For a language L, **L-formulas** are finite strings comprised of symbols from L, the equality relation, variables, the logical connectives, quantifiers, and parentheses.

```
inductive formula (L : lang)
| tt : formula
| ff : formula
| eq  : term L 0 → term L 0 → formula
| rel : Π (n : ℕ), L.R n → vector (term L 0) n
        → formula
| neg : formula → formula
| and : formula → formula → formula
| or  : formula → formula → formula
| exi : ℕ → formula → formula
| all : ℕ → formula → formula
```

We construct the definition of formula based on cases according to equality =; logical connectives negation ¬, conjunction ∧, and disjunction ∨; and existential ∃ and universal ∀ quantifiers.

3.5 Sentences

Definition: A variable is **free** in a formula if it is not quantified. Otherwise, this variable is **bound**.

Definition: Sentences are formulas with no free variables.

Since determining if a formula is a sentence depends on the variables appearing in the formula, we define helper functions to:

- extract a list of variables appearing in a formula and
- determine whether or not a given formula is variable-free.

With these, we may easily define sentences in Lean.

```
def vars_in_formula {L : lang} : formula L → finset ℕ
| ⊤'                  := ∅
| ⊥'                  := ∅
| (₁t='₂t)            := vars_in_term ₁t ∪ vars_in_term ₂t
| (formula.rel _ r ts) := vars_in_list (ts.to_list)
| (¬' ϕ)        := vars_in_formula ϕ
| (ϕ₁ ∧' ϕ₂)  := vars_in_formula ϕ₁ ∪ vars_in_formula ϕ₂
| (ϕ₁ ∨' ϕ₂)  := vars_in_formula ϕ₁ ∪ vars_in_formula ϕ₂
| (∃' v ϕ)    := vars_in_formula ϕ ∪ {v}
| (∀' v ϕ)    := vars_in_formula ϕ ∪ {v}

def is_var_free (n : ℕ) {L : lang} : formula L → Prop
| ⊤'                  := true
| ⊥'                  := true
| (₁t='₂t)            := true
| (formula.rel _ r ts) := true
| (¬' ϕ)        := is_var_free ϕ
| (ϕ₁ ∧' ϕ₂)  := is_var_free ϕ₁ ∧ is_var_free ϕ₂
| (ϕ₁ ∨' ϕ₂)  := is_var_free ϕ₁ ∧ is_var_free ϕ₂
| (∃' v ϕ)    := v ≠ n ∧ is_var_free ϕ
| (∀' v ϕ)    := v ≠ n ∧ is_var_free ϕ


def var_is_bound {L : lang} (n : ℕ) (ϕ : formula L) : Prop := ¬ is_var_free n ϕ


def is_sentence {L : lang} (ϕ : formula L) : Prop :=
  ∀ n : ℕ, (n ∈ vars_in_formula ϕ → var_is_bound n ϕ)

def sentence (L : lang) : Type := ϕ{ : formula L // is_sentence ϕ}
```

3.6 Models

With what we have defined so far, we can now define in Lean what it means for an L-structure to model an L-sentence.

Definition: Let L be a language, $\mathcal{M}$ be an L-structure, and φ be an L-sentence. We say $\mathcal{M} \models \varphi$ (read: $\mathcal{M}$ models φ) if the interpretations of symbols from φ according to $\mathcal{M}$ yield the truth of φ.

We define this inductively in Lean as follows.

```
def models {L : lang} {M : struc L} :
    (ℕ → M.univ) → formula L →  Prop
| va ⊤'           := true
| va ⊥'           := false
| va (₁t =' ₂t)   := (term_interpretation M va ₁t)
    = (term_interpretation M va ₂t)
| va (formula.rel _ r ts) := vector.map (term_interpretation M va) ts ∈ M.R ts.
    length r
| va ¬' ϕ              :=  ¬ models va ϕ
| va (ϕ₁ ∧' ϕ₂)       := models va (ϕ₁) ∧ models va (ϕ₂)
| va (ϕ₁ ∨' ϕ₂)       := models va (ϕ₁) ∨ models va (ϕ₂)
| va (∃' v ϕ)         :=
    ∃ (x : M.univ), models (λ n, if n=v then x else va n) ϕ
| va (∀' v ϕ)         :=
    ∀ (x : M.univ), models (λ n, if n=v then x else va n) ϕ
```

4 Future Work

The next steps in this project will entail:

- Formalizing the notion of definability of sets, and
- Formalizing dense linear orderings and o-minimality

5 Acknowledgements

We would like to thank the Illinois Geometry Lab and the Department of Mathematics at the University of Illinois at Urbana Champaign for supporting this project. We are grateful to our faculty mentor and graduate mentors for their continued support, guidance and mentorship during the course of this project.

References

[1] David Marker, *Introduction to Model Theory.* Springer-Verlag New York, 2002.

[2] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean.* https://leanprover.github.io/theorem_proving_in_lean/