Completeness of Theories of Arithmetic in Lean

By

Nicholas Pilotti

An Undergraduate Thesis

Submitted to the Department of Computer and Information Science

University of Pennsylvania

In Partial Fulfillment of the Requirements

for the Degree of Bachelor of Science in Engineering

May 2022

Preface

My purpose in this paper is to present what I have learned of Lean and my experience implementing concepts from introductory logic. Lean has received greater attention in the past several years as a language for formalizing pure mathematics. This requires for more people to learn Lean, and while the results achieved in this project are by no means original, I hope that it will be another resource for those who want to learn theorem proving. For those already in the formalization community, I hope this will be a useful case study for what an undergraduate can do in two semesters, starting with no knowledge of Lean or direct access to the experts.

There are already several great resources available for learning Lean, and the Lean community website provides an excellent interface for beginners to find information on the language. However, in my experience, there is a steep learning curve from understanding the basic technical aspects of the language and beginning the work of formalizing results yourself. For one, the mathlib library is vast and lacks documentation or explanation that is accessible to beginners. It may not be obvious why mathlib is implemented the way it is or how to use the results available. If one hope to learn from the literature, there are many fantastic papers that detail formalization results. Although, these papers cover topics that most undergraduates do not have background in. This is of course fine, because their purpose is to either educate working mathematicians on theorem proving, demonstrate the feasibility of Lean as a tool for formalizing mathematics, or to present progress to the pre-existing formalization community. I hope that this project will bridge the gap between a basic introduction to the language and a guide to formalizing a result. I think my selected topic is well suited to this because it will be accessible to most undergraduate students in either mathematics or computer science, but at the same time the objects under study are complex enough (in the computer science sense) to present some of the challenges that arise in formalization.

This paper was completed as part of an undergraduate senior thesis in Computer Science. I want to thank my mentors, Caleb Stanford and Li-Yao Xia, for their time, invaluable contributions, and advice. Early on, the Zulip community chat was an essential resource for learning the language and I would have been lost without it. I also would like to thank my thesis advisor Rajeev Alur, reader Oleg Sokolsky, and undergraduate supervisor Joseph Devietti.

Abstract

I present my implementation of results from elementary logic in the theorem prover Lean 3, with the aim of proving completeness of certain theories of arithmetic. I give a description of my implementation of languages, proofs, and sequent calculus rules. I detail my proofs of disjunctive normal form and quantifier elimination on number theoretic formulas involving only zero and the successor operation.

Introduction

It is well known that Gödel proved that arithmetic, under a sufficiently strong set of axioms, is an incomplete theory. On the other hand, the theory of arithmetic which includes only an order relation and the operations of successor and addition, is a complete theory. The sentences of this theory are uninteresting, in the sense that they cannot state any unresolved conjectures of number theory or really any deep theorems at all. However, this result has at least a few interesting consequences. For the mathematician, it sets a lower bound on how complex a theory of arithmetic must be in order to construct paradoxical statements, as Gödel was able to do from the Peano Axioms. Particularly for model theorists, the proof given here is one of the easiest examples of proving completeness through quantifier elimination, a common technique in the field. Quantifier elimination can sometimes be proven through purely syntactic operations on formulas. This is practically quite useful in the field of formal verification, because such a proof will usually provide an explicit decision procedure for any sentence in that theory.

This paper is interested in that last application, particularly in implementing such a proof in the formal verification language Lean. Intended for undergraduates and mathematicians interested in theorem proving, I present a formalized proof that arithmetic restricted to just a successor function, $(\mathbb{N}, 0, S)$, is a complete theory. More advanced implementations of some of the same concepts can be found in the flypitch project [1] and more recently in mathlib.

The proof follows the introductory logic textbook by Enderton [2]. Given a theory $Th(\Gamma)$ in a language L, we prove that any formula ϕ is equivalent to a quantifier free formula. To prove quantifier elimination in a particular theory, it suffices to check quantifier elimination on formulas which are a single quantifier on a conjunction of literals. The proof of this little lemma relies on the result that in any theory all formulas are equivalent to a formula in disjunctive

normal form. To implement this in Lean, I built up a typed theory of languages and proofs. This paper will only give a broad outline of the implementation and the code can be found in the public GitHub repository for this project [4].

The Lean Community and Mathlib

Lean is an open-source theorem prover and programming language released by Microsoft Research in 2013. Like other theorem provers, such as Coq, Lean is a functional language based on the calculus of inductive constructions. While there are technical differences between Lean and other programming languages of its type, one advantage it has right now for use in mathematics is the community that has been built around it. The project of formally verifying proofs in mathematics goes back to at least the early 20th century and has played a central role in the development of modern logic, but theorem provers have received more attention in computer science departments—where they are used to formally verify software—than from pure mathematicians.

On the other hand, in the past several years there has been a focused community effort from mathematicians and computer scientists to formalize the standard undergraduate mathematics curriculum and beyond in Lean. The lean mathematical library, mathlib, is an open-source repository maintained on github by the Lean community. Collaboration happens on the online Zulip chatroom, where anyone is welcome to contribute new code or ask questions.

The Curry-Howard Isomorphism

The basis for Lean's theorem proving power is the Curry-Howard isomorphism, a correspondence between proofs of a *theorem* and instances of a *type*. As an example, consider the logical formula "$P \rightarrow Q$." We commonly interpret this as saying that from some proposition denoted by P, we may always deduce the proposition Q. The Curry-Howard isomorphism suggests another way of understanding this. We can interpret the formula "$P \rightarrow Q$" as the type of a function that takes a proof of the proposition P and from it constructs a proof of Q. That is, the statement of a theorem is a type in a computer language, and the proof of that theorem is an instance of that type.

As another example, consider the formula $P \wedge Q \rightarrow Q \wedge P$. A proof of this formula under the Curry-Howard isomorphism would go as follows: Letting P and Q be variables denoting

propositional statements, suppose we have a proof (or "instance") of the proposition $P \wedge Q$. Apply the rule called "and elimination on the left," which is a function that takes a proof of $P \wedge Q$ and constructs a proof of P, and likewise the rule "and elimination on the right" that takes a proof of $P \wedge Q$ and constructs a proof of Q. Now apply the rule called "and introduction," which is a function that takes a proof of Q and a proof P and constructs a proof of $Q \wedge P$. What we have done is described a procedure for constructing a proof of $Q \wedge P$ from any proof of $P \wedge Q$. In other words, we just defined a function with type $P \wedge Q \ \rightarrow Q \wedge P$.

Those who want to learn more about how to use Lean are encouraged to read the textbook *Theorem Proving in Lean* [3], which can be found on the Lean Community website. For the moment, it will only be necessary to understand inductive types in Lean, which I use as an opportunity to introduce my implementation of languages and formulas.

Inductive Types and Languages

For anyone with significant experience reading or writing proofs in mathematics, they will quickly find that the most straightforward way to write down a proof in natural mathematical language does not easily translate into computer code. In standard mathematics, a notion of type is often only implied, while in Lean the success of a proof is often a measure of the extent to which it leverages types. So, while proofs in Lean are often semantically the same as proofs carried out in pen and paper mathematics, the way in which the idea is explicated in code can be made more elegant by a good use of types.

As an illustration of this point, I will introduce my definition of languages, formulas, and the hierarchy of types I made for this project. The code snippet defining a language is as follows:

```
structure language :=
(functions : ℕ → Type) (relations : ℕ → Type)
```

A language is a structure (a construction in Lean which comparable to a class in object-oriented programming such as Java or C++) containing functions and relations. For each natural number n, there is a type of n-ary functions and a type of n-ary relations. Terms and formulas are defined by

```
inductive term
```

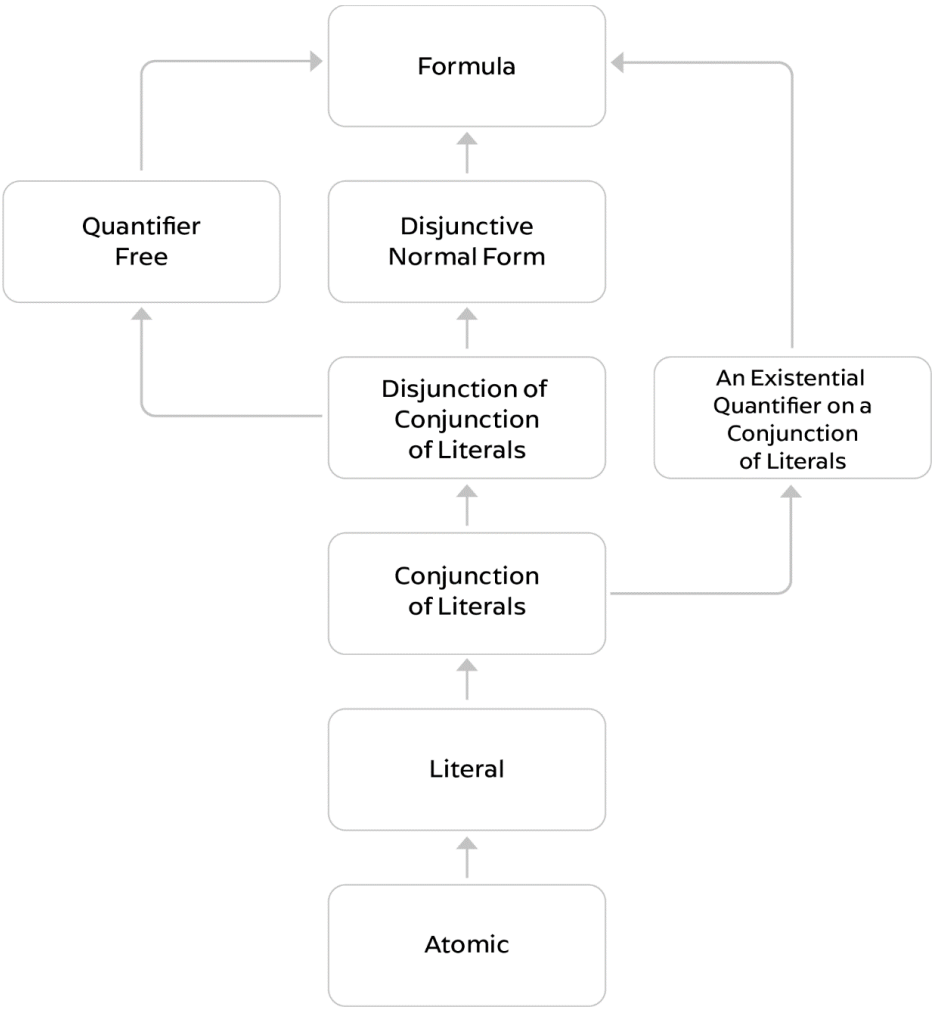
```
| var           : ℕ → term
| func {n : ℕ}  : L.functions n → (fin n → term) → term
```

```
inductive formula
| falsum      : formula
| eq          : @term L → @term L → formula
| rel {n : ℕ} : L.relations n → (fin n → @term L) → formula
| neg         : formula → formula
| or          : formula → formula → formula
| all         : ℕ → formula → formula
```

These are inductive definitions that come with several type constructors. Constructors can be applied to instances of other types. For example, the **var** constructor applied to a natural number produces a term, meant to represent a variable, usually denoted by v_i in written mathematics. The **func** constructor takes a natural number $\boldsymbol{n}$ designating arity, a function symbol (such as `` `+` `` or `` `*` ``), and a list of $\boldsymbol{n}$ arguments to the function which are all terms.

In order to make Lean's system of types work more seamlessly, coercions should be defined for identifying one type with another. Coercions are an example of what is known in Lean as typeclasses, data which may be attached to types and retrieved by the compiler when needed. The **has_coe A B** typeclass is simply a container for a function from **A** to **B.** The Lean compiler will search for an instance of **has_coe** whenever an instance of type **A** is given, but an instance of type **B** was expected, and automatically apply the coercion.

In my proof of disjunctive normal form and the quantifier elimination lemma, a hierarchy of type definitions was used to ultimately simplify the code. Each of these definitions also comes with a coercion to a higher type, represented by the tree below.



For an example of coercions in Lean, see the following code snippet:

```
/- Conjunctions of literals definition -/
inductive conjunction_literal
| literal : literal L → conjunction_literal
| conjunction : conjunction_literal → conjunction_literal → conjunction_literal

/- Conversion from a literal to a conjunction of literals -/
def literal_to_conjunction_literal : literal L → conjunction_literal L
| literal := conjunction_literal.literal literal

/- has_coe typeclass instance -/
instance literal_to_conjunction_literal_coe (L : language) :
  has_coe (literal L) (conjunction_literal L) :=
  ⟨literal_to_conjunction_literal L⟩
```

As we discussed at the beginning of this section, writing good type definitions will vastly simplify proofs. The proof of disjunctive normal form will ultimately only need to induct on the type definitions provided and apply the proof rules defined in the following section. Creating a carefully constructed hierarchy of types is crucial to formalization practice, as one only needs to look at mathlib (c.f. *algebra/groups/defs.lean* for one good example) to see. In fact, once one is comfortable with completing lemmas, I found that the most difficult part of using Lean is having the right definitions for types to make lemma proving as easy as possible.

Proofs and Sequent Calculus Rules

My definition of proofs takes a sequent calculus approach. Axioms are represented by an arbitrary type and there is a finite list for keeping track of the context. Introduction and elimination rules are given for the primitive formula connectives defined above, along with an axiom for classical logic. The cut rule is assumed as an axiom, since the proof of cut elimination falls well outside the scope of this project.

```
Inductive Prf (A : Type) [has_coe A (formula L)] : list (formula L) → formula L →
Prop
| Axiom : ∀ {Γ : list (formula L)} a ϕ, a = ϕ → Prf Γ ϕ
| Assumption : ∀ {Γ : list (formula L)} n ϕ, Γ.nth n = some ϕ → Prf Γ ϕ
| Bot_elim : ∀ {Γ : list (formula L)} ϕ, Prf Γ F → Prf Γ ϕ
| Not_elim : ∀ {Γ : list (formula L)} ϕ ψ, Prf Γ ~ϕ → Prf Γ ϕ → Prf Γ ψ
| By_contradiction : ∀ {Γ : list (formula L)} ϕ, Prf (~ϕ::Γ) F → Prf Γ ϕ
| Or_intro_left : ∀ {Γ : list (formula L)} ϕ ψ, Prf Γ ϕ → Prf Γ (ϕ or ψ)
| Or_intro_right : ∀ {Γ : list (formula L)} ϕ ψ, Prf Γ ψ → Prf Γ (ϕ or ψ)
| Or_elim : ∀ {Γ : list (formula L)} ϕ ψ χ, Prf Γ (ϕ or ψ) → Prf (ϕ::Γ) χ → Prf
(ψ::Γ) χ → Prf Γ χ
| All_intro : ∀ {Γ : list (formula L)} ϕ n m, var_not_free_in_axioms_context m A
Γ → Prf Γ (replace_formula_with n (term.var m) ϕ) → Prf Γ (formula.all n ϕ)
| All_elim : ∀ {Γ : list (formula L)} n t ϕ ψ, Prf Γ (formula.all n ϕ) →
substitutable_for t n ϕ → Prf ((replace_formula_with n t ϕ) :: Γ) ψ → Prf Γ ψ
| Cut : ∀ {Γ : list (formula L)} ϕ ψ, Prf Γ ϕ → Prf (ϕ::Γ) ψ → Prf Γ ψ
```

Lean offers flexibility in changing notation, which is crucial to making code more readable. I chose to use the notation 'A| Γ ⊢ ϕ' for provability (i.e., from the axioms **A** and the context **Γ**, the formula **ϕ** follows)

In *prf.lean*, I formally prove the standard logical rules by applying the proof constructors above. I prove the introduction and elimination rules for the other logical connectives, DeMorgan's laws, distribution of conjunction over disjunction and vice versa, and results on quantifiers. For example:

```
def And_elim_left : A|[(p and q)] ⊢ p := begin
      apply By_contradiction,
      apply Not_elim,
      apply Assumption 1, refl,
      apply Or_intro_left,
      apply Assumption 0, refl,
    end
```

However, the above code snippet above is not sufficient for reasoning about proofs in my construction. For every proof rule, there are lemmas named **R_** and **L_** which convert it into a corresponding left sequent calculus rule or right sequent calculus rule. If the goal is to prove something of the form $(*)\quad A \mid [\mathrm{P}, \mathrm{X}, \mathrm{Y}, \mathrm{Z}] \vdash P \vee Q$, I would need to first convert a rule like **Or_intro_left** into the right sequence calculus rule $A \mid \Gamma \vdash P \rightarrow A \mid \Gamma \vdash P \vee Q$. Lean matches the consequent of this expression $(*)$, and it is transformed into $A \mid [\mathrm{P}, \mathrm{X}, \mathrm{Y}, \mathrm{Z}] \vdash P$, which I may resolve with **Prf.Assumption 0**. Alternatively, the goal might look like $(**)\quad A \mid [\mathrm{P} \wedge Q, \mathrm{X}, \mathrm{Y}, \mathrm{Z}] \vdash P$, in which case I would convert **And_elim_left** into the left sequence calculus rule $A \mid \Gamma \vdash \mathrm{P} \wedge Q \rightarrow A \mid P :: \Gamma \vdash \phi \rightarrow \mathrm{A} \mid \Gamma \vdash \phi$. Again, the consequent is matched on the expression $(**)$ and the new goal becomes $A \mid [\mathrm{P}, \mathrm{P} \wedge Q, \mathrm{X}, \mathrm{Y}, \mathrm{Z}] \vdash P$, which again can be resolved through **Prf.Assumption 0**. The types of **R_** and **L_** are as follows:

```
def R_ : (A|[p] ⊢ q) → ((A|Γ ⊢ p) → (A|Γ ⊢ q))
```

```
def L_ : (A|[p] ⊢ q) → (A|Γ ⊢ p) → (A|(q::Γ) ⊢ r) → (A|Γ ⊢ r)
```

Disjunctive Normal Form and Quantifier Elimination

Using the strategy from Enderton, there are two lemmas required to prove completeness of a theory. First, show that any formula is equivalent to a formula in disjunctive normal form. That is, of the form

$$(P_{11} \wedge P_{12} \wedge \dots) \vee (P_{21} \wedge P_{22} \wedge \dots) \vee \dots$$

where all P_{ij} are literals and with possibly any number of quantifiers outside of the expression. Next, the goal is to show that any formula is equivalent to a quantifier free formula in the theory in question. Since all the functions and relations in arithmetic are decidable, then this proves that the theory is complete. Quantifier elimination is in general not possible in an arbitrary theory, but it suffices to check quantifier elimination on formulas of the form

$$\exists x(P_1 \wedge P_2 \wedge \dots)$$

where all P_i are literals. The proof of this lemma is a relatively simple induction and calculation after converting to disjunctive normal form.

In Lean, these proofs are made straightforward by the preceding work and I will not detail them here, but they can be found in the files *dnf.lean* and *quantifier_elimination.lean*.

Completeness of Arithmetic with Successor

In *number_theory.lean*, the language of number theory with successor is defined, along with the usual axioms. I prove that number theory with successor admits quantifier elimination on formulas which are a single existential quantifier on a conjunction of literals. The language defined contains one constant zero (‘0’), one unary function symbol successor (‘S’), and no relation symbols.

```
/- The functions of number theory with successor -/
inductive NT_succ_func : ℕ → Type
| zero : NT_succ_func 0
| succ : NT_succ_func 1

/- The relations of number theory with successor -/
inductive NT_succ_rel : ℕ → Type

/- The language of number theory with successor -/
def NT_succ : language :=
  ⟨NT_succ_func, NT_succ_rel⟩
```

The axioms are defined by an inductive type and a coercion is used for mapping axioms into the formulas they are meant to represent.

```
/- The axioms of number theory with successor -/
inductive NT_succ_Γ
| eq1 : NT_succ_Γ
| eq2 : NT_succ_Γ
| ax1 : NT_succ_Γ
| ax2 : NT_succ_Γ
| ax3 : ℕ → NT_succ_Γ

open NT_succ_Γ

def NT_succ_Γ_to_formula : NT_succ_Γ → formula (NT_succ)
-- Equality is reflexive
| eq1     := formula.all 0 (v₀ ≃ v₀)
-- Functions preserve equality
| eq2     := formula.all 0 (formula.all 1 (v₀ ≃ v₁ ⇒ (succ v₀) ≃ (succ v₁)))
-- Zero does not have a predecessor
| ax1     := formula.all 0 ~((succ v₀) ≃ zero)
-- Successor is injective
| ax2     := formula.all 0 (formula.all 1 ((((succ v₀) ≃ (succ v₁)) ⇒ (v₀ ≃
v₁))))
-- There are no loops
| (ax3 n) := formula.all 0 ~(nth_succ v₀ n ≃ v₀)

instance NT_succ_Γ_to_formula_NT_succ : has_coe NT_succ_Γ (formula NT_succ) :=
⟨NT_succ_Γ_to_formula⟩
```

The proof itself is ultimately a large induction on all the possible cases. For example, for case of a formula which is of the form $\exists x, t_1 = t_2$, a function is used to construct an equivalent quantifier free formula, dividing into subcases, with a recursive step applied on just the last case of $\exists x, St_1 = St_2$.

```
/- A function that converts a formula of the form t₁ ≃ t₂ to an equivalent
quantifier free formula -/
/- Disclaimer: Much of the code here is not syntactically correct, but simplified
for the convenience of the reader -/
@[simp]
def ex_t₁_eq_t₂_to_qf (n : ℕ) : term NT_succ → term NT_succ → qf NT_succ
-- vᵢ ≃ vⱼ
| (v m₁) (v m₂) := if n = m₁ then ~F else (if n = m₂ then ~F else ((v m₁) ≃ (v
m₂)))
-- vᵢ ≃ 0
| (v m₁) (zero arg) := if n = m₁ then ~F else ((v m₁) ≃ (zero' arg))
-- 0 ≃ vᵢ
```

```
| (zero arg) (v m₂) := if n = m₂ then ~F else (zero' arg) ≃ (v m₂)
/- ... -/
-- succ x ≃ succ y
| (succ t₁) (succ t₂) := ex_t₁_eq_t₂_to_qf (t₁ 0) (t₂ 0)
```

Several lemmas are used frequently throughout. For example, if in case of $\exists x, t_1 = t_2$ we have x appearing in neither t_1 or t_2, the formula is simply equivalent to $t_1 = t_2$. I will leave the details of these cases to the curious reader, who can view the most up-to-date version of the code in the repository [4].

References

[1] J. M. Han, F. van Doorn. A Formal Proof of the Independence of the Continuum Hypothesis. In Proc. 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3372885.3373826 2020

[2] H. B. Enderton, "Natural Numbers with Successor" in A Mathematical Introduction to Logic. London: Academic Press, 2001, pp. 178 – 193.

[3] J. Avigad, L. de Moura, S. Kong. Theorem Proving in Lean. 2016

[4] N. Pilotti, QuantifierElimination, GitHub repository, https://github.com/pilottinick/QuantifierElimination