Teaching Logic and Mechanized Reasoning with Lean 4

Jeremy Avigad
(jww Marijn Heule and Wojciech Nawrocki)

Department of Philosophy
Department of Mathematical Sciences

Hoskinson Center for Formal Mathematics

Carnegie Mellon University

November 21, 2021

Overview

This fall, Marijn Heule and I are teaching a new course in Computer Science at Carnegie Mellon, *15-217 Logic and Mechanized Reasoning*.

Wojciech Nawrocki is our teaching assistant.

The course number signifies that it is appropriate for second-year students in Computer Science.

The prerequisites are 15-150 Functional Programming and 15-151 Mathematical Foundations for Computer Science.

First-year CS students also take a course in imperative programming.

We have 11 students.

Course description

Symbolic logic is fundamental to computer science, providing a foundation for the theory of programming languages, the theory of databases, AI, knowledge representation, automated reasoning, and formal verification. Formal methods based on logic complement statistical methods and machine learning by providing rules of inference and means of representation with precise semantics. These methods are central to hardware and software verification, and have also been used to solve open problems in mathematics.

Course description

This course is an introduction to symbolic logic on three levels: theory, implementation, and application. We will present the underlying mathematical theory, and students will develop the mathematical skills that are needed to design and reason about logical systems in a rigorous way. We will also show students how to represent logical objects in a functional programming language, Lean, and how to implement fundamental logical algorithms. Finally, we will show students how to use contemporary automated reasoning tools, including SAT solvers, SMT solvers, and first-order theorem provers, to solve challenging problems, and we will show students how to use Lean as an interactive theorem prover.

Overview

A notable feature of the course is that there are three parallel strands:

- *Theory:* we explore the syntax and semantics of classical propositional logic and first-order logic
- *Implementation:* we implement basic syntactic operations, semantic evaluation, simple decision procedures
- *Application:* we solve interesting problems with SAT solvers, SMT solvers, and theorem provers.

Also notable: the course is based on Lean 4, which also serves as a front end to CaDiCaL, Z3 (or CVC4 or CVC5), and Vampire.

Mechanics

We are writing the textbook in real time:

- https://avigad.github.io/lamr
- https://avigad.github.io/lamr/logic_and_mechanized_reasoning.pdf

There is a github repository for the course:

- https://github.com/avigad/lamr

It is set up for use with Gitpod.

Most students install Lean and VS Code and use it on their own machines.

There is a course page:

- https://www.cs.cmu.edu/~mheule/15217-f21/

Overview

I will go over the table of contents:

- https://github.com/avigad/lamr

Using Lean 4

We decided to use the Lean 4 programming language and proof assistant.

Drawbacks:

- It is in prerelease.
- The documentation is very limited.
- The library of formal theorems is very limited.

Mitigating factors:

- We didn't come across any serious bugs.
- Homework assignments were similar to in-class examples.
- The course isn't about formal theorem proving.

Using Lean 4

Advantages:

- As a functional programming language, it's great for implementing logical systems. (Note: Lean itself is implemented in Lean.)
- It has imperative features that are good for applications, like coding up problems for a SAT solver.
- It has mechanisms for special purpose syntax, for example, for writing formulas and defining interpretations.
- The editor interface supports interactive exploration.
- We can also use it as a proof assistant.
- Students really like the language and syntax.

The joy of functional programming

The set of propositional formulas is generated inductively as follows:

- Each variable p is a formula.
- $\top$ and $\bot$ are formulas.
- If A is a formula, so is $\neg A$ ("not A").
- If A and B are formulas, so are
  - $A \wedge B$ ("A and B"),
  - $A \vee B$ ("A or B"),
  - $A \to B$ ("A implies B"), and
  - $A \leftrightarrow B$ ("A if and only if B").

The joy of functional programming

```
inductive PropForm
| var    : String → PropForm
| tr     : PropForm
| fls    : PropForm
| neg    : PropForm → PropForm
| conj   : PropForm → PropForm → PropForm
| disj   : PropForm → PropForm → PropForm
| impl   : PropForm → PropForm → PropForm
| biImpl : PropForm → PropForm → PropForm
deriving Inhabited, Repr, DecidableEq
```

The joy of functional programming

Substitution for terms is defined recursively:

$$\sigma\, x = \sigma(x)$$
$$\sigma\, f(t_1, \ldots, t_n) = f(\sigma\, t_1, \ldots, \sigma\, t_n)$$

```
partial def subst (σ : FOAssignment FOTerm) :
    FOTerm → FOTerm
| var x   => σ x
| app f l => app f $ l.map (subst σ)
```

Recursion is also great for normal form transformations.

The joy of functional programming

Semantics $\mathcal{M} \models_\sigma A$ is defined recursively:

- $\mathcal{M} \models_\sigma t = t'$ if and only if $[\![t]\!]_{\mathcal{M},\sigma} = [\![t']\!]_{\mathcal{M},\sigma}$.
- $\mathcal{M} \models_\sigma R(t_0, \ldots, t_{n-1})$ iff $R^{\mathcal{M}}([\![t_0]\!]_{\mathcal{M},\sigma}, \ldots, [\![t_{n-1}]\!]_{\mathcal{M},\sigma})$.
- $\mathcal{M} \models_\sigma \top$ is always true.
- $\mathcal{M} \models_\sigma \bot$ is always false.
- $\mathcal{M} \models_\sigma A \wedge B$ if and only if $\mathcal{M} \models_\sigma A$ and $\mathcal{M} \models_\sigma B$.
- $\mathcal{M} \models_\sigma A \vee B$ if and only if $\mathcal{M} \models_\sigma A$ or $\mathcal{M} \models_\sigma B$.
- $\mathcal{M} \models_\sigma A \to B$ if and only if $\mathcal{M} \not\models_\sigma A$ or $\mathcal{M} \models_\sigma B$.
- $\mathcal{M} \models_\sigma A \leftrightarrow B$ if and only if $\mathcal{M} \models_\sigma A$ and $\mathcal{M} \models_\sigma B$ either both hold or both don't hold.
- $\mathcal{M} \models_\sigma \exists x.\, A$ if and only if for some $a \in |\mathcal{M}|$, $\mathcal{M} \models_{\sigma[x \mapsto a]} A$.
- $\mathcal{M} \models_\sigma \forall x.\, A$ if and only if for every $a \in |\mathcal{M}|$, $\mathcal{M} \models_{\sigma[x \mapsto a]} A$.

The joy of functional programming

```
def FOForm.eval {α} [Inhabited α] [BEq α]
    (M : FOModel α) (σ : FOAssignment α) : FOForm → Bool
| eq t1 t2 => t1.eval M.fn σ == t2.eval M.fn σ
| rel r ts =>  M.rel r (ts.map $ FOTerm.eval M.fn σ)
| tr => true
| fls => false
| neg A => !(eval M σ A)
| conj A B => (eval M σ A) && (eval M σ B)
| disj A B => (eval M σ A) || (eval M σ B)
| impl A B => !(eval M σ A) || (eval M σ B)
| biImpl A B => (!(eval M σ A) || (eval M σ B)) &&
                (!(eval M σ B) || (eval M σ A))
| ex x A => M.univ.any fun val =>
                          eval M (σ.update x val) A
| all x A => M.univ.all fun val =>
                           eval M (σ.update x val) A
```

Imperative features

```
def isPrime (n : Nat) : Bool := do
  if n < 2 then false else
    for i in [2:n] do
      if n % i = 0 then
        return false
      if i * i > n then
        return true
    true

def mulTable : Array (Array Nat) := do
  let mut table := #[]
  for i in [:10] do
    let mut row := #[]
    for j in [:10] do
      row := row.push ((i + 1) * (j + 1))
    table := table.push row
  table
```

Imperative features

```
/-- Encodes the given Sudoku as CNF. -/
def cnfEncode : Sudoku → CnfForm
  | s@{ dim, rows : Sudoku } => do
    let mut cnf : CnfForm := []
    let sz := dim*dim

    -- Each cell contains at least one number
    for i in [:sz] do
      for j in [:sz] do
        cnf := (List.range sz).map (mkLit i j ·) :: cnf

    -- Each number appears at most once in each row
    for i in [:sz] do
      for j₁ in [:sz] do
        for j₂ in [:j₁] do
          for k in [:sz] do
            cnf := [-(mkLit i j₁ k),
                    -(mkLit i j₂ k)] :: cnf
    ...
```

Syntax

```
#check prop!{p ∧ q → (r ∨ ¬ p) → q}

/- impl (conj (var "p") (var "q"))
        (impl (disj (var "r") (neg (var "p")))
              (var "q")) : PropForm -/

#check propassign!{p, -q, r}

#eval prop!{p ∧ q → (r ∨ ¬ p) → q}.eval
    propassign!{p, -q, r}

#check cnf!{p, -p q -r, -p q}

#eval prop!{(p1 ∧ p2) ∨ (q1 ∧ q2)}.toCnfForm
```

Syntax

Note: $[\![t[s/x]]\!]_\sigma = [\![t]\!]_{\sigma[x \mapsto [\![s]\!]_\sigma]}$.

```
def arith_ex1 := term!{ plus(times(%x, two),
                             plus(%y, three)) }
def arith_ex2 := term!{ plus(one, times(three, %z)) }
def arith_ex3 := term!{ plus(%z, two) }

#eval (arith_ex1.subst
        assign!{ x ↦ arith_ex2, y ↦ arith_ex3 }).eval
        arithFnInterp assign!{z ↦ 7}

#eval arith_ex1.eval arithFnInterp assign!{
    x ↦ (arith_ex2.eval arithFnInterp assign!{z ↦ 7}),
    y ↦ (arith_ex3.eval arithFnInterp assign!{z ↦ 7}) }
```

Lean as a proof assistant

```
example (h : ¬ (p ∨ q)) : ¬ p ∧ ¬ q := by
  apply And.intro
  . intro hp
    exact h (Or.inl hp)
  . intro hq
    exact h (Or.inr hq)

theorem reverse_append :
    reverse (as ++ bs) = reverse bs ++ reverse as := by
  rw [reverse_def, reverseAux_append, reverse_def,
    ←reverseAux_append', nil_append,
      reverse_def]
```

Using Lean 4

Advantages:

- As a functional programming language, it's great for implementing logical systems.
- It has imperative features that are good for applications.
- It has mechanisms for special purpose syntax.
- The editor interface supports interactive exploration.
- We can also use it as a proof assistant.
- Students like the language and syntax.

Overview

Remember, a notable feature of the course is that there are three parallel strands:

- *Theory:* syntax and semantics of propositional logic and first-order logic
- *Implementation:* implement basic syntactic operations, evaluation, simple decision procedures
- *Application:* solve interesting problems with SAT solvers, SMT solvers, and theorem provers.

I'll provide examples of the types of problems we ask students to solve on homework and exams.

Theory

Problem. In class, we described an algorithm to solve the tower of Hanoi problem and proved that n disks can be moved from one peg to another with $2^n - 1$ steps. Prove, as clearly as you can, that this is optimal: it is impossible to move n disks from one peg to another with a smaller number of steps.

Theory

Problem. Remember the recursive definition of the greatest common divisor function:

$$gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ gcd(y, mod(x, y)) & \text{otherwise} \end{cases}$$

Notice that the easiest way to show that the recursion is well founded is to notice that the second argument decreases with each recursive call.

Show that for every nonnegative x and y, there are integers a and b such that $gcd(x, y) = ax + by$. You can use the fact that for nonzero y, we have $x = div(x, y) \cdot y + mod(x, y)$, where $div(x, y)$ denotes integer division.

Theory

Problem. Use the definition of "A is a subformula of B" in Section 4.1 to prove that if A, B, and C are any propositional formulas, A is a subformula of B, and B is a subformula of C, then A is a subformula of C.

Problem. Prove that for any A, B, p, and τ,

$$[\![A[B/p]]\!]_\tau = [\![A]\!]_{\tau[p \mapsto [\![B]\!]_\tau]}.$$

In other words, we can evaluate $A[B/p]$ at τ by evaluating B at τ, and then evaluating A with p replaced by that result.

Theory

Some of the most challenging problems for students involve just unpacking definitions.

Problem. Prove the following carefully, using the semantic definitions in Section 4.2: let Γ and Γ' be sets of propositional formulas and let A be a propositional formula. If $\Gamma \models A$ and $\Gamma' \supseteq \Gamma$, then $\Gamma' \models A$.

Problem. As we did for propositional logic, we can prove:

$$\mathcal{M} \models_\sigma A[t/x] \quad \text{if and only iff} \quad \mathcal{M} \models_{\sigma[x \mapsto [\![t]\!]_{\mathcal{M},\sigma}]} A.$$

Use this fact and the semantic definitions to show that for every formula A, every model $\mathcal{M}$, every term t, and every assignment σ, we have

$$\mathcal{M} \models_\sigma (\forall x.\ A) \to A[t/x].$$

Theory

Problem. In Section 8.3, we outline a method for extracting a resolution refutation of a set of clauses Γ from a failed search. The method relies on the following claim:

> *If there are a resolution proof of a clause C from Γ such that $[\![C]\!]_{\tau[p \mapsto \top]} = \bot$ and a resolution proof of a clause D from Γ such that $[\![D]\!]_{\tau[p \mapsto \bot]} = \bot$, then there is a resolution proof of a clause E from Γ such that $[\![E]\!]_{\tau} = \bot$.*

Prove this claim.

Implementation

Problem. A natural number n is *perfect* if it is equal to the sum of the divisors less than n. Write a Lean function (with return type Bool) that determines whether a number n is perfect. Use it to find all the perfect numbers less than 1,000.

Other exercises involved labeled binary trees, and another involved binary coefficients.

Implementation

Problem. Write a function in Lean that implements substitution for propositional formulas, and test it on one or two examples.

Problem. In class and in the textbook, we discuss a Lean function `PropForm.eval` that evaluates a propositional formula with respect to a truth assignment. Define a similar function, `CnfForm.eval`, that evaluates a formula in conjunctive normal form. (Do it directly: don't translate it to a propositional formula.)

Implementation

Problem. Write in Lean a predicate `isAutarky` that takes an assignment τ : `PropAssignment` and a CNF formula Γ : `CnfForm` and returns a Boolean that indicates whether τ is an autarky for Γ.

Problem. Write in Lean a function `getPure` that takes a CNF formula Γ : `CnfForm` and returns a `List Lit` of all pure literals in Γ. The function does not need to find all pure literals until fixpoint, only the literals the are pure in Γ.

Implementation

Barwise and Etchemendy's *Tarski's World* is a valuable tool for helping students understand to read and write first-order statements.

I'll show you our little implementation.

We also gave students a partial implementation of Fourier–Motzkin and asked them to fill in the rest.

I'll show you that as well.

Applications

We show students how to use SAT solvers to solve Sudoku puzzles and graph coloring problems.

I'll show you how that looks in Lean.

We'll take a look at Assignment 6, which asks them to find grid colorings that avoid monochromatic rectangles.

We'll also take a look at Assignment 7, which asks them to solve an instance of the NumberMind game.

Applications

We show students how to use SMT solvers to find magic squares.

I'll show you how that looks in Lean.

We also show them how to confirm the correctness of a procedure in *Hacker's Delight*.

We'll take a look at Assignment 11, which has them packing almost squares into an almost square.

Applications

Finally, we show students how to write problems in Lean and send them to Vampire.

I will show you the Aunt Agatha problem and a Smullyan asylum problem.

We told students to verify Smullyan's conclusion about another asylum problem for homework, but then Vampire told us that the hypotheses are inconsistent. There is no asylum that satisfies them.

Alexander Bentkamp followed up with a pen-and-paper proof of this, and so did Seulkee Baek.

Conclusions

Lean is a great platform for teaching students about logic and its applications.

- It's functional programming language with imperative features.
- Customizable syntax is helpful for examples.
- The VS Code interface and continuous compilation provide instant feedback, and support experimentation and exploration.
- The fact that Lean, in and of itself, is a logical framework is an added bonus.

Conclusions

The class has been very enjoyable to teach.

- Theory is most meaningful when it connects to applications.
- It becomes clear that theory is *needed* for the implementations and applications.
- It is very satisfying to code things up and see instant results.
- Cool applications, even toy examples, are a strong motivation.

The materials are publicly available.

We'll keep working on them.

Feedback is welcome.