Lecture 8: Software Verification, Automation, and LLMs in Lean

Stefano M. Nicoletti

1 Lecture 8: Software Verification, Automation, and LLMs in Lean

1.1 Purpose of the lecture

We begin with commands that suggest a proof or solve a goal automatically. After trying them on a few examples, we use them to verify a list-reversal algorithm.

In the previous lecture, we briefly introduced lists as an inductive type. We return to that definition to understand the parameter `α` and define append and reverse. We prove by induction that reversing twice returns the original list. After the list proofs, we use `decide` and `omega`, then ask an LLM for help and check its proposed code with Lean.

1.2 Further forms of automation

The `?` variants search for a proof and display a suggested command in the Infoview. When they close the goal, we can replace them with that command.

We want `a < c` from the assumptions `a < b` and `b < c`. `exact?` searches the library for an applicable result and tries to supply its arguments.

```
example (a b c : Nat) (hAB : a < b) (hBC : b < c) : a < c := by
  exact?
```

We want to build a conjunction from two assumptions and an implication. `apply?` searches for a result to apply and tries to prove its premises. It closes this goal, though in other cases it may leave premises for us to prove.

```
example (P Q R : Prop) (hP : P) (hQ : Q) (hQR : Q → R) : P ∧ R := by
  apply?
```

We want `P ∧ (Q ∨ R)`, knowing that `P` and `Q` hold. `simp?` uses these assumptions to simplify the connectives and suggests sufficient rules.

```
example (P Q R : Prop) (hP : P) (hQ : Q) : P ∧ (Q ∨ R) := by
  simp? [hP, hQ]
```

We want `Q` from the assumptions `P ∨ Q`, `P → R`, and `¬R`. `grind?` combines these facts. The `P` case gives `R`, contradicting `¬R`, so the `Q` case remains.

```
example (P Q R : Prop) (hPQ : P ∨ Q) (hPR : P → R) (hNotR : ¬R) : Q := by
  grind?
```

Here the suggestion is `grind only`, without a list of lemmas. The assumptions `hPQ`, `hPR`, and `hNotR` are already available in the goal's context. We can justify the conclusion by considering the two cases of `hPQ`: from `P` we obtain a contradiction, while in the `Q` case we have exactly what we want to prove. `grind` finds a proof automatically, but its suggestion does not list all its internal steps. `only` excludes the registered grind theorems, leaving the local assumptions and its reasoning methods available. With `simp only [lemma]`, we instead supplied a simplification rule. Without that lemma, `simp only` does not automatically know the property of our function. It can still perform basic simplifications, but it does not run the same search as `grind`.

The search performed by `exact?` and `apply?` uses declarations in the environment, so imports and Lean versions can change the suggestions. These examples were checked with Lean 4.29.1.

`simp` uses equalities and equivalences to simplify expressions. `grind` also tries to derive new facts from the assumptions and combine them. It can distinguish cases, apply a universal rule to an object, and use equalities: from `a = b`, for example, it can derive `f a = f b`. This last step is called congruence. It also uses solvers, procedures specialized for problems such as arithmetic. This combination of logical reasoning and specialized procedures is inspired by SMT solvers. If the search fails, another strategy may still prove the goal.

1.3 Testing and formal verification

A test runs a program on selected inputs and compares the result with the expected one. It can reveal an error, while passing the test tells us about the inputs examined. In formal verification, we state a property and prove that it holds for every input covered by the statement. We can use both: tests help us check concrete examples, and a proof justifies the general property.

1.4 An inductive type for lists

We define a list without using the library's `List` type. The parameter `α` represents the type of the stored elements. The same definition can therefore be used for lists of natural numbers, strings, or other objects.

```
inductive CustomList α( : Type) where
  | empty
  | cons (head : α) (tail : CustomList α)

open CustomList
```

A `CustomList α` is built in two ways. `empty` contains no elements. `cons x xs` contains the element `x` in front of the list `xs`. The tail `xs` is itself a list and is the smaller input on which recursion and induction can proceed.

For example, in the list containing Ada, Hypatia, and Eva, the head is Ada and the tail contains Hypatia and Eva. Looking at that tail, we find Hypatia as its head and Eva in its tail. After Eva, only the empty list remains. A recursive function on lists repeats its work on these successive tails until it reaches the empty list.

The parameter `α` makes the definition polymorphic: we can choose the element type while keeping the same constructors. `CustomList Natural` contains the natural numbers defined in earlier lectures, while `CustomList String` contains strings.

For example, we can construct a list containing three strings:

```
def exampleList : CustomList String :=
  .cons "Ada" (.cons "Hypatia" (.cons "Eva" .empty))
```

1.5 Append and reverse

To reverse a list, we need to place an element at the end of a result that has already been constructed. We first define append. The function examines the first list. If it is empty, append returns the second list. In the `cons` case, it retains the head and recursively appends the tail.

```
def append α{ : Type} (xs ys : CustomList α) : CustomList α :=
  match xs with
  | .empty => ys
  | .cons x xs => .cons x (append xs ys)
```

The braces in `{α : Type}` make `α` an implicit argument. When we write `append xs ys`, Lean infers the element type from the types of `xs` and `ys`. We therefore do not pass `α` manually at each call.

Computation follows the first list. If `xs` is `empty`, there are no elements to preserve and the result is `ys`. In the `cons x xs` branch, the name `xs` refers to the tail of the list being examined. The call `append xs ys` appends this tail to `ys`, then `cons x` puts the head back in front of the result. The function thus calls itself, but on a list with one fewer element. Using square brackets as illustrative notation:

```
append [zero, one] [two]
= zero :: append [one] [two]
= zero :: (one :: append [] [two])
= zero :: (one :: [two])
= [zero, one, two].
```

Here `[]` represents `empty`, and `x :: xs` represents `cons x xs`.

The `reverse` algorithm follows the same inductive structure. The empty list remains empty. In the `cons x xs` case, it reverses the tail `xs` and places `x` at the end using `append`.

```
def reverse α{ : Type} (xs : CustomList α) : CustomList α :=
  match xs with
  | .empty => .empty
  | .cons x xs => append (reverse xs) (.cons x .empty)
```

`reverse` is also polymorphic. In the `cons x xs` case, the recursive call first reverses the tail. The list containing only `x` is then appended at the end. To understand this step, take `[zero, one]`: its head is `zero` and its tail is `[one]`. We reverse `[one]`, obtaining `[one]` again, then add `zero` at the end:

```
reverse [zero, one]
= append (reverse [one]) [zero]
= append (append (reverse []) [one]) [zero]
= append (append [] [one]) [zero]
= append [one] [zero]
= [one, zero].
```

To compute `reverse [zero, one]`, we first need the result of `reverse [one]`, which in turn needs `reverse []`. Recursion stops here: the definition already specifies the result for the empty list. We can then complete the append operations, first obtaining `[one]` and then `[one, zero]`.

Each recursive call receives the tail, which contains one fewer `cons` constructor. Because a list has finitely many elements, we reach the `empty` case.

1.6 Stating a correctness property

One property of the algorithm is that reversing twice returns the original list:

```
lecture08_reverse_reverse {α : Type} (xs : CustomList α) :
  reverse (reverse xs) = xs
```

The statement must hold for every type `α` and every list `xs : CustomList α`. It expresses a property of reversal. Later we will see why this property alone is not enough to specify the algorithm's behaviour.

In theorem declarations too, `{α : Type}` introduces an implicit parameter. The theorem holds for every element type. When we use it with a list `xs`, Lean infers `α` from the list's type: for `xs : CustomList String`, it chooses `String`. The braces mean that we can normally omit this argument. With `(xs : CustomList α)`, we instead declare an explicit argument, the list to which we apply the theorem.

To prove this property, we first establish results about `append` and about the interaction between `reverse` and `append`. We then use those results in the final theorem.

1.7 Properties of append

Appending the empty list on the right leaves a list unchanged. Because `append` examines its first argument, we prove the result by induction on `xs`.

```
theorem lecture08_append_empty_right
    α{ : Type} (xs : CustomList α) :
    append xs .empty = xs := by
  induction xs with
  | empty =>
      rfl
  | cons x xs ih =>
      simp [append, ih]
```

In the `empty` case, `append empty empty` reduces directly to `empty`. In the `cons x xs` case, the definition retains `x` and appends the empty list to the tail. The induction hypothesis tells us precisely that `append xs empty = xs`. Using this equality gives the same list, `cons x xs`, on both sides. The command `simp [append, ih]` lets Lean use the definition and the induction hypothesis to close the goal.

This equation removes an append with the empty list on the right and produces a simpler form. We can add it to the local simp-set:

```
attribute [local simp] lecture08_append_empty_right
```

We also need associativity of append. We induct on `xs`: its shape determines `append xs ys`, which in turn allows the outer append on the left to proceed.

```
theorem lecture08_append_assoc
    α{ : Type} (xs ys zs : CustomList α) :
    append (append xs ys) zs = append xs (append ys zs) := by
  induction xs with
  | empty =>
      rfl
  | cons x xs ih =>
      simp [append, ih]
```

In the `cons` case, both sides keep the same head `x`. The induction hypothesis gives the equality between the appended tails.

We do not add associativity to the simp-set. It changes grouping but does not identify a generally simpler form. We name it only in commands that require it.

1.8 Reversing an appended list

When two appended lists are reversed, the two parts occur in the opposite order. We use this lemma to prove the final property.

```
attribute [local simp] lecture08_append_empty_right

theorem lecture08_reverse_append
    α{ : Type} (xs ys : CustomList α) :
    reverse (append xs ys) = append (reverse ys) (reverse xs) := by
  induction xs with
  | empty =>
      simp [append, reverse]
  | cons x xs ih =>
      simp [append, reverse, ih, lecture08_append_assoc]
```

In the `empty` case, `simp` automatically uses the local theorem about appending the empty list. In the `cons` case, the induction hypothesis handles the tail, while `lecture08_append_assoc` groups the append operations in the same way on both sides.

The theorem about reversing an appended list is also useful as an automatic rule in the final proof, so we add it to the local simp-set:

```
attribute [local simp] lecture08_reverse_append
```

1.9 Reversing twice

We can now prove the original property. The induction follows the two constructors of `CustomList α`. In the `cons` case, the previous lemma describes the reversal of the append operation produced by the algorithm, and the induction hypothesis simplifies the double reversal of the tail.

```
attribute [local simp] lecture08_append_empty_right lecture08_reverse_append

theorem lecture08_reverse_reverse
    α{ : Type} (xs : CustomList α) :
    reverse (reverse xs) = xs := by
  induction xs with
  | empty =>
      simp [reverse]
  | cons x xs ih =>
      simp [reverse, ih, append]
```

The proof follows the same structure as the programs: it treats the empty list and the case where we add a head to a tail. Induction lets us conclude for every list, even without knowing its elements or length.

A function that always returned its input would also satisfy the double-reversal equation. This property is therefore not a complete specification of `reverse`. Describing the algorithm's behaviour requires further properties, such as the one connecting `reverse` with `append`, or one that states how element positions change.

1.10 Further automation: decide and omega

In these examples we use `Nat` and `List` from the standard library. `[2, 4, 6]` is Lean syntax for a `List Nat`, distinct from our `CustomList`. We want to verify a concrete concatenation and an inequality. `decide` computes the concatenation, compares the lists, and checks that `7 < 10`. Both checks succeed, so Lean closes the goal.

```
example : ([2, 4] ++ [6] : List Nat) = [2, 4, 6] ∧ (7 : Nat) < 10 := by
  decide
```

Eva is at least 18 and Marta is three years older than Eva. We want to prove that Marta is at least 21. We do not fix their ages, but derive the result from these two assumptions. `omega` handles linear arithmetic on `Nat` and `Int` (equalities and inequalities involving sums and multiplication by constants, not products of two variables).

```
example (evaAge martaAge : Nat)
    (hDifference : evaAge + 3 = martaAge) (hAge : 18 ≤ evaAge) :
    21 ≤ martaAge := by
  omega
```

`omega` is not a general solver for products such as `x * y` when both factors are variables.

Unknown variables can block the calculation used by `decide`. For example, in `n < n + 1`, we do not have two concrete numbers to compare. `omega` can reason about these constraints and prove the inequality for an arbitrary natural number.

1.11 Using an LLM as a Lean assistant

An LLM, or *large language model*, is a language model that we can ask for explanations or code in natural language. Here we use it to propose a proof based on the definitions and lemmas in our file. The proposal may contain errors, so we read it and have Lean check it.

1.11.1 Install Codex in VS Code

We use the official Codex extension on Windows, macOS, or Linux. Open the course project in VS Code and check that Lean and the Infoview work. You need a ChatGPT account with access to Codex. Availability and limits depend on the account.

1. Open *Extensions*: `Ctrl+Shift+X` on Windows and Linux, or `Cmd+Shift+X` on macOS.

2. Search for `@id:OpenAI.chatgpt`. Check the name *Codex* and the publisher *OpenAI*. You can also open the official extension page.

3. Select *Install*. If prompted, reload the window.

4. Select the Codex icon in the sidebar. If it is missing, open the *Command Palette* with `Ctrl+Shift+P` on Windows and Linux, or `Cmd+Shift+P` on macOS. Find and run *Codex: Open Codex Sidebar*.

The extension is enough for this activity. You do not need to install the Codex desktop app or CLI as well. On Windows too, you can use it directly in VS Code. Keep the project in the window where Lean already works.

1.11.2 Sign in with ChatGPT

1. In the Codex panel, select *Sign in with ChatGPT*.

2. Complete sign-in with your ChatGPT account in the browser.

3. Return to VS Code. If prompted, allow the browser to reopen the editor.

4. Check that you can start a conversation in the panel.

For current instructions, see OpenAI's official documentation on the IDE extension, sign-in, and the Windows environment.

1.11.3 Start from the Lean file

Save the file and include the statement you want to work on in the conversation, checking the context shown by Codex. A first request could be:

"Explain this statement and the role of its parameters. Use only concepts already present in the file. For now, do not modify files or run commands."

Compare the response with the code. When you request changes, review the differences and check the result with Lean. Codex proposes explanations and code, while Lean checks the proofs.

1.11.4 Ask for a proof

We want to prove that equal reversed lists have equal original lists. The assumption is `h : reverse xs = reverse ys`, and the goal is `xs = ys`.

A request to the assistant can specify both the intended result and constraints on the proof:

"First explain a strategy for this statement. Preserve the definitions, assumptions, and conclusion. Use the existing lemmas. Do not add axioms, `sorry`, or `admit`. Then propose a step-by-step proof and check it with Lean."

Check the proposal in Lean. If there is an error, give the assistant the message and the remaining goal when asking for a correction. We can also ask it to repeat the checks and corrections until Lean accepts the proof, changing only the proof and keeping the theorem statement unchanged. The earlier constraints still apply: do not add axioms, `sorry`, or `admit`.

1.11.5 Proof

Rewrite `xs` as `reverse (reverse xs)`, using the double-reversal theorem backwards. Then use `h` to replace `reverse xs` with `reverse ys`. The goal becomes `reverse (reverse ys) = ys`, which follows from the same theorem.

```
theorem lecture08_reverse_injective α{ : Type} (xs ys : CustomList α)
    (h : reverse xs = reverse ys) : xs = ys := by
  -- Rewrite xs as its double reversal.
  rw ←[ lecture08_reverse_reverse xs]
  -- Use h to replace reverse xs with reverse ys.
  rw [h]
  -- Reversing ys twice returns ys.
  rw [lecture08_reverse_reverse ys]
```

Finish by comparing the statement and definitions with the originals, reading the proof, and checking the file with Lean.

`#print axioms lecture08_reverse_injective` displays its axiomatic dependencies. For this proof, the course environment reports `propext`, the standard axiom identifying equivalent propositions. It does not report `sorryAx`, which would indicate an incomplete proof.

The dependency on `propext` comes from the earlier lemmas proved with `simp`. For example, the simplifier uses `eq_self` to replace the proposition `t = t` with `True`. Treating these logically equivalent propositions as equal uses `propext`. The final theorem inherits this dependency through `lecture08_reverse_reverse`, even though its proof contains only rewrites. The list concerns the written proof: another proof of the same statement might use different principles.

The kernel is the part of Lean that checks the resulting proof, including proofs obtained through tactics or an assistant. It verifies that the proof establishes the written statement. We must check that this statement expresses the intended property.

1.12 Sources

- Avigad, de Moura, Kong and Ullrich, Theorem Proving in Lean 4, Axioms and Computation: https://lean-lang.org/theorem_proving_in_lean4/Axioms-and-Computation/.
- Lean project, *The Lean Language Reference*, "Tactic Reference": https://lean-lang.org/doc/reference/latest/Tactic-Proofs/Tactic-Reference/.
- Lean project, *The Lean Language Reference*, "The grind tactic": https://lean-lang.org/doc/reference/latest/The--grind--tactic/.
- Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich, *Theorem Proving in Lean 4*, "Inductive Types": https://lean-lang.org/theorem_proving_in_lean4/Inductive-Types/.
- Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich, *Theorem Proving in Lean 4*, "Induction and Recursion": https://lean-lang.org/theorem_proving_in_lean4/Induction-and-Recursion/.
- Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich, *Theorem Proving in Lean 4*, "Tactics": https://lean-lang.org/theorem_proving_in_lean4/Tactics/.
- David Thrane Christiansen, *Functional Programming in Lean*, "Datatypes and Patterns": https://lean-lang.org/functional_programming_in_lean/Getting-to-Know-Lean/Datatypes-and-Patterns/.
- Lean project, *The Lean Language Reference*, "Recursive Definitions": https://lean-lang.org/doc/reference/latest/Definitions/Recursive-Definitions/.
- Kevin Buzzard and Jon Eugster, *Natural Number Game*: https://github.com/leanprover-community/NNG4.
- Vrije Universiteit Amsterdam, *Logical Verification*: https://research.vu.nl/en/courses/logical-verification-5/.
- Marieke Huisman and Anton J. Wijs, *Concise Guide to Software Verification: From Model Checking to Annotation Checking*: https://research.utwente.nl/en/publications/concise-guide-to-software-verification-from-model-checking-to-ann/.
- Hana Chockler, Joseph Y. Halpern, and Orna Kupferman, *What Causes a System to Satisfy a Specification?*: https://arxiv.org/abs/cs/0312036.
- Lean project, The Lean Language Reference, The Simplifier: https://lean-lang.org/doc/reference/latest/The-Simplifier/.
- Lean project, Lean 4.29.1, Init/Tactics.lean: https://github.com/leanprover/lean4/blob/v4.29.1/src/Init/Tactics.lean.