Lecture 1: What Is an Interactive Theorem Prover?

Stefano M. Nicoletti

1 Lecture 1: What Is an Interactive Theorem Prover?

1.1 Aim of the Lecture

This lecture introduces the general idea of an interactive theorem prover and then presents Lean as one specific system of this kind. The aim is not yet to learn many tactics or to formalize complex examples. The aim is to understand what it means to build a proof interactively, what the user sees during that construction, and why a system such as Lean can be useful for analyzing arguments, programs, mathematical proofs, and scientific models.

The lecture has three parts:

- a theoretical part: what is a theorem prover?
- a classroom demonstration;
- a set of exercises.

1.2 What Is an Interactive Theorem Prover?

A theorem prover is a system that helps build and check formal proofs. Some theorem provers are mostly automatic: the user states a problem and the system tries to find a proof. An interactive theorem prover, by contrast, puts the interaction between user and system at the center (Source: Theorem Proving in Lean 4, Propositions and Proofs).

In an interactive system the user writes definitions, statements, and proof steps. The system responds by showing the current state of the proof, or *proof state*, and accepts only steps that respect the formal rules of the system. The complete proof is accepted as correct because it can be checked mechanically: it is a *machine-checkable proof*.

A proof assistant does not replace human judgment. It does not decide by itself what the right formalization of a problem is, nor does it establish whether the premises of an argument are empirically or historically well founded. Its contribution is circumscribed: once statements, definitions, and assumptions have been fixed, it checks whether a given formal construction really proves what it claims to prove.

1.3 Lean

Lean is a specific interactive theorem prover. It is also a programming language. This double nature is central: Lean lets us write computational objects, mathematical structures, logical statements, and proofs in one formal environment (Source: de Moura and Ullrich, Lean 4).

In Lean we can:

- define objects;
- formulate statements;
- build proofs;
- follow the development of a proof interactively;
- formally check the result.

For this first lecture, it is enough to keep one idea fixed: Lean does not check an informal text, but a formal object. If the proof is accepted, the system has checked that the object we constructed has the type required by the statement. But what does this mean?

1.4 Objects and Types

In Lean every object, or *term*, has a type. The type tells us what kind of object we are dealing with and guides what it makes sense to do with it (Source: Theorem Proving in Lean 4, Dependent Type Theory).

For example, objects that represent natural numbers, of type *Nat*, can be used in computational reasoning: we can add, multiply, define recursive functions, or prove arithmetic properties. Objects that represent propositions, of type *Prop*, instead enter reasoning about truth and falsity.

The type of an object therefore informs the way Lean can treat it. If we have a numeric object, Lean expects operations and properties appropriate for numbers. The systematic relation between types, terms, propositions, and proofs will be addressed in the next lectures.

1.5 Interactive Proofs

In an interactive proof we can observe the current state of the proof. This state is often called the *proof state*.

The proof state contains at least two fundamental pieces of information:

- the *goal*, namely what remains to be proved at that point;
- the local *hypotheses and assumptions* available at that point.

At the beginning, the goal coincides with the target of the statement. During the proof it can change: it can be transformed into a simpler goal, split into several goals, or closed when exactly the required object is supplied. A proof is complete when no open goals remain.

Hypotheses are the information we can use at a given moment in the proof. Some come directly from the statement. Others can be introduced during the proof. Some are local, for example inside one branch of a proof by cases.

1.6 Tactics

A *tactic* is a command that transforms the proof state. It is not the final proof in the most fundamental sense, but it is a practical way to build one interactively.

A tactic can, for example:

- introduce a new assumption;
- apply an available rule;
- split a problem into cases;
- produce an intermediate result;
- close a goal when an object of the required type is available.

The pedagogical perspective of the first lecture is this: an interactive proof is a controlled sequence of transformations of the proof state. The user proposes a step; Lean updates the goal and the context, or rejects the step if it is not formally valid.

1.7 Why Use a Theorem Prover?

1.7.1 Argumentation

A theorem prover forces us to distinguish premises, inferential steps, and conclusion. This is useful even when the main interest is not mathematical. In ordinary arguments many premises often remain implicit, or the transition from the premises to the conclusion is left to the competence of the reader.

A formalization makes the structure visible:

- which assumptions are being used;
- which conclusion is being targeted;
- which inferential step connects premises and conclusion.

Of course, this does not solve every problem in argumentation. Lean does not decide whether a premise is acceptable, whether a source is reliable, or whether the formalization really captures the informal argument. But it does check the formal part of the inferential steps.

1.7.2 Software Correctness

A program can compile, pass many tests, and still violate the intended specification. Tests show that some cases work. A formal proof can instead establish that a desired property holds for all possible executions of a given program.

Formal verification is relevant when we want stronger guarantees than tests alone can provide: compilers, protocols, critical systems, algorithms, and software infrastructures may require explicit and checkable properties (Source: VU Amsterdam, Logical Verification).

1.7.3 Mathematics

In mathematics, a formalized proof makes the relation between definitions, lemmas, and conclusion checkable. Every step must be justified relative to explicit rules. This does not remove the value of informal mathematical writing, but it adds a layer of mechanical control.

Lean is especially important because of its mathematical library, `mathlib`, built collaboratively. Once a result has been formalized, it can be reused by others as part of a library of verified mathematics (Source: The mathlib Community, The Lean Mathematical Library).

1.7.4 Physics

In physics, a theorem prover can help make the assumptions of a model explicit, check mathematical or logical steps, and clarify which assumptions belong to a derivation. It does not decide whether a physical model is empirically adequate. But it can help separate the empirical question from the formal one: given a certain model and certain assumptions, what follows?

Physlib is a community project for formalizing results in physics in Lean 4. It is a useful example of how formalization can become collaborative work outside pure mathematics as well (Source: Physlib).

1.8 What Does Lean Check?

Lean checks the correctness of a formal construction relative to a formalized statement. Lean does not decide, however:

- whether an empirical premise is true;
- whether a historical source is reliable;
- whether a physical model is adequate for a phenomenon;
- whether a formalization captures everything we wanted to say;
- whether a proof is pedagogically clear.

Lean checks that, given certain definitions and assumptions, the formal conclusion follows according to the rules of the system. Mechanical checking does not eliminate human judgment, but it moves many errors or oversights from informal reading to the explicit checking of a formal object.

1.9 How Can We Trust Lean?

Trust in Lean depends on how the system is built. The central idea is to reduce the part of the system we have to trust.

Lean can offer complex interfaces, sophisticated tactics, and huge libraries. However, the final result must be checked by a relatively small kernel. The *kernel* is the part of the system that checks whether the constructed proof is correct. For this reason it belongs to the *trusted computing base*: it is the part we have to trust for verification to have value (Source: de Moura and Ullrich, Lean 4).

The principle of the small kernel is pragmatic: if the system contains many complex tools, we want errors in those tools not to be enough, by themselves, to accept a wrong proof.

Trust in Lean is also supported by the open and community-based character of the system. Kernels, libraries, and tools can be publicly discussed, inspected, corrected, and compared with independent implementations or checkers. Projects such as *Lean4Lean* show a further

direction: using formal tools to verify parts of the metalanguage and the formal checker itself (Source: Carneiro, Lean4Lean).

1.10 Argument and Good Argument

We begin our journey into Lean by formalizing some arguments. But first: an argument connects premises and conclusion through an inference (Source: Stanford Encyclopedia of Philosophy, Argument and Argumentation).

- The premises are what we start from.
- The conclusion is what we want to support.
- The inference is the transition from premises to conclusion.

A good argument requires at least three dimensions.

Dimension	Question
Acceptability	Are the premises true, credible, or defensible?
Validity	Does the conclusion follow from the premises?
Soundness/correctness	Are the premises acceptable or true, and is the inference valid?

Validity concerns the form of the inferential transition. An argument can have plausible premises but an invalid inference. Soundness, or correctness, requires both: true premises and a valid inference.

Lean enters mainly at the formal dimension of validity. It can check that a conclusion follows validly from certain formalized premises. It cannot, by itself, establish that those premises are acceptable in the philosophical, historical, scientific, or empirical context in which we use them.

1.11 Ordinary Language and Logical Form

Before formalizing, we need to recognize some recurring forms in ordinary language. In this lecture we restrict ourselves to three fundamental connectives (Source: MIT OpenCourseWare, Logic I).

Ordinary language	Logical form
We have both pieces of information	AND, namely $\wedge$
We have at least one of two alternatives	OR, namely $\vee$
If a condition or hypothesis holds, then a consequence follows	IMPLIES, namely $\to$

1.12 Installing Lean

To work with Lean we will use VS Code with the Lean 4 extension. The standard recommended by the official installation page is (Source: Lean official install page):

- install VS Code;
- install the Lean 4 VS Code extension;
- follow the setup guide opened by the extension;
- open the project folder in VS Code.

The official page remains the operational reference for the current setup.

1.13 Classroom Demonstration

The file used for live coding in the lecture is `Classroom.lean`, which you can find on the reference website. The purpose of the file is to see the construction of the proof while the proof state changes. The examples proceed gradually: first we check types and expressions, then we build elementary proofs, then we use ∧, →, ¬, and ∨.

At the beginning we see some orientation commands:

```
#check Prop
#check Nat
#check Float

#eval 3 + 1
#eval 3.5 + 2
```

`#check` asks Lean for the type of an expression. `#eval` instead asks Lean to evaluate a computational expression. At this point we are not yet proving theorems: we are observing that Lean always works with typed objects.

The first proof exercise is minimal:

```
-- Natural language: if we assume that it rains, then we can conclude that it
   rains.
example (Rains : Prop) (hRains : Rains) : Rains := by
  sorry
```

`sorry` is a placeholder: it tells Lean to temporarily accept a hole in the proof. The complete version is:

```
-- Natural language: if we assume that it rains, then we can conclude that it
   rains.
example (Rains : Prop) (hRains : Rains) : Rains := by
  exact hRains
```

The tactic `exact` closes the goal by providing exactly a term of the required type. The goal is `Rains`; in the context we have `hRains : Rains`; therefore `exact hRains` is enough.

The next step introduces an implication:

```
-- Natural language: if it rains, then it rains.
example (Rains : Prop) : Rains → Rains := by
```

```
  intro h -- intro informativeName, any name that helps us remember the
   hypothesis
  exact h
```

The tactic `intro` is used when the goal is an implication. To prove `Rains → Rains`, we temporarily assume `Rains` and call this hypothesis `h`. At that point the goal becomes `Rains`, which is already available as `h`. So we can close the proof with `exact`.

Now we move to a conjunction:

```
-- Natural language: if it rains then I take the umbrella, and it rains, then
-- this implies that I take the umbrella.
example (Rains TakeUmbrella : Prop) :
    ((Rains → TakeUmbrella) ∧ Rains) → TakeUmbrella := by
  intro hypotheses
  have hRainsUmbrella := hypotheses.left
  have hRains := hypotheses.right
  have hTakeUmbrella := hRainsUmbrella hRains
  exact hTakeUmbrella
```

Here `hypotheses` is a conjunction. The left part is the rule `Rains → TakeUmbrella`; the right part is the proof of `Rains`. The tactic `have` introduces an intermediate result into the context. First we separate the left part of the conjunction with `.left`, placing it in `hRainsUmbrella`; then we do the same with the right part and `hRains`. We then apply `hRainsUmbrella` to `hRains` and obtain `TakeUmbrella`.

The same argument can be written more briefly:

```
-- Shorter proof
example (Rains TakeUmbrella : Prop) :
    ((Rains → TakeUmbrella) ∧ Rains) → TakeUmbrella := by
  intro h
  exact h.left h.right
```

Here we do not give intermediate names. `h.left` is the function `Rains → TakeUmbrella`; `h.right` is the proof of `Rains`; therefore `h.left h.right` is a proof of `TakeUmbrella`.

We can also declare the assumptions directly in the preamble of the example:

```
-- We declare the assumptions explicitly in the preamble of the example.
-- Natural language: assume that, if it rains, then I take the umbrella;
-- assume also that it rains; therefore I take the umbrella.
example (Rains TakeUmbrella : Prop)
    (hRainsUmbrella : Rains → TakeUmbrella) (hRains : Rains) :
    TakeUmbrella := by
  exact hRainsUmbrella hRains
```

In this case there is no need to extract anything from a conjunction: the two assumptions are already separate in the context.

The next step is to give a name to the reasoning schema so that it can be reused in other proofs:

```
-- We encode, verify, and use the valid reasoning schema. Braces mark implicit
-- arguments: Lean can often infer them from the type of the hypotheses.
-- Natural language: if P is true and Q follows from P, then Q is true.
theorem lecture01_modus_ponens
    {P Q : Prop}
    (hP : P) (hPQ : P → Q) :
    Q := by
  exact hPQ hP
```

This is modus ponens. The braces in `{P Q : Prop}` mark implicit arguments: Lean can often infer them from the type of the hypotheses. Once the theorem has been proved, we can use it by applying it to a concrete case:

```
example (Rains TakeUmbrella : Prop)
    (hRainsUmbrella : Rains → TakeUmbrella) (hRains : Rains) :
    TakeUmbrella := by
  exact lecture01_modus_ponens hRains hRainsUmbrella
```

Here `exact` closes the goal with the application of the general theorem to the concrete case.

We then formalize an argumentative chain:

```
-- Natural language: if we have an assumption, if a consequence follows from
-- the assumption, and if a conclusion follows from the consequence, then we
-- have the conclusion.
theorem lecture01_informal_argument_schema
    (Assumption Consequence Conclusion : Prop) :
    (Assumption ∧ (Assumption → Consequence)) ∧
      (Consequence → Conclusion) →
    Conclusion := by
  intro hArgument
  have hFirstStep := hArgument.left
  have hAssumption := hFirstStep.left
  have hStep1 := hFirstStep.right
  have hStep2 := hArgument.right
  have hConsequence := hStep1 hAssumption
  have hConclusion := hStep2 hConsequence
  exact hConclusion
```

Here the structure is nested. First we separate the first block from the second; then we extract the assumption and the first implication; then we build the consequence; finally we apply the second step to the consequence.

Negation in Lean is read as implication to `False`. The following example says that, if drinking wine implies being drunk and we are not drunk, then we did not drink wine:

```
-- Natural language: if I drink wine then I get drunk; but I am not drunk;
-- therefore I did not drink wine.
theorem lecture01_wine_example
    (DrinkWine Drunk : Prop) :
    ((DrinkWine → Drunk) ∧ ¬Drunk) → ¬DrinkWine := by
```

```
  intro hArgument
  have hRule := hArgument.left
  have hNotDrunk := hArgument.right
  intro hDrinkWine
  have hDrunk := hRule hDrinkWine
  exact hNotDrunk hDrunk
```

The final goal `¬DrinkWine` means `DrinkWine → False`. This is why we use `intro hDrinkWine`: we temporarily assume `DrinkWine` and look for a contradiction. The rule produces `Drunk`; the hypothesis `hNotDrunk` turns `Drunk` into `False`.

Another important step is disjunction. If we know `Rains ∨ Snows`, we do not know which of the two sides is true. Therefore we must show that the conclusion follows in both cases:

```
-- Natural language: if it rains or it snows, and in each of the two cases I
-- take the umbrella, then I take the umbrella.
example (Rains Snows TakeUmbrella : Prop) :
    ((Rains → TakeUmbrella) ∧
      (Snows → TakeUmbrella)) ∧
    (Rains ∨ Snows) →
    TakeUmbrella := by
  intro hArgument
  have hRules := hArgument.left
  have hRainsOrSnows := hArgument.right
  have hRainsUmbrella := hRules.left
  have hSnowsUmbrella := hRules.right
  -- We do not know which side of the ∨`` is valid, so we consider both cases.
  cases hRainsOrSnows with
  -- First case: `Rains ∨ Snows` is true because `Rains` holds; prove `
   TakeUmbrella`.
  | inl hRains =>
      exact hRainsUmbrella hRains
  -- Second case: `Rains ∨ Snows` is true because `Snows` holds; prove `
   TakeUmbrella`.
  | inr hSnows =>
      exact hSnowsUmbrella hSnows
```

The tactic `cases` opens one branch for each way in which a disjunction can be true. In the `inl` branch we have a proof of `Rains`; in the `inr` branch we have a proof of `Snows`. In both branches we must produce the same result: `TakeUmbrella`.

The same idea can be written with `Or.elim`. Here we also see the tactic `apply`: instead of immediately providing the complete proof, we apply a general principle that transforms the goal into more specific subgoals.

```
-- The same proof, using `Or.elim` directly.
example (Rains Snows TakeUmbrella : Prop) :
    ((Rains → TakeUmbrella) ∧
      (Snows → TakeUmbrella)) ∧
    (Rains ∨ Snows) →
    TakeUmbrella := by
  intro hArgument
```

```
  have hRules := hArgument.left
  have hRainsOrSnows := hArgument.right
  have hRainsUmbrella := hRules.left
  have hSnowsUmbrella := hRules.right
  apply Or.elim hRainsOrSnows
  -- If the left side of the ∨`` holds, namely `Rains`,
  -- we use the rule `Rains → TakeUmbrella`.
  · intro hRains
    exact hRainsUmbrella hRains
  -- If the right side of the ∨`` holds, namely `Snows`,
  -- we use the rule `Snows → TakeUmbrella`.
  · intro hSnows
    exact hSnowsUmbrella hSnows
```

`Or.elim` says explicitly: to use a proof of `A ∨ B`, it is enough to give a proof of the conclusion `C` from `A` and a proof of the same conclusion `C` from `B`.

The final example combines conjunctions, disjunctions, and a chain of implications:

```
-- Natural language: if the source is authentic or the data are coherent, and
-- each of the two cases supports the thesis, and if a supported thesis makes
-- the conclusion plausible, then the conclusion is plausible.
theorem lecture01_history_of_science_example
    (AuthenticSource CoherentData SupportedThesis PlausibleConclusion : Prop)
   :
    ((AuthenticSource ∨ CoherentData) ∧
      ((AuthenticSource → SupportedThesis) ∧
        (CoherentData → SupportedThesis))) ∧
    (SupportedThesis → PlausibleConclusion) →
    PlausibleConclusion := by
  intro hArgument
  have hFirstStep := hArgument.left
  have hSourceOrData := hFirstStep.left
  have hSupportRules := hFirstStep.right
  have hSourceSupports := hSupportRules.left
  have hDataSupport := hSupportRules.right
  have hSupportConcludes := hArgument.right
  have hThesis :=
    Or.elim hSourceOrData hSourceSupports hDataSupport
  have hConclusion := hSupportConcludes hThesis
  exact hConclusion
```

The disjunction `AuthenticSource ∨ CoherentData` is eliminated with `Or.elim`. The two functions `hSourceSupports` and `hDataSupport` show that, whichever side of the disjunction is valid, we obtain `SupportedThesis`. At that point we apply `hSupportConcludes` and obtain `PlausibleConclusion`.

The main tactics seen in this section are:

- `intro`: introduces a hypothesis when the goal is an implication;
- `exact`: closes the goal by providing a term of the required type;
- `have`: introduces an intermediate result with a name;

- `cases`: splits a disjunction into its cases;
- `apply`: applies a rule or general principle and generates the necessary subgoals.

1.14 Exercises and Solutions

Additional files with more exercises and solutions (`Exercises.lean` and `Solutions.lean`) are available on the reference website.

1.15 Sources and Further Reading

- Lean official install page: link.
- Jeremy Avigad, Leonardo de Moura, Soonho Kong, Sebastian Ullrich, *Theorem Proving in Lean 4*, chapter "Propositions and Proofs": link.
- Jeremy Avigad, Leonardo de Moura, Soonho Kong, Sebastian Ullrich, *Theorem Proving in Lean 4*, chapter "Dependent Type Theory": link.
- Leonardo de Moura and Sebastian Ullrich, "The Lean 4 Theorem Prover and Programming Language": link.
- Mario Carneiro, "Lean4Lean: Towards a Verified Typechecker for Lean, in Lean": link.
- The mathlib Community, "The Lean Mathematical Library": link.
- Physlib, physics in Lean 4: link.
- Stanford Encyclopedia of Philosophy, "Argument and Argumentation": link.
- MIT OpenCourseWare, Logic I: link.
- VU Amsterdam, Logical Verification: link.