Interactive theorem proving and Lean

CMPUT 664 · Formal Verification · October 14, 2026

First-order logic

In first-order logic, quantifiers bind individuals in a domain. Predicate symbols remain fixed.

For example:

$$\forall x.\ P(x) \to Q(x).$$

- x ranges over elements of the domain.
- P and Q are fixed predicate symbols.
- Every element satisfying P also satisfies Q.

Higher-order logic

Higher-order logic allows quantification over **functions and predicates**.

For fixed domains A, B, and C, function composition gives:

$$\forall f : A \to B.\ \ \forall g : B \to C.\ \ \exists h : A \to C.$$
$$\forall x : A.\ \ h(x) = g(f(x)).$$

The variables f, g, and h range over functions; x ranges over elements of A.

A direct proof constructs a function h and shows that it satisfies the equation.

Interactive theorem proving

An **interactive theorem prover** lets us supply steps of an argument, shows what remains to prove, and checks the completed proof.

For composition, we supply the function whose existence the statement asserts.

Our construction, given arbitrary $f : A \to B$ and $g : B \to C$:

$$h(x) := g(f(x)).$$

What the system must check:

$$h : A \to C \qquad \text{and} \qquad \forall x : A.\ h(x) = g(f(x)).$$

Here the equation follows directly from the definition. In a larger proof, we can supply further arguments or use automation to establish the remaining claims.

Representing the construction

To check our proposed h, the prover must verify its type and the equation it satisfies.

We will explain these checks using **typed lambda calculus**, a small language for defining and applying functions.

For the construction $h(x) := g(f(x))$, we need rules that explain:

- how to write a function with parameter x and body $g(f(x))$;
- why its input and output types are A and C;
- how applying it to an input produces the result $g(f(x))$.

Types and terms

Types let a checker verify that function inputs and outputs fit together.

A **term** is an expression. A **type** describes the kind of value the expression may produce. We write $t : T$ to state that the expression t has type T.

$$x : A \qquad f : A \to B$$

Read these as:

- x has type A;
- f takes an input of type A and returns a result of type B.

Typing statements also apply to compound expressions: $f(x) : B$ says that applying f to x produces a result of type B.

Typed lambda calculus

Lambda notation defines a function. A **typing statement** describes its type.

Function expression: $\lambda x.\ x$

The dot separates the parameter from the body. This says “take x, return x”: the identity function.

Parameter annotation: $\lambda x : A.\ x$

The input parameter x must have type A.

Typing statement:

$$(\lambda x : A.\ x)\ :\ A \to A.$$

The inner $: A$ annotates x. The outer $: A \to A$ states the type of the whole function.

Application and evaluation

Applying a lambda expression evaluates its body by substituting the supplied argument for the parameter. This step is called **beta reduction**.

For example, let $a : A$. Applying the identity function gives:

$$(\lambda x : A.\ x)\, a \ \longrightarrow\ a.$$

The parameter x in the body is replaced by a.

We write $f\ a$ for $f\,(a)$; parentheses group compound arguments, as in $g\,(f\ a)$.

Function composition

Composition connects two functions whose intermediate types match. It gives the function $h : A \to C$ needed in our opening claim.

For $f : A \to B$ and $g : B \to C$, the construction is:

$$\lambda x : A.\ g(f\ x)\ :\ A \to C.$$

Follow the input to check the result type:

Step	Result type
Take an input x	A
Apply f to get $f\ x$	B
Apply g to get $g\ (f\ x)$	C

Composition as a higher-order function

Composition is a **higher-order function**: it takes two functions and returns their composed function.

Its type states this interface:

$$(A \to B) \to \Big((B \to C) \to (A \to C)\Big).$$

The expression accepts f, then g, then the input x:

$$\lambda f : A \to B.\ \lambda g : B \to C.\ \lambda x : A.\ g(f\ x).$$

Each lambda introduces one argument. Function arrows associate to the right, so the type can also be written:

$$(A \to B) \to (B \to C) \to (A \to C).$$

Recording a logical argument

To prove an implication $U \to V$, we may temporarily assume U and derive V. This establishes $U \to V$ and discharges the assumption.

For example, consider the composition of two implications:

$$(P \to Q) \to (Q \to R) \to (P \to R).$$

The two outer implications let us assume $P \to Q$ and $Q \to R$. The remaining goal is $P \to R$:

1. Assume P.
2. The first implication gives Q.
3. The second implication gives R.
4. Since the assumption P led to R, conclude $P \to R$.

Step 4 discharges P. Discharging the two outer assumptions then gives the full theorem.

Implications as functions on proofs

A constructive proof of $P \to Q$ transforms any proof of P into a proof of Q. We can represent it as a **function on proofs**.

Write $p : P$ for a proof of P, and $f : P \to Q$ for a proof of the implication. Then:

$$f\ p : Q.$$

- Applying f to p records **modus ponens**.
- Defining a function that takes a proof of P and returns a proof of R records a proof of $P \to R$.

The inputs are proofs, not truth values.

The composition proof term

A **proof term** records both the use of premises and the assumptions under which the argument works.

For composition, the complete term is:

$$\lambda f\,.\,\lambda g.\,\lambda p.\,g(f\ p)\ :\ (P \to Q) \to (Q \to R) \to (P \to R).$$

Read it from the assumptions to the conclusion:

- The lambdas bind $f\ :\ P \to Q, g\ :\ Q \to R$, and $p\ :\ P$.
- $f\ p$ uses the first implication to obtain a proof of Q.
- $g(f\ p)$ uses the second implication to obtain a proof of R.

Checking a proof term

The checker verifies the supplied argument by checking types at each step. It does not have to discover the argument.

For the body $g(f\ p)$:

Type check	**Logical meaning**
f expects P; p has type P	The first implication has its required premise
$f\ p$ has type Q, which g expects	The second implication has its required premise
$g(f\ p)$ has type R	The conclusion follows
Bind p, then g and f	Discharge the assumptions to obtain the theorem

For arbitrary P, Q, and R, the expression $g\ p$ would fail: g needs a proof of Q, while p proves P.

Curry–Howard correspondence

The **Curry–Howard correspondence** relates typing rules to logical proof rules. In the constructive implication fragment, checking a term's type checks its proof.

The correspondence we have just seen is:

Typed expression	**Logical interpretation**
A type	A proposition
A term of that type	A proof of that proposition
Define a function	Prove an implication under an assumption
Apply a function	Use an implication with a proved premise

Goals as missing terms

A **goal** is what remains to prove. For $U \to V$, we can assume U temporarily and make V the new goal.

$$(P \to Q) \to \Big((Q \to R) \to (P \to R)\Big).$$

Temporary assumption introduced	**Remaining goal**
$f : P \to Q$	$(Q \to R) \to (P \to R)$
$g : Q \to R$	$P \to R$
$p : P$	R

$$f : P \to Q, \quad g : Q \to R, \quad p : P \vdash R.$$

Left of $\vdash$ are the assumptions we may use. On the right, R is still to be proved.

The unfinished term is $\lambda f.\ \lambda g.\ \lambda p.\ ?m$, where the missing body $?m$ must have type R.

Tactics automate proof construction

Tactics let us specify a proof step while the prover builds the corresponding term and determines the remaining goals.

With goal R, our choice is "use $g : Q \to R$." The tactic inserts $g\ ?q$ and determines that the missing argument must prove Q.

We can continue by choosing steps for the new goals:

Our instruction	Term constructed so far	Remaining goal
Use g	$g\ ?q$	Q
Use f	$g(f\ ?p)$	P
Use the assumption p	$g(f\ p)$	None

A tactic can also construct many proof steps from one instruction, such as rewriting an expression or searching for a proof.

The same proof in Lean

Lean is a programming language and an interactive proof assistant. It checks a theorem by checking that its proof term has the stated proposition as its type.

We can supply the finished composition term directly:

```
1 theorem compose (P Q R : Prop) :
2     (P → Q) → (Q → R) → (P → R) :=
3   fun f g p => g (f p)
```

`P : Prop` declares a proposition; `p : P` would be a proof of it.

The proposition after `:` is the required type. The expression after `:=` supplies the proof, with `fun` as Lean's notation for lambda abstraction.

The same proof with tactics

Lean executes tactic commands inside a `by` block to construct a proof term.

Our earlier instructions become the script below. Lean builds the same term, `fun f g p => g (f p)`:

```
1 theorem composeBy (P Q R : Prop) :
2     (P → Q) → (Q → R) → (P → R) := by
3   intro f g p
4   apply g
5   apply f
6   exact p
```

`intro` introduces the assumptions. The two `apply` steps change the goal from R to Q, then to P. `exact p` fills the last hole.

Automation constructs a term too

A search tactic can choose proof steps as well as construct the term. Its result is checked just like a term we write ourselves.

For composition, Lean can find the applications automatically:

```
1 theorem composeAuto (P Q R : Prop) :
2     (P → Q) → (Q → R) → (P → R) := by
3   intro f g p
4   solve_by_elim
```

Here `solve_by_elim` finds the chain of applications using the available assumptions, producing `g (f p)` as the body.

If the search fails, the goal remains unproved.

Checking the result of automation

Lean can use complex tactics because its kernel separately checks the proof terms they produce.

Elaboration constructs the term. It fills in omitted types and arguments, and runs tactic commands when we use a tactic proof.

The kernel checks the term. It verifies that the term has the theorem's type, using the rules of Lean's logic and the declared assumptions.

Our direct proof, tactic script, and automated proof all lead to the same check:

$$(\lambda f\,.\,\lambda g.\,\lambda p.\,g(f\ p))\ :\ (P \to Q) \to (Q \to R) \to (P \to R).$$

An ill-typed candidate is rejected, regardless of which tactic produced it.

Proofs about arbitrary values

To prove a universal statement, we give a method that takes any input value and proves the property required at that value.

Extend composition to properties $P(x)$, $Q(x)$, and $R(x)$ of $x : A$. Our goal is:

$$\forall x : A.\ P(x) \to R(x).$$

Assume we have proofs:

$$f : \forall x : A.\ P(x) \to Q(x), \qquad g : \forall x : A.\ Q(x) \to R(x).$$

For any chosen x, the same argument works: assume $P(x)$, use f at x to obtain $Q(x)$, then use g at x to obtain $R(x)$.

Dependent function types

A **dependent function type** allows the result type to depend on the input value. This is what our universal proof requires.

In $(x : A) \to B(x)$, the result type may mention x. An ordinary $A \to B$ has a fixed result type.

For the preceding goal, the proof has type:

$$h : (x : A) \to \Big(P(x) \to R(x)\Big).$$

At input a, the result proves $P(a) \to R(a)$. The term is:

$$\lambda x.\ \lambda p.\ g\ x\ (f\ x\ p).$$

Applying f and g to x specializes their universal proofs to that value. The inner applications perform composition.

Returning to the existence claim

A constructive existence proof supplies a **witness** and a proof that it has the required property. The chosen witness determines which proposition must be proved.

Given $f : A \to B$ and $g : B \to C$, our opening claim was:

$$\exists h : A \to C.\ \forall x : A.\ h(x) = g(f(x)).$$

Its two parts are:

1. Supply the witness $h = \lambda x.\ g(f(x))$.
2. For every x, prove the required equality. Expanding h gives $g(f(x)) = g(f(x))$.

Dependent type theory

Lean's logical foundation is **dependent type theory**: rules for forming and checking types and terms, with types that may depend on values.

The kernel checks proof terms using this theory's rules. Functions and predicates can themselves be arguments, supporting higher-order statements.

The proof structures we have used fit this foundation:

Statement	What its proof supplies
$P \to Q$	A function taking a proof of P to a proof of Q
$\forall x : A.\ R(x)$	A function taking x to a proof of $R(x)$
$\exists x : A.\ R(x)$	A witness a together with a proof of $R(a)$

From a statement to a checked theorem

Types connect the statement we want to prove, the proof we construct, and the check that accepts it.

1. **State the theorem as a type.** Dependent type theory lets the proposition refer to values and quantify over functions.
2. **Construct a term of that type.** We can write it directly, develop it through goals and tactics, or ask automation to find it.
3. **Check the term.** The kernel verifies that it has the theorem's type under the rules and assumptions of the logic.

For our opening theorem, the proof supplies the function $h = \lambda x.\ g(f(x))$ and evidence that it satisfies the required equation.

ATP and ITP

ITP lets us choose the structure of an argument and use automation within it.

- In **ATP**, we supply a goal and premises, and a procedure searches for a proof.
- In **ITP**, we can supply constructions and intermediate lemmas, inspect the resulting goals, and invoke automation on those goals.

In our composition proof, we chose the steps with `apply g` and `apply f`, then let `solve_by_elim` make those choices in the automated version.

For a larger proof, choosing a useful intermediate lemma can make the remaining goals easier to solve. Interaction gives us a way to contribute that structure while retaining automation and checking.