Formalizing Mathematics in Lean

Ruth Plümer & Carlos Martínez

Abstract

Over the next few days we'll work with Lean, a programming language that allow us to prove theorems (that is, a proof assistant). It is in the fascinating intersection of mathematics and computer science.

Firstly, a note on these lecture notes: they'll mostly focus on the theory of dependent types, so you'll see little about Lean in them – but fear not: alongside these, during class we'll be learning actual Lean to apply the theory described in the notes (and will most likely share the Lean code for your later use).

1. Type Systems and Higher-Order Functions

If you have ever played around with a statically typed programming language (so, anything that is not Python), you may have needed to spend a lot of time specifying the *types* of your variables; that is, what sort of things they are. In C-like languages, with their somewhat boring type system, you must have encountered things like:

- `int` for integers with a certain precision;
- `bool` for booleans (i.e. True or False); and then more complex types such as:
- `T[]` for a list/array of elements of type T.
- We could even consider a function such as `int add_one(int n) ...` as implicitly having a type: *a function that takes ints to ints.*

Then, if we go on to cooler languages like Haskell (which happens to have some similarities to Lean), this idea is taken to its logical extreme:

- Of course we have *primitive* types such as `Integer`, `Bool`, `String`, etc.
- But now, the creation of compound types can get very robust: for types U and V, you also have the types $[U]$ (for lists of elements of type U), (U,V) (for pairs whose first component is of type U and the second of type V), and the most powerful:
- $U \to V$ (for functions that map elements of type U to elements of type V), exactly mirroring the notation $f : U \to V$ that you may have seen before.

And when we say that the idea is taken to its logical extreme, we mean it: in such languages, you suddenly can work with *higher-order functions* with types such as `(Integer -> Integer) -> (Integer -> Integer)`, that is, a function that takes in a *function from integers to integers*, and outputs another function of that type (cool, huh?) and you can just as easily construct a list containing functions of a certain type...

2. The Simply Typed Lambda Calculus

Just like mathematicians have long used sets to construct every other mathematical object, we could instead consider an universe where *(almost) everything is a type* (which at this point doesn't really seem too different from a set). We'll start with a simple type system and slowly enrich it. The adventure begins with the *simply typed lambda calculus*, which can be seen as a tiny (but powerful!) programming language of sorts.

Notation. We denote 'x has type T' by writing $x : T$. This is called a *typing judgment*. Furthermore, we write $C \vdash x : T$ to mean that x has type T in the *context* C (which could be a list of typing judgments). Then, we also say that x is an *inhabitant* of type T.

For now, let's suppose we already have some *primitive types* (that is, types that cannot be decomposed into simpler types), such as $\mathbb{Z}$ and Bool (fear not: later on we'll see how even these can be constructed!). We'll consider two sorts of types: *function types* and *product types* (strictly speaking, the simply typed lambda calculus only as function types).

Definition 2.1. (Function types).
Given two types U and V, we can construct the type $U \to V$ consisting of *functions whose input has type U and its output has type V*. We construct an inhabitant of this type through *lambda abstraction*: if $x : U$ and $e : V$ is an expression possibly containing x, $\mathsf{fun}\, x \mapsto e$ is the function whose input is x and its output is e; it has type $U \to V$.

Example 2.2. For any type T, we can construct the identity function $\mathsf{id}_T : T \to T$ by $\mathsf{id} := \mathsf{fun}\, x \mapsto x$.

Example 2.3. Suppose we wanted to make an operator that given a function $f : \mathbb{Z} \to \mathbb{Z}$, it gives us $f \circ f : \mathbb{Z} \to \mathbb{Z}$. How? By defining $\mathsf{double} : (\mathbb{Z} \to \mathbb{Z}) \to (\mathbb{Z} \to \mathbb{Z})$ such that $\mathsf{double} := \mathsf{fun}\, f \mapsto (\mathsf{fun}\, x \mapsto f(f(x)))$! Then, $\mathsf{double}\, f = f \circ f$, as expected.

Definition 2.4. (Pair types).
Given two types U and V, we can construct the type $U \times V$ consisting of *pairs whose first component has type U and its second component has type V*. In particular, if we have $x : U$ and $y : V$, then we can construct $(x, y) : U \times V$. For simplicity, suppose that for each pair of types U, V, we also have *projection functions* $\mathsf{fst}_{U,V} : (U \times V) \to U$ and $\mathsf{snd}_{U,V} : (U \times V) \to V$ that extract the first and second components of a pair, respectively.

How can we make functions that take in more than one argument? Given what we've explained thus far, the most conceptually straightforward way is to use the product types (assuming that we have defined triplets as a pair of a pair and something else, etc.)– so for example, we could define $\mathsf{add} : (\mathbb{R} \times \mathbb{R}) \to \mathbb{R}$ such that $\mathsf{fun}\, p \mapsto (\mathsf{fst}(p) + \mathsf{snd}(p))$. But the cooler way uses a trick called *currying*: We can instead define $\mathsf{add} : \mathbb{R} \to (\mathbb{R} \to \mathbb{R})$ such that $\mathsf{add} := \mathsf{fun}\, x \mapsto (\mathsf{fun}\, y \mapsto x + y)$; this way, $\mathsf{add}(x)(y) = x + y$. The good thing about this approach is that we also get a family of *partially applied* functions for free: for each $x : \mathbb{R}$, $\mathsf{add}(x)$ is the function that adds x to its input.

3. The Curry-Howard Correspondence

Our simple type system is already quite powerful. To see why, let's now make a few interesting observations about elements in our types:

1. If we have elements $a : A$, $b : B$, then we can construct an inhabitant $(a, b) : A \times B$.
2. If, given an element $a : A$, we can construct an element $b : B$, then we can construct an inhabitant $(\mathsf{fun}\, a \mapsto b) : A \to B$.

Now, let's talk about something that looks seemingly unrelated: propositional logic (given that this is not a course on logic, we won't be too detailed – think of *propositions* as mathematical statements without quantifiers).

1. If we have a proof a of proposition A and a proof b of proposition B, then we can construct a proof of $A \wedge B$ (A and B).
2. If, given a proof a of proposition A, we can construct a proof b of proposition B, then we can construct a proof of $A \to B$ (if A, then B).

What this analogy seems to suggest is that in fact, we can think of *propositions as types, and proofs as inhabitants of those types!* This is what we call the Curry-Howard Correspondence (or if you want to be fancier, Curry-Howard Isomorphism – because what we're actually doing is treating these analogous structures as identical!)

It's precisely this little observation that makes Lean possible. And its a very powerful observation: we can alternate between both perspectives at any time to make things more convenient, as we'll see shortly. In summary, what we are saying is this:

1. We have two type constructors: given types A, B, we can construct the types $A \to B$ and $A \times B$.
2. We have a type 'universe' Prop whose inhabitants (also types) will correspond to propositions — whenever $A, B : \mathsf{Prop}$, we write $A \wedge B$ instead of $A \times B$.
3. If we have a proposition $P : \mathsf{Prop}$, we call any inhabitant $h : P$ a *proof* of P.

So we can finally do propositional logic in its full glory, we need to postulate the existence of:

- A type $\mathrm{True} : \mathsf{Prop}$ with at least one inhabitant τ.
- A type $\mathrm{False} : \mathsf{Prop}$ with *no inhabitants*, and for each proposition P, a function of type $\mathrm{False} \to P$. If we somehow obtain an inhabitant of False, we have a contradiction, and we allow ourselves to then obtain a proof of *any* proposition!

What about negation? We can actually define $\neg P$ as $P \to \mathrm{False}$ and it will work well for us (think about why).

4. Example: Modus Ponens

Now, let's try to prove some pieces of classical logic with our new machinery: first, everyone's favorite *modus ponens*: $((P \to Q) \wedge P) \to Q$.

To prove something, we must find an inhabitant. We can proceed as follows:

1. It has a function type, so it must be of the form $\mathsf{fun}\, h : (P \to Q) \wedge P \mapsto \_ : Q$. That is, given a proof h of $(P \to Q) \wedge P$, we must construct a proof of Q (the placeholder).
2. We can obtain proofs of $P \to Q$ and P by applying the previously mentioned functions $\mathsf{fst}_{P,Q}$ and $\mathsf{snd}_{P,Q}$ (we can do this since $A \wedge B$ stands for $A \times B$ in this context). That is, $\mathsf{fst}_{P,Q}(h) : P \to Q$ and $\mathsf{snd}_{P,Q}(h) : P$.
3. Then, finally, we have that $(\mathsf{fst}_{P,Q}(h))(\mathsf{snd}_{P,Q}(h)) : Q$, which we can use to fill the placeholder!
4. So we finally have a proof of modus ponens: $\mathsf{fun}\, h : (P \to Q) \wedge P \mapsto (\mathsf{fst}_{P,Q}(h))(\mathsf{snd}_{P,Q}(h))$! Don't worry: Lean has many facilities to make such a proof much more pleasant to write (but this is more or less what it would look like under the hood).

In conclusion: not only are functions useful, they're also fun!

5. Dependent Types and Predicate Logic

Without quantifiers (*for all, there exists, ...*) there's really no way to express interesting mathematical statements. We'll begin by seeing things from the type-theoretical perspective and then see we'll see how the Curry-Howard Correspondence strikes again.

First some motivation: often in mathematics we find it useful to have functions whose *output type* depends on the value of the input. For example, we may want to define a function ZeroVector that takes in a natural number $n : \mathbb{N}$, and it gives you the zero vector $\mathbf{0}$ in $\mathbb{R}^n$. But thus far, there's no way to represent the output type of the ZeroVector that is not a bit clunky or ugly, since function types assume that the output type is fixed (unless we do something like $\bigcup_{n\in\mathbb{N}} \mathbb{R}^n$, but we have no idea what a set or an infinite union is, yet).

It's time to define more powerful type constructors (and by the way, the old ones will just be special cases of these):

Definition 5.1. (Dependent function types, or Π-types).
Given a type universe[1] $\mathcal{U}$, a type $A : \mathcal{U}$ and a family of types[2] $B : A \to \mathcal{U}$, we can construct the type $\prod_{t:A} B(t)$ consisting of *functions whose input t has type A and their output has type $B(t)$*. To construct inhabitants for this type, we use good old lambda abstraction — we just make sure that for each input, the output matches the (dependent) output type.

Example 5.2. Going back to ZeroVector, it is a function whose input n has type $\mathbb{N}$ and its output has type $\mathbb{R}^n$ – so ZeroVector has type $\prod_{n:\mathbb{N}} \mathbb{R}^n$.

Example 5.3. Before, it was a bit clunky to say that for each proposition P, we postulate the existence of a function of type $\text{False} \to P$; now, we can just postulate the existence of one function $\mathsf{exFalso} : \prod_{P:\mathsf{Prop}} (\text{False} \to P)$, from which we can extract all the other functions.

What happens more often, especially when defining mathematical structures, is that we have a tuple (which we can think of as just as a nested pair) where the type of the later components depend on the earlier components. For example, a *magma* is a set (type...) S equipped with a binary operation $\circ : S \times S \to S$; in other words, a pair $(S, \circ)$ where $S : \mathcal{U}$ for some type universe $\mathcal{U}$, and $\circ : S \times S \to S$. But again, we have no straightforward way to express the type of this magma, given that the type of the second component actually depends on the value of the first component. Dependent types to the rescue!

Definition 5.4. (Dependent pair types, or Σ-types).
Given a type universe $\mathcal{U}$, a type $A : \mathcal{U}$ and a family of types $B : A \to \mathcal{U}$, we can construct a type $\sum_{t:A} B(t)$ consisting of *pairs whose first component t has type A and their second component has type $B(t)$*.

Example 5.5. Going back to magmas: it's a pair whose first component S has type $\mathcal{U}$, and its second component has type $S \times S \to S$ (which is a function of S) – therefore, its type would be $\sum_{S:\mathcal{U}} S \times S \to S$.

Exercise 5.1. To extract the components of an inhabitant of a Σ-type, we'll need to postulate the existence of the two functions fst and snd — but fortunately, we no longer need them to be two families of functions, we only need two functions. What would their types be?

[1] That is, a type from which we'll get the types we plan to work with. A good example is Prop!

[2] Sorry – more fancy/confusing notation is yet to come. By *family of types* we mean a function that takes in a parameter and gives you a type.

6. Predicates!

As a refresher on predicates, suppose we have the statement "*for each integer n, there is a negation m*". In usual set-theoretical notation with quantifiers, we would write this as

$$(\forall n \in \mathbb{Z})(\exists m \in \mathbb{Z})(n+m=0)$$

In case you haven't seen quantifiers:

- $(\forall x \in S)P(x)$ stands for "*for all x in S, the statement $P(x)$ is true*". Writing $P(x)$ essentially conveys that it depends on x – so for example, if $P(n)$ stood for $n+1=2$, then $P(1)$ would stand for $1+1=2$.

- $(\exists x \in S)P(x)$ stands for "*there exists an x in S such that the statement $P(x)$ is true*".

So unpacking the previous expression, it would read as "*for all $n \in \mathbb{Z}$, there exists an $m \in \mathbb{Z}$ such that $n+m=0$*".

We can note the following about the quantifiers (and let's use : instead of $\in$ because it objectively looks better):

1. To prove $(\forall x : S)P(x)$, given an arbitrary $x : S$, we must be able to provide a proof of $P(x)$.

2. To prove $(\exists x : S)P(x)$, it suffices to provide $x : S$ and a proof of $P(x)$.

...and, going back to our fancy dependent types:

1. An inhabitant of the type $\prod_{x:S} P(x)$ must have the form $\mathsf{fun}(x : S) \mapsto \_$, where the placeholder has type $P(x)$ – in other words, in order to obtain an inhabitant of this type, given an arbitrary $x : S$, we must be able to provide a term of type $P(x)$.

2. An inhabitant of the type $\sum_{x:S} P(x)$ must have the form (x, h) where $x : S$ and $h : P(x)$, so it suffices to provide x and h.

And as you can see, our amazing dependent types behave identically to the quantifiers (when we're working in Prop, that is)! So when we're working with propositions, Π-types are universal quantifiers and Σ-types are existential quantifiers. Given that $n+m=0 : \mathsf{Prop}$ (once we define the symbols such as $+, =, 0$), in type-theoretical notation the old statement becomes:

$$\prod_{n:\mathbb{Z}} \sum_{m:\mathbb{Z}} (n+m=0)$$

And to prove this statement is just to produce an element of this type.

7. Tactic Cheat Sheet

sorry	Substitutes a proof, throws a warning instead
rw [h]	If h is an assumption of the form of an equation and the LHS appears somewhere in the goal, rw[h] will substitute the first occurrence of the LHS of h by the RHS of h.
rw [← h]	Replaces the first occurrence of the RHS of the assumption h in the goal by the LHS of h.
rw [h1] at h2	Replaces the first occurrence of the LHS of the assumption h1 in h2 by the RHS of h1.
repeat rw [h]	Replaces every occurrence of the LHS of h in the goal by its RHS.
nth_rewrite i [h]	Rewrites with h at the ith occurrence of the LHS of h in the goal.
rfl	Proves an equation of the form $a = a$.
exact	Proves the goal if it is identical to the hypothesis / lambda term / lemma given as an argument.
exact?	Searches for a lemma in Mathlib which can prove the current goal.
assumption	Proves the goal if it is identical to some assumption.
ring	Immediately proves an identity that can be proved by elementary term manipulation (without using assumptions or identities about functions) in the integers, rational numbers or reals (or general rings).
apply h	If h is an assumption of the form $P \to Q$ and the goal is of the form Q, apply h replaces the goal by P.
apply h1 at h2	If h1 is an assumption of the form $P \to Q$ and the assumption h2 is of the form P, apply h1 at h2 changes h2 to Q.
specialize	If h1 is an assumption of the form $P \to Q$ and h2 is an assumption of the form P, specialize h1 h2 changes h1 to Q. If h is a hypothesis of the form $\forall x, A(x)$ and x is a variable of the correct type, specialize h x changes h to $A(x)$.
have h :	Introduces a new assumption h (with proof!)
intro	If the goal is of the form $P \to Q$, intro h introduces an assumption h : P and changes the goal into Q. If the goal is of the form $\forall x, A(x)$, intro x introduces a variable x and changes the goal into $A(x)$.

constructor	If the goal is of the form $P \wedge Q$, constructor introduces two different branches of the proof: one with goal P and one with goal Q.
left/right	If the goal is of the form $P \vee Q$, left or right changes the goal to P or Q, respectively.
obtain	If h is a hypothesis of the form $P \wedge Q$, obtain $\langle$h1, h2$\rangle$:= h yields two assumptions h1 : P and h2 : Q. If h is a hypothesis of the form $P \vee Q$, obtain h1 | h2 := h introduces two different branches of the proof: one with hypothesis h1 : P and one with hypothesis h2 : Q. If h is a hypothesis of the form $\exists x, P(x)$, obtain $\langle$x, hx$\rangle$:= h yields a variable x and a hypothesis hx : P(x).
exfalso	Replaces the goal by False (useful if you suspect that the hypotheses are contradictory).
contradiction	Proves the goal if there are two immediately contradictory assumptions.
by_contra h	Starts a proof by contradiction: Introduces a new hypothesis which is the negation of the current goal and replaces the current goal by False.
by_cases h : P	Starts two branches of the proof: One with an additional assumption h : P and one with an additional assumption h : Q.
push_neg	Pushes negations inside quantifiers and connectives.
ext x	If the goal is of the form $s_1 = s_2$ with s and g sets, ext x replaces the goal by $x \in s_1 \leftrightarrow x \in s_2$.
calc	Creates an environment in which one can proof a chain of equations.
linarith	Proves the goal if it is a linear combination from the assumptions.
trans	When proving an inequality $x \leq y$, transy lets you prove $x \leq y$ and $y \leq z$ instead.
gcongr	Uses monotonicity of functions in order to prove inequalities.
congr	Similar to gcongr, but for equations.
induction	Proves a statement over an inductive type by induction by giving seperate proofs the base case (zero) and the induction step succ with induction hypothesis ih.

References

[1] The Univalent Foundations Program (2013) *Homotopy Type Theory: Univalent Foundations of Mathematics* — http://homotopytypetheory.org/book/

[2] Avigad, J. Massot, P. (2025) *Mathematics in Lean* — https://leanprover-community.github.io/mathematics_in_lean/

[3] Baanen, A. et al. (2025) *The Hitchhiker's Guide to Logical Verification* — https://github.com/lean-forward/logical_verification_2025/blob/main/hitchhikers_guide_2025_desktop.pdf