

Minicourse: Propositions as Types (PAT)

Dhruv and Franklin

July 9, 2022

1 Installing Lean and Agda

Go here to learn how to install Agda: <https://agda.readthedocs.io/en/latest/getting-started/installation.html>

Go here to learn how to install Lean: https://leanprover-community.github.io/get_started.html

2 Types in Lean

The proof assistant called Lean is based on the mathematical system of *type theory*. Every term in type theory has a type, which can be checked in Lean using the `#check` keyword. For instance, running `#check 5` will return `5 : ℕ`, showing that 5 has type \mathbb{N} . We can also run `#check ℕ` which returns `ℕ : Type`, stating that \mathbb{N} itself is a type. In general, you can think of types as *kind of* like sets in an intuitive sense - they have **elements** or **inhabitants**, and if an object a is an element of the type A , then we write $a : A$ to denote this relationship, and we may equivalently say " a has type A " or " a is of type A ".

The Lean library has many types already included, such as types corresponding to the familiar sets $\mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$, as well as other types such as `empty`, an "empty type" with no elements, and `bool`, the "boolean type" with two elements corresponding to "true" and "false".

We can also construct new types from existing types. For instance, if we already have two types A, B defined, then we may construct the **function type** denoted $A \rightarrow B$, which is the type whose inhabitants are the functions with domain A and codomain B . Elements of function types can be constructed using **lambda expressions**, such as the following:

```
def foo := λ (x : ℕ), x + 5
```

This may look cryptic, but `λ (x : ℕ)` essentially means "I'm defining a function that takes an input x of type \mathbb{N} ", and the subsequent `x + 5` says that the value $x + 5$ will be returned. In other words, this defines the function $f : \mathbb{N} \rightarrow \mathbb{N}$ given by $f(x) = x + 5$.

There is something a little funny about how functions of multiple variables are defined in Lean. In mathematics, we often express a function of two arguments, such as addition, as a function from $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ - that is, a function from the set of *ordered pairs* of natural numbers to the set of natural numbers. In Lean (and other functional programming languages in general) we will often prefer to express functions with multiple arguments using a different technique, called **currying**. The type signature of addition in Lean would look like $\mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$ or $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$. (The arrow \rightarrow used to form function types is **right-associated** as a convention.) Rather than a function mapping pairs of natural number to natural numbers, the curried addition function maps natural numbers to functions on the natural numbers. To see how this works, let's define an addition function:

```
def add := λ (x y : ℕ), x + y
```

If we run `#check add`, Lean will tell us that `add : ℕ → ℕ → ℕ`. With the classical addition function, it would not make sense to call the addition function with only one argument (since it only accepts pairs of natural numbers as input), but this *curried* addition function can be **partially applied** to only a single argument. For instance, `add 1` is the function $\mathbb{N} \rightarrow \mathbb{N}$ defined by $f(x) = x + 1$, and

`add 2` is the function $\mathbb{N} \rightarrow \mathbb{N}$ defined by $f(x) = x + 2$, and so on, so that each $n : \mathbb{N}$ is mapped to a different *function*: namely, the function which adds n to its argument. Therefore, an expression like `add 2 3` represents the function `add 2` evaluated at the argument `3`, which is $2 + 3 = 5$. As a matter of fact, it is possible to define functions on pairs of natural numbers in Lean, but you can also write a function which *takes as its argument* a function on pairs from $\mathbb{N} \times \mathbb{N}$, and produces as output a curried function! That is, we can write a function with this type signature:

```
def curry_N : (N × N → N) → (N → N → N)
```

In fact, in general, if A, B are arbitrary types and $A \times A$ is the type of pairs of elements of A , we can write a function converting uncurried functions $A \times A \rightarrow B$ into curried functions $A \rightarrow A \rightarrow B$:

```
def curry : (A × A → B) → (A → A → B)
```

...but we won't go into how to actually write these functions here, or how the type $A \times A$ is defined. If you're interested, this would be a good thing to explore on your own.

There is another special kind of type in Lean called an **inductive type**. The elements of an inductive type can be specified using **constructors**, which enumerate all of the possible ways of constructing an element of that type. For instance, the **unit type** is an inductive type with one element, defined as follows:

```
inductive my_unit
| unit : my_unit
```

We have defined our own special unit type called `my_unit`, whose only inhabitant is given by the constructor `unit`. We know that `unit` is its only element, because `my_unit` does not have any other constructors - there simply isn't *any other way* to produce an element of the type `my_unit`. As another example, we can write an inductive type defining the natural numbers, like this:

```
inductive my_nat
| zero : my_nat
| succ : my_nat → my_nat
```

This inductive definition states that there are two ways of constructing a natural number: either by using `zero`, or by supplying a preexisting natural number as an argument to `succ` to produce a new natural number. Since these are the only two constructors, we can safely say that every natural number of type `my_nat` is either 0 i.e. `zero`, or the successor of some other natural number i.e. `succ n` where $n : \text{my_nat}$. Because every natural number must take one of these two forms, we can define functions case-by-case or piecewise, using a technique called **pattern matching**, like this:

```
def my_nat_double : my_nat → my_nat
| zero      := zero
| (succ n) := succ (succ (my_nat_double n))
```

We can define the above function to double natural numbers by using the fact that doubling 0 results in 0, and doubling the successor of a number produces the successor of the successor of its double. That is, $2 \cdot 0 = 0$ and $2 \cdot (n + 1) = 2 \cdot n + 1 + 1$.

3 Primer on propositional logic

Moving on to a completely unrelated topic - let's talk about **propositional logic** (sometimes also called **sentential logic**). In propositional logic, we use variable names like p, q, r, s to denote sentences/statements which could be true or false, for instance:



- p = "the sky is blue"
- q = "there is a natural number between 0 and 1"
- r = "the sky is blue and there is a natural number between 0 and 1"
- etc.

There are also several logical operations that we can use to combine simpler sentences into more complex ones:

- " **p and q** " is denoted $p \wedge q$, and it is true if both of p, q are true.
- " **p or q** " is denoted $p \vee q$, and it is true if either of p, q is true.
- "**not p** " is denoted $\neg p$, and it is true if p is false.
- " **p implies q** " is denoted $p \rightarrow q$, and it is true unless p is true and q is false. That is, q must be true if p is true, but if p is false, the truthiness of q does not matter.

Using these operations, we can express more complex sentences in terms of simpler ones. For instance, for the example sentences p, q, r listed earlier, we would have $r = p \wedge q$. Some sentences are **tautologies**, or sentences expressed in terms of p, q, r, \dots that are true no matter whether p, q, r, \dots are true or false. For instance, $p \rightarrow p$ and $p \wedge q \rightarrow q \vee p$ are some simple examples of tautologies.

If we want to prove that a complex sentence involving sentential symbols p, q, r, \dots is a tautology, we can make use of several *basic rules of inference* applied in sequence. A **proof**, then, consists of a sequence of sentences, each of which follows from the previous sentences according to the rules of inference. There are two main types of rules of inference: **introduction rules**, which provide a way of proving compound sentences like $\neg p$, $p \wedge q$, $p \vee q$ or $p \rightarrow q$ from simpler ones; and **elimination rules**, which allow us to "break down" compound sentences to obtain simpler ones. For instance, the following inference rules are the only ones we'll need for this write-up (a longer list can be found [here](#)):

1. **Conjunction introduction.** If p has been demonstrated earlier in the proof, and q has been demonstrated earlier in the proof, then you may infer $p \wedge q$.
2. **Conjunction elimination.** If $p \wedge q$ has been demonstrated earlier in the proof, then you may infer p . Also, if $p \wedge q$ has been demonstrated earlier in the proof, then you may infer q .
3. **Conditional introduction.** If you complete a proof of q which assumes p as its hypothesis, then you may infer $p \implies q$.
4. **Conditional elimination aka  Modus Ponens .** If $p \rightarrow q$ has been demonstrated earlier in the proof, and p has been demonstrated earlier in the proof, then you may infer q .

Let's do a couple of example proofs. A simple one would be to prove the tautology

$$p \rightarrow q \rightarrow p$$

or, equivalently,

$$p \rightarrow (q \rightarrow p)$$

because the \rightarrow "implies" operator is implicitly **right-associated**. (Does this remind you of any other right-associated connectives we've seen so far?) There is only one way to introduce a sentence of the form $x \rightarrow y$, and that is to write a proof of y which uses x as its hypothesis, and then use *conditional introduction*. So let's do it! We will start our proof like this:

1 p Assumption

Now, using the assumption p , we must prove $q \rightarrow p$. But once again, we are trying to prove an implication, meaning that we will have to use conditional introduction and start by assuming q .

1 p Assumption

2 q Assumption

We have just assumed q , and we already have that p is true by an earlier assumption. Therefore, we may infer $q \rightarrow p$ by *conditional introduction*:

- 1 p Assumption
- 2 q Assumption
- 3 $q \rightarrow p$ Implication introduction

Finally, we have started with the assumption p and deduced $q \implies p$, meaning that we may use *conditional introduction* once more:

- 1 p Assumption
- 2 q Assumption
- 3 $q \rightarrow p$ Implication introduction
- 4 $p \rightarrow (q \rightarrow p)$ Implication introduction

This completes our proof! Now let's do a simple example involving the logical conjunction \wedge , just to see how it works. Here's a good tautology to try proving:

$$p \wedge (q \wedge r) \rightarrow (p \wedge q) \wedge r$$

First of all, since we are trying to prove an implication, we must start by assuming the hypothesis $p \wedge (q \wedge r)$ and proving the conclusion $(p \wedge q) \wedge r$.

- 1 $p \wedge (q \wedge r)$ Assumption

Next we can use conjunction elimination twice to "break down" the nested conjunction $p \wedge (q \wedge r)$ into p, q, r :

- 1 $p \wedge (q \wedge r)$ Assumption
- 2 p Conjunction Elimination
- 3 $q \wedge r$ Conjunction Elimination
- 4 q Conjunction Elimination
- 5 r Conjunction Elimination

Now that we have p, q and r , we can "build them up" into the desired conjunction $(p \wedge q) \wedge r$ using *conjunction introduction* twice:

- 1 $p \wedge (q \wedge r)$ Assumption
- 2 p Conjunction Elimination
- 3 $q \wedge r$ Conjunction Elimination
- 4 q Conjunction Elimination
- 5 r Conjunction Elimination
- 6 $p \wedge q$ Conjunction Introduction
- 7 $(p \wedge q) \wedge r$ Conjunction Introduction

Finally, since we began a proof with the assumption $p \wedge (q \wedge r)$ and ended with the conclusion $(p \wedge q) \wedge r$, we may deduce the desired implication as before:

1	$p \wedge (q \wedge r)$	Assumption
2	p	Conjunction Elimination
3	$q \wedge r$	Conjunction Elimination
4	q	Conjunction Elimination
5	r	Conjunction Elimination
6	$p \wedge q$	Conjunction Introduction
7	$(p \wedge q) \wedge r$	Conjunction Introduction
8	$p \wedge (q \wedge r) \rightarrow (p \wedge q) \wedge r$	Implication Introduction

During the minicourse, we did another example all together on the chalkboard:

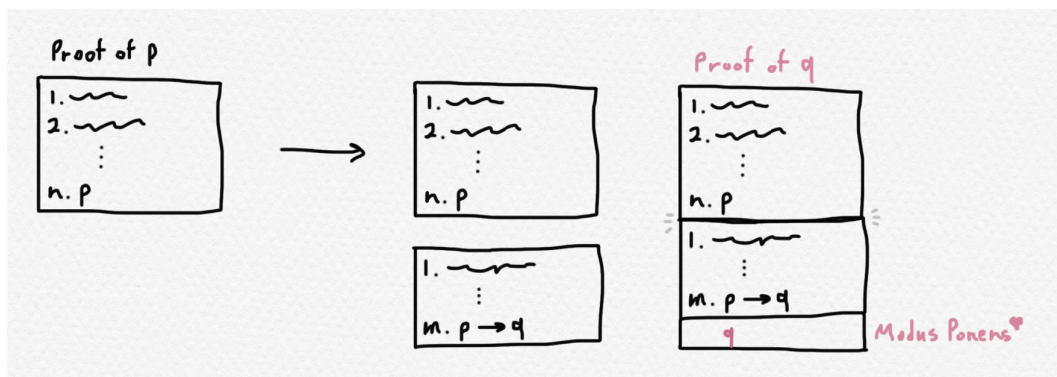
$$(p \wedge q \rightarrow r) \rightarrow (p \rightarrow q \rightarrow r)$$

Our proof looked something like this:

1	$p \wedge q \rightarrow r$	Assumption
2	p	Assumption
3	q	Assumption
4	$p \wedge q$	Conjunction Introduction
5	r	😬 Modus Ponens 😊
6	$q \rightarrow r$	Conditional Introduction
7	$p \rightarrow q \rightarrow r$	Conditional Introduction
8	$(p \wedge q \rightarrow r) \rightarrow (p \rightarrow q \rightarrow r)$	Conditional Introduction

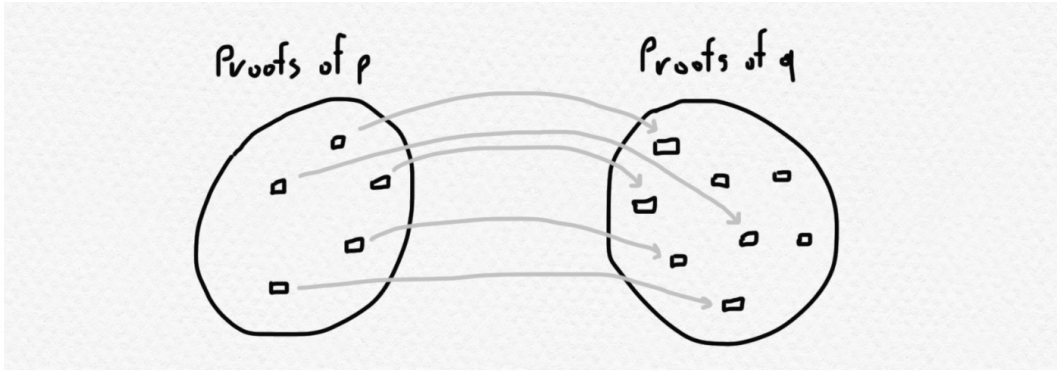
There's a reason we looked at the specific example $(p \wedge q \rightarrow r) \rightarrow (p \rightarrow q \rightarrow r)$ in the minicourse. Do you have any ideas why? Does it remind you of any concepts that were discussed elsewhere?

Before continuing to the next section, a brief conceptual note. Recall that the elimination rule for the implication arrow \rightarrow (what's it called, again?) allows us to prove q given p and $p \rightarrow q$. So if we have a proof of $p \rightarrow q$ on hand, and p is some complex sentence of which we also have a long, multi-line proof, we can *modify* this proof of p to obtain a proof of q by just tacking on the proof of $p \rightarrow q$, and then adding one more line at the end where we infer q by conditional elimination.



Now, there isn't necessarily one single canonical proof of p - there may be many different ways of proving p . But no matter the proof of p that we supply, sticking this proof of $p \rightarrow q$ onto the end of it will produce a proof of q , and it will produce a different proof of q for each possible distinct proof of p that we may supply. If we conceptualized all of the possible proofs of p as being compiled together into some kind of set-like collection, and all the proofs of q into a different collection, then we could

almost think of this proof of $p \rightarrow q$ as a way of transforming each proof of p from the former collection into a proof of q in the latter collection:



Does this remind you of anything?

4 Writing functions in Lean

Back to Lean! In a functional programming language like Lean, it can be a fun puzzle to practice writing functions with certain type signatures. We did a couple of very simple examples earlier using lambda expressions - let's walk through some more now. First of all, we can declare a couple of arbitrary types called $\alpha, \beta, \gamma, \delta$ like this:

```
constants  $\alpha \beta \gamma \delta$  : Type
```

Before diving into some examples, a quick note about how strict Lean can be when it comes to types. Whenever, say, you have a function $f : \alpha \rightarrow \beta$ and you try to evaluate it at some argument x , Lean will verify that $x : \alpha$ before evaluating. If x does not have type α , then Lean will throw a type error when you try to evaluate `f x`. Even if, for example, f has the type signature $f : \mathbb{Q} \rightarrow \mathbb{Q}$ and $x : \mathbb{Z}$, Lean will not allow you to evaluate `f x` because of the type mismatch. In order to do something like this, you would need to manufacture an "inclusion function" $i : \mathbb{Z} \rightarrow \mathbb{Q}$ which converts an integer to a corresponding rational number with the same value, and then evaluate `f (i x)`.

Now let's try to write some functions. For instance, we can try to write a function of the following type signature:

```
def swap : ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow \beta \rightarrow \alpha \rightarrow \gamma$ 
```

Recalling how curried functions work from earlier, there are a couple different ways of interpreting the input/output types of such a function. We could interpret it as taking one argument of type $\alpha \rightarrow \beta \rightarrow \gamma$ and returning a value of type $\beta \rightarrow \alpha \rightarrow \gamma$; or as taking two arguments of types $\alpha \rightarrow \beta \rightarrow \gamma$ and β respectively and returning a value of type $\alpha \rightarrow \gamma$; or as taking three arguments of types $\alpha \rightarrow \beta \rightarrow \gamma$ and β and finally α and then returning an output of type γ . To accept the first input, we can use a lambda expression:

```
def swap : ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow \beta \rightarrow \alpha \rightarrow \gamma$  :=  
 $\lambda f$ , ???
```

After taking the input $f : \alpha \rightarrow \beta \rightarrow \gamma$, our function should return another function of type $\beta \rightarrow \alpha \rightarrow \gamma$. But this function should accept an input of type β , which we could call, say, b :

```
def swap : ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow \beta \rightarrow \alpha \rightarrow \gamma$  :=  
 $\lambda f$ ,  $\lambda b$ , ???
```

This new function, having accepted an input of type β , should produce a *third* function of type $\alpha \rightarrow \gamma$. This function should accept an input of type α , which we will call a :

```
def swap : ( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow \beta \rightarrow \alpha \rightarrow \gamma$  :=  
 $\lambda f$ ,  $\lambda b$ ,  $\lambda a$ , ???
```

Finally, we have a function $f : \alpha \rightarrow \beta \rightarrow \gamma$ at our disposal, as well as elements $a : \alpha$ and $b : \beta$, and are tasked with producing an element of the type γ . How can we accomplish this? We just have to evaluate f with the arguments a and b !

```
def swap : (α → β → γ) → β → α → γ :=
λ f, λ b, λ a, f a b
```

By the way, there's a reason we've decided to call this function `swap`. Do you have any ideas why?

Let's do a couple more examples. Maybe we can figure out how to define a function of the following type signature:

```
def pog : ((α → β) → γ → δ) → γ → β → δ
```

First, this function takes an argument `f` of type $(\alpha \rightarrow \beta) \rightarrow \gamma \rightarrow \delta$:

```
def pog : ((α → β) → γ → δ) → γ → β → δ :=
λ f, ???
```

...then it will produce another function, which takes an argument of type $c : \gamma$...

```
def pog : ((α → β) → γ → δ) → γ → β → δ :=
λ f, λ c, ???
```

...and this will produce another function in turn, which takes an argument of type $b : \beta$...

```
def pog : ((α → β) → γ → δ) → γ → β → δ :=
λ f, λ c, λ b, ???
```

...and now, having accepted a function $f : (\alpha \rightarrow \beta) \rightarrow \gamma \rightarrow \delta$ and elements $c : \gamma$, $b : \beta$ we are tasked with producing something of type δ . Of course, the only thing at our disposal which can possibly produce something of type δ is f , so we will have to call f with some arguments. Therefore, we may as well write f in our lambda expression, and leave blanks for the arguments which we have not yet determined:

```
def pog : ((α → β) → γ → δ) → γ → β → δ :=
λ f, λ c, λ b, f ??? ???
```

The second argument will have to be something of type γ , and we have already gotten our hands on something of this type - namely $c : \gamma$. Thus, we may use

```
def pog : ((α → β) → γ → δ) → γ → β → δ :=
λ f, λ c, λ b, f ??? c
```

The first argument to f will have to be something of type $\alpha \rightarrow \beta$. We don't have any such functions available to us yet, but we *do* have an element of β . Therefore, we are able to define a *constant function* from $\alpha \rightarrow \beta$, namely a function that maps every element of α to the same value $b : \beta$, using a lambda function like this:

```
def pog : ((α → β) → γ → δ) → γ → β → δ :=
λ f, λ c, λ b, f (λ a, b) c
```

Let's look at one more example and try to write a function with the following type signature:

```
def contrapositive : (α → β) → ((β → empty) → (α → empty))
```

Now, this is a strange type signature. What does it mean for a function to have the `empty` type as its codomain? How can a function possibly produce output of the `empty` type, when this type has *no inhabitants at all*? If there actually existed a function $\beta \rightarrow \text{empty}$, the type β would *also have to be empty*, for if there existed an inhabitant $b : \beta$, then $f(b) : \text{empty}$ would be an inhabitant of the empty type, which is impossible.

Nevertheless, we are able to write a function of the above type signature which type-checks in Lean. This function accepts three arguments, which we will call `imp`: $\alpha \rightarrow \beta$, `notβ`: $\beta \rightarrow \text{empty}$, and `a`: α :

```
def contrapositive : (α → β) → ((β → empty) → (α → empty)) :=
λ imp, λ notβ, λ a, ???
```

Now, if $a : A$ and $\text{imp} : \alpha \rightarrow \beta$, then we may evaluate imp at the argument a to obtain an expression $\text{imp } a$ of type β . And if $\text{not} : \beta \rightarrow \text{empty}$, then we may evaluate $\text{not} \beta$ at the argument $\text{imp } a$ to obtain the expression $\text{not} \beta (\text{imp } a)$ of type empty . Hence, we may use the following definition:

```
def contrapositive : ( $\alpha \rightarrow \beta$ )  $\rightarrow$  (( $\beta \rightarrow \text{empty}$ )  $\rightarrow$  ( $\alpha \rightarrow \text{empty}$ )) :=
 $\lambda$  imp,  $\lambda$  not $\beta$ ,  $\lambda$  a, not $\beta$  (imp a)
```

This type-checks in Lean! But wait a second. How could we produce something of type empty , when empty does not contain any inhabitants? Well, as we mentioned earlier, if this function were to be evaluated at its first two arguments, it would be necessary for α to be empty anyways, so that it would be *impossible* to supply the third argument of type α . Thus, the above function *cannot be completely evaluated*, only partially evaluated. How strange! (And why on earth did we decide to call it `contrapositive`?)

5 Propositions as Types

By now, you may have caught on that there is a very nice analogy between proofs in propositional logic and elements of types in Lean. In particular, we can think of types as propositions, where each element of that type is "evidence" of that proposition, or a proof that it is true. In this way, inhabited types all represent true propositions, whereas false propositions can be represented by empty types (for there can be *no proof* for a false proposition).

Here's a table summarizing some of the interesting connections to be made in this analogy. I've left a couple entries in this table blank so that you can think for yourself about how to complete this analogy.

Type Theory	Logic
Type	Proposition
Element of a type	Proof of a proposition
Inhabited type	True proposition
Empty type	False proposition
Function type	Conditional/implication
Argument to a function	Assumption/hypothesis
Output of a function	Conclusion
Function into the empty type	Negation of a proposition
Function evaluation	💖 Modus Ponens 🐱
???	Conjunction ("and")
???	Disjunction ("or")
???	Predicate
???	Universal quantifier (\forall "for all")
???	Existential quantifier (\exists "there exists")
Currying	???