Mathematical project in L∃∀N

An innocent mathematicians guide to lean

Andreas Bøgh Poulsen, studienummer: 201805425

February 21, 2023

Contents

Abstract

Introduction

The following is a project, in which I try to learn how to do formalized mathematics, using Lean as my proof checker. This document is a report on my learnings, and is intended as a resource for other mathematicians, who may wish to learn about Lean.

1 Martin-Löf dependent type theory

Dependent type theory is a logical theory, comparable to first order logic. Similarly to how we usually think we do mathematics in first order logic with ZFC set theory on top, we can translate our mathematical theories into other logical theories. In this chapter I'll give a taste of how dependent type theory works as a formal system. If you're only interested in learning Lean, feel free to skip this section.

1.1 Inference rules

A Inference rule is on the form

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \wedge\text{-intro}$$

which is read as follows: if we, in a context Γ, can prove P and in the same environment can prove Q, then we can prove $P \wedge Q$ in the context Γ.

The defining feature of *t*ype theory is, that every element has a type. Thus the above is meaningless, as P and Q have no type. Compare this to ZFC, where everything is either a proposition from first order logic, or a set. This leads to weird statements like $0 \in 1$, which is well-posed since everything is a set, but does not carry a meaning in our "usual" way of doing mathematics. Type theory asks that every element has a type. This is particularly helpful when doing computerized proofs, as it helps the proof-checker catch weird statements like $0 \in 1$. Since 1 has the type of a natural number and not the type of a set, Lean can give an error, instead of silently trying to prove what may well have been a typo.

In type theory the above rewrite rule would look like this:

$$\frac{\Gamma \vdash P : \mathit{Prop} \qquad \Gamma \vdash Q : \mathit{Prop}}{\Gamma \vdash P \wedge Q : \mathit{Prop}} \wedge\text{-intro}$$

Everything is read the same, except $P : \mathit{Prop}$ is read "P has type Prop". Prop is the type of propositions. I will not spend too much time going through every single single inference rule. I will, however, introduce the defining features of dependent type theory: dependent types, and show how they are used.

1.1 · Definition. Type theory has four different *judgements*.

1. $\Gamma \vdash A$ type says A is a well-formed type in context Γ.
2. $\Gamma \vdash A \doteq B$ type says A and B are judgementally equal types in context Γ.
3. $\Gamma \vdash a : A$ says a is an element of type A in context Γ.
4. $\Gamma \vdash a \doteq b$ type says a and b both have type A and are judgementally equal.

As we would expect, there are axioms making this an equivalence relation:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \doteq a : A} \qquad \frac{\Gamma \vdash a \doteq b : A}{\Gamma \vdash b \doteq a : A} \qquad \frac{\Gamma \vdash a \doteq b : A \qquad \Gamma \vdash b \doteq c : A}{\Gamma \vdash a \doteq c : A}$$

and similarly for types. There is also a rule stating that you can substitute judgementally equal elements anywhere.

Judgemental equality is actually a very strong equality, and many objects we usually consider equal, cannot be proven judgementally equal. Later we'll introduce a weaker equality, that captures better our usual understanding of equality. Stay tuned, the formulation may surprise you.

We need to introduce dependent types as well as functions, before we can get going.

1.2 · Definition. A *dependent type* is a type of the form $\Gamma, x : A \vdash B(x)$ type with a rule letting us assume elements of that type:

$$\frac{\Gamma \vdash A \text{ type}}{\Gamma a : A \vdash a : A}$$

A *section* of a dependent type $B(x)$ is an element $\Gamma, x : A \vdash b(x) : B(x)$.

Note that for different $x : A$ in the context, $B(x)$ may be different type. Using dependent types we can introduce functions:

1.3 · Definition. A *function type* is the type of sections of a dependent type $B(x)$, given by the following introduction rules:

$$\frac{\Gamma, x : A \vdash B(x) \text{ type}}{\Gamma \vdash \Pi_{x:A} B(x) \text{ type}} \qquad \frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma \vdash \lambda x.\, b(x) : \Pi_{x:A} B(x)}$$

and has the following evaluation rules:

$$\frac{\Gamma \vdash f : \Pi_{x:A} B(x)}{\Gamma, x : A \vdash f(x) : B(x)} \qquad \frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma, x : A \vdash (\lambda x.\, b(x))(x) \doteq b(x) : B(x)}$$

Remark Not all types are dependent. If $B(x)$ is independent of x we will just write B and functions as $A \to B$. This arrow binds stronger that Π, so that $\Pi_{a:A} B \to C$ is read as $\Pi_{a:A} (A \to B)$.

1.2 Logic in type theory

We now have the building blocks to start formulating usual logic in dependent type theory. The basic idea is to interpret types as propositions. A proof of a proposition corresponds to an element of a type. Thus a false proposition is a type without any elements, and a true proposition is a true proposition is a type with at least one element. We can introduce canonical false and true propositions:

1.4 · Definition. The types of false and true.

The empty type (false) is given by

$$\frac{}{\vdash \emptyset} \qquad \frac{}{\vdash ind_\emptyset : \Pi_{x:\emptyset} P(x)}$$

and the unit type (true) is given by

$$\frac{}{\vdash \mathbf{1}\ \text{type}} \qquad \frac{}{\vdash \bullet : \mathbf{1}} \qquad \frac{}{\vdash ind_{\mathbf{1}} : P(\bullet) \to \Pi_{x:\mathbf{1}} P(x)}$$

So $\emptyset$ is a false proposition, and $\mathbf{1}$ is a true proposition, with the proof $\bullet : \mathbf{1}$. What would other logical operators look like in this interpretation? Implication simply becomes a function. $f : A \to B$ says "f takes an element of A and produces an element of B" or as propositions "f takes a proof of A and produces a proof of B", which is exactly what an implication does.

In this light, the induction rule of $\emptyset$ states, that given a proof of false, we can prove everything about that element. In particular, $P(x)$ doesn't have to depend on x, så given a proof of false, we can prove anything! The induction principle for $\mathbf{1}$ is comparatively boring, stating that if something is true about $\bullet$, then it's true about every element of $\mathbf{1}$. I other words: if something is true assuming true, and we have a proof of true, that something is true.

We can interpret something being false $\neg A$ as the type $A \to \emptyset$. Then "A is false" translates to "assuming A, I can prove false". We can then prove the statement $(A \implies B) \implies (\neg B \implies \neg A)$. In type theory, this translates to $(A \to B) \to ((B \to \emptyset) \to (A \to \emptyset))$. The construction is as follows:

1.5 · Theorem. $(A \implies B) \implies (\neg B \implies \neg B)$

Proof. We construct an element of the desired type:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash A\ \text{type}}{\Gamma, a : A \vdash a : A} \quad \Gamma \vdash B\ \text{type}}{\Gamma \vdash A \to B\ \text{type}}}{\Gamma, h : A \to B \vdash h : A \to B}}{\Gamma, a : A, h : A \to B \vdash h(a) : B} \quad \dfrac{\dfrac{\dfrac{\Gamma \vdash B\ \text{type}}{\Gamma, b : B \vdash b : B} \quad \dfrac{}{\vdash \emptyset\ \text{type}}}{\Gamma \vdash B \to \emptyset\ \text{type}}}{\Gamma, f : B \to \emptyset \vdash f : B \to \emptyset}}{\Gamma, a : A, f : B \to \emptyset, h : A \to B \vdash f(h(a)) : \emptyset}}{\Gamma, f : B \to \emptyset, h : A \to B \vdash \lambda a.\, f(h(a)) : A \to \emptyset}}{\Gamma, h : A \to B \vdash \lambda f.\, \lambda a.\, f(h(a)) : (B \to \emptyset) \to (A \to \emptyset)}}{\Gamma \vdash \lambda h.\, \lambda f.\, \lambda a.\, f(h(a)) : (A \to B) \to ((B \to \emptyset) \to (A \to \emptyset))}$$

□

You'll note that we didn't use $ind_\emptyset$ in the construtions. Indeed, this is a special case of the more general formula $(A \to B) \to ((B \to C) \to (A \to C))$, which we get simply by composing functions. We'll denote $f \circ g := \lambda x.\, f(g(x))$ and refer to the above proof tree for its construction.

So how do we actually use the induction principle $ind_\emptyset$? Well, we can't prove much right now, but if we introduce *or*:

1.6 · Definition. The type of disjunction

$$\dfrac{\Gamma \vdash A\ \text{type} \quad \Gamma \vdash B\ \text{type}}{\Gamma \vdash A \vee B\ \text{type}} \qquad \dfrac{\Gamma \vdash a : A \quad \Gamma \vdash B\ \text{type}}{\Gamma \vdash \iota_1 : A \to A \vee B} \qquad \dfrac{\Gamma \vdash A\ \text{type} \quad \Gamma \vdash b : B}{\Gamma \vdash \iota_2 : B \to A \vee B}$$

$$\dfrac{\Gamma \vdash A\ \text{type} \quad \Gamma \vdash B\ \text{type}}{\Gamma \vdash ind_\vee : (\Pi_{a:A} P(\iota_1(a))) \to (\Pi_{b:B} P(\iota_2(b))) \to (\Pi_{z:A \vee B} P(z))}$$

$$\dfrac{\Gamma \vdash a : A \quad \Gamma \vdash l : \Pi_{a:A} P(a) \quad \Gamma \vdash r : \Pi_{b:B} P(b)}{\Gamma \vdash ind_\vee(l, r, \iota_1(a)) \doteq l(a) : P(a)}$$

$$\dfrac{\Gamma \vdash b : B \quad \Gamma \vdash l : \Pi_{a:A} P(a) \quad \Gamma \vdash r : \Pi_{b:B} P(b)}{\Gamma \vdash ind_\vee(l, r, \iota_2(b)) \doteq r(b) : P(b)}$$

we can prove the following: $\neg A \to (A \vee B) \to B$.

$$\dfrac{\Gamma \vdash A\ \text{type} \quad \Gamma \vdash B\ \text{type}}{\Gamma \vdash \lambda h.\, \lambda z.\, ind_\vee(ind_\emptyset \circ h, id, z) : (A \to \emptyset) \to (A \vee B) \to B}$$

Okay, that was quite a mouthful. Let's work through the rules for $\vee$ in order.

First, assuming two types A and B, we can form the disjunction $A \vee B$. We have two rules for forming elements of $A \vee B$, namely ι_1 and ι_2 which take an element of A, resp. B and forms an element of $A \vee B$. Next line, we have a way to use a disjunction. Given a proof of P assuming A and a proof of P assuming B, we get proof of P assuming $A \vee B$. The final two lines state, that $ind_\vee$ behaves the way we expect it to.

Using these, the proof if the assertion above becomes

1.7 · Theorem. $\neg A \implies (A \vee B) \implies B$

Proof. We construct an element of the desired type:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash A,B \text{ type}}{\Gamma \vdash A \vee B \text{ type}}}{\Gamma, z : A \vee B \vdash z : A \vee B} \qquad \dfrac{\dfrac{\dfrac{\Gamma \vdash A \text{ type} \quad \dfrac{}{\vdash \emptyset \text{ type}}}{\Gamma \vdash A \to \emptyset \text{ type}}}{\Gamma, h : A \to \emptyset \vdash h} \qquad \dfrac{}{\vdash ind_\emptyset : \ldots}}{\Gamma \vdash ind_\emptyset \circ h : A \to B} \qquad \dfrac{\Gamma \vdash A,B \text{ type}}{\Gamma \vdash ind_\vee : \ldots}}{\Gamma, h : A \to \emptyset, z : A \vee B \vdash ind_\vee(ind_\emptyset \circ h, id, z) : B}}{\Gamma, h : A \to \emptyset \vdash \lambda z.\, ind_\vee(ind_\emptyset \circ h, id, z) : (A \vee B) \to B}}{\Gamma \vdash \lambda h.\, \lambda z.\, ind_\vee(ind_\emptyset \circ h, id, z) : (A \to \emptyset) \to (A \vee B) \to B}$$

□

I have omitted some types to make the tree fit the page, but the crux of the argument is, that from an implication $A \to \emptyset$ and a proof of A, we can use $ind_\emptyset$ to prove B. Thus we derive a function $A \to B$, which we can use, together with $id : B \to B$ to prove B from a $A \vee B$.

Okay, so we have negation, implication and disjunction. I encourage you to imagine how conjunction would be defined. But what about quantors? We'll postpone the existential quantor until later, as it's formulation is quite subtle, but universal quantification is surprisingly straightforward. $\forall x.P(x)$ states that for every x x, we get a proof of $P(x)$. That sounds like a function to me. And indeed, we simply define $\forall := \Pi$. Thus, implication is a non-dependent function, while universal quantification is a dependent function.

This may be surprising, but it actually highlights a strength of dependent type theory as a logical framework: everything, even proofs, is just elements of types. The disjunction, as defined above, is also known as the coproduct in functional programming languages. In the next section, we'll take full advantage of this idea.

1.3 The natural numbers

So far we've only thought about propositions. Let's introduce to natural numbers, as an example of something non-propositional.

1.8 · Definition. The natural numbers

$$\frac{}{\vdash \mathbb{N}\ \text{type}} \qquad \frac{}{\vdash 0_{\mathbb{N}} : \mathbb{N}} \qquad \frac{}{\vdash succ_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}$$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n)\ \text{type} \quad \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \quad \Gamma \vdash p_s : \Pi_{n:\mathbb{N}}(P(n) \to P(succ_{\mathbb{N}}(n)))}{\Gamma \vdash ind_{\mathbb{N}}(p_0, p_s) : \Pi_{n:\mathbb{N}} P(n)}$$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n)\ \text{type} \quad \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \quad \Gamma \vdash p_s : \Pi_{n:\mathbb{N}}(P(n) \to P(succ_{\mathbb{N}}(n)))}{\Gamma \vdash ind_{\mathbb{N}}(p_0, p_s, 0_{\mathbb{N}}) \doteq p_0 : P(0_{\mathbb{N}})}$$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n)\ \text{type} \quad \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \quad \Gamma \vdash p_s : \Pi_{n:\mathbb{N}}(P(n) \to P(succ_{\mathbb{N}}(n)))}{\Gamma \vdash ind_{\mathbb{N}}(p_0, p_s, succ_{\mathbb{N}}(n)) \doteq p_s(n, ind_{\mathbb{N}}(p_0, p_s, n)) : P(succ_{\mathbb{N}}(n))}$$

The first three rules govern the construction of natural numbers, and the next rule is the induction rule. If we for a moment assume P is a predicate, it reads "Given a predicate P, a proof of $P(0)$ and proof of $P(n) \implies P(succ(n))$ we get a proof of $\forall n : P(n)$." The two final rules simply state, that induction behaves as we expect.

All these inference rules are quite heavy. Let's introduce some lighter notation:

$$\begin{aligned} &\mathtt{type} \vdash \mathbb{N} := \\ &\mid\ 0_{\mathbb{N}} : \mathbb{N} \\ &\mid\ \mathtt{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N} \end{aligned}$$

Everything in the inference rules is derivable from this definition. I particular the induction principle becomes

$$ind_{\mathbb{N}} : P(0_{\mathbb{N}}) \to (\Pi_{n:\mathbb{N}}(P(n) \to P(succ_{\mathbb{N}}(n)))) \to \Pi_{n:\mathbb{N}} P(n)$$

We can similarly define

$$\begin{aligned} &\mathtt{type} \vdash \mathrm{A} \vee \mathrm{B} := \\ &\mid\ \iota_1 : \mathrm{A} \to \mathrm{A} \vee \mathrm{B} \\ &\mid\ \iota_2 : \mathrm{B} \to \mathrm{A} \vee \mathrm{B} \end{aligned}$$

We can also observe, that the proof trees so far can be automatically generated, since we every construct so far is introduced by exactly one inference rule. Thus, for the proof of $(A \to \emptyset) \to (A \vee B) \to B$ we'll just write $\lambda h.\, \lambda z.\, ind_{\vee}(ind_{\emptyset} \circ h, id, z)$ as the proof, and omit the proof tree.

Let's define addition and prove some identities. We would like addition to respect

the following specification:

$$add_{\mathbb{N}}(0, n) \doteq n$$
$$add_{\mathbb{N}}(succ_{\mathbb{N}}(m), n) \doteq succ(add_{\mathbb{N}}(m, n))$$

and we would like to do it using the induction rule on $\mathbb{N}$. Remember $ind_{\mathbb{N}}(p_0, p_s)$ has type $\Pi_{n:\mathbb{N}} P(n)$ and addition needs to have type $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$. Thus $P(n)$ needs to be the type $\mathbb{N} \to \mathbb{N}$. The idea is that $ind_{\mathbb{N}}(p_0, p_s, n)$ should produce a function adding n to a number. Then $ind_{\mathbb{N}}(p_0, p_s, n)(m)$ computes $n + m$.

First, let's define $p_0 := id : \mathbb{N} \to \mathbb{N}$. This is a function taking a number and adding 0 to it. Then we need to define $p_s : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$, that is, given a number n and a function adding n to a number, return a function adding $n+1$ to a number. This is simply $p_s(n, f) := succ_{\mathbb{N}} \circ f$. Thus

1.9 · Definition. Addition on the natural numbers

$$add_{\mathbb{N}} := \lambda m.\, \lambda n.\, ind_{\mathbb{N}}(id, \lambda x.\, \lambda f.\, succ_{\mathbb{N}} \circ f, m)(n) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}.$$

We can see that it satisfies our specification:

$$\begin{aligned} add_{\mathbb{N}}(0_{\mathbb{N}}, n) &\doteq (\lambda m.\, \lambda n.\, ind_{\mathbb{N}}(id, \lambda x.\, \lambda f.\, succ_{\mathbb{N}} \circ f, m)(n)(0_{\mathbb{N}}, n) \\ &\doteq ind_{\mathbb{N}}(id, \lambda x.\, \lambda f.\, succ_{\mathbb{N}} \circ f, 0_{\mathbb{N}})(n) \\ &\doteq id(n) \\ &\doteq n \end{aligned}$$

$$\begin{aligned} add_{\mathbb{N}}(succ_{\mathbb{N}}(m), n) &\doteq ind_{\mathbb{N}}(id, \lambda x.\, \lambda f.\, succ_{\mathbb{N}} \circ f, succ_{\mathbb{N}}(m))(n) \\ &\doteq (\lambda x.\, \lambda f.\, succ_{\mathbb{N}} \circ f)(n, ind_{\mathbb{N}}(id, \lambda x.\, \lambda f.\, succ_{\mathbb{N}} \circ f, m))(n) \\ &\doteq (succ_{\mathbb{N}} \circ ind_{\mathbb{N}}(id, \lambda x.\, \lambda f.\, succ_{\mathbb{N}} \circ f), m)(n) \\ &\doteq succ(ind_{\mathbb{N}}(id, \lambda x.\, \lambda f.\, succ_{\mathbb{N}} \circ f), m)(n)) \\ &\doteq succ(add_{\mathbb{N}}(m, n)) \end{aligned}$$

We can check that 1 + 2 = 3:

$$\begin{aligned} &add_{\mathbb{N}}(succ_{\mathbb{N}}(0_{\mathbb{N}}), succ_{\mathbb{N}}(succ_{\mathbb{N}}(0_{\mathbb{N}}))) \\ &\quad \doteq ind_{\mathbb{N}}(id, \lambda x.\, \lambda f.\, succ_{\mathbb{N}} \circ f, succ_{\mathbb{N}}(0_{\mathbb{N}}))(succ_{\mathbb{N}}(succ_{\mathbb{N}}(0_{\mathbb{N}})))) \\ &\quad \doteq (succ_{\mathbb{N}} \circ ind_{\mathbb{N}}(id, \lambda x.\, \lambda f.\, succ_{\mathbb{N}} \circ f, 0_{\mathbb{N}}))(succ_{\mathbb{N}}(succ_{\mathbb{N}}(0_{\mathbb{N}}))) \\ &\quad \doteq (succ_{\mathbb{N}} \circ id)(succ_{\mathbb{N}}(succ_{\mathbb{N}}(0_{\mathbb{N}}))) \\ &\quad \doteq succ_{\mathbb{N}}(succ_{\mathbb{N}}(succ_{\mathbb{N}}(0_{\mathbb{N}}))) \end{aligned}$$

1.4 Equality

We have seen how the induction principle on types can help us to both prove propositions about them (as we did with $\vee$), and define functions on them (as we

did with $\mathbb{N}$). However, there are a couple of notable propsitions about $\mathbb{N}$, which we can't show. Notably, that $\neg(succ_{\mathbb{N}} \doteq 0)$ and that $add_{\mathbb{N}}(n,m) \doteq add_{\mathbb{N}}(m,n)$. The first we can't show, because we have no way of negating a judgement. $A \doteq B$ is not a type, so $\neg(A \doteq B)$ isn't well-formed. The other, we can show for any given n, m, but not in general. This is because we need to prove it by induction, but we can't pass an assumption of $n \doteq m$ along to the induction step, since it isn't a type.

To get past both of these problem, we introduce a type of equality:

1.10 · Definition. The type of equality is given by

$$\begin{array}{l}\texttt{type (a b : A)} \vdash a =_A b := \\ \texttt{| refl : } \Pi_{x:A}\ \texttt{x=}_A\texttt{x}\end{array}$$

with derived induction principle

$$\texttt{ind}_{=_A} : \Pi_{a:A}(P(a) \to \Pi_{b:A}(a =_A b \to P(b)))$$

This states that for any two elements $a, b : A$, we have the type corresponding to the proposition "a equals b". It also states that for any $x : A$, there is an element of type $x =_A x$. Note that we can only compare elements of the same type. The induction principle states: "given $a : A$, a proof/element of $P(a)$, a $b : A$ and a proof of $a =_A b$, we obtain a proof/element of $P(b)$."

It's remarkable that there is no axioms about transitivity or symmetry. These can in fact be derived from the induction principle.

1.11 · Theorem. *Equality is transitive, i.e. there is a function*

$$trans_{=_A} : \Pi_{a,b,c:A}(a =_A b) \to (b =_A c) \to (a =_A c)$$

Proof.

$$trans_{=_A}(a,b,c) := \lambda h_1.\, \lambda h_2.\, ind_{=_A}(b, h_1, c, h_2)$$ □

Short and sweet, although the lack of type annotations makes it a little hard to decipher. It might help if we specialize the type of $ind_{=_A}$. In our case $P(x)$ means $a =_A x$:

$$\begin{array}{lllll} ind_{=_A} : \Pi_{a:A}(P(a) & \to & \Pi_{b:A}\, a =_A b & \to & P(b)) \\ ind_{=_A} : \Pi_{b:A}(a =_A b & \to & \Pi_{c:A}\, b =_A c & \to & (a =_A c)). \end{array}$$

Similarly, symmetry is just

1.12 · Theorem. *Equality is symmetric, i.e. there is a function*

$$symm_{=_A} : \Pi_{a,b:A}(a =_A b) \to (b =_A a)$$

Proof.

$$symm_{=_A}(a,b) := \lambda h.\, ind_{=_A}(a, refl(a), b, h)$$ □

We can even prove that equality is preserved by functions:

1.13 · Theorem. *Function application preserves equality, i.e. there is a function*

$$fun\_eq : \Pi_{a,b:A}\Pi_{f:A\to B}(a =_A b) \to (f(a) =_B f(b))$$

Proof.

$$fun\_eq(a,b,f) := \lambda h.\, ind_{=_B}(a, refl(f(a)), b, h)$$ □

We can use this equality to prove the things I mentioned earlier: $0_{\mathbb{N}} \neq succ_{\mathbb{N}}(n)$ and $add_{\mathbb{N}}(n,m) = add_{\mathbb{N}}(m,n)$. But first, let's clean up our notation even more. We'll omit the subscript indicating type, whenever the type is clear from context. Also, instead of using the *ind* function all the time, we can simply define our functions on each constructor. For example we could define

1.14 · Definition. Addition by pattern matching

```
def add : ℕ → ℕ → ℕ
| 0      , n := n
| succ(m), n := succ_ℕ(add(m, n))
```

and have the specification mechanically translated to functions for $ind_{\mathbb{N}}$. With this, let's prove commutativity. First, we have $add(0,n) \doteq n$ and we get $add(n,0) = n$ by induction:

1.15 · Lemma. $add(n,0) = n \doteq add(0,n)$

Proof.

```
def add_zero : Π_{n:ℕ} add(n,0) = n
| 0      := refl
| succ(n) := fun_eq(add(n, 0), n, succ, add_zero(n))
```

□

Note that the last line proves $add(succ(n),0) = succ(add(n,0)) = succ(n)$, which is exactly the induction step. Thus we have $add(n,0) = add(0,n)$. For the induction step, we need another lemma. We have $add(succ(m),n) \doteq succ(add(m,n))$, and by induction we get:

1.16 · Lemma. $add(m, succ(n)) = succ(add(m,n)) \doteq add(succ(m), n)$

Proof.

```
def succ_add : Π_{m,n:ℕ} add(m,succ(n)) = succ(add(m,n))
| 0       , n := refl ,
| succ(m) , n := fun_eq( add(m, succ(n))
                        , succ(add(m, n))
                        , succ
                        , succ_add)
```

□

Since $add(succ(m), n) \doteq succ(add(m,n))$, this lemma states that $add(succ(m), n) = add(m, succ(n))$. Together, we have

1.17 · Theorem. *Addition is commutative*

Proof.

```
def add_comm : Π_{m,n:ℕ} add(m,n) = add(n,m)
| 0       , n := symm(add(n, 0), add(0, n), add_zero(n))
| succ(m) , n := trans( succ(add(m, n))
                      , succ(add(n, m))
                      , add(n, succ(m))
                      , fun_eq(add(m, n)
                               , add(n, m)
                               , add_comm(m, n))
                      , symm( add(n, succ(m))
                            , succ(add(n, m))
                            , succ_add(n, m) )
                      )
```

□

1.5 Higher order types

I promised to prove $0 \neq succ(n)$, but this is surprisingly hard. At least, it requires a little bit more machinery. Namely, the concept of higher order types, or functions that produces types. We have actually already seen them, $=_A$ is example of a higher order type. We can see $=_A$ as a function $A \to A \to Type$, where *Type* is the type of types. Does that even make sense? We'll talk more about it in the next section, but for now, we'll just assume that every type is itself an element of type *Type*.

This enables us to produce functions such as

```
def nat_equals : ℕ → ℕ → Type
```

```
| 0        , 0        := 1
| 0        , succ(n)  := ∅
| succ(n), 0          := ∅
| succ(n), succ(m)    := nat_equals(n, m)
```

which is exactly what we need. Remember, that $0 \neq succ(n) \doteq \neg(0 = succ(n)) \doteq (0 = succ(n)) \to \emptyset$. Thus we have:

1.18 · Theorem. *0 is the first element of* $\mathbb{N}$*, i.e. we have* $\Pi_{n:\mathbb{N}} \neg(0 = succ(n))$.

Proof. We interpret $\bullet$: nat_equals(0, 0), since nat_equals(0, 0) $\doteq$ **1**. Then, using the assumption $0 = succ(n)$, we rewrite the last 0 in that type, to obtain an element of type nat_equals(0, succ(n)).

```
def zero_ne_succ : Π_{n:ℕ}(0 = succ(n)) → nat_equals(0, succ(n))
| n, h := ind_=(0, •, succ(n), h)
```

And this is our desired function, since nat_equals(0, succ(n)) $\doteq \emptyset$. □

1.6 Propositions as some types and universes

So far, we've assumed that there is no difference between propositions and types. However, this view doesn't quite capture what a proposition is. To illustrate, let's define the existential quantifier, also known as a dependent pair:

1.19 · Definition. A sigma type/dependent pair is the type of pairs $(a, b(a))$, where the second entry is allowed to depend on the first:

```
type P : A → Type ⊢ Σ_{a:A} P(a)
| intro_Σ : Π_{a:A} B(a) → Σ_{a:A} B(a)
```

with derived induction rule:

```
ind_Σ : (Π_{a:A} Π_{x:B(a)} P(intro_Σ(a, x))) → Π_{z:Σ_{a:A} B(a)} P(z)
```

It states that I can prove $\Sigma_{a:A} B(a)$ by exhibiting an element of type A and a proof of $B(a)$. Thus it corresponds to $\exists a : B(a)$. At least in its construction. However, the induction rule is too strong. Indeed, we can construct projection functions:

1.20 · Theorem. *The* Σ *type has projection functions of the following type*

```
p_1 : Σ_{a:A} B(a) → A
p_2 : Π_{z:Σ_{a:A} B(a)} B(p_1(z))
```

Proof. We define the functions as follows:

```
p_1(z) := ind_Σ(λa. λx. a)
p_2(z) := ind_Σ(λa. λx. x)
```

□

The projection functions encode the axiom of choice. Given a proof of $\exists x : P(x)$, we can now pick an element x. This might be okay, if we only care about classical mathematics, but it would be good to have the option, whether or not to assume this axiom, instead of having it forced upon us.

The insight that solves this, is that a proposition doesn't have any "content", it is simply true or false. That is, once we have proved $\exists x : P(x)$, it should forget everything that went into the proof, and just remember the fact, that it is true. In other words, the type of a proposition should either be empty, or contain a single element.

This immediately tells us, that Σ types are not propositions, since it potentially has many different elements. Disjunctions are also not propositions, since $\iota_1(a) \neq \iota_2(b)$. Natural number certainly aren't propositions, which is to expected, so what is a proposition? The types $\emptyset$ and $\mathbf{1}$ are propositions, since they contain respectively 0 and 1 element. Also, if two types A and B are propositions, then the type $A \to B$ is a proposition, and $\Sigma_{a:A} B$ is also a proposition. In this case $\Sigma_{a:A} B$ is a conjunction, "A and B."

We can recover propositionality for existentials and disjunctions, by introducing *universes*. Everything has a type, including types themselves. We used this to introduce nat_equals : $\mathbb{N} \to \mathbb{N} \to \mathit{Type}$. I claimed that $\mathbb{N}$, $\emptyset$ and $\mathbf{1}$ all have type *Type*. What is the type then, of *Type*? Russels paradox still works in type theory, so we can't have $\mathit{Type} : \mathit{Type}$.

1.21 · Definition. A *universe* is a type, which has types as its elements.

In our type theory, we introduce a tower of universes, indexed by the natural numbers. To stay consistent with Lean, we call them *sorts*.

1.22 · Definition. For each $n \in \mathbb{N}$ we have a universe $\mathit{Sort}\ n$. We also have $\mathit{Sort}\ n : \mathit{Sort}(1 + n)$.

We define two special universes

$$\mathit{Prop} := \mathit{Sort}\ 0$$
$$\mathit{Type} := \mathit{Sort}\ 1$$

As an axiom, we have that elements of *Prop* are propositions:

$$\mathit{prop}_\bullet : \Pi_{P:\mathit{Prop}} \Pi_{x,y:P}\, x =_P y$$

Prop is the universe of propositions, and *Type* is the universe of regular types. The crucial this about universes, it that the induction principle of a type in universe

Sort n can only produce types in that same universe. Thus, we can define the existential quantifier:

1.23 · Definition. The existential quantifier is a Σ type in *Prop*:

$$\begin{aligned}&\texttt{type P : A} \to \texttt{Prop} \vdash \exists_{a:A} P(a) \texttt{ : Prop}\\ &\texttt{| intro}_\exists \texttt{ : } \Pi_{a:A} P(a) \to \exists_{a:A} P(a)\end{aligned}$$

with derived induction principle:

$$\texttt{ind}_\exists \texttt{ : } (\Pi_{a:A}\Pi_{x:B(a)} P(\texttt{intro}_\exists(a,x))) \to \Pi_{z:\exists_{a:A} P(a)} P(z)$$

Remark. We now have to annotate our type definitions with the universe they belong to. However, we haven't specified which universe A belongs to. In that case, our definition is *polymorphic* over universes, i.e. the definition applies for any $A : \mathit{Sort}\ n$.

Now, since $P(a)$ is always a *Prop*, we cannot define p_1 like we could for Σ types. However, if we wish to prove a *Prop*, we have access to $a : A$ using the induction principle.

We can define other propositions too:

1.24 · Definition. The order relation on the natural numbers is given by

$$\begin{aligned}&\texttt{type n, m : } \mathbb{N} \vdash n \le m \texttt{ : Prop}\\ &\texttt{| zero\_le : } \Pi_{n:\mathbb{N}}\, 0 \le n\\ &\texttt{| succ\_le : } \Pi_{n,m:\mathbb{N}} (n \le m) \to (succ(n) \le succ(m))\end{aligned}$$

1.25 · Definition. We have the type of a number being even:

```
type n : ℕ ⊢ even(n) : Prop
| zero_even : even(0)
| step_even : Π_{n:ℕ} even(n) → even(succ(succ(n)))
```

We don't have, in general, the law of excluded middle, i.e. $A \vee (A \to \emptyset)$. However, in some cases we can prove it:

1.26 · Definition. A proposition $A : \mathit{Prop}$ for which we have $A \vee \neg A$ is called *decidable*.

1.27 · Theorem. *even(n) is decidable.*

Proof. Similarly to the proof of theorem 1.18, we construct a dependent type

```
def is_even : ℕ → Prop
| 0, h              := 1
| succ(0), h        := ∅
| succ(succ(n)), h := is_even(n, h)
```

```
def ext : Π_{n:ℕ} even(n) → is_even(n)
| 0, h          := •
| succ(0), h :=

def step : Π_{n:ℕ}even(n) → is_even(n) → is_even(succ(succ(n)))
| n, h, b := b

even_dec : Π_{n:ℕ} → even(n)∨¬even(n)
| 0 := ι_1(zero_even)
| succ(0) := ι_2(λh. ind_{even}(0, •, succ(0), h))
| succ(succ(n)) := ind_∨( ι_1(λh. step_even(n, h))
                         , ι_2(λh. )
                         , even_dec(n))
```

2 Mathematics in type theory and Lean

3 Gröbner bases as an extended example