Mathematical project in L∃∀N

An innocent mathematicians guide to lean

Andreas Bøgh Poulsen, studienummer: 201805425

March 30, 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.

We'll build a dependent type theory, which is similar to the one used by Lean. The goal is not to match the calculus of constructions (which is used in Lean), but rather to develop a theory together, to see how and why the choices Lean have made, makes sense. 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:

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

Everything is read the same, except $P : Prop$ 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 judgemental equality 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:

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

When $B(x)$ is independent of x we simply write B. In that case:

$$\frac{\Gamma \vdash B \text{ type}}{\Gamma, b : B \vdash b : B}$$

Every element has a unique type, up to judgemental equality.

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

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 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 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:

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

□

You'll note that we didn't use $ind_\emptyset$ in the constrution. 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: $\neg A \to (A \vee B) \to B$.

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

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:

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

□

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

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

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

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

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

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

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

For a more thorough derivation of the induction principle, see section A in the appendix. 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

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

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 $a =_A x$:

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

Similarly, symmetry is just

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

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

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*

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

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      := id
| succ(m) := succ_ℕ ∘ add(m)
```

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(m, n))
```

□

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. Remember for any $a, b : A$, $a =_A b$ is a type, so 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 to prove $0 \neq succ(n)$. Remember, that $0 \neq succ(n) \doteq \neg(0 = succ(n)) \doteq (0 = succ(n)) \to \emptyset$, so what we need to prove is this:

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

Proof. Given $n : \mathbb{N}$, we want to prove $0 = succ(n) \to \emptyset$, so we assume $0 = succ(n)$ and will try to produce an element of the empty type. To do this, we interpret $\bullet$: nat_equals (0, 0), since nat_equals (0, 0) $\doteq \mathbf{1}$. Then, using the assumption $0 = succ(n)$, we can rewrite the last 0 in that type, to obtain an element of type nat_equals (0, succ(n)), which is judgementally equal to $\emptyset$.

```
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. □

Remark. It's surprising that our types seem to have so much structure, that equality is transitive and the natural numbers satisfy the Peano axioms without us ever mentioning them in the definition. This is a consequence of the derived induction principle, which forces the types to be "free" in some sense. Thus the natural numbers automatically becomes the free monoid on one generator, which is a model of the Peano axioms.

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*

$$p_1 : (\Sigma_{a:A} B(a)) \to A$$
$$p_2 : \Pi_{z:\Sigma_{a:A} B(a)} B(p_1(z))$$

Proof. We define the functions as follows:

$$\mathrm{p}_1(\mathrm{z}) := \mathrm{ind}_\Sigma(\lambda a.\, \lambda x.\, a)$$
$$\mathrm{p}_2(\mathrm{z}) := \mathrm{ind}_\Sigma(\lambda a.\, \lambda x.\, x)$$

□

The projection functions encode the axiom of choice[1]. Given a proof of $\exists x : P(x)$, we can now pick an element x satisfying $P(x)$. 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 the function nat_equals : $\mathbb{N} \to \mathbb{N} \to \mathit{Type}$, and I claimed that $\mathbb{N}$, $\emptyset$ and $\mathbf{1}$ all have type Type (which isn't actually quite true). What is the type then, of Type? Russels paradox still works in type theory, so we can't have $\mathit{Type} : \mathit{Type}$. The answer lies in the notion of *universes*.

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 $\mathit{Sort}\ n$. These universes contain each other as elements, so we have have $\mathit{Sort}\ n : \mathit{Sort}(1 + n)$.

[1] Or at least, they are equivalent to the axiom of choice. For more details, see section 2.7

We define two special universes

$$Prop := Sort\ 0$$
$$Type := Sort\ 1$$

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

$$prop_\bullet : \Pi_{P:Prop}\Pi_{x,y:P}\, x =_P y$$

and the induction principle on a *Prop* can only produce elements, whose type is in *Prop*.

Prop is the universe of propositions, and *Type* is the universe of regular types. The crucial thing about universes, is that it allows us to restrict what we can do with propositions. Thus, we can define the existential quantifier:

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

$\mathtt{type}\ (\mathtt{P} : \mathtt{A} \to \mathtt{Prop}) \vdash \exists_{a:A} P(a) : \mathtt{Prop}$
$\mid\ \mathtt{intro}_\exists : \Pi_{a:A} P(a) \to \exists_{a:A} P(a)$

with derived induction principle:

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

where $P : Prop$.

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 $A : Sort\ n$.

Now, since $P(z)$ 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

$\mathtt{type}\ (\mathtt{n}, \mathtt{m} : \mathbb{N}) \vdash n \leq m : \mathtt{Prop}$
$\mid\ \mathtt{zero\_le} : \Pi_{n:\mathbb{N}}\, 0 \leq n$
$\mid\ \mathtt{succ\_le} : \Pi_{n,m:\mathbb{N}}(n \leq m) \to (\mathtt{succ(n)} \leq \mathtt{succ(m)})$

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

$\mathtt{type}\ (\mathtt{n} : \mathbb{N}) \vdash \mathtt{even(n)} : \mathtt{Prop}$
$\mid\ \mathtt{zero\_even} : \mathtt{even(0)}$
$\mid\ \mathtt{step\_even} : \Pi_{n:\mathbb{N}}\ \mathtt{even(n)} \to \mathtt{even(succ(succ(n)))}$

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 $A : Prop$ 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)
| n, h := ind_even(•, λm.λh. •, n, h)

def if_even_eq_empty : Π_{n:ℕ}¬even(n) → is_even(n) = ∅
| n, h :=

def even_dec : Π_{n:ℕ} → even(n)∨¬even(n)
| 0 := ι₁(zero_even)
| succ(0) := ι₂(λh. ext(succ(0), h))
| succ(succ(n)) := ind_∨( λh.ι₁(step_even(n, h))
                        , λx.ι₂()
                        , even_dec(n))
```

2 Mathematics in type theory and Lean

We saw in the last chapter, that working in fully formal type theory was too cumbersome, so we gradually introduced shorter notation. However, it became dificult to figure out the types of terms during proofs. If we were to write out all the types, the proof would drown in type annotations. How can we remedy this situation?

We can introduce a computer. This computer can derive all the information we leave out, and show it to us when we want it. It can also check for us, that we only perform allowed operations. Thus it can check the correctness of our proofs.

There are many such computer programs, the most prominent of which are Coq, Agda, Isabelle and Lean. I run the risk of upsetting a lot of people when I say, that these are not that dissimilar. They are all built on a type theory similar to the one above, and they serve the same purpose. By learning one, it becomes possible to read, if not write, proofs in all of them. They do of course differ on some points: the community of Lean is generally more inclined to use classical logic in their proofs, where especially Agda and Isabelle stick closer to constructive logic.

Lean is interesting for a couple of reasons: it has a large library of formalized mathematics called mathlib. This is in contrast to Coq, where a lot of work is spread among different projects, so you may have difficulties combining results from disparate projects. For example if you were to formalize topological groups in Lean, mathlib contains definitions of both topological spaces, continuity and groups. So it's straightforward to define a topological group as a group with a topology such that the groups operations are continous. The Coq standard library, mathcomp, on the other hand, contains neither groups nor topology, so you need to hunt after projects online, that formalize groups and topology. Lean has the advantage that documentation for the entire mathlib is centralized, so you only ever have to look in one place for you theorems and definitions.

Second reason is, that Lean have proven itself to be capable of modern mathematics via the Liquid Tensor project [2]. This seems to indicate, that Lean is powerful enough to tackle most of the theorems we may wish to formalize.

Of course, the other provers have advantages. Coq and Agda are much better suited for doing Homotopy Type Theory than Lean[2] and higher categories. Also, if you're formalizing algorithms, the pervasive use of classical logic in mathlib may be an obstacle. Case in point: the division algorithm developed in this project cannot be executed on examples, because polynomials in Lean use classical logic in their definition. Coq and Agda are much more pure in this sense.

When reading this section, it's good to have Lean session running, and paste the given code snippets into Lean, so you can follow along. If you don't want to spend a lot of time installing Lean, you can use online editor at `https://leanprover-community.github.io/lean-web-editor/`. Just remove the lines that are already there and replace them with your own.

2.1 Vernaculars and user interface

Lean is built on a very small core logic, called the Calculus of Constructions, which is similar to the pure type theory described in section 1. Just like how we moved away from this core logic, Lean builds a *vernacular* or another language around this core. This is the language that we are going to interact with.

The syntax is slightly different from what we've seen so far, but the structure is the same. We can define natural numbers like so:

2.1 · Definition. The natural numbers in Lean

```
inductive N : Type
| z : N
| s : N → N
```

and we can define addition like before:

[2] This is a consequence of defining equality to be a proposition. HoTT requires equality to have a richer structure.

2.2 · **Definition.** Addition in Lean

```
def N_add : N → N → N
| z := id
| (s n) := s ∘ (N_add n)
```

Let's see how a proof of commutativity looks in Lean.

2.3 · **Theorem.** *Addition is commutative, i.e. there is a function*

```
add_comm :
```

Proof.

```
def N_add_zero : ∀n:N, N_add z n = N_add n z
| z     := rfl
| (s n) := congr_arg id (congr_arg s (N_add_zero n))

def N_add_succ : ∀ n m:N, N_add n m.s = N_add n.s m
| z     m := rfl
| (s n) m := congr_arg s (N_add_succ n m)

def N_add_comm : ∀ n m:N, N_add n m = N_add m n
| z     m := N_add_zero m
| (s n) m := eq.subst (eq.symm (N_add_succ m n))
                      (congr_arg s (N_add_comm n m))
```

We need the function `congr_arg : (f : α → β) → (h : a = b) → f a = f b` and `eq.symm : (a = b) → b = a` as well as `eq.subst : a = b → P a → P b` to carry out the proof. Luckily, these are already in Lean, so we don't have to prove them from scratch. Also, Lean can infer a lot of information, that we previously had to provide. For example `eq.symm`, which we had to provide both `a` and `b`, can now infer those two.

You might notice that there are a couple of differences from the way we did the proof of theorem 1.17. First, we need to use `congr_arg id` in `N_add_zero`, where I didn't before. I left it out, because we can define the identity function in such a way, that $id(x) \doteq x$ in which case it isn't nessecary. That isn't how it's defined in Lean, however, so we need it now. Also, `N_add_comm` is shorter, because we use `eq.subst`, which corresponds to $ind_=$ instead of transitivity. I find the proof by transitivity easier to read without type annotations. In Lean however, we can get type info anywhere, so this isn't a concern.

2.2 Tactic mode

Okay, so we can write proofs in Lean. But they're still kind of hard to write, even though all of our proofs so far have been pretty basic, because you need to piece the functions together in your head. Lean provides a different way of writing proofs, called *tactic mode*. Here is the above proof, in tactic mode:

```
def N_add_comm' : ∀ n m:N, N_add n m = N_add m n
| z     m := N_add_zero m
| (s n) m := begin
```

```
  rw N_add_succ m n,
  exact congr_arg s (N_add_comm n m),
end
```

It's difficult to read on paper, so I encourage you to copy it into a Lean session. Tactic mode lives between `begin ... end` and is a series of commands, that produce a proof. That is, everything in tactic mode translates into a proof in the normal style, but it's easier to iteratively build a proof in tactic mode.

If you copy the code above into a Lean session and place your cursor somewhere between `begin` and `end` you will see Lean displaying some information to you. This is because in tactic mode, Lean is an *interactive* theorem prover. It helps you orient yourself in the proof, by displaying your hypotheses and the statement you're working to prove.

In tactic mode, you work in a *context* of hypotheses, and work towards proving a *goal*. You can use commands to change the goal. The `rw h` command uses a hypothesis `h : a = b` and substitutes every occurence of `a` in the goal with `b`. It is equivalent to `eq.subst (eq.symm h) _` where is the underscore is replaced with the proof generated by the rest of the tactic block. That is, when you start the tactic block above, you are presented with

```
1 goal
N_add_comm': ∀ (n m : N), N_add n m = N_add m n
nm: N
⊢ N_add n.s m = N_add m n.s
```

after the first line (`rw N_add_succ m n`), the goal changes to

```
⊢ N_add n.s m = N_add m.s n
```

Next command is called `exact`. It returns whatever is given to it. In other words, it closes the goal, by providing a term of the correct type. After that line, Lean should tell you, that the goal has been accomplished. This means the proof is done.

We will do most of our proofs from now on in tactic mode. I'll exaplain new tactics as they arrive, but take a look at [1] for a full overview.

Just to get a feel for things, let's do another one of our previous proofs in tactic mode:

```
example (A B : Prop) : (¬ A) → (A ∨ B) → B := begin
  intro na,
  intro h,
  cases h, {
    cases (na h),
  }, {
    exact h,
  }
end
```

Copy the proof into your own Lean session, and step through the proof. Can you figure

out the structure? The `intro` tactic introduces assumptions. Our proposition has two assumptions, so we introduce twice, to get `na : ¬A` and `h : A ∨ B`. Next, `cases` on a hypothesis applies the induction principle on that hypothesis. In the case of disjunctions, the induction principle takes two functions, one for each case, and produces a proof. Thus, `cases h` produces two new goals, one with `h : A` and one with `h : B`. In the first case we use the induction principle on `false` to close the goal, and in the second we just use the given `h : B`.

2.3 The library of mathematics, mathlib

We introduced our own definition of the natural numbers, but if we actually wanted to do something with them, we would have a lot of work ahead of us. We still haven't defined multiplication, nor proven that it's commutative, we have no concept of prime numbers. We would want to show that every number has a unique prime decomposition, we'd want to introduce greatest common divisor and Eulers totient function, there's a lot of foundational tools still missing. We could do it, and perhaps it would be a good exercise, but we don't have to.

Lean has a large catalogue of mathematics called the mathlib. You can look up the documentation at `https://leanprover-community.github.io/mathlib_docs/`. Try searching for `zmod.chinese_remainder`. It states that, assuming $m, n \in \mathbb{Z}$ are coprime, there is a ring isomorphism between $\mathbb{Z}/mn\mathbb{Z}$ and $\mathbb{Z}/m\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$. You can click on most of the statement, to see the definition of $\mathbb{Z}/n\mathbb{Z}$ as a type, see the definition of `coprime` as well as the definition of a ring isomorphism. It's a good way to explore the mathlib, but also a bit overwhelming. Let's work through a simple example together:

Do an identity on the natural numbers using induction.

We'll gradually introduce more of the mathlib, as it becomes nessecary. You can get an overview of what is in the mathlib here: `https://leanprover-community.github.io/mathlib-overview.html`. Right now, think of your favorite part of undergraduate math, and check if it's in the mathlib.

2.4 Abstract structures

The type $\mathbb{N}$ seems to capture the natural numbers well, but most of mathematics doesn't work with concrete objects like natural numbers. More likely, we'll work with a group G or a vector space V. How do we do that in Lean?

We can introduce a type `group G`, which acts as a proof that `G` is a group. Then, the assumption "let G be a group" can be said as "let G be a set and g_G a be a proof that G has a group structure". This translates cleanly into Lean, for example `theorem Z_univ (G : Type) (gs : gro` stating that there is a group homomorphism from $\mathbb{Z}$ into any non-empty group (here `G1 →* G2` is the type of group homomorphisms from `G1` to `G2`).

So how what would this type `group G` look like? Close to how mathematicians usually

define groups: a group is a tuple $(G, \cdot : G \to G \to G, \ ^{-1} : G \to G)$ with the usual group axioms. In Lean, the group axioms are a part of the tuple:

```
@[class]
structure group (G : Type) :=
  (mul : G → G → G)
  (mul_assoc : ∀g₁ g₂ g₃ : G, mul (mul g₁ g₂) g₃ = mul g₁ (mul g₂ g₃))
  (one : G)
  (inv : G → G)
  (one_mul : ∀ g : G, mul one g = g)
  (mul_one : ∀ g : G, mul g one = g)
  (mul_inv : ∀ g : G, mul (inv g) g = one)
```

Remark: `structure` is just a neater way of writing a tuple, intead of as a product of types. Also, if you look at the definition of `group` in mathlib, you'll see a lot more going on. This is because Lean includes a lot of convenience functions in the definition as well. Feel free to ignore those.

This tuple has everything we need to work with groups, but it becomes cumbersome to pass the assumption that $\mathbb{Z}$ is a group along to every lemma, that talks about groups. To solve this, Lean introduces *type classes*, indicated by the `@[class]` above. These lay in the background, and doesn't need to be passed along explicitly. An example is

```
lemma mul_right_inv {G : Type} [group G] (g : G) :
  mul a (inv a) = one := begin
    rw [←mul_left_inv (inv g), inv_eq_of_mul (mul_left_inv g)]
  end
```

Here the assumption that `G` is a group is passed in square brackets and we can use it like so: `mul_right_inv 2 : mul 2 (inv 2) = 1`. The group itself is written in curly brackets, which means that Lean infers that from context. Thus both the group and the proof that it is a group is inferred from context, so we don't have to write them explicitly.

In addition to abbreviating notation, type classes can be automatically generated. For example, if G is a group and S is any type, the functions $S \to G$ have a natural pointwise group structure. This can be expressed in Lean as follows:

```
instance (G S : Type) [gs : group G] : group (S → G) := {
  mul := λf g, λs, mul (f s) (g s),
  mul_assoc := λf g h, begin apply funext, intro s, rw mul_assoc, end,
  one := λ_, one,
  inv := λf, λs, inv (f s),
  one_mul := λg, begin apply funext, intro s, rw one_mul, end,
  mul_one := λg, begin apply funext, intro s, rw mul_one, end ,
  mul_inv := λg, begin apply funext, intro s, rw mul_inv, end,
}
```

This automatically gives us a group structure on `S → ℤ` as well as on functions to any other group.

2.5 Well-founded recursion and well-orders

If you know about computability theory, you might have noticed a flaw in our treatment of the natural numbers. We've only allowed primitive recursive functions, i.e. functions defined using induction in one variable. The Ackermann function is an example of a function, which is not primitively recursive:

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m,A(m+1,n)) \end{aligned}$$

Neither the first nor the second argument always decreases, so we cannot define A by induction in either argument.

The way we treat this in mathematics, is using well-orders. We define the function A, and show that every recursive call decreases the arguments with respect to some well-order.

2.4 · Definition. A well-order is a total order $\leq$, i.e. a reflexive, antisymmetric, transitive and total relation, on a set S where every nonempty subset of S has a least element. Formally:

$$\forall A \subseteq S : A \neq \varnothing \implies \exists a \in A : \forall b \in A : a \leq b.$$

The most well-known well-order is the usual order on the natural numbers. In the case of the Ackermann function A, the pair (m,n) decreases lexicographically in each recursive call.

2.5 · Proposition. *If $\leq$ is a well-order on S, then so is the lexicographic order $\leq_{lex}$ on S^n, given by $(s_1,\dots,s_n) \leq_{lex} (t_1,\dots,t_n)$ if $s_i = t_i$ for $i = 1,\dots,k$ and $s_k < t_k$ for some $k \in \mathbb{N}$.*

Proof. The proof is by induction in n. If $n = 1$ we are done since $\leq$ is a well-order.

In the induction step, let $A \subseteq S^{n+1}$ and let $A' = \{(s_1,\dots,s_n) \mid (s_1,\dots,s_n,s_{n+1} \in A)\}$. By the induction hypothesis A' has a least element x'. Let $X = \{(s_1,\dots,s_{n+1}) \mid (s_1,\dots,s_n) = x\}$. Now, let x be the element of X with the least final entry. Then x is a least element of A.□

Now, the reason well-orders are good for recursive functions is the following:

2.6 · Proposition. *Let $\leq$ be a well-order on S and $\{a_i\}$ be a strictly decreasing sequence of elements in S. Then a_i is a finite sequence.*

Proof. Viewing $\{a_i\}$ as a subset of S, well-ordering gives an $n \in \mathbb{N}$ such that $\forall m \geq n : a_m = a_n$. Thus a_i stops at n. □

This is what enables general recursion. Since every time we call the Ackermann function, the arguments strictly decreases according to some well-order, the sequence of calls must be finite. Thus it is well-defined.

This is all well and good, but how do we do this in Lean? It turns out, well-orders have a very nice constructive counterpart, which enables us to define well-founded recursion as an induction principle.

2.7 · Definition. Let S be a type, $\lhd : S \to S \to Prop$ be a relation and $x : S$. x is *accessible* if for every element $y \in S$ with $y \lhd x$, we have that y is accessible. We write $acc_{\lhd} x$ to say that x is accessible.

Formally:

$$acc_{\lhd} x := \forall y \in S : y \lhd x \implies acc_{\lhd} y$$

2.8 · Proposition. *Every $n \in \mathbb{N}$ is accessible with respect to the usual ordering relation $<$.*

Proof. The proof is by strong induction. Since there is no number less than 0, every element $n < 0$ is accessible. Thus 0 is accessible.

Now, assume every $m < n$ is accessible. This is exactly the definition of n being accessible. Thus n is accessible. □

In Lean we define the type `acc r : S → Prop` such that `acc r x` is a proof that `x` is accessible with respect to the relation `r`.

2.9 · Definition. The type of accessible elements

```
inductive acc {S : Sort u} (r : S → S → Prop) : S → Prop
| intro : ∀(s : S), (∀ (t : S), r t s → acc t) → acc s
```

This type has induction principle

```
acc.rec : (Π (s : S), (Π (t : S), r t s → acc r t)
                    → (Π (t : S), r t s → P t) → P s)
        → Π {x : S}, acc r x → P x
```

There are a couple of new syntactic things going on here. First, the argument `{S : Sort u}` in curly braces is an *implicit* argument, which you don't have to supply, Lean infers it from context. Also, because the relation `r` is to the left of the colon in first line, we don't have to supply it in the recursive references to `acc`.

Let's see that this formulation captures the same concept:

2.10 · Theorem. *Let $\leq$ be a total order on a set S. Then $\leq$ is a well-order if and only if every element of S is accessible w.r.t. $<$.*

Proof. First, assume $\leq$ is a well-order, and assume for contradiction that not every element of S is accessible. Let $A = \{x \in S \mid \neg acc_{<} x\}$ be the set of inaccessible elements. Since $\leq$ is a well-order, this set has a least element, say x_0. However, every element less than x_0 is accessible, which means x_0 is accessible, which is a contradiction.

On the other hand, assume every element of S is accessible and let $A \subseteq S$ with $A \neq \emptyset$. We prove a slight variation of the desired statement; that for all $x \in S$, any subset $A \subseteq S$ containing x must have a least element. We do this by induction in x.

The induction principle on $acc_{<}$ states, that if we, assuming every $y < x$ where y is accessible and $P\ y$ holds, can prove that $P\ x$ holds, then $P\ s$ holds for every accessible $s \in S$. Thus, let $s \in S$ and assume that every subset containing a $t < s$ has a least element.

Let $A \subseteq S$ be a subset with $s \in A$. Then, consider $A' = \{s' \in A \mid s' < s\}$. We have two cases: if $A' = \varnothing$ then s is a least element of A. Otherwise, A' has a least element s_0, since it contains an element strictly less than s. Then this s_0 is also a least element of A. Thus A has a least element.

Thus, every set $A \subseteq S$ containing an accessible element has a least element. Since A is nonempty and every element in S is accessible, A has a least element. □

That last implication might be hard to wrap your head around, so here it is in Lean:

```
lemma acc_has_min {S : Type} (r : S → S → Prop) (wf : ∀s:S, acc r s)
  (A : set S) (h : A.nonempty) : ∃s₀ ∈ A, ∀ x ∈ A, ¬ r x s₀ := begin
    rcases h with ⟨ a, ha ⟩,
    refine (acc.rec _ (wf a)) ha,
    intros x acc_x H hx,
    let A' := {s' ∈ A | r s' x},
    cases set.eq_empty_or_nonempty A', {
      existsi x,
      existsi hx,
      intros x' hx' nax',
      have x'_in_A' : x' ∈ A' := ⟨ hx', nax' ⟩,
      rw h at x'_in_A',
      exact set.not_mem_empty x' x'_in_A',
    }, {
      rcases h with ⟨ t, ht ⟩,
      have rtx : r t x := ht.2,
      have t_in_A : t ∈ A := set.mem_of_subset_of_mem
                                (set.sep_subset _ _) ht,
      exact H t rtx t_in_A,
    }
  end
```

Now we have everything we need to do well-founded recursion. We define a quick wrapper around the induction on `acc`:

```
variables {S : Type} {C : S → Type} {r : S → S → Prop}

def fix (F : Πx, (Πy, r y x → C y) → C x)
        (s : S) (h : acc r s) : C s :=
  acc.rec (λx' _ ih, F x' ih) h
```

Using this, we can define the Ackermann function as such:

```
def ack_aux : Π (p1 : ℕ ×ₗ ℕ) (h : Π (p2 : ℕ ×ₗ ℕ),
                                   p2 < p1 → ℕ ×ₗ ℕ), ℕ ×ₗ ℕ
| ⟨ 0   , n   ⟩ _ := (0, n+1)
| ⟨ m+1, 0   ⟩ h := h (m, 1) sorry
| ⟨ m+1, n+1 ⟩ h := h (m, (h (m+1, n) sorry).snd) sorry

def ack (p : ℕ × ℕ) : ℕ := (fix ack_aux p ((prod.lex_wf
                                            nat.lt_wf
                                            nat.lt_wf).apply p)).snd
```

Remark: `ℕ ×ₗ ℕ` denotes the product ordered by the lexicographic order. This is because the usual order on the product is the pointwise order, which is not a total order.

Remark: The mathlib has remarkably few lemmas on the lexicographic order on products, which is why this definition uses `sorry` wherever we have to prove that the recursive call is decreasing. It would be a good exercise to implement the needed lemmas:

```
lemma l1 (m : ℕ) : to_lex (m, 1) < to_lex (m + 1, 0) :=
begin admit, end
lemma l2 (m n : ℕ) : to_lex (m + 1, n) < to_lex (m + 1, n + 1) :=
begin admit, end
lemma l3 (m n x: ℕ) : to_lex (m, x) < to_lex (m + 1, n + 1) :=
begin admit, end
```

You're going to need the lemmas `prod.lex.lt_iff` and `nat.lt_succ_self` from mathlib.

2.6 Well-founded recursion using the equation compiler

Now that the mathematical foundation for well-founded recursion is in place, we would like a better notation. Lean can actually express the Ackermann function directly:

```
def ack2 : ℕ → ℕ → ℕ
| 0     n     := n+1
| (m+1) 0     := ack2 m 1
| (m+1) (n+1) := ack2 m (ack2 (m+1) n)
```

We don't need to supply the well-order since Lean knows there is a well-order on `ℕ × ℕ` namely the lexicographic order, and it can even prove that the recursive calls are decreasing on its own. This means we can also express addition more naturally:

```
def add : ℕ → ℕ → ℕ
| 0     n := n
| (m+1) n := (add m n) + 1
```

2.7 Sets

So far, we've used types to fill the role of sets. However, that's not always practical. We don't have subtypes, so there is no way to express the concept of a subset. This means we can't talk about partitions or restrict a function to a subset. How would we talk about these things in Lean?

The answer is simple. The statement $a \in A$ is a proposition, so we simply define

```
def set (α : Type) := α → Prop
def mem (α : Type) (S : set α) (a : α) : Prop := S a
```

so the set A is actually a predicate, deciding whether any element a is in A or not. We can define notation, so that `a ∈ S := S a` and there we go. A subset is simply a set, for which every element is conatined in the superset

```
def subset (α : Type) (S T : set α) : Prop := ∀(a : α), a ∈ S → a ∈ T
```

By unfolding definitions, this is actually equal to `∀(a : α), S a → T a`. We can define unions as `S ∪ T := ∀(a:α), S a ∨ T a` and similarly intersections.

Finally, we can define the type of elements in a set, which would act as a subtype of the type α:

```
structure subtype {α : Type} (S : set α) : Type :=
(val : α)
(property : val ∈ S)
```

We can use these constructions to give the axiom of choice for sigma-types, in a slightly unusual formulation: Let $\sim$ be a relation between two types α and β and assume $\forall a : \alpha, \exists b : \beta, a \sim b$. Then there exists a function $f : \alpha \to \beta$ such that $\forall a : \alpha, a \sim f(a)$.

```
def AC {α β : Type} {r : α → β → Prop}
  (H : ∀(a : α), psigma (λb, r a b)) :
  psigma (λ(f : α → β), ∀(a : α), r a (f a))
  ⟨ λa, (H a).fst, λa, (H a).snd ⟩
```

Remark: for reasons unknown to me, the definition of sigma-types in Lean forbid propositions as the second element. Instead, we use `psigma`, which allows them. `psigma (β : α → Type)` means `Σa:α, β(a)`.

The way turn this into the usual axiom of choice is by letting `A := subtype (set α)`, `B := α` and the relation `a ~ b` denote `b ∈ a`. Then, assuming every set in `A` is nonempty, we have that every `a ∈ A` is in relation to some `b : B`, so `AC` gives us a choice function.

3 Gröbner bases as an extended example

In this section we'll develop the theory of Gröbner bases in polynomial rings over fields. We'll see both "normal" informal proofs and the proofs in Lean, to get an understanding of the difference.

We'll proceed as follows: first, we prove Dicksons lemma, which enables us to prove that every polynomial ring has a Gröbner basis. Then we define term orderings and prove that they form well-orders. Then we define Gröbner bases. Finally, we define the multivariate division algorithm, and prove its basic properties. Finally, we'll prove that division by a Gröbner basis gives 0 if and only if the polynomial is in the ideal generated by the Gröbner basis.

3.1 Dicksons lemma

Dicksons lemma is a sort of well-ordering axiom for $\mathbb{N}^n$.

3.1 · Lemma. *Let $n \in \mathbb{N}$ and $S \subseteq \mathbb{N}^n$ be a non-empty subset. Then there exists a finite set $V = \{x_1, \ldots, x_r\} \subseteq S$ such that*

$$S \subseteq \bigcup_{i=1}^{r} x_i + \mathbb{N}^n.$$

Proof. The proof is by induction in n. If $n = 0$, we have there is only a single vector of length 0, the empty vector. Thus $\mathbb{N}^0 = \{()\}$. In this case, we can choose $V = S$, which must be a finite set.

For the induction step, assume every subset $S' \subseteq \mathbb{N}^n$ has a finite subset giving a lower bound of S', and let $S \subseteq \mathbb{N}^{n+1}$.

Now, let's define some notation. Define $t : \mathbb{N}^{n+1} \to \mathbb{N}^n$ given by $t((s_1, \ldots, s_{n+1})) = (s_2, \ldots, s_{n+1})$ to be the function removing the first entry of a vector. Let's also define $U(\{x_1, \ldots, x_r\}) = \bigcup_{i=1}^{r} x_i + \mathbb{N}^l$ where $x_i \in \mathbb{N}^l$.

Let $S' = \{t(x) \mid x \in S\}$ be the set of vectors in S where we remove the first entry. The induction hypothesis then gives us a finite set $v' = \{x'_1, \ldots, x'_r\} \subseteq S'$ such that $S' \subseteq U(v')$.

Since $v' \subseteq S'$, we can find a finite set $v = \{x_1, \ldots, x_r\} \subseteq S$ such that $v' = \{t(x) \mid x \in v\}$. Now, this v may not be a lower bound of S, only of the last n entries. So, let's try to extend v to be a lower bound of the first entry as well.

We can define $M = max(\{s_1 \mid s \in v\})$ and let $S_{\geq M} = \{s \in S \mid s_1 \geq M\}$. Here, s is a vector and s_1 denotes the first entry of s. Then we at least have $S_{\geq M} \subseteq U(v)$.

Now, for all $i = 1, \ldots, M-1$ define $S_i = \{s \in S \mid s_1 = i\}$, where again s_1 is the first entry of s, and $S'_i = \{t(s) \mid s \in S_i\}$. We then have $S = S_{\geq M} \cup \bigcup_{i=0}^{M-1} S_i$. By using the induction hypothesis on each of these S_i, we get finite sets $v'_i \subseteq S'_i$ such that $S'_i \subseteq U(v'_i)$. Similarly to before, we can find $v_i \subseteq S_i$ such that $v'_i = \{t(x) \mid x \in v_i\}$. Then we have $S_i \subseteq U(v_i)$.

Finally, let $V = v \cup \bigcup_{i=0}^{M-1} v_i \subseteq S$. To see that $S \subseteq U(V)$, let $s \in S$. Then either $s \in S_{\geq M}$ or $s \in S_i$ for some i. In the first case, there is a $s' \in v$ such that $s = s' + x$ for some $x \in \mathbb{N}^{n+1}$. Since $s' \in v \subseteq V$ we can use $s' \in V$ to prove $s \in U(V)$.

Similarly, if $s \in S_i$ for some i, we have $s \in U(v_i)$, thus $s = s' + x$ for some $s' \in v_i$ and $x \in \mathbb{N}^{n+1}$. Since $s' \in v_i \subseteq V$, taking $s' \in V$ proves $s \in U(V)$, and this concludes the proof. □

When doing this proof in Lean, we prove an equivalent, but slightly different statement. First, $\mathbb{N}^n$ is interpreted as a vector of length n with entries in $\mathbb{N}$, not as the cartesian product of n copies of $\mathbb{N}$. Next, instead of writing $\bigcup_{i=0}^{r} x_i + \mathbb{N}^n$ we unfold this definition to

$$\{s \in \mathbb{N}^n \mid \exists s' \in \{x_1, \ldots, x_r\}, x \in \mathbb{N}^n : s = s' + x\}.$$

But that is not all. Addition on vectors isn't defined in mathlib, so we need to define it and prove a bunch of lemmas, to show that it behaves well. The code for this is in the

appendix, section C.

3.2 · Lemma. *Dicksons lemma in Lean*

Let $n \in \mathbb{N}$ and $S \subseteq \mathbb{N}^n$. Then there exists a finite set $V \subseteq S$ such that

$$S \subseteq \{s \in \mathbb{N}^n \mid \exists s' \in V, x \in \mathbb{N}^n : s = s' + x\}.$$

Proof. This proof is rather long, and we need to build

First, we define some notation:

```
    cases h with x hx,
    apply exists.intro (finset.has_singleton.singleton x),
    split, {
      rw coe_singleton,
      exact set.singleton_subset_iff.2 hx,
    }, {
```

This is so that `upper_set V` $= \{s \in \mathbb{N}^n \mid \exists s' \in \{x_1, \ldots, x_r\}, x \in \mathbb{N}^n : s = s' + x\}$.

Now, let's look at the proof from before. First, we have the base case, $n = 0$:

```
lemma dickson_zero (S : set (vector ℕ 0)) :
  ∃ v : finset (vector ℕ 0), ↑v ⊆ S ∧ S ⊆ upper_set v := begin
  by_cases S.nonempty, {
    cases h with x hx,
    apply exists.intro (finset.has_singleton.singleton x),
    split, {
      rw coe_singleton,
      exact set.singleton_subset_iff.2 hx,
    }, {
      intros s hs,
      rw upper_set,
      existsi [s, x, finset.mem_singleton_self x],
      rw vector.eq_nil s,
      simp,
    }
  }, {
    rw set.not_nonempty_iff_eq_empty at h,
    apply exists.intro ∅,
    split, {
      rw finset.coe_empty,
      exact set.empty_subset _,
    }, {
      rw h,
      exact set.empty_subset _,
    },
  }
end
```

Second, there was a step where we used the axiom of choice, do go from a finite set $v' \subseteq \mathbb{N}^n$ to a finite set $v \subset \mathbb{N}^{n+1}$ such that $t(v) = v'$. The proof that this is possible is

called `single_preimage`, and is in the appendix, section D. Here, there is also a version of the axiom of choice, formulated using subtypes, called `subtype_axiom_of_choice`, which we'll also need.

Third, we'll need some lemmas about how this `upper_set` behaves. We'll need that $U(\varnothing) = \varnothing$:

```
lemma upper_set_of_empty_eq_empty (n : ℕ) : @upper_set n ∅ = ∅ := begin
  rw upper_set,
  rw ←set.subset_empty_iff,
  intros x hx,
  rcases hx with ⟨ x', s, hs, hx ⟩,
  exfalso,
  exact finset.not_mem_empty s hs,
end
```

and we'll need

```
lemma lift_upperset {n : ℕ} (i : ℕ) (S : set (vector ℕ n.succ))
                    (v : finset (vector ℕ n.succ))
                    (H : (tail '' S) ⊆ upper_set (image tail v))
                    (H2 : ∀ s ∈ S, i ≤ head s)
                    (H3 : ∀ s ∈ v, head s ≤ i) : S ⊆ upper_set v :=
begin
  intros s hs,
  rw upper_set,
  rw mem_set_of_eq,
  have s'_in_S' := mem_image_of_mem tail hs,
  have s'_in_upperset := mem_of_subset_of_mem H s'_in_S',
  rcases s'_in_upperset with ⟨ x',s0', hs0', hs0''⟩,
  rcases finset.mem_image.mp hs0' with ⟨ s0, s0_in_v, hs0 ⟩,
  let x := (head s - head s0) ::ᵥ x',
  existsi [x, s0, s0_in_v],
  rw ←(cons_head_tail s),
  rw eq_comm,
  rw eq_cons_iff,
  split, {
    simp *,
    refine nat.sub_add_cancel _,
    specialize H3 s0 s0_in_v,
    specialize H2 s hs,
    exact le_trans H3 H2,
  }, {
    simp *,
  }
end
```

Finally, we need that $S = S_{\geq M} \cup \bigcup_{i=0}^{M} S_i$:

```
lemma dickson_partition (n M : ℕ) (S : set (vector ℕ n.succ)) :
  S = {s ∈ S | s.head ≥ M} ∪ ⋃(i : fin M), {s ∈ S | i.val = s.head} :=
begin
  apply set.eq_of_subset_of_subset, {
```

```
    intros s hs,
    cases nat.decidable_le M s.head, {
      rw not_le at h,
      let i : fin M := ⟨ s.head, h ⟩,
      apply set.mem_union_right,
      rw mem_Union,
      existsi i,
      apply mem_sep hs,
      simp,
    }, {
      apply set.mem_union_left,
      apply mem_sep hs,
      exact h,
    }
  }, {
    intros s hs,
    cases hs, {
      exact (mem_sep_iff.mp hs).left
    }, {
      rw set.mem_Union at hs,
      cases hs with i hs,
      exact (mem_sep_iff.mp hs).left,
    }
  }
end
```

With all that out of the way, here is the proof, with the same structure as the informal proof:

```
theorem dickson (n : ℕ) (S : set (vector ℕ n)) :
  ∃ v : finset (vector ℕ n), ↑v ⊆ S ∧ S ⊆ upper_set v :=
begin
  -- The proof is by induction in n
  induction n with n n_ih, {
    -- The base case is handled in a lemma
    exact dickson_zero S,
  }, {
    -- Now, let S' = tail(S) and use the induction hypothesis
    -- to find v'
    let S' := image vector.tail S,
    have ih := n_ih S',
    cases ih with v' hv,
    cases hv with v'_sub_S' S'_sub,
    -- Find v s.t. tail(v) = v'
    have ex_v := single_preimage S v' vector.tail v'_sub_S',
    cases ex_v with v,
    cases ex_v_h with v_sub_S tv_eq_v',
    -- We need that v ≠ ∅.
    -- If not, we get that S = ∅
    cases (@finset.decidable_nonempty (vector ℕ n.succ) v),
    {
      -- If v = ∅ then v' = ∅, so S' = ∅, which implies S = ∅.
      -- When S = ∅, it's easy.
```

```
  rw finset.not_nonempty_iff_eq_empty at h,
  apply exists.intro v,
  rw h,
  split, {
    rw finset.coe_empty,
    exact empty_subset S,
  }, {
    rw h at tv_eq_v',
    rw finset.image_empty at tv_eq_v',
    have upper_empty := upper_set_of_empty_eq_empty n,
    rw tv_eq_v' at upper_empty,
    rw upper_empty at S'_sub,
    rw subset_empty_iff at S'_sub,
    rw set.image_eq_empty at S'_sub,
    rw S'_sub,
    exact empty_subset _,
  }
}, {
  -- Now that v ≠ ∅, we can find the maximum.
  have image_v_nonempty := finset.nonempty.image h head,
  let M : ℕ := finset.max' (finset.image head v) image_v_nonempty,
  -- Now, partition S into S_gtM and S_i as in the proof.
  let Si := λ (i : (fin M)), ({s ∈ S | i.val = head s}),
  let S_gtM := {s ∈ S | M ≤ head s},
  let S_U := S_gtM ∪ ⋃ i, Si i,
  -- Show that this is actually a partition, using a lemma
  have S_eq_S_U : S = S_U := dickson_partition n M S,
  -- Show that S_gtM ⊆ upper_set v, using a lemma
  have S_gtM'_sub_S' : tail '' S_gtM ⊆ S' := image_subset tail
                                               (sep_subset S _),
  have c_gtM : S_gtM ⊆ upper_set v := lift_upperset M S_gtM v
    (subset_trans S_gtM'_sub_S' (eq.subst tv_eq_v'.symm S'_sub))
    (λs hs, hs.2) (λs hs, le_max' (image head v) (head s)
                    (mem_image_of_mem head hs)),

  -- We use the induction hypothesis to find v_i'
  let t' := λ (i : fin M),  n_ih ((@tail ℕ n.succ) '' (Si i)),

  -- For technical reasons, it's easier to define the finite sets
  -- v_i using subtypes.
  -- Thus, vi is a function from fin M to finite sets b, s.t.
  -- b ⊆ Si i and Si i ⊆ upper_set b.
  have vi' := @subtype_axiom_of_choice
              (fin M)
              (finset (vector ℕ n))
              (λ (i : fin M) (v : finset (vector ℕ n)),
                P ((@tail ℕ n.succ) '' (Si i)) v)
              t',
  let vi : Π (i : fin M), {b // P (Si i) b} :=
  begin
    intro i,
    let b' := vi' i,
    have P_b' := b'.property,
```

```
  cases P_b',
  have ex_b := single_preimage (Si i) b'.val vector.tail P_b'_left,
  choose b hb using ex_b,
  exact ⟨ b, begin
    split, {
      exact hb.1,
    }, {
      rw ←hb.2 at P_b'_right,
      refine lift_upperset i (Si i) b P_b'_right
        (λs hs, le_of_eq hs.2)
        (λs hs, begin exact le_of_eq
          (mem_of_subset_of_mem hb.1 hs).2.symm end),
    }
  end ⟩,
end,
-- Then, we can unpack the subtype, to get two functions,
-- vi_val giving us finite sets and vi_P giving us proofs,
-- that vi_val i satisfies the proper conditions.
let vi_val := λ (i : fin M), (vi i).val,
have vi_P := λ (i : fin M), (vi i).property,
-- All that work lets us define the finite set V
let V := v ∪ finset.bUnion (finset.univ) vi_val,
existsi V,
-- Now, we have to prove that V ⊆ S and S ⊆ upper_set V
split, {
  -- V ⊆ S since every constituent of V is a subset of S
  rw coe_union,
  refine union_subset v_sub_S _,
  rw coe_bUnion,
  refine Union_subset _,
  intro i,
  apply Union_subset,
  intro _,
  exact subset_trans (vi_P i).1 (sep_subset S _),
},{
  -- Now, we prove that S ⊆ upper_set V
  rw S_eq_S_U,
  intro s,
  assume hs,
  cases ((set.mem_union _ _ _).mp hs), {
    -- If s ∈ S_gtM, we use the s', x we know exists since
    -- S_gtM ⊆ upper_set v
    have s_in_upper_v := set.mem_of_subset_of_mem c_gtM h_1,
    rcases s_in_upper_v with ⟨ x, s', s'_in_v, hs' ⟩,
    have s'_in_V : s' ∈ V := finset.mem_union_left _ s'_in_v,
    exact ⟨x, s', s'_in_V, hs'⟩,
  }, {
    -- If s ∈ USi i, then find the i s.t. s ∈ Si i.
    rw mem_Union at h_1,
    cases h_1 with i s_in_Si,
    -- Find the right vi and get that Si i ⊆ upper_set vi v
    cases (vi_P i) with vi_sub_Si Si_sub_upper,
    have s_in_upper := set.mem_of_subset_of_mem Si_sub_upper s_in_Si,
```

```
          rcases s_in_upper with ⟨ x, s', s'_in_vi, hs' ⟩,
          -- Prove that s' ∈ Uvi i, to then prove that s' ∈ V.
          have s'_in_Uvi : s' ∈ (finset.bUnion univ vi_val) := begin
            rw finset.mem_bUnion,
            apply exists.intro i,
            apply exists.intro (finset.mem_univ i),
            exact s'_in_vi,
          end,
          have s'_in_V : s' ∈ V := finset.mem_union_right _ s'_in_Uvi,
          exact ⟨ x, s', s'_in_V, hs' ⟩,
        }
      },
    }
  }
end
```

□

References

[1] Jeremy Avigad, Gabriel Ebner **and** Sebastian Ullrich. ?The Lean Reference Manual? **inchapter** 6: URL: https://leanprover.github.io/reference/tactics.html.

[2] Lean Community. *Liquid Tensor Experiment*. https://github.com/leanprover-community/lean-liquid. 2022.

A Inductively defined types and the induction principle

Here, we'll go through the rules governing the definition of inductive datatypes and their derived induction principle. Usually, you don't need to worry about these details, as you can just ask Lean, but it's nice to have an overview. Recall that an inductive type is given by its constructors. Foe example, the natural numbers are given by:

```
inductive N : Type
| z : N
| s : N → N
```

Note, that the constructors can depend on the type, but their final result must be the type we're defining. There are some restrictions on how we're allowed to depend on the type we're defining. We say that *non-positive* occurences of `N` are forbidden. In the function type `A → B`, we say that `A` is in the negative position and `B` is in the positive position. Looking at each argument for every constructor, if `N` occurs in a function type, it must only ever occur in a positive position. Thus, the above is allowed, since `N` never occurs in a function type in any argument. Similarly, the following constructors are allowed:

```
| s1 : N → N → N
| s2 : (false → N) → N
```

However, all of the following constructors are disallowed:

```
| s4 : (N → N) → N
| s3 : (false → N → false) → N
| s5 : (false → N → N) → N
```

You might be able to see why this restriction is here: for non-propositional types, the induction principle forces the constructors to be injective. However, we can't have an injection $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ by Cantors diagonal argument. And even though technically the constructor `s3 : (false → N → false) → N` won't pose this issue, since there is only a single function `false → N → false`, proving this is non-trivial, so Lean forbids it.

Also, the type definition must be monotone in universes. This means, for example in the product type:

```
universes u v
inductive my_prod {A : Sort u} {B : Sort v} : A → B → Sort (max u v)
| intro (a : A) (b : B) : my_prod a b
```

when we take arguments from a certain universe, the resulting type must be in that same universe, or higher. Thus, when we accept two inputs, that may lie in their own universe, the resulting `my_prod A B` must lie in the highest of the two universes, denoted by `max u v`. This rule doesn't apply for types in `Prop`, thus the following is allowed:

```
universes u v
inductive my_prod {A : Sort u} {B : Sort v} : A → B → Sort 0
| intro (a : A) (b : B ) : my_prod a b
```

since `Prop := Sort 0`.

Let's move on to the induction principle. The induction principle, called the recursor in Lean, will be a dependent function, taking one argument for each constructor. In this case it looks likes this:

```
N.rec : Π {P : N → Sort l}, P z → (Π (n : N), P n → P (s n)) → Π (n : N), P n
```

Each argument to the recursor is a function, taking the same arguments as the corresponding constructor, as well as the constructor taken on those arguments, and returning something of type `P (c a..)` where `c` is the constructor and `a..` is the list of arguments for that constructor. If a constructor takes no arguments, as `z`, the corresponding function is not a function, just a term of type `P z`. The second argument has type `(Π (n : N), P n → P (s n))`

In the case of `N` which lives in the universe `Type`, `P (c a..)` can live in any universe. However, when we're working with types in `Prop`, things are a little more complicated.

Usually, when recursing over a type `T : Prop`, the result must also lie in `Prop`. In this case, `T` only has a single element, so the result `P t` cannot vary. Thus we simplify, so the return type is just `P`.

However, there is an exception when the type we're recursing over only has one constructor. The arguments to this constructor comes from three categories of types: they are either specific type defined previously, an abstract type indexed by the type or the type we're defining itself. For a contrived example, consider the type `T l` of trees with nodes of type `L` and integers at the nodes, whose left-most leaf is `l : L`:

```
inductive T {L : Sort i} : L → Sort (max i 1)
| leaf : Π l : L, T l
| node {l1 l2 : L} : ℤ → T l1 → T l2 → T l1
```

Here, `Leaf` takes an argument of a indexed type `L`, and `Node` takes first an argument of a previously defined type, and then two arguments of type `T`, the type we're defining.

If the type has only a single constructor, and that constructor only takes arguments of index types or of `Prop` types, then the recursor can return types in any universe. For example, in the case of conjunction:

```
inductive eq {α : Sort u} (a : α) : α → Prop
| refl : eq a
```

We have only one constructor, and that constructor takes an argument of an index type. Thus, the recursion principle on `eq` is:

```
eq.rec :
  ∀ {α : Sort u} {a : α} {P : α → Prop},
  (α → P a) → ∀ {b : α}, eq a b → P b
```

B Useful tactics

intro/intros

rw

have/let

apply/refine

split/left/right

cases

induction

C A monoid structure on `vector N n`

```
import data.nat.basic
import data.vector
import data.vector.zip
```

```
namespace vector
open vector

instance (n : ℕ) : has_add (vector ℕ n) :=
  ⟨ λ v1 v2, zip_with (+) v1 v2 ⟩
instance (n : ℕ) : has_zero (vector ℕ n) :=
  ⟨ repeat 0 n ⟩

lemma add_eq_zip_add {n : ℕ} (v1 v2 : vector ℕ n) :
  v1 + v2 = zip_with (+) v1 v2 := rfl

@[simp]
lemma zip_with_head {α β γ : Type*} {n : ℕ} (f : α → β → γ)
  (x : vector α n.succ) (y : vector β n.succ) :
  (zip_with f x y).head = f (x.head) (y.head) := begin
    repeat {rw ←nth_zero},
    exact (zip_with_nth f x y 0),
  end

@[simp]
lemma add_head {n : ℕ} (v1 v2 : vector ℕ n.succ) :
  (v1 + v2).head = v1.head + v2.head := begin
    rw add_eq_zip_add,
    simp *,
  end

@[simp]
lemma add_tail {n : ℕ} (v1 v2 : vector ℕ n.succ) :
  (v1 + v2).tail = v1.tail + v2.tail := begin
    repeat {rw add_eq_zip_add},
    simp *,
  end

lemma add_zero {n : ℕ} (v : vector ℕ n) : v + 0 = v := begin
  induction n, {
    rw vector.eq_nil v,
    simp,
  }, {
    rcases exists_eq_cons v with ⟨ head, tail, h⟩,
    rw h,
    rw vector.eq_cons_iff,
    split, {
      simp *,
      refl,
    }, {
      simp *,
      exact n_ih tail,
    }
  }
end
lemma cons_add_eq_add_cons {n : ℕ} (v1 v2 : vector ℕ n) (a b : ℕ) :
  (a ::ᵥ v1) + (b ::ᵥ v2) = (a + b) ::ᵥ (v1 + v2) := begin
```

```
    rw add_eq_zip_add,
    rw eq_cons_iff,
    split, {
      rw zip_with_head,
      rw cons_head,
      rw cons_head,
    },{
      rw zip_with_tail,
      rw cons_tail,
      rw cons_tail,
      refl,
    }
  end

lemma add_comm {n : ℕ} (v1 v2 : vector ℕ n) : v1 + v2 = v2 + v1 := begin
  induction n with n n_ih, {
    simp *,
  }, {
    rcases exists_eq_cons v1 with ⟨ x, xs, hx ⟩,
    rcases exists_eq_cons v2 with ⟨ y, ys, hy ⟩,
    rw [hx, hy],
    repeat {rw cons_add_eq_add_cons},
    rw n_ih,
    rw nat.add_comm,
  }
end

lemma vector.zero_add {n : ℕ} (v : vector ℕ n) : 0 + v = v := begin
  rw vector.add_comm,
  rw vector.add_zero,
end

lemma vector.add_assoc {n : ℕ} (v1 v2 v3 : vector ℕ n) :
  (v1 + v2) + v3 = v1 + (v2 + v3) := begin
    induction n with n n_ih, {
      simp *,
    }, {
      rcases exists_eq_cons v1 with ⟨ x, xs, hx ⟩,
      rcases exists_eq_cons v2 with ⟨ y, ys, hy ⟩,
      rcases exists_eq_cons v3 with ⟨ z, zs, hz ⟩,
      rw [hx, hy, hz],
      repeat {rw cons_add_eq_add_cons},
      rw nat.add_assoc,
      rw n_ih,
    }
  end

instance (n : ℕ) : add_comm_monoid (vector ℕ n) := {
  add := has_add.add,
  add_assoc := vector.add_assoc,
  zero := 0,
  zero_add := vector.zero_add,
  add_zero := vector.add_zero,
```

```
  add_comm := vector.add_comm,
}

end vector
```

D A preimage of a finite set

If we're given a function $f : A \to B$ and a finite set $v' = \{x'_1, \ldots, x'_n\} \subset B$ such that $v' \subseteq \operatorname{Im} f$, we can find a finite set $v \subseteq A$ such that $f(v) = v'$. This is done by considering each $x'_i \in v'$ and picking some $x_i \in f^{-1}(\{x'_i\})$. Then $v = \{x_i \mid i = 1, \ldots, n\}$.

When we translate this to Lean, things get a little more complex, as we need to be working with an actual set. So we're given a function $f : \alpha \to \beta$ and a set of elements from α, $S : set\alpha$ and a finite set $v' \subseteq f(S)$. Then we find a finite set $v : finset\alpha$ so that $v \subseteq S$ and $v' = f(v)$.

```
lemma single_preimage {α β : Type*} [decidable_eq β] [decidable_eq α]
  (S : set α) (v' : finset β) (f : α → β) (sub : ↑v' ⊆ f '' S) :
  (∃ (v : finset α), ↑v ⊆ S ∧ finset.image f v = v') :=
begin
  let h := set.mem_image f S,
  let h' : ∀ (y : subtype (f '' S)), ∃ (x : α), x ∈ S ∧ f x = y := begin
    intro y,
    exact (h y.val).mp y.property,
  end,
  choose F hF using axiom_of_choice h',
  let FF : subtype (f '' S) → α := F,
  let v'' : finset (subtype (f '' S)) := @finset.subtype β (f '' S)
                                           (dec_pred (f '' S)) v',
  let v := finset.image FF v'',
  apply exists.intro v,
  apply and.intro, {
    simp *,
    rw set.subset_def,
    intros x _,
    exact (hF x).left,
  }, {
    simp *,
    rw ←coe_inj,
    rw coe_image,
    rw coe_image,
    apply eq_of_subset_of_subset, {
      simp *,
      intros x hx,
      have hF_x := hF x,
      simp *,
      rw finset.mem_coe at hx,
      rw finset.mem_subtype at hx,
      exact hx,
    }, {
```

```
      intros x h_x,
      simp *,
      have x_sub_fS := mem_of_subset_of_mem sub h_x,
      let a := FF ⟨ x, x_sub_fS ⟩,
      existsi a,
      split, {
        existsi x,
        apply and.intro h_x,
        existsi x_sub_fS,
        refl,
      }, {
        exact (hF ⟨ x, x_sub_fS ⟩).right,
      }
    },
  },
end
```