Mathematical project in L∃∀ℕ

An innocent mathematicians guide to lean

Andreas Bøgh Poulsen *201805425*

AARHUS
UNIVERSITET
INSTITUT FOR MATEMATIK

Supervisor: Niels Lauritzen

Contents

1 Martin-Löf dependent type theory **3**
- 1.1 Intuition and how to read the notation . . . 3
  - 1.1.1 Deduction rules . . . 4
- 1.2 Inference rules . . . 4
- 1.3 Logic in type theory . . . 6
- 1.4 The natural numbers . . . 9
- 1.5 Equality . . . 11
- 1.6 Higher order types . . . 14
- 1.7 Propositions as some types and universes . . . 15

2 Mathematics in type theory and Lean **17**
- 2.1 Learning Lean and the purpose of this section . . . 18
- 2.2 Vernaculars and user interface . . . 18
- 2.3 The library of mathematics, mathlib . . . 20
- 2.4 Abstract structures as type classes . . . 20
  - 2.4.1 Type classes for notation . . . 21
  - 2.4.2 Type aliases . . . 22
- 2.5 Well-founded recursion and well-orders . . . 23
  - 2.5.1 Well-founded recursion using the equation compiler . . . 26
- 2.6 Sets . . . 27
  - 2.6.1 Notation for sets and subtypes . . . 27

3 Gröbner bases as an extended example **28**
- 3.1 Dicksons lemma . . . 28
- 3.2 Multivariate polynomials and Dicksons lemma . . . 37
- 3.3 Initial terms and Gröbner bases . . . 39
- 3.4 The division algorithm . . . 42

A Inductively defined types and the induction principle **48**

B A monoid structure on `vector N n` **50**

C A preimage of a finite set **52**

D Lemmas about the initial term **53**

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.

The first section covers a variant of the type theory used in Lean. In it, I build a mathematical foundation, on which to base your understanding of Lean. The second section introduces Lean as a tool to assist the tedious work of doing formalized mathematics. Here, I also showcase some of the "engineering" aspects of Lean, which are needed to scale a formalism to the state of modern mathematics. In particular, I highlight the use of type classes, which will be needed in section 3. The third section is dedicated to walking through a worked example, which is the development of Gröbner bases for polynomial rings over fields. This is to show the work required to bring a modern algebraic theory into Lean. It will also serve as a reference example for others, who wish to formalize non-trivial mathematics in 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 section, I'll explain how dependent type theory works as a formal system. The development will largely follow the exposition of [3].

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 inductive constructions (which is used in Lean), but rather to develop a theory together, to see how and why the choices Lean has made, make sense. If you're only interested in learning Lean, feel free to skip this section.

1.1 Intuition and how to read the notation

The following development may seem very notation-heavy and needlessly abstract, so here's a little primer: Suppose I have three functions (or morphisms)

$$f : A \to B$$
$$g : A \to C$$
$$h : B \to C$$

The the composition $h \circ f : A \to C$ make sense, but the composition $h \circ g$ doesn't. Keeping track of domains and codomains in this small example is not a problem, but when more functions get involved, it can become unwieldy. This is where a type-checking compiler, as found in programming languages like Java and Haskell, can help.

Similarly, if we have three propositions

$$p_1 = P \implies Q$$
$$p_2 = P \implies R$$
$$p_3 = Q \implies R$$

have can deduce $P \implies R$ from p_1 and p_3, but we cannot deduce that from p_1 and p_2. This is similar to the situation above. In fact, we translate our propositions directly into functions and get a compiler to check it for us.

Now, most propositions are not fixed statements, they are parameterized by variables, for example $p_4 = x \leq 0$. Since the truth of this proposition is dependent on the element x, we can't translate it into any fixed type. So we extend the type theory of our programming language, so that types can depend on variables, then build a type-checker for this type theory, and then we get a checker for our propositions.

1.1.1 Deduction rules

The rules for our language are given by deduction rules. There are two types of deduction rules: typing rules and evaluation rules. The typing rules are only needed by the type checker. They are statements of the form *"if $f : A \to B$ and $g : B \to C$ are functions with given domains and codomains, then $g \circ f : A \to C$ is a function with the given domain and codomain."* Strictly speaking, this is all we need to formulate mathematics, as we only need the type checker. However, it will turn out to be useful to have an actual programming language. This means we need to define what $g \circ f$ does. This is given by an evaluation rule: *"if $a \in A$ is an element and $f : A \to B$ and $g : B \to C$ are functions with given domains and codomains, then $(g \circ f)(a) = g(f(a))$."* This tells us how to evaluate the terms we build. In the text below, rules that end in A Type or $a : A$ are typing rules, and rules like $f(x) \doteq y$ are evaluation rules.

1.2 Inference rules

An 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 context 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 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 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 *judgments*.

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. $\Gamma, a : A, b : A \vdash a \doteq b : A$ 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:

$$\frac{\Gamma \vdash B \ \textsc{Type}}{\Gamma, b : B \vdash b : B}$$

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:

$$\frac{\Gamma, x : A \vdash B(x) \ \textsc{Type}}{\Gamma \vdash \prod_{x:A} B(x) \ \textsc{Type}} \qquad \frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma \vdash \lambda x.\, b(x) : \prod_{x:A} B(x)}$$

and has the following evaluation rules:

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

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 $\prod_{a:A} B \to C$ is read as $\prod_{a:A} (A \to B)$.

1.3 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

$$\frac{}{\vdash \emptyset \ \textsc{Type}} \qquad \frac{}{\vdash ind_\emptyset : \prod_{x:\emptyset} P(x)}$$

and the unit type (true) is given by

$$\frac{}{\vdash \mathbf{1} \ \textsc{Type}} \qquad \frac{}{\vdash \bullet : \mathbf{1}} \qquad \frac{}{\vdash ind_{\mathbf{1}} : P(\bullet) \to \prod_{x:\mathbf{1}} P(x)}$$

Remark. The functions $ind_\emptyset$ and $ind_{\mathbf{1}}$ are called induction functions or induction rules.

They govern the behaviour of these and all our future types.

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. $(A \implies B) \implies (\neg B \implies \neg A)$

Proof. We construct an element of the desired type:

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

□

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

$$\frac{\Gamma \vdash A\ \textsc{Type} \qquad \Gamma \vdash B\ \textsc{Type}}{\Gamma \vdash A \vee B\ \textsc{Type}} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash B\ \textsc{Type}}{\Gamma \vdash \iota_1 : A \to A \vee B} \qquad \frac{\Gamma \vdash A\ \textsc{Type} \qquad \Gamma \vdash b : B}{\Gamma \vdash \iota_2 : B \to A \vee B}$$

The disjunction is equipped with the following induction function:

$$\frac{\Gamma \vdash A\ \textsc{Type} \qquad \Gamma \vdash B\ \textsc{Type}}{\Gamma \vdash ind_\vee : (\Pi_{a:A} P(\iota_1(a))) \to (\Pi_{b:B} P(\iota_2(b))) \to (\Pi_{z:A\vee B} P(z))}$$

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

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

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

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

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:

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

□

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.

1.4 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

$$\frac{}{\vdash \mathbb{N}\ \textsc{Type}} \qquad \frac{}{\vdash 0_{\mathbb{N}} : \mathbb{N}} \qquad \frac{}{\vdash succ_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}$$

has the following induction rule:

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

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

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

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:

```
type ⊢ ℕ
| 0ℕ : ℕ
```

$$\mid \ \mathrm{succ}_{\mathbb{N}} \ : \ \mathbb{N} \to \mathbb{N}$$

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

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

The derived induction principle is the cornerstone of the Calculus of Inductive Constructions, which is the logic that Lean is built on.[1] For a more thorough derivation of the induction principle, see section A in the appendix. We can similarly define

$$\begin{array}{l} \mathrm{type} \ \vdash \ \mathrm{A} \vee \mathrm{B} \\ \mid \ \iota_1 \ : \ \mathrm{A} \ \to \ \mathrm{A} \vee \mathrm{B} \\ \mid \ \iota_2 \ : \ \mathrm{B} \ \to \ \mathrm{A} \vee \mathrm{B} \end{array}$$

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:

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

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

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

[1] Not all type theories include automatically derived induction principles. In some cases, it clashes with additional structure, you may wish to put on your types. However, in most cases it's not an issue and the derived induction principle provides a principled way to develop the theory of mathematics.

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 $1 + 2 = 3$:

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

1.5 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

$$\begin{aligned} &\mathtt{type}\ (\mathtt{a}\ \mathtt{b}\ :\ \mathtt{A}) \vdash a =_A b \\ &\mid\ \mathtt{refl}\ :\ \Pi_{x:A}\ \mathtt{x}{=_A}\mathtt{x} \end{aligned}$$

with derived induction principle

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

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*

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

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 $a =_A x$:

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

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 ∘ 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) )
)
```

□

1.6 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 the element $\bullet$: 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 the 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.

1.7 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:

$$\begin{aligned}&\mathtt{type}\ \ (\mathtt{P}\ :\ \mathtt{A}\ \to\ \mathtt{Type})\ \vdash\ \Sigma_{a:A}P(a)\\ &|\ \ \mathtt{intro}_\Sigma\ :\ \Pi_{a:A}\,B(a)\to\Sigma_{a:A}B(a)\end{aligned}$$

with derived induction rule:

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

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*

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

Proof. We define the functions as follows:

$$\begin{aligned}&\mathtt{p}_1(\mathtt{z})\ :=\ \mathtt{ind}_\Sigma(\lambda a.\,\lambda x.\,a)\\ &\mathtt{p}_2(\mathtt{z})\ :=\ \mathtt{ind}_\Sigma(\lambda a.\,\lambda x.\,x)\end{aligned}$$

□

The projection functions encode the axiom of choice[2]. 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, as we've defined them, 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

[2] Or at least, they are equivalent to the axiom of choice. For more details, see section 2.6

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 $\mathit{Sort}\ n : \mathit{Sort}(1+n)$.

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:

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

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

Universes are closed under functions, so if $A : \mathit{Sort}\ n$ and $B : \mathit{Sort}\ m$ then $A \to B : \mathit{Sort}\ \max(n,m)$.

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 $P : \mathit{Prop}$.

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

```
type (n, m : ℕ) ⊢ n ≤ m : Prop
| zero_le : Π_{n:ℕ} 0 ≤ n
| succ_le : Π_{n,m:ℕ}(n ≤ m) → (succ(n)≤succ(m))
```

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)))
```

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 group 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.

The 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[3] 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 Learning Lean and the purpose of this section

This document is in no way a complete guide to theorem proving in Lean. To learn how to use Lean, I wholeheartedly recommend "The Natural Number Game"[1]. In it, you'll learn how to use tactic mode, how to structure a proof into lemmas and theorems, and how to apply previously proven facts in new proofs.

This document is supposed to be a reference on topics I found difficult, when doing this project. Thus, it contains notes on using type aliases to steer type class resolution, the mathematical underpinnings of well-founded recursion, and more. It will not cover every problem you encounter, but I hope it will help you, when you try to apply Lean to your field of mathematics.

2.2 Vernaculars and user interface

Lean is built on a very small core logic, called the Calculus of Inductive 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:

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

2.1 · **Definition**. The natural numbers in Lean

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

and we can define addition like before:

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 : ∀ n m:N, N_add n m = N_add m n
```

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.

2.3 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.

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 as type classes

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 the type `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 : group G) (g : G) : ℤ →* G` 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 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_left_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 ... 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. If we want to supply those arguments explicitly, we can write `@mul_right_inv ℚ (ℚ.group) 2 : mul 2 (inv 2) = 1`.

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.

2.4.1 Type classes for notation

Lean uses type classes to solve another problem: any given function can only have one type. Thus, if we defined `add : N → N`, we cannot later use `add` to mean addition on the rationals, reals or anything else. However, using type classes, we can define a class

```
@[class]
structure has_add (A : Type) :=
  (add : A → A → A)
```

```
infixl ` + `:65  := has_add.add
```

This defines a type class of types having a function defined on them called `add`. Below, we define `+` to be a left-associating infix operator, which means `has_add.add`. Thus `1 + 1 = has_add.add 1 1` by definition. The number `65` indicates how strongly the operator binds, to make addition bind weaker than f.ex. multiplication.

Similarly, Lean defines `has_mul (*)`, `has_le (≤)`, `has_subset (⊆)` and many more. Type classes can "extend" each other, so a group can be defined as something of class `has_mul`, for which the group axioms hold. Take a look at how `group` is defined in the mathlib. You might be surprised.

2.4.2 Type aliases

Since type classes are resolved automatically, sometimes you get an instance, that you don't want. Maybe you want the group structure on $\mathbb{Z}$ given by $m \star n := m+n+1$. What do you do then?

The answer is to define a type alias. This is a new name, that refer to the same type, like so:

```
def ℤ_alt := ℤ

def to_ℤ_alt : ℤ → ℤ_alt := id
def of_ℤ_alt : ℤ_alt → ℤ := id
```

Now, `ℤ_alt` does not have an instance of `group`, so you're free to define your own. But, since `ℤ` and `ℤ_alt` are definitionally equal, we inherit every function defined on `ℤ` already. For example:

```
instance : group ℤ_alt := {
  mul := λ m n, int.add (int.add m n) 1,
  one := to_ℤ_alt (-1),
  inv := λ m, to_ℤ_alt ((-1) + (-of_ℤ_alt m) + (-1)),
  ...
}
```

uses the addition already defined on `ℤ`. However, since `ℤ_alt` has no instances, we don't get the addition symbol. We can tell Lean to inherit some of the type classes from `ℤ` like so:

```
@[derive [has_add, has_one, has_neg]]
def ℤ_alt := ℤ

instance : group ℤ_alt := {
  mul := λ m n, m + n + 1,
  one := -1,
  inv := λ m, (-1) + (-m) + (-1),
  ...
}
```

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 either $(s_1,\dots,s_n) = (t_1,\dots,t_n)$ or $s_i = t_i$ for $i = 1,\dots,k-1$ 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 $i = n$. □

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.

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
```

Note that because the relation `r` is given to the left of the colon in the first line, we don't have to supply it in the recursive references to `acc`. In a sense, it is fixed throughout the definition, so we don't need to repeat it.

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 $P\ y$ holds for every $y < x$ where y is accessible, 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 $A' = \emptyset$ 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), ℕ
| ⟨ 0   , n   ⟩ _ := n+1
| ⟨ m+1, 0   ⟩ h := h (m, 1) sorry
| ⟨ m+1, n+1 ⟩ h := h (m, (h (m+1, n) sorry)) sorry

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

Remark: `ℕ ×ₗ ℕ` denotes the product ordered by the lexicographic order. This is because the default order on the product is the pointwise order, which is not a total order. See section 2.4.2 about type aliases for more details.

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.

2.5.1 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
```

Sometimes, the well-order isn't obvious, or Lean may not be able to prove that the recursive calls decreases. In that case we can use `using_well_founded` to specify the well-order and the proof. For example for addition:

```
def add2 : ℕ × ℕ → ℕ
| ⟨0,   n⟩ := n
| ⟨m+1, n⟩ := add2 ⟨m, n⟩ + 1
  using_well_founded {
    rel_tac := λ _ _, `[exact {
      r := λM N, to_lex M < to_lex N,
      wf := prod.lex_wf nat.lt_wf nat.lt_wf,
    }],
    dec_tac := `[begin
      simp,
      rw prod.lex.lt_iff,
      simp,
      rw nat.lt_add_one_iff,
    end],
  }
```

2.6 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?

We can define sets in Lean. 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 contained 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` be a subtype of `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.

2.6.1 Notation for sets and subtypes

Lean defines special notation for sets, so our statements look a little more familiar. I haven't been able to find documentation for these notations, so here is a quick summary:

```
-- Let A : Type* and P : A → Prop
{x : A | P x} := P : set A
-- Let S : set A
{x ∈ S | P x} := {x : A | x ∈ S ∧ P x}
-- If you're looking for lemmas about sets of the form
-- {x ∈ S | P x}, they're called 'sep' in mathlib
-- F.ex. mem_sep_eq {α : Type u} {s t : set α} : {x ∈ s | x ∈ t} = s ∩
 ↪  t

-- Membership of a set is just the fact that the element
-- satisfyes the defining property:
x ∈ {a : A | P a} := {a : A | P a} x := P x

-- Subtypes also have special syntax:
{x // P x} := subtype P
```

If you have an element of a set, say `h : x ∈ {s ∈ S | P s}` and need to get `h' : P x`, you can do that as follows:

```
-- Since x ∈ {s ∈ S | P s} := {s ∈ S | P s} x := (x ∈ S) ∧ P x we can
 ↪  do:
h' := h.2
```

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. The full code for this example can be found at https://github.com/0708andreas/Lean-Gr-bner-bases

We'll proceed as follows: first, we prove Dicksons lemma, which will later enable us to prove that every ideal in a polynomial ring has a Gröbner basis. Then we define term orderings and prove that they are well-orders. Then we define Gröbner bases, and prove that they exist for all ideals. Finally, we define the multivariate division algorithm, and prove its basic properties. Notably, since I ran out of time, we do not prove that division by a Gröbner basis is independent of ordering, and we don't prove that a polynomial reduces to zero under division with a Gröbner basis if and only if it is in the ideal spanned by the basis.

3.1 Dicksons lemma

Dicksons lemma is a sort of well-ordering theorem for $\mathbb{N}^n$, but instead of finding a single least element, we find a finite collection of least elements.

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 B.

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 a bunch of lemmas and notations in order for the proof to resemble the human proof.

First, we define some notation:

```
def upper_set {n : ℕ} (v : finset (vector ℕ n)) : (set (vector ℕ n)) :=
  {s : vector ℕ n | ∃ (x s' : vector ℕ n) (H : s' ∈ v), s = x + s'}

def P {n : ℕ} (S : set (vector ℕ n)) (v : finset (vector ℕ n)) : Prop
 ↪  :=
    ↑v ⊆ S ∧
      S ⊆ upper_set v
```

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,
    existsi ∅,
    rw [h, coe_empty],
    simp,
  }
end
```

Second, there was a step where we used the axiom of choice, to 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 C.

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 that if $v \subset S$ is a finite subset of a set of vectors, such that $t(S) \subseteq U(t(v))$ and for any pair of vectors $s \in S$ and $x \in v$, the first entry of v is less than the first entry of s, then $S \subseteq U(v)$. This was skipped in the informal proof, but is not entirely trivial:

```
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), eq_comm, 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, {
        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.
  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 get the existence of
  -- v_i' and then use axiom of choice to pick it.
  let vi' := λ i, classical.some (n_ih ((@tail ℕ n.succ) '' (Si
   ↪  i))),
  -- Find a finite set v_i s.t. tail(v_i) = v_i'
  let vi := λ i, classical.some (single_preimage (Si i) (vi' i)
   ↪  (vector.tail) (classical.some_spec (n_ih ((@tail ℕ n.succ) ''
   ↪  (Si i))))).1),
  -- And prove that v_i ⊆ S_i ∧ S_i ⊆ upper_set v_i
  have vi_P : ∀ i, P (Si i) (vi i) := begin
    intro i,
    have P_v' := classical.some_spec (n_ih ((@tail ℕ n.succ) '' (Si
     ↪  i))),
    have P_v := classical.some_spec (single_preimage (Si i) (vi' i)
     ↪  (vector.tail) (classical.some_spec (n_ih ((@tail ℕ n.succ)
     ↪  '' (Si i))))).1),
    split,
    exact P_v.1,
    have vi'_eq_some : vi' i = some _ := rfl,
    rw ←vi'_eq_some at P_v',
    rw ←P_v.2 at P_v',
    refine lift_upperset i (Si i) (vi i) P_v'.2
      (λs hs, le_of_eq hs.2)
      (λs hs, le_of_eq (mem_of_subset_of_mem P_v.1 hs).2.symm),
  end,
  -- All that work lets us define the finite set V
```

```
      let V := v ∪ finset.bUnion (finset.univ) vi,
      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) := 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
```

□

However, multivariate polynomials in Lean are not using vectors to represent terms.

Instead, they fix a type σ and let terms be finitely supported functions from σ to $\mathbb{N}$. However, this is not a convenient setting to prove Dicksons lemma, since we need to do induction on the number of variables. There is no good way to remove and add elements from types, so we use vectors instead. Now, we need to translate the proposition into using sets of `σ →₀ ℕ` instead of sets of `vector n ℕ`. Note, that `→₀` is notation for "finitely supported functions."

The tool we use for this translation is monoid isomorphisms. If we have a function `f : (σ →₀ ℕ) → vector n ℕ` such that `f (v1 + v2) = (f v1) + (f v2)` and this `f` has an inverse `g : vector n ℕ → (σ →₀ ℕ)`, then we should be able to replace every occurence of `vector n ℕ` in `dickson` with `σ →₀ ℕ`.

Since nothing in this world is easy, there is no monoid isomorphism in mathlib between `σ →₀ ℕ` for finite σ and `vector n ℕ`. There is, however, a regular isomorphism between vectors and functions from the canonical type of n elements called `fin n`. We can extend this to be a monoid isomorphism like so:

```
theorem vector_N_equiv_fin_to_N (n : ℕ) :
  (vector ℕ n) ≃+ (fin n → ℕ) :=
  add_equiv.mk' (equiv.vector_equiv_fin ℕ n) (begin
    intros x y,
    unfold_coes,
    rw vector_equiv_to_fun_eq_nth,
    refine funext _,
    simp ,
  end)
```

Now, we can construct the following equivalences for finite σ of cardinality n:

```
(vector ℕ n) ≃+ (fin n → ℕ) ≃+ (fin n →₀ ℕ) ≃+ (σ →₀ ℕ)
```

Like so:

```
def linear_equiv_fun_on_finite {α M : Type*} [fintype α]
 ↪  [add_comm_monoid M] :
  (α →₀ M) ≃+ (α → M) :=
{ map_add' := λ f g, rfl,
  .. finsupp.equiv_fun_on_finite }

theorem vector_N_equiv_fin_fto_N (n : ℕ) :
  (vector ℕ n) ≃+ (fin n →₀ ℕ) :=
    add_equiv.trans (vector_N_equiv_fin_to_N n) (add_equiv.symm
     ↪  linear_equiv_fun_on_finite)

theorem vector_N_equiv_finite_fto_N {σ : Type*} [f : finite σ] :
  (vector ℕ (nat.card σ)) ≃+ (σ →₀ ℕ) := begin
    rw finite_iff_exists_equiv_fin at f,
    choose n hn using f,
    have e := classical.choice hn,
    rw nat.card_eq_of_equiv_fin e,
    have r := @finsupp.dom_congr (fin n) σ ℕ _ e.symm ,
```

```
    exact add_equiv.trans (vector_N_equiv_fin_fto_N n) r,
  end
```

And now we can translate Dicksons lemma using this:

```
def mv_upper_set {σ : Type*} [finite σ] (n : ℕ) (v : finset (σ →₀ ℕ)) :
↪  (set (σ →₀ ℕ)) :=
  {x : σ →₀ ℕ | ∃ (x' s : σ →₀ ℕ) (H : s ∈ v), x = x' + s}

theorem mv_dickson {σ : Type*} [decidable_eq σ] [finite σ] (S : set (σ
↪  →₀ ℕ)) :
  ∃ v : finset (σ →₀ ℕ), ↑v ⊆ S ∧ S ⊆ mv_upper_set (nat.card σ) v :=
  ↪  begin
    let n := nat.card σ,
    let e := @vector_N_equiv_finite_fto_N σ _,
    let S' : set (vector ℕ (nat.card σ)) := image (e.symm) S,
    have v' := dickson n S',
    cases v' with v' hv',
    let v : finset (σ →₀ ℕ) := finset.image (e) v',
    cases hv' with v'_sub_S' S'_sub_upper,
    existsi v,
    split, {
      intros x hx,
      rw [finset.mem_coe, finset.mem_image] at hx,
      rcases hx with ⟨ a, ⟨ ha, h⟩ ⟩,
      rw ←h,
      rw ←finset.mem_coe at ha,
      have hha := mem_of_mem_of_subset ha v'_sub_S',
      rw mem_image _ _ _ at hha,
      rcases hha with ⟨ b, ⟨ hb, h ⟩⟩,
      rw [←h, add_equiv.apply_symm_apply _ _],
      exact hb,
    }, {
      intros s hs,
      let a := e.symm s,
      have ha : a ∈ S' := set.mem_image_of_mem _ hs,
      have a_in_upper : a ∈ upper_set v' := mem_of_mem_of_subset ha
      ↪  S'_sub_upper,
      rcases a_in_upper with ⟨x, x', x'_in_v', h⟩,
      have e_h := congr (rfl : ↑e = ↑e) h,
      rw ←add_equiv.to_fun_eq_coe at e_h,
      rw e.map_add' _ _ at e_h,
      existsi [e.to_fun x, e.to_fun x', finset.mem_image_of_mem
      ↪  (e.to_fun) x'_in_v'],
      have s_eq := add_equiv.apply_symm_apply e s,
      rw ←add_equiv.to_fun_eq_coe at s_eq,
      rw s_eq at e_h,
      exact e_h,
    },
  end
```

3.2 Multivariate polynomials and Dicksons lemma

Now, let's introduce multivariate polynomials. In Lean, they are defined in the mathlib as:

```
def mv_polynomial (σ : Type*) (R : Type*) [comm_semiring R] :=
↪  add_monoid_algebra R (σ →₀ ℕ)
```

Here, `add_monoid_algebra` takes a semiring `R` and an additive monoid `G` and produces the type of multivariate polynomials with variables in `G`, i.e. formal sums of elements of `G` with coefficients from `R`. It also defines multiplication to be the convolution, which is exactly what we want.

The type `σ →₀ ℕ` is finitely supported functions from a type σ to ℕ. Thus, elements of σ acts as our variables, and the value in ℕ under this function indicates the power of that variable. Note, that σ doesn't have to be a finite type. In our cases, it will assume it to be finite.

Since we assume σ to be finite, we have a bijection between functions `σ →₀ ℕ` and vectors of ℕ with the same length as the cardinality of σ. In the mathematical discussions, I will use vectors to decribe terms. For example, if $\sigma = \{x, y, z\}$ then x^2y is denoted by $(2, 1, 0)$ in vector form and $\begin{pmatrix} x & y & z \\ 2 & 1 & 0 \end{pmatrix}$ in function form. These are all referred to as *terms*.

In order to define Gröbner bases, we need the notion of a term order:

3.3 · Definition. Let $n \in \mathbb{N}$. Then a term order $\leq$ is a total order on $\mathbb{N}^n$ such that

1. For all terms $v, v_1, v_2 \in \mathbb{N}^n$ we have $v_1 \leq v_2 \implies v + v_1 \leq v + v2$
2. For all terms $v \in \mathbb{N}^n$ we have $0 \leq v$.

To carry this definition over to Lean, we need a type alias, see section 2.4.2, to avoid using the default pointwise ordering defined on functions.

```
def mv_term (σ : Type u) : Type u := (σ →₀ ℕ)
class term_order (σ : Type u) [finite σ] extends linear_order (mv_term
↪  σ) :=
  (additive : ∀ v v₁ v₂ : mv_term σ, v₁ ≤ v₂ → v + v₁ ≤ v + v₂)
  (zero_le : ∀ v : mv_term σ, 0 ≤ v)
```

Let's prove a simple lemma about term orders, that we're going to need later:

3.4 · Lemma. *Let $\leq$ be a term order and $v, v_1, v_2 \in \mathbb{N}^n$. Then $v \leq v_1 \implies v \leq v_1 + v_2$.*

Proof. Since $0 \leq v_2$ we have $v \leq v_1 = v_1 + 0 \leq v_1 + v_2$, so the conclusion follows by transitivity.

```
lemma weaken_le [t : term_order σ] (v v1 v2 : mv_term σ) :
v ≤ v1 → v ≤ (v1 + v2) := begin
  assume h,
```

```
  have v1_le_sum := term_order.additive v1 0 v2 (term_order.zero_le
  ↪  v2),
  rw add_monoid.add_zero _ at v1_le_sum,
  exact le_trans h v1_le_sum,
end
```

□

Now, we can formulate the main theorem of this section:

3.5 · Theorem. *Let $\leq$ be a term order. Then $\leq$ is a well-order*

Proof. Let $S \subseteq \mathbb{N}^n$ be a non-empty set of vectors. By Dicksons lemma (3.1) we can find a finite set $v \subseteq S$ such that $S \subseteq U(v) = \{s \in \mathbb{N}^n \mid \exists s' \in v, x \in \mathbb{N}^n : s = s' + x\}$.

Now, let $s_0 \in v$ be the smallest element of v with respect to $\leq$. Then s_0 is a lower bound for v. Furthermore, let $s \in S$. Since $s \in U(v)$ we can find $s' \in v$ and $x \in \mathbb{N}^n$ such that $s = s' + x$, thus $s' \leq s$. But since $s' \in v$ we have $s_0 \leq s'$ so $s_0 \leq s$. Thus s_0 is a least element of S.

In lean it looks like this:

```
lemma term_order_is_well_order [t : term_order σ] (S : set (mv_term σ))
↪  (h : S.nonempty) :
  (∃ t₀ ∈ S, ∀ t ∈ S, ¬ t < t₀) := begin
    have d := mv_dickson S,
    cases d with v hv,
    have v_nonempty : v.nonempty := begin
      have some_in_S := set.nonempty.some_mem h,
      have some_in_upper := mem_of_subset_of_mem hv.right some_in_S,
      rcases some_in_upper with ⟨ x, s0, hs0, _ ⟩,
      exact ⟨ s0, hs0 ⟩
    end,
    let s₀ := @min' _ t.to_linear_order v v_nonempty,
    have s0_in_v : s₀ ∈ v := finset.min'_mem v _,
    rw ←mem_coe at s0_in_v,
    have s0_in_S := mem_of_subset_of_mem hv.left s0_in_v,
    existsi s₀,
    existsi s0_in_S,
    intros s hs,
    have s_in_upper := mem_of_subset_of_mem hv.right hs,
    rcases s_in_upper with ⟨ w, ⟨ r, ⟨ hr, s_eq ⟩ ⟩ ⟩,
    rw s_eq,
    rw add_comm w r,
    apply not_lt_of_le,
    apply weaken_le,
    exact @min'_le _ t.to_linear_order v r hr,
  end
```

□

3.6 · Corollary. *Given er term-order $\leq$, the type* `mv_term σ` *has a well-founded order given by $<$, i.e. every element of* `mv_term σ` *is accessible w.r.t. $<$.*

Proof. Apply theorem 2.10 to theorem 3.5

```
instance [term_order σ] : has_well_founded (mv_term σ) := {
  r := (<),
  wf := begin
    rw well_founded.well_founded_iff_has_min,
    intros s hs,
    exact term_order_is_well_order s hs,
  end,
}
```

□

3.3 Initial terms and Gröbner bases

When dividing one polynomial by another, we want to start with the largest term. However, there is no natural way of defining the largest term in a multivariate polynomial. Term orders give us a way of comparing terms, which enables us to find the largest one.

3.7 · **Definition.** Let $\leq$ be a term order, and $f \in k[\sigma]$ be a non-zero multinomial over a field k. The inital term of f is the largest term of f under the order $\leq$, and is denoted by $\mathrm{IN}(f)$.

We denote by $\mathrm{LT}(f)$ the monomial ocurring as a summand in f with term $\mathrm{IN}(f)$.

In Lean:

```
def IN {σ : Type u} [decidable_eq σ] [finite σ] [term_order σ] (f :
↪  mv_polynomial σ R) : (mv_term σ) :=
  if h : (f = 0) then
    0
  else
    finset.max' f.support (mv_polynomial.non_empty_support_of_ne_zero f
    ↪  h)
```

Note, that we have defined `IN 0 := 0` even though usually we don't define the initial term of a polynomial with no terms. This is standard practice in Lean, to have the function defined on the entire domain. Similarly, the square root, `real.sqrt : ℝ → ℝ` is defined for negative reals, but theorems about square roots assume x to be positive. That way, you don't have to supply a proof that your number is positive every time you take a square root, only when you want to prove something about it.

When we do these kinds of "broadenings," it's convenient to have theorem stating the definition within the usual domain:

```
lemma IN_of_non_zero_eq {σ : Type u} [decidable_eq σ] [finite σ]
↪  [term_order σ] (f : mv_polynomial σ R) (h : f ≠ 0) :
  IN f = finset.max' f.support
  ↪  (mv_polynomial.non_empty_support_of_ne_zero f h):= begin
    rw IN,
```

```
    simp [h],
  end
```

Let's prove a quick lemma:

3.8 · Lemma. *For a polynomial $f \in k[\sigma]$, the initial term $\mathrm{IN}(f)$ is a term of f, i.e. it's coefficient is non-zero.*

Proof.

```
lemma coeff_IN_nonzero [term_order σ] (f : mv_polynomial σ R) (h : f ≠
↪  0) :
  coeff (IN f) f ≠ 0 := begin
    rw [←mv_polynomial.mem_support_iff, IN_of_non_zero_eq _ h],
    exact finset.max'_mem _ _,
  end
```

□

We're goin to need a lot more lemmas like this. They live in the appendix, section D

Now, we get to Gröbner bases.

3.9 · Definition. Let $I \subseteq k[\sigma]$ be an ideal in a polynomial ring and $\leq$ be a term order. A *Gröbner basis* for I with respect to $\leq$ is a finite set of non-zero polynomials $G = \{f_1, f_2, \ldots, f_n\} \subseteq I$ such that for any polynomial $f \in I$, there is an $f_i \in G$ with $\mathrm{LT}(f_i) \mid \mathrm{LT}(f)$.

In Lean:

```
def grobner_basis [term_order σ] (F : finset (mv_polynomial σ R)) (I :
↪  ideal (mv_polynomial σ R)) : Prop :=
  (∀f ∈ F, f ∈ I ∧ f ≠ (0:mv_polynomial σ R) ) ∧
  (∀ f ∈ I, f ≠ 0 → ∃ f' ∈ F, ((LT f') ∣ (LT f)))
```

Note that the Lean definition doesn't use finite sets, but instead represents a the Gröbner basis as a finite sequence. This mirrors the definitions naming of the polynomials as $\{f_1, \ldots, f_n\}$.

When we do the division algorithm, we'll be able to prove that a Gröbner basis spans the ideal I, but we can't do that yet. What we can prove, is that Gröbner bases exists for all ideals.

3.10 · Theorem. *Let $I \subseteq k[\sigma]$ be an ideal in a polynomial ring and $\leq$ be a term order. Then there exists a Gröbner basis $G \subseteq I$ for I with respect to $\leq$.*

Proof. Let $I \subseteq k[\sigma]$ be an ideal, and let $S = \mathrm{IN}(I \setminus \{0\})$ be the set of initial terms of the non-zero polynomials in I. Dicksons lemma gives us a finite set $v' \subseteq S$ such that

$$S \subseteq U(v') = \{x \in \mathbb{N}^n \mid \exists x' \in \mathbb{N}^n, s \in v' : x = x' + x\}$$

Now, let $v \subseteq I \setminus \{0\}$ be a finite set of non-zero polynomials such that $\mathrm{IN}(v) = v'$. I claim that v is a Gröbner basis of I.

Indeed, let $f \in I$ and let $x = \mathrm{IN}(f)$. Then there is some $f_i \in v$ where $x = x' + \mathrm{IN}(f_i)$ for some $x' \in \mathbb{N}^n$. But this means that $\mathrm{LT}(f_i) \mid \mathrm{LT}(f)$ since we work over a field. To see this, write $\mathrm{LT}(f) = c \cdot \mathrm{IN}(f)$ and $\mathrm{LT}(f_i) = c' \cdot \mathrm{IN}(f_i)$. Then

$$\begin{aligned}
\mathrm{LT}(f_i) \times \left(\frac{c}{c'} \cdot x \right) &= (c' \cdot \mathrm{IN}(f_i)) \times \left(\frac{c}{c'} \cdot x \right) \\
&= \left(c' \frac{c}{c'} \right) \cdot (f_i + x) \\
&= c \cdot \mathrm{IN}(f) \\
&= \mathrm{LT}(f)
\end{aligned}$$

□

Thus, G is a Gröbner basis for I.

In Lean, the proof looks like this:

```
theorem exists_grobner_basis [term_order σ] (I : ideal (mv_polynomial σ
↪  R)) :
  ∃ F : finset (mv_polynomial σ R), grobner_basis F I := begin
    let SI := {f : mv_polynomial σ R | f ∈ I ∧ f ≠ 0 },
    let S := IN '' SI,
    have exi_V := mv_dickson S,
    rcases exi_V with ⟨ V, V_sub_S, S_sub_upper ⟩,
    let Vf := single_preimage SI V IN V_sub_S,
    choose Vf H using Vf,
    existsi Vf,
    split, {
      intros f hf,
      have f_in_SI := mem_of_subset_of_mem H.left hf,
      exact f_in_SI,
    }, {
      intros f hf f_ne_0,
      have f_in_SI : f ∈ SI := ⟨ hf, f_ne_0⟩,
      have in_f_in_S : IN(f) ∈ S := mem_image_of_mem IN f_in_SI,
      have in_f_in_upper := mem_of_subset_of_mem S_sub_upper in_f_in_S,
      cases in_f_in_upper with x hx,
      rcases hx with ⟨ s, ⟨ hs, f_eq⟩  ⟩,
      rw [←H.right, finset.mem_image] at hs,
      rcases hs with ⟨ f', ⟨ hf', f'_eq ⟩⟩,
      existsi f',
      split, {
        exact hf',
      }, {
        have f'_in_SI : f' ∈ SI := mem_of_subset_of_mem H.left hf',
        have f'_ne_zero : f' ≠ 0 := f'_in_SI.right,
        existsi (monomial x ((coeff (IN f) f)/(coeff (IN f') f'))),
        rw [LT, LT, monomial_mul],

        rw div_eq_inv_mul,
        rw mul_inv_cancel_left₀ (coeff_IN_nonzero f' f'_ne_zero),
        simp *,
        conv in (x + s) { rw add_comm_monoid.add_comm,},
```

```
    }
  },
end
```

3.4 The division algorithm

The division algorithm for multivariate polynomials is a generalization of the usual division algorithm for polynomials in a single variable. The algorithm is captured in the proof of the following proposition.

3.11 · Proposition. *Let $f \in k[x_1, \ldots, x_n]$ be a polynomial over a field k, let $\leq$ be a term order and let $F_{1,\ldots,F_m} \in k[x_1, \ldots, x_n]$ be a set of non-zero polynomials.*

Then there exists a set of polynomials $h_1, \ldots, h_n, r \in k[x_1, \ldots, x_n]$ such that

$$f = h_1 F_1 + \cdots + h_m F_m + r$$

where either $r = 0$ or no term of r is divisible by $\mathrm{LT}(F_i)$ for any $i \in \{1, \ldots, m\}$.

Proof. To start, set $s \leftarrow f$ and set $r \leftarrow, h_1 \leftarrow 0 \ldots, h_m \leftarrow 0$. If $s = 0$ we are done. If not, consider $\mathrm{IN}(s)$.

- If there exists an i such that $\mathrm{IN}(F_i) \mid \mathrm{IN}(s)$, choose the smallest such i and set

$$s \leftarrow s - \frac{\mathrm{IN}(s)}{\mathrm{IN}(F_i)} F_i$$

$$h_i \leftarrow h_i + \frac{\mathrm{IN}(s)}{\mathrm{IN}(F_i)}$$

- Otherwise set

$$s \leftarrow s - \mathrm{IN}(s)$$

$$r \leftarrow r + \mathrm{IN}(s)$$

Repeat this procedure until $s = 0$. Since the initial term of s always strictly decreases, this will terminate within a finite number of steps, pr. lemma 3.5 and proposition 2.6

Each step maintains the invariant that

$$f = h_1 F_1 + \cdots + h_m F_m + r + s$$

and at the end we have $s = 0$, so we get

$$f = h_1 F_1 + \cdots + h_m F_m + r.$$

Furthermore, we only add terms to r that are not divisible by any intial term of any F_i, so the last assertion holds as well. □

In Lean, this proposition is split into two parts: first is a function, that computes the h_i and r. Second is the proof that the results of this function satisfies the properties above. First we define a function, that performs a single step in the division algorithm:

```
def mv_div_step {n : ℕ} (f : mv_polynomial σ R) (F : fin n →
 ↪  mv_polynomial σ R)
                        (a : fin n → mv_polynomial σ R) (r :
                         ↪  mv_polynomial σ R)
                        (s : mv_polynomial σ R) :
  (fin n → mv_polynomial σ R) × mv_polynomial σ R × mv_polynomial σ R
   ↪  :=
--          a                          r                   s
  if h : (∃ i:fin n, (LT (F i)) | (LT s)) then
    let i := fin_find' (λi, (LT (F i)) | (LT s)) h in
      (function.update a i ((a i) + monomial_div s (F i)),
       r,
       s - (monomial_div s (F i))*(F i))
  else
    (a, r + (LT s), s - (LT s))
```

In the above we use the function `monomial_div`. It is defined as follows:

```
def monomial_div (g f: mv_polynomial σ R) : mv_polynomial σ R :=
  monomial ((IN g) - (IN f)) ((g.coeff (IN g))/(f.coeff (IN f)))
```

Now, before we can define the actual algorithm, there is one more technical challenge we need to overcome: if the polynomial s is a constant polynomial, then $\mathrm{IN}(s) = 0$. In this case, the initial term doesn't decrease in the recursive step, since in the next step we will get to $s = 0$ and $\mathrm{IN}(0) = 0$. It's not a problem, because in that case the next step of the algorithm will be the last, but Lean doesn't know that.

We fix this by defining a variant of the inital term: `IN' s : with_bot (mv_term σ)`, which is equal to `IN s` when $s \neq 0$, but is a special value `⊥` when $s = 0$. We can extend the ordering, so that `⊥ < t` for all terms t. This is still a well-order, and now we can use that `IN'` strictly decreases in every step.

3.12 · Definition. Alternative version of IN.

```
def IN' {σ : Type u} [decidable_eq σ] [finite σ] [term_order σ] (f :
 ↪  mv_polynomial σ R) : with_bot (mv_term σ) :=
  if h : f = 0 then
    ⊥
  else
    ↑(IN f)
```

3.13 · Lemma. *The domain of* `IN'` *(i.e.* `with_bot (mv_term σ)`*) is well-ordered*

Proof.

```
instance [term_order σ] : has_well_founded (with_bot (mv_term σ)) := {
  r := linear_order.lt,
  wf := with_bot.well_founded_lt (has_well_founded.wf),
}
```

□

And now we can finally state the division algorithm:

```
noncomputable def mv_div_aux {n : ℕ} (f : mv_polynomial σ R)
  (F : fin n → mv_polynomial σ R) :
  (fin n → mv_polynomial σ R) ×
  (mv_polynomial σ R) × (mv_polynomial σ R) →
  (fin n → mv_polynomial σ R) × (mv_polynomial σ R) × (mv_polynomial σ
  ↪ R)
| ⟨ a, r, s ⟩  :=
  if hs : s = 0 then
    (a, r, s)
  else
    have (IN' (mv_div_step f F a r s).snd.snd) < (IN' s),
          from mv_div_step_decreases f F a r s hs,
    (mv_div_aux (mv_div_step f F a r s))

  using_well_founded {
    rel_tac := λ _ _, `[exact {
      r := λ N M, IN' N.snd.snd < IN' M.snd.snd,
      wf := (inv_image.is_well_founded _ _).wf
    }],
    dec_tac := `[exact this],
  }
```

The structure of this definition is as follows: given `(a, r, s)` we check if `s=0`. If so, we are done, and simply return the input. If not, we first prove that the recursive call we're about to do decreases in the third argument with respect to `IN'`. Note, a tuple `(a, r, s)` is actually `(a, (r, s))`, so the third entry is the second entry of the second entry, hence the `(...).snd.snd`. The proof of this is delegated to the lemma `mv_div_step_decreases`. Finally, we perform a single step of the division algorithm, and call ourselves recursively.

After the main body, there is a `using_well_founded` block, where we tell Lean which well-order relation to prove. We use the lemma `inv_image.is_well_founded`, which states that if we have a function $f : \alpha \to \beta$ and β is well-ordered, then we get a well-order[4] on α given by $a_1 < a_2 \iff f(a_1) < f(a_2)$. We use this, because the input to the function is of type `(fin n → mv_polynomial σ R) × (mv_polynomial σ R) × (mv_polynomial σ R)` so we use `λ N, IN' N.snd.snd` to get into `with_bot (mv_term σ)` where we have a well-order. Finally, we tell `dec_tac` to use the proof that we gave above, to show that the recursive call decreases.

Let's take a moment to orient ourselves. We identified some technical problems, that we needed to fix, before we could state the division algorithm. Then we wrote down a single step of the algorith. Finally, we wrote the recursive algorithm, that performs the step until we're finished, and proved that it terminates using well-foundedness. We

[4]Technically, we get a well-founded relation, but that relation is not an order. Well-foundedness still works, though.

didn't *actually* prove that the recursive call decreases, I'll let you read the proof on your own if you want to.

So now, we need to prove that the algorithm behaves the way we want it to. Just to get started, let's show something very simple: when `mv_div_aux` terminates, `s = 0`. You might just look at it and say "obviously, in the base case we have `s = 0`, and that's the only thing that matters," but nothing is that simple in Lean. Instead, we prove it inductively: we show that `s = 0` in the base case, and that it is preserved in the recursive steps.

```
lemma mv_div_aux_s_eq_zero {n : ℕ} (f : mv_polynomial σ R)
  (F : fin n → mv_polynomial σ R) :
  ∏N:(fin n → mv_polynomial σ R) ×
  (mv_polynomial σ R) × (mv_polynomial σ R),
  (mv_div_aux f F N).snd.snd = 0
| ⟨ a, r, s ⟩ :=
  if hs : s = 0 then
    begin
      rw mv_div_aux, simp [hs],
    end
  else
    have (IN' (mv_div_step f F a r s).snd.snd) < (IN' s),
          from mv_div_step_decreases f F a r s hs,
    begin
      rw mv_div_aux, simp [hs],
      exact mv_div_aux_s_eq_zero (mv_div_step f F a r s),
    end
    using_well_founded {
      rel_tac := λ _ _, `[exact {
        r := λN M, IN' N.snd.snd < IN' M.snd.snd,
        wf := (inv_image.is_well_founded _ _).wf,
      }],
      dec_tac := `[exact this],
    }
```

The proof is not dificult, it is just a matter of unfolding definitions and checking that it is true, but it shows how we do proofs about recursive functions.

Now, let's prove that we have

$$f = h_1 F_1 + \cdots + h_m F_m + r + s$$

First, we prove that if the equality already holds for some $h_1, \ldots, h_m, r, s$, then it still holds after applying `mv_div_step`:

```
lemma mv_div_step_inv1 {n : ℕ}
  (f : mv_polynomial σ R) (F : fin n → mv_polynomial σ R)
  (a : fin n → mv_polynomial σ R) (r : mv_polynomial σ R)
  (s : mv_polynomial σ R)
  (h : f = (∑ i, (a i)*(F i)) + r + s) :
  (f = ∑ i, ((mv_div_step f F a r s).fst i)*(F i) +
```

```
        (mv_div_step f F a r s).snd.fst +
        (mv_div_step f F a r s).snd.snd ) :=
```

The proof is long and tedious, so look it up if you want to. We're just going to use it:

```
lemma mv_div_aux_spec1 {n : ℕ} (f : mv_polynomial σ R)
  (F : fin n → mv_polynomial σ R) :
  Π (N:(fin n → mv_polynomial σ R) ×
  (mv_polynomial σ R) × (mv_polynomial σ R))
  (h : f = (∑i, (N.fst i) * (F i)) + N.snd.fst + N.snd.snd ),
  f = (∑i, (mv_div_aux f F N).fst i * F i) + (mv_div_aux f F N).snd.fst
   ↪  + (mv_div_aux f F N).snd.snd
| ⟨ a, r, s ⟩ h :=
  if hs : s = 0 then
    begin
      rw [mv_div_aux],
      simp [hs, h],
    end
  else
    have (IN' (mv_div_step f F a r s).snd.snd) < (IN' s),
      from mv_div_step_decreases f F a r s hs,
    begin
      rw [mv_div_aux],
      simp [hs, h],
      simp at h,
      rw ←h,
      refine mv_div_aux_spec1 (mv_div_step f F a r s) _,
      refine mv_div_step_inv1 f F a r s h,
    end
    using_well_founded {
      rel_tac := λ _ _, `[exact {
        r := λN M, IN' N.fst.snd.snd < IN' M.fst.snd.snd,
        wf := (inv_image.is_well_founded _ _).wf,
      }],
      dec_tac := `[exact this],
    }
```

And similarly we can prove that the remainder is either zero or indivisible in any term by any leading term of the F_i's.

Finally, we can wrap it all into a neat little package:

```
def mv_div {n : ℕ} (f : mv_polynomial σ R) (F : fin n → (mv_polynomial
 ↪  σ R)) :
  (fin n → (mv_polynomial σ R)) × (mv_polynomial σ R) :=
  ((mv_div_aux f F (λ_, 0, 0, f)).fst, (mv_div_aux f F (λ_, 0, 0,
   ↪  f)).snd.fst)
def mv_div_q {n : ℕ} (f : mv_polynomial σ R) (F : fin n →
 ↪  (mv_polynomial σ R)) :
  (fin n → mv_polynomial σ R) := (mv_div f F).fst
def mv_div_r {n : ℕ} (f : mv_polynomial σ R) (F : fin n →
 ↪  (mv_polynomial σ R)) :
  (mv_polynomial σ R) := (mv_div f F).snd
```

where `mv_div_r` gives the remainder and `mv_div_q` gives the quotients. Similarly, we can translate the `spec` lemmas:

```
theorem mv_div_spec1 {n : ℕ} (f : mv_polynomial σ R) (F : fin n →
↪  (mv_polynomial σ R)) :
  f = (∑ m : fin n, ((mv_div_q f F) m * (F m))) + (mv_div_r f F) :=
  begin
    rw [mv_div_q, mv_div_r, mv_div],
    simp,
    have C := (mv_div_aux_spec1 f F (λ (_x : fin n), 0, 0, f) begin
    ↪  simp, end),
    have s_eq_0 := (mv_div_aux_s_eq_zero f F (λ (_x : fin n), 0, 0,
    ↪  f)),
    rw [s_eq_0, add_zero] at C,
    exact C,
  end
theorem mv_div_spec2 {n : ℕ} (f : mv_polynomial σ R) (F : fin n →
↪  (mv_polynomial σ R)) :
  (mv_div_r f F) = 0 ∨ ∀ (m : fin n) (c ∈ (mv_div_r f F).support), ¬ LT
  ↪  (F m) | monomial c 1 :=
  begin
    rw [mv_div_r, mv_div],
    have C := (mv_div_aux_spec2 f F (λ (_x : fin n), 0, 0, f) begin
    ↪  simp, end),
    simp at C,
    simp,
    exact C,
  end
```

And that concludes our journey, thank you for coming along. Of course, a lot was left out, but most of it is tedious detail. I hope this large example has demonstrated the ideas about well-founded recursion, type aliases and set theory I talked about in the previous section.

References

[1] Kevin Buzzard **and** Mohammad Pedramfar. *The Natural Number Game*. https://www.ma.imperial.ac.uk/~buzzard/xena/natural_number_game/.

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

[3] Egbert Rijke. *Introduction to Homotopy Type Theory*. 2022. arXiv: 2212.11082 [math.LO].

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 leaves 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 equality:

```
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_1} {a : α} {P : α → Sort u_2},
  (α → P a) → ∀ {b : α}, eq a b → P b
```

B 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
@[simp]
lemma add_nth {n : ℕ} {i : fin n} (v1 v2 : vector ℕ n) :
  (v1 + v2).nth i = v1.nth i + v2.nth i := begin
    rw add_eq_zip_add v1 v2,
    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, eq_cons_iff],
    split, {
      rw [zip_with_head, cons_head, cons_head],
    },{
      rw [zip_with_tail, cons_tail, 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, 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
```

C 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 $f : \alpha \to \beta$, a set of elements from α, S and a finite set $v' \subseteq f(S)$. Then we find a finite set v 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
```

D Lemmas about the initial term

Note: this part of the proof is not completed. To complete the proofs, we can use `admit` in some places.

```
lemma IN_mem_support [term_order σ] (f : mv_polynomial σ R) (hf : f ≠
 ↪ 0) :
IN f ∈ f.support := begin
  rw IN,
  simp only [hf, not_false_iff, dif_neg],
  exact finset.max'_mem _ _,
end
@[simp]
lemma IN_neg [term_order σ] (f : mv_polynomial σ R) : IN (-f) = IN f :=
 ↪ begin
  rw [IN, IN],
  by_cases f = 0, {
    have h' : -f = 0 := by simp [h],
    simp [h, h'],
  }, {
    have h' : ¬(-f = 0) := by simp [h],
    simp [h, h'],
  }
end
lemma IN_add_le_max [term_order σ] (f g : mv_polynomial σ R) (hf : f ≠
 ↪ 0) (hg : g ≠ 0) :
```

```
IN (f+g) ≤ max (IN f) (IN g) := begin
  simp,
  by_cases h : f+g = 0, {
    rw h,
    left,
    exact term_order.zero_le _,
  }, {
    have IN_fg : IN (f+g) ∈ (f+g).support := IN_mem_support _ (h),
    have IN_fug := finset.mem_of_subset mv_polynomial.support_add
     ↪  IN_fg,
    rw finset.mem_union at IN_fug,
    cases IN_fug, {
      left,
      conv in (IN f) {rw IN,},
      simp [hf],
      exact finset.le_max' _ _ IN_fug,
    }, {
      right,
      conv in (IN g) {rw IN,},
      simp [hg],
      exact finset.le_max' _ _ IN_fug,
    }
  }
end
@[simp]
lemma IN_zero [term_order σ] : IN (0 : mv_polynomial σ R) = 0 := begin
  rw IN,
  simp,
end
@[simp]
lemma LT_zero [term_order σ] : LT (0 : mv_polynomial σ R) = 0 := begin
  rw LT,
  simp,
end
@[simp]
lemma IN_monomial [term_order σ] (s : σ →₀ ℕ) (c : R) (hc : c ≠ 0) : IN
 ↪  (monomial s c) = s := begin
  rw IN,
  simp [hc],
  conv in (((monomial s) c).support) {rw support_monomial},
  simp [hc],
end
@[simp]
lemma IN_mul [term_order σ] (f : mv_polynomial σ R) (hf : f ≠ 0) (s : σ
 ↪  →₀ ℕ) (c : R) (hc : c ≠ 0) :
IN ((monomial s c)*f) = IN (monomial s c) + IN f := begin
  rw [IN_monomial _ _ hc, IN],
  have H : monomial s c * f ≠ 0 := begin
    intro hX,
    rw ←zero_mul f at hX,
    have hX' := is_domain.mul_right_cancel_of_ne_zero hf hX,
    rw monomial_eq_zero at hX',
    exact hc hX',
```

```
  end,
  simp [H],
  conv in (monomial s c * f) {rw mv_polynomial.mul_def},
  simp [H],
  apply eq_of_le_of_not_lt, {
    refine finset.max'_le _ _ _ _,
    intros y hy,
    have hy' := finset.mem_of_subset (finsupp.support_sum) hy,
    rw finset.mem_bUnion at hy',
    rcases hy' with ⟨i, hi, hy'⟩,
    have hy'' := finset.mem_of_subset (support_monomial_subset) hy',
    rw finset.mem_singleton at hy'',
    rw [IN, hy''],
    simp [hf],
    apply term_order.additive,
    exact finset.le_max' _ _ hi,
  }, {
    intro hX,
    rw finset.max'_lt_iff at hX,
    specialize hX (s + IN f),
    apply @eq.not_lt _ _ (s + IN f : mv_term σ) (s + IN f : mv_term σ)
     ↪  rfl,
    apply hX,
    rw [mv_polynomial.mem_support_iff, sum_def, coeff_sum],
    simp [hc],
    exact coeff_IN_nonzero f hf,
  }
end

lemma erase_IN' [term_order σ] (f s : mv_polynomial σ R) (hf : f ≠ 0)
 ↪  (hs : s ≠ 0) (hsf : s - f ≠ 0) (h : LT f = LT s) : IN (s - f) ≠ IN
 ↪  s:= begin
  suffices h2 : coeff (IN s) (s - f) = 0, {
    intro hX,
    have h2' := coeff_IN_nonzero (s - f) hsf,
    rw hX at h2',
    exact h2' h2,
  },
  rw [LT, LT, monomial_eq_monomial_iff] at h,
  rw [sub_eq_add_neg, coeff_add, coeff_neg],
  cases h, {
    cases h with hIN h_coeff,
    nth_rewrite 0 ←h_coeff,
    rw [hIN, add_neg_self],
  }, {
    exfalso,
    exact coeff_IN_nonzero f hf h.1,
  }
end

lemma nonzero_of_LT_nonzero [term_order σ] (f : mv_polynomial σ R) (h :
 ↪  LT f ≠ 0) : f ≠ 0 := begin
  intro hX,
```

```
  rw [hX, LT_zero] at h,
  exact h rfl,
end
lemma eq_zero_of_LT_eq_zero [term_order σ] (f : mv_polynomial σ R) (h :
 ↪  LT f = 0) : f = 0 := begin
  by_contra hX,
  rw [LT, monomial_eq_zero] at h,
  exact coeff_IN_nonzero f hX (h),
end
```