Parametric Gröbner bases

Geometry & applications

Andreas Bøgh Poulsen *201805425*

AARHUS
UNIVERSITET
INSTITUT FOR MATEMATIK

Supervisor: Niels Lauritzen

Contents

Introduction

1 Preliminaries

This project will assume familiarity with ring theory, multivariate polynomials over fields. A familiarity with Gröbner bases will be beneficial, but we will introduce the necesary notations and definitions. Let R be a Noetherian, commutative ring and $X = (x_1, x_2, \ldots, x_n)$ be an ordered collection of symbols. We denote the ring of polynomials in these variables $R[X]$. Given two (disjoint) sets of variables X and Y, we will use $R[X, Y]$ to mean $R[X \cup Y]$, which is naturally isomorphic to $R[X][Y]$. A monomial is a product of variables and a term is a monomial times a coefficient. We denote a monomial as X^v for some $v \in \mathbb{N}^n$.

1.1 · Definition (Monomial order, leading term). A *monomial order* is a total order $<$ on the set of monomials satisfying that $u < v \implies wu < wv$.

Given a monomial order $<$ and a polynomial $f \in R[X]$, the *leading term* of f is the term with the largest monomial w.r.t. $<$ and is denoted by $\mathrm{lt}_<(f)$. If $\mathrm{lt}_<(f) = a \cdot m$ for some monomial m and $a \in R$, then we denote $\mathrm{lm}_<(f) = m$ and $\mathrm{lc}_<(f) = a$. If $<$ is clear from context, it will be omitted.

These definitions naturally extend to sets of polynomials, so given a set of polynomials $F \subset k[X]$, we denote $\mathrm{lm}_<(F) := \{\mathrm{lm}_<(f) \mid f \in F\}$. The above definitions work over a general ring (and we will use that), for from here, we'll work over a field k. With this, we can give the definition of a Gröbner basis.

1.2 · Definition (Gröbner basis). Let $G \subset k[X]$ be a finite set of polynomials and $<$ be a monomial order. We say G is a *Gröbner basis* if $\langle \mathrm{lt}_<(G) \rangle = \mathrm{lt}_<(\langle G \rangle)$

2 Definitions and initial results

The purpose of this project is to study parametric Gröbner bases, so let's introduce those. The bare concept is rather simple.

2.1 · Definition (Parametric Gröbner basis). Let k, k_1 be fields, U and X be collections of variables and $F \subset k[U, X]$ be a finite set of polynomials. A *parametric Gröbner basis* is a finite set of polynomials $G \subset k[U, X]$ such that $\sigma(G)$ is a Gröbner basis of $\langle \sigma(F) \rangle$ for any ring homomorphism $\sigma : k[U] \to k_1$.

We call such a $\sigma : k[U] \to k_1$ a *specialization*. By the linearity of σ, all such ring homomorphisms can be characterized by their image of U. Thus, we can identify $\{\sigma : k[U] \to k_1 \mid \sigma \text{ is a ring hom.}\}$ with the affine space k_1^m when U has m elements. For $\alpha \in k_1^m$ we'll denote the corresponding map

$$\sigma_\alpha(u_i) = \alpha_i \quad \text{for } u_i \in U$$

extended linearly.

When we work with these parametric Gröbner bases, it will be more convenient to have a bit more information attached to them, namely which elements are required for which σ. Since σ is described by an $\alpha \in k_1^m$, we can restrict them using subsets of k_1^m.

2.2 · Definition (Vanishing sets & algebraic sets). Let E be a finite subset of $k[X]$. Then the *vanishing set* of E is $V(E) := \{v \in k^n \mid e(v) = 0 \ \ \forall e \in E\}$.

An *algebraic set* is a set of the form $V(E) \setminus V(N)$ for two finite subsets E and N of $k[X]$.

2.3 · Definition (Gröbner system). Let A be an algebraic set and $G \subset k[U,X]$ be a finite set. Then (A,G) is called a *segment of a Gröbner system* if $\sigma_\alpha(G)$ is a Gröbner basis of $\sigma_\alpha(\langle G\rangle)$ for all $\alpha \in A$. A set $\{(A_1,G_1),\dots,(A_t,G_t)\}$ is called a *Gröbner system* if each (A_i,G_i) is a segment of a Gröbner system.

2.4 · Example. *Let $X = \{x,y\}$ and $U = \{u\}$ and consider the polynomials $f(x,y,u) = ux^2+x$ and $g(x,y,u) = xy+1$. When $u \neq 0$, a Gröbner basis of $\langle f,g\rangle$ could be $(y-u, ux+1)$, whatever u may be.* TODO

Skriv om Kalkbrener

2.5 · Definition (Leading coefficient w.r.t. variables). Let $f \in k[U][X]$. Then the leading term of f is denoted $\mathrm{lt}_U(f)$, the leading coefficient is $\mathrm{lc}_U(f)$ and the leading monomial is $\mathrm{lm}_U(f)$. These notations are also used when $f \in k[U,X]$, just viewing f as a polynomial in $k[U][X]$.

Note that $\mathrm{lc}_U(f) \in k[U]$, i.e. the leading term is a polynomial in $k[U]$ times a monomial in X.

From this point, we assume that the monomial order on $k[U,X]$ satisfies $x > u$ for all $x \in X$ and $u \in U$. This monomial order restricts to a monomial order on $k[X]$, denoted by $<_X$. Note that this assumption is not too restrictive, as both the lexicographic, reverse lexicographic and graded versions of those satisfies this assumption.

2.6 · Lemma. *Let G be a Gröbner basis of an ideal $\langle F\rangle$ w.r.t. $<$, let $\alpha \in k_1^m$ and set $G_\alpha := \{\sigma_\alpha(g) \in G \mid \sigma_\alpha(\mathrm{lc}_U(g)) \neq 0\} = \{g_1,g_2,\dots,g_l\} \subset k_1[X]$. Then G_α is a Gröbner basis of the ideal $\langle\sigma_\alpha(F)\rangle$ w.r.t. $<_X$ if and only if $\sigma_\alpha(g)$ is reducible to 0 modulo G_α for every $g \in G$.*

Proof. To we written after Kalkbrener □

We will use this lemma in a slightly different formulation:

2.7 · Lemma. *Let $G = \{g_1,g_2,\dots,g_k\}$ be a Gröbner basis of an ideal $\langle F\rangle$ in $k[U,X]$ w.r.t $<$ and let $\alpha \in k_1^m$. If $\sigma_\alpha(\mathrm{lc}_U(g)) \neq 0$ for each $g \in G \setminus (G \cap k[U])$, then $\sigma_\alpha(G)$ is a Gröbner basis of $\langle\sigma_\alpha(F)\rangle$.*

Proof. Let $G_\alpha = \{\sigma_\alpha(g) \mid \sigma_\alpha(\mathrm{lc}_U(g)) \neq 0\}$. If there is any $g \in G$, such that $\sigma_\alpha(g) \in k_1 \setminus \{0\}$, then $g \in G \cap k[U]$ since $\sigma_\alpha(\mathrm{lc}_U(g)) \neq 0$ for all $g \in G \setminus K[U]$. Furthermore, since $g \in \langle F\rangle$, we get that $\langle\sigma_\alpha(F)\rangle = k_1[X]$ and $\sigma_\alpha(G)$ is a Gröbner basis.

If there is no such g, then $\alpha \in V(G \cap k[U])$. Take any $g \in G$. If $\sigma_\alpha(g) \in G_\alpha$, then $\mathrm{lt}(\sigma_\alpha(g)) = \sigma_\alpha(\mathrm{lc}_U(g)) \cdot \mathrm{lm}_U(g)$ since $x > u$ for all $x \in X$ and $u \in U$. Thus, $\sigma_\alpha(g)$ is reducible to 0 modulo G_α, since it's leading term is divisible by its own leading term. On the other hand, if $\sigma_\alpha(g) \notin G_\alpha$, then $\sigma_\alpha(g) = 0$, so is immediately reducible to zero. Thus $\sigma_\alpha(G)$ is a Gröbner basis of $\langle \sigma_\alpha(F) \rangle$ by lemma 2.6. □

With lemma 2.6 in mind, we can start constructing Gröbner systems. Let G be a reduced Gröbner basis of an ideal $\langle F \rangle \subset k[U, X]$, and let $H = \{\mathrm{lc}_U(g) \mid g \in G \setminus k[U]\}$. Then $\left(k_1^m \setminus \bigcup_{h \in H} V(h), G\right)$ is a segment of a Gröbner system. Thus, to make a Gröbner system, we need to find segments covering $\bigcup_{h \in H} V(h) = V(\mathrm{lcm}\{h \mid h \in H\})$.

If we take G to be a reduced Gröbner basis, then $h \notin \langle F \rangle$ for any $h \in H$ since then the corresponding leading term would be divisible by a leading term in G. This is not allowed when G is reduced. Hence, we can find a Gröbner basis G_1 of $F \cup \{h\}$, which will then form a segment $(k_1^m \setminus \bigcup_{h \in H_1} V(h), G_1)$ where $H_1 = \{\mathrm{lc}_U(g) \mid g \in G_1\}$.

3 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 [1].

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.

3.1 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.

3.1.1 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.

3.2 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.

3.1 · 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.

3.2 · 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:

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

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

and has the following evaluation rules:

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

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

3.3 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:

3.4 · Definition. The types of false and true.

The empty type (false) is given by

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

and the unit type (true) is given by

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

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:

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

Proof. We construct an element of the desired type:

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

□

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*:

3.6 · 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

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

Proof. We construct an element of the desired type:

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

□

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.

3.4 The natural numbers

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

3.8 · 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 **??** 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

3.9 · 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 $1 + 2 = 3$:

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

3.5 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:

3.10 · 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.

3.11 · 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

3.12 · 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:

3.13 · 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

3.14 · 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:

3.15 · 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:

3.16 · 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

3.17 · 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) )
)
```

□

3.6 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:

3.18 · 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.

3.7 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:

3.19 · 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:

3.20 · 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 **??**

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*.

3.21 · 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*.

3.22 · 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:

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

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

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:

3.24 · 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))
```

3.25 · 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)))
```

References

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