An Invitation to Type Theory: Addition is Commutative in Peano Arithmetic

Nico Beck

January 3, 2025

In this text I like to show a formal proof of the fact that addition is commutative in Peano arithmetic. The proof is written in and checked by the proof assistant lean. The logical system which lean uses is the dependent type theory called *calculus of inductive constructions*. The code is shown and explained below. We will start by talking a little bit about the formal system underlying lean.

1 The calculus of inductive constructions

The type theory of Lean manipulates formal syntactic expressions of the form $\Gamma \vdash \texttt{t : A}$. Here Γ is a *context*, t is a term and A is a type. The full expression is called a sequent. The meaning of the sequent is that in a context Γ we can judge the term t to be of type A. This is a common form of judgement in informal mathematics. For example consider the sentence: *If A is a set, a is an element of A and f is a function from A to A, then fa is an element of A.* In type theory this sentence would be translated into the derivable sequent

```
A : Type, a : A, f : A → A ⊢ f a : A
```

which is surprisingly close to the structure of the original sentence. Lean has a special type `Prop` of propositions. For example the following sequent is derivable.

```
N : Type, S : N → N, 0 : N ⊢ ∀n:N, S n + 0 = S n : Prop
```

It states that the term `∀n:N, S n + 0 = S n` is a proposition in the context written down in front of the $\vdash$ symbol. This does not mean that we have proven the proposition yet, only that we can judge the expression to be a proposition. Assume now we can derive two sequents $\Gamma \vdash \texttt{P : Prop}$ and $\Gamma \vdash \texttt{p : P}$. Then we interpret the second sequent as asserting that in the context Γ the term p is a proof of the proposition P. It is an important idea of modern type theory that proofs can be encoded as terms of the proposition which they prove. The derivation rules of the type theory[1] are such that constructing a term of a proposition P is essentially the same as writing down a formal proof of P. To convince you that this is true we will later expand some of the proof terms in the example into actual proof trees. The fact that proofs are terms has a lot of nice consequences:

- The deductive system needs only rules for manipulating typing judgements of the form $\Gamma \vdash \texttt{a : A}$ and nothing more. The core of the proof assistant Lean is not much more than a type checker. If you want to prove a proposition `P` then you write down `P:Prop` in your text-editor, and if you done everything correctly, then Lean will tell you that indeed

[1] I haven't told you any of them yet.

`P:Prop`. If you want to prove your proposition `P`, then you construct a term `p` of type `P` and write down `p:P` in your text editor. Lean only checks if the typing judgement `p:P` is indeed correct. All of this will make much more sense when we look at the example below.

- Proofs are first class citizens (in programming talk). They have the same status as other terms in the formalism.
- There are far reaching symmetries between the term forming rules for proofs of `P` where `P : Prop` and the term forming rules of `A` where `A : Type`. This symmetry goes under the name Curry-Howard isomorphism. In fact, the similarities are so big that the type theory of Lean sets `Prop = Type 0` and `Type = Type 1` and formulates rules for `Type i` with `i=0,1` in parallel. This cuts the number of derivation rules effectively in half. In classical logic the object language and the logic on top of it are two strictly separated syntactic layers. Not so in modern type theory. A type theory which merges logic and object language to an even stronger degree is homotopy type theory.
- If you do not like that proofs are terms, then you can think of the proof terms as of a clever encoding of proof trees or derivations in some other deductive system. After all, we can not really put a proof tree into an computer.

2 Object language and axioms of PA

Now it is time to show you the code. We start by defining the language of Peano arithmetic. The code is fairly self-explanatory.

```
constant N : Type
constant S : N → N
constant 0 : N
constant plus : N → N → N
constant mul : N → N → N

infix ` + `:65 := plus
infix ` · `:75 := mul
```

The commands at the end allow us to write `n + m` instead of `+ n m`. Similar as in functional programming it is usually easier the work with the curried version of functions out of a product. By adjunction a term of type `N × N → N` naturally corresponds to a term of type `N → (N → N)`, but it is easier to work with terms of the second type. When we write `constant name : A`, then Lean just accepts the type judgement and uses it in the rest of the file. Lean does not accept nonsense such as `constant c : ∀+)gfwqf∀`. Next we like to add the axioms of Peano arithmetic. An axiom is a proof of a proposition which doesn't need to be checked and is not a composite of other proofs. Hence it makes sense to use again the `constant` command to force Lean to accept our axioms in the rest of the file without questioning them.

```
constant ax1 : ∀n:N,          ¬(S n = n)
constant ax2 : ∀n:N, ∀m:N,   (S n = S m → n = m)
constant ax3 : ∀n:N,           n + 0 = n
constant ax4 : ∀n:N, ∀m:N,    n + S m = S(n + m)
constant ax5 : ∀n:N,           n · 0 = 0
constant ax6 : ∀n:N, ∀m:N,    n · (S m) = n · m + n
constant ax7 : ∀P:N → Prop, (P 0) → (∀n:N, P n → P (S n)) → ∀n:N, P n
```

These are all the axioms. The first six axioms are fairly self-explanatory. The only one which we have to discuss a little bit more is the last axiom, which encodes induction. In Peano arithmetic the seventh axiom is actually an axiom scheme, that is a method to write down infinitely many axioms at once. For each well formed predicate `P` on `N` there is an induction axiom relative to `P`. Clearly we can not write down infinitely many axioms in a computer file. The best fix in Lean is to use higher order logic. We quantify over the type `N → Prop` of predicates on `N`. The only thing you have to understand is that whenever we can derive `Γ ⊢ P : N → Prop`, then Lean will accept the following judgement.

```
Γ ⊢ ax7 P : P 0 → (∀n:N, P n → P(S n)) → ∀n:N, P n
```

So in some sense `ax7` is also an axiom scheme. This is the only instance of higher order logic which we will use.

3 How does proving work?

It is time to look at some code. We work from the end of the proof to its beginning, because that is the logical order in which one would construct the proof. In a correct lean file the lemmas need of course be in the opposite order. Here is our main theorem and its proof.

```
theorem comm : ∀n:N, ∀m:N, n + m = m + n :=
  λn:N, ax7 (λm:N, n + m = m + n) (comm_start n) (comm_step n)
```

I will now explain the proof term step by step. We proof the statement by induction on `m` and for this reason we use the axiom seven. The expression `comm_start n` and `comm_step n` are proofs of the induction start and induction step relative to a fixed `n`. We will have to construct them below. Let us for the moment just assume that the judgements below are true, and work from there.

```
n : N ⊢ comm_start n : n + 0 = 0 + n
n : N ⊢ comm_step n : (n + m = m + n) → (n + S m = S m + n)
```

Remember that axiom seven is an axiom scheme. To use it we need to feed it with a predicate on `N` first. The following judgement is definitely true.

```
n : N, m : N ⊢ n + m = m + n : Prop
```

What this means is that given an arbitrary term of `N` we can construct a proposition by substituting the term for `m`. In type theory we think of a predicate `N → Prop` as a method for turning terms of `N` into terms of `Prop`. We have just described such a method, and the rules of type theory give us notation for it. We are allowed to make the following judgement.

```
n : N ⊢ λm:N, n + m = m + n : N → Prop
```

The variable `m` is now no longer free in the expression following it. It is bounded. You may think of the λ-abstraction as of the type theorists version of the $\mapsto$ symbol in mathematics. A mathematician will write $f : \mathbb{R} \to \mathbb{R}$, $x \mapsto x^2$ to denote a function. The idea is that if the mathematician has any particular real number, say 3, then she will judge that $f(3) = 3^2$. Note that she has obtained an expression for $f(3)$ by substituting 3 for the variable x in x^2. A computer scientist denotes the function f by `λx:ℝ, x·x : ℝ → ℝ`. The application rule for the term is the same. We have that `(λx:ℝ, x·x) 3 = 3·3`. This rule is called β-reduction. It is one of the rules which add dynamics to a type theory and make it a programming language. In fact, the equality above is even stronger than propositional equality and we should in principle

not use the same sign for it. A proposition is something which you have to provide a proof for, and whether you can or not depends on the rules of the surrounding logic. The equality of the β-reduction is closer to being a syntactic truth[2]. The system of lean makes internally no difference between the two expressions `(λx:ℝ, x·x) 2` and the expression 2·2. They are equal for all purposes and Lean won't even notice when you replace one by the other anywhere in your code. They are automatically reduced to some kind of normal form in the kernel of Lean[3]. This helps us to understand how the predicate we have just constructed works. Say we have any term of type `N`, for example `S S 0`. Then by β-reduction we treat the terms `(λm:N, n + m = m + n) (S S 0)` and `n + S S 0 = S S 0 + n` as equal for all purposes.

We now have a predicate `λm:N, n + m = m + n : N → Prop` to which we can apply the axiom scheme of induction. This means we can judge again. The judgement below is fairly long, because we have obtained it from the axiom scheme by replacing all occurrences of the variable `P` by our particular predicate.

```
n : N ⊢ ax7 (λm:N, n + m = m + n) :
        (n + 0 = 0 + n) → ∀m:N, (n + m = m + n → n + S m = S m + n)
        → ∀m:N, n + m = m + n
```

It is the last term of which we want to get a proof. Thus it is now clear how we can proceed. We only have to feed the term above a proof of the induction start and then of the induction step. We can make the following judgements.

```
n : N ⊢ ax7 (λm:N, n + m = m + n) (comm_start n) :
        ∀m:N, (n + m = m + n → n + S m = S m + n)
        → ∀m:N, n + m = m + n

n : N ⊢ ax7 (λm:N, n + m = m + n) (comm_start n) (comm_step n) :
          ∀m:N, n + m = m + n
```

Note that the proof term in the last line general method for turning a term `t` of type `N` into a term of type

```
(∀m:N, n + m = m + n)[t/n] = (∀m:N, t + m = m + t)
```

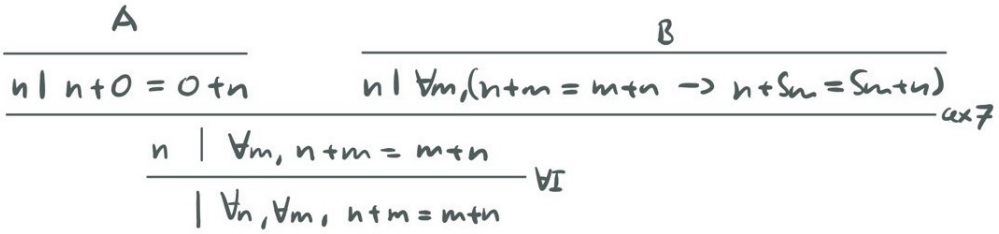
For this reason lean allows us to bind the free variable `n` on the right hand side, and we can make the following judgement.

```
⊢ λn:N, ax7 (λm:N, n + m = m + n) (comm_start n) (comm_step n) : ∀n:N, ∀m:
  N, n + m = m + n
```

I have now explained the proof term of our main theorem. What remains is to show you the proof terms of the induction step and induction start. Also for completeness, here is a tree which represents the proof of the main theorem. I have used a different notation to divide context and statement, and I have not typed the expressions so that it looks more like a classical single-sorted first order logic proof.

[2]Other phrases people use are: it is judgmentally true, or true by definition. In informal mathematics people do not make a difference between equalities which hold because or your surrounding and axioms, and equalities which hold just by definition of the symbols that you have used. In type theory it is very important to make that difference. Terms which are judgmentally equal will always be propositionally equal. If in a type theory judgemental and propositional equality agree, then it is called extensional.

[3]This is a bit of a lie, because Lean actually brakes the good meta-properties that judgemental equality should have by using it for quotient types. Judgemental equality in Lean is not decidable. This led to a lot of conflict between the Lean and the Coq community recently. But for our purpose all of this does not matter at all.



Here A and B stand for proof trees of the induction start and induction step respectively which we still have to provide. So let us next look at the proof of the induction start.

```
lemma comm_start : ∀n:N, n + 0 = 0 + n :=
      ax7 (λn:N, n + 0 = 0 + n) (refl (0 + 0)) comm_start_step
```

The term `comm_start n` should be of type `n + 0 = 0 + n` whenever `n` is of type `N`. This means that `comm_start` should be a proof of `∀n:N, n + 0 = 0 + n`. The proof term works in the same way as the proof term for the main lemma. We proof `comm_start` by induction on `n`. We can write down the following chain of judgements.

```
⊢ ax7 : ∀P: N → Prop, P 0 → ∀m:N, (P m → P(S m)) → ∀m:N, P m

⊢ ax7 (λn:N, n + 0 = 0 + n) :
        0 + 0 = 0 + 0 → ∀n:N, (n + 0 = 0 + n → S n + 0 = 0 + S n)
        → ∀n:N, n + 0 = 0 + n
```

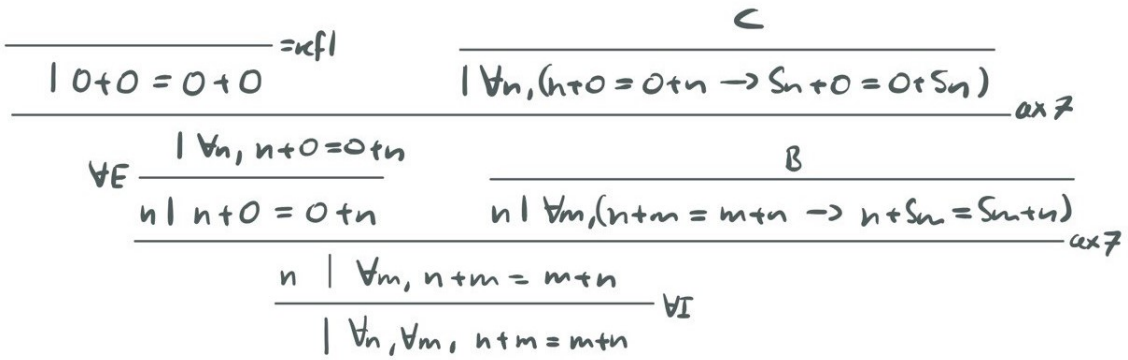
But now we need a proof that `0 + 0 = 0 + 0`. The two sides of the equality sign are exactly equal. How can we tell Lean that this is the case (computers are very stupid). There is a keyword `refl` which produces a proof of `t = t` whenever you feed it a term `t`. Thus we make another judgement.

```
⊢ ax7 (λn:N, n + 0 = 0 + n) (refl (0 + 0)):
        ∀n:N, (n + 0 = 0 + n → S n + 0 = 0 + S n)
        → ∀n:N, n + 0 = 0 + n
```

Next we need a proof of the induction step. We outsource this into another lemma `comm_start_step` which we will have to prove later. Assume we have already done so. Then we can make the judgement

```
⊢ ax7 (λn:N, n + 0 = 0 + n) (refl (0 + 0)) comm_start_step :
        ∀n:N, n + 0 = 0 + n
```

which is exactly the proof term of the lemma `comm_start`. In terms of proof trees, we can replace A as follows.



Our job is to close the remaining two open nodes C and B of the tree, that is to provide proofs for `comm_step` and `comm_start_step`. It turns out that proving `comm_start_step` isn't so difficult. It is a manipulation of equations using the axioms 3 and 4. Proving the induction step of `comm` is more difficult and needs an entire new lemma `Schaukel`.

4 The code

I believe I have explained enough to make most parts of the Lean code understandable, so it is now time to present to you the full proof.

```
  -- Induction start for the Schaukel lemma
lemma Schaukel_start : ∀m:N, m + S 0 = S m + 0 :=
  λ m : N,
  calc
    m + S 0 = S (m + 0) : ax4 m 0
    ...     = S m       : congr_arg S (ax3 m)
    ...     = S m + 0   : eq.symm(ax3 (S m))

-- Induction step for the Schaukel lemma
lemma Schaukel_step : ∀n:N, (∀m:N, m + S n = S m + n)
              → (∀m:N, m + S (S n) = S m + S n)
      :=
  λ n, λ p, λ m,
  calc
    m + S (S n) = S(m + S n)  : ax4 m (S n)
    ...         = S (S m + n) : congr_arg S (p m)
    ...         = S m + S n   : eq.symm (ax4 (S m) n)

lemma Schaukel : ∀n:N, ∀m:N, m + S n = S m + n :=
  ax7 (λn:N, ∀m:N, m + S n = S m + n) Schaukel_start Schaukel_step

-- Induction step for comm_start
lemma comm_start_step : ∀n:N, n + 0 = 0 + n → S n + 0 = 0 + S n :=
  λ n : N, λ p : n + 0 = 0 + n,
  calc
    S n + 0 = S n        :   ax3 (S n)
    ...     = S (n + 0)  :   congr_arg S (eq.symm (ax3 n))
    ...     = S (0 + n)  :   congr_arg S p
    ...     = 0 + S n    :   eq.symm (ax4 0 n)
```

```
-- Induction start for the main theorem
lemma comm_start : ∀n:N, n + 0 = 0 + n :=
  ax7 (λn:N, n + 0 = 0 + n) (refl (0 + 0)) comm_start_step

-- Induction step for the main theorem
lemma comm_step : ∀n:N, ∀m:N, n + m = m + n → n + (S m) = (S m) + n :=
    λ n, λ m, λ p,
    calc
      n + S m = S (n + m) : ax4 n m
      ...     = S (m + n) : congr_arg S p
      ...     = m + S n   : eq.symm (ax4 m n)
      ...     = S m + n   : Schaukel n m

-- Finally the proof of the main theorem
theorem comm : ∀n:N, ∀m:N, n + m = m + n :=
  λn:N, ax7 (λm:N, n + m = m + n) (comm_start n) (comm_step n)
```

We already understand the proof terms of `comm` and `comm_start`. It is should also be clear how the proof term of `Schaukel` works. What I haven't explained yet is the `calc` command which all of the other proof terms use. It is a general principle that computers and formal languages are very stupid. You have to tell them every little step in full detail. Especially dealing with equality is surprisingly difficult in a formal language. Look for example at the proof of `comm_step`. Mathematicians like to chain equal signs such as in

$$n + Sm = S(n + m) = S(m + n) = m + Sn = Sm + n$$

and they will tell you: the first equal sign is true because of axiom four, the second because we have assumed that we have a proof of `n + m = m + n` in the induction step, the third equal sign is true because of axiom four again, and the last equal sign holds because of the Schaukel lemma, which we have shown before. The `calc` command in lean is not part of the core type theory. Instead it is additional syntax on top of the type theory which let's you chain down equalities in a way that you are familiar with from informal mathematics. Behind each of the equal signs in a `calc` command you have a proof that the equality in the same line holds. Lean will produce an official term of `n + S m = S m + n` in the core language[4] before it checks everything for correctness. The tools which Lean uses is that addition is transitive. There is a keyword `Eq.trans` in Lean (which is part of the core language), which given proofs of `t = t'` and `t' = t''` will produce a proof of `t = t''`.

I will stop here. If you need more explanations, or if you like to know about texts where you can read about type theory and categorical logic, then you can ask me in the Hauptseminar or send me an email. My email address is nico@zedat.fu-berlin.de. Also I am happy to help you code in Lean, if you want to learn it. The coolest part of type theory is, that it has semantic in settings which are much more general than sets. If you like to hear about that too, just ask me. Type theory (especially homotopy type theory and its semantic, ∞-categories) is at the moment a very active field of research both in mathematics and theoretical computer science, less so in philosophy unfortunately. If you like to get an idea about what categorical logic can do (not in a foundational but in a practical sense), then you should look at the PhD thesis and talks of Ingo Blechschmidt. There are results in classical mathematics which can be reformulated in an internal language of the subject at hand, and which simplify significantly

[4]The core language is the calculus of inductive constructions.

in the internal language. Higher order logic has semantic in every topos, but the catch is that only intuitionistic reasoning is sound. This means that intuitionistic logic is useful even for a classical mathematicians. The internal logic of some fields (notably schemes and algebraic spaces) forces you to be an intuitionist, if you want to use their internal language and work synthetically. I find this really cool. If you want more explanation, please ask me! I am more than happy to talk about it.