Propositions as Types

Jan 30, 2018

1 Introduction

Sorry the lecture was so sloppy! Dependent types are a difficult concept to fully grasp, and equality types are particularly persnickety. I should have made sure I understood the material better before I taught it, but hopefully I can make it up to you with these notes ^_^

2 Propositional Logic

Propositional logic is the most basic logic that people routinely reason about. It forms the basis of almost all the other logics that people study, [1] and is stupid in a lot of ways, which I'll briefly formalize in some footnotes for the interested. But to start, we should probably have a basic description of what propositional logic is:

Definition 2.1. A sentence in **Propositional Logic** is recursively defined as follows:[2]

$$\begin{aligned} \phi, \psi &:= A, B, C, \ldots && \text{Basic terms} \\ &\mid\ \phi \wedge \psi && \text{Conjunction (and)} \\ &\mid\ \neg\phi && \text{Negation} \end{aligned}$$

We can extend this definition in the obvious way to allow for certain convenient operations that we like.

$$\begin{aligned} \phi \vee \psi &:= \neg(\neg\phi \wedge \neg\psi) && \text{Disjunction (or)} \\ \phi \to \psi &:= \psi \vee \neg\phi && \text{Implication} \\ \phi \leftrightarrow \psi &:= (\phi \to \psi) \wedge (\psi \to \phi) && \text{Bi-implication} \end{aligned}$$

[1] Yes! There are multiple logics, and people reason about the differences between them. If you didn't know this, now you do!

[2] I am intentionally leaving our set of "basic terms" abstract, because this is the standard way of studying a logic. In reality these basic terms are instantiated to an actual set, for instance
{ `it is raining`, `it is tuesday` }
and one can proceed reasoning about terms such as
`it is raining` $\wedge$ `it is tuesday`
However, for most logicians the eventual instance of the logic of study is of little interest, and so the basic terms are left abstract.

2.1 Tautologies

So now we know how to combine preexisting sentences into larger ones, but we still know nothing of truth! We typically extend our logic with a **valuation** function v which takes in each basic proposition and outputs a truth value. Then these truth values are combined in order to find the truth value of any particular sentence ϕ.

However, it is more interesting to abstract away these valuation functions, and ask for **tautologies**.

Definition 2.2. A proposition ϕ is called a **tautology** iff for any valuation function v, ϕ is true.

2.2 Connection to Type Theory

In Type Theory, we represent propositions as types. For instance, the proposition

$$(\phi \wedge (\phi \to \psi)) \to \psi$$

is represented by the following type

$$(\mathbf{A} \times (\mathbf{A} \to \mathbf{B})) \to \mathbf{B}$$

Here we define true sentences to be those types which have values inside them. We define our basic terms to be polymorphic variables (since we aren't allowed to reference their valuation), $\wedge$ to be the product of two types, and $\neg$ to be a function from the type to $\mathbf{0}$, the empty type. [3]

While we could use the definition of $\vee$ in terms of $\wedge$ and $\neg$, it is easier to use the preexisting sumtype to represent disjunction. It is nonobvious that this works, but it would make for a challenging exercise. That regular function types correspond to $\to$ is easier to show, but still a good exercise.

It is easy (for sufficient definitions of easy) to show that a proposition is true in type theory in this sense iff that proposition is a tautology in constructive predicate logic. However, I have omitted such a proof from these notes for brevity. [4]

[3]We do this because functions take each value in the domain to a value of the codomain. So if the codomain is empty, and such a function exists, then the domain must be empty as well.

[4]The main reason I'm not showing this is because it's somewhat beyond the scope of these notes to fully axiomize constructive predicate logic. However, given such an axiomization (which the interested should google) the proof is by structural induction on the proposition.

2.3 Examples

1. $(\phi \wedge (\phi \to \psi)) \to \psi$
As a type: $\mathbf{A} \times (\mathbf{A} \to \mathbf{B}) \to \mathbf{B}$
This is inhabited by (in sml syntax): `fn (a, f) => f a`

2. $(\phi \to (\psi \to \phi))$
As a type: $\mathbf{A} \to \mathbf{B} \to \mathbf{A}$
This is inhabited by: `fn a => fn b => a`

3. $(\phi \wedge \psi \to \phi)$ As a type: $\mathbf{A} \times \mathbf{B} \to \mathbf{A}$
This is inhabited by: `fn (a,b) => a`

2.4 Exercises

1. $(\phi \to (\phi \vee \psi))$
2. $(\phi \to (\psi \to \chi)) \leftrightarrow (\phi \wedge \psi \to \chi)$
3. $(\phi \to \psi) \to (\neg\psi \to \neg\phi)$

2.5 Decidability

Now, propositional logic is a great starting point for understanding logics, but it is not a particularly useful logic in and of itself. This is because we know all of the tautologies already! It's a solved problem, and here's why:

Proof. Take some sentence ϕ, which is comprised of of finitely many basic terms $\{A_i\}$ (where $0 \le i < n$) connected with finitely many $\neg$ and $\wedge$. Recall ϕ is a tautology if and only if ϕ remains true for any valuation of these A_i. Since each A_i can attain one of two possible truth values, there are only 2^n many possible valuations ...
Try them all. $\square$

The way we get more powerful (and in general undecidable) logics is by allowing **quantifiers** which allows us to simultaneously consider an entire family of propositional logic sentences at once. When imbued with quantifiers, our propositional logic becomes **predicate logic**, and our type theory becomes much more interesting.

3 Predicate Logic

With predicate logic, we introduce a quantifier, $\forall$, which allows us to take a family of propositions $\phi(x)$ and ask if each proposition is a tautology simultaneously! With set theory, we now introduce a second set of axioms, which defines the notion of a **set**, which is the object we quantify over. For example, the family of propositions

$$\phi(x) := |x| \ge 0$$

is a potential object of study. We could then ask the question

$$(\forall x \in \mathbb{N})\ \phi(x)$$

wherein we **quantify** over the naturals (a set, which exists outside of our logical system), and the new formula is a predicate tautology iff for each possible x we can plug in, the resulting propositional formula is a propositional tautology.

Further, we can define the **dual** of $\forall$, $\exists$, as follows:

$$(\exists x \in A)\ \phi(x) := \neg(\forall x \in A)\ \neg\phi(x)$$

3.1 Type Theory

This is where things start to get really cool, because in type theory we already have a notion of a set-like object built into the theory. Types themselves have values, which we can quantify over. So there is no need to introduce a second set of axioms, we simply allow types to act on themselves!

To this end, we want to introduce a family of propositions (types), which depend on particular elements of a set (values in a type). So we allow for **type families** **B**(x:**A**). Here, each x:**A** gets a corresponding type, which is inhabited precisely when the proposition given by **B** is true at x.

These types which *depend* on values of some other type are called **dependent types**, and these are the vehicles for doing interesting theorem proving with types. As for why one might want to do interesting theorem proving with types, there are two major selling points. One for the CSy math people, and one for the mathy CS people:

- Using type theory instead of set theory lets our proofs be computer checkable. As our theorems become proressively more abstract, they become harder and harder to reason about. Humans are notoriously sloppy when it comes to logic, and writing our proofs in the language of type theory lets our computers double check our work. Theorems are in NP. Given a potential proof of the theorem, it is relatively easy for a computer to check correctness.

- If you want to enforce more and more complicated invariants at a type level (i.e. your program won't compile if your code allows for certain mistakes) you need a more powerful logic in your type system. For instance, imagine an algorithm which takes in a sorted list and a value which could be in the list, and returns either the index of that element in the list (alongside a proof that such an index is nonnegative) or NONE if the value is not in the list. Such a function might have the following type:
  ```
  (xs :  'a list) -> xs sorted -> 'a -> ((n :  int), n non-neg) option
  ```
  Its first argument is an `'a list`, which is named `xs`. This is important because our second argument is a value inhabiting `(xs sorted)`. `sorted` is a type

family over `'a list`, and we want a proof that `xs : 'a list` is sorted. [5] Such a proof is an inhabitant of `xs sorted`, so this argument prevents us from calling our function on a list which we don't know is sorted. We then take in a value to look for in this list, and if we can't find it, our code returns `NONE`. If we can find it, however, our code returns an index `n`, and a proof that `n` is nonnegative.

To be able to call this function, one might imagine a mergesort implementation with the type `'a list -> ((xs : 'a list) * xs sorted)` [6]

3.2 Quantifiers in Type Theory

Now we know how to express type families, which play the role of propositional families in predicate logic. But how do we express quantification? Let's start with the type theoretic version of $\forall$:

Definition 3.1. A **Pi-type**, denoted $\prod_{x:\mathbf{A}} \mathbf{B}(x)$, is a function which takes an (x:**A**) and returns a value in **B**(x).

You can think of values in a Pi-type as functions mapping values to proofs that those values have a certain property. If the Pi-type is inhabited, then its inhabitant is such a function. Call it f. Then for every (x:**A**), f x has type **B**(x), so in particular **B**(x) is inhabited for each x, and thus the proposition holds for each x, so the corresponding sentence $(\forall (x : \mathbf{A}))\mathbf{B}(x)$ holds (if you excuse the abuse of notation). In this way, Pi-types represent universal quantification quite naturally.

We *could* define a dual of Pi-types by putting a negation on each side, but similar to disjunction, there is a more natural type theoretic approach which has the same effect.

[5] Sticklers will mention that xs does NOT have type `'a list`, since this implies polymorphism and xs has a fixed type for some particular type `'a`. However, I'm glossing over this detail for the purpose of legibility.

[6] These sorts of type level proofs are a field of very active research right now. I mentioned earlier that predicate logic is undecidable in general, and so these dependent types allow for code which has an undecidable typechecking problem (exercise: prove this). A lot of effort is in trying to categorize which of these type familes allow for decidable type checking, and trying to make the most useful invariants possible while still keeping the compilation process decidable

Definition 3.2. A **Sigma-type**, denoted $\sum_{x:\mathbf{A}} \mathbf{B}(x)$, is a tuple whose first slot contains a value of (x:**A**), and whose second slot contains a value of type **B**(x).

This expresses existential quantification in a very interesting and powerful way. A sigma type is the type of all tuples of elements of (x:**A**) and proofs that x satisfies some property. Thus if you ignore the second element of the tuple, $\sum_{x:\mathbf{A}} \mathbf{B}(x)$ is the analogue of $\{x \in A | B(x)\}$. Clearly if this type is inhabited, there is at least one such (x:**A**), and so the existential claim is met.

3.3 Proving Claims By Induction

In type theory, we postulate the existence of certain types, from which we build all of our other types. These base types come preequipped with a notion of induction on their constructors, which we use to prove universal claims about these types. For instance:

Type	Constructors	Induction Term
$\mathbf{1}$	$\texttt{()} : \mathbf{1}$	$\texttt{ind}_{\mathbf{1}} : \mathbf{P}(\texttt{()}) \to \prod_{x:\mathbf{1}} \mathbf{P}(x)$
$\mathbf{A}+\mathbf{B}$	$\texttt{inL} : \mathbf{A} \to \mathbf{A}+\mathbf{B}$, $\texttt{inR} : \mathbf{B} \to \mathbf{A}+\mathbf{B}$	$\texttt{ind}_{+} : \prod_{a:\mathbf{A}} \mathbf{P}(\texttt{inL a}) \to \prod_{b:\mathbf{B}} \mathbf{P}(\texttt{inR b}) \to \prod_{x:\mathbf{A}+\mathbf{B}} \mathbf{P}(x)$
$\mathbf{A}\times\mathbf{B}$	$\texttt{(,)} : \mathbf{A} \to \mathbf{B} \to \mathbf{A}\times\mathbf{B}$	$\texttt{ind}_{\times} : \prod_{a:\mathbf{A}} \prod_{b:\mathbf{B}} \mathbf{P}(\texttt{(a,b)}) \to \prod_{x:\mathbf{A}\times\mathbf{B}} \mathbf{P}(x)$
$\mathbb{N}$	$\texttt{0} : \mathbb{N}$, $\texttt{S} : \mathbb{N} \to \mathbb{N}$	$\texttt{ind}_{\mathbb{N}} : \mathbf{P}(\texttt{0}) \to \prod_{\texttt{n}:\mathbb{N}} (\mathbf{P}(\texttt{n}) \to \mathbf{P}(\texttt{S n})) \to \prod_{\texttt{n}:\mathbb{N}} \mathbf{P}(\texttt{n})$

These induction principles are just formal ways of saying that if each way of constructing the type has a proof, then every element of that type has a proof. For example, to be of type $\mathbf{A}+\mathbf{B}$, you have to be either `inL a` or `inR b`. So to prove that every element of $\mathbf{A}+\mathbf{B}$ has a property **P**, it suffices to show that for each (a : **A**), **P**(`inL a`) and for each (b : **B**), **P**(`inR b`). This is precisely what the type of $\texttt{ind}_{+}$ tells us.

3.4 The Identity Type

There's another type which we postulate, but it's both the least intuitive and the most important, so I'm giving it its own section. Since propositions are types, and equality is in many ways the most important proposition, we need a way of writing equality as a type. To that end, we define the identity type for any (x : **A**), (y : **A**)

$$x =_{\mathbf{A}} y$$

This type is inhabited precisely when x = y. It has one constructor, reflexivity:

$$\texttt{refl}_x : x =_{\mathbf{A}} x$$

So from here, we define its induction principle:

$$\mathtt{ind}_{x=} : \mathbf{P}(x, x, \mathtt{refl}_x) \to \prod_{y:\mathbf{A}} \prod_{\alpha : x =_{\mathbf{A}} y} \mathbf{P}(x, y, \alpha)$$

This takes in a proof that works for every constructor (recall there is only x = x, inhabited by $\mathtt{refl}_x$), and outputs a proof for each element of the identity type (which is all x = y, inhabited by some proof of equality α).

This is an absurdly powerful means of induction, and it almost feels immoral how quickly it trivializes proofs. For instance:

Theorem 3.1. $\prod_{f:\boldsymbol{A}\to\boldsymbol{B}} \prod_{x:\boldsymbol{A}} \prod_{y:\boldsymbol{A}} (x =_{\boldsymbol{A}} y) \to (fx =_{\boldsymbol{B}} fy)$

Proof. Since pi-types are dependant functions, rewrite the innermost function as a pi-type.

$$\prod_{f:\mathbf{A}\to\mathbf{B}} \prod_{x:\mathbf{A}} \prod_{y:\mathbf{A}} \prod_{\alpha : x =_{\mathbf{A}} y} (fx =_{\mathbf{B}} fy)$$

So we want a function whose first two inputs are f and x.

$$\lambda f.\lambda x.\mathit{something}$$

and we know *something* should have the following type

$$\prod_{y:\mathbf{A}} \prod_{\alpha : x =_{\mathbf{A}} y} (fx =_{\mathbf{B}} fy)$$

But this looks like what $\mathtt{ind}_{x=}$ returns! In this case, $\mathbf{P}(x, y, \alpha)$ is $(fx =_{\mathbf{B}} fy)$ (which happens to not depend on α). So we know we can get a value to use for our *something* by calling $\mathtt{ind}_{x=}$ on a value of type $\mathbf{P}(x, x, \mathtt{refl}_x)$ which, for our $\mathbf{P}$, is a value of type $(fx =_B fx)$. But we know what has that type!

$$\mathtt{refl}_{fx} : fx =_{\mathbf{B}} fx$$

So

$$\mathtt{ind}_{x=}(\mathtt{refl}_{fx}) : \prod_{y:\mathbf{A}} \prod_{\alpha : x =_{\mathbf{A}} y} (fx =_{\mathbf{B}} fy)$$

So

$$\lambda f.\lambda x.\mathtt{ind}_{x=}(\mathtt{refl}_{fx}) : \prod_{f:\mathbf{A}\to\mathbf{B}} \prod_{x:\mathbf{A}} \prod_{y:\mathbf{A}} \prod_{\alpha : x =_{\mathbf{A}} y} (fx =_{\mathbf{B}} fy)$$

As desired. □

We will call this function $\mathtt{ap}_f(x, y, \alpha)$

Exercise:

1. Construct $\mathtt{sym} : \prod_{x:\mathbf{A}} \prod_{y:\mathbf{A}} (x =_{\mathbf{A}} y) \to (y =_{\mathbf{A}} x)$
2. Construct $\mathtt{trans} : \prod_{x:\mathbf{A}} \prod_{y:\mathbf{A}} \prod_{z:\mathbf{A}} (x =_{\mathbf{A}} y) \to (y =_{\mathbf{A}} z) \to (x =_{\mathbf{A}} z)$

(oftentimes we suppress writing the first two arguments to `trans`. You'll see why in the next section)

4 Addition on $\mathbb{N}$ is Commutative

Let's go ahead and show the following benign looking claim is true:

$$\prod_{m:\mathbb{N}} \prod_{n:\mathbb{N}} (n + m) =_{\mathbb{N}} (m + n)$$

We'll break this into two parts. First, let's define addition on $\mathbb{N}$, then we'll prove the claim.

4.1 Addition on $\mathbb{N}$

We define addition in the standard inductive way, given by the following function:

Definition 4.1.

$$\begin{aligned} &\texttt{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ &\texttt{add 0 n} = \texttt{n} \\ &\texttt{add (S m) n} = \texttt{S(add m n)} \end{aligned}$$

We write `m+n` in place of `add m n` for convenience.

4.2 The Proof

Theorem 4.1.

$$\prod_{m:\mathbb{N}} \prod_{n:\mathbb{N}} (\texttt{n+m}) =_{\mathbb{N}} (\texttt{m+n})$$

Proof. We proceed by induction on $\mathbb{N}$. We want a term of type

$$\prod_{m:\mathbb{N}} \prod_{n:\mathbb{N}} (\texttt{n+m}) =_{\mathbb{N}} (\texttt{m+n})$$

so $\mathbf{P}$(m) is given by

$$\prod_{n:\mathbb{N}} (\texttt{n+m}) =_{\mathbb{N}} (\texttt{m+n})$$

So to apply $\texttt{ind}_{\mathbb{N}}$, we want a term of type $\mathbf{P}(\texttt{0})$

$$\prod_{n:\mathbb{N}} (\texttt{n+0}) =_{\mathbb{N}} (\texttt{0+n})$$

and a term of type $\prod_{m:\mathbb{N}} \mathbf{P}(\texttt{m}) \to \mathbf{P}(\texttt{S m})$

$$\prod_{m:\mathbb{N}} \left(\left(\prod_{n:\mathbb{N}} (\texttt{n+m}) =_{\mathbb{N}} (\texttt{m+n}) \right) \to \left(\prod_{n:\mathbb{N}} (\texttt{n + S m}) =_{\mathbb{N}} (\texttt{S m + n}) \right) \right)$$

Let's start with the base case, and construct a term of type

$$\prod_{n:\mathbb{N}} (\texttt{n+0}) =_{\mathbb{N}} (\texttt{0+n})$$

Well, we know by the definition of addition that `0+n` is `n`, so we want a term of type

$$\prod_{n:\mathbb{N}} (\texttt{n+0}) =_{\mathbb{N}} (\texttt{n})$$

We proceed (again) by induction on $\mathbb{N}$.
It suffices to find a term of type

$$\texttt{0+0} =_{\mathbb{N}} \texttt{0}$$

and a term of type

$$\prod_{n:\mathbb{N}} ((\texttt{n+0} =_{\mathbb{N}} \texttt{n}) \to (\texttt{S n + 0} =_{\mathbb{N}} \texttt{S n}))$$

For the base case, `0+0` is `0` by definition, so we want a term of type $\texttt{0} =_{\mathbb{N}} \texttt{0}$. But we know we have one of those!

$$\texttt{refl}_{\texttt{0}} : \texttt{0} =_{\mathbb{N}} \texttt{0}$$

For the inductive step, we want

$$\lambda(n : \mathbb{N}).\lambda(\alpha : (\texttt{n+0} =_{\mathbb{N}} \texttt{n})).\textit{something}$$

where *something* has type ($\texttt{S n + 0} =_{\mathbb{N}} \texttt{S n}$). But by the definition of addition, we know (`S n + 0`) is `S (n+0)`. So really, we want *something* to have the type

$$(\texttt{S (n + 0)} =_{\mathbb{N}} \texttt{S n})$$

But `S` is just a function. And we know (by α) that $\texttt{n+0} =_{\mathbb{N}} \texttt{n}$. So by the proof from the last section, we know that equality is preserved by function application. Importantly

$$\texttt{ap}_{\texttt{S}}(\texttt{n+0}, \texttt{n}, \alpha) : (\texttt{S (n + 0)} =_{\mathbb{N}} \texttt{S n})$$

So the inductive step will be witnessed by the following term:

$$\lambda n.\lambda\alpha.\texttt{ap}_{\texttt{S}}(\texttt{n+0}, \texttt{n}, \alpha) : \prod_{n:\mathbb{N}} ((\texttt{n+0} =_{\mathbb{N}} \texttt{n}) \to (\texttt{S n + 0} =_{\mathbb{N}} \texttt{S n}))$$

So, to finish up the base case of our overall theorem,

$$\texttt{ind}_{\mathbb{N}}(\texttt{refl}_{\texttt{0}}, \lambda n.\lambda\alpha.\texttt{ap}_{\texttt{S}}(\texttt{n+0}, \texttt{n}, \alpha)) : \prod_{n:\mathbb{N}} (\texttt{n+0}) =_{\mathbb{N}} (\texttt{n})$$

Remembering that `n` is `0 + n`, let's call this value $\texttt{BASE} : \prod_{n:\mathbb{N}} (\texttt{n+0}) =_{\mathbb{N}} (\texttt{0+n})$

Now for the fun bit. We want a term of type

$$\prod_{m:\mathbb{N}}\left(\left(\prod_{n:\mathbb{N}}(\texttt{n+m}) =_{\mathbb{N}} (\texttt{m+n})\right) \to \left(\prod_{n:\mathbb{N}}(\texttt{n + S m}) =_{\mathbb{N}} (\texttt{S m + n})\right)\right)$$

This will look like

$$\lambda m.\lambda\alpha.something$$

where *something* has the following type

$$\left(\prod_{n:\mathbb{N}}(\texttt{n + S m}) =_{\mathbb{N}} (\texttt{S m + n})\right)$$

Here α is a function which, given `n` : $\mathbb{N}$, spits out a proof that $\texttt{n+m} =_{\mathbb{N}} \texttt{m+n}$

Get ready, because we're proceeding by induction on `n`
For our base case, we want a term of type

$$\texttt{0 + S m} =_{\mathbb{N}} \texttt{S m + 0}$$

which, by the definition of addition, is the same as

$$\texttt{S m} =_{\mathbb{N}} \texttt{S m + 0}$$

which should look familiar. Using `sym` from an earlier exercise,

$$\texttt{sym(BASE(S m))} : \texttt{S m} =_{\mathbb{N}} \texttt{S m + 0}$$

So now it suffices to show a term of type

$$\prod_{n:\mathbb{N}} ((\texttt{n + S m} =_{\mathbb{N}} \texttt{S m + n}) \to (\texttt{S n + S m} =_{\mathbb{N}} \texttt{S m + S n}))$$

This will look like

$$\lambda n.\lambda\beta.\mathit{something}$$

where β is a proof that $(\texttt{n + S m} =_{\mathbb{N}} \texttt{S m + n})$ and *something* has type

$$(\texttt{S n + S m} =_{\mathbb{N}} \texttt{S m + S n})$$

Using the definition of addition on both sides, we recover

$$(\texttt{S (n + S m)} =_{\mathbb{N}} \texttt{S (m + S n)})$$

Now notice that

$$\texttt{ap}_{\texttt{S}}(\beta) : \texttt{S (n + S m)} =_{\mathbb{N}} \texttt{S (S m + n)}$$

Further, using the definition of addition again,

$$\texttt{ap}_{\texttt{S}}(\beta) : \texttt{S (n + S m)} =_{\mathbb{N}} \texttt{S (S (m + n))}$$

But now, remembering our outer induction hypothesis α,

$$\texttt{sym}(\texttt{ap}_{\texttt{S}\circ\texttt{S}}(\alpha(\texttt{n}))) : \texttt{S (S (m + n))} =_{\mathbb{N}} \texttt{S (S (n + m))}$$

so by another exercise, (omitting the first two arguments to `trans` for legibility)

$$\texttt{trans}(\texttt{ap}_{\texttt{S}}(\beta), \texttt{sym}(\texttt{ap}_{\texttt{S}\circ\texttt{S}}(\alpha(\texttt{n})))) : \texttt{S (n + S m)} =_{\mathbb{N}} \texttt{S (S (n + m))}$$

Using the definition of addition again (on both sides), we step backwards to

$$\texttt{trans}(\texttt{ap}_{\texttt{S}}(\beta), \texttt{sym}(\texttt{ap}_{\texttt{S}\circ\texttt{S}}(\alpha(\texttt{n})))) : \texttt{S n + S m} =_{\mathbb{N}} \texttt{S (S n + m)}$$

Seeing the light at the end of the tunnel, we recognize

$$\texttt{ap}_{\texttt{S}}(\alpha(\texttt{S n})) : \texttt{S (S n + m)} =_{\mathbb{N}} \texttt{S (m + S n)}$$

which, by the definition of addition is

$$\texttt{ap}_{\texttt{S}}(\alpha(\texttt{S n})) : \texttt{S (S n + m)} =_{\mathbb{N}} \texttt{S m + S n)}$$

And, finally

$$\texttt{trans}(\texttt{trans}(\texttt{ap}_{\texttt{S}}(\beta), \texttt{sym}(\texttt{ap}_{\texttt{S}\circ\texttt{S}}(\alpha(\texttt{n})))), \texttt{ap}_{\texttt{S}}(\alpha(\texttt{S n}))) : \texttt{S n + S m} =_{\mathbb{N}} \texttt{S m + S n}$$

Now! To put it all together!
To finish the inner induction,

$$\lambda n.\lambda\beta.\texttt{trans}(\texttt{ap}_{\texttt{S}}(\beta), \texttt{sym}(\texttt{ap}_{\texttt{S}\circ\texttt{S}}(\alpha(\texttt{n})))), \texttt{ap}_{\texttt{S}}(\alpha(\texttt{S n}))) : \prod_{n:\mathbb{N}} ((\texttt{n + S m} =_{\mathbb{N}} \texttt{S m + n}) \to (\texttt{S n + S m} =_{\mathbb{N}} \texttt{S m + S n}))$$

So, after much effort,

$$\lambda m.\lambda\alpha.\texttt{ind}_{\mathbb{N}}(\texttt{sym}(\texttt{BASE}(\texttt{S m})), \texttt{trans}(\texttt{trans}(\texttt{ap}_{\texttt{S}}(\beta), \texttt{sym}(\texttt{ap}_{\texttt{S}\circ\texttt{S}}(\alpha(\texttt{n})))), \texttt{ap}_{\texttt{S}}(\alpha(\texttt{S n})))) : \prod_{m:\mathbb{N}} \left(\left(\prod_{n:\mathbb{N}} (\texttt{n+m}) =_{\mathbb{N}} (\texttt{m+n}) \right) \to \left(\prod_{n:\mathbb{N}} (\texttt{n + S m}) =_{\mathbb{N}} (\texttt{S m + n}) \right) \right)$$

Call this term

$$\texttt{EFFORT} : \prod_{m:\mathbb{N}} \left(\left(\prod_{n:\mathbb{N}} (\texttt{n+m}) =_{\mathbb{N}} (\texttt{m+n}) \right) \to \left(\prod_{n:\mathbb{N}} (\texttt{n + S m}) =_{\mathbb{N}} (\texttt{S m + n}) \right) \right)$$

Now, for the coup de grâce

$$\texttt{ind}_{\mathbb{N}}(\texttt{BASE}, \texttt{EFFORT}) : \prod_{m:\mathbb{N}} \prod_{n:\mathbb{N}} (\texttt{n+m}) =_{\mathbb{N}} (\texttt{m+n})$$

And, for the people who want to see it in full, unadulterated glory. Here is the entire proof term:

$$\begin{aligned} &\texttt{ind}_{\mathbb{N}}(\texttt{ind}_{\mathbb{N}}(\texttt{refl}_{\texttt{0}}, \lambda n.\lambda\alpha.\texttt{ap}_{\texttt{S}}(\texttt{n+0}, \texttt{n}, \alpha)), \\ &\qquad \lambda m.\lambda\alpha.\texttt{ind}_{\mathbb{N}}(\texttt{sym}((\texttt{ind}_{\mathbb{N}}(\texttt{refl}_{\texttt{0}}, \lambda n.\lambda\gamma.\texttt{ap}_{\texttt{S}}(\texttt{n+0}, \texttt{n}, \gamma)))(\texttt{S m})), \\ &\qquad\quad \texttt{trans}(\texttt{trans}(\texttt{ap}_{\texttt{S}}(\beta), \texttt{sym}(\texttt{ap}_{\texttt{S}\circ\texttt{S}}(\alpha(\texttt{n})))), \texttt{ap}_{\texttt{S}}(\alpha(\texttt{S n}))))) : \\ &\qquad\qquad\qquad \prod_{m:\mathbb{N}} \prod_{n:\mathbb{N}} (\texttt{n+m}) =_{\mathbb{N}} (\texttt{m+n}) \end{aligned}$$

as desired.

□