Topic 3: Propositions as types

May 18, 2014

Propositions as types

We have seen that the main mathematical objects in a type theory are types. But remember that in conventional foundations, as based on logic and set theory, there are not only sets, but also logical propositions! So what are propositions in type-theoretic foundations? The short answer: propositions are types as well!

But wait, then what are supposed to be the elements of these types? Again, the short answer: they are the *proofs* of that proposition! In particular, in type theory, proofs are not special objects, but on par with perhaps more familiar mathematical objects such as numbers, functions etc. Also, proofs matter: as we will see, the inference rule

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash x : A}{\Gamma \vdash f(x) : B} \tag{1}$$

can be understood as follows, in case that A and B play the role of propositions: if f proves that A implies B and a proves that A, then $f(a)$ proves that B. So a proof in the sense of an element $x : A$ can be seen as a witness of the correctness of the proposition A.

So now if proofs are elements of propositions and propositions are types, then what are the logical connectives? Equation (1) may give some idea: they correspond to *type formers* like '$\to$'. If A and B are types which represent propositions, then the function $A \to B$ represents the proposition 'A implies B'. Thinking of elements as proofs, this makes perfect sense: the possible proofs of an implication $A \to B$ corresponds then to all the possible functions which turn a proof of A into a proof of B.

Definitions vs theorems. An intriguing aspect of the propositions-as-types correspondence is that the difference between definitions and proofs of theorems ceases to exist; which judgements should be considered as definitions and which ones as theorems or lemmas is an issue of exposition which only matters when we explain our mathematics to our fellow mathematicians.

For example in `Coq`, it is possible to define function composition through a proof:

```
(* Function composition *)

Definition compose (A B C : Type) : (A → B) → (B → C) → (A → C).
Proof.
        intros f g x.
        apply g.
        apply f.
        exact x.
Defined.
```

What is going on here is that the individual steps in the proof are a different way of writing down the individual parts of the expression $\lambda f.\lambda g.\lambda x.g(f(x))$. On the other hand, we can also interpret $(A \to B) \to (B \to C) \to (A \to C)$ as the statement that if A implies B and B implies C, then A implies C. This is a theorem to which the exact same proof applies:

```
(* Transitivity of implication *)

Theorem trans (A B C : Type) : (A → B) → (B → C) → (A → C).
Proof.
        intros f g x.
        apply g.
        apply f.
        exact x.
Qed.
```

An important aspect of this unification of definitions and proofs is that the style of mathematics in HoTT is at times quite different. In particular, HoTT is *proof-relevant*: often, it not only matters *that* one has a proof of some theorem, but it is also relevant *what* that proof is. For example, a proof of some theorem may be referenced later as part of the *statement* of another theorem: the above example of the proof of transitivity of implication illustrates this nicely, since in some later situation we may want to interpret this proof as function composition and formulate another theorem *about* function composition. This is why doing mathematics in HoTT requires a good memory. Computer proof assistants like `Coq` can assist with this.

Logical connectives as type formers. We now reproduce a table from the HoTT book which lists the basic type formers and corresponding logical connectives of arity zero, one and two:

Logic	Type theory
True	$\mathbf{1}$
False	0
A and B	$A \times B$
A or B	$A + B$
A implies B	$A \to B$
A if and only if B	$(A \to B) \times (B \to A)$
Not A	$A \to \mathbf{0}$

We have already encountered the type former '$\to$' in Topic 2. We now proceed to define the other type formers by writing down the inference rules which can be used for working with them, before commenting further on the significance of the 'propositions-as-types' idea.

The unit type '1'

Again, the rules for the unit type are of the familiar kinds:

1. Formation rule:

$$\frac{\Gamma \ \mathtt{ctx}}{\Gamma \vdash \mathbf{1} : \mathtt{Type}}$$

In words: $\mathbf{1}$ can be constructed as a type in any context.

2. Introduction rule:

$$\frac{\Gamma \ \mathtt{ctx}}{\Gamma \vdash \star : \mathbf{1}}$$

In any context, we can construct an element of $\mathbf{1}$ denoted $\star : \mathbf{1}$.

3. Elimination rule:

$$\frac{\Gamma, x : \mathbf{1} \vdash C : \mathtt{Type} \qquad \Gamma \vdash c : C[\star/x] \qquad \Gamma \vdash a : \mathbf{1}}{\Gamma \vdash \mathrm{ind}_{\mathbf{1}}(x.C, c, a) : C[a/x]}$$

Here, the notation '$x.C$' means that x, which may occur as a variable in the expression C, is now regarded as a *bound* variable in the same expression. With the rules that we have until now, our theory does not allow the derivation of judgements of the form $\Gamma, x : \mathbf{1} \vdash C : \mathtt{Type}$ in which C depends explicitly on the variable x; but we will soon encounter rules which permit the construction of such *dependent types*, and then this becomes relevant.

This rules states that if C is a type, possibly depending on or indexed by some variable $x : \mathbf{1}$, then in order to construct an element of $C[a/x]$ for some given expression $a : \mathbf{1}$, it is sufficient to construct an element of $C[\star/x]$. This is an indirect way of expressing the idea that the unit type $\mathbf{1}$ has exactly one element. This is consistent with the philosophy that the relevant aspects of a mathematical object ($\mathbf{1}$ in this case) are not its internal structure, but the ways in which it interacts with other mathematical objects. The reason for writing '$\mathrm{ind}_{\mathbf{1}}$' is that the elimination rule can also be interpreted as *induction* over the unit type $\mathbf{1}$, in the sense of a 'reduction to the base case', where the base case is $\star : \mathbf{1}$.

If one thinks of C as a proposition indexed by a variable $x : \mathbf{1}$, then this rule states that in order to prove C for a certain expression a in place of x, then it is sufficient to prove it in the case where x is replaced by $\star$.

Again, this rule needs to be supplemented by a corresponding rule stating that $\mathrm{ind}_{\mathbf{1}}$ preserves judgemental equality in all its arguments:

$$\frac{\Gamma, x : \mathbf{1} \vdash C \equiv C' : \mathtt{Type} \qquad \Gamma \vdash c \equiv c' : C[\star/x] \qquad \Gamma \vdash a \equiv a' : \mathbf{1}}{\Gamma \vdash \mathrm{ind}_{\mathbf{1}}(x.C, c, a) \equiv \mathrm{ind}_{\mathbf{1}}(x.C', c', a') : C[a/x]}$$

In the conclusion, we also could have replaced $C[a/x]$ by $C[a'/x]$, since the assumption $a \equiv a'$ guarantees that $C[a'/x] \equiv C[a/x]$. (Again, this is a metatheoretical property.)

From now on, we will omit explicit mention of such 'companion' rules stating preservation of judgemental equalities. We take it to be implicitly understood that if we write down some inference rule for a typing judgement, then the corresponding rule for 'propagating' a judgemental equality from top to bottom is assumed as well.

4. Computation rule:

$$\frac{\Gamma, x : \mathbf{1} \vdash C : \mathtt{Type} \qquad \Gamma \vdash c : C[\star/x]}{\Gamma \vdash \mathrm{ind}_{\mathbf{1}}(x.C, c, \star) \equiv c : C[\star/x]}$$

This rule tells us what happens if we apply the elimination rule to the canonical element $\star : \mathbf{1}$, namely we simply recover the given c up to judgemental equality.

Although the terminology 'unit type' suggests that $\mathbf{1}$ may have only one element, the candidate judgement

$$\Gamma, a : \mathbf{1} \vdash a \equiv \star : \mathbf{1}$$

is not derivable from these inference rules! It would be analogous to the 'uniqueness principle' that we postulated for function types. It is not perfectly clear to us why one postulates a uniqueness principle for function types but not for the unit type. However, we will see later in which sense the unit type does have exactly one element.

The empty type '0'

1. Formation rule:

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathbf{0} : \mathtt{Type}}$$

2. Elimination rule:

$$\frac{\Gamma, x : \mathbf{0} \vdash C : \mathtt{Type} \qquad \Gamma \vdash a : \mathbf{0}}{\Gamma \vdash \mathrm{ind}_{\mathbf{0}}(x.C, a) : C[a/x]}$$

   This rule has a beautiful interpretation in terms of propositions-as-types: if C represents a proposition in a context in which $x : \mathbf{0}$, i.e. x is a proof of False, then one automatically obtains a proof of $C[a/x]$ for any a. In other words: falsity implies anything!

Here, the absence of an introduction rule does not mean that the empty type somehow deviates from the usual paradigm—to the contrary! Usually, there is one introduction rule for every way in which an element of the type under consideration can be constructed. In the case of the empty type, there is no way to construct an element, and hence there is no introduction rule.

Product types

Intuitively, for two types A and B, the product type $A \times B$ has as elements the pairs (x, y). But again, instead of defining the type by saying which elements it contains, we rather specify operational rules which tell us how to *use* the product type and its elements.

1. Formation rule:

$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma \vdash B : \mathtt{Type}}{\Gamma \vdash A \times B : \mathtt{Type}}$$

   This rules tells us that a product type $A \times B$ can be formed for any given types A and B in any context.

2. Introduction rule:

$$\frac{\Gamma \vdash x : A \qquad \Gamma \vdash y : B}{\Gamma \vdash (x, y) : A \times B}$$

   Any pair of elements of A and B yields an element of $A \times B$, which we denote using the usual notation for a pair.

3. Elimination rule:

$$\frac{\Gamma, p : A \times B \vdash C : \mathtt{Type} \qquad \Gamma, x : A, y : B \vdash g : C[(x,y)/p] \qquad \Gamma \vdash q : A \times B}{\Gamma \vdash \mathrm{ind}_{A\times B}(p.C, x.y.g, q) : C[q/p]}$$

   This rule shows that to derive an assertion C involving any element $q : A \times B$, it suffices to derive it for all explicitly given pairs $(x, y) : A \times B$.

4. Computation rule:

$$\frac{\Gamma, p : A \times B \vdash C : \mathtt{Type} \qquad \Gamma, x : A, y : B \vdash g : C[(x,y)/p] \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathrm{ind}_{A\times B}(p.C, x.y.g, (a,b)) \equiv g[a/x, b/y] : C[(a,b)/p]}$$

   As usual, the computation rule tells us what happens when we apply the elimination rule on an element obtained via the introduction rule.

As an example application, we can construct the *product projections* $\mathrm{pr}_1 : A \times B \to A$ and $\mathrm{pr}_2 : A \times B \to B$ which map each pair to the corresponding component,

$$\mathrm{pr}_1 :\equiv \lambda(q : A \times B).\mathrm{ind}_{A\times B}(p.A, x.y.x, q)$$
$$\mathrm{pr}_2 :\equiv \lambda(q : A \times B).\mathrm{ind}_{A\times B}(p.B, x.y.y, q).$$

Again, there is no rule which would tell us that any $p \equiv (\mathrm{pr}_1(p), \mathrm{pr}_2(p))$ for any p, and in fact this statement is not derivable from the rules of HoTT, although it is intuitive. We will get back to this point later.

Coproduct Types

What we mean by a 'coproduct' $A + B$ of types A and B is what correponds to a *disjoint union* in a set-theoretic context: intuitively, an element of $A + B$ is either an element of A or an element of B. The coproduct type is operationally specified through the following inference rules:

1. Formation rule:

$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma \vdash B : \mathtt{Type}}{\Gamma \vdash A + B : \mathtt{Type}}$$

   This rule exemplifies how to form a coproduct type $A + B$ given A and B.

2. Introduction rules:

$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma \vdash B : \mathtt{Type} \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathrm{inl}(a) : A + B}$$

   This rule associates to every expression $a : A$ an element $\mathrm{inl}(a) : A + B$. Similarly, there is another introduction rule that associates to every element $b : B$ the corresponding $\mathrm{inr}(b) : B$,

$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma \vdash B : \mathtt{Type} \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathrm{inr}(b) : A + B}$$

   Here, 'inl' and 'inr' stand for 'in left' and 'in right', respectively.

3. Elimination rule:

$$\frac{\Gamma, z : A + B \vdash C : \mathtt{Type} \qquad \Gamma, x : A \vdash c : C[\mathrm{inl}(x)/z] \qquad \Gamma, y : B \vdash d : C[\mathrm{inr}(y)/z] \qquad \Gamma \vdash e : A + B}{\Gamma \vdash \mathrm{ind}_{A+B}(z.C, x.c, y.d, e) : C[e/z]}$$

As usual for an elimination rule, this states that in order to prove a statement concerning some element $e : A + B$, it suffices to prove it for all 'base cases', meaning all elements of the form $\mathrm{inl}(a)$ and $\mathrm{inr}(b)$.

4. Computation rules:

$$\frac{\Gamma, z : A + B \vdash C : \mathtt{Type} \qquad \Gamma, x : A \vdash c : C[\mathrm{inl}(x)/z] \qquad \Gamma, y : B \vdash d : C[\mathrm{inr}(y)/z] \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathrm{ind}_{A+B}(z.C, x.c, y.d, \mathrm{inl}(a)) \equiv c[a/x] : C[\mathrm{inl}(a)/z]}$$

Again, this explains what happens when one applies the elimination to one of the 'base cases'. There is an analogous rule for the other base case:

$$\frac{\Gamma, z : A + B \vdash C : \mathtt{Type} \qquad \Gamma, x : A \vdash c : C[\mathrm{inl}(x)/z] \qquad \Gamma, y : B \vdash d : C[\mathrm{inr}(y)/z] \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathrm{ind}_{A+B}(z.C, x.c, y.d, \mathrm{inr}(b)) \equiv d[b/y] : C[\mathrm{inr}(b)/z]}$$

This ends our list of type formers which represent the logical connectives for propositional logic. We thus arrive at a situation in which propositions are represented by types and proofs by elements of that type. So proving a proposition means finding an element of that type. Let us give an example to understand this better. Consider the proposition "if A then (if B then A)"; it is a tautology, even in the system that we are considering. In our type-theoretic language, it would be formulated as the type $A \to (B \to A)$ in the context $A, B : \mathtt{Type}$. Finding an element of this type means finding an expression E which fits into a typing judgement

$$A, B : \mathtt{Type} \vdash E : A \to (B \to A). \tag{2}$$

Here is an example of what this expression can be and what the associated proof tree looks like:

$$\cfrac{\cfrac{\cfrac{\cfrac{\vdots}{(A, B : \mathtt{Type}, x : A, y : B)\ \mathtt{ctx}}}{A, B : \mathtt{Type}, x : A, y : B \vdash x : A}}{A, B : \mathtt{Type}, x : A \vdash (\lambda(y : B).x) : B \to A}}{A, B : \mathtt{Type} \vdash \lambda(x : A).\lambda(y : B).x : A \to (B \to A)}$$

It should be clear how to complete this proof tree by deriving $(A, B : \mathtt{Type}, x : A, y : B)\ \mathtt{ctx}$. In `Coq`, which automatically takes care of contexts, this proof would simply look like this:

```
(* Proving a simple tautology *)

Theorem tautology (A B : Type) : A → (B → A).

Proof.
    intro a.
    intro b.
    exact a.
Qed.
```

Proofs as programs. So we have seen that propositions become types whose elements play the role of proofs. But there is another interpretation of elements of types: they are programs doing a computation! For example, function composition

$$A, B, C : \mathtt{Type} \vdash \lambda f.\lambda g.\lambda x.g(f(x)) : (A \to B) \to (B \to C) \to (A \to C) \tag{3}$$

can be understood as a program which takes an $f : A \to B$ and a $g : B \to C$ as inputs and returns the composite $g \circ f : A \to C$. From this point of view, any type theory is a *programming language*! We will later see how to write more elaborate programs in type theory, like a program for sorting lists.

In particular, every proof of a proposition has an interpretation of a program which computes an actual element of the proposition-as-type.

Type-checking vs theorem-proving. An important distinction is between *finding* an element of a given type in a given context, as in (2), and *verifying* that a claimed typing judgement is indeed derivable. In terms of the logical interpretation, the former problems consists in finding a proof of a conjectural statement, while the latter boils down to verifying such a proof. In the programming context, the latter is also known as *type-checking*, i.e. verifying that all constructions used in a program are consistent with the specified variable types.

The reason that it can be checked algorithmically whether a claimed typing judgement is derivable is the following: for every expression formed from the inference rules, the expression essentially 'remembers' which inference rules have been applied in order to construct it. For example with (3), it is clear that the last inference rule used in constructing that typing judgement must have been a lambda abstraction, i.e. an application of the introduction rule for function types. Hence the main parts of the proof tree of any typing judgement can be reconstructed by decomposing the given expression step by step, starting with the outermost part.

Again, this simplicity of type-checking is a metatheoretical property. As before, we restrain from a rigorous treatment of this issue and leave it at the present informal discussion.

Constructivism. The fact that every proof in type theory can be interpreted as a program or algorithm means that type theory is inherently *constructive*: proving a proposition boils down to writing down an explicit element of the corresponding type, which has an interpretation as a program which can possibly be run on a computer. This has as a consequence that some statements which are true in classical logic have no analog in type theory. For example, there is no expression E which would fit into a valid judgement of the form

$$A : \mathtt{Type} \vdash E : A + (A \to \mathbf{0}). \tag{4}$$

As a proposition, the statement $A + (A \to \mathbf{0})$ translates into $A \vee \neg A$, which is the *law of exluded middle*. One can easily show that this holds for any proposition A in classical logic: if A is true, then certainly $A \vee \neg A$ must be true as well; on the other hand, if A is false, then $\neg A$ is true and $A \vee \neg A$ holds likewise.

The intuitive reason that no judgement of the form (4) holds is that there is no program to prove it. After all, such a program would have to return an actual element of $A + (A \to \mathbf{0})$, and since any element of $A + (A \to \mathbf{0})$ is (at least intuitively) either an element of A or an element of $A \to \mathbf{0}$, this means that whatever algorithm E would fit into such a judgement, it would necessarily decide for any given proposition A whether it is true or false. But it is known that no such algorithm

can exist, at least in more expressive theories than we currently have; for example, by the negative solution to Hilbert's 10th problem, there is no algorithm which could determine whether a given polynomial equation with integer coefficients has an integer solution or not.

So although the above judgement does not hold, it is still possible to use the law of excluded middle if one is inclined to do so: this can be done by assuming an additional $em : A + (A \to \mathbf{0})$ in the context, which corresponds to a hypothetical proof of excluded middle for A. As an example, consider this complicated-looking typing judgement:

$$\begin{aligned} A, B :&\mathtt{Type}, em : A + (A \to \mathbf{0}) \\ &\vdash \lambda F.\mathrm{ind}_{A+(A\to\mathbf{0})}\left(z.A, x.x, y.F(\lambda(a:A).\mathrm{ind}_{\mathbf{0}}(w.B, y(a))), em\right) : ((A \to B) \to A) \to A. \end{aligned} \tag{5}$$

We derive this below. In classical logic, the statement 'if A implies B implies A, then A' is known as *Peirce's law*. As this example shows, type theory is *agnostic* with respect to the law of excluded middle: it does not follow from the basic inference rules, but there is no harm in assuming it as an additional axiom. This has the interesting feature that it makes uses of the law of excluded middle completely explicit.

We now describe the proof tree for (5), while slowly moving to a more informal mode of reasoning in order to aid comprehensibility. Constructing the element of $(A \to B) \to A) \to A$ in (5) means that we need to assign to every $F : (A \to B) \to A$ an element of A. This is the informal explanation of lambda abstraction, by which we can move one step up in the proof tree and arrive at

$$\begin{aligned} A, B :&\mathtt{Type}, em : A + (A \to \mathbf{0}), F : (A \to B) \to A \\ &\vdash \mathrm{ind}_{A+(A\to\mathbf{0})}\left(z.A, x.x, y.F(\lambda(a:A).\mathrm{ind}_{\mathbf{0}}(w.B, y(a))), em\right) : A. \end{aligned}$$

Since now $\mathrm{ind}_{A+(A\to\mathbf{0})}$ is outermost, one can go up one more step in the proof tree by using the elimination rule for the coproduct $A + (A \to \mathbf{0})$, which plays the rule of a *case distinction* splitting the problem into $x : A$, i.e. A holds, and $y : A \to \mathbf{0}$, i.e. A does not hold. If we abbreviate the present context by Γ, then this reduces the problem to deriving the *four* typing judgements

$$\begin{gathered} \Gamma, z : A + (A \to \mathbf{0}) \vdash A : \mathtt{Type} \\ \Gamma, x : A \vdash x : A \\ \Gamma, y : A \to \mathbf{0} \vdash F(\lambda(a:A).\mathrm{ind}_{\mathbf{0}}(w.B, y(a))) : A \\ \Gamma \vdash em : A + (A \to \mathbf{0}). \end{gathered}$$

The important ones are the second, which tells us that if A holds then we simply return the given proof of A, and the third, which says that if the negation of A holds, then we return the complicated-looking element. These four judgements are all trivial to derive except for the third, so we focus on this. Since the outermost construction in this judgement is a function application (elimination rule for function types), going one step up in the proof tree yields the two branches

$$\begin{gathered} \Gamma, y : A \to \mathbf{0} \vdash F : (A \to B) \to A \\ \Gamma, y : A \to \mathbf{0} \vdash \lambda(a:A).\mathrm{ind}_{\mathbf{0}}(w.B, y(a)) : A \to B, \end{gathered}$$

so we find that the complicated-looking element is given by an application of F to a complicated-looking function $A \to B$. Again, deriving the first one is trivial, so we continue with the second. It

says that the function $A \to B$ arises from an application of lambda abstraction to the judgement

$$\Gamma, y : A \to \mathbf{0}, a : A \vdash \mathrm{ind}_{\mathbf{0}}(w.B, y(a)) : B.$$

This comes from the elimination rule for $\mathbf{0}$, which requires

$$\Gamma, y : A \to \mathbf{0}, a : A, w : \mathbf{0} \vdash B : \mathtt{Type}$$

$$\Gamma, y : A \to \mathbf{0}, a : A \vdash y(a) : \mathbf{0}$$

Again the first one follows from a simple formation rule, while the second one can now be reduced to judgements that are finally all trivial,

$$\Gamma, y : A \to \mathbf{0}, a : A \vdash y : A \to \mathbf{0}$$

$$\Gamma, y : A \to \mathbf{0}, a : A \vdash a : A.$$

Over the course of the following lectures, we will turn this very formal style of reasoning into more and more informal language. Nevertheless, one should always be able to transform an informal argument back into a formal one, since only then can one achieve the level of accuracy required in mathematical proofs. An informal proof of Peirce's law would be as follows:

Proof. We assume the law of excluded middle, so we can distinguish the two cases that A holds and that the negation of A holds. If A holds, then we simply return the given proof of A; while if the negation of A holds, then we take the proof $y : A \to \mathbf{0}$ of this, compose it with the trivial function $\mathbf{0} \to B$ in order to obtain a function $A \to B$, and then evaluate $F : (A \to B) \to A$ on this composite function. □

It should be clear that this informal style is much more concise and also more comprehensible to a human reader, which is why we prefer it in order to do actual mathematics.

Finally, let us note that the situation with the axiom of choice is similar to the case of excluded middle, but more subtle. We will not discuss this now, though.

But what about quantifiers?

So far, we have only discussed propositional logic in the propositions-as-types paradigm. However, doing any kind of non-trivial mathematics requires the use of quantifiers! For example, in the statement that 'there exists no triple of positive natural numbers (x, y, z) such that $x^3 + y^3 = z^3$', the expression $x^3 + y^3 = z^3$ is a proposition which *depends* on variables $x, y, z : \mathbb{N}$. These variables become bound after the introduction of the existential quantifier to the proposition

$$\exists x, \exists y, \exists z,\ x^3 + y^3 + z^3.$$

Until next time, you want to try and figure out for yourself what the type-theoretic generalization of quantifiers might be!