An Introduction to Type Theory

Dan Christensen

UWO, May 8, 2014

Outline:

- A brief history of types
- Type theory as set theory
- Type theory as logic

History of Type Theory

Initial ideas due to Bertrand Russell in early 1900's, to create a foundation for mathematics that avoids Russell's paradox.

Studied by many logicians and later computer scientists, particularly Church, whose λ-calculus (1930's and 40's) is a type theory.

In 1972, Per Martin-Löf extended type theory to include dependent types. It is this form of type theory that we will focus on.
One of the key features is that it *unifies* set theory and logic!

In 2006, Awodey and Warren, and Voevodsky, discovered that type theory has *homotopical* models. Understanding this interpretation will be the goal of the overall seminar, but not of this talk.

2012–2013: a special year at the IAS, which led to a book.

Last week: \$7.5M Dept of Defense grant for a group led by Awodey.

Type theory as set theory: Notation

Type theory studies things called types, which can be thought of as sets. The analog of elements of a set are called terms.

We write $a : A$ to indicate that a is a term of type A.

And we write $A : \texttt{Type}$ to indicate that A is a type.

A judgement is a string of symbols that may or may not be provable from the rules of type theory. A judgement always contains the turnstile symbol $\vdash$, separating context from the conclusion.

Examples (using things we haven't defined yet):

$$\begin{aligned} \vdash \quad & 0 : \mathbb{N} \\ x : \mathbb{N},\, y : \mathbb{N} \vdash \quad & x + y : \mathbb{N} \\ f : \mathbb{R} \to \mathbb{R},\, x : \mathbb{R} \vdash \quad & f(x) : \mathbb{R} \end{aligned}$$

Type theory as set theory: more judgements

Judgements can also involve asserting that something is a type.

Examples:

$$\vdash \mathbb{N} : \mathtt{Type}$$

$$A : \mathtt{Type},\ B : \mathtt{Type} \vdash A \times B : \mathtt{Type}$$

$$A : \mathtt{Type},\ B : \mathtt{Type} \vdash A \to B : \mathtt{Type}$$

The last one says that if A and B are types, then there is a type written $A \to B$ (thought of as B^A).

Finally, judgements can assert that terms are judgementally equal, roughly, "equal by definition". Examples:

$$a : A \vdash a \equiv a : A$$

$$\vdash (\lambda x.2 + x)(3) \equiv 2 + 3 : \mathbb{N}$$

(Here $\lambda x.2 + x$ denotes the function $x \mapsto 2 + x$ in $\mathbb{N} \to \mathbb{N}$.)

Constructing types: the empty type

Now we begin introducing the "rules of the game", which say which judgements we can derive.

We begin by introducing the empty type:

$$\frac{}{\vdash \varnothing : \mathtt{Type}}$$

Now we can prove our first theorem, namely $\vdash \varnothing : \mathtt{Type}$!

We also have a (weak) universal property:

$$\frac{\Gamma \vdash p : \varnothing \qquad \Gamma \vdash C : \mathtt{Type}}{\Gamma \vdash \mathtt{abort}(p) : C}$$

Intuitively, for any type C, there is a function $\varphi : \varnothing \to C$, and so for any element p of $\varnothing$, $\varphi(p)$ is a well-defined element of C.

The unit type

Next, the unit type, analogous to a set with one element:

$$\frac{}{\vdash \texttt{unit} : \texttt{Type}} \qquad \frac{}{\vdash \texttt{tt} : \texttt{unit}}$$

$$\frac{\Gamma \vdash C : \texttt{Type} \qquad \Gamma \vdash c : C \qquad \Gamma \vdash p : \texttt{unit}}{\Gamma \vdash \texttt{triv}(p, c) : C}$$

The last rule above is a (weak) universal property.

There is also a computation rule:

$$\frac{\Gamma \vdash C : \texttt{Type} \qquad \Gamma \vdash c : C}{\Gamma \vdash \texttt{triv}(\texttt{tt}, c) \equiv c : C}$$

This is the first appearance of $\equiv$.

Products

Product types are asserted to exist:

$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma \vdash B : \mathtt{Type}}{\Gamma \vdash A \times B : \mathtt{Type}} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}$$

$$\frac{\Gamma \vdash C : \mathtt{Type} \qquad \Gamma \vdash p : A \times B \qquad \Gamma, x : A, y : B \vdash c : C}{\Gamma \vdash \mathtt{unpack}(p, c) : C}$$

$$\frac{\Gamma \vdash C : \mathtt{Type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B \qquad \Gamma, x : A, y : B \vdash c : C}{\Gamma \vdash \mathtt{unpack}((a, b), c) \equiv c[a/x, b/y] : C}$$

Note that there are again four rules playing similar roles: type constructor, term constructor, eliminator and computation rule.

Aside: Projections

We interrupt our list of the rules to show how we can use `unpack` to define the projections. Our context will be

$$\Gamma :\equiv A : \texttt{Type},\ B : \texttt{Type},\ p : A \times B.$$

Then:

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

Here A plays the role of C as well.

To show that defining p_1 to be $\texttt{unpack}(p, x)$ is reasonable, we use the computation rule to prove

$$a : A,\ b : B \vdash \texttt{unpack}((a, b), x) \equiv a$$

One can define p_2 similarly.

Sums

Next we define sums, which are like disjoint unions or coproducts:

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

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

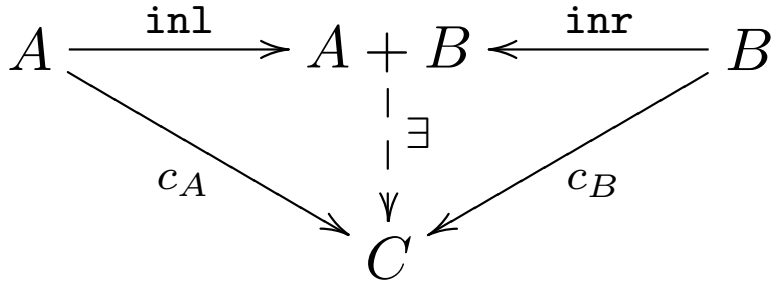
$$\frac{\Gamma \vdash C : \mathtt{Type} \qquad \Gamma \vdash p : A + B \qquad \Gamma, x : A \vdash c_A : C \qquad \Gamma, y : B \vdash c_B : C}{\Gamma \vdash \mathtt{case}(p, c_A, c_B) : C}$$

$$\frac{\Gamma \vdash C : \mathtt{Type} \qquad \Gamma \vdash a : A \qquad \Gamma, x : A \vdash c_A : C \qquad \Gamma, y : B \vdash c_B : C}{\Gamma \vdash \mathtt{case}(\mathtt{inl}(a), c_A, c_B) \equiv c_A[a/x] : C}$$

The notation $c_A[a/x]$ means that in the expression c_A, all free x's have been replaced by a's. There is also an analogous computation rule for `inr`.

Aside: Intuition and example

Intuitively, the rules for sums give the usual (weak) universal property:



Also, now we are able to construct some larger types, such as

$$F :\equiv (\mathtt{unit} + \mathtt{unit}) \times (\mathtt{unit} + \mathtt{unit}),$$

which has terms

$$t_{ll} :\equiv (\mathtt{inl}(\mathtt{tt}), \mathtt{inl}(\mathtt{tt})), \quad t_{lr} :\equiv (\mathtt{inl}(\mathtt{tt}), \mathtt{inr}(\mathtt{tt})),$$
$$t_{rl} :\equiv (\mathtt{inr}(\mathtt{tt}), \mathtt{inl}(\mathtt{tt})), \quad t_{rr} :\equiv (\mathtt{inr}(\mathtt{tt}), \mathtt{inr}(\mathtt{tt})).$$

Then $\mathtt{case}(p, t_{ll}, t_{lr})$ defines an element of F that depends on an element $p : \mathtt{unit} + \mathtt{unit}$.

Functions

Function types are also a primitive notion:

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

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : A \to B}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B}$$

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.b)(a) \equiv b[a/x] : B}$$

Recall that $\lambda x.b$ should be thought of as the function $x \mapsto b$.

Note that $f : A \to B$ means that f is a term whose type is $A \to B$, and that this matches usual mathematical notation.

Aside: Example

We saw above that $p : \mathtt{unit} + \mathtt{unit} \vdash \mathtt{case}(p, t_{ll}, t_{lr}) : F$, where $F :\equiv (\mathtt{unit} + \mathtt{unit}) \times (\mathtt{unit} + \mathtt{unit})$.

Thus we get a term

$$\lambda p.\mathtt{case}(p, t_{ll}, t_{lr}) : \mathtt{unit} + \mathtt{unit} \to F$$

which we think of as a function from a two element set to a four element set.

Writing f for this function, we know that

$$f(\mathtt{inl}(\mathtt{tt})) \equiv t_{ll} \quad \text{and} \quad f(\mathtt{inr}(\mathtt{tt})) \equiv t_{lr}.$$

Functions with multiple arguments

Intuitively, a function $A \times B \to C$ with two arguments can be thought of as a function $A \to C^B$. This is called currying.

The latter is more convenient in type theory, so we will often consider types such as $A \to B \to C$, which is parsed as $A \to (B \to C)$.

For $g : A \to B \to C$, we still write $g(a, b)$ for $g(a)(b)$.

As an example, one can see that

$$\lambda f.\lambda g.\lambda a.g(f(a)) : (A \to B) \to (B \to C) \to (A \to C)$$

defines a function which composes two given functions.

We will see another example in a moment.

The natural number type $\mathbb{N}$

Intuition: If C is a set, $c_0 \in C$ is an element and $g : \mathbb{N} \times C \to C$ is a function, then there is a function $f : \mathbb{N} \to C$ such that

$$f(0) = c_0 \quad \text{and} \quad f(n+1) = g(n, f(n)).$$

Formally:

$$\frac{}{\vdash \mathbb{N} : \mathtt{Type}} \qquad \frac{}{\vdash 0 : \mathbb{N}} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathtt{succ}(n) : \mathbb{N}}$$

$$\frac{\Gamma \vdash C : \mathtt{Type} \qquad \Gamma \vdash c_0 : C \qquad \Gamma, m : \mathbb{N}, c : C \vdash c_s : C \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathtt{ind}(c_0, c_s, n) : C}$$

$$\frac{\Gamma \vdash C : \mathtt{Type} \qquad \Gamma \vdash c_0 : C \qquad \Gamma, m : \mathbb{N}, c : C \vdash c_s : C}{\Gamma \vdash \mathtt{ind}(c_0, c_s, 0) \equiv c_0 : C}$$

$$\frac{\Gamma \vdash C : \mathtt{Type} \qquad \Gamma \vdash c_0 : C \qquad \Gamma, m : \mathbb{N}, c : C \vdash c_s : C \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathtt{ind}(c_0, c_s, \mathtt{succ}(n)) \equiv c_s[n/m, \mathtt{ind}(c_0, c_s, n)/c] : C}$$

Recursion

Combining the above rules with the rules for function types, we see that in the appropriate context we can get a function

$$f :\equiv \lambda n.\mathtt{ind}(c_0, c_s, n) : \mathbb{N} \to C$$

If we write $g :\equiv \lambda m.\lambda c.c_s : \mathbb{N} \to C \to C$, then we have

$$f(0) \equiv c_0 \quad \text{and} \quad f(\mathtt{succ}(n)) \equiv g(n, f(n))$$

as expected.

Example

Let's construct the function $\mathbb{N} \to \mathbb{N}$ that doubles its input.

We want

$$\texttt{double}(0) = 0 \quad \text{and} \quad \texttt{double}(\texttt{succ}(n)) = \texttt{succ}(\texttt{succ}(\texttt{double}(n))).$$

Here's the proof that such a function exists:

$$\dfrac{\dfrac{}{\vdash \mathbb{N} : \texttt{Type}} \quad \dfrac{}{\vdash 0 : \mathbb{N}} \quad \dfrac{\dfrac{\dfrac{}{m : \mathbb{N},\, c : \mathbb{N} \vdash c : \mathbb{N}}}{m : \mathbb{N},\, c : \mathbb{N} \vdash \texttt{succ}(c) : \mathbb{N}}}{m : \mathbb{N},\, c : \mathbb{N} \vdash \texttt{succ}(\texttt{succ}(c)) : \mathbb{N}} \quad \dfrac{}{n : \mathbb{N} \vdash n : \mathbb{N}}}{\dfrac{n : \mathbb{N} \vdash \texttt{ind}(0, \texttt{succ}(\texttt{succ}(c)), n) : \mathbb{N}}{\lambda n.\texttt{ind}(0, \texttt{succ}(\texttt{succ}(c)), n) : \mathbb{N} \to \mathbb{N}}}$$

So we define $\texttt{double} :\equiv \lambda n.\texttt{ind}(0, \texttt{succ}(\texttt{succ}(c)), n) : \mathbb{N} \to \mathbb{N}$.

Addition

We can also use recursion to define addition on $\mathbb{N}$, by taking C to be the type $\mathbb{N} \to \mathbb{N}$.

We let $c_0 :\equiv \lambda n.n : \mathbb{N} \to \mathbb{N}$, the identity function, thought of as $n \mapsto 0 + n$.

For $m : \mathbb{N}$ and $c : \mathbb{N} \to \mathbb{N}$, we let $c_s :\equiv \lambda n.\mathtt{succ}(c(n)) : \mathbb{N} \to \mathbb{N}$. Intuitively, if c is $n \mapsto m + n$, then c_s is $n \mapsto 1 + m + n$.

Now we define $\mathtt{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ to be $\lambda m.\lambda n.\mathtt{ind}(c_0, c_s, m)(n)$.

We abbreviate $\mathtt{add}(m, n)$ as $m + n$.

Coq

There are more rules of type theory, but let's delay talking about them for now.

Instead, I'd like to illustrate how what we know so far works in Coq, a proof assistant based on type theory.

Coq was named after Thierry Coquand, and means "rooster" in French. It also alludes to "Coc", the "calculus of constructions".

Unfortunately, much of the notation is different from what we've described, but the ideas are very similar.

Dependent types and products

One of the key ideas in Martin-Löf type theory is that of dependent types.

Examples:

$$A : \texttt{Type},\ B : \texttt{Type} \vdash\ A \to B : \texttt{Type}$$

$$a : A,\ B : A \to \texttt{Type} \vdash\ B(a) : \texttt{Type}$$

Note that we are thinking of `Type` as being a type itself, and we could use recursion to define a function $B : \mathbb{N} \to \texttt{Type}$ that sends n to $\mathbb{N} \times \cdots \times \mathbb{N}$ (n factors).

Thinking of a dependent type $B : A \to \texttt{Type}$ as a family of sets indexed by an indexing set, we assert that the product exists.

A term of $\prod_{x:A} B(x)$ should be thought of as a function f sending each $x : A$ to an $f(x) : B(x)$. Notice that both the value of $f(x)$ and the type of $f(x)$ depend on x.

Dependent Products

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma, x : A \vdash B : \texttt{Type}}{\Gamma \vdash \prod_{x:A} B : \texttt{Type}}$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : \prod_{x:A} B}$$

$$\frac{\Gamma \vdash f : \prod_{x:A} B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[a/x]}$$

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.b)(a) \equiv b[a/x]}$$

Functions

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma, \qquad \vdash B : \texttt{Type}}{\Gamma \vdash A \to B : \texttt{Type}}$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : A \to B}$$

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B}$$

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.b)(a) \equiv b[a/x]}$$

Dependent Sums

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma, x : A \vdash B : \texttt{Type}}{\Gamma \vdash \sum_{x:A} B : \texttt{Type}}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \sum_{x:A} B}$$

$$\frac{\Gamma \vdash C : \texttt{Type} \qquad \Gamma \vdash p : \sum_{x:A} B \qquad \Gamma, x : A, y : B \vdash c : C}{\Gamma \vdash \texttt{unpack}(p, c) : C}$$

$$\frac{\Gamma \vdash C : \texttt{Type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x] \qquad \Gamma, x : A, y : B \vdash c : C}{\Gamma \vdash \texttt{unpack}((a, b), c) \equiv c : C}$$

What's the non-dependent special case of this? $A \times B$!

What's left?

We've now covered almost all of the rules of type theory, with the exception of:

Some structural rules, which are fairly obvious.

Identity types, which are subtle and which will be discussed in a future lecture.

Differences from ordinary set theory: No intersections, unions, or subsets. Every term has exactly one type.

Nevertheless, with this foundation, all of the usual constructions of mathematics can be done. For example, one can construct the real numbers and do analysis.

Type theory as logic

Instead of thinking of types as sets, we can think of them as propositions.

And we interpret $q : Q$ to mean that Q is true. More generally, a judgement

$$x_1 : P_1, \ldots, x_n : P_n \vdash q : Q$$

is interpreted as saying that if $P_1, \ldots, P_n$ are true, then Q is true.

Remarkably, the laws we have given are exactly what is needed to do predicate logic!

Conjunction

For example, the product $P \times Q$ is inhabited if and only if both P and Q are, so it is reasonable to interpret the product $P \times Q$ as "P and Q". The rules

$$\frac{\Gamma \vdash P : \mathtt{Type} \qquad \Gamma \vdash Q : \mathtt{Type}}{\Gamma \vdash P \times Q : \mathtt{Type}} \qquad \frac{\Gamma \vdash a : P \qquad \Gamma \vdash b : Q}{\Gamma \vdash (a, b) : P \times Q}$$

$$\frac{\Gamma \vdash R : \mathtt{Type} \qquad \Gamma \vdash p : P \times Q \qquad \Gamma, x : P, y : Q \vdash c : R}{\Gamma \vdash \mathtt{unpack}(p, c) : R}$$

tell us that we can construct products; that if P and Q are true, so is $P \times Q$; and that if $P \times Q$ is true and P and Q implies R, then R is true.

The last rule gave us projection maps $P \times Q \to P$ and $P \times Q \to Q$, which show that if $P \times Q$ is true, then both of P and Q are true.

Disjunction, implication, etc

Similarly, the other rules match standard logical constructions:

Types	$\longleftrightarrow$	Propositions
$P \times Q$	$\longleftrightarrow$	P and Q
$P + Q$	$\longleftrightarrow$	P or Q
$P \to Q$	$\longleftrightarrow$	P implies Q
$\varnothing$	$\longleftrightarrow$	false
`unit`	$\longleftrightarrow$	true
$\prod_{x:A} P(x)$	$\longleftrightarrow$	$\forall x P(x)$
$\sum_{x:A} P(x)$	$\longleftrightarrow$	$\exists x P(x)$

This is called the Curry-Howard correspondence.
Note that there are *three* kinds of implication in type theory.

Induction

The rules for $\mathbb{N}$, when slightly generalized to replace $C : \mathtt{Type}$ with a dependent type, give exactly the usual rule for proof by induction:

$$\frac{\Gamma, m : \mathbb{N} \vdash C : \mathtt{Type} \qquad \Gamma \vdash n : \mathbb{N} \\ \Gamma \vdash c_0 : C[0/m] \qquad \Gamma, m : \mathbb{N}, c : C \vdash c_s : C[\mathtt{succ}(m)/m]}{\Gamma \vdash \mathtt{ind}(c_0, c_s, n) : C[n/m]}$$

If $C(0)$ is true and $C(m)$ implies $C(m+1)$, then $C(n)$ is true for each n.

Example of a proof

Theorem. For any P and Q, $P \to (Q \to P)$.

Proof. Assume P. We need to prove $Q \to P$. So assume Q. P follows by assumption. □

Formally:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p : P \vdash p : P}}{p : P, q : Q \vdash p : P}}{p : P \vdash \lambda q.p : Q \to P}}{\vdash \lambda p.\lambda q.p : P \to (Q \to P)}$$

(All lines should start with the context $P : \texttt{Type}$, $Q : \texttt{Type}$.)

Show this and other examples in Coq.

Many non-trivial theorems have been formalized, including $\pi_1(S^1) \cong \mathbb{Z}$ (2013), the four color theorem (2004), and the Feit-Thompson theorem (2012), all in Coq.
A C compiler has been written in Coq and proved correct.

Type theory versus ZFC

The usual foundation for mathematics is Zermelo-Fraenkel set theory, a system of axioms expressed in first order logic.

This is very different from type theory, in which the logic and the set theory are at exactly the same level.

Benefits of type theory

- Type checking of expressions is computable (as we saw in Coq). In other words, **proofs can be checked**, even when the detailed steps aren't given.
- Judgemental equality of terms is computable. In fact, you can just apply simplification rules in any order and you are guaranteed to reach a canonical form.
- From this, you can **prove** that type theory is consistent!
- From type theory proofs, you can automatically extract algorithms.

Intuitionistic logic

The logic we've just described is intuitionistic or constructive by default. You can't prove the law of excluded middle "P or not P" (i.e., show that $P + (P \to \varnothing)$ is inhabited) from the rules we have.

This is another **benefit**. It means that the theorems you prove are true in more models. In particular, they are true in topoi and in homotopical situations.

And if you want to use LEM, you can simply assume it as an axiom.

Surprisingly, the most natural form of the axiom of choice is a tautology in type theory! But there are other versions that can be assumed as an axiom when needed.

Moreover, Voevodsky's "Univalence" axiom *does* hold in homotopical situations, and can often be used in place of LEM and AC. When this axiom is assumed, then one speaks of "homotopy type theory".

Possible topics for future talks

- Identity types
- Univalence
- Homotopy type theory
- $\pi_1(S^1) \cong \mathbb{Z}$
- Models of (homotopy) type theory, completeness
- Doing mathematics in type theory: category theory, set theory, analysis
- More about Coq

References

The Homotopy Type Theory book produced by the Univalent Foundations Program at the IAS is freely available and is an excellent source.

To get started, I found it very helpful to read the slides from Mike Shulman's series of seven lectures in a seminar at UCSD. In particular, this lecture was heavily influenced by the second lecture of that series.

The Coq homework exercises for that lecture were a transforming experience.

Thanks for listening!