Topic 2: Typed lambda calculus

May 8, 2014

Some theorems in Coq

Before we start the formal development of type theory, it should be helpful to have some ideas of what the upcoming type-theoretic rules are going to mean. Consider the following examples of code in the interactive theorem prover Coq:

```
(* Lawvere's fixed point theorem in Coq *)

Lemma eq_app (A B : Type) (f1 f2 : A → B) (a:A) : (f1 = f2) → (f1 a = f2 a).
Proof.
        intro assumption.
        rewrite assumption.
        reflexivity.
Qed.

Theorem Lawvere (X Y : Type) (f : X → X → Y) : (forall g : X → Y, exists x : X, g =
    f x) → (forall h : Y → Y, exists y : Y, h y = y).
Proof.
        intro assumption.
        intro h.
        specialize (assumption (fun x ⇒ h (f x x))).
        destruct assumption.
        exists (f x x).
        apply (eq_app X Y (fun x ⇒ h (f x x)) (f x) x).
        apply H.
Qed.
```

The lemma states the following: given that A and B are taken to be types, $f1$ and $f2$ functions $A \to B$ and a an element of A, then an equality $f1 = f2$ implies an equality $f1(a) = f2(a)$. The theorem can be parsed in a similar way.

So what exactly is going on here? What do these symbols and keywords stand for? And what are the inference rules that have been applied in order to prove this theorem?

The answer is that Coq is based on a type theory called the Calculus of Constructions. We will not study this exact type theory in this course, so suffice it to say that it has many features in common with HoTT. We now start our formal developments with the introduction of two very simple type theories: a trivial one containing just a bare minimum of structure, and then we expand it to the simply typed lambda calculus in which one can reason about functions. The basic structures occurring in these type theories will be just as in the above Coq proofs, so it will help to keep in mind how these are structured.

HoTT will be an extension of these two very basic type theories.

Trivial type theory

In this basic (and completely inexpressive) type theory—and also in all the upcoming ones—there are five kinds of *judgements*:

$$\Gamma \ \texttt{ctx}, \qquad \Gamma \vdash A : \texttt{Type}, \qquad \Gamma \vdash a : A, \tag{1}$$

$$\Gamma \vdash A \equiv B : \texttt{Type}, \qquad \Gamma \vdash a \equiv b : A. \tag{2}$$

The symbol '≡' is called *judgemental equality*. Note that a judgemental equality statement $A \equiv B :$ `Type`, does not mean that $A \equiv B$ is a type; it rather indicates that A and B are judgementally equal *as types*. Similarly, $a \equiv b : A$ indicates that a and b are judgementally equal as elements of A.

The first kind of judgement expresses that the expression Γ is a well-formed *context*. In the above `Coq` examples, the context is given by all the stuff in brackets between the name of the statement and the colon separating it from the actual statement; a context represents a list of assumptions which can then be used in other judgements, in which Γ stands for a context. The turnstile symbol '⊢' is shorthand for the claim

In the list of assumptions given by Γ, the following is true: ...

and is denoted by an unbracketed colon in `Coq`. The stuff to the right of Γ represents the actual statement. The judgements in the second line are about *judgemental equality*; the reason to use the funny notation '≡' for judgemental equality is that we will later encounter another kind of equality denoted '='. Roughly speaking, judgemental equality states that two things are 'trivially equal' or 'equal by definition'.

Note that all of these explanations are intuitive interpretations of what the different judgements mean; we have not yet specified any formal rules on how to create or manipulate judgements. This is what we do now:

1. Context formation:

   The first context formation rule looks very trivial:

   $$\frac{}{\cdot \ \texttt{ctx}}$$

   There is also a rule to construct less trivial contexts:

   $$\frac{(x_1 : A_1, \ldots, x_n : A_n) \ \texttt{ctx}}{(x_1 : A_1, \ldots, x_n : A_n, B : \texttt{Type}) \ \texttt{ctx}}$$

   In each of these two rules, the stuff above the horizontal bar is a list of judgements which one already needs to have derived in order to formally conclude the new judgement below the bar. In this sense, we have written down an *inference rule*. In the first rule, this list is empty, while below the bar, '·' stands for the empty expression; so this rule says that the empty expression is always a context.

   In the second rule, B needs to be a new variable symbol distinct from the one that are already present; in the above `Coq` examples, this means that each new variable symbol in the context must be assigned a different name.

From now on, we will be sloppy about the use of brackets, and not distinguish between $x : A, y : B$ and $(x : A, y : B)$ and $(x : A)\ (y : B)$. As in the above examples, we can also abbreviate $(a : A, b : A)$ to $a, b : A$.

The remaining context formation rule is this:

$$\frac{x_1 : A_1, \ldots, x_{n-1} : A_{n-1} \vdash A_n : \mathtt{Type}}{(x_1 : A_1, \ldots, x_n : A_n)\ \mathtt{ctx}}$$

Here, A_n is an expression which may in principle contain the variables $x_1, \ldots, x_{n-1}$. However, in the basic type theories that we discuss today, there are no rules which one could apply in order to obtain such a *dependent type*.

2. Assumptions are derivable:

$$\frac{(x_1 : A_1, \ldots, x_n : A_n)\ \mathtt{ctx}}{x_1 : A_1, \ldots, x_n : A_n \vdash x_i : A_i}$$

This inference rule says that if one has a context which consists of a list of variables of certain types, then one can derive that each variable x_i has the type which is assumed in the context. Since there are no other ways for obtaining contexts other than applying the above context formation rules, it is guaranteed that every context is simply a list of typing assumptions, so that the present rule is always applicable.

3. Judgemental equality rules:

$$\frac{\Gamma \vdash A : \mathtt{Type}}{\Gamma \vdash A \equiv A : \mathtt{Type}} \qquad \frac{\Gamma \vdash A \equiv B : \mathtt{Type}}{\Gamma \vdash B \equiv A : \mathtt{Type}} \qquad \frac{\Gamma \vdash A \equiv B : \mathtt{Type} \quad \Gamma \vdash B \equiv C : \mathtt{Type}}{\Gamma \vdash A \equiv C : \mathtt{Type}}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \equiv a : A} \qquad \frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash b \equiv a : A} \qquad \frac{\Gamma \vdash a \equiv b : A \quad \Gamma \vdash b \equiv c : A}{\Gamma \vdash a \equiv c : A}\ .$$

These rules state that judgemental equality is an equivalence relation, both on types and on elements of types.

4. Judgemental equality preserves judgements:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A \equiv B : \mathtt{Type}}{\Gamma \vdash a : B} \qquad \frac{\Gamma \vdash a \equiv b : A \quad \Gamma \vdash A \equiv B : \mathtt{Type}}{\Gamma \vdash a \equiv b : B}$$

As already mentioned, these inference rules represent derivations of new judgements from already known judgements. Proving any sort of non-trivial statement requires many applications of inference rules; one can visualize these in a *proof tree*:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\ \mathtt{ctx}}}{(A : \mathtt{Type})\ \mathtt{ctx}}}{A : \mathtt{Type} \vdash A : \mathtt{Type}}}{\vdash (A : \mathtt{Type}, a : A)\ \mathtt{ctx}}}{A : \mathtt{Type}, a : A \vdash a : A}}{A : \mathtt{Type}, a : A \vdash a \equiv a : A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\ \mathtt{ctx}}}{(A : \mathtt{Type})\ \mathtt{ctx}}}{A : \mathtt{Type} \vdash A : \mathtt{Type}}}{\vdash (A : \mathtt{Type}, a : A)\ \mathtt{ctx}}}{A : \mathtt{Type}, a : A \vdash A : \mathtt{Type}}}{A : \mathtt{Type}, a : A \vdash A \equiv A : \mathtt{Type}}$$

$$A : \mathtt{Type}, a : A \vdash a \equiv a : A$$

What has happened here is that we have derived the simple judgement $A : \mathtt{Type}, a : A \vdash a \equiv a : A$ using the inference rules of our trivial type theory. Each horizontal bar stands for an application of one inference rule. Note that we already obtained the target statement on the left branch, but we can nevertheless derive it again, even making use of itself. Also, note that the left and the right branch actually coincide in the upper part.

Additional rules for simply typed lambda calculus

As one may imagine, our trivial type theory has very low expressivity in the sense that no interesting judgements can be derived. Over the following lectures, we will successively enhance it by adding more and more structure. We now starting to do this by adding *function types*. A function type represents the collection of all functions with given domain and codomain types. This allows us to talk about functions in our type theory, which thereby becomes the *simply typed lambda calculus*.

The *type former* for function types is denoted '$\to$'. The inference rules for function types are as follows:

1. Formation rules:

$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma \vdash B : \mathtt{Type}}{\Gamma \vdash A \to B : \mathtt{Type}} \qquad \frac{\Gamma \vdash A \equiv A' : \mathtt{Type} \qquad \Gamma \vdash B \equiv B' : \mathtt{Type}}{\Gamma \vdash (A \to B) \equiv (A' \to B') : \mathtt{Type}}$$

   This rule tells us that for every two types A and B, one can *form* the function type $A \to B$. In the expression $A \to B$, the arrow '$\to$' is a new symbol which has not been part of our trivial type theory. We also learn that two functions types $A \to B$ and $A' \to B'$ become judgementally equal as soon as their domain and codomain are.

   It is interesting to note that the usual notation for a function $f : A \to B$ can now be understood to be part of a typing judgement.

2. Introduction rules:

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : A \to B} \qquad \frac{\Gamma \vdash A \equiv A' : \mathtt{Type} \qquad \Gamma \vdash B \equiv B' : \mathtt{Type} \qquad \Gamma, x : A \vdash b \equiv b' : B}{\Gamma \vdash \lambda(x : A).b \equiv \lambda(x : A').b' : A \to B}$$

   The first rule tells us that if we have an expression b of type B which may possibly depend on $x : A$ (and on the other variables in Γ), then we may *introduce* a function $A \to B$ by regarding the expression b as a function of x. Constructing a function in this way is called *lambda abstraction*. We will usually abbreviate $\lambda(x : A).b$ to $\lambda x.b$, since the type of x is clear from the context.

   The second rule states that this construction preserves judgemental equality in all its arguments.

3. Elimination rules:

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

   This rules tells us how to apply a function. Since there is no function in the judgement below the bar, one can say that the function has been *eliminated*. Again, there is another rule

stating that this function elimination preserves equality judgements:

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

4. Computation rules:

   If b is an expression containing a variable x and a is another expression, then we write $b[a/x]$ for the new expression in which a has been substituted for x.

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

   This rule tells us how the introduction rule and the elimination rule interact: if one has a function obtained by lambda abstraction from an expression b and applies this function to an expression a, then the result is the same 'on the nose' as replacing every occurrence of the function variable x by a. Applying the computation rule in this way is also known as *β-conversion*.

5. Uniqueness principle:

   $$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f \equiv (\lambda x.f(x)) : A \to B}$$

   This principle tells us that any function is equal to evaluating that function on a variable and then considering the resulting expression as a function in that variable. This rule is also known as *η-conversion*.

As an example of how to apply these rules of inference, we now construct the composition of functions, which is the judgement

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

In words: if A, B, C are types, and we have function variables $f : A \to B$ and $g : B \to C$, then we obtain a function $\lambda x.g(f(x)) : A \to C$, which we call the *composition* of f and g.

If we write Γ for the context in (3), then that this Γ is indeed a context can be seen as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{A, B, C : \mathtt{Type} \vdash A : \mathtt{Type}} \qquad \dfrac{\vdots}{A, B, C : \mathtt{Type} \vdash B : \mathtt{Type}}}{A, B, C : \mathtt{Type} \vdash A \to B : \mathtt{Type}}}{(A, B, C : \mathtt{Type}, f : A \to B)\ \mathtt{ctx}}}{A, B, C : \mathtt{Type}, f : A \to B \vdash B : \mathtt{Type}} \qquad \dfrac{\vdots}{A, B, C : \mathtt{Type}, f : A \to B \vdash C : \mathtt{Type}}}{A, B, C : \mathtt{Type}, f : A \to B \vdash B \to C : \mathtt{Type}}}{(A, B, C : \mathtt{Type}, f : A \to B, g : B \to C)\ \mathtt{ctx}}$$

Here, we have abbreviated $(A : \mathtt{Type}, B : \mathtt{Type}, C : \mathtt{Type})$ to $A, B, C : \mathtt{Type}$ and already omitted those parts of the proof tree which are completely trivial. The right branch is the same as the left branch, so also that has been omitted. If one continues all branches all the way up, one will find

that each branch eventually ends with the empty context, as required for a valid derivation of a judgement.

The proof tree for (3) itself, i.e. for the construction of function composition, now looks like this:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma\ \texttt{ctx}}}{\Gamma \vdash A : \texttt{Type}}}{(\Gamma, x : A)\ \texttt{ctx}}}{\Gamma, x : A \vdash g : B \to C} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma\ \texttt{ctx}}}{\Gamma \vdash A : \texttt{Type}}}{(\Gamma, x : A)\ \texttt{ctx}}}{\Gamma, x : A \vdash f : A \to B} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma\ \texttt{ctx}}}{\Gamma \vdash A : \texttt{Type}}}{(\Gamma, x : A)\ \texttt{ctx}}}{\Gamma, x : A \vdash x : A}}{\Gamma, x : A \vdash f(x) : B}}{\dfrac{\Gamma, x : A \vdash g(f(x)) : C}{\Gamma \vdash \lambda x.g(f(x)) : A \to C}}$$

We already showed that $\Gamma\ \texttt{ctx}$ is a valid judgement. So in order to complete the branches of this proof tree, we can simply stick a copy of the above proof tree for $\Gamma\ \texttt{ctx}$ onto each leaf.

One can also consider this function composition itself as a function of f and g:

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

We omit the proof tree for this statement.

Higher function types. As we have just seen, it is frequently useful to form function types of function types. As a more basic example of this, consider $f : A \to (B \to C)$; this says that f is a function which takes an argument $x : A$ and returns a function which takes an argument $y : B$ and returns some value $f(x)(y) : C$. One usually abbreviate this $f(x)(y)$ to $f(x, y)$,

$$f(x, y) :\equiv f(x)(y),$$

and understands it as a function with two arguments. Since these kinds of functions are much more frequent than the ones of type $(A \to B) \to C$, one typically omits the brackets in $A \to (B \to C)$ and simply writes $A \to B \to C$.

Some metatheoretical properties

As the above `Coq` example might illustrate, the most important kind of judgements are the *typing judgements* $\Gamma \vdash x : A$. We are now going to discuss some *metatheoretical* properties of typing judgements. This refers to statements *about* judgements which cannot be formulated as judgements themselves, so that they live outside of the theory rather than inside it. We will not develop this metatheory in any detail and not prove any of the following remarks, but it is nevertheless very important to be aware of these considerations and results of this kind.

Rigorous theorems about some of the metatheoretical properties which we discuss can be found here:

Gilles Barthe, Thierry Coquand: *An Introduction to Dependent Type Theory.*

Do not try reading this at this stage—unless you already know a lot more type theory than we have introduced so far!

Adding common-sense inference rules. Besides the inference rules which we listed above, there are many others which can be derived from them by combining them into partial proof trees. There are yet other which cannot be derived from them, but they nevertheless seem so obvious and 'common sense' that it would be extremely strange if one were not allowed to use them. Take for example the *substitution rule*:

$$\frac{\Gamma \vdash a : A \quad \Gamma, x : A, \Delta \vdash b : B}{\Gamma, \Delta[a/x] \vdash b[a/x] : B[a/x]}$$

Here, also Δ is an arbitrary expression playing the role of a list of typing assumptions so that $(\Gamma, x : A, \Delta)$ is a context. The notation '$\Delta[a/x]$' stands for the same expression Δ, with every occurrence of the variable x replaced by the expression a; similarly for $b[a/x]$ and $B[a/x]$. Intuitively, this rule tells us that we can always substitute a free variable of type A by any expression of type A which can be formed in the present context. As soon as we switch to a more informal mode of reasoning, we will make judicious use of this rule and similar ones.

Assuming this additional inference rule modifies the type theory in the sense that this rule cannot be derived from the other ones. Nevertheless, it is not necessary for the derivation of any judgement: any judgement which can be derived using the substitution rule can also be derived without it. For this reason, there is no harm in using it, although it is—strictly speaking—not part of our type theory. All of this is very reminiscent of the cut elimination theorem in sequent calculus.

There are other rules of a similar flavor which can be assumed to hold, but which are not necessary for the derivation of any judgement. As soon as we switch to a more informal mode of reasoning in order to do actual mathematics within type theory, we will become guilty of secretly using such rules without making them explicit.

Every expression has at most one type. Type theories may or may not have the property that every expression in a given context has at most one type. More precisely, what we mean by this property is that if $\Gamma \vdash x : A$ and $\Gamma \vdash x : B$, then $\Gamma \vdash A \equiv B : \texttt{Type}$. It can be shown that the two type theories we have been setting up in this lecture do have this property, and most other type theories also enjoy this property at least in some approximate sense.

On the other hand, there are also expressions that are *ill-typed*: an expression x is ill-typed in a context Γ if there exists no A such that $\Gamma \vdash x : A$. Some expressions are ill-typed in any context. For example, there is no way to assign a type to the expression $f(f)$: in order to apply f, its type must be a function type $f : A \to B$; but then, its argument would have to be of type A, which it is not in this particular case.

As soon as one considers a type theory with *universes*—as we will do later on—becomes more subtle. Typically, while it does still hold in a certain approximate or intuitive sense, it is technically violated. We will learn more about this soon.

Judgemental equality judgements imply typing judgements. Any sensible type theory should have the property that if $\Gamma \vdash a \equiv b : A$, then $\Gamma \vdash a : A$ and $\Gamma \vdash b : A$. This is indeed the case for the type theories that we have been looking at so far, and will also be true in all the extensions that we will be considering.

Contexts in judgements are contexts. A similar consistency property is that if $\Gamma \vdash x : A$ is a typing judgement, then necessarily Γ `ctx`. Also, if $\Gamma \vdash x \equiv y : A$, then Γ `ctx` as well. We also

have that if $(\Gamma, x : A)$ $\texttt{ctx}$, then also $\Gamma \vdash A : \texttt{Type}$, since the latter is the only way to obtain that $(\Gamma, x : A)$ $\texttt{ctx}$. This property is important for the above introduction and elimination rules for function types: strictly speaking, the introduction rule should be written as

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

But the metatheoretical properties discussed here guarantee that $\Gamma \vdash A : \texttt{Type}$ and $\Gamma \vdash B : \texttt{Type}$ are valid judgements as soon as $\Gamma, x : A \vdash b : B$ is so.

Type checking: typing judgements have canonical proofs. The notion of *type checking* may be familiar from programming languages which have types: it is a verification guaranteeing that a certain program, which for example takes a number and outputs a string, does indeed comply with this specification. For us, type checking means determining whether an alleged typing judgement $\Gamma \vdash x : A$ is indeed valid or not.

In other words, type checking asks whether there exists a proof tree showing that the given $\Gamma \vdash x : A$ is indeed a valid judgement. In the simply type lambda calculus, and also in many other type theories, there is a simple algorithm to do that which can be illustrated by example as follows. In the judgement,

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

the 'outermost' part of the expression which is to be type-checked is λf; therefore, any derivation of this judgement must necessarily use the introduction rule for functions as its final step, since this is the only rule which creates lambda abstractions. Before applying this step, we therefore are dealing with the alleged judgement

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

which remains to be type-checked. The same argument can be applied two more times, and we then need to check whether

$$A, B, C : \texttt{Type}, f : A \to B, g : B \to C, x : A \vdash g(f(x)) : C$$

is a valid judgement. But even from here, we can continue in a similar way: the only way to obtain a function application in an expression is by using the elimination rule for functions. Continuing up in this way, one can reconstruct the entire proof tree.

So, at least intuitively, the expression x in a typing judgement $\Gamma \vdash x : A$ contains complete information about its entire proof tree! (Again, there are some subtleties here, since the proof tree for a typing judgement is not strictly unique.)