Inside *Coq*'s Type System [1]

Pierre Castéran

Beijing, August 2009

[1] This lecture corresponds mainly to the chapters : "Dependent Products or Pandora's Box" and 5 : "Everyday Logic" of the book.

In this class, we shall present some important aspects of *Coq*'s type system, and how it allows us to express such concepts as :

- Logical (basic and derived) rules of inference,
- Properties of programs (and their proof)
- Specifications of programs
- Advanced type paradigms : polymorphism, dependent types, higher order types, etc.

This class is mainly based on examples. For more precise definitions, please refer to *Coq*'s user manual or/and Coq'Art book.

It is important to know that *Coq*'s type system (the Calculus of Inductive Constructions) has the following properties (necessary to its soundness) :

- Every well-formed term has (at least) a type,
- Every well-formed type is also a term of the calculus

One quickly infers that every well-formed type has also a type, which in turns has a type, and so on ... The type of a type is always a constant, which is called a sort.

Some types and their sort

$$\begin{aligned}
\texttt{nat} &: \texttt{Set} \\
\texttt{bool} &: \texttt{Set} \\
\texttt{nat}\to\texttt{nat} &: \texttt{Set} \\
\texttt{True} &: \texttt{Prop} \\
\forall\texttt{n :nat, n < n+1} &: \texttt{Prop} \\
\texttt{Set} &: \texttt{Type}_0 \\
\texttt{Prop} &: \texttt{Type}_0 \\
\texttt{Prop}\to\texttt{Prop} &: \texttt{Type}_0 \\
\texttt{nat}\to\texttt{Prop} &: \texttt{Type}_0 \\
\texttt{Type}_i &: \texttt{Type}_{i+1}
\end{aligned}$$

The *universe levels* (in green) are used *internally* for avoiding inconsistencies like `Type :Type`. They are not available to the user, and not printed.

Typing judgements

The heart of the *Coq* system is a *type checker*, *i.e.* an algorithm which checks whether some expression is well formed according to some set of declarations. This kind of algorithm is part of almost every programming language's compiler.

The main difference is that *Coq*'s typing system is much more powerful than "ordinary" programming languages such as `C`, `Java`, and even `Standard ML`, `OCaml`, etc.

Let Γ be a context (set of variable declarations, constant definitions,...), t be some term and A some type . A *typing judgement* is a formula $\Gamma \vdash t : A$ which is read " Taking into account the declarations of Γ, the term t is well-formed and its type is A ".
For instance, if the context Γ contains the declarations :

```
S :  nat → nat
plus : nat → nat → nat
f : list nat → nat
l : list nat
i : nat
```

Then the judgement $\Gamma \vdash S(f\ l) + i : nat$ expresses that the term S(f l)+i is well formed in the context Γ, and that its type is nat.

Sequents and typing judgements

Let us consider some judgement $\Gamma \vdash t : A$, where $\Gamma \vdash A :$ `Prop`.
We say that in the context Γ, the term t is a proof (a *proof term*) of the proposition A in the context Γ.
For example, let us consider the following context :

```
P : Prop
Q : Prop
H : P → Q → R
H0 : P → Q
```

Then the term fun (p :P) ⇒ H p (H0 p) is a term of type P → R (i.e. a proof [term] of the proposition P → R under the hypotheses H and H0).
The implication $A \rightarrow B$ appears to be the type of functions which associate to any proof of A some proof of B.

A first look at the Curry-Howard correspondance

Concept	Programming	Logic
$v : A$	Variable declaration	hypothesis
Γ	Variable declarations	hypotheses
$\Gamma \vdash t : A$	t has type A (under Γ)	t is a proof of A (assuming Γ)
$A \rightarrow B$	functional type	implication
	type checking	proof verification

This is the real way *Coq* :

- checks whether a given proof of A is correct (with a type checking algorithm),
- stores the proofs of theorems in the form of well-typed terms.

Dependent products

The dependent product construct provides a unified formalism for implication, universal quantification, types of programming languages (including polymorphism and much more).

A *dependent product* is a type of the form $\forall\, v : A,\ B$ where A and B are types and v is a *bound variable* whose scope covers B. Thus, the variable v may have free occurrences in B.

For instance, in the type $\forall$ `n:nat, 0<n`$\rightarrow\forall$ `p:nat, 0 < n+p`, the green occurrences of `n` are bound by the most external $\forall$, whereas they are free in the term `0<n`$\rightarrow\forall$ `p:nat, 0 < n+p`

Dependent product and functional types

We have already seen some types of the form $A \rightarrow B$ during previous lectures. In fact $A \rightarrow B$ is just an abbreviation for the product $\forall\, v : A,\ B$, *where* v *doesn't have any free occurrence in* B. Here are some examples :

Expanded form	abbreviated form
$\forall\, n : \mathtt{nat},\ \mathtt{nat}$	$\mathtt{nat} \rightarrow \mathtt{nat}$
$\forall$ n :nat ,$\forall$ H : P n, Q n	$\forall$ n :nat, P n $\rightarrow$ Q n

That explains why the same tactics intro and apply work the same way on implication and universal quantifications.

Typing rules for dependent products

The well-formedness of a dependent product $\forall\, v : A,\ B$ as well as its type, depends of the sorts s of A and s' of B.
Variations on s and s' will allow us to show the big expressive power of *Coq*'s type system.
We shall use examples for presenting the most interesting cases.

Some examples

s	s'	Example	
Prop	Prop	$0 < n \rightarrow 0 < n^* n$: Prop	implication
Set	Set	nat $\rightarrow$ nat : Set	simple functional type
Set	Prop	$\forall$ n:nat, $0 \leq n$: Prop	first order quantifier
Set	Type	nat $\rightarrow$ Prop : Type	type of a predicate on nat
Type	Type	Prop$\rightarrow$ Prop$\rightarrow$ Prop : Type	type of $\vee$, $\wedge$, etc.

There was a mistake in the previous version of this slide (distributed on the keys) : fourth line, third column of this table : it was written Set instead of Prop.

Polymorphism

A polymorphic function such as `rev` (list reversing) has type
$\forall$ `A:Type, list A` $\rightarrow$ `list A`.
This polymorphic type is a dependent product, formed with sorts $s = s' =$ `Type`.

Similarly, the type of the `fold_right` functional is
$\forall$ A B : Type, (B $\rightarrow$ A $\rightarrow$ A) $\rightarrow$ A $\rightarrow$ list B $\rightarrow$ A.

Dependent types

Assume we want to consider programs which operate on binary words. It is natural to consider a type that is parameterized by the word size.
We will show later how to define a constant `binary_word` of type nat $\rightarrow$Set. The formation of this type is allowed by *Coq*'s type system (with sorts $s = \texttt{Set}$ and $s' = \texttt{Type}$).
This declaration will allow us to write such definitions as :

```
Definition short : Set := binary_word 32.

Definition long : Set := binary_word 64.
```

A function for bitwise xor-ing two vectors of the same length will have the following type :

```
∀ n:nat, binary_word n →
         binary_word n→
         binary_word n.
```

A concatenation function will have the type below :

```
∀ n p:nat, binary_word n →
           binary_word p →
           binary_word (n+p).
```

Typing rules for dependent products

Let $\forall\, v : A,\, B$ be some well-formed product (w.r.t. some context Γ). As for other constructs, we have to give typing rules for building and applying terms of this type.
It is important to notice that, since the dependent product generalizes the arrow $\longrightarrow$ and the first-order universal quantifier, these rules have to be compatible with the rules for functional types and implication.

Elimination rule

As for $A \to B$, the elimination rule for $\forall\, v : A,\ B$ is the *application rule* :

$$\textbf{App}\quad \frac{\Gamma \vdash f\ :\ \forall\, v : A,\ B \qquad \Gamma \vdash a\ :\ A}{\Gamma \vdash f\ \ a\ :\ B\{v/a\}}\ .$$

Notation : $B\{v/a\}$ is the result of replacing the free occurrences of v in B with a.
Associated tactic : apply f.

Notice that f is a function, which receives the argument a in the rule above.

Introduction rule

Let Γ be some context, and some dependent product $\forall\, v : A,\ B$ well-formed in Γ. Then we state the introduction rule :

$$\frac{\Gamma, v : A \vdash t \ : \ B}{\Gamma \vdash \texttt{fun}\, v \ : \ A \Rightarrow t \ : \ \forall\, v : A,\ B}$$

Notice that this rule is an extension of the introduction rule for arrow types :

$$\frac{\Gamma, v : A \vdash t \ : \ B}{\Gamma \vdash \texttt{fun}\, v \ : \ A \Rightarrow t \ : \ A \rightarrow B}$$

The associated tactic is intro v.

Advanced Examples

We have already seen how to use dependent products in first-order logic.
It is interesting to consider other uses of this construct :

- polymorphic functions,
- types depending on data,
- function specifications,
- higher-order logic :representation of rules, and construction of derived rules, reasoning on functions and predicates, etc.
- definition of new types constructs
- etc.

Polymorphism

The module `List` of the Standard Library contains a definition of a constant list : Type→Type. Lists can be built with the help of two constructors [2] :

```
@nil : ∀ A : Type, list A
@cons : ∀ A : Type, A → list A → list A
```

Thus the following terms are well typed :

```
list nat : Set
list Prop : Type
@nil nat : list nat
@cons nat : nat → list nat → list nat
@cons nat 3 (@cons nat 8 (@nil nat)) : list nat
 plus::mult::minus::nil : list (nat → nat → nat)
```

[2] The symbol `@` is used here to write terms with all their implicit arguments

A short remark about notation

Let us consider again the type of cons :

```
About cons.
```
cons : ∀ A : Type, A → list A → list A
Argument A is implicit . . .

This answer means that, among the three arguments of cons :

- A type A,
- an element of type A,
- a list of type list A

the first one can be deduced from the other arguments. Quite similarly, the argument of nil can be deduced from the type of a list in which it appears.

The following notations are equivalent in *Coq* :

```
@cons nat 3 (@cons nat 5 (@nil nat))
cons 3 (cons 5 nil)
3::5::nil
```

On the other hand, if cons or nil don't receive enough arguments, it is necessary to make some type information explicit, as in the following examples :

```
@nil                @cons
@nil nat            (nil (A:=nat))
@cons  nat          (cons (A:=nat))
```

Here are some examples of use of polymorphic types :
Since rev has the type $\forall$ A : Type, list A $\rightarrow$ list A, and its argument A is implicit, the following terms are well typed :

```
@rev nat :  list nat → list nat
@rev nat (1 :: 2 :: 3 :: nil): list nat
rev (1 :: 2 :: 3 :: nil): list nat
```

We notice that the first argument of rev is used to instantiate the polymorphic parameter A.

The following definition is correct too :

```
Definition add_r {A:Type}(a:A)(l:list A) : list A :=
   l ++ (a :: nil).

Eval compute in add_r 4 (1::2::3::nil).
1::2::3::4::nil : list nat

Eval compute in map (fun f => f 9 5)
                    (add_r mult (plus::minus::nil)).
14::4::45::nil : list nat

Check add_r mult.
 add_r mult :
  list (nat →nat→nat) → list (nat→nat→nat)
```

Logical rules

Higher order logic (quantification on predicates and functions) allows us to represent logical rules inside the *Coq* system, as well as induction schemes.
Here are some examples :

```
or_introl : ∀ A B:Prop, A → A ∨ B
False_ind :  ∀ A:Prop, False → A
ex_intro  :  ∀ (A : Type) (P : A → Prop) (x : A),
               P x → ex P
nat_ind :  ∀ P : nat → Prop,
           P 0 →
           (∀ n : nat, P n → P (S n)) →
           ∀ n : nat, P n
nat_ind (fun n:nat ⇒ 0 ≤ n) (le_n 0):
 (∀ n : nat, 0 ≤ n → 0 ≤ S n) → ∀ n : nat, 0 ≤ n
```

One can also prove and use new "derived" rules.

```
Lemma or_False : ∀ P: Prop, P ∨ False → P.
Proof.
 intros  P [H | ff];[assumption | destruct ff].
Qed.

Lemma  or_False' : ∀ (A: Set)(P : A → Prop),
                   (∀ x:A, P x ∨ False) →
                   ∀ x:A, P x.
Proof.
 intros A P H x.   (*  P x  *)
 apply or_False.   (*  P x  ∨ False *)
 apply H.
Qed.
```

Here is an interesting example, we shall study as an exercise :

```
Lemma exm_double_neg : (∀ P:Prop, P ∨ ∼P) →
                       (∀ A:Prop, ∼∼A → A).
Proof.
 intros  Exm A H.
```

1 subgoal

Exm : ∀ P : Prop, P ∨ ∼ P
A : Prop
H : ∼∼ A
==========
A

```
(* (Exm A) is a term of type  A ∨ ∼A. *)
 destruct (Exm A) as [a|a'].
(* end of proof as an exercise *)
```

```
Lemma double_neg_exm : (∀ A:Prop, ~ ~ A → A) →
                       (∀ P:Prop, P ∨ ~P).
Proof.
 intros D P.
```

D : ∀ A : Prop, ~ ~ A → A
P : Prop
==========
P ∨ ~ P

```
apply D.
```

D : ∀ A : Prop, ~ ~ A → A
P : Prop
==========
~ ~ (P ∨ ~ P)

...

One can also create new connectives and quantifiers :

```
Definition ex_or (A B : Prop) :=  A ∧ ~ B ∨
                                  B ∧ ~ A.

Definition exists_one {A:Type} (P:A→Prop) : Prop :=
  ∃ a:A, P a ∧ ∀ b:A, P b → a = b.
```

```
Goal  exists_one  (fun i : nat ⇒ ∀ j:nat, i ≤ j).
Proof.
 unfold ex1.
```

1 subgoal

==========

exists a : nat,
(forall j : nat, a <= j) ∧
(forall b : nat, (forall j : nat, b <= j) -> a = b)

```
 exists 0; split.
```

2 subgoals

==========

∀ j : nat, 0 ≤ j

subgoal 2 is:
∀ b : nat, (∀ j : nat, b ≤ j) → 0 = b

```
Lemma ex_or_imp : forall P Q, ex_or P Q -> P -> ~ Q.
Proof.
 unfold ex_or;tauto.
Qed.
```

```
Lemma exists_one_exists :  forall (A:Type)(P:A->Prop),
       exists_one P -> exists x, P x.
Proof.
 intros A P H;destruct H as [x [H1 _]];exists x; auto.
Qed.
```

Just for the fun

It is possible to use the formation rules of dependent products to define propositions [resp. predicates] as universal quantification on *all propositions (including themselves)*. This kind of definition is called *impredicative*. We give them as material for some exercises, but they are not used in *Coq* developments.

```
Definition my_False : Prop := ∀ P: Prop, P.

Definition my_or (A B:Prop): Prop :=
  ∀ C:Prop, (A → C) → (B → C) → C.
```

The rule of conversion

Let us consider again our previous declarations :

```
Parameter binary_word : nat → Set.

Definition short : Set := binary_word 32.
Definition long : Set := binary_word 64.

bitwise_xor :∀ n:nat, binary_word n →
                      binary_word n→
                      binary_word n.

binary_word_concat: ∀ n p:nat, binary_word n →
                               binary_word p →
                               binary_word (n+p).
```

Let us consider the term binary_word_concat 32 32. According to our typing rules, its type is : binary_word 32 → binary_word 32 → binary_word (32 + 32).
However, it is possible to type the following definition :

```
Definition short_concat : short → short → long :=
           binary_word_concat 32 32.
```

Coq's type system has an important property : since types are terms too, computation rules (simplification, constant unfolding, evaluation, etc.) apply also to types.

Moreover, the *rule of conversion* tells us that if some term t has type A, and terms A and B can be reduced into the same term/type C then t has type B too.

Our definition of short_concat uses this property, through the expansion of the constants short, long and the simplification of $32 + 32$ into 64.

Function Specifications

The following dependent product is allowed in *Coq* :

$\forall$ `A : Type, (A` $\rightarrow$ `Prop)` $\rightarrow$ `Type : Type`

This is the type of the constant sig. This constant is often used under the form sig (fun x :$A \Rightarrow t$), abbreviated into $\{$x :$A \mid t\}$. In fact, sig P is an inductive type, whose unique constructor is :

`exist` :$\forall$ `a:A,` P `a` $\rightarrow$ `sig` P

That means that a term of type sig P is built from a data $a : A$ and a proof of $P\ a$.

Let us consider for instance the following type :

$$\forall\ \texttt{n : nat, } 2 \leq \texttt{n} \rightarrow \texttt{\{p:nat | n} \leq \texttt{p} \wedge \texttt{prime p = true \}}$$

This kind of type is called a *strong specification*. This example defines the type of functions which take as argument a natural number n such that $2 \leq n$ and return a number p, *together with a proof that* $n \leq p$ *and* p *is a prime number*. Such a function is called a *certified program*.

The constant sumbool of type `Prop→Prop→Set` allows us to specify certified decision functions.
Formally, if A and B are two propositions, then sumbool A B (also written $\{A\}+\{B\}$) is the type of terms of one of the two forms :

- left B t_A where $t_A : A$
- right A t_B where $t_B : B$

For instance, the following type :

```
∀ (A:Type)(R:A → A → Prop),
 total_order A R →
 (∀ a b:A, {R a b} +{~R a b}) →
 ∀ (l:list A),
  {l' : list A | sorted R l' ∧ permutation l l'}
```

is a specification of polymorphic list sorting functions (the definition of the predicates total_order, sorted and permutation is left as an exercise).

Certified programs are often built interactively. Though they are hardly readable mixtures of functional programming and proofs, they enjoy some interesting properties :

- their interactive development is guided by the specification, so that they are built and proved at the same time,
- the proof of correction is inside the term,
- *extraction algorithms* are able to erase the logical arguments and return safe functional programs (for `Haskell`, `OCaml` etc.)

Using the conversion rule

We have already seen the conversion rule, which allows to consider computations on types (see documentation and/or the book (chapter 2) for a presentation of *Coq*'s computation rules : β, δ, ι, and ζ rules).
Using this rule in a goal-directed proof allows us to change the current goal's conclusion A for some type B that is convertible with A and better suited for applying some tactic. This conversion is sometimes automatic, driven by the tactic (*e.g.* destruct, induction, rewrite . . .at, and often apply.)
In other cases, auxiliary tactics like pattern or change are used to force the conversion.

```
Parameter odd : nat→ Prop.
Lemma odd_S_2n : ∀ n:nat, odd (1 +  (2 * n)).
Admitted.
Hint Resolve  odd_S_2n.

Goal odd 37.
change (odd (1 + (2 * 18))).
auto.
Qed.
```

See also : pattern.

Using the conversion rule (2)

Lets us admit the following lemma :

```
Lemma Even_ind : ∀ (P:nat→Prop),
   P 0 →
  (∀ p:nat, P p → P (S (S p))) →
 ∀ n:nat, Even n → P n.

Lemma Even_plus_Even :
  ∀ i, Even i → ∀ n, Even n → Even (n + i).
intros i Hi.
```

i : nat

Hi : Even i

==========

∀ n : nat, Even n → Even (n + i)

We want now to apply Even_ind.

The following terms are convertible :

```
∀ n : nat, Even n → Even (n + i)
```

```
∀ n : nat, Even n → (fun n:nat => Even (n + i)) n
```

Then we can apply Even_ind
(with the substitution $P := fun\ n : nat \Rightarrow Even(n + i)$).

```
apply Even_ind.
```

```
 i : nat
 Hi : Even i
 ============================
  Even (0 + i)      (* P 0 *)
```

```
Subgoal 2 is :
 (* ∀  p : nat, P p  → P (S (S p) *)
∀  p : nat, Even (p + i) → Even (S (S p) + i)
```

In some situations, one must use pattern to decompose a proposition into some application $P\ t \ldots$.

Problems with equality

```
Lemma concat_assocative : ∀ (n p q: nat)
                            (u : binary_word n)
                            (v : binary_word p)
                            (w : binary_word q),
       binary_word_concat  (binary_word_concat u v) w =
      binary_word_concat  u (binary_word_concat  v w).
```

The term "binary_word_concat u (binary_word_concat v w)" has type "binary_word (n + (p + q))" while it is expected to have type "binary_word (n + p + q)".

```
Require Import JMeq.
```

JMeq : ∀ A : Type, A → ∀ B : Type, B → Prop
JMeq_refl: ∀ (A : Type) (x : A), JMeq x x

```
Lemma concat_associative : ∀ (n p q: nat)
                             (u : binary_word n)
                             (v : binary_word p)
                             (w : binary_word q),
  JMeq (binary_word_concat (binary_word_concat u v) w)
       (binary_word_concat u (binary_word_concat v w)).
```