Advanced Features: Type Classes and Relations (* to *****)

Pierre Castéran

Suzhou, Paris, 2011, Shanghai, 2012

In this lecture, we present shortly two quite new and useful features of the *Coq* system :

- Type classes are a nice way to formalize (mathematical) structures,
- User defined relations, and rewriting non-Leibniz "equalities" (*i.e.* for instance, equivalences).

In this lecture, we present shortly two quite new and useful features of the *Coq* system :

- Type classes are a nice way to formalize (mathematical) structures,
- User defined relations, and rewriting non-Leibniz "equalities" (*i.e.* for instance, equivalences).

- More details are given in *Coq*'s reference manual,
- A tutorial is available : `www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/`
- We hope you will replay the proofs, enjoy, and try to use these features.

In this lecture, we present shortly two quite new and useful features of the *Coq* system :

- Type classes are a nice way to formalize (mathematical) structures,
- User defined relations, and rewriting non-Leibniz "equalities" (*i.e.* for instance, equivalences).

- More details are given in *Coq*'s reference manual,
- A tutorial is available : `www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/`
- We hope you will replay the proofs, enjoy, and try to use these features.

Demo files :

All the examples are available at the tutorial's page.

Type Classes

A simple example : computing a^n

The following definition is very naïve, but obviously correct.

```
Fixpoint power (a:Z)(n:nat) :=
  match n with 0%nat => 1
             | S p =>  a * power a p
  end.
```

```
Compute power 2 40.
```

$= 1099511627776$

$: Z$

A simple example : computing a^n

The following definition is very naïve, but obviously correct.

```
Fixpoint power (a:Z)(n:nat) :=
  match n with 0%nat => 1
             | S p =>  a * power a p
  end.
```

```
Compute power 2 40.
```

$= 1099511627776$

$: Z$

Thus, the function power can be considered as a specification for more efficient algorithms.

The binary exponentiation algorithm

Let's define an auxiliary function ...

```
Function binary_power_mult  (acc x:Z)(n:nat)
                 {measure (fun i=>i) n} : Z
  (* acc * (power x n) *) :=
 match n with 0%nat => acc
              | _ => if Even.even_odd_dec n
                     then  binary_power_mult
                           acc (x * x) (div2 n)
                     else  binary_power_mult
                          (acc * x) (x * x) (div2 n)
  end.
intros;apply lt_div2; auto with arith.
intros;apply lt_div2; auto with arith.
Defined.
```

...and the main function.

```
Definition binary_power (x:Z)(n:nat) :=
        binary_power_mult 1 x n.

Compute binary_power 2 40.
```

1099511627776: Z

. . .and the main function.

```
Definition binary_power (x:Z)(n:nat) :=
        binary_power_mult 1 x n.

Compute binary_power 2 40.
```

1099511627776: Z

- Is binary_power correct (w.r.t. power) ?

...and the main function.

```
Definition binary_power (x:Z)(n:nat) :=
        binary_power_mult 1 x n.

Compute binary_power 2 40.
```

1099511627776: Z

- Is `binary_power` correct (w.r.t. `power`)?
- Is it worth proving this correctness only for powers of integers?

...and the main function.

```
Definition binary_power (x:Z)(n:nat) :=
        binary_power_mult 1 x n.

Compute binary_power 2 40.
```

1099511627776: Z

- Is binary_power correct (w.r.t. power)?
- Is it worth proving this correctness only for powers of integers?
- And prove it again for powers of real numbers, matrices?

...and the main function.

```
Definition binary_power (x:Z)(n:nat) :=
       binary_power_mult 1 x n.

Compute binary_power 2 40.
```

1099511627776: Z

- Is `binary_power` correct (w.r.t. `power`) ?
- Is it worth proving this correctness only for powers of integers ?
- And prove it again for powers of real numbers, matrices ? NO !

Monoids

We aim to prove the equivalence between `power` and `binary_power` for any structure consisting of a binary associative operation that admits a neutral element

Monoids

We aim to prove the equivalence between `power` and `binary_power` for any structure consisting of a binary associative operation that admits a neutral element

Definition

A *monoid* is a mathematical structure composed of :

- A carrier A
- A binary, associative operation $\circ$ on A
- A neutral element $1 \in A$ for $\circ$

```
Class Monoid {A:Type}(dot : A -> A -> A)(unit : A)
: Type := {
 dot_assoc : forall x y z:A,
     dot x (dot y z)= dot (dot x y) z;
 unit_left : forall x, dot unit x = x;
 unit_right : forall x, dot x unit = x }.
```

In fact such a class is stored as a record, parameterized with A, dot and unit. Just try Print monoid.

An alternative ?

```
Class Monoid'  : Type := {
  carrier: Type;
  dot : carrier -> carrier -> carrier;
  one : carrier;
  dot_assoc : forall x y z:carrier, dot x (dot y z)=
                                    dot (dot x y) z;
  one_left : forall x, dot one x = x;
  one_right : forall x, dot x one = x}.
```

An alternative ?

```
Class Monoid'  : Type := {
  carrier: Type;
  dot : carrier -> carrier -> carrier;
  one : carrier;
  dot_assoc : forall x y z:carrier, dot x (dot y z)=
                                    dot (dot x y) z;
  one_left : forall x, dot one x = x;
  one_right : forall x, dot x one = x}.
```

No !

- Bas Spitters and Eelis van der Weegen,
  Type classes for mathematics in type theory,
  CoRR, abs/1102.1323, 2011.

In short, it would be clumsy to express "two monoids on the same carrier".

Defining power *in any monoid*

```
Generalizable Variables A dot one.

Fixpoint power `{M :Monoid A dot one}(a:A)(n:nat) :=
  match n with 0%nat => one
             | S p => dot a (power a p)
 end.

Lemma power_of_unit `{M :Monoid A dot one} :
  forall n:nat, power one n = one.
Proof.
 induction n as [| p Hp];simpl;
     [|rewrite Hp;simpl;rewrite unit_left];trivial.
Qed.
```

Building an instance of the class Monoid

```
Require Import ZArith.
Open Scope Z_scope.

Instance ZMult : Monoid  Zmult 1.
split.
```

3 subgoals

============================

*forall x y z : Z, x * (y * z) = x * y * z*

subgoal 2 is:

*forall x : Z, 1 * x = x*

subgoal 3 is:

*forall x : Z, x * 1 = x*

```
Qed.
```

Each subgoal has been solved by intros ;ring.

Instance Resolution

```
About power.
```

power :
forall (A : Type) (dot : A -> A -> A) (one : A),
Monoid dot one -> A -> nat -> A

Arguments A, dot, one, M are implicit and maximally inserted

Instance Resolution

```
About power.
```

power :
forall (A : Type) (dot : A -> A -> A) (one : A),
Monoid dot one -> A -> nat -> A

Arguments A, dot, one, M are implicit and maximally inserted

```
Compute power 2 100.
```

= 1267650600228229401496703205376 : Z

Instance Resolution

```
About power.
```

power :
forall (A : Type) (dot : A -> A -> A) (one : A),
Monoid dot one -> A -> nat -> A

Arguments A, dot, one, M are implicit and maximally inserted

```
Compute power 2 100.
```

= 1267650600228229401496703205376 : Z

```
Set Printing Implicit.
Check power 2 100.
```

@power Z Zmult 1 ZMult 2 100 : Z

```
Unset Printing Implicit.
```

The *instance* `ZMult` is inferred from the type of 2.

2×2 Matrices on any Ring

```
Require Import Ring.

Section matrices.
 Variables (A:Type)
           (zero one : A)
           (plus mult minus : A -> A -> A)
           (sym : A -> A).
 Notation "0" := zero.
 Notation "1" := one.
 Notation "x + y" := (plus x y).
 Notation "x * y " := (mult x y).

 Variable rt :
  ring_theory  zero one plus mult minus sym (@eq A).
 Add  Ring Aring : rt.
```

```
Structure M2 : Type := {c00 : A;  c01 : A;
                        c10 : A;  c11 : A}.

Definition Id2 : M2 := Build_M2  1 0 0 1.

Definition M2_mult (m m':M2) : M2 :=
 Build_M2 (c00 m * c00 m' + c01 m * c10 m')
          (c00 m * c01 m' + c01 m * c11 m')
          (c10 m * c00 m' + c11 m * c10 m')
          (c10 m * c01 m' + c11 m * c11 m').

Global Instance M2_Monoid : Monoid  M2_mult Id2.
...
Defined.

End matrices.
```

```
Compute power (Build_M2 1 1 1 0) 40.
```

$= \{|$

c00 := 165580141;

c01 := 102334155;

c10 := 102334155;

c11 := 63245986 $|\}$

: M2 Z

```
Compute power (Build_M2 1 1 1 0) 40.
```

$= \{|$

$c00 := 165580141;$

$c01 := 102334155;$

$c10 := 102334155;$

$c11 := 63245986\ |\}$

$: M2\ Z$

```
Definition fibonacci (n:nat) :=
  c00 (power  (Build_M2 1 1 1 0) n).

Compute  fibonacci 20.
```

$= 10946$

$: Z$

A generic proof of correctness of `binary_power`

We are now able to prove the equivalence of `power` and `binary_power` *in any monoid*.

A generic proof of correctness of `binary_power`

We are now able to prove the equivalence of `power` and `binary_power` *in any monoid*.

Note

We give only the structure of the proof. The complete development will be distributed (for `coq8.3pl2`)

Let us consider an arbitrary monoid

```
Section About_power.

Require Import Arith.
 Context `(M:Monoid A dot one ).
```

Let us consider an arbitrary monoid

```
Section About_power.

Require Import Arith.
 Context `(M:Monoid A dot one ).

Ltac monoid_rw :=
    rewrite (@one_left A dot one M) ||
    rewrite (@one_right A dot one M)||
    rewrite (@dot_assoc A dot one M).

  Ltac monoid_simpl := repeat monoid_rw.
  Local Infix "*" := dot.
  Local Infix "**" := power (at level 30, no associativity).
```

Within this context, we prove some useful lemmas

```
Lemma power_x_plus : forall x n p,
 x ** (n + p) =  x ** n *  x ** p.
Proof.
 induction n;simpl.
 intros;  monoid_simpl;trivial.
 intro p;rewrite (IHn p).  monoid_simpl;trivial.
Qed.
```

Within this context, we prove some useful lemmas

```
Lemma power_x_plus : forall x n p,
 x ** (n + p) =  x ** n *  x ** p.
Proof.
 induction n;simpl.
 intros;  monoid_simpl;trivial.
 intro p;rewrite (IHn p).  monoid_simpl;trivial.
Qed.

Lemma power_of_power : forall x n p,
     (x ** n) ** p = x ** (p * n).
 Proof.
   induction p;simpl;
   [| rewrite power_x_plus; rewrite IHp]; trivial.
Qed.
```

```
Lemma binary_power_mult_ok :
  forall n a x,  binary_power_mult  M  a x n = a * x ** n.

...

Lemma binary_power_ok : forall x n,
     binary_power (x:A)(n:nat) = x ** n.
Proof.
  intros n x;unfold binary_power;
   rewrite binary_power_mult_ok;
   monoid_simpl;auto.
Qed.

End About_power.
```

Subclasses

```
Class Abelian_Monoid `(M:Monoid ):= {
  dot_comm : forall x y, (dot x  y = dot y  x)}.

Instance ZMult_Abelian : Abelian_Monoid ZMult.
split.
 exact Zmult_comm.
Defined.
```

```
Section Power_of_dot.
 Context `{M: Monoid A} {AM:Abelian_Monoid M}.

Theorem power_of_mult :  forall n x y,
  power (dot x y)  n =  dot (power x  n) (power y n).
Proof.
 induction n;simpl.
 rewrite one_left;auto.
 intros; rewrite IHn; repeat rewrite dot_assoc.
 rewrite <- (dot_assoc  x y (power x n));
 rewrite (dot_comm y (power x n)).
 repeat rewrite dot_assoc;trivial.
Qed.
```

More about class types

- Download Coq's latest development version,
- Download the tutorial on type classes (written for Coq 8.4 , but sources are also available for Coq8.3)
- Bas Spitters, Eelis van der Weegen : Type Classes for Mathematics in Type Theory

More about class types

- Download Coq's latest development version,
- Download the tutorial on type classes (written for Coq 8.4 , but sources are also available for Coq8.3)
- Bas Spitters, Eelis van der Weegen : Type Classes for Mathematics in Type Theory

It is possible to define and export notations for operations on type classes. See `Monoid_op_classes.v`

```
power_of_mult :
forall (A : Type) (dot : monoid_binop A) (one : A)
       (M : Monoid dot one),
Abelian_Monoid M ->
forall (n : nat) (x y : A),
  (x * y)%M ** n = (x ** n * y ** n)%M
```

Introduction to Setoids

Let us recall how rewrite works.

- This tactic uses eq_rect,

```
eq_rect
     : forall (A : Type) (x : A) (P : A -> Type),
       P x -> forall y : A, x = y -> P y
```

Introduction to Setoids

Let us recall how rewrite works.

- This tactic uses eq_rect,

```
eq_rect
     : forall (A : Type) (x : A) (P : A -> Type),
       P x -> forall y : A, x = y -> P y
```

`eq_rect` allows us to replace y with x in *any context P in the goal*.

Introduction to Setoids

Let us recall how rewrite works.

- This tactic uses eq_rect,

```
eq_rect
     : forall (A : Type) (x : A) (P : A -> Type),
       P x -> forall y : A, x = y -> P y
```

eq_rect allows us to replace y with x in *any context P in the goal*.

Some equivalence relations are weaker than equality, which means :

- They are easier to prove (Leibniz equality is quite strong)

Introduction to Setoids

Let us recall how rewrite works.

- This tactic uses eq_rect,

```
eq_rect
     : forall (A : Type) (x : A) (P : A -> Type),
       P x -> forall y : A, x = y -> P y
```

eq_rect allows us to replace y with x in *any context P in the goal*.

Some equivalence relations are weaker than equality, which means :

- They are easier to prove (Leibniz equality is quite strong)
- They may be harder to use by rewrite : we cannot replace a term by an equivalent one in *any context* . replacing

We would like to use rewrite with relations *(easier to prove)* than $x = y$.

Assume you are lost in Manhattan, or in any city with the same geometry : square blocks, square blocks, and so on.

Assume you are lost in Manhattan, or in any city with the same geometry : square blocks, square blocks, and so on.

You ask some by-passer how to go to some other place, and you probably will get an answer like that :

> *"go two blocks northward, then one block eastward, then three blocks southward, and finally two blocks westward".*

Assume you are lost in Manhattan, or in any city with the same geometry : square blocks, square blocks, and so on.

You ask some by-passer how to go to some other place, and you probably will get an answer like that :

> *"go two blocks northward, then one block eastward, then three blocks southward, and finally two blocks westward".*

You thank this kind person, and you go one block southward, then one block westward.

Data for moving in the discrete plane

```
Record Point : Type :=
 {Point_x : Z;
  Point_y:Z}.

Definition Point_O := Build_Point 0 0.

Definition translate (dx dy:Z) (P : Point) :=
  Build_Point (Point_x P + dx) (Point_y P + dy).




Inductive direction : Type := North | East | South | West.
Definition route := list direction.
```

Route Equivalence

```
Fixpoint move (r:route) (P:Point) : Point :=
 match r with
  | nil => P
  | North :: r' => move r' (translate 0 1 P)
  | East :: r' => move r' (translate 1 0 P)
  | South :: r' => move r' (translate 0 (-1) P)
  | West :: r' => move r' (translate (-1) 0 P)
 end.

Definition route_equiv : relation route :=
  fun r r' => forall P:Point , move r P = move  r' P.

Infix "=r=" := route_equiv (at level 70):type_scope.
```

Route Equivalence is not Leibniz equality

```
Example Ex1 :
    East::North::West::South::East::nil =r= East::nil.
Proof.
intro P;destruct P;simpl. unfold route_equiv, trans-
late;simpl;f_equal; ring.
Qed.

Example Ex1' :
  East::North::West::South::East::nil <> East::nil.
Proof. discriminate. Qed.

Example Ex1'' :
 length (East::nil) <
 length (East::North::West::South::East::nil).
...
```

Rewriting : A Failed Attempt

```
 Lemma Fail1 : forall r,
     r =r= r++(East::North::West::South::nil).
Proof.
 induction r; simpl.
 ...



 1 subgoal

a : direction
r : list direction
IHr : r =r= r ++ East :: North :: West :: South :: nil
============================
a :: r =r= a :: r ++ East :: North :: West :: South :: nil

 rewrite IHr.
   error ..;
```

Equivalence Relations

```
Instance route_equiv_refl : Reflexive route_equiv.
Proof. intros r p;reflexivity. Qed.
```

```
Instance route_equiv_sym : Symmetric route_equiv.
Proof. intros r r' H p; symmetry;apply H. Qed.
```

```
Instance route_equiv_trans : Transitive route_equiv.
Proof. intros r r' r'' H H' p; rewrite H; apply H'. Qed.
```

```
Instance route_equiv_Equiv : Equivalence route_equiv.
```

Equivalence Relations

```
 Instance  route_equiv_refl : Reflexive  route_equiv.
Proof. intros r p;reflexivity. Qed.
```

```
Instance route_equiv_sym : Symmetric  route_equiv.
Proof.  intros r r' H p; symmetry;apply H. Qed.
```

```
Instance  route_equiv_trans : Transitive route_equiv.
Proof.  intros r r' r'' H H' p; rewrite H; apply H'. Qed.
```

```
Instance route_equiv_Equiv : Equivalence  route_equiv.
```

We are now able to use the tactics reflexivity, symmetry, and transitivity with the relation =r=.

```
Example E2 : North::South::nil =r= West::East::nil.
Proof.
 transitivity (@nil direction).

2 subgoals

  ============================
   North :: South :: nil =r= nil

subgoal 2 is:
 nil =r= West :: East :: nil
```

```
Example E3 :
  West::North::South::nil =r= West::West::East::nil.
Proof.
 rewrite E2.
 Error message !
```

```
Example E3 :
  West::North::South::nil =r= West::West::East::nil.
Proof.
 rewrite E2.
 Error message !
```

We have to prove and tell to *Coq* that if r=r=r' then $d::r$=r=$d::r'$.
We say that `cons` `is` *`Proper`* w.r.t. `=r=`

```
Lemma route_cons :
forall r r' d, r =r= r' -> d::r =r= d::r'.
Proof.
 intros r r' d H P;destruct d;simpl;rewrite H;reflexivity.
Qed.

Instance cons_route_Proper (d:direction):
    Proper (route_equiv ==> route_equiv) (cons d) .
Proof.
 intros r r' H ;apply route_cons;assumption.
Qed.
```

cons_route_Proper allows us to replace a route with an =r= equivalent one in a context composed by "cons" :

```
Goal forall r r', r =r= r' ->
                  South::West::r =r= South::West::r'.
intros r r' H; rewrite H;reflexivity.
Qed.
```

Note that all functions are not proper : for instance, there exist two routes that are equivalent, but not of the same length.
Thus length is not proper w.r.t. =r= and =.

```
Example length_not_Proper :
   ~Proper (route_equiv ==> @eq nat) (@length _).
Proof.
 intro H; generalize (H (North::South::nil) nil);
 simpl;intro H0.
 discriminate H0.
 intro P;destruct P; simpl;unfold translate;
  simpl;f_equal;simpl;ring.
Qed.
```

We want now to use rewrite H on the route_equiv relation in contexts built with the app function.

```
Lemma route_compose :
  forall r r' P, move (r++r') P = move r' (move r P).
Proof.
 induction r as [|d s IHs]; simpl;
   [auto | destruct d; intros;rewrite IHs;auto].
```

We want now to use rewrite H on the route_equiv relation in contexts built with the app function.

```
Lemma route_compose :
  forall r r' P, move (r++r') P = move r' (move r P).
Proof.
 induction r as [|d s IHs]; simpl;
   [auto | destruct d; intros;rewrite IHs;auto].

Instance app_route_Proper :
  Proper (route_equiv==>route_equiv ==> route_equiv)
 (@app direction).
 intros r r' H r'' r''' H' P.
Proof.
  repeat rewrite route_compose; rewrite H, H';reflexivity.
Qed.
```

```
Example Ex3 : forall r, North::East::South::West::r =r= r.
Proof. intros r P;destruct P;simpl.
  unfold route_equiv, translate;simpl;do 2 f_equal;ring.
Qed.

Example Ex4 : forall r r', r =r= r' ->
                North::East::South::West::r =r= r'.
Proof. intros r r' H. now rewrite Ex3. Qed.
```

Setoids and Monoids

```
Set Implicit Arguments.
Require Import Morphisms Relations.

 Class EMonoid (A:Type)(E_eq :relation  A)
               (dot : A->A->A)(one : A):={
  E_rel :> Equivalence E_eq;
  dot_proper :> Proper (E_eq ==> E_eq ==> E_eq) dot;
  E_dot_assoc : forall x y z:A, E_eq (dot x (dot y z))
                                     (dot (dot x y) z);
  E_one_left : forall x, E_eq (dot one  x) x;
  E_one_right : forall x, E_eq (dot x  one) x}.
```

Extract from Demo file Lost_in_Ny.v

```
Instance Route : EMonoid   route_equiv (@app _) nil.
Proof
split.
 apply route_equiv_Equiv.
 apply app_route_Proper.
 intros x y z P;repeat rewrite  route_compose; trivial.
  intros x  P;repeat rewrite  route_compose; trivial.
 intros x  P;repeat rewrite  route_compose; trivial.
```

Some other results in the demo file

```
Definition opposite (d d':direction):= match d,d' with
        | North,South => True
        | South, North => True
        | East, West => True
        | West, East => True
        | _, _ => False
end.

Inductive  Useless_steps_in (r:route) : Type :=
Useless_i: forall r0 r1 r2 d d1,
   r = r0++(d::r1)++(d1::r2) ->
   opposite d d1 ->
   Useless_steps_in r.
```

```
Lemma Useless_steps_shorter (r:route) :
  Useless_steps_in r ->
   {r' : route | r =r= r' /\
                 (length r' < length r)%nat}.
Proof.
```

We provide also a boolean function `route_eqb` for deciding route equivalence. Thus proofs of equivalence can be very short and automatic :

We provide also a boolean function `route_eqb` for deciding route equivalence. Thus proofs of equivalence can be very short and automatic :

```
Ltac route_eq_tac := rewrite route_equiv_equivb;reflexivity.

(** another proof of Ex1, using computation  *)

Example Ex1' : East::North::West::South::East::nil =r= East::n
Proof. route_eq_tac. Qed.
```