

# Advanced Features: Type Classes and Relations

Pierre Castéran

Suzhou, Paris, 2011

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 will be available soon.
- 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 will be available soon.
- We hope you will replay the proofs, enjoy, and try to use these features.

### Demo files :

`Power_Mono.v`, `Monoid.v`, `EMonoid.v`, `Trace_Monoid.v`.

The file `Monoid_op_classes.v` is given for advanced experiments only.

## 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*

=====

$$\text{forall } x \ y \ z : Z, x * (y * z) = x * y * z$$

*subgoal 2 is:*

$$\text{forall } x : Z, 1 * x = x$$

*subgoal 3 is:*

$$\text{forall } x : Z, x * 1 = x$$

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.

## $2 \times 2$ 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 `coq8.3p12` )

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,
- Read Papers by Matthieu Sozeau on the implementation
- Bas Spitters, Eelis van der Weegen : Type Classes for Mathematics in Type Theory

## More about class types

- Download Coq's latest development version,
- Read Papers by Matthieu Sozeau on the implementation
- 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`,
- without other hypotheses, the proposition  $x = y$  can only be proven through `eq_refl`

```
eq_rect
```

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

```
Inductive eq (A : Type) (x : A) : A -> Prop :=  
| eq_refl : x = x
```

# Introduction to Setoids

Let us recall how `rewrite` works.

- This tactic uses `eq_rect`,
- without other hypotheses, the proposition  $x = y$  can only be proven through `eq_refl`

```
eq_rect
```

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

```
Inductive eq (A : Type) (x : A) : A -> Prop :=  
| eq_refl : x = x
```

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

## An example : Trace Monoids

The following instruction sequences are equivalent but not equal.

```
x = y+1;
y = z * z;
for(int i=0; i<n; i++)
    x +=z
if (y >= 0) then
    System.out.println("y=" + y + " x =" + x);
```

```
x = y+1;
for(int i=0; i<n; i++)
    x +=z;
y = z * z;
System.out.println("y=" + y + " x =" + x);
```



- *Trace monoids* a.k.a. *free partially commutative monoids* are models of concurrent programming.
- They describe which actions are independent, *i.e.* can *commute*.
- For instance,  $x+=z$  can commute with  $y=z*z$ , but not with  $x=y+1$

- *Trace monoids* a.k.a. *free partially commutative monoids* are models of concurrent programming.
- They describe which actions are independent, *i.e.* can *commute*.
- For instance,  $x+=z$  can commute with  $y= z*z$ , but not with  $x= y+1$

In order to simplify our development, we consider three basic actions :  $a$ ,  $b$  and  $c$ , and represents programmes as lists of actions.

The lists  $a::b::a::c::b::a::nil$  and  $a::a::b::c::a::b::nil$  should be equivalent, but not equal !

```
Require Import List
      Relation_Operators
      Operators_Properties.
```

```
Section Partial_Com.
```

```
Inductive Act : Set := a | b | c.
```

```
Example Diff : a::b::nil <> b::a::nil.
discriminate.
Qed.
```

Let us define the relation partial commutation, generated by **a** and **b**

(\* One transposition of **a** and **b** \*)

```
Inductive transpose : list Act -> list Act -> Prop :=  
  transpose_hd : forall w, transpose(a::b::w) (b::a::w)  
|transpose_tl : forall x w u, transpose w u ->  
  transpose (x::w) (x::u).
```

We can now consider the reflexive, symmetric and transitive closure of transpose :

```
Definition commute := clos_refl_sym_trans _ transpose.
```

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

We now declare `commute` as an instance of the `Equivalence` type class :

```
Instance Commute_E : Equivalence  commute.  
split;[constructor 2|constructor 3|econstructor 4];eauto.  
Qed.
```

We are now able to use the tactics **reflexivity**, **symmetry**, and **transitivity** on goals of the form  $x == y$ .

Example ex0 : b::a::nil == a::b::nil.

symmetry.

repeat constructor.

Qed.

Example ex0 :  $b::a::\text{nil} == a::b::\text{nil}$ .

*symmetry.*

repeat constructor.

Qed.

Example ex1 :  $a::b::b::\text{nil} == b::b::a::\text{nil}$ .

*transitivity (b::a::b::nil).*

repeat constructor.

repeat constructor.

Qed.

Goal forall w, w++(a::b::nil) == w++(b::a::nil).

Proof.

induction w;simpl.

constructor. constructor.

*a0 : Act*

*w : list Act*

*IHw : w ++ a :: b :: nil == w ++ b :: a :: nil*

=====

*a0 :: w ++ a :: b :: nil == a0 :: w ++ b :: a :: nil*

rewrite IHw.

*Error message*

Abort.



We need to prove and register that if  $u == v$  then  $x::u == x::v$ .

We need to prove and register that if  $u == v$  then  $x::u == x::v$ .

Require Import Setoid Morphisms.

```
Instance cons_commute_Proper (x:Act):  
  Proper (commute ==> commute) (cons x).
```

```
intros l l' H.
```

*1 subgoal*

*x : Act*

*l : list Act*

*l' : list Act*

*H : l == l'*

=====

*x :: l == x :: l'*

...

*Qed.*

Note that the following statement is also correct :

Lemma `cons_transpose_Proper` :

```
Proper (eq ==> transpose ==> transpose)
      (@cons Act).
```

We are now able to use `rewrite` in contexts formed by the `cons` operator.

```
Goal forall u v, u == v -> (a::b::u) == (b::a::v).
```

```
Proof.
```

```
  intros u v H;rewrite H.
```

```
  constructor;constructor.
```

```
Qed.
```

We can now consider again our failed attempt.

Goal forall w, w++(a::b::nil) == w++(b::a::nil).

Proof.

```
induction w;simpl.
```

```
constructor. constructor.
```

We can now consider again our failed attempt.

Goal forall w, w++(a::b::nil) == w++(b::a::nil).

Proof.

induction w;simpl.

constructor. constructor.

*1 subgoal*

*a0 : Act*

*w : list Act*

*!Hw : w ++ a :: b :: nil == w ++ b :: a :: nil*

=====

*a0 :: w ++ a :: b :: nil == a0 :: w ++ b :: a :: nil*

We can now consider again our failed attempt.

Goal forall w, w++(a::b::nil) == w++(b::a::nil).

Proof.

induction w;simpl.

constructor. constructor.

*1 subgoal*

*a0 : Act*

*w : list Act*

*IHw : w ++ a :: b :: nil == w ++ b :: a :: nil*

=====

*a0 :: w ++ a :: b :: nil == a0 :: w ++ b :: a :: nil*

rewrite IHw; reflexivity.

Qed.

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

```
Instance append_commute_Proper_1 :
  Proper (Logic.eq ==> commute ==> commute) (@app Act).
...
(* usage :

   H : v == w
   -----
   u ++ v == u ++ w

[setoid_]rewrite H.
*)
Qed.
```



```

Instance append_commute_Proper_2 :
Proper (commute ==> Logic.eq ==> commute) (@app Act).
(* usage :

  H : u == v
  -----
  u ++ w == v ++ w

[setoid_]rewrite H.
*)
Qed.

```

```
Instance append_Proper :  
Proper (commute ==> commute ==> commute) (@app Act).  
Proof.  
intros x y H z t H0;transitivity (y++z).  
rewrite H;reflexivity.  
rewrite H0;reflexivity.  
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 Trace\_Monoid.v

```
Instance PCom : EMonoid commute (@List.app Act) nil.  
Proof  
split.  
apply Commute_E.  
apply append_Proper.  
intros;rewrite <- app_assoc;reflexivity.  
simpl;reflexivity.  
intros;rewrite app_nil_r;reflexivity.  
Qed.
```

## Conclusion : Learning *Coq*

- It is important to look again at the examples and exercises, as well as the *Coq* documentation.
- With Yves, we propose 200 (solved) exercises, at [www.labri.fr/perso/casteran/CoqArt/contents.html](http://www.labri.fr/perso/casteran/CoqArt/contents.html)

## Conclusion : Learning *Coq*

- It is important to look again at the examples and exercises, as well as the *Coq* documentation.
- With Yves, we propose 200 (solved) exercises, at [www.labri.fr/perso/casteran/CoqArt/contents.html](http://www.labri.fr/perso/casteran/CoqArt/contents.html)
- Suscribe to the `coq-club` mailing list !

## Conclusion : Learning *Coq*

- It is important to look again at the examples and exercises, as well as the *Coq* documentation.
- With Yves, we propose 200 (solved) exercises, at [www.labri.fr/perso/casteran/CoqArt/contents.html](http://www.labri.fr/perso/casteran/CoqArt/contents.html)
- Suscribe to the `coq-club` mailing list!
  - ▶ Don't hesitate to ask questions!

## Conclusion : Learning *Coq*

- It is important to look again at the examples and exercises, as well as the *Coq* documentation.
- With Yves, we propose 200 (solved) exercises, at [www.labri.fr/perso/casteran/CoqArt/contents.html](http://www.labri.fr/perso/casteran/CoqArt/contents.html)
- Suscribe to the `coq-club` mailing list !
  - ▶ Don't hesitate to ask questions !
  - ▶ Look at the questions by other people, and at the answers.



## Conclusion : Learning *Coq*

- It is important to look again at the examples and exercises, as well as the *Coq* documentation.
- With Yves, we propose 200 (solved) exercises, at [www.labri.fr/perso/casteran/CoqArt/contents.html](http://www.labri.fr/perso/casteran/CoqArt/contents.html)
- Suscribe to the `coq-club` mailing list !
  - ▶ Don't hesitate to ask questions !
  - ▶ Look at the questions by other people, and at the answers.
  - ▶ Be the first to answer !

## Conclusion : Learning *Coq*

- It is important to look again at the examples and exercises, as well as the *Coq* documentation.
- With Yves, we propose 200 (solved) exercises, at [www.labri.fr/perso/casteran/CoqArt/contents.html](http://www.labri.fr/perso/casteran/CoqArt/contents.html)
- Suscribe to the `coq-club` mailing list !
  - ▶ Don't hesitate to ask questions !
  - ▶ Look at the questions by other people, and at the answers.
  - ▶ Be the first to answer ! It's easy ! A question asked in Beijing at 3 p.m. can be answered in Europe at 9 a.m.

- Look at the **user contributions** page in `coq.inria.fr`. You will find a lot of examples and tools on many domains : math, computer science, games, etc.
- Submitting your Coq development as a contribution provides visibility to your work and the assurance to have it maintained and made compatible with the forthcoming versions of Coq. For the Coq developers, it helps to evaluate the robustness and efficiency of the evolutions of Coq.

- Look at the **user contributions** page in `coq.inria.fr`. You will find a lot of examples and tools on many domains : math, computer science, games, etc.
- Submitting your Coq development as a contribution provides visibility to your work and the assurance to have it maintained and made compatible with the forthcoming versions of Coq. For the Coq developers, it helps to evaluate the robustness and efficiency of the evolutions of Coq.
- Don't forget that using *Coq* is like a game : You want to be able to type `Qed` before 6 p.m., and the system wants your proof to be complete and correct.

- Look at the **user contributions** page in `coq.inria.fr`. You will find a lot of examples and tools on many domains : math, computer science, games, etc.
  - Submitting your Coq development as a contribution provides visibility to your work and the assurance to have it maintained and made compatible with the forthcoming versions of Coq. For the Coq developers, it helps to evaluate the robustness and efficiency of the evolutions of Coq.
- 
- Don't forget that using *Coq* is like a game : You want to be able to type `Qed` before 6 p.m., and the system wants your proof to be complete and correct.
  - Quite often, the system helps you. It's a proof **assistant**.