A Gentle Introduction to Type Classes and Relations in Coq

Pierre Castéran
Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France.
CNRS, LaBRI, UMR 5800, F-33400 Talence, France.

Matthieu Sozeau
INRIA Paris & PPS, Univ. Paris Diderot

November 3, 2016
Chapter 1

Introduction

This document aims to give a small introduction to two important and rather new features of the Coq proof assistant [Coq Development Team 2012, Bertot and Casteran 2004]: type classes and user defined relations. We will mainly use examples for introducing these concepts, and the reader should consult Coq’s reference manual for further information.

Type classes are presented in Chapter 18 “Type Classes” of Coq’s reference manual, and user defined relations in Chapter 25.

The complete definitions and proof scripts are structured as follows:

**Power_mono.v** Sadly monomorphic definitions of function $x \in \mathbb{Z}$, $n \in \mathbb{N} \mapsto x^n$ (naïve and binary algorithm).

**Matrices.v** A data type for $2 \times 2$ matrices.

**Monoid.v** The Monoid type class; proof of equivalence of the naïve and the binary power functions in any monoïd. The abelian (a.k.a. commutative) monoid structure.

**Monoid_op_classes.v** Same contents as Monoid.v, but with operational type classes.

**Alternate_defs.v** Discussion about various possible representations of the monoid mathematical structure.

**Lost_in_NY.v** An introduction to user defined relations, and rewriting.

**EMonoid.v** Monoids with respect to an equivalence relation.

**Trace_Monoid.v** The trace monoid (a.k.a. partial commutation monoid).

A tar file of these scripts (for Coq V8.4) is available at the following address: [http://www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/src-V8.4.tar.gz](http://www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/src-V8.4.tar.gz)
1.1 A little history

Type classes were first introduced in Haskell by Wadler and Blott [1989] and the Isabelle proof assistant [Nipkow and Snelting 1991] at the beginning of the 90’s, to handle ad-hoc polymorphism elegantly. Sozeau and Oury [2008] adapted the design to Coq, making use of the powerful type system of Coq to integrate a lightweight version of type classes which we’ll introduce here.

The purpose of type class systems is to allow a form of generic programming for a class of types. Classes are presented as an interface comprised of functions specific to a type, and in the case of Isabelle and Coq, proofs about those functions as well. One can write programs that are polymorphic over any type which has an instance of the class, and later on, instantiate those to a specific type and implementation (called instance) of the class. Polymorphism is ad-hoc as opposed to the parametric polymorphism found in e.g. ML: while parametrically polymorphic functions behave uniformly for any instantiation of the type, ad-hoc polymorphic functions have type-specific behaviors.
Chapter 2

An Introductory Example: Computing $x^n$

2.1 A Monomorphic Introduction

Let us consider a simple arithmetic operation: raising some integer $x$ to the $n$-th power, where $n$ is a natural number.

The following function definition is a direct translation of the mathematical concept:

```
Require Import ZArith.
Open Scope Z_scope.

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

```
Compute power 2 40.
= 1099511627776 : Z
```

This definition can be considered as a very naïve way of programming, since computing $x^n$ requires $n$ multiplications. Nevertheless, this definition is very simple to read, and everyone can admit that it is correct with respect to the mathematical definition of raising $x$ to the $n$-th power. Thus, we can consider it as a specification: when we write more efficient but less readable functions for exponentiation, we should be able to prove their correctness by proving in Coq their equivalence\footnote{i.e. extensional equality} with the naïve `power` function.

For instance the following function allows one to compute $x^n$, with a number of multiplications proportional to $\log_2(n)$:
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.
Proof.
intros;apply lt_div2; auto with arith.
intros;apply lt_div2; auto with arith.
Defined.

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

Compute binary_power 2 40.
1099511627776: Z

We want now to prove binary_power’s correctness, i.e. that this function and the naïve power function are pointwise equivalent.

Proving this equivalence in Coq may require a lot of work. Thus it is not worth at all writing a proof dedicated only to powers of integers. In fact, the correctness of binary_power with respect to power holds in any structure composed of an associative binary operation on some domain, that admits a neutral element. For instance, we can compute powers of square matrices using the most efficient of both algorithms.

Thus, let us throw away our previous definition, and try do define them in a more generic framework.

Reset power.

2.2 On Monoids

Definition 2.1 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 \( \circ \)

Such a mathematical structure can be defined in Coq as a type class [Sozeau and Oury 2008] [Spitters and van der Weegen 2011]. In the following definition, parameterized by a type \( A \) (implicit), a binary operation \( \text{dot} \) and a
neutral element \text{unit}, three fields describe the properties that \text{dot} and \text{unit} must satisfy.

Class Monoid \{A:Type\}(\text{dot} : A -> A -> A)(\text{one} : A) : Prop := {
  \text{dot assoc} : forall x y z:A,
  \text{dot x (dot y z)} = \text{dot (dot x y) z};
  \text{unit left} : forall x, \text{dot one x} = x;
  \text{unit right} : forall x, \text{dot x one} = x }.

Note that other definitions could have been given for representing this mathematical structure. See Sect 3.9.1 for more details.

From an implementational point of view, such a type class is just a record type, \text{i.e.} an inductive type with a single constructor \text{Build_Monoid}.

Print Monoid.

Record Monoid (A : Type) (\text{dot} : A -> A -> A) (\text{one} : A) : Prop := \text{Build_Monoid}
\{ \text{dot assoc} : forall x y z : A, \text{dot x (dot y z)} = \text{dot (dot x y) z};
  \text{one left} : forall x : A, \text{dot one x} = x;
  \text{one right} : forall x : A, \text{dot x one} = x \}

For Monoid: Argument A is implicit and maximally inserted
For Build_Monoid: Argument A is implicit
For Monoid: Argument scopes are \{type_scope _ _\}
For Build_Monoid: Argument scopes are \{type_scope _ _ _ _ _\}

Nevertheless, the implementation of type classes by M. Sozeau provides several specific tools —dedicated tactics for instance —, and we advise the reader not to replace the Class keyword with Record or Inductive.

With the command About, we can see the polymorphic type of the fields of the class Monoid:

About one_left.
\text{one left}:
forall (A : Type) (\text{dot} : A -> A -> A) (\text{one} : A),
\text{Monoid dot one -> forall x : A, dot one x} = x

Arguments A, dot, one, Monoid are implicit and maximally inserted
Argument scopes are \{type_scope _ _ _ _ _\}
one_left is transparent

2.2.1 Classes and Instances

Members of a given class are called \text{instances} of this class. Instances are defined to the Coq system through the Instance keyword. Our first example is a definition of the monoid structure on the set \( Z \) of integers, provided with integer multiplication, with 1 as a neutral element. Thus we give these parameters to the Monoid class (note that \( Z \) is implicitly given).
Instance \texttt{ZMult} : Monoid \texttt{Zmult} 1.

For this instance to be created, we need to prove that the binary operation \texttt{Zmult} is associative and admits 1 as neutral element. Applying the constructor \texttt{Build_Monoid} — for instance with the tactic \texttt{split} — generates three subgoals.

\texttt{split.}
\texttt{3 subgoals}
\texttt{forall x y z : Z, x * (y * z) = x * y * z}

\texttt{subgoal 2 is:}
\texttt{forall x : Z, 1 * x = x}

\texttt{subgoal 3 is:}
\texttt{forall x : Z, x * 1 = x}

Each subgoal is easily solved by \texttt{intros; ring}.

When the proof is finished, we register our instance with a simple \texttt{Qed}.

Note that we used \texttt{Qed} because we consider a class of sort \texttt{Prop}. In some cases where instances must store some informative constants, ending an instance construction with \texttt{Defined} may be necessary.

\texttt{Check Zmult.}
\texttt{ZMult : Monoid Zmult 1}

We explained on the preceding page why it is better to use the \texttt{Class} keyword than \texttt{Record} or \texttt{Inductive}. For the same reason, the definition of an instance of some class should be written using \texttt{Instance} and not \texttt{Lemma}, \texttt{Theorem}, \texttt{Example}, etc., nor \texttt{Definition}.

### 2.2.2 A generic definition of \texttt{power}

We are now able to give a definition of the function \texttt{power} that can be applied with any instance of class \texttt{Monoid}:

A first definition could be:

\texttt{Fixpoint power \{A:Type\}{\texttt{dot}:A->A->A}{\texttt{one}:A}{\texttt{M}: Monoid \texttt{dot one}}
\texttt{(a:A)(n:nat) :=}
\texttt{match n with 0%nat \Rightarrow one}
\texttt{\mid S p \Rightarrow \texttt{dot} a (power a p)}
\texttt{end.}

\texttt{Compute power 2 10.}
\texttt{= 1024 : Z}

Happily, we can make the declaration of the three first arguments implicit, by using the \texttt{Generalizable Variables} command:
Reset power.

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.

Compute power 2 10.
= 1024 : Z

The variables A dot one appearing in the binder for M are implicitly bound before the binder for M and their types are inferred from the Monoid A dot one type. This syntactic sugar helps abbreviate bindings for classes with parameters. The resulting internal Coq term is exactly the same as the first definition above.

2.2.3 Instance Resolution

The attentive reader has certainly noticed that in the last computation, the binary operation Zmult and the neutral element 1 need not to be given explicitly\(^2\). The mechanism that allows Coq to infer all the arguments needed by the power function to be applied is called instance resolution.

In order to understand how it operates, let’s have a look at power’s type:

About power.

\[
\text{power :}
\forall (A : \text{Type}) (\text{dot} : A \to A \to A) (\text{one} : A),
\text{Monoid dot one} \to A \to \text{nat} \to A
\]

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.

We see that the instance ZMult has been inferred from the type of 2. We are in the simple case where only one monoid of carrier Z has been declared as an instance of the Monoid class.

The implementation of type classes in Coq can retrieve the instance ZMult from the type Z, then filling the arguments Zmult and 1 from ZMult’s definition.

\(^2\)This is quite different from the basic implicit arguments mechanism, already able to infer the type Z from the type of 2.
2.2.4 More Monoids

2.2.4.1 Matrices over some ring

We all know that multiplication of square matrices is associative and admits
identity matrices as neutral elements. For simplicity’s sake let us restrict our
study to $2 \times 2$ matrices over some ring.

We first load the Ring library, the open a section with some useful declarations
and notations. The reader may consult Coq’s documentation about the
Ring library.

Require Import Ring.

Section matrices.

Variables (A:Type)

(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.

We can now define a carrier type for $2 \times 2$-matrices, as well as matrix mul-
tiplication and the identity matrix:

Structure M2 : Type := {c00 : A; c01 : A;

\hspace{1cm} 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').

As for multiplication of integers, we can now define an instance of Monoid
for the type M2.

Global Instance M2_Monoid : Monoid M2_mult Id2.

(* Proof skipped *)

Qed.

End matrices.
We want now to play with $2 \times 2$ matrices over $\mathbb{Z}$. We declare an instance $M2\mathbb{Z}$ for this purpose, and can use directly the function $\text{power}$.

**Instance $M2\mathbb{Z}$:**

```coq
Instance M2Z : Monoid _ _ := M2_Monoid Zth.
```

**Compute**

```coq
Compute power (Build_M2 1 1 1 0) 40.
= {|
c00 := 165580141;
c01 := 102334155;
c10 := 102334155;
c11 := 63245986 |
} : M2 Z
```

**Definition**

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

**Compute**

```coq
Compute fibonacci 20.
= 10946 : Z
```

### 2.3 Reasoning within a Type Class

We are now able to consider again the equivalence between two functions for computing powers. Let us define the binary algorithm for any monoid:

First, we define an auxiliary function. We use the `Program` extension to define an efficient version of exponentiation using an accumulator. The function is defined by well-founded recursion on the exponent $n$:

**Function**

```coq
Function binary_power_mult (A:Type) (dot:A->A->A) (one:A) (M: @Monoid A dot one) (acc x:A)(n:nat){measure (fun i=>i) n} : A (* acc * (x ** n) *) :=
  match n with
  | 0%nat => acc
  | _  => if Even.even_odd_dec n
        then binary_power_mult _ acc (dot x x) (div2 n)
        else binary_power_mult _ (dot acc x) (dot x x) (div2 n)
  end.
```

**Definition**

```coq
Definition binary_power '{M:Monoid} x n := binary_power_mult M one x n.
```

**Compute**

```coq
Compute binary_power 2 100.
= 1267650600228229401496703205376 : Z
```
2.3.1 The Equivalence Proof

The proof of equivalence between `power` and `binary_power` is quite long, and can be split in several lemmas. Thus, it is useful to open a section, in which we fix some arbitrary monoid \( M \). Such a declaration is made with the `Context` command, which can be considered as a version of `Variables` for declaring arbitrary instances of a given class.

Section About_power.

Require Import Arith.

Context `{M:Monoid A dot one}.

It is a good practice to define locally some specialized notations and tactics.

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).

2.3.2 Some Useful Lemmas About power

We start by proving some well-known equalities about powers in a monoid. Some of these equalities are integrated later in simplification tactics.

Lemma power_x_plus : forall x n p, x ** (n + p) = x ** n * x ** p.
Proof.
  induction n as [| p IHp];simpl.
  intros; monoid_simpl;trivial.
  intro q;rewrite (IHp q); monoid_simpl;trivial.
Qed.

Ltac power_simpl := repeat (monoid_rw || rewrite <- power_x_plus).

Lemma power_commute : forall x n p, 
  x ** n * x ** p = x ** p * x ** n.
(* Proof skipped *)

Lemma power_commute_with_x : forall x n,

10
\[ x \times x^{n} = x^{n} \times x. \]
(* Proof skipped *)

Lemma power_of_power : \text{forall } x \text{ n \ p, } (x \times n)^{p} = x \times (p \times n).
(* Proof skipped *)

Lemma power_S : \text{forall } x \text{ n, } x \times x^{n} = x^{S n}.
(* Proof skipped *)

Lemma sqr : \text{forall } x, x^{2} = x \times x.
(* Proof skipped *)

Ltac factorize := repeat (
    rewrite <- power_commute_with_x ||
    rewrite <- power_x_plus ||
    rewrite <- sqr ||
    rewrite power_S ||
    rewrite power_of_power).

Lemma power_of_square : \text{forall } x \text{ n, } (x \times x)^{n} = x^{n} \times x^{n}.
(* Proof skipped *)

2.3.3 Final Steps
We are now able to prove that the auxiliary function \text{binary_power_mult} satisfies its intuitive meaning. The proof — partly skipped — uses well-founded induction and the lemmas proven in the previous section:

Lemma \text{binary_power_mult_ok} :
\text{forall } n \text{ a \ x, } \text{binary_power_mult } M \text{ a \ x \ n} = a \times x^{n}.
Proof.
    intro n; pattern n; apply lt_wf_ind.
    (* Proof skipped *)

Then the main theorem follows immediately:

Lemma \text{binary_power_ok} : \text{forall } (x:A)(n:nat), \text{binary_power } x \text{ n} = x^{n}.
Proof.
    intros n x; unfold \text{binary_power}; rewrite \text{binary_power_mult_ok};
    monoid_simpl;auto.
    Qed.

2.3.4 Discharging the Context
It is time to close the section we opened for writing our proof of equivalence. The theorem \text{binary_power_ok} is now provided with an universal quantification over all the parameters of any monoid.
About binary_power_ok.

\[
\text{binary\_power\_ok} : \forall (A : \text{Type}) \ (\text{dot} : A \to A \to A) \ (\text{one} : A) \ (M : \text{Monoid} \ \text{dot} \ \text{one}) \ (x : A) \ (n : \text{nat}), \ \text{binary\_power} \ x \ n = \text{power} \ x \ n
\]

Arguments \(A\), \(\text{dot}\), \(\text{one}\), \(M\) are implicit and maximally inserted
Argument scopes are \([\text{type\_scope} \ _ \ _ \ _ \ _ \ \text{nat\_scope}]\)
\text{binary\_power\_ok} is opaque
Expands to: \text{Constant} \ \text{Top.binary\_power\_ok}

Check \text{binary\_power\_ok} 2 20.

\[
\text{binary\_power\_ok} \ 2 \ 20
\]

: \text{binary\_power\_ok} \ 2 \ 20 = \text{power} \ 2 \ 20

Let \(M\text{fib} := \text{Build\_M2} \ 1 \ 1 \ 1 \ 0\).

Check \text{binary\_power\_ok} \ M\text{fib} \ 56.

\[
\text{binary\_power\_ok} \ M\text{fib} \ 56
\]

: \text{binary\_power\_ok} \ M\text{fib} \ 56 = \text{power} \ M\text{fib} \ 56

2.3.5 Subclasses

We could prove many useful equalities in the section \text{about\_power}. Nevertheless, we couldn’t prove the equality \((xy)^n = x^n y^n\), because it is false in general — consider for instance the free monoid of strings, or simply matrix multiplication. Nevertheless, this equality holds in every commutative (a.k.a. abelian) monoid.

Thus we say that abelian monoids form a subclass of the class of monoids, and prove this equality in a context declaring an arbitrary instance of this subclass.

Structurally, we parameterize the new class \text{Abelian\_Monoid} by an arbitrary instance \(M\) of \text{Monoid}, and add a new field stating the commutativity of \(\text{dot}\).

Please keep in mind that we declared \(A\), \(\text{dot}\) and \(\text{one}\) as \text{generalizable variables}, hence we can use the backquote symbol here:

Class \text{Abelian\_Monoid} ‘(M:\text{Monoid} \ A \ \text{dot} \ \text{one}) := {\n\text{dot\_comm} : \forall x y, \ \text{dot} \ x \ y = \text{dot} \ y \ x}.

A quick look at the representation of \text{Abelian\_Monoid} as a record type helps us understand how this class is implemented.

Print \text{Abelian\_Monoid}.

Record \text{Abelian\_Monoid} (A : \text{Type}) (\text{dot} : A \to A \to A) \ (\text{one} : A) \ (M : \text{Monoid} \ \text{dot} \ \text{one}) : \text{Prop} := \text{Build\_Abelian\_Monoid}
For building an instance of \texttt{Abelian\_Monoid}, we can start from \texttt{ZMult}, the monoid on \texttt{Z}, adding a proof that integer multiplication is commutative.

\begin{verbatim}
Instance ZMult_Abelian : Abelian\_Monoid ZMult.
Proof.
  split.
  exact Zmult_comm.
Defined.
\end{verbatim}

We can now prove our equality by building an appropriate context. Note that we can specify just the parameters of the monoid here in the binder of the abelian monoid, an instance of monoid on those same parameters is automatically generalized. Superclass parameters are automatically generalized inside quoted binders. Again, this is simply syntactic sugar.

Section Power\_of\_dot.
Context '{AM:Abelian\_Monoid A dot one}.

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.

End Power\_of\_dot.

Check power\_of\_mult 3 4 5.
\begin{verbatim}
  power\_of\_mult 3 4 5
  : power (4 * 5) 3 = power 4 3 * power 5 3
\end{verbatim}
Chapter 3

Relations and rewriting

3.1 Introduction: Lost in Manhattan

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.
If we represent such routes by lists of directions, you will consider that the two considered routes

- North::North::East::South::South::South::West::West::nil
- South::West::nil

are not equal — because the former route is much longer than the other one —, but equivalent, which means that they both lead to the same point.
Then you notice that the route West::North::East::South::nil is equivalent to nil (which means “don’t move!”).
From both previous equivalences you want to infer simply that the long route

(North::North::East::South::South::South::West::West::nil)++
(West::North::East::South::nil)

is equivalent to South::West::nil.
Moreover, you can consider this equivalence as a congruence w.r.t. route concatenation. For instance, one can easily show that the above routes are equivalent using a lemma stating that if the route $r$ is equivalent to $r'$, and $s$ is equivalent to $s'$ then $r+s$ is equivalent to $r'+s'$. We will say that
list concatenation is a proper function with respect to the equivalence between routes.

The Coq system provides now some useful tools for considering relations and proper functions, allowing to use the rewrite tactics for relations that are weaker than the Leibniz equality. Examples of such relations include route-equivalence, non-canonical representation of finite sets, partial commutation monoids, etc.

Let us now start with our example.

3.2 Data Types and Definitions

We first load some useful modules, for representing directions, routes, and the discrete plane on which routers operate:

Require Import List ZArith Bool.
Open Scope Z_scope.

Inductive direction : Type := North | East | South | West.

Definition route := list direction.

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

Definition Point_O := Build_Point 0 0.

3.3 Route Semantics

A route is just a “program” for moving from some point to another one. The function move associates to any route a function from Point to Point.

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

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 Point_eqb (P P’:Point) :=
  Zeq_bool (Point_x P) (Point_x P’) &&
Lemma Point_eqb_correct : forall p p', Point_eqb p p' = true <-> p = p'.

Proof skipped.
Qed.

We consider that two routes are "equivalent" if they define the same moves. For instance, the routes East::North::West::South::East::nil and East::nil are equivalent.

Definition route_equiv : relation route :=
  fun r r' => forall P, move r P = move r' P.
Infix "=r=" := route_equiv (at level 70):type_scope.

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

3.4 On Route Equivalence

It is easy to study some abstract properties of the relation route_equiv. First, we prove that this relation is reflexive, symmetric and transitive:

Proof.
  intros r p;reflexivity.
Qed.

Proof.
  intros r r' H p; symmetry;apply H.
Qed.

Proof.
  intros r r' r'' H H' p; rewrite H; apply H'.
Qed.

Note that, despite these properties, the tactics reflexivity, symmetry, and transitivity are not directly usable on the type route.

Goalforall r, route_equiv r r.
intro; reflexivity.
The last error message comes from the fact that the tactic reflexivity consults a base of registered instance of a class named Reflexive. There exists also classes named Symmetric, Transitive, and Equivalence. This last class gathers the properties of reflexivity, symmetry and transitivity. Thus, we can register an instance of the Equivalence class (and not as a lemma as above).

Proof. split;
  [apply route_equiv_refl |
  apply route_equiv_sym |
  apply route_equiv_trans].
Qed.

The tactics reflexivity, etc. can now be applied to the relation route_equiv.

Goal forall r, r =r= r.
Proof. intro; reflexivity. Qed.

Note that it is possible to consider relations that are not equivalence relations, but are only transitive, or reflexive and transitive, etc. by using the right type classes.

### 3.5 Proper Functions

Since routes are represented as lists of directions, we wish to prove some route equivalences by composition of already proven equivalences, using some lemmas on consing and appending routes.

For instance, we can prove that if we add the same direction in front of two equivalent routes, the routes we obtain are still equivalent:

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

This lemma can be applied to re-use our previous example Ex1.

Goal (South::East::North::West::South::East::nil) =r= (South::East::nil).
Proof. apply route_cons;apply Ex1. Qed.

Note that the proof of route_cons contains a case analysis on d. Let us try to have a more direct proof:
Lemma route_cons' : forall r r' d, r =r= r' -> d::r =r= d::r'.
Proof.
  intros r r' d H;rewrite H.

Toplevel input, characters 80-89:
Error: Found no subterm matching "move r ?17869" in the current goal.
Abort.

What we really need is to tell to the rewrite tactic how to use route_cons for using an equivalence r =r= r' for replacing r with r' in a term of the form cons d r for proving directly the equivalence cons d r =r= cons d r'. In other words, we say that cons d is proper w.r.t. the relation route_equiv.

In Coq this fact can be declared as an instance of:

Proper (route_equiv ==> route_equiv) (cons d)

This notation, which requires the library Morphisms to be loaded, expresses that if two routes r and r' are related through route_equiv, then the routes d::r and d::r' are also related by route_equiv.

In the notation used by Proper the first occurrence of route_equiv refers to the arguments of cons d, and the second one to the result of this consing. The user must require and import the Morphisms module to use this notation.

Require Import Morphisms.

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

We can now use rewrite to replace some term by an equivalent one, provided the context is made by applications of one or several functions of the form cons d:

Goal forall r r', r =r= r' -> South::West::r =r= South::West::r'.
Proof.
  intros r r' H.

1 subgoal

r : route
r' : route
H : r =r= r'

===============================================
South :: West :: r =r= South :: West :: r'
Since the context of the variable \( r \) is composed of applications of proper function calls, the tactic `rewrite H` can be used to replace the occurrence of \( r \) by \( r' \):

`rewrite H; reflexivity. Qed.`

### 3.6 Some Other instances of Proper

We prove also that the function `move` is proper with respect to route equivalence and Leibniz equality on points:

**Instance move_Proper : Proper (route_equiv ==> eq ==> eq) move.**

*Proof.*

```
intros r r' Hr_r' p q Hpq. rewrite Hpq; apply Hr_r'.
Qed.
```

Let us prove now that list concatenation is proper w.r.t. `route_equiv`.

First, a technical lemma, that relates `move` and list concatenation:

**Lemma route-compose : forall r r' P, move (r++r') P = move r' (move r P).**

*Proof.*

```
induction r as [\_|d IHs]; simpl;
[auto | destruct d; intros; rewrite IHs; auto].
Qed.
```

Then the proof itself:

**Instance app_route_Proper : Proper (route_equiv ==> route_equiv ==> route_equiv) (@app direction).**

*Proof.*

```intros r r' H r'' r''' H' P.
repeat rewrite route-compose.
```

1 subgoal

```
  r : route
  r' : route
  H : r =r= r'
  r'' : route
  r''' : route
  H' : r'' =r= r'''
  P : Point
  now rewrite H, H'.
Qed.
```
We are now able to compose proofs of route equivalence:

Example Ex2 : 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 Ex3 : forall r r', r =r= r' ->
         North::East::South::West::r =r= r'.
Proof. intros r r' H. now rewrite Ex2. Qed.

Example Ex4 : forall r r', r++ North::East::South::West::r' =r= r++r'.
Proof. intros r r'. now rewrite Ex2. Qed.

This generalized rewriting principle applies to equivalence relations, but also
to e.g. order relations and can be used to rewrite under binders, where the usual
rewrite tactic fails. It is presented in more detail in the reference manual and
the article by Sozeau [2009].

A non proper function

It is easy to give an example of a non-proper function. Two routes may be
equivalent, but have different lengths.

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

3.7 Deciding Route Equivalence

Proofs of examples like Ex2 may seem too complex for proving that some routes
are equivalent. It is better to define a simple boolean function for deciding
equivalence, and use reflection in case we have closed terms of type route.

We first define a boolean function :

Definition route_eqb r r' : bool :=
    Point_eqb (move r Point_O) (move r' Point_O).

Some technical lemmas proved in Lost_in_NY lead us to prove the following
equivalence, that allows us to prove some path equivalences through a simple
computation:

Lemma route_equiv_equivb : forall r r', route_equiv r r' <->
                         route_eqb r r' = true.
Ltac route_eq_tac := rewrite route_equiv_equivb; reflexivity.

Example Ex1' : route_equiv (East::North::West::South::East::nil) (East::nil).
Proof. route_eq_tac. Qed.

3.8 Monoids and Setoids

We would like to prove that route concatenation is associative, and admits the empty route nil as a neutral element, making the type route the carrier of some monoid.

The Monoid class type defined in Sect. 3.9.2 cannot be used for this purpose, since the properties of associativity and being a neutral element are defined in terms of Leibniz equality.

We give below a definition of a new class EMonoid parameterized by an equivalence relation:

  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 }.

Fixpoint Epower '{M: EMonoid A E_eq dot one}(a:A)(n:nat) :=
match n with
| 0%nat => one
| S p => dot a (Epower a p)
end.

We register an instance of EMonoid:

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

We can readily compute exponentiation of routes using this monoid:

Goal forall n, Epower (South::North::nil) n =r= nil.
Proof. induction n; simpl.
  reflexivity.
Exercise It seems that route concatenation is commutative as far as route equivalence is concerned. Is this true? In this case, define a subclass of EMonoid that handles commutativity, and build an instance of this class for route concatenation and equivalence.

3.9 Advanced Features of Type Classes

3.9.1 Alternate definitions of the class Monoid

Note that A, dot and one are parameters of this definition, whilst they could have been defined as fields of a structure.

The following variant is correct too:

```coq
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}.
```

However, there is a flaw in the definition of Monoid': if for some reason one needs to consider two monoid structures M and M' on the same carrier, the sharing of the considered carrier would be very clumsy to express and to reason about.

Section TwoMonoids.
Variables M M' : Monoid'.
Hypothesis same_carrier : @carrier M = @carrier M'.
Hypothesis same_one : @one M = @one M'.
Toplevel input, characters 52-59:
Error:
In environment
  M : Monoid'
  M' : Monoid'
  same_carrier : carrier = carrier
The term "one" has type "carrier" while it is expected to have type "carrier".

A possible solution would be using JM equality, but we don’t get the simplicity of the approach of Sect 2.2.

Require Import JMeq.
Hypothesis same_one : JMeq (@one M) (@one M').
Any definition using both monoids in the same expression will be filled with coercions between the propositionally equal but not definitionally equal carriers and operations, so this is an unworkable solution.

A second variant could be to consider the carrier $A$ as the only parameter of the definition and to leave the binary operation and neutral element as fields of the class:

```lean
Class Monoid' (A:Type) := { 
  dot : A -> A -> A;
  one : A;
  dot_assoc : forall x y z:A, 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};
```

It is possible to define the subclass of abelian monoids with this representation:

```lean
Class AbelianMonoid' (A:Type) := { 
  underlying_monoid => Monoid' A;
  dot_comm : forall x y, dot x y = dot y x }; 
```

Section Foo.
  Variable A : Type.
  Context (AM : AbelianMonoid' A).
  Goal forall x y z, dot x (dot y z) = dot y (dot x z).
  Proof. intros x y z.
     repeat rewrite dot_assoc.
     now rewrite (dot_comm x y).
  Qed.
End Foo.

Require Import Arith.

Instance Mnat : Monoid' nat.
Proof. split with plus 0;auto with arith. Defined.

Instance AMnat : AbelianMonoid' nat.
Proof. split with Mnat. unfold dot;auto with arith. Defined.

Goal dot 3 4 = 7.
Proof. reflexivity. Qed.

Nevertheless, the reader will find in a paper by Spitters and van der Weegen [2011] arguments for disregarding this variant: In short, some clumsiness would appear if we want to make Monoid’ a member of new classes like Group, Ring, etc.
3.9.2 Operational Type Classes

Let us come back to the definitions of section Sect. 2.2. Let us assume we wish to write functions and mathematical statements with a syntax as close as possible to the standard mathematical notation. For instance, nested applications of the \texttt{dot} operation should be written with the help of an infix operator.

The \texttt{Infix} command cannot be used directly, since \texttt{dot} is not declared as a global constant.

\texttt{Infix "\*" := \texttt{dot} (at level 50, left associativity):M\_scope.}

\textit{Error: The reference \texttt{dot} was not found in the current environment.}

A solution from ibid. consists in declaring a \textit{singleton type class} for representing binary operators:

\texttt{Class monoid\_binop (A:Type) := monoid\_op : A -> A -> A.}

\textit{Nota:} Unlike multi-field class types, \texttt{monoid\_op} is not a constructor, but a transparent constant such that \texttt{monoid\_op f} can be \(\eta\beta\)-reduced into \texttt{f}.

It is now possible to declare an infix notation:

\texttt{Delimit Scope M\_scope with M.}
\texttt{Infix "\*" := monoid\_op: M\_scope.}
\texttt{Open Scope M\_scope.}

We can now give a new definition of \texttt{Monoid}, using the type \texttt{monoid\_binop} \(A\) instead of \(A \to A \to A\), and the infix notation \(x \ast y\) instead of \texttt{monoid\_op x y}:

\texttt{Class Monoid (A:Type)(dot \texttt{monoid\_op} A)(one : A) : Type :=}
\texttt{dot\_assoc : forall x y z \text{ A}, x \ast (y \ast z)= x \ast y \ast z;}
\texttt{one\_left : forall x, one \ast x = x;}
\texttt{one\_right : forall x, x \ast one = x.}

For building monoids like \texttt{ZMult} or \texttt{M2\_Monoid}, we first declare instances of \texttt{monoid\_binop}.

\texttt{Instance Zmult\_op : monoid\_binop Z := Zmult.}
\texttt{Instance ZMult : Monoid Zmult\_op 1.}
\texttt{Proof. split; intros; unfold Zmult\_op, monoid\_op; ring. Defined.}

Section matrices.

Variables (A:Type)
\texttt{(zero one \text{ A})}
\texttt{(plus mult minus : A \to A \to A)}
\texttt{(sym : A \to A).}

\texttt{(* Definitions skipped, see section 2.2.4.1 *)}
Global Instance M2_mult_op : monoid_binop M2 := M2_mult.

Global Instance M2_Monoid : Monoid M2_mult_op Id2.
Proof. (* Proof skipped *) Defined.
End matrices.

For dealing with matrices on $\mathbb{Z}$, we may instantiate the parameters $A$, $\text{zero}$, etc.

Instance M2_Z_op : monoid_binop (M2 Z) := M2_mult Zplus Zmult .
Instance M2_mono_Z : Monoid (M2_mult_op _) (Id2 _) := M2_Monoid Zth.

Compute $((\text{Build}_M2 1 1 1 0) \times (\text{Build}_M2 1 1 1 0))\%M.$
= $\begin{array}{c}
| c00 := 2; c01 := 1; c10 := 1; c11 := 1 |
\end{array}$
: M2 Z

Compute $((\text{Id2} 0 1) \times (\text{Id2} 0 1))\%M.$
= $\begin{array}{c}
| c00 := 1; c01 := 0; c10 := 0; c11 := 1 |
\end{array}$
: M2 Z

It is now easy to use the infix notation $\times$ in the definition of functions like power:

Generalizable Variables $A$ dot one.

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

Infix "**" := power (at level 30, no associativity):M_scope.

Compute $(2 ** 5) ** 2.$
= $1024 : \mathbb{Z}$

Compute $(\text{Build}_M2 1 1 1 0) ** 40.$
= $\begin{array}{c}
| c00 := 165580141;
| c01 := 102334155;
| c10 := 102334155;
| c11 := 63245986 |
\end{array}$
: M2 Z

All the techniques we used in Sect. 2.3.1 can be applied with operational type classes. The main section of this proof is as follows:

Section About_power.
Context `{Monoid A}.
Open Scope M_scope.

Ltac monoid_rw :=
  rewrite one_left || rewrite one_right || rewrite dot_assoc.
Ltac monoid_simpl := repeat monoid_rw.

Lemma power_x_plus : forall x n p, x ** (n + p) = x ** n * x ** p.
  Proof. (* Proof skipped *) Qed.
End About_power.

Notice that, when the section is closed, theorem statements keep the notations of M_scope:

Open Scope M_scope.
About power_of_power.

power_of_power :
forall {A : Type} (dot : monoid_binop A) (one : A)
(M : Monoid dot one) (x : A) (n p : nat),
(x ** n) ** p = x ** (p * n)

If we want to consider monoids w.r.t. some equivalence relation, it is possible to associate an operational type class to the considered relation:

Require Import Setoid Morphisms.

Class Equiv A := equiv : relation A.
Infix "==" := equiv (at level 70):type_scope.

Open Scope M_scope.
Class EMonoid {A:Type}(E_eq : Equiv A)
(E_dot : monoid_binop A)(E_one : A):= {
  E_rel := Equivalence equiv;
  dot_proper := Proper (equiv ==> equiv ==> equiv) E_dot;
  E_dot_assoc := forall x y z:A,
    x * (y * z) == x * y * z;
  E_one_left := forall x, E_one * x == x;
  E_one_right := forall x, x * E_one == x}.

The overloaded == notation will refer to the appropriate equivalence relation on the type of the arguments. One can develop in this fashion a hierarchy of structures. In the following we define the structure of semirings starting from a refinement of EMonoid.

(* Overloaded notations *)
Class RingOne A := ring_one : A.
Class RingZero A := ring_zero : A.
Class RingPlus A := ring_plus => monoid_binop A.
Class RingMult A := ring_mult => monoid_binop A.
Infix "+" := ring_plus.
Infix "*" := ring_mult.
Notation "0" := ring_zero.
Notation "1" := ring_one.

Typeclasses Transparent RingPlus RingMult RingOne RingZero.

Class Distribute '{Equiv A} (f g: A -> A -> A): Prop :=
{ distribute_l a b c: f a (g b c) == g (f a b) (f a c)
   ; distribute_r a b c: f (g a b) c == g (f a c) (f b c) }.

Class Commutative {A B} '{Equiv B} (m: A -> A -> B): Prop :=
commutativity x y : m x y == m y x.

Class Absorb A '{Equiv A} (m: A -> A -> A) (x : A) : Prop :=
{ absorb_l c : m x c == x
   ; absorb_r c : m c x == x }.

Class ECommutativeMonoid '{Equiv A} (E_dot : monoid_binop A) (E_one : A):=
{ e_commononoid_monoid => EMonoid equiv E_dot E_one;
  e_commononoid_commutative => Commutative E_dot }.

Class ESemiRing (A:Type) (E_eq :Equiv A) (E_plus : RingPlus A) (E_zero : RingZero A)
(E_mult : RingMult A) (E_one : RingOne A):=
{ add_monoid => ECommutativeMonoid equiv ring_plus ring_zero ;
  mul_monoid => EMonoid equiv ring_mult ring_one ;
  eering_dist => Distribute ring_mult ring_plus ;
  eering_0_mul => Absorb ring_mult 0 }.

Note that we use a kind of multiple inheritance here, as a semiring contains two monoids, one for addition and one for multiplication, that are related by distributivity and absorption laws. To distinguish between the corresponding monoid operations, we introduce the new operational type classes Ring*. These classes are declared Transparent for typeclass resolution, so that their expansion to monoid_binop can be used freely during conversion: they are just abbreviations used for overloading notations.

We also introduce classes for the standard properties of operations like commutativity, distributivity etc... to be able to refer to them generically.

We can now develop some generic theory on semirings using the overloaded lemmas about distributivity or the constituent monoids. Resolution automatically goes through the ESemiRing structure to find proofs regarding the underlying monoids.
Section SemiRingTheory.
Context `{ESemiRing A}.

Definition ringtwo := 1 + 1.
Lemma ringtwomult : forall x : A, ringtwo * x == x + x.
Proof.
  intros. unfold ringtwo.
  rewrite distribute_r.
  now rewrite (E_one_left x).
Qed.

End SemiRingTheory.

3.9.3 Instance Resolution

Let us consider again the term written “binary_power 2 55”. This term —
with all its arguments explicited — is “@binary_power Z Zmult_op 1 2 55”.
The implicit arguments Z, Zmult_op and 1 have been inferred from the type Z of
2, looking at the instances of Monoid that were compatible with the instanciation
A:=Z.

If we consider another instance of Monoid with the same domain, the instance
resolution mechanism, which relies on auto, may not respect the user’s intuition.

For instance, let us temporarily consider another monoid on Z.

Section Z_additive.
Instance Z_plus_op := Zplus.

Instance ZAdd : Monoid Z_plus_op 0.
Proof. split;intros;unfold Z_plus_op, mon_op;ring. Defined.

Unfortunately, two instances of monoid_binop are linked to the type Z. Thus,
a notation like 2 * 5 looks ambiguous:

In the interaction below, the instance Z_plus_op is preferred over Z_mult_op,

Compute 2 * 5.
= 7
: Z

To force an unambiguous interpretation regardless of the order of decla-
rations, one solution is to add to the instance declaration a cost, which is a
natural number: the lowest the cost, the higher is the priority. Let us consider
the following two instance declarations:

Instance Zmult_op := monoid_binop Z | 1 := Zmult.
Instance Zplus_op := monoid_binop Z | 2 := Zplus.
Compute $(2 \times 5) \% M.$

$= 10 : \text{nat}$

Now the highest priority (lowest cost) instance is selected.
Conclusion

This concludes our tour of type classes and generalized rewriting. For more information on classes, one can consult the lecture notes by Sozeau [2012] which develops other examples of use. Spitters and van der Weegen [2011] develop a large hierarchy of structures using classes, including pervasive use of generalized rewriting. The current version of their library is available online at: https://github.com/math-classes. The library on categories and categorical syntax and semantics by Benedikt Ahrens [Ahrens] uses heavily type classes and is worth reading.

Generalized rewriting [Sozeau 2009] is an example of the use of type classes to develop a *generic* tactic that can apply to arbitrary user constants as long as “Proper” instances are declared on them. As we’ve seen, the very nature of type classes is to do proof search at typechecking-time. They are hence very fit to implement generic tactics that depend on some properties of the terms fed to them, for example associativity or commutativity of functions in the work of Braibant and Pous [2012]. They can also be used to concisely specify domain-specific proof automation, solving side conditions for a particular class of problems as in Gonthier et al. [2011]. These papers provide more advanced examples and design patterns for working with type classes.
Bibliography


