SOFTWARE FOUNDATIONS

VOLUME 4: QUICKCHICK: PROPERTY-BASED TESTING IN COQ

TABLE OF CONTENTS INDEX

TYPECLASSES

A TUTORIAL ON TYPECLASSES IN COQ

In real-world programming, it is often necessary to convert various kinds of data structures into strings so that they can be printed out, written to files, marshalled (arrange or assemble (a group of people, especially soldiers) in order) for sending over the network, or whatever. This can be accomplished by writing string converters for each basic type

- `showBool : bool → string`
- `showNat : nat → string`
- etc.

plus combinators for structured types like `list` and pairs

- `showList : {A : Type} (A → string) → (list A) → string`
- `showPair : {A B : Type} (A → string) → (B → string) → A * B → string`

that take string converters for their element types as arguments. Once we've done this, we can build string converters for more complex structured types by assembling them from these pieces:

- `showListOfPairsOfNats = showList (showPair showNat showNat)`

While this idiom gets the job done, it feels a bit clunky in at least two ways. First, it demands that we give names to all these string converters (which must later be remembered!) whereas it seems the names could really just be generated in a uniform way from the types involved. Moreover, the definitions of converters like `showListOfPairsOfNats` are always derived in a quite mechanical way from their types, together with a small collection of primitive converters and converter combinators. (like showNat, showBool; like showList, showPair)

The designers of Haskell addressed this clunkiness through *typeclasses*, a mechanism by which the typechecker is instructed to automatically construct "type-driven" functions [Wadler and Blott 1989]. (Readers not already familiar with typeclasses should note that, although the word sounds a bit like "classes" from object-oriented programming, this apparent connection is rather misleading. A better analogy is actually with *interfaces*

from languages like Java. But best of all is to set aside object-oriented preconceptions and try to approach typeclasses with an empty mind!)

Many other modern language designs have followed Haskell's lead, and Coq is no exception. However, because Coq's type system is so much richer than that of Haskell, and because typeclasses in Coq are used to automatically construct not only programs but also proofs, Coq's presentation of typeclasses is quite a bit less "transparent": to use typeclasses effectively, one must have a fairly detailed understanding of how they are implemented. Coq typeclasses are a power tool: they can make complex developments much more elegant and easy to manage when used properly, and they can cause a great deal of trouble when things go wrong!

This tutorial introduces the core features of Coq's typeclasses, explains how they are implemented, surveys some useful typeclasses that can be found in Coq's standard library and other contributed libraries, and collects some advice about the pragmatics of using typeclasses.

Basics

Classes and Instances

To automate converting various kinds of data into strings, we begin by defining a typeclass called `Show`.

```
Class Show A : Type :=
  {
    show : A → string
  }.
```

The `Show` typeclass can be thought of as "classifying" types whose values can be converted to strings — that is, types `A` such that we can define a function `show` of type `A → string`.

We can declare that `bool` is such a type by giving an `Instance` declaration that witnesses this function:

```
Instance showBool : Show bool :=
  {
    show := fun b:bool ⇒ if b then "true" else "false"
  }.

Compute (show true).
```

Other types can similarly be equipped with `Show` instances — including, of course, new types that we define.

```
Inductive primary := Red | Green | Blue.

Instance showPrimary : Show primary :=
  {
    show :=
      fun c:primary ⇒
```

```
        match c with
        | Red ⇒ "Red"
        | Green ⇒ "Green"
        | Blue ⇒ "Blue"
        end
  }.

Compute (show Green).
```

The `show` function is sometimes said to be *overloaded*, since it can be applied to arguments of many types, with potentially radically different behavior depending on the type of its argument.

Converting natural numbers to strings is conceptually similar, though it requires a tiny bit of programming:

```
Fixpoint string_of_nat_aux (time n : nat) (acc : string) : string
:=
  let d := match n mod 10 with
           | 0 ⇒ "0" | 1 ⇒ "1" | 2 ⇒ "2" | 3 ⇒ "3" | 4 ⇒ "4" |
5 ⇒ "5"
           | 6 ⇒ "6" | 7 ⇒ "7" | 8 ⇒ "8" | _ ⇒ "9"
           end in
  let acc' := d ++ acc in
  match time with
    | 0 ⇒ acc'
    | S time' ⇒
      match n / 10 with
        | 0 ⇒ acc'
        | n' ⇒ string_of_nat_aux time' n' acc'
      end
  end.

Definition string_of_nat (n : nat) : string :=
  string_of_nat_aux n n "".

Instance showNat : Show nat :=
  {
    show := string_of_nat
  }.

Compute (show 42).
```

Exercise: 1 star (showNatBool)

Write a `Show` instance for pairs of a nat and a bool.

```
  (* FILL IN HERE *)
```
□

Next, we can define functions that use the overloaded function `show` like this:

```
Definition showOne {A : Type} `{Show A} (a : A) : string :=
  "The value is " ++ show a.

Compute (showOne true).
Compute (showOne 42).
```

The parameter `{Show A} is a *class constraint*, which states that the function showOne is expected to be applied only to types A that belong to the Show class.

Concretely, this constraint should be thought of as an extra parameter to showOne supplying *evidence* that A is an instance of Show — i.e., it is essentially just a show function for A, which is implicitly invoked by the expression show a.

More interestingly, a single function can come with multiple class constraints:

```
Definition showTwo {A B : Type}
           `{Show A} `{Show B} (a : A) (b : B) : string :=
  "First is " ++ show a ++ " and second is " ++ show b.

Compute (showTwo true 42).
Compute (showTwo Red Green).
```

In the body of showTwo, the type of the argument to each instance of show determines which of the implicitly supplied show functions (for A or B) gets invoked.

Exercise: 1 star (missingConstraint)

e.g.: if we omit `{Show A} we'll get:
Unable to satisfy the following constraints:
In environment:
A : Type
B : Type
H : Show B
a : A
b : B

?Show : "Show A"

What happens if we forget the class constraints in the definitions of showOne or showTwo? Try deleting them and see what happens. □

Of course, Show is not the only interesting typeclass. There are many other situations where it is useful to be able to choose (and construct) specific functions depending on the type of an argument that is supplied to a generic function like show. Another typical example is equality checkers.

Here is another basic example of typeclasses: a class Eq describing types with a (boolean) test for equality.

```
Class Eq A :=
  {
    eqb: A → A → bool;
  }.
```

And here are some basic instances:

```
Instance eqBool : Eq bool :=
  {
    eqb := fun (b c : bool) ⇒
       match b, c with
         | true, true ⇒ true
         | true, false ⇒ false
         | false, true ⇒ false
         | false, false ⇒ true
       end
  }.

Instance eqNat : Eq nat :=
  {
    eqb := beq_nat
  }.
```

One possible confusion should be addressed here: Why should we need to define a typeclass for boolean equality when Coq's *propositional* equality (`x = y`) is completely generic? The answer is that, while it makes sense to *claim* that two values `x` and `y` are equal no matter what their type is, it is not possible to write a decidable equality *checker* for arbitrary types. In particular, equality at types like `nat→nat` is undecidable.

really deep!!

Exercise: 3 stars, optional (boolArrowBool)

There are some function types, like `bool→bool`, for which checking equality makes perfect sense. Write an `Eq` instance for this type.

```
  (* FILL IN HERE *)
```

□

Parameterized Instances: New Typeclasses from Old

What about a `Show` instance for pairs?

Since we can have pairs of any types, we'll want to parameterize our `Show` instance by two types. Moreover, we'll need to constrain both of these types to be instances of `Show`.

```
Instance showPair {A B : Type} `{Show A} `{Show B} : Show (A * B)
:=
  {
    show p :=
      let (a,b) := p in
        "(" ++ show a ++ "," ++ show b ++ ")"
  }.

Compute (show (true,42)).
```

Similarly, here is an `Eq` instance for pairs...

```
Instance eqPair {A B : Type} `{Eq A} `{Eq B} : Eq (A * B) :=
  {
    eqb p1 p2 :=
      let (p1a,p1b) := p1 in
      let (p2a,p2b) := p2 in
      andb (eqb p1a p2a) (eqb p1b p2b)
  }.
```

...and here is `Show` for lists:

```
Fixpoint showListAux {A : Type} (s : A → string) (l : list A) :
string :=
  match l with
    | nil ⇒ ""
    | cons h nil ⇒ s h
    | cons h t ⇒ append (append (s h) ", ") (showListAux s t)
  end.

Instance showList {A : Type} `{Show A} : Show (list A) :=
  {
```

```
    show l := append "[" (append (showListAux show l) "]")
  }.
```

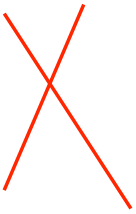
Exercise: 3 stars (eqEx)

Write an `Eq` instance for lists and `Show` and `Eq` instances for the `option` type constructor.

```
  (* FILL IN HERE *)
```

□

Exercise: 3 stars, optional (boolArrowA)



Generalize your solution to the `boolArrowBool` exercise to build an equality instance for any type of the form `bool→A`, where `A` itself is an `Eq` type. Show that it works for `bool→bool→nat`.

```
  (* FILL IN HERE *)
```

□

Class Hierarchies

We often want to organize typeclasses into hierarchies. For example, we might want a typeclass `Ord` for "ordered types" that support both equality and a less-or-equal comparison operator.

A possible (but bad) way to do this is to define a new class with two associated functions:

```
Class OrdBad A :=
  {
    eqbad : A → A → bool;
    lebad : A → A → bool
  }.
```

The reason this is bad is because we now need to use a new equality operator (`eqbad`) if we want to test for equality on ordered values.

```
Definition lt {A: Type} `{Eq A} `{OrdBad A} (x y : A) : bool :=
  andb (lebad x y) (negb (eqbad x y)).
```

A much better way is to parameterize the definition of `Ord` on an `Eq` class constraint:

```
Class Ord A `{Eq A} : Type :=
  {
    le : A → A → bool
  }.

Check Ord.
```

(The old class `Eq` is sometimes called a "superclass" of `Ord`, but, again, this terminology is potentially confusing: Try to avoid thinking about analogies with object-oriented programming!)

When we define instances of `Ord`, we just have to implement the `le` operation.

```
Instance natOrd : Ord nat :=
  {
```

```
    le := Nat.leb
  }.
```

Functions expecting to be instantiated with an instance of `Ord` now have two class constraints, one witnessing that they have an associated `eqb` operation, and one for `le`.

```
Definition max {A: Type} `{Eq A} `{Ord A} (x y : A) : A :=
  if le x y then y else x.
```

Exercise: 1 star (missingConstraintAgain)

What does Coq say if the `Ord` class constraint is left out of the definition of `max`? What about the `Eq` class constraint? □

```
===>
Unable to satisfy the following constraints:
In environment:
A : Type
H : Eq A
x, y : A

?H : "Eq A"

?Ord : "Ord A"
```

it should be fine

Exercise: 3 stars (ordMisc)

Define `Ord` instances for options and pairs.

```
(* FILL IN HERE *)
```

□

Exercise: 3 stars (ordList)

For a little more practice, define an `Ord` instance for lists.

```
(* FILL IN HERE *)
```

□

How It Works

Typeclasses in Coq are a powerful tool, but the expressiveness of the Coq logic makes it hard to implement sanity checks like Haskell's "overlapping instances" detector.

As a result, using Coq's typeclasses effectively — and figuring out what is wrong when things don't work — requires a clear understanding of the underlying mechanisms.

Implicit Generalization

The first thing to understand is exactly what the "backtick" notation means in declarations of classes, instances, and functions using typeclasses. This is actually a quite generic mechanism, called *implicit generalization*, that was added to Coq to support typeclasses but that can also be used to good effect elsewhere.

The basic idea is that unbound variables mentioned in bindings marked with ` are automatically bound in front of the binding where they occur.

To enable this behavior for a particular variable, say `A`, we first declare `A` to be implicitly generalizable:

```
Generalizable Variables A.
```

By default, Coq only implicitly generalizes variables declared in this way, to avoid puzzling behavior in case of typos. There is also a `Generalize Variables All`

command, but it's probably not a good idea to use it!

Now, for example, we can shorten the declaration of the `showOne` function by omitting the binding for `A` at the front.

```
                                {A:Type}
Definition showOne1 `{Show A} (a : A) : string :=
  "The value is " ++ show a.
```

Coq will notice that the occurrence of `A` inside the `` `{...} `` is unbound and automatically insert the binding that we wrote explicitly before.

```
Print showOne1.
(* ==>
    showOne1 =
      fun (A : Type) (H : Show A) (a : A) => "The value is " ++ show a
           : forall A : Type, Show A -> A -> string

    Arguments A, H are implicit and maximally inserted
*)
```

The "implicit and maximally generalized" annotation on the last line means that the automatically inserted bindings are treated as if they had been written with `{...}`, rather than `(...)`. The "implicit" part means that the type argument `A` and the `Show` witness `H` are usually expected to be left implicit: whenever we write `showOne1`, Coq will automatically insert two unification variables as the first two arguments. This automatic insertion can be disabled by writing `@`, so a bare occurrence of `showOne1` means the same as `@showOne1 _ _`. The "maximally inserted" part says that these arguments should inserted automatically even when there is no following explicit argument.

In fact, even the `` `{Show A} `` form hides one bit of implicit generalization: the bound name of the `Show` constraint itself. You will sometimes see class constraints written more explicitly, like this...

```
Definition showOne2 `{_ : Show A} (a : A) : string :=
  "The value is " ++ show a.
```

... or even like this:

```
Definition showOne3 `{H : Show A} (a : A) : string :=
  "The value is " ++ show a.
```

The advantage of the latter form is that it gives a name that can be used, in the body, to explicitly refer to the supplied evidence for `Show A`. This can be useful when things get complicated and you want to make your code more explicit so you can better understand and control what's happening.

We can actually go one bit further and omit `A` altogether, with no change in meaning (though, again, this may be more confusing than helpful):

```
Definition showOne4 `{Show} a : string :=
  "The value is " ++ show a.
```

If we ask Coq to print the arguments that are normally implicit, we see that all these definitions are exactly the same internally.

```
Set Printing Implicit.
Print showOne.
Print showOne1.
Print showOne2.
Print showOne3.
Print showOne4.
(* ==>
    showOne =
        fun (A : Type) (H : Show A) (a : A) =>
          "The value is " ++ @show A H a
      : forall A : Type, Show A -> A -> string
*)
Unset Printing Implicit.
```

The examples we've seen so far illustrate how implicit generalization works, but you may not be convinced yet that it is actually saving enough keystrokes to be worth the trouble of adding such a fancy mechanism to Coq. Where things become more convincing is when classes are organized into hierarchies. For example, here is an alternate definition of the `max` function:

```
Definition max1 `{Ord A} (x y : A) :=
  if le x y then y else x.
```

If we print out `max1` in full detail, we can see that the implicit generalization around `` `{Ord A} `` led Coq to fill in not only a binding for `A` but also a binding for `H`, which it can see must be of type `Eq A` because it appears as the second argument to `Ord`. (And why is `Ord` applied here to two arguments instead of just the one, `A`, that we wrote? Because `Ord`'s arguments are maximally inserted!)

```
Set Printing Implicit.
Print max1.
(* ==>
     max1 =
       fun (A : Type) (H : Eq A) (H0 : @Ord A H) (x y : A) =>
         if @le A H H0 x y then y else x

   : forall (A : Type) (H : Eq A),
       @Ord A H -> A -> A -> A
*)

Check Ord.
(* ==> Ord : forall A : Type, Eq A -> Type *)

Unset Printing Implicit.
```

For completeness, a couple of final points about implicit generalization. First, it can be used in situations where no typeclasses at all are involved. For example, we can use it to write quantified propositions mentioning free variables, following the common informal convention that these are to be quantified implicitly.

```
Generalizable Variables x y.

Lemma commutativity_property : `{x + y = y + x}.
Proof. intros. omega. Qed.
```

```
Check commutativity_property.
```

The previous examples have all shown implicit generalization being used to fill in forall binders. It will also create `fun` binders, when this makes sense:

```
Definition implicit_fun := `{x + y}.
```

Defining a function in this way is not very natural, however. In particular, the arguments are all implicit and maximally inserted (as can be seen if we print out its definition)...

```
Print implicit_fun.
```

... so we will need to use @ to actually apply the function:

```
(* Compute (implicit_fun 2 3). *)
(* ==>
    Error: Illegal application (Non-functional construction):
    The expression "implicit_fun" of type "nat"
    cannot be applied to the term
     "2" : "nat"
*)
      @: makes implicit args explicit!!
Compute (@implicit_fun 2 3).
```

Writing `` `(...) ``, with parentheses instead of curly braces, causes Coq to perform the same implicit generalization step, but does *not* mark the inserted binders themselves as implicit.

```
Definition implicit_fun1 := `(x + y).
Print implicit_fun1.
Compute (implicit_fun1 2 3).
```

Records are Products

Although records are not part of its core, Coq does provide some simple syntactic sugar for defining and using records.

Record types must be declared before they are used. For example:

```
Record Point :=
  Build_Point
    {
      px : nat;
      py : nat
    }.
```

Internally, this declaration is desugared into a single-field inductive type, roughly like this:

```
Inductive Point : Set :=
  | Build_Point : nat → nat → Point.
```

Elements of this type can be built, if we like, by applying the `Build_Point` constructor directly.

```
Check (Build_Point 2 4).
```

Or we can use more familar record syntax, which allows us to name the fields and write them in any order:

```
Check {| px := 2; py := 4 |}.
Check {| py := 4; px := 2 |}.
```

We can also access fields of a record using conventional "dot notation" (with slightly clunky concrete syntax):

```
Definition r : Point := {| px := 2; py := 4 |}.

Compute (r.(px) + r.(py)).
```

Record declarations can also be parameterized:

```
Record LabeledPoint (A : Type) :=
  Build_LabeledPoint
    {
      lx : nat;
      ly : nat;
      label : A
    }.
```

(Note that the field names have to be different. Any given field name can belong to only one record type. This greatly simplifies type inference!)

```
Check {| lx:=2; ly:=4; label:="hello" |}.
(* ==>
     {| lx := 2; ly := 4; label := "hello" |}
        : LabeledPoint string
*)
```

Exercise: 1 star (rcdParens)

Note that the `A` parameter in the definition of `LabeledPoint` is bound with parens, not curly braces. Why is this a better choice? □

Typeclasses are Records

Typeclasses and instances, in turn, are basically just syntactic sugar for record types and values (together with a bit of magic for using proof search to fill in appropriate instances during typechecking, as described below).

Internally, a typeclass declaration is elaborated into a parameterized `Record` declaration:

record

```
Set Printing All.
Print Show.
(* ==>
     Record Show (A : Type) : Type :=
       Build_Show
         { show : A -> string }
*)
Unset Printing All.
```

(If you run the `Print` command yourself, you'll see that `Show` actually displays as a `Variant`; this is Coq's terminology for a single-field record.)

Analogously, `Instance` declarations become record values:

value

```
Print showNat.
(* ==>
    showNat = {| show := string_of_nat |}
        : Show nat
*)
```

Note that the syntax for record values is slightly different from `Instance` declarations.

very confusing clunky syntax!!!

Record values are written with curly-brace-vertical-bar delimiters, while `Instance` declarations are written here with just curly braces. (To be precise, both forms of braces are actually allowed for `Instance` declarations, and either will work in most situations; however, type inference sometimes works a bit better with bare curly braces.)

Similarly, overloaded functions like `show` are really just record projections, which in turn are just functions that select a particular argument of a one-constructor `Inductive` type.

```
Set Printing All.
Print show.
(* ==>
    show =
      fun (A : Type) (Show0 : Show A) =>
        match Show0 with
          | Build_Show _ show => show
        end
   : forall (A : Type), Show A -> A -> string

   Arguments A, Show are implicit and maximally inserted  *)
Unset Printing All.
```

Inferring Instances

So far, all the mechanisms we've seen have been pretty simple syntactic sugar and binding munging. The real "special sauce" of typeclasses is the way appropriate instances are automatically inferred (and/or constructed!) during typechecking.

For example, if we write `show 42`, what we actually get is `@show nat showNat 42`:

```
Definition eg42 := show 42.

Set Printing Implicit.
Print eg42.
Unset Printing Implicit.
```

How does this happen?

First, since the arguments to `show` are marked implicit, what we typed is automatically expanded to `@show _ _ 42`. The first `_` should obviously be replaced by `nat`. But what about the second?

By ordinary type inference, Coq knows that, to make the whole expression well typed, the second argument to `@show` must be a value of type `Show nat`. It attempts to find or construct such a value using a variant of the `eauto` proof search procedure that refers to a "hint database" called `typeclass_instances`.

Exercise: 1 star (HintDb)

Uncomment and execute the following command. Search for "For Show" in the output and have a look at the entries for `showNat` and `showPair`.

```
(* Print HintDb typeclass_instances. *)
```

□

very interesting!!

We can see what's happening during the instance inference process if we issue the `Set Typeclasses Debug` command.

```
Set Typeclasses Debug.
Check (show 42).
(* ==>
     Debug: 1: looking for (Show nat) without backtracking
     Debug: 1.1: exact showNat on (Show nat), 0 subgoal(s)
*)
```

In this simple example, the proof search succeeded immediately because `showNat` was in the hint database. In more interesting cases, the proof search needs to try to assemble an *expression* of appropriate type using both functions and constants from the hint database. (This is very like what happens when proof search is used as a tactic to automatically assemble compound proofs by combining theorems from the environment.)

```
Check (show (true,42)).
(* ==>
     Debug: 1: looking for (Show (bool * nat)) without backtracking
     Debug: 1.1: simple apply @showPair on (Show (bool * nat)), 2 subgoal(s)
     Debug: 1.1.3 : (Show bool)
     Debug: 1.1.3: looking for (Show bool) without backtracking
     Debug: 1.1.3.1: exact showBool on (Show bool), 0 subgoal(s)
     Debug: 1.1.3 : (Show nat)
     Debug: 1.1.3: looking for (Show nat) without backtracking
     Debug: 1.1.3.1: exact showNat on (Show nat), 0 subgoal(s)      *)

Unset Typeclasses Debug.
```

In the second line, the search procedure decides to try applying `showPair`, from which it follows (after a bit of unification) that it needs to find an instance of `Show Nat` and an instance of `Show Bool`, each of which succeeds immediately as before.

In summary, here are the steps again:

```
show 42
   ===>   { Implicit arguments }
@show _ _ 42
   ===>   { Typing }
@show (?A : Type) (?Show0 : Show ?A) 42
```

```
    ===>   { Unification }
@show nat (?Show0 : Show nat) 42
    ===>   { Proof search for Show Nat returns showNat }
@show nat showNat 42
```

Typeclasses and Proofs

Since programs and proofs in Coq are fundamentally made from the same stuff, the mechanisms of typeclasses extend smoothly to situations where classes contain not only data and functions but also proofs.

This is a big topic — too big for a basic tutorial — but let's take a quick look at a few things.

Propositional Typeclass Members

The `Eq` typeclass defines a single overloaded function that tests for equality between two elements of some type. We can extend this to a subclass that also comes with a proof that the given equality tester is correct, in the sense that, whenever it returns `true`, the two values are actually equal in the propositional sense (and vice versa).

```
Class EqDec (A : Type) {H : Eq A} :=
  {
    eqb_eq : ∀ x y, eqb x y = true ↔ x = y
  }.
```

To build an instance of `EqDec`, we must now supply an appropriate proof.

```
Instance eqdecNat : EqDec nat :=
  {
    eqb_eq := Nat.eqb_eq
  }.
```

If we do not happen to have an appropriate proof already in the environment, we can simply omit it. If the `Instance` declaration does not give values for all the class members, Coq will enter proof mode and ask the user to use tactics to construct inhabitants for the remaining fields.

```
Instance eqdecBool' : EqDec bool :=
  {
  }.
Proof.
  intros x y. destruct x; destruct y; simpl; unfold iff; auto.
Defined.
```

(If we are omitting *all* the fields of an instance declaration, we can also omit the `:= {}` if we like. Note that the proof needs one more line.)

```
Instance eqdecBool'' : EqDec bool.
Proof.
  constructor.
```

```
  intros x y. destruct x; destruct y; simpl; unfold iff; auto.
Defined.
```

Given a typeclass with propositional members, we can use these members in proving things involving this typeclass.

Here, for example, is a quick (and somewhat contrived) example of a proof of a property that holds for arbitrary values from the `EqDec` class...

```
Lemma eqb_fact `{EqDec A} : ∀ (x y z : A),
  eqb x y = true → eqb y z = true → x = z.
Proof.
  intros x y z Exy Eyz.
  rewrite eqb_eq in Exy.
  rewrite eqb_eq in Eyz.
  subst. reflexivity. Qed.
```

There is much more to say about how typeclasses can be used (and how they should not be used) to support large-scale proofs in Coq. See the suggested readings below.

Substructures

Naturally, it is also possible to have typeclass instances as members of other typeclasses: these are called *substructures*. Here is an example adapted from the Coq Reference Manual.

```
Require Import Coq.Relations.Relation_Definitions.

Class Reflexive (A : Type) (R : relation A) :=
  {
    reflexivity : ∀ x, R x x
  }.

Class Transitive (A : Type) (R : relation A) :=
  {
    transitivity : ∀ x y z, R x y → R y z → R x z
  }.

Generalizable Variables z w R.

Lemma trans3 : ∀ `{Transitive A R},
    `{R x y → R y z → R z w → R x w}.
Proof.
  intros.
  apply (transitivity x z w). apply (transitivity x y z).
  assumption. assumption. assumption. Defined.

Class PreOrder (A : Type) (R : relation A) :=
  { PreOrder_Reflexive :> Reflexive A R ;
    PreOrder_Transitive :> Transitive A R }.
```

The syntax `:>` indicates that each `PreOrder` can be seen as a `Reflexive` and `Transitive` relation, so that, any time a reflexive relation is needed, a preorder can be used instead.

```
Lemma trans3_pre : ∀ `{PreOrder A R},
    `{R x y → R y z → R z w → R x w}.
Proof. intros. eapply trans3; eassumption. Defined.
```

Some Useful Typeclasses

Dec

The `ssreflect` library defines what it means for a proposition `P` to be decidable like this...

```
Require Import ssreflect ssrbool.

Print decidable.
(* ==>
     decidable = fun P : Prop => {P} + {~ P}
*)
```

... where `{P} + {¬ P}` is an "informative disjunction" of `P` and `¬P`.

It is easy to wrap this in a typeclass of "decidable propositions":

```
Class Dec (P : Prop) : Type :=
  {
    dec : decidable P
  }.
```

We can now create instances encoding the information that propositions of various forms are decidable. For example, the proposition `x = y` is decidable (for any specific `x` and `y`), assuming that `x` and `y` belong to a type with an `EqDec` instance.

```
Instance EqDec__Dec {A} `{H : EqDec A} (x y : A) : Dec (x = y).
Proof.
  constructor.
  unfold decidable.
  destruct (eqb x y) eqn:E.
  - left. rewrite <- eqb_eq. assumption.
  - right. intros C. rewrite <- eqb_eq in C. rewrite E in C.
inversion C.
Defined.
```

Similarly, we can lift decidability through logical operators like conjunction:

```
Instance Dec_conj {P Q} {H : Dec P} {I : Dec Q} : Dec (P ∧ Q).
Proof.
  constructor. unfold decidable.
  destruct H as [D]; destruct D;
    destruct I as [D]; destruct D; auto;
      right; intro; destruct H; contradiction.
Defined.
```

Exercise: 3 stars (dec_neg_disj)

Give instance declarations showing that, if `P` and `Q` are decidable propositions, then so are `¬P` and `P∨Q`.

```
  (* FILL IN HERE *)
```

□

Exercise: 4 stars (Dec_All)

The following function converts a list into a proposition claiming that every element of that list satiesfies some proposition `P`:

```
Fixpoint All {T : Type} (P : T → Prop) (l : list T) : Prop :=
  match l with
  | [] ⇒ True
  | x :: l' ⇒ P x ∧ All P l'
  end.
```

Create an instance of `Dec` for `All P l`, given that `P a` is decidable for every `a`.

```
(* FILL IN HERE *)
```

□

One reason for doing all this is that it makes it easy to move back and forth between the boolean and propositional worlds, whenever we know we are dealing with decidable propositions.

In particular, we can define a notation `P?` that converts a decidable proposition `P` into a boolean expression.

```
Notation "P '?'" :=
  (match (@dec P _) with
   | left _ ⇒ true
   | right _ ⇒ false
   end)
  (at level 100).
```

Now we don't need to remember that, for example, the test for equality on numbers is called `beq_nat`, because instead of this...

```
Definition silly_fun1 (x y z : nat) :=
  if andb (beq_nat x y) (andb (beq_nat y z) (beq_nat x z))
  then "all equal"
  else "not all equal".
```

... we can just write this:

```
Definition silly_fun2 (x y z : nat) :=
  if (x = y ∧ y = z ∧ x = z)?
  then "all equal"
  else "not all equal".
```

Monad

In Haskell, one place typeclasses are used very heavily is with the `Monad` typeclass, especially in conjunction with Haskell's "`do` notation" for monadic actions.

Monads are an extremely powerful tool for organizing and streamlining code in a wide range of situations where computations can be thought of as yielding a result along with some kind of "effect." Examples of possible effects include

- input / output
- state mutation

- failure
- nondeterminism
- randomness
- etc.

There are many good tutorials on the web about monadic programming in Haskell. Readers who have never seen monads before may want to refer to one of these to fully understand the examples here.

In Coq, monads are also heavily used, but the fact that typeclasses are a relatively recent addition to Coq, together with the fact that Coq's `Notation` extension mechanism allows users to define their own variants of `do` notation for specific situations where a monadic style is useful, have led to a proliferation of definitions of monads: most older projects simply define their own monads and monadic notations — sometimes typeclass-based, often not — while newer projects use one of several generic libraries for monads. Our current favorite (as of Summer 2017) is the monad typeclasses in Gregory Malecha's `ext-lib` package:

https://github.com/coq-ext-lib/coq-ext-lib/blob/v8.5/theories/Structures/Monad.v

Once the `ext-lib` package is installed (e.g., via OPAM), we can write:

```
Require Export ExtLib.Structures.Monads.
Export MonadNotation.
Open Scope monad_scope.
```

The main definition provided by this library is the following typeclass:

```
Class Monad (M : Type → Type) : Type :=
  {
    ret : ∀ {T : Type}, T → M T ;
    bind : ∀ {T U : Type}, M T → (T → M U) → M U
  }.
```

That is, a type family `M` is an instance of the `Monad` class if we can define functions `ret` and `bind` of the appropriate types. (If you `Print` the actual definition, you'll see something more complicated, involving `Polymorphic Record bla bla`... The `Polymorphic` part refers to Coq's "universe polymorphism," which does not concern us here.)

For example, we can define a monad instance for `option` like this:

```
Instance optionMonad : Monad option :=
  {
    ret T x :=
      Some x ;
    bind T U m f :=
      match m with
        None ⇒ None
      | Some x ⇒ f x
      end
  }.
```

The other nice thing we get from the `Monad` library is lightweight notation for bind: Instead of

```
bind m1 (fun x ⇒ m2),
```

we can write

```
x <- m1 ;; m2.
```

Or, if the result from m_1 is not needed in m_2, then instead of

```
bind m1 (fun _ ⇒ m2),
```

we can write

```
m1 ;; m2.
```

This allows us to write functions involving "option plumbing" very compactly.

For example, suppose we have a function that looks up the `n`th element of a list, returning `None` if the list contains less than `n` elements.

```
Fixpoint nth_opt {A : Type} (n : nat) (l : list A) : option A :=
  match l with
    [] ⇒ None
  | h::t ⇒ if eqb n 0 then Some h else nth_opt (n-1) t
  end.
```

We can write a function that sums the first three elements of a list (returning `None` if the list is too short) like this:

```
Definition sum3 (l : list nat) : option nat :=
  x0 <- nth_opt 0 l ;;
  x1 <- nth_opt 1 l ;;   nice!
  x2 <- nth_opt 2 l ;;
  ret (x0 + x1 + x2).
```

One final convenience for programming with monads is a collection of operators for "lifting" functions on ordinary values to functions on monadic computations. `ExtLib` defines three — one for unary functions, one for binary, and one for ternary. The definitions (slightly simplified) look like this:

```
Definition liftM
           {m : Type → Type}
           {M : Monad m}
           {T U : Type} (f : T → U)
         : m T → m U :=
  fun x ⇒ bind x (fun x ⇒ ret (f x)).

Definition liftM2
           {m : Type → Type}
           {M : Monad m}
           {T U V : Type} (f : T → U → V)
         : m T → m U → m V :=
    fun x y ⇒ bind x (fun x ⇒ liftM (f x) y).
```

```
Definition liftM3
           {m : Type → Type}
           {M : Monad m}
           {T U V W : Type} (f : T → U → V → W)
         : m T → m U → m V → m W :=
    fun x y z ⇒ bind x (fun x ⇒ liftM2 (f x) y z).
```

For example, suppose we have two `option nat`s and we would like to calculate their sum (unless one of them is `None`, in which case we want `None`). Instead of this...

```
Definition sum3opt (n1 n2 : option nat) :=
  x1 <- n1 ;;
  x2 <- n2 ;;
  ret (x1 + x2).
```

...we can use `liftM2` to write it more compactly:

```
Definition sum3opt' (n1 n2 : option nat) :=
  liftM2 plus n1 n2.
```

The `/examples` directory in the `ext-lib` Github repository (https://github.com/coq-ext-lib/coq-ext-lib) includes some further examples of using monads in Coq.

Others

Two popular typeclasses from the Coq standard library are `Equivalence` (and the associated classes `Reflexive`, `Transitive`, etc.) and `Proper`. They are described in the second half of *A Gentle Introduction to Type Classes and Relations in Coq*, by Castéran and Sozeau.
http://www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/typeclassestut.pdf.

A much larger collection of typeclasses for formalizing mathematics is described in *Type Classes for Mathematics in Type Theory*, by Bas Spitters and Eelis van der Weegen.
https://arxiv.org/pdf/1102.1323.pdf

Controlling Instantiation

"Defaulting"

The type of the overloaded `eqb` operator...

```
Check @eqb.
(* ==>
     @eqb : forall A : Type, Eq A -> A -> A -> bool    *)
```

... says that it works for any `Eq` type. Naturally, we can use it in a definition like this...

```
Definition foo x := if eqb x x then "Of course" else "Impossible".
```

... and then we can apply `foo` to arguments of any `Eq` type.

Right?

```
Fail Check (foo true).
(* ==>
     The command has indeed failed with message:
     The term "true" has type "bool" while it is expected
       to have type "bool -> bool". *)
```

Huh?!

Here's what happened:

- When we defined `foo`, the type of `x` was not specified, so Coq filled in a unification variable (an "evar") `?A`.
- When typechecking the expression `eqb x`, the typeclass instance mechanism was asked to search for a type-correct instance of `Eq`, i.e., an expression of type `Eq ?A`.
- This search immediately succeeded because the first thing it tried worked; this happened to be the constant `eqBoolBool : Eq (bool→bool)`. In the process, `?A` got instantiated to `bool→bool`.
- The type calculated for `foo` was therefore `(bool→bool)->(bool→bool)→bool`.

The lesson is that it matters a great deal *exactly* what problems are posed to the instance search engine.

Exercise: 1 star (debugDefaulting)

Do `Set Typeclasses Debug` and verify that this is what happened. □

Manipulating the Hint Database

One of the ways in which Coq's typeclasses differ most from Haskell's is the lack, in Coq, of an automatic check for "overlapping instances."

That is, it is completely legal to define a given type to be an instance of a given class in two different ways.

```
Inductive baz := Baz : nat → baz.

Instance baz1 : Show baz :=
  {
    show b :=
      match b with
        Baz n ⇒ "Baz: " ++ show n
      end
  }.

Instance baz2 : Show baz :=
  {
    show b :=
      match b with
        Baz n ⇒ "[" ++ show n ++ " is a Baz]"
      end
  }.

Compute (show (Baz 42)).
(* ==>
     = "42 is a Baz"
     : string   *)
```

When this happens, it is unpredictable which instance will be found first by the instance search process; here it just happened to be the second. The reason Coq doesn't do the overlapping instances check is because its type system is much more complex than Haskell's — so much so that it is very challenging in general to decide whether two given instances overlap.

The reason this is unfortunate is that, in more complex situations, it may not be obvious when there are overlapping instances.

One way to deal with overlapping instances is to "curate" the hint database by explicitly adding and removing specific instances.

To remove things, use `Remove Hints`:

```
Remove Hints baz1 baz2 : typeclass_instances.
```

To add them back (or to add arbitrary constants that have the right type to be intances — i.e., their type ends with an applied typeclass — but that were not created by `Instance` declarations), use `Existing Instance`:

```
Existing Instance baz1.
Compute (show (Baz 42)).
(* ==>
     = "Baz: 42"
     : string    *)

Remove Hints baz1 : typeclass_instances.
```

Another way of controlling which instances are chosen by proof search is to assign *priorities* to overlapping instances:

```
Instance baz3 : Show baz | 2 :=
  {
    show b :=
      match b with
        Baz n ⇒ "Use me first! " ++ show n
      end
  }.

Instance baz4 : Show baz | 3 :=
  {
    show b :=
      match b with
        Baz n ⇒ "Use me second! " ++ show n
      end
  }.

Compute (show (Baz 42)).
(* ==>
     = "Use me first!  42"
     : string  *)
```

0 is the highest priority.

If the priority is not specified, it defaults to the number of binders of the instance. (This means that more specific — less polymorphic — instances will be chosen over less

specific ones.)

`Existing Instance` declarations can also be given explicit priorities.

```
Existing Instance baz1 | 0.
Compute (show (Baz 42)).
(* ==>
     = "Baz: 42"
     : string    *)
```

Debugging

Instantiation Failures

One downside of using typeclasses, especially many typeclasses at the same time, is that error messages can become puzzling.

Here are some relatively easy ones.

```
Inductive bar :=
  Bar : nat → bar.

Fail Definition eqBar :=
  eqb (Bar 42) (Bar 43).

(* ===>
    The command has indeed failed with message:
    Unable to satisfy the following constraints:
       ?Eq : "Eq bar"  *)

Fail Definition ordBarList :=
  le [Bar 42] [Bar 43].

(* ===>
    The command has indeed failed with message:
    Unable to satisfy the following constraints:
      ?H : "Eq (list bar)"
      ?Ord : "Ord (list bar)" *)
```

In these cases, it's pretty clear what the problem is. To fix it, we just have to define a new instance. But in more complex situations it can be trickier.

A few simple tricks can be very helpful:

- Do `Set Printing Implicit` and then use `Check` and `Print` to investigate the types of the things in the expression where the error is being reported.
- Add some `@` annotations and explicitly fill in some of the arguments that should be getting instantiated automatically, to check your understanding of what they should be getting instantiated with.
- Turn on tracing of instance search with `Set Typeclasses Debug.`

The `Set Typeclasses Debug` command has a variant that causes it to print even more information: `Set Typeclasses Debug Verbosity 2.` Writing just `Set Typeclasses Debug` is equivalent to `Set Typeclasses Debug Verbosity 1.`

Another potential source of confusion with error messages comes up if you forget a `. For example:

```
Fail Definition max {A: Type} {Ord A} (x y : A) : A :=
  if le x y then y else x.
(* ===>
     The command has indeed failed with message:
     Unable to satisfy the following constraints:
     UNDEFINED EVARS:
      ?X354==A |- Type (type of Ord) {?T}
      ?X357==A0 Ord A x y |- Eq A (parameter H of @le) {?H}
      ?X358==A0 Ord A x y |- Ord A (parameter Ord of @le) {?
Ord}  *)
```

The `UNDEFINED EVARS` here is because the binders that are automatically inserted by implicit generalization are missing.

Nontermination

An even more annoying way that typeclass instantiation can go wrong is by simply diverging. Here is a small example of how this can happen.

Declare a typeclass involving two types parameters `A` and `B` — say, a silly typeclass that can be inhabited by arbitrary functions from `A` to `B`:

```
Class MyMap (A B : Type) : Type :=
  {
    mymap : A → B
  }.
```

Declare instances for getting from `bool` to `nat`...

```
Instance MyMap1 : MyMap bool nat :=
  {
    mymap b := if b then 0 else 42
  }.
```

... and from `nat` to `string`:

```
Instance MyMap2 : MyMap nat string :=
  {
    mymap := fun n : nat ⇒
      if le n 20 then "Pretty small" else "Pretty big"
  }.

Definition e1 := mymap true.
Compute e1.

Definition e2 := mymap 42.
Compute e2.
```

Notice that these two instances don't automatically get us from `bool` to `string`:

```
Fail Definition e3 : string := mymap false.
```

We can try to fix this by defining a generic instance that combines an instance from `A` to `B` and an instance from `B` to `C`:

```
Instance MyMap_trans {A B C : Type} `{MyMap A B} `{MyMap B C} :
MyMap A C :=
  { mymap a := mymap (mymap a) }.
```

This does get us from `bool` to `string` automatically:

```
Definition e3 : string := mymap false.
Compute e3.
```

However, although this example seemed to work, we are actually in a state of great peril: If we happen to ask for an instance that doesn't exist, the search procedure will diverge.

```
(*
Definition e4 : list nat := mymap false.
*)
```

Exercise: 1 star (nonterm)

Why, exactly, did the search diverge? Enable typeclass debugging, uncomment the above `Definition`, and see what gets printed. (You may want to do this from the command line rather than from inside an IDE, to make it easier to kill.)

Alternative Structuring Mechanisms

Typeclasses are just one of several mechanisms that can be used in Coq for structuring large developments. Others include:

- canonical structures
- bare dependent records
- modules and functors

An introduction to canonical structures and comparisons between canonical structures and typeclasses can be found here:

- Assia Mahboubi and Enrico Tassi. Canonical Structures for the working Coq user. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, ITP 2013, 4th Conference on Interactive Theorem Proving, volume 7998 of LNCS, pages 19–34, Rennes, France, 2013. Springer. https://hal.inria.fr/hal-00816703v$_1$/document
- Gonthier et al., "How to make ad hoc proof automation less ad hoc", JFP 23 (4): 357–401, 2013. (This explains some weaknesses of typeclasses and why canonical structures are used in in the `mathcomp` libraries.)

A useful discussion of typeclasses vs. dependent records is: https://stackoverflow.com/questions/29872260/coq-typeclasses-vs-dependent-records

Advice from Experts

In the process of preparing this chapter, we asked people on the `coq-club` mailing list for their best tips on preventing and debugging typeclass confusions, and on best practices for choosing between typeclasses and other large-scale structuring mechanisms such as modules and canonical structures. We received a number of generous replies, which we should eventually digest and incorporate into the material above. For the moment, they are recorded here essentially as posted (lightly edited for ease of reading).

Matthieu Sozeau

The fact that typeclass resolution resorts to unrestricted proof search is a blessing and a curse for sure. Errors tell you only that proof search failed and the initial problem while in general you need to look at the trace of resolution steps to figure out what's missing or, in the worst case, making search diverge. If you are writing obviously recursive instances, not mixing computation with the search (e.g. Unfolding happening in the indices necessary for instances to match), and not creating dependent subgoals then you're basically writing Haskell 98-style instances and should get the same "simplicity".

I think for more elaborate use cases (terms in indices, dependencies and computation), when encountering unexpected failure the debug output (Set Typeclasses Debug) is necessary. Testing the logic program, using Checks for example is a good way to explore the proof search results. One can also debug interactively by switching to the tactic mode and looking at the `typeclasses eauto` behavior. We're facing the same issues as logic programming and I don't know of a silver bullet to debug these programs.

For common issues a newcomer is likely to get, there are missing instances which can be prevented/tested using some coverage tests, divergence which can be understood by looking at the trace (usually it's because of a dangerous instance like transitivity or symmetry which has to be restricted or removed, and sometimes because of a conversion which makes an instance always applicable), and ambiguity when the user does not get the instance he expected (due to overlapping mainly, priorities can help here).

One advantage of modules (over typeclasses) is that everything is explicit but the abstraction cost a bit higher as you usually have to functorize whole modules instead of individual functions, and often write down signatures separate from the implementations. One rule of thumb is that if there are potentially multiple instances of the same interface for the same type/index then a module is preferable, but adding more indexes to distinguish the instances is also possible.

John Wiegley

One thing that always gets me is that overlapping instances are easy to write with no warning from Coq (unlike Haskell, which ensures that resolution always pick a single instance). This requires me to often use:

```
Typeclasses eauto := debug.
```

and switch to my *coq* buffer to see which lookup did not resolve to the instance I was expecting. This is usually fixed by one of two things:

- Change the "priority" of the overlapping instance (something we cannot do in Haskell).
- Change the Instance to a Definition — which I can still use it as an explicitly passed dictionary, but this removes it from resolution.

Another scenario that often bites me is when I define an instance for a type class, and then intend to write a function using the type class and forget to provide it as an argument. In Haskell this would be an error, but in Coq it just resolves to whatever the last globally defined instance was.

For example, say I write a function that uses a functor, but forget to mention the functor:

```
Definition foo (C D : Category) (x y : C) (f : x ~> y)
           : fobj x ~> fobj y :=
  fmap f.
```

In Haskell this gives an error stating that no Functor is available. In Coq, it type checks using the highest priority `C --> D` functor instance in scope. I typically discover that this has happened when I try to use `foo` and find the Functor to be too specific, or by turning on Printing All and looking at the definition of `foo`. However, there are times when `foo` is deep down in an expression, and then it's not obvious *at all* why it's failing.

The other way to solve this is to manually ensure there are no Instance definitions that might overlap, such as a generic `Instance` for `C --> D`, but only instances from specific categories to specific categories (though again, I might define several functors of that same type). It would be nice if I could make this situation into an error.

Finally, there is the dreaded "diamond problem", when referring to type classes as record members rather than type indices:

```
Class Foo := {
  method : Type → Type
}.

Class Bar := {
  bar_is_foo :> Foo
}.

Class Baz := {
  baz_is_foo :> Foo
}.

Class Oops := {
  oops_is_bar :> Bar
  oops_is_baz :> Baz
}.
```

Oops refers to two Foos, and I need explicit evidence to know when they are the same Foo. I work around this using indices:

```
Class Foo := {
  method : Type → Type
}.

Class Bar (F : Foo) := {
}.

Class Baz (F : Foo) := {
}.

Class Oops (F : Foo) := {
  oops_is_bar :> Bar F
  oops_is_baz :> Baz F
}.
```

Only those classes which might be multiply-inherited need to be lifted like this. It forces me to use Sections to avoid repetition, but allows Coq to see that base classes sharing is plainly evident.

The main gotcha here for the newcomer is that it is not obvious at all when the diamond problem is what you're facing, since when it hits the parameters are hidden indices, and you end up with goals like:

```
A, B : Type
F : Foo
O : Oops
H : @method (@bar_is_foo (@oops_is_bar O)) A = @method F B
--------------------
@method F A = @method F B
```

You can't apply here without simplying in H. However, what you see at first is:

```
A, B : Type
F : Foo
O : Oops
H : method A = method B
--------------------
method A = method B
```

As a newcomer, knowing to turn on Printing All is just not obvious here, but it quickly becomes something you learn to do whenever what looks obvious is not.

Other than these things, I use type classes heavily in my own libraries, and very much enjoy the facility they provide. I have a category-theory library that is nearly all type classes, and I ran into every one of the problems described above, before learning how to "work with Coq" to make things happy.

Michael Soegtrop

What I had most problems with initially is that some type classes have implicit parameters. This works quite well when the nesting level of these parameters is small, but when the nesting of parameters gets deeper one can have the issue that unification takes long, that error messages are hard to understand and that later in a proof one might require certain relations between different parameters of a type class which are not explicit when the initial resolution was done and that an instance is chosen which is not compatible with these requirements, although there is one which would be compatible. The solution is typically to explicitly specify some of the implicit parameters, especially their relation to each other. Another advantage of stating certain things explicitly is that it is easier to understand what actually happens.

Abhishek Anand

Typeclasses are merely about inferring some implicit arguments using proof search. The underlying modularity mechanism, which is the ability to define "existential types" using induction, was *always* there in Coq: typeclasses merely cuts down on verbosity because more arguments can now be implicit because they can be automatically inferred. Relying on proof search often brings predictability concerns. So, guidance on taming proof search would be very useful: Chapter 13 of Chipala's Certified Programming with Dependent Types (CPDT) might be a good background for using typeclasses. Also, it is good to keep in mind that if typeclass-resolution fails to instantiate an implicit argument, some/all of those arguments can be provided manually. Often, just providing one such implicit argument gives enough clues to the inference engine to infer all of them. I think it is important to emphasize that typeclass arguments are just implicit arguments.

Also, another design decision in typeclasses/records is whether to unbundle. The following paper makes a great case for unbundling: Spitters, Bas, and Eelis Van Der Weegen. "Type Classes for Mathematics in Type Theory." MSCS 21, no. Special Issue 04 (2011): 795–825. doi:10.1017/S0960129511000119. http://arxiv.org/pdf/1102.1323v$_1$.pdf.

I think the above paper is missing one argument for unbundling: I've seen many libraries that begin by making an interface (say I) that bundles *all* the operations (and their correctness properties) they will *ever* need and then *all* items in the library (say L) are parametrized by over I. A problem with this bundled approach is impossible to use *any* part of D if you are missing *any* operation (or proof of a logical property of the operation) in the interface I, even if parts of D don't actually need that operation: I've run into situations where it is impossible to cook up an operation that 90 percent of L doesn't use anyway. When using the unbundled approach, one can use Coq's Section mechanism to ensure that definitions/proofs only depend on items of the interface they actually use, and not on a big bundle.

Further Reading

On the origins of typeclasses in Haskell:

- How to make ad-hoc polymorphism less ad hoc Philip Wadler and Stephen Blott. 16'th Symposium on Principles of Programming Languages, ACM Press, Austin, Texas, January 1989. http://homepages.inf.ed.ac.uk/wadler/topics/type-classes.html

The original paper on typeclasses In Coq:

- Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. TPHOLs 2008. https://link.springer.com/chapter/10.1007%2F978-3-540-71067-7_23

Sources for this tutorial:

- Coq Reference Manual: https://coq.inria.fr/refman/Reference-Manual023.html
- Casteran and Sozeau's "Gentle Introduction": http://www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/typeclassestut.pdf
- Sozeau's slides from a talk at Penn: https://www.cis.upenn.edu/~bcpierce/courses/670Fall12/slides.pdf

Some of the many tutorials on typeclasses in Haskell:

- https://en.wikibooks.org/wiki/Haskell/Classes_and_types
- http://learnyouahaskell.com/types-and-typeclasses and http://learnyouahaskell.com/making-our-own-types-and-typeclasses