Type Classes in Coq

Elisabet Lobo-Vesga

EAFIT University

28th April, 2014

What is Coq? [1]

Coq is a proof assistant developed in France since 1989. It is based on an formal language called Calculus of Inductive Constructions (CIC). Coq allows to:

- Define functions or predicates
- State mathematical theorems
- Interactively develop formal proofs of these theorems
- Check these proofs
- Extract certified programs to languages like OCaml or Haskell
- Use a *tactic* language for letting the user define its own proof methods

[1] Coq website `http://coq.inria.fr/what-is-coq`

The Coq bundle[1]

- Arithmetics in $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{Q}$
- Libraries about list, finite sets, finite maps, etc.
- coqtop: interactive mode
- coqide: graphical user interface
- coqdoc and coq-tex: documentation tools
- coqc : the compiler (batch compilation)
- coqchk: stand-alone proof verifier (validation of compiled libraries)

Introduction to COQ [2]

Declarations

A declaration associates a *name* with a *specification*.

- Name: identifier
- Specification: formal expression as logical propositions (`Prop`), mathematical collections (`Set`) and abstract types (`Type`)

```
name  : sort

0     : nat
nat   : Set
Set   : Type
Prop  : Type
>     : nat → nat → Prop
list  : Type → Type
```

[2] Huet, G., Kahn, G. and Paulin-Mohring, C. (2007). The Coq Proof Assistant. A Tutorial.

Introduction to Coq [2]

Definitions

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.

Definition one              := (S O).
Definition two : nat        := S one.
Definition double (m:nat) := plus m m.
```

```
Variables A B C : Prop.

Lemma lem :
  (A -> B -> C) -> (A -> B) -> A -> C.

Proof.
intro  H.
intros H' HA.
apply  H.
exact  HA.
apply  H'.
assumption.
Qed.
```

HASKELL Type Classes

Definition

"Typeclasses define a set of functions that can have different implementations depending on the type of data they are given." [3]

[3] O'Sullivan, B., Goerzen, J. and Stewart, D. (2008). Real World Haskell. Chapter 6.

Parametric polymorphism

"Occurs when a function is defined over a range of types, acting in the **same** *way for each type."*[4]

Ad-Hoc polymorphism (Overloading)

"Occurs when a function is defined over several different types, acting in a **different** *way for each type."*[4]

[4] Walder, P. and Stephen, B. (1998). How to make *ad-hoc* polymorphism less *ad hoc*.

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b


data List a = [] | a : [a]

data Maybe a  =  Nothing | Just a
```

Haskell Type Classes

Implementation

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b


instance Functor List where
  fmap _ []     = []
  fmap f (x:xs) = f x : fmap f xs

instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just a) = Just (f a)
```

Type Classes in Coq [5]

Syntax of Class and Instance declarations

$$
\begin{array}{ll}
\texttt{\textbf{Class} Id } (\alpha_1{:}\tau_1) \cdots (\alpha_n{:}\tau_n)\ [{:}\,\textit{sort}]\ {:=}\ \{ & \\
\quad \texttt{f}_1 & {:}\ \textit{type}_{f1}; \\
\quad \vdots & \\
\quad \texttt{f}_m & {:}\ \textit{type}_{fm}\}.
\end{array}
$$

$$
\begin{array}{ll}
\texttt{\textbf{Instance} } \textit{ident}{:}\texttt{Id}\ \textit{term}_1 \cdots \textit{term}_n\ {:=}\ \{ & \\
\quad \texttt{f}_1 & {:=}\ \textit{term}_{f1}; \\
\quad \vdots & \\
\quad \texttt{f}_m & {:=}\ \textit{term}_{fm}\}.
\end{array}
$$

Where $\alpha_i{:}\tau_i$ are called **parameters** of the class and $f_k{:}type_k$ are called the **methods**.

[5] The Coq Development Team. Reference Manual - The Coq Proof Assistant - Inria (Version 8.4pl4). Chapter 19 Type Classes.

Type Classes in Coq[5]

Example of Class and Instance declarations

```
Class EqDec (A : Type) := {
  eqb        : A → A → bool ;
  eqb_prop:
    ∀ x y, eqb x y = true ⇒ x = y}.

Instance eq_bool : EqDec bool := {
  eqb x y := if x then y else negb y}.

Proof.
intros x y H.
destruct x ; destruct y ;
discriminate || reflexivity.
Qed.
```

Type Classes in Coq[5]

Using Type Classes

Binding classes

```
Definition neqb {A} {eqa : EqDec A}
(x y : A) := negb (eqb x y).
```

Superclasses

```
class (Eq a) => Ord a where
  le :: a -> a -> Bool
```

```
Class Ord A {E : EqDec A} := {
  le : A → A → bool}.
```

Type Classes in Coq [5]

Using Type Classes

Substructures

```
Definition neqb {A} {eqa : EqDec A}
  (x y : A) := negb (eqb x y).
```

Superclasses

```
class (Eq a) => Ord a where
  le :: a -> a -> Bool
```

```
Class Ord A {E : EqDec A} := {
  le : A → A → bool}.
```