First-Class Type Classes, in Coq

Matthieu Sozeau
Joint work with Nicolas Oury

LRI, Univ. Paris-Sud - Démons Team & INRIA Saclay - ProVal Project

ProVal Workgroup
March 3rd 2008
Orsay


Making ad-hoc polymorphism less *ad hoc*

```
class Eq A where
    (==) :: A → A → Bool

class Eq Bool where
    x == y = if x then y else not y
```

Making ad-hoc polymorphism less *ad hoc*

```
class Eq A where
    (==) :: A → A → Bool

class Eq Bool where
    x == y = if x then y else not y

in :: Eq A ⇒ A → [A] → Bool
 in x []       = True
 in x (y : ys) = x == y || in x ys
```

Parameterized instances

```
instance (Eq A) ⇒ Eq [A] where
  [] == []                 = True
  (x : xs) == (y : ys)     = x == y && xs == ys
  _ == _                   = False
```

Parameterized instances

```
instance (Eq A) ⇒ Eq [A] where
  [] == []                 = True
  (x : xs) == (y : ys)     = x == y && xs == ys
  _ == _                   = False

instance (Eq A) ⇒ Eq (Term A) where
  Var x == Var y           = x == y
  App f l == App f' l'     = f == f' && l == l'
  _ == _                   = False
```

A structuring concept

$$\texttt{class}\ \mathbf{Num}\ A\ \texttt{where}$$
$$\quad (+) :: A \to A \to A \ldots$$

$$\texttt{class}\ (\mathbf{Num}\ A) \Rightarrow \mathbf{Fractional}\ A\ \texttt{where}$$
$$\quad (/) :: A \to A \to A \ldots$$

$$\texttt{class}\ (\mathbf{Fractional}\ A) \Rightarrow \mathbf{Floating}\ A\ \texttt{where}$$
$$\quad \mathbf{exp} :: A \to A \ldots$$

Outline

1. Type Classes in Coq
   - A cheap implementation
   - Demo – Programming with monads
2. Superclasses and substructures
   - The power of Pi
   - Demo – Categories
3. Generalized rewriting with class

Motivations

- Overloading: in programs, specifications and proofs.

Motivations

- Overloading: in programs, specifications and proofs.
- Better than HASKELL? Proofs are part of classes, added guarantees.

`Class` **Eq** $A :=$
  eq : $A \rightarrow A \rightarrow$ **bool** ;
  equiv : **equivalence** A **eq**.

Motivations

- Overloading: in programs, specifications and proofs.
- Better than HASKELL? Proofs are part of classes, added guarantees.

  `Class` **Eq** $A :=$
  eq : $A \to A \to$ **bool** ;
  equiv : **equivalence** A **eq**.

- Extensions: dependent types give new power to type classes.

  `Class` **Reflexive** A (R : **relation** A) :=
  reflexive : $\forall\ x,\ R\ x\ x$.

Implementation

- Parameterized dependent records

$$\texttt{Class}\ \mathbf{Id}\ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=$$
$$\quad \mathbf{f}_1 : \phi_1\ ;\ \cdots\ ; \mathbf{f}_m : \phi_m.$$

Implementation

- Parameterized dependent records

$$\texttt{Record } \mathbf{Id}\ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=$$
$$\quad \mathbf{f}_1 : \phi_1\ ;\ \cdots\ ; \mathbf{f}_m : \phi_m.$$

Implementation

- Parameterized dependent records

  `Record` **Id** $(\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=$
  $\quad \mathbf{f}_1 : \phi_1 \; ; \cdots ; \mathbf{f}_m : \phi_m.$

  Instances are just definitions of type **Id** $\overrightarrow{t_n}$.

Implementation

- Parameterized dependent records

  $\texttt{Record}\ \mathbf{Id}\ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=$
  $\quad \mathbf{f}_1 : \phi_1\ ;\ \cdots\ ; \mathbf{f}_m : \phi_m.$

  Instances are just definitions of type $\mathbf{Id}\ \overrightarrow{t_n}$.

- Custom implicit arguments

  $\mathbf{f}_1 : \forall \overrightarrow{\alpha_n : \tau_n}, \mathbf{Id}\ \overrightarrow{\alpha_n} \rightarrow \phi_1$

Implementation

- Parameterized dependent records

  $\texttt{Record}$ **Id** $(\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=$
  $\quad \mathbf{f}_1 : \phi_1 \; ; \cdots ; \mathbf{f}_m : \phi_m.$

  Instances are just definitions of type **Id** $\overrightarrow{t_n}$.

- Custom implicit arguments

  $\mathbf{f}_1 : \forall \{\overrightarrow{\alpha_n : \tau_n}\}, \{\textbf{Id } \overrightarrow{\alpha_n}\} \rightarrow \phi_1$

Implementation

- Parameterized dependent records

  $\texttt{Record}\ \mathbf{Id}\ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) :=$
  $\quad \mathbf{f}_1 : \phi_1\ ;\ \cdots\ ; \mathbf{f}_m : \phi_m.$

  Instances are just definitions of type $\mathbf{Id}\ \overrightarrow{t_n}$.

- Custom implicit arguments

  $\mathbf{f}_1 : \forall \{\overrightarrow{\alpha_n : \tau_n}\}, \{\mathbf{Id}\ \overrightarrow{\alpha_n}\} \rightarrow \phi_1$

- Proof-search tactic with instances as lemmas

  eqa : $\mathbf{Eq}\ A \vdash \mathbf{Eq}\ (\mathbf{list}\ A)$

Elaboration with classes, an example

$(\lambda x\ y : \mathbf{bool}.\ \mathbf{eq}\ x\ y)$

Elaboration with classes, an example

$$(\lambda x\ y : \textbf{bool}.\ \textbf{eq}\ x\ y)$$

$$\rightsquigarrow \{ \text{ implicit arguments } \}$$

$$(\lambda x\ y : \textbf{bool}.\ \textbf{eq}\ (?_A : \texttt{Type})\ (?_{eq} : \textbf{Eq}\ ?_A)\ x\ y)$$

Elaboration with classes, an example

$$(\lambda x\ y : \textbf{bool}.\ \textbf{eq}\ x\ y)$$

$$\rightsquigarrow \{ \text{ implicit arguments } \}$$

$$(\lambda x\ y : \textbf{bool}.\ \textbf{eq}\ (?_A : \texttt{Type})\ (?_{eq} : \textbf{Eq}\ ?_A)\ x\ y)$$

$$\rightsquigarrow \{ \text{ unification } \}$$

$$(\lambda x\ y : \textbf{bool}.\ \textbf{eq bool}\ (?_{eq} : \textbf{Eq bool})\ x\ y)$$

Elaboration with classes, an example

$(\lambda x\ y : \textbf{bool}.\ \textbf{eq}\ x\ y)$

$\rightsquigarrow$ { implicit arguments }

$(\lambda x\ y : \textbf{bool}.\ \textbf{eq}\ (?_A : \texttt{Type})\ (?_{eq} : \textbf{Eq}\ ?_A)\ x\ y)$

$\rightsquigarrow$ { unification }

$(\lambda x\ y : \textbf{bool}.\ \textbf{eq bool}\ (?_{eq} : \textbf{Eq bool})\ x\ y)$

$\rightsquigarrow$ { proof search for **Eq bool** returns **Eq_bool** }

$(\lambda x\ y : \textbf{bool}.\ \textbf{eq bool Eq\_bool}\ x\ y)$

Outline

Outline

Fields or Parameters ?

When one doesn't have manifest types and `with` constraints...

`Class` **Functor** :=
  A : `Type`; **B** : `Type`; **src** : **Category A** ; **dst** : **Category B** ; ...

or

`Class` **Functor** *obj obj'* :=
  src : **Category** *obj* ; **dst** : **Category** *obj'* ; ...

or

`Class` **Functor** *obj* (**Category** *obj*) *obj'* (**Category** *obj'*) := ...

???

Sharing by equalities

$$\texttt{def}\ \textbf{adjunction}\ (F : \textbf{Functor})\ (G : \textbf{Functor}),$$
$$\textbf{src}\ F = \textbf{dst}\ G \rightarrow \textbf{dst}\ F = \textbf{src}\ G\ \ldots$$

Obfuscates the goals and the computations, awkward to use.

Sharing parameters

`Class` [**Category** *obj*, **Category** *obj'*] $\Rightarrow$ **Functor**

$\equiv$

`Record` **Functor** *obj* (_ : **Category** *obj*) *obj'* (_ : **Category** *obj'*)

Sharing parameters

`Class` [**Category** *obj*, **Category** *obj'*] $\Rightarrow$ **Functor**
$\equiv$
`Record` **Functor** *obj* (_ : **Category** *obj*) *obj'* (_ : **Category** *obj'*)

`def` **adjunction**
  [*C* : **Category** *obj*, *D* : **Category** *obj'*,
    F : **Functor** *obj C obj' D*,
    G : **Functor** *obj' D obj C*] := ...

Uses the dependent product and named instances.

Superclasses and substructures

A *superclass* becomes a parameter, a *substructure* is a method which is also an instance.

Example

`class` **Monoid** $A :=$
  monop : $A \to A \to A$; ...

`class` **Group** $A :=$
  monoid :> **Monoid** A ; ...

`def` **foo** [**Group** A] (x : A) : A := **monop** x x.

Remark Similar to the existing `Structures` based on coercive subtyping.

Outline

Outline

Respect and the **Morphism** type class

```
def respectful A (R : relation A) B (R' : relation B) :
  relation (A → B) :=
  fun f g ⇒ ∀ x y : A, R x y → R' (f x) (g y).

Notation " R ==> R' " :=
  (respectful R R') (right associativity).
Notation " R --> R' " := (respectful (inverse R) R').

Class Morphism A (R : relation A) (m : A) :=
  respect : R m m.
```

Some morphism instances

- "Usual" morphisms:

```
Instance Morphism (Prop → Prop → Prop)
  (iff ==> iff ==> iff) iff.
```

Some morphism instances

- "Usual" morphisms:

  `Instance` **Morphism** (`Prop` → `Prop` → `Prop`)
    (**iff** ==> **iff** ==> **iff**) **iff**.

- Abstract morphism instances:

  `Instance` [**Transitive** *A* *R*] ⇒
    ? **Morphism** (*R* −− > **eq** ==> **impl**) *R*.

- Morphisms for partial applications.

Setoid rewriting algorithm

- Build a set of constraints of the form **Morphism** $(?_R ==> ?_{R'})$ *m* by structural recursion over the term. Return a term built from various instances of **respect** as a proof that rewriting is possible.
  - At rewrite locations, $?_R$ is instantiated.
  - At the top, we want a **Morphism** $(?_R ==>$ **inverse impl**$)$.
- Solve the constraints using type class constraint solving/proof search. Voilà !

Summary

- Supports arbitrary relations
- Support for subrelations, just by adding the proper **Morphism** instances
- Support for reasonning on abstract structures for free using the type class mechanism. Just add a **Morphism** substructure and you're done.
- Support for the old syntax (with Nicolas Tabareau, WIP). Part of the standard library ported (Numbers, Ring done).
- Roughly the same performance without optimization.

Conclusions

- ✓ An implementation of type classes.
- ✓ Useful for programming, proving and specifying.
- ✓ A structuring concept for users and tactic implementers.