Coq with Classes
Matthieu Sozeau

IMDEA Seminar
October 20th 2011
Madrid, Spain
This talk

1. A quick overview of Coq and elaboration
2. Type Classes
In the design spaces of DTPs and ITPs...

- Full-spectrum dependent types
  - Single, unified term-type language, SN
  - Phase distinction issues (for runtime, see Brady, Barras)
- Core language design:
  - De Bruijn principle ("small" core, externally checkable terms)
  - Striving for minimality/purity and "accessibility" of models
  - Open-world, generative. Powerful module system
- External language design:
  - Unification is central (implicits, tactics) and incomplete
  - Definitional coercion systems for accessibility of the language
In the design spaces of DTPs and ITPs...

- Proof language design:
  - Separate tactic language $\mathcal{L}_{\text{tac}}$.
  - Proving tools: proof search, tactics.
  - Development tools: derived definitions (\texttt{FUNCTION}, Schemes...).

- User interface and interaction: not discussed here.
Elaboration: compiling high-level constructs to the core language, using the metalanguage.

✔ Advantages: metatheory done once and for all (just kidding!). Freedom in the transformations, extensibility and modularity.

✗ Concerns: “abstraction leaks”, efficiency, correctness.

Compare with:

► Reflexive methods: less freedom, more assurance, full correctness, smaller scope (but see Epigram 2).

► “Axiomatic” methods, e.g. Agda’s built-in pattern-matching. Less assurance, more freedom.

Acknowledgment McBride and McKinna’s work (OLEG, Epigram), KISS.
Defining functions with:

- Rich types while separating algorithms and proofs.
- Generic types, passing information implicitly.
- Rich data and control flow, keeping information transparently.
- Complex recursion behaviors and efficient evaluation.
- Support for reasoning after the fact: elimination principles and proof tools (search, rewriting).
Program

- Programming with subset types/refinement types
- Well-founded recursion

Thesis We can program as usual and still use rich types

Program Fixpoint div (a : nat) (b : nat | b ≠ 0) { wf lt a } : { (q, r) : nat × nat | a = b × q + r ∧ r < b } :=
if less_than a b then (O, a)
else
  let '(q', r) := div (a - b) b in
  (S q', r).
Derive Subterm for \texttt{vector}.

\textbf{Equations} \texttt{unzip \{A B n\} (v : vector (A \times B) n) : vector A n \times vector B n :=}

\texttt{unzip A B n v by rec v :=}

\texttt{unzip A B \?(O) Vnil := (Vnil, Vnil) ;}

\texttt{unzip A B \?(S n) (Vcons (pair x y) n v) with unzip v := \{}

\texttt{\quad (pair xs ys) := (Vcons x xs, Vcons y ys) \}.}
A brief tour of **Coq, Program and Equations**

1. **Program**
2. **Equations**

2. **Type Classes**
   - Type Classes from **Haskell**
   - Type Classes in **Coq**

3. **Conclusion**
Solutions for overloading

- **Intersection types**: closed overloading by declaring multiple signatures for a single constant (e.g. CDuce, Stardust).

- **Bounded quantification** and **class-based** overloading. Overloading circumscribed by a subtyping relation (e.g. structural subtyping à la OCaml).
Solutions for overloading

- **Intersection types**: closed overloading by declaring multiple signatures for a single constant (e.g. C\textsc{duce}, S\textsc{tardust}).

- **Bounded quantification** and **class-based** overloading. Overloading circumscribed by a subtyping relation (e.g. structural subtyping à la O\textsc{caml}).

**Context:**

- **Modularity**: separate definitions of the specializations.

- **Constrained by C\textsc{oq}**: a fixed kernel language!
Solutions for overloading

- **Intersection types**: closed overloading by declaring multiple signatures for a single constant (e.g. CDuce, Stardust).

- **Bounded quantification and class-based overloading**. Overloading circumscribed by a subtyping relation (e.g. structural subtyping à la OCaml).

Context:
- **Modularity**: separate definitions of the specializations.
- **Constrained by CoQ**: a fixed kernel language!

Solution:

Elaborate Type Classes, a kind of bounded quantification where the subtyping relation needs not be internalized.
Making *ad-hoc* polymorphism less *ad hoc*

In **Haskell**, Wadler & Blott, POPL’89.  
Also in **Isabelle**, Nipkow & Snelting, FPCA’91.

```haskell
class Eq a where
    (==) :: a → a → Bool

instance Eq Bool where
    x == y = if x then y else not y
```
Making *ad-hoc* polymorphism less *ad hoc*

In Haskell, Wadler & Blott, POPL’89. Also in Isabelle, Nipkow & Snelting, FPCA’91.

```haskell
class Eq a where
  (==) :: a → a → Bool

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

in :: Eq a ⇒ a → [a] → Bool
in x [] = False
in x (y : ys) = x == y || in x ys
```
instance (Eq a) ⇒ Eq [a] where

  []  ==  []         =  True

  (x : xs)  ==  (y : ys)  =  x  ==  y  &&  xs  ==  ys

  _  ==  _          =  False
instance \((\text{Eq } a) \Rightarrow \text{Eq } [a]\) where
\[
\begin{align*}
[] &= [] & = \text{True} \\
(x : xs) &= (y : ys) & = x == y \&\& xs == ys \\
_ &= _ & = \text{False}
\end{align*}
\]

class \text{Num } a \text{ where}
\((+) :: a \rightarrow a \rightarrow a \ldots\)

class \((\text{Num } a) \Rightarrow \text{Fractional } a \text{ where}
\((/) :: a \rightarrow a \rightarrow a \ldots\)

Type Classes

1. A brief tour of **Coq**, **Program** and **Equations**
   - **Program**
   - **Equations**

2. Type Classes
   - Type Classes from **Haskell**
   - Type Classes in **Coq**

3. Conclusion
Motivations

- Overloading in programs, specifications and proofs.

Motivations

- Overloading in programs, specifications and proofs.

Class Eq $A \equiv \{\text{eqb} : A \to A \to \text{bool}; \quad \text{eq} = \text{eqb} : \forall x y : A, x = y \leftrightarrow \text{eqb} x y = \text{true}\}$.

Class Reflexive $A (R : \text{relation } A) \equiv \text{reflexive} : \forall x, R x x$.
Motivations

- Overloading in programs, specifications and proofs.
- A safer Haskell: Proofs are part of instances.

```
Class Eq A := {
  eqb : A → A → bool ;
  eq_eqb : ∀ x y : A, x = y ↔ eqb x y = true }.
```
Motivations

- Overloading in programs, specifications and proofs.
- A safer Haskell Proofs are part of instances.

\[
\text{Class Eq } A := \{ \\
\quad \text{eqb : } A \rightarrow A \rightarrow \text{bool} ; \\
\quad \text{eq_eqb : } \forall x, y : A, x = y \iff \text{eqb } x \ y = \text{true} \}.
\]

- Extension Dependent types give new power to type classes.

\[
\text{Class Reflexive } A \ (R : \text{relation } A) := \\
\quad \text{reflexive : } \forall x, R x x.
\]
A cheap implementation

- Parametrized dependent records

\[
\text{Class } \text{Id} \ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \\
\{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}.
\]
A cheap implementation

- Parametrized dependent records

\[
\text{Record } \text{Id} \ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \\
\{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}.
\]
Parametrized dependent records

\[
\text{Record } \text{ld} \ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \\
\{ \! f_1 : \phi_1 ; \cdots ; f_m : \phi_m \! \}.
\]

Instances are just definitions of type \( \text{ld} \overset{\cdot}{\rightarrow} t_n \).
A cheap implementation

- Parametrized dependent records

\[
\text{Record } \text{ld} \ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \\
\{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}.
\]
Instances are just definitions of type \( \text{ld} \ t_n \).

- Custom implicit arguments of projections

\[
f_1 : \forall \ x : \alpha_n \rightarrow \text{ld} \ x \rightarrow \phi_1
\]
A cheap implementation

- Parametrized dependent records

\[
\text{Record Id } (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \\
\{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}.
\]

Instances are just definitions of type \( \text{Id } t_n \).

- Custom implicit arguments of projections

\[
f_1 : \forall \{ \alpha_n : \tau_n \}, \{ \text{Id } \alpha_n \} \rightarrow \phi_1
\]
Elaboration with classes, an example

\((\lambda x \ y : \text{bool}. \ \text{eqb} \ x \ y)\)
Elaboration with classes, an example

\[(\lambda x \ y : \textit{bool}. \ \textit{eqb} \ x \ y)\]
\[
\leadsto \ {\{ \text{Implicit arguments} \} \}
\[
(\lambda x \ y : \textit{bool}. \ \texttt{@eqb} \ (?_A : \textit{Type}) \ (?_\textit{eq} : \textit{Eq} \ ?_A) \ x \ y)
\]
Elaboration with classes, an example

\[(\lambda x\ y : \text{bool}. \ eqb\ x\ y)\]

\[\leadsto \{\ \text{Implicit arguments} \}\]

\[(\lambda x\ y : \text{bool}. \ ?eqb\ (?_A : \text{Type})\ (?eq : \text{Eq} \ ?_A)\ x\ y)\]

\[\leadsto \{\ \text{Unification} \}\]

\[(\lambda x\ y : \text{bool}. \ ?eqb\ \text{bool}\ (?eq : \text{Eq} \ \text{bool})\ x\ y)\]
Elaboration with classes, an example

\((\lambda x\ y:\ bool.\ eqb\ x\ y)\)

\(\leadsto\ \{\ \text{Implicit arguments}\ \}\)

\((\lambda x\ y:\ bool.\ \text{@eqb}\ (?_A:\ Type)\ (?_{eq}\ :\ Eq\ ?_A)\ x\ y)\)

\(\leadsto\ \{\ \text{Unification}\ \}\)

\((\lambda x\ y:\ bool.\ \text{@eqb}\ bool\ (?_{eq}\ :\ Eq\ bool)\ x\ y)\)

\(\leadsto\ \{\ \text{Proof search for Eq bool returns Eq_bool}\ \}\)

\((\lambda x\ y:\ bool.\ \text{@eqb}\ bool\ Eq\_bool\ x\ y)\)
Type Class resolution

Proof-search tactic with instances as lemmas:

\[ A : \text{Type}, \ eqa : \ \text{Eq} \ A \vdash ? : \ \text{Eq} \ (\text{list} \ A) \]

- Simple depth-first search with higher-order unification
  - Returns the first solution only
- Extensible through \( L_{\text{tac}} \)
Numeric overloading

Class \textbf{Num} \( \alpha \) := \{ \text{zero} : \alpha ; \text{one} : \alpha ; \text{plus} : \alpha \to \alpha \to \alpha \}.
Class \( \text{Num} \ \alpha \) := \{ \text{zero} : \alpha ; \text{one} : \alpha ; \text{plus} : \alpha \to \alpha \to \alpha \}.

Instance \( \text{nat}_\text{num} \) : \( \text{Num} \ \text{nat} \) :=
\{ \text{zero} := 0\% \text{nat} ; \text{one} := 1\% \text{nat} ; \text{plus} := \text{Peano.plus} \}.

Instance \( \text{Z}_\text{num} \) : \( \text{Num} \ \text{Z} \) :=
\{ \text{zero} := 0\% \text{Z} ; \text{one} := 1\% \text{Z} ; \text{plus} := \text{Zplus} \}.

Check (\( \lambda x : \text{nat} \), \( x + (1 + 0 + x) \)).

Check (\( \lambda x : \text{Z} \), \( x + (1 + 0 + x) \)).

(* Defaulting *)

\( \text{Check} (\lambda x, x + 1) \).
Class \texttt{Num} $\alpha := \{ \text{zero} : \alpha ; \text{one} : \alpha ; \text{plus} : \alpha \to \alpha \to \alpha \}$. 

Instance \texttt{nat_num} : \texttt{Num} \texttt{nat} :=
\{ zero := 0\texttt{nat} ; one := 1\texttt{nat} ; plus := \texttt{Peano.plus} \}.

Instance \texttt{Z_num} : \texttt{Num} \texttt{Z} :=
\{ zero := 0\texttt{Z} ; one := 1\texttt{Z} ; plus := \texttt{Zplus} \}.

Notation "0" := zero.
Notation "1" := one.
Infix "+" := plus.
Numeric overloading

Class \textbf{Num} $\alpha := \{ \text{zero : } \alpha ; \text{one : } \alpha ; \text{plus : } \alpha \rightarrow \alpha \rightarrow \alpha \}$.

\textbf{Instance} \texttt{nat
num} : Num \texttt{nat} :=
\{ zero := 0\%nat ; one := 1\%nat ; plus := \texttt{Peano.plus} \}.

\textbf{Instance} \texttt{Z
num} : Num \texttt{Z} :=
\{ zero := 0\%Z ; one := 1\%Z ; plus := \texttt{Zplus} \}.

\textbf{Notation} "0" := zero.
\textbf{Notation} "1" := one.
\textbf{Infix} "+" := plus.

Check ($\lambda x : \texttt{nat}, x + (1 + 0 + x)$).
Check ($\lambda x : \texttt{Z}, x + (1 + 0 + x)$).
Numeric overloading

Class $\text{Num } \alpha := \{ \text{zero : } \alpha ; \text{one : } \alpha ; \text{plus : } \alpha \rightarrow \alpha \rightarrow \alpha \}$.  

Instance $\text{nat}_\text{num} : \text{Num } \text{nat} := \{ \text{zero := 0\%nat ; one := 1\%nat ; plus := Peano.plus } \}$.  

Instance $\text{Z}_\text{num} : \text{Num } \text{Z} := \{ \text{zero := 0\%Z ; one := 1\%Z ; plus := Zplus } \}$.  

Notation "0" := zero.  
Notation "1" := one.  
Infix "+" := plus.  

Check $(\lambda x : \text{nat}, x + (1 + 0 + x))$.  
Check $(\lambda x : \text{Z}, x + (1 + 0 + x))$.  

(* Defaulting *)  
Check $(\lambda x, x + 1)$.  

Matthieu Sozeau - Coq with Classes
Class Reflexive \( \{ A \} \) (\( R : \text{relation} \ A \) ) :=

\[ \text{refl} : \forall x, R x x. \]
Class Reflexive \{ A \} (R : \text{relation } A) :=
    refl : \forall x, R x x.

Instance eq_refl A : Reflexive (@eq A) := @refl_equal A.
Instance iff_refl : Reflexive iff.
Proof. red. tauto. Qed.
Dependent classes

Class Reflexive \( \{A\} \) (\( R : \text{relation} \ A \) :=

\[
\text{refl} : \forall x, R \ x \ x.
\]

Instance \( \text{eq_refl} \ A : \text{Reflexive} \ (\#eq \ A) := @\text{refl_equal} \ A. \)

Instance \( \text{iff_refl} : \text{Reflexive} \ \text{iff} \).

Proof. \text{red. tauto. Qed.} \)

Goal \( \forall P, P \leftrightarrow P. \)

Proof. \text{apply refl. Qed.} \)

Goal \( \forall A (x : A), x = x. \)

Proof. \text{intros} A ; \text{apply refl. Qed.} \)
Dependent classes

Class Reflexive \( \{A\} \) \( (R : \text{relation } A) \) :=
\[
\text{refl} : \forall x, R \ x \ x.
\]

Instance eq_refl \( A : \text{Reflexive } (@eq \ A) \) := @refl_equal \( A \).

Instance iff_refl : Reflexive iff.
Proof. red. tauto. Qed.

Goal \( \forall P, P \leftrightarrow P \).
Proof. apply refl. Qed.

Goal \( \forall A \ (x : A), x = x \).
Proof. intros \( A \); apply refl. Qed.

Ltac reflexivity’ := apply refl.

Lemma foo ‘\( \{\text{Reflexive nat } R\} : R \ 0 \ 0 \).
Proof. intros. reflexivity’. Qed.
Building hierarchies of classes:

\[
\text{Class Fractional} \left\{ \text{Num} \ \alpha \right\} := \\
\{ \text{div} : \alpha \rightarrow \{ y : \alpha \mid y \neq 0 \} \rightarrow \alpha \}.
\]

\[
\text{Class Equivalence} \ \alpha := \\
\{ \text{equiv_refl} : \rightarrow \text{Reflexive} \ \alpha ; \ \\
\text{equiv_sym} : \rightarrow \text{Symmetric} \ \alpha ; \ \\
\text{equiv_trans} : \rightarrow \text{Transitive} \ \alpha \}
\]

+ Special support for binding super-classes

Tried and tested by P. Letouzey, S. Lescuyer on FSets (JFLA’10), B. Spitters and E. van der Weegen (ITP’10)…
Related work

Type Classes implementations:

- In **Haskell** by Wadler *et al.* (POPL’89, FO, second class)
- In **Isabelle** by Nipkow *et al.* (POPL’93, same)
- In **Agda** by Devriese and Pieperssens (ICFP’11, non-recursive proof search)

In **Coq** and **Matita**:

- Coercive Subtyping and **Canonical Structures** (Saïbi, POPL’97). Used by Gonthier *et al.* (TPHOLs’09), Nanevski *et al.* (ICFP’11).
- Unification hints, a more general framework studied by Asperti *et al.* (TPHOLs’09).
Experiments and formalizations in Coq

- Sets, Maps etc... (Letouzey, Lescuyer . . . )
- Domain theory, probability monad (Paulin, . . . )
- Generalized rewriting (Sozeau, JFR’09)
- ACI rewriting (Braibant & Pous, ITP’11)
- Universal algebra, category theory and computable reals (Spitters et al., ITP’10)
Current issues and perspectives

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification
Hope These are all researched in the logic programming community

▶ Undeterministic proof-search
Current issues and perspectives

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- Undeterministic proof-search
  ⇒ Determinacy inference (Kriener and King, ICLP'11)
Proof search efficiency and control issues...

**Prerequisite** Proper formalization of unification

*Hope* These are all researched in the logic programming community

- Undeterministic proof-search
  - $\Rightarrow$ Determinacy inference (*Kriener* and *King*, ICLP’11)
- No forward reasoning or reordering of constraints
Proof search efficiency and control issues...

**Prerequisite** Proper formalization of unification

**Hope** These are all researched in the logic programming community

- Undeterministic proof-search
  \[ \Rightarrow \text{Determinacy inference (KRIENER and KING, ICLP'11) } \]

- No forward reasoning or reordering of constraints
  \[ \Rightarrow \text{Mode analysis (à la PROLOG, TWELF) } \]
Current issues and perspectives

Proof search efficiency and control issues...

**Prerequisite** Proper formalization of unification

**Hope** These are all researched in the logic programming community

- Undeterministic proof-search
  ⇒ Determinacy inference *(Kriener and King, ICLP’11)*

- No forward reasoning or reordering of constraints
  ⇒ Mode analysis *(à la Prolog, Twelf)*

- Risk of non-termination
Current issues and perspectives

Proof search efficiency and control issues...

**Prerequisite** Proper formalization of unification

**Hope** These are all researched in the logic programming community

- Undeterministic proof-search
  - $\Rightarrow$ Determinacy inference (Kriener and King, ICLP’11)

- No forward reasoning or reordering of constraints
  - $\Rightarrow$ Mode analysis (à la Prolog, Twelf)

- Risk of non-termination
  - $\Rightarrow$ Termination analysis, requires modes
Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- Undeterministic proof-search
  ⇒ Determinacy inference (Kriener and King, ICLP’11)
- No forward reasoning or reordering of constraints
  ⇒ Mode analysis (à la Prolog, Twelf)
- Risk of non-termination
  ⇒ Termination analysis, requires modes
- Little sharing and intelligence in the proof-search
Proof search efficiency and control issues...

**Prerequisite** Proper formalization of unification

**Hope** These are all researched in the logic programming community

- Undeterministic proof-search
  \[ \Rightarrow \text{Determinacy inference (Kriener and King, ICLP'11)} \]
- No forward reasoning or reordering of constraints
  \[ \Rightarrow \text{Mode analysis (à la Prolog, Twelf)} \]
- Risk of non-termination
  \[ \Rightarrow \text{Termination analysis, requires modes} \]
- Little sharing and intelligence in the proof-search
  \[ \Rightarrow \text{Focusing, strategies.} \]
Proof search efficiency and control issues...

**Prerequisite** Proper formalization of unification

**Hope** These are all researched in the logic programming community

- Undeterministic proof-search
  - $\Rightarrow$ Determinacy inference (Kriener and King, ICLP’11)

- No forward reasoning or reordering of constraints
  - $\Rightarrow$ Mode analysis (à la Prolog, Twelf)

- Risk of non-termination
  - $\Rightarrow$ Termination analysis, requires modes

- Little sharing and intelligence in the proof-search
  - $\Rightarrow$ Focusing, strategies.

- Scoping of instances... through modules only.
✓ A lightweight and general implementation of type classes, available in Coq v8.2.

✓ A type-theoretic explanation and extension of type classes concepts (TPHOLs’08, with Nicolas Oury).

Success of the elaboration point-of-view!

✓ Progress in accessibility and scalability of the tool.

✗ Youth! Efficiency and controllability concerns.
Coq with Classes

Matthieu Sozeau
INRIA Paris & PPS, Paris 7 University