Coq with Classes
Matthieu Sozeau

PLAS Seminar
November 7th 2011
Canterbury, UK
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 (\textsc{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.
Our focus

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).
Programming with subset types/refinement types

Well-founded recursion

Thesis We can program as usual and still use rich types

Program Fixpoint div \( (a : \text{nat}) (b : \text{nat} \mid b \neq 0) \{ \text{wf} \ \text{lt} \ a \} : \{(q, r) : \text{nat} \times \text{nat} \mid a = b \times q + r \land r < b \} := \)
if less_than \( a \ b \) then \((O, a)\)
else
let \( '(q', r) := \text{div} \ (a - b) \ b \) in
\((S \ q', r)\).
True dependent pattern-matching
Recursion on inductive families
Reasoning on function definitions

Derive Subterm for `vector`.

Equations `unzip \{ A B n \} (v : vector (A \times B) n)`

: `vector A n \times vector B n :=`

unzip `A B n v` by `rec v :=`

unzip `A B ?(O) Vnil := (Vnil, Vnil) ;`

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

| `(pair xs ys) := (Vcons x xs, Vcons y ys) \}`.
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
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. **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!
Solutions for overloading

- **Intersection types**: closed overloading by declaring multiple signatures for a single constant (e.g. **CDue**, **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.
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
```
Parametrized instances and super-classes

\[
\text{instance } (\text{Eq } a) \Rightarrow \text{Eq } [a] \text{ where }
\]
\[
\begin{align*}
\text{} \hspace{1cm} & [] == [] & = \text{True} \\
\text{} \hspace{1cm} & (x : xs) == (y : ys) & = x == y && xs == ys \\
\text{} \hspace{1cm} & _ == _ & = \text{False}
\end{align*}
\]
instance (Eq a) ⇒ Eq [a] where
    [] == [] = True
    (x : xs) == (y : ys) = x == y && xs == ys
    _ == _ = False

class Num a where
    (+) :: a → a → a 

class (Num a) ⇒ Fractional a where
    (/) :: a → a → a 

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

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

3. Conclusion
▶ Overloading in programs, specifications and proofs.

Class \( \text{Eq} \) \( A \) :=
\[
\text{eqb} : A \to A \to \text{bool};
\text{eq} \text{eqb} : \forall x y : A, x = y \leftrightarrow \text{eqb} x y = \text{true}.
\]

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

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

- **Extension** Dependent types give new power to type classes.

```haskell
Class Reflexive A (R : relation A) :=
    reflexive : ∀ x, R x x.
```
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

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

- Parametrized dependent records

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

Instances are just definitions of type \( \text{Id} \xrightarrow{} t_n \).
A cheap implementation

- Parametrized dependent records

\begin{align*}
\text{Record } \text{Id} \ (\alpha_1 : \tau_1) \cdots (\alpha_n : \tau_n) := \\
\{ f_1 : \phi_1 ; \cdots ; f_m : \phi_m \}.
\end{align*}

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

- Custom implicit arguments of projections

\[ f_1 : \forall \ \alpha_n : \tau_n, \ \text{Id} \ \alpha_n \rightarrow \phi_1 \]
Parametrized dependent records

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

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

Custom implicit arguments of projections

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

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

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

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

\[(\lambda x \ y : \text{bool}. \ \text{eqb} \ (?_A : \text{Type}) \ (\_eq : \text{Eq} \ ?_A) \times y)\]
\[\rightsquigarrow \{ \text{Unification} \}\]

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

\((\lambda x \ y : \text{bool. } \text{eqb } x \ y)\)
\(\leadsto \{ \text{Implicit arguments} \}\)
\((\lambda x \ y : \text{bool. } \text{eqb } (?_A : \text{Type}) (?_eq : \text{Eq } ?_A) \times y)\)
\(\leadsto \{ \text{Unification} \}\)
\((\lambda x \ y : \text{bool. } \text{eqb bool } (?_eq : \text{Eq bool}) \times y)\)
\(\leadsto \{ \text{Proof search for Eq bool returns Eq.bool} \}\)
\((\lambda x \ y : \text{bool. } \text{eqb bool Eq.bool } x \ y)\)
Proof-search tactic with instances as lemmas:

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

- Simple depth-first search with higher-order unification
  - Returns the first solution only
+ Extensible through \( \mathcal{L}_{\text{tac}} \)
Class Num $\alpha := \{ \text{zero} : \alpha ; \text{one} : \alpha ; \text{plus} : \alpha \rightarrow \alpha \rightarrow \alpha \}$. 
Numeric overloading

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

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

Instance Z_num : Num Z :=
\{ zero := 0\%Z ; one := 1\%Z ; plus := Zplus \}.
Class $\text{Num} \, \alpha := \{ \text{zero} : \alpha ; \text{one} : \alpha ; \text{plus} : \alpha \to \alpha \to \alpha \}$.

Instance $\text{nat\_num} : \text{Num} \, \text{nat} :=$
\{ \text{zero} := 0\%\text{nat} ; \text{one} := 1\%\text{nat} ; \text{plus} := \text{Peano\_plus} \}.

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

Notation "$0" := \text{zero}.$

Notation "$1" := \text{one}.$

Infix "$+" := \text{plus}.$
Class \( \text{Num} \ \alpha := \{ \text{zero} : \alpha ; \text{one} : \alpha ; \text{plus} : \alpha \rightarrow \alpha \rightarrow \alpha \} \).

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

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

Notation \( "0" := \text{zero} \).
Notation \( "1" := \text{one} \).
Infix \( "+" := \text{plus} \).

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

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

Instance **nat_num** : Num nat :=

\[
\{ \text{zero} := 0\%\text{nat} ; \text{one} := 1\%\text{nat} ; \text{plus} := \text{Peano}.\text{plus} \}.
\]

Instance **Z_num** : Num Z :=

\[
\{ \text{zero} := 0\%\text{Z} ; \text{one} := 1\%\text{Z} ; \text{plus} := \text{Zplus} \}.
\]

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 \((\lambda x, x + 1)\).
Class Reflexive \{A\} (R : relation A) :=
refl : \forall x, R x x.
Dependent classes

Class Reflexive \{ A \} (R : 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.

Ltac reflexivity' := apply refl.
Lemma foo' {Reflexive nat R favorite} : R 0 0.
Proof. intros. reflexivity'. Qed.
Class Reflexive \{A\} (R : 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.

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

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

Class Reflexive \{ A \} (R : 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.

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 '{\{ Reflexive nat R \} : R 0 0.
Proof. intros. reflexivity'. Qed.
A structuring tool: super-classes and substructures

Building hierarchies of classes:

```coq
Class Fractional \{\text{Num } \alpha\} :=
\{ \text{div} : \alpha \rightarrow \{ y : \alpha \mid y \neq 0 \} \rightarrow \alpha \}.
```

```coq
Class Equivalence \alpha :=
\{ \text{equiv_refl} : \text{Reflexive } \alpha ;
   \text{equiv_sym} : \text{Symmetric } \alpha ;
   \text{equiv_trans} : \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 Piessens (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)
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 (**Krien**er 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
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**)

**Matthieu Sozeau - Coq with Classes**
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
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
  \[\Rightarrow\] 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
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
  ⇒ Termination analysis, requires modes

- Little sharing and intelligence in the proof-search
  ⇒ 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  
  ⇒ 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  
  ⇒ 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

PLAS Seminar
November 7th 2011
Canterbury, UK