Coq with Classes
Matthieu Sozeau
This talk

- A quick overview of Coq
- Elaboration
- 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 minimalty/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
Proof language design:

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

User interface and interaction: not discussed here.
Our angle

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 \( \text{div} (a : \text{nat}) (b : \text{nat} \mid b \neq 0) \{ \text{wf} \ \text{lt} \ a \} : \{ (q, r) : \text{nat} \times \text{nat} \mid a = b \times q + r \land r < b \} := \)

\[
\begin{align*}
\text{if less\_than a b then } & (O, a) \\
\text{else } & \text{let } '(q', r) := \text{div} (a - b) b \text{ in} \\
& (S q', r).
\end{align*}
\]
- 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)
}.
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. \textsc{CDuce}, \textsc{Stardust}).
- **Bounded quantification** and **class-based** overloading. Overloading circumscribed by a subtyping relation (e.g. structural subtyping à la \textsc{OCaml}).

**Context:**

- **Modularity**: separate definitions of the specializations.
- **Constrained by \textsc{Coq}**: a fixed kernel language!

**Solution:**

\textbf{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 (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 ...
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
Overloading in programs, specifications and proofs.
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 }.
```
Motivations

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

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

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

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

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

Instances are just definitions of type \texttt{Id} $\overrightarrow{t_n}$. 
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 \( \text{ld} \ x_1 \to t_1 \).

- Custom implicit arguments of projections

\[
f_1 : \forall \quad \alpha : \tau_1 , \quad \text{ld} \ x_1 \to \phi_1
\]
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 \(\text{id} \rightarrow t_n\).

- Custom implicit arguments of projections
  
  \[f_1 : \forall \{\alpha_n : \tau_n\}, \{\text{id} \rightarrow \alpha_n\} \rightarrow \phi_1\]
Elaboration with classes, an example

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

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

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

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

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

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

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

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

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

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

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

\[(\lambda x \ y : \text{bool}. \ @\text{eqb} (A : \text{Type}) (\text{eq} : \text{Eq} A) \times y)\]

\[\rightsquigarrow \ \{ \text{Unification} \} \]

\[(\lambda x \ y : \text{bool}. \ @\text{eqb} \text{bool} (\text{eq} : \text{Eq} \text{bool}) \times y)\]

\[\rightsquigarrow \ \{ \text{Proof search for Eq bool returns Eq.bool} \} \]

\[(\lambda x \ y : \text{bool}. \ @\text{eqb} \text{bool} \text{Eq.bool} \times y)\]
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 \( L_{\text{tac}} \)
Class Num $\alpha := \{ \text{zero}: \alpha; \text{one}: \alpha; \text{plus}: \alpha \rightarrow \alpha \rightarrow \alpha \}$. 
Class Num $\alpha := \{ \text{zero : } \alpha; \text{one : } \alpha; \text{plus : } \alpha \to \alpha \to \alpha \}$. 

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 \}. 

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

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

(* Defaulting *) 

Check (\lambda x, x + 1).
Numeric overloading

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 "$+" := \text{plus}.$
Numeric overloading

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

Instance \texttt{nat}_\texttt{num} : \texttt{Num }\texttt{nat} :=
\begin{align*}
\{ \\texttt{zero} & := 0\%\texttt{nat} ; \texttt{one} := 1\%\texttt{nat} ; \texttt{plus} := \texttt{Peano}.
\end{align*}

Instance \texttt{Z}_\texttt{num} : \texttt{Num }\texttt{Z} :=
\begin{align*}
\{ \\texttt{zero} & := 0\%\texttt{Z} ; \texttt{one} := 1\%\texttt{Z} ; \texttt{plus} := \texttt{Zplus} \}.
\end{align*}

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

Check (\lambda \texttt{x} : \texttt{nat}, \texttt{x} + (1 + 0 + \texttt{x})).
Check (\lambda \texttt{x} : \texttt{Z}, \texttt{x} + (1 + 0 + \texttt{x})).
Class $\text{Num } \alpha := \{ \text{zero : } \alpha \ ; \ \text{one : } \alpha \ ; \ \text{plus : } \alpha \to \alpha \to \alpha \ \}$.  

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

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

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

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

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

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.
Building hierarchies of classes:

```
Class Fractional '{Num α} :=
{ div : α → { y : α | y ≠ 0 } → α }.
```

```
Class Equivalence α :=
{ equiv_refl : Reflexive α ;
  equiv_sym : Symmetric α ;
  equiv_trans : Transitive α }
```

+ 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)…
Efficiency and control:

- Risk of non-termination
- No forward reasoning
- Little sharing and intelligence in the proof-search (focusing? strategies?)
- Scoping of instances through modules only

**Hope** These are all researched in the logic programming community.
Success of the elaboration point-of-view!

✔ Progress in accessibility and scalability of the tool.

✗ Practical shortcomings: Youth! efficiency and controllability concerns.

✗ Foundational shortcomings: $\eta$-rules, esp. K.
Coq with Classes
M. Sozeau - INRIA Paris

Journées PPS 2011
September 5th 2011
Trouville, France