



CoQ with Classes

Matthieu Sozeau

Project Team πr^2
INRIA Paris

Journées PPS 2011
September 5th 2011
Trouville, France

- ▶ A quick overview of Coq
- ▶ Elaboration
- ▶ Type Classes

- ▶ 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

- ▶ Proof language design:
 - ▶ Separate tactic language \mathcal{L}_{tac} .
 - ▶ Proving tools: proof search, tactics.
 - ▶ Development tools: derived definitions (`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).

- ▶ 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).
  
```

- ▶ True dependent pattern-matching
- ▶ Recursion on inductive families
- ▶ Reasoning on function definitions

Derive Subterm for vector.

Equations `unzip` $\{A\ B\ n\}$ $(v : \text{vector } (A \times B) n)$
 $: \text{vector } A\ n \times \text{vector } B\ n :=$
`unzip` $A\ B\ n\ v$ by `rec` $v :=$
`unzip` $A\ B\ ?(O)\ Vnil := (Vnil, Vnil) ;$
`unzip` $A\ B\ ?(S\ n)\ (Vcons\ (\text{pair } x\ y)\ n\ v)$ with `unzip` $v := \{$
 $| (\text{pair } xs\ ys) := (Vcons\ x\ xs, Vcons\ y\ ys) \}$.

- 1 A brief tour of COQ, PROGRAM and EQUATIONS
 - PROGRAM
 - EQUATIONS
- 2 Type Classes
 - Type Classes from HASKELL
 - Type Classes in COQ
- 3 Conclusion

- ▶ **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`).

- ▶ **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. `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.

```
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.

```
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

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

Parametrized instances and super-classes

```
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.

- ▶ **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 }.
```

- ▶ **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 }.
```

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

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

- ▶ Parametrized dependent records

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

- ▶ Parametrized dependent records

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

- ▶ Parametrized dependent records

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

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

- ▶ Parametrized dependent records

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

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

- ▶ Custom implicit arguments of projections

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

- ▶ Parametrized dependent records

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

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

- ▶ Custom implicit arguments of projections

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

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

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

$\rightsquigarrow \{ \text{Implicit arguments} \}$

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

Elaboration with classes, an example

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

$\rightsquigarrow \{ \text{Implicit arguments} \}$

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

$\rightsquigarrow \{ \text{Unification} \}$

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

Elaboration with classes, an example

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

$\rightsquigarrow \{ \text{Implicit arguments} \}$

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

$\rightsquigarrow \{ \text{Unification} \}$

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

$\rightsquigarrow \{ \text{Proof search for Eq bool returns Eq_bool} \}$

$(\lambda x y : \text{bool}. @\text{eqb } \text{bool } \text{Eq_bool } x y)$

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 \mathcal{L}_{tac}

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

Numeric overloading

Class Num α := { zero : α ; one : α ; 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 `Num` α := { `zero` : α ; `one` : α ; `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` }.

Notation `"0"` := `zero`.

Notation `"1"` := `one`.

Infix `"+"` := `plus`.

Class `Num` α := { `zero` : α ; `one` : α ; `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` }.

Notation `"0"` := `zero`.

Notation `"1"` := `one`.

Infix `"+"` := `plus`.

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

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

Numeric overloading

Class `Num` α := { `zero` : α ; `one` : α ; `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` }.

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 :  $\forall x, R\ x\ x$ .
```

```
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.
```

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.
```

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.
```

Building hierarchies of classes:

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

```
Class Equivalence  $\alpha$  :=  
  { equiv_refl :> Reflexive  $\alpha$  ;  
    equiv_sym  :> Symmetric  $\alpha$  ;  
    equiv_trans :> 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)...

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: η -rules, esp. K.



Inria
INVENTEURS DU MONDE NUMÉRIQUE

CoQ with Classes

M. Sozeau - INRIA Paris



Inria

Journées PPS 2011
September 5th 2011
Trouville, France