Coq with Classes

Matthieu Sozeau

Journées PPS 2011
September 5th 2011
Trouville, France

Project Team πr^2
INRIA Paris

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 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 (Function, 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).

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).
```

Equations

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

```
Derive Subterm for vector.

Equations unzip {A B n} (v : vector (A × B) n)
          : vector A n × 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

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. 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 ...
```

Type Classes

Motivations

- Overloading in programs, specifications and proofs.

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.

```
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.
```

A cheap implementation

- Parametrized dependent records

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

A cheap implementation

- Parametrized dependent records

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

A cheap implementation

- Parametrized dependent records

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

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

A cheap implementation

- Parametrized dependent records

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

  Instances are just definitions of type Id $\overrightarrow{t_n}$.
- Custom implicit arguments of projections

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

A cheap implementation

- Parametrized dependent records

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

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

- Custom implicit arguments of projections

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

Elaboration with classes, an example

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

Elaboration with classes, an example

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

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

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

Elaboration with classes, an example

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

$\rightsquigarrow$ { Implicit arguments }

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

$\rightsquigarrow$ { Unification }

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

Elaboration with classes, an example

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

$\rightsquigarrow$ { Implicit arguments }

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

$\rightsquigarrow$ { Unification }

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

$\rightsquigarrow$ { Proof search for Eq bool returns $\mathrm{Eq\_bool}$ }

$(\lambda x\ y : \mathsf{bool}.\ @\mathrm{eqb}\ \mathsf{bool}\ \mathrm{Eq\_bool}\ x\ y)$

Type Class resolution

Proof-search tactic with instances as lemmas:

$$A : \texttt{Type}, \mathit{eqa} : \mathsf{Eq}\ A \vdash ? : \mathsf{Eq}\ (\mathsf{list}\ A)$$

- ▶ Simple depth-first search with higher-order unification
- – Returns the first solution only
- \+ Extensible through $\mathcal{L}_{\mathsf{tac}}$

Numeric overloading

$$\texttt{Class}\ \mathrm{Num}\ \alpha := \{\ \mathrm{zero} : \alpha\ ;\ \mathrm{one} : \alpha\ ;\ \mathrm{plus} : \alpha \to \alpha \to \alpha\ \}.$$

Numeric overloading

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

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 }.
```

Numeric overloading

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

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.
```

Numeric overloading

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

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 (λ x : nat, x + (1 + 0 + x)).
Check (λ x : Z, x + (1 + 0 + x)).
```

Numeric overloading

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

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 (λ x : nat, x + (1 + 0 + x)).
Check (λ x : Z, x + (1 + 0 + x)).

(* Defaulting *)

Check (λ x, x + 1).
```

Dependent classes

```
Class Reflexive {A} (R : relation A) :=
  refl : ∀ x, R x x.
```

Dependent classes

```
Class Reflexive {A} (R : relation A) :=
  refl : ∀ 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 : ∀ x, R x x.

Instance eq_refl A : Reflexive (@eq A) := @refl_equal A.

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

Goal ∀ P, P ↔ P.
Proof. apply refl. Qed.

Goal ∀ A (x : A), x = x.
Proof. intros A ; apply refl. Qed.
```

Dependent classes

```
Class Reflexive {A} (R : relation A) :=
  refl : ∀ x, R x x.

Instance eq_refl A : Reflexive (@eq A) := @refl_equal A.

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

Goal ∀ P, P ↔ P.
Proof. apply refl. Qed.

Goal ∀ 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:

```
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)...

Current Issues

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.

Conclusions

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.

Coq with Classes

M. Sozeau - INRIA Paris

Journées PPS 2011
September 5th 2011
Trouville, France