First-Class Type Classes

Matthieu Sozeau
Joint work with Nicolas Oury

LRI, Univ. Paris-Sud - Démons Team & INRIA Saclay - ProVal Project

Gallium Seminar
November 3rd 2008
INRIA Rocquencourt

Solutions for overloading

- Intersection types: overloading by declaring multiple signatures for a single constant (e.g. ℂDuce).
- Bounded quantification and class-based overloading. Overloading circumscribed by a subtyping relation (e.g. structural subtyping à la OCaml).

Solutions for overloading

- Intersection types: overloading by declaring multiple signatures for a single constant (e.g. ℂDuce).
- Bounded quantification and class-based overloading. Overloading circumscribed by a subtyping relation (e.g. structural subtyping à la OCaml).

Our objective:

- Modularity: separate definitions of the specializations
- The setting is Coq: no intentional type analysis, no latitude on the kernel language!

Making ad-hoc polymorphism less *ad hoc*

```
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*

```
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
```

Parameterized instances

$$
\begin{array}{ll}
\texttt{instance}\ (\mathsf{Eq}\ A) \Rightarrow \mathsf{Eq}\ [A]\ \texttt{where} & \\
\quad \texttt{[]} == \texttt{[]} & = \mathsf{True} \\
\quad (x : xs) == (y : ys) & = x == y \ \&\& \ xs == ys \\
\quad \_ == \_ & = \mathsf{False}
\end{array}
$$

A structuring concept

```
class Num A where
  (+) :: A → A → A ...
class (Num A) ⇒ Fractional A where
  (/) :: A → A → A ...
class (Fractional A) ⇒ Floating A where
  exp :: A → A ...
```

The MLer point of view

A system of modules and functors with sugar for implicit instantiation and functorization.

Motivations

- Overloading: in programs, specifications and proofs.

Motivations

- Overloading: in programs, specifications and proofs.
- A safer HASKELL Proofs are part of classes, added guarantees. Better extraction.

  Class Eq A :=
  eqb : $A \rightarrow A \rightarrow$ bool ;
  eq_eqb : $\forall\, x\, y$, reflects (eq x y) (eqb x y).

Motivations

- Overloading: in programs, specifications and proofs.
- A safer HASKELL Proofs are part of classes, added guarantees. Better extraction.

  Class Eq A :=
  eqb : $A \to A \to$ bool ;
  eq_eqb : $\forall$ x y, reflects (eq x y) (eqb x y).

- Extensions: dependent types give new power to type classes.

  Class Reflexive A (R : relation A) :=
  reflexive : $\forall$ x, R x x.

Outline

1. Type Classes in Coq
   - A cheap implementation
   - Example: Numbers and monads
2. Superclasses and substructures
   - The power of Pi
   - Example: Categories
3. Extensions
   - Dependent classes
   - Logic Programming
4. Summary, Related, Current and Future Work

Ingredients

- Dependent records: a singleton inductive type containing each component and some projections.

Ingredients

- Dependent records: a singleton inductive type containing each component and some projections.
- Implicit arguments: inferring the value of arguments (e.g. types).

  ```
  Definition id {A : Type} (a : A) : A := a.
  ```

  ```
  Check (@id : Π A, A → A).
  Check (@id nat : nat → nat).
  ```

Ingredients

- Dependent records: a singleton inductive type containing each component and some projections.
- Implicit arguments: inferring the value of arguments (e.g. types).

  Definition id $\{A : \mathtt{Type}\}$ $(a : A) : A := a$.

  Check $(@\mathrm{id} : \Pi\ A, A \rightarrow A)$.
  Check $(@\mathrm{id}\ \mathsf{nat} : \mathsf{nat} \rightarrow \mathsf{nat})$.
  Check $(@\mathrm{id}\ \_ : \mathsf{nat} \rightarrow \mathsf{nat})$.

Ingredients

- Dependent records: a singleton inductive type containing each component and some projections.
- Implicit arguments: inferring the value of arguments (e.g. types).

```
Definition id {A : Type} (a : A) : A := a.

Check (@id : Π A, A → A).
Check (@id nat : nat → nat).
Check (@id _ : nat → nat).
Check (id : nat → nat).
```

Ingredients

- Dependent records: a singleton inductive type containing each component and some projections.
- Implicit arguments: inferring the value of arguments (e.g. types).

```
Definition id {A : Type} (a : A) : A := a.

Check (@id : Π A, A → A).
Check (@id nat : nat → nat).
Check (@id _ : nat → nat).
Check (id : nat → nat).

Check (id 3).
```

Implementation

- Parameterized dependent records

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

Implementation

- Parameterized dependent records

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

Implementation

- Parameterized dependent records

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

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

Implementation

- Parameterized dependent records

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

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

- Custom implicit arguments of projections

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

Implementation

- Parameterized dependent records

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

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

- Custom implicit arguments of projections

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

Implementation

- Parameterized dependent records

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

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

- Custom implicit arguments of projections

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

- Proof-search tactic with instances as lemmas

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

Elaboration with classes, an example

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

Elaboration with classes, an example

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

$\rightsquigarrow$ { implicit arguments }

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

Elaboration with classes, an example

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

$\rightsquigarrow$ { implicit arguments }

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

$\rightsquigarrow$ { unification }

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

Elaboration with classes, an example

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

$\rightsquigarrow$ { implicit arguments }

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

$\rightsquigarrow$ { unification }

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

$\rightsquigarrow$ { proof search for Eq bool returns Eq_bool }

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

Outline

Numeric overloading

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

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)).
```

Monad

```
Class Monad (η : Type → Type) :=
  unit : ∀ {α}, α → η α ;
  bind : ∀ {α β}, η α → (α → η β) → η β ;

  bind_unit_left : ∀ α β (x : α) (f : α → η β),
    bind (unit x) f = f x ;
  bind_unit_right : ∀ α (x : η α), bind x unit = x ;
  bind_assoc : ∀ α β δ
    (x : η α) (f : α → η β) (g : β → η δ),
    bind x (fun a : α ⇒ bind (f a) g) = bind (bind x f) g.
```

Monad

```
Class Monad (η : Type → Type) :=
  unit : ∀ {α}, α → η α ;
  bind : ∀ {α β}, η α → (α → η β) → η β ;

  bind_unit_left : ∀ α β (x : α) (f : α → η β),
    bind (unit x) f = f x ;
  bind_unit_right : ∀ α (x : η α), bind x unit = x ;
  bind_assoc : ∀ α β δ
    (x : η α) (f : α → η β) (g : β → η δ),
    bind x (fun a : α ⇒ bind (f a) g) = bind (bind x f) g.
Infix ">>=" := bind (at level 55).
Notation "x ← T ; E" := (bind T (fun x : _ ⇒ E))
  (at level 30, right associativity).
Notation "'return' t" := (unit t) (at level 20).
```

Definitions

```
Program Instance identity_monad : Monad id :=
  unit α a := a ;
  bind α β m f := f m.
```

Definitions

```
Program Instance identity_monad : Monad id :=
  unit α a := a ;
  bind α β m f := f m.

Section Monad_Defs.
  Context [ mon : Monad η ].
```

Definitions

```
Program Instance identity_monad : Monad id :=
  unit α a := a ;
  bind α β m f := f m.

Section Monad_Defs.
  Context [ mon : Monad η ].

  Definition ap {α β} (f : α → β) (x: η α) : η β :=
    a ← x ; return (f a).

  Definition join {α} (x : η (η α)) : η α :=
    x ≫= id.
```

Proofs

```
Lemma do_return_eta : ∀ α (u : η α),
    x ← u ; return x = u.
Proof. intros α u. rewrite ← (eta_expansion unit).
```

```
η : Type → Type
mon : Monad η
α : Type
u : η α
================================
 u ≫= unit = u
```

Proofs

```
Lemma do_return_eta : ∀ α (u : η α),
    x ← u ; return x = u.
Proof. intros α u. rewrite ← (eta_expansion unit).
```

```
η : Type → Type
mon : Monad η
α : Type
u : η α
==============================
 u >>= unit = u
```

```
    apply bind_unit_right.
Qed.
End Monad_Defs.
```

Outline

1. Type Classes in Coq
   - A cheap implementation
   - Example: Numbers and monads
2. Superclasses and substructures
   - The power of Pi
   - Example: Categories
3. Extensions
   - Dependent classes
   - Logic Programming
4. Summary, Related, Current and Future Work

Fields or Parameters ?

When one doesn't have manifest types and `with` constraints...

```
Class Functor :=
  A : Type; B : Type;
  src : Category A ; dst : Category B ; ...

Class Functor obj obj' :=
  src : Category obj ; dst : Category obj' ; ...

Class Functor obj (src : Category obj) obj' (dst : Category obj')

  := ...
```

???

Sharing by equalities

```
Definition adjunction (F : Functor) (G : Functor),
  src F = dst G → dst F = src G ...
```

Obfuscates the goals and the computations, awkward to use.

Sharing by parameters

$$\texttt{Class}\ \{(C : \mathsf{Category}\ \mathit{obj}, D : \mathsf{Category}\ \mathit{obj'})\} \Rightarrow \mathsf{Functor} := \ldots$$

$$\equiv$$

$$\texttt{Class}\ \mathsf{Functor}\ \{(C : \mathsf{Category}\ \mathit{obj}, D : \mathsf{Category}\ \mathit{obj'})\} := \ldots$$

Sharing by parameters

$$\texttt{Class}\ \{(C : \mathsf{Category}\ \mathit{obj},\ D : \mathsf{Category}\ \mathit{obj'})\} \Rightarrow \mathsf{Functor} := \ldots$$

$$\equiv$$

$$\texttt{Class}\ \mathsf{Functor}\ \{(C : \mathsf{Category}\ \mathit{obj},\ D : \mathsf{Category}\ \mathit{obj'})\} := \ldots$$

$$\equiv$$

$$\texttt{Record}\ \mathsf{Functor}\ \{\mathit{obj}\}\ (C : \mathsf{Category}\ \mathit{obj})$$
$$\{\mathit{obj'}\}(D : \mathsf{Category}\ \mathit{obj'}) := \ldots$$

Sharing by parameters

```
Class {(C : Category obj, D : Category obj')} ⇒ Functor := ...
                              ≡
Class Functor {(C : Category obj, D : Category obj')} := ...
                              ≡
Record Functor {obj} (C : Category obj)
    {obj'}(D : Category obj') := ...

Definition adjunction [ C : Category obj, D : Category obj' ]
    (F : Functor C D) (G : Functor D C) := ...
```

Uses the dependent product and named, first-class instances.

Implicit Generalization

An old convention: the free variables of a statement are implicitly universally quantified. E.g., when defining a set of equations:

$$
\begin{aligned}
x + y &= y + x \\
x + 0 &= 0 \\
x + \mathsf{S}\ y &= \mathsf{S}\ (x + y)
\end{aligned}
$$

Implicit Generalization

An old convention: the free variables of a statement are implicitly universally quantified. E.g., when defining a set of equations:

$$\begin{array}{lcl} x + y & = & y + x \\ x + 0 & = & 0 \\ x + \mathsf{S}\ y & = & \mathsf{S}\ (x + y) \end{array}$$

We introduce new syntax to automatically generalize the free variables of a given term or binder:

$$\begin{array}{lcl} \Gamma \vdash `(t) : \mathtt{Type} & \triangleq & \Gamma \vdash \Pi_{\mathcal{FV}(t)\setminus\Gamma}, t \\ \Gamma \vdash `(t) : T : \mathtt{Type} & \triangleq & \Gamma \vdash \lambda_{\mathcal{FV}(t)\setminus\Gamma}, t \\ \overrightarrow{(x_i : \tau_i)}\ \{(y : T)\} & \triangleq & \overrightarrow{(x_i : \tau_i)}\ \{(\mathcal{FV}(T) \setminus \overrightarrow{x_i})\}\ (y : T) \\ \overrightarrow{(x_i : \tau_i)}\ ((y : T)) & \triangleq & \overrightarrow{(x_i : \tau_i)}\ (\mathcal{FV}(T) \setminus \overrightarrow{x_i})\ (y : T) \end{array}$$

Substructures

A **superclass** becomes a parameter, a **substructure** is a method which is also an instance.

```
Class Monoid A :=
  monop : A → A → A ; ...
Class Group A :=
  grp_mon :> Monoid A ; ...
```

Substructures

A **superclass** becomes a parameter, a **substructure** is a method which is also an instance.

```
Class Monoid A :=
  monop : A → A → A ; ...
Class Group A :=
  grp_mon : Monoid A ; ...
Instance grp_mon [ Group A ] : Monoid A.
```

Substructures

A **superclass** becomes a parameter, a **substructure** is a method which is also an instance.

```
Class Monoid A :=
  monop : A → A → A ; ...
Class Group A :=
  grp_mon : Monoid A ; ...
Instance grp_mon [ Group A ] : Monoid A.
Definition foo [ Group A ] (x : A) : A := monop x x.
```

Similar to the existing `Structure`s based on coercive subtyping.

Outline

1. Type Classes in Coq
   - A cheap implementation
   - Example: Numbers and monads
2. Superclasses and substructures
   - The power of Pi
   - Example: Categories
3. Extensions
   - Dependent classes
   - Logic Programming
4. Summary, Related, Current and Future Work

Category

```
Class Category (obj : Type) (hom : obj → obj → Type) :=
  morphisms :> ∀ a b, Setoid (hom a b) ;
  id : ∀ a, hom a a;
  compose : ∀ {a b c}, hom a b → hom b c → hom a c;
  id_unit_left : ∀ ((f : hom a b)), compose f (id b) == f;
  id_unit_right : ∀ ((f : hom a b)), compose (id a) f == f;
  assoc : ∀ a b c d (f : hom a b) (g : hom b c) (h : hom c d),
    compose f (compose g h) == compose (compose f g) h.
Notation " x '∘' y " := (compose y x)
  (left associativity, at level 40).
```

Abstract instances

```
Definition opposite (X : Type) := X.

Program Instance opposite_category {( Category obj hom )} :
  Category (opposite obj) (flip hom).
```

Abstract instances

```
Definition opposite (X : Type) := X.

Program Instance opposite_category {( Category obj hom )} :
  Category (opposite obj) (flip hom).

Class {(C : Category obj hom)} ⇒ Terminal (one : obj) :=
  bang : ∀ x, hom x one ;
  unique : ∀ x (f g : hom x one), f == g.
```

An abstract proof

```
Definition isomorphic [ Category obj hom ] a b :=
  { f : hom a b & { g : hom b a |
    f ∘ g == id b ∧ g ∘ f == id a } }.
Lemma terminal_isomorphic [ C : Category obj hom ] :
  ‘( Terminal C x → Terminal C y → isomorphic x y ).
Proof.
  intros. red.
  do 2 ∃ (bang _).
  split ; apply unique.
Qed.
```

Outline

Dependent classes (demo script)

```
Class Reflexive {A} (R : relation A) := refl : Π x, R x x.
Print Reflexive.
Instance eq_refl A : Reflexive (@eq A).
Proof. red. apply refl_equal. Qed.
Instance iff_refl : Reflexive iff.
Proof. red. tauto. Qed.
Goal Π P, P ↔ P.
Proof. apply refl. Qed.
Goal Π 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.
```

Boolean formulas

```
Inductive formula :=
| cst : bool → formula
| not : formula → formula
| and : formula → formula → formula
| or : formula → formula → formula
| impl : formula → formula → formula.
```

Boolean formulas

```
Inductive formula :=
| cst : bool → formula
| not : formula → formula
| and : formula → formula → formula
| or : formula → formula → formula
| impl : formula → formula → formula.

Fixpoint interp f :=
  match f with
    | cst b ⇒ if b then True else False
    | not b ⇒ ¬ interp b
    | and a b ⇒ interp a ∧ interp b
    | or a b ⇒ interp a ∨ interp b
    | impl a b ⇒ interp a → interp b
  end.
```

Reification

```
Class Reify (prop : Prop) :=
  reification : formula ;
  reify_correct : interp reification ↔ prop.
```

Reification

```
Class Reify (prop : Prop) :=
  reification : formula ;
  reify_correct : interp reification ↔ prop.

Check (@reification : Π prop : Prop, Reify prop → formula).

Implicit Arguments reification [[Reify]].
```

Reification

```
Class Reify (prop : Prop) :=
  reification : formula ;
  reify_correct : interp reification ↔ prop.

Check (@reification : Π prop : Prop, Reify prop → formula).

Implicit Arguments reification [[Reify]].

Program Instance true_reif : Reify True :=
  reification := cst true.
Program Instance not_reif [ Rb : Reify b ] : Reify (¬ b) :=
  reification := not (reification b).
```

Reification

```
Class Reify (prop : Prop) :=
  reification : formula ;
  reify_correct : interp reification ↔ prop.

Check (@reification : Π prop : Prop, Reify prop → formula).

Implicit Arguments reification [[Reify]].

Program Instance true_reif : Reify True :=
  reification := cst true.
Program Instance not_reif [ Rb : Reify b ] : Reify (¬ b) :=
  reification := not (reification b).

Example example_prop :=
  reification ((True ∧ ¬ False) → ¬ ¬ False).

Check (refl_equal _ : example_prop =
  impl (and (cst true) (not (cst false))) (not (not (cst false)))).
```

Outline

Summary

- ✓ A **lightweight** and **general** implementation of type classes, "available" in Coq v8.2
- ✓ A type-theoretic **explanation** and **extension** of type-classes concepts

On top of that:

- Realistic test-case: a new setoid-rewriting tactic built on top of classes.
- A system to automatically infer instances by Matthias Puech.

How does it compare to Canonical Structures?

- Declaration of a Class and instances instead of using implicit coercions + declaration of some canonical structures.

```
Class Coercion (from to : Type) :=
  coerce : from → to.
```

How does it compare to Canonical Structures?

- Declaration of a Class and instances instead of using implicit coercions + declaration of some canonical structures.
- Indexing on parameters only but less sensible to the shape of unification problems (simpler to explain!).

How does it compare to Canonical Structures?

- Declaration of a Class and instances instead of using implicit coercions + declaration of some canonical structures.
- Indexing on parameters only but less sensible to the shape of unification problems (simpler to explain!).
- Based on an extensible resolution system instead of recursive unification of head constants.

Ongoing and future work

- Debugging!
- Refined, parameterized proof-search (ambiguity checking, mode declarations, discrimination nets . . .)
- Integration with the proof shell: move to **open** terms
- Improve extraction and embedding of HASKELL programs

The End

http://coq.inria.fr/V8.2beta/