First-Class Type Classes

Matthieu Sozeau
Joint work with Nicolas Oury

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

TPHOLs'08
August 20th 2008
Montréal, Canada

Making ad-hoc polymorphism less *ad hoc*

```
class Eq A where
    (==) :: A → A → Bool

class 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

class 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

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

Parameterized instances

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

instance (Eq A) ⇒ Eq (Term A) where
  Var x == Var y           = x == y
  App f l == App f' l'     = f == f' && l == l'
  _ == _                   = False
```

A structuring concept

$$\texttt{class}\ \mathbf{Num}\ A\ \texttt{where}$$
$$\quad (+) :: A \to A \to A \ldots$$

$$\texttt{class}\ (\mathbf{Num}\ A) \Rightarrow \mathbf{Fractional}\ A\ \texttt{where}$$
$$\quad (/) :: A \to A \to A \ldots$$

$$\texttt{class}\ (\mathbf{Fractional}\ A) \Rightarrow \mathbf{Floating}\ A\ \texttt{where}$$
$$\quad \mathrm{exp} :: A \to A \ldots$$

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 → A → bool ;
  eq_eqb : ∀ 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

Implementation

- Parameterized dependent records

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

Implementation

- Parameterized dependent records

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

Implementation

- Parameterized dependent records

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

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

Implementation

- Parameterized dependent records

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

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

- Custom implicit arguments of projections

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

Implementation

- Parameterized dependent records

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

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

- Custom implicit arguments of projections

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

Implementation

- Parameterized dependent records

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

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

- Custom implicit arguments of projections

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

- Proof-search tactic with instances as lemmas

  A : Type, eqa : **Eq** $A \vdash ?$: **Eq** (**list** A)

Elaboration with classes, an example

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

Elaboration with classes, an example

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

$\rightsquigarrow$ { implicit arguments }

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

Elaboration with classes, an example

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

$\rightsquigarrow$ { implicit arguments }

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

$\rightsquigarrow$ { unification }

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

Elaboration with classes, an example

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

$\rightsquigarrow$ { implicit arguments }

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

$\rightsquigarrow$ { unification }

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

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

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

Outline

Numeric overloading

`Class` **Num** α := zero : α ; one : α ; plus : $\alpha \to \alpha \to \alpha$.

`Notation` "0" := zero.
`Notation` "1" := one.
`Infix` "+" := plus.

Numeric overloading

`Class` **Num** α := zero : α ; one : α ; plus : $\alpha \to \alpha \to \alpha$.

`Notation` "0" := zero.
`Notation` "1" := one.
`Infix` "+" := plus.

`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.

Notation "0" := zero.
Notation "1" := one.
Infix "+" := plus.


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

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 ].

  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 (α:=α))), bind_unit_right.
  reflexivity.
Qed.

End Monad_Defs.
```

Outline

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* (**Category** *obj*) *obj'* (**Category** *obj'*) := ...

???

Sharing by equalities

$\texttt{def}$ adjunction $(F : \textbf{Functor})$ $(G : \textbf{Functor})$,
$\quad\quad \text{src } F = \text{dst } G \rightarrow \text{dst } F = \text{src } G \ldots$

Obfuscates the goals and the computations, awkward to use.

Sharing by parameters

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

$$\equiv$$

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

$$\equiv$$

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

Sharing by parameters

`Class` $((C : \textbf{Category}\ obj, D : \textbf{Category}\ obj')) \Rightarrow \textbf{Functor} := \ldots$

$\equiv$

`Class` **Functor** $((C : \textbf{Category}\ obj, D : \textbf{Category}\ obj')) := \ldots$

$\equiv$

`Record` **Functor** $\{obj\}\ (C : \textbf{Category}\ obj)$
$\{obj'\}(D : \textbf{Category}\ obj') := \ldots$

`def` adjunction $[\ C : \textbf{Category}\ obj, D : \textbf{Category}\ obj'\ ]$
$(F : \textbf{Functor}\ C\ D)\ (G : \textbf{Functor}\ D\ C) := \ldots$

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

Superclasses and substructures

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

Example

`Class` **Monoid** $A :=$
  monop : $A \to A \to A$; ...

`Class` **Group** $A :=$
  grp_mon :> Monoid A ; ...

Superclasses and substructures

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

Example

`Class` **Monoid** $A :=$
  monop : $A \to A \to A$; ...

`Class` **Group** $A :=$
  grp_mon : Monoid A ; ...

`Instance` grp_mon [**Group** A] : **Monoid** A.

Superclasses and substructures

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

Example

`Class` **Monoid** $A :=$
  monop : $A \to A \to A$; ...

`Class` **Group** $A :=$
  grp_mon : Monoid A ; ...

`Instance` grp_mon [**Group** A] : **Monoid** A.

`def` foo [**Group** A] (x : A) : A := monop x x.

Similar to the existing `Structures` based on coercive subtyping.

Outline

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 : ∀ a b (f : hom a b),
    compose f (id b) == f;
  id_unit_right : ∀ a b (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 $(X : \texttt{Type}) := X$.

`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.
  ∃ (bang x). ∃ (bang (one:=x) y).
  split.
    apply unique.
    apply (unique (one:=x)).
Qed.
```

Outline

Dependent classes

`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).
`Proof. red. reflexivity. Qed.`

`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).
Proof. red. reflexivity. Qed.

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

Ltac refl := typeclass_app refl.

Goal ∀ P, P ↔ P.
Proof. refl. Qed.

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

Lemma foo [ Reflexive nat R ] : R 0 0.
Proof. intros. refl. Qed.
```

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.

Program Instance true_reif : Reify True :=
  reification := cst true.

Program Instance false_reif : Reify False :=
  reification := cst false.

Program Instance not_reif [ Rb : Reify b ] : Reify (~ b) :=
  reification := not (reification b).
⋮

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

Conclusions

- ✓ An implementation of type classes, available in Coq v8.2, useful for programming and proving.
- ✓ A new setoid-rewriting tactic built on top of classes.
- ✓ A type-theoretic explanation and extension of type-classes concepts.

A good structuring concept for users and tactic implementers.

Ongoing and future work

- Refined, parameterized proof-search (ambiguity checking, mode declarations . . .)
- Integration with the proof shell
- Improve extraction to Haskell and embedding of Haskell programs