Modular and generic developments in Coq

Matthieu Sozeau

MPRI - Proof assistants course
February 11th 2013

Different concepts

Modules are:

- Second-class:
  `Module Type Set. Parameter elt : Type. ...`
- ML-style: functors, signature ascription `M <: T`, specialization `Set with Definition elt := nat`.

Good for: namespace managment, implementation hiding. Seldom used in mathematical developments. A considerable part of the kernel.

Type-classes are:

- First-class (records):
  `Class Set A := ... = Record Set (A : Type) := ...`
- Haskell-style: proof-search for structures, restricted specialization `Set nat`.

Good for: overloading (compile-time binding of names), lightweight generic code. Closer to informal math practice. Outside the kernel.

Solutions for overloading

Context:

- Modularity: separate definitions of the specializations.
- Constrained by Coq: a fixed kernel language!

Solutions for overloading

Context:

- Modularity: separate definitions of the specializations.
- Constrained by Coq: a fixed kernel language!

Existing paradigms:

- Intersection types: closed overloading by declaring multiple signatures for a single constant (e.g. ℂDuce, Stardust).
  `op : (bool -> bool) /\ (int -> int).`
- Bounded quantification and class-based overloading. Overloading circumscribed by a subtyping relation (e.g. structural or nominal subtyping).

```
class type opp = object method op : t end
let foo (opv : opp) = (opv#op = opv)
```

Solutions for overloading

Context:

- Modularity: separate definitions of the specializations.
- Constrained by Coq: a fixed kernel language!

Existing paradigms:

- Intersection types: closed overloading by declaring multiple signatures for a single constant (e.g. CDuce, Stardust).
  `op : (bool -> bool) /\ (int -> int).`
- Bounded quantification and class-based overloading. Overloading circumscribed by a subtyping relation (e.g. structural or nominal subtyping).

```
class type opp = object method op : t end
let foo (opv : opp) = (opv#op = opv)
```

Solution:

Elaborate Type Classes, a kind of bounded quantification where the subtyping relation needs not be internalized.

Type Classes

Enhancing type inference

Overloading:

- Generic programming with interfaces instead of implementations.
- Generic proving: refer to semantic concepts rather than names. E.g. `reflexivity (R:=R)` instead of `R_refl`. Use arbitrary proof search to find instances, mimicking math practice.

Inference of *arbitrary* additional structure on types or values.

Making *ad-hoc* polymorphism less *ad hoc*

In HASKELL, Wadler & Blott, POPL'89.
In ISABELLE, Nipkow & Snelting, FPCA'91.

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

Making *ad-hoc* polymorphism less *ad hoc*

In HASKELL, Wadler & Blott, POPL'89.
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.
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

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

Parameterizing

Parametrized instances

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

Super-classes

```
class Num a where
  (+) :: a -> a -> a ...

class (Num a) => Fractional a where
  (/) :: a -> a -> a ...
```

Type Classes

A cheap implementation

- Parametrized dependent records

  `Class` 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 conclusion Id $\overrightarrow{t_n}$.

A cheap implementation

- Parametrized dependent records

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

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

- Custom implicit arguments of projections

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

A cheap implementation

- Parametrized dependent records

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

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

- Custom implicit arguments of projections

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

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 }

$\lambda x\ y : \mathsf{bool}.\ @\mathrm{eqb}\ \_\ \_\ x\ y$

Elaboration with classes, an example

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

$\rightsquigarrow$ { Implicit arguments }

$\lambda x\ y : \mathsf{bool}.\ @\mathrm{eqb}\ \_\ \_\ x\ y$

$\rightsquigarrow$ { Typing }

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

Elaboration with classes, an example

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

$\rightsquigarrow$ { Implicit arguments }

$\lambda x\ y : \mathsf{bool}.\ @\mathrm{eqb}\ \_\ \_\ x\ y$

$\rightsquigarrow$ { Typing }

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

$\rightsquigarrow$ { 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 }

$\lambda x\ y : \mathsf{bool}.\ @\mathrm{eqb}\ \_\ \_\ x\ y$

$\rightsquigarrow$ { Typing }

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

$\rightsquigarrow$ { Unification }

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

$\rightsquigarrow$ { Proof search for $\mathsf{Eq}\ \mathsf{bool}$ 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:

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

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

Type Classes

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)).
```

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).
```

Instance inference

```
Class EqDec A := eq_dec : ∀ x y : A, {x = y} + {x ≠ y}.
Instance: EqDec nat := { eq_dec := eq_nat_dec }.
Instance: EqDec bool := { eq_dec := bool_dec }.
```

Instance inference

```
Class EqDec A := eq_dec : ∀ x y : A, {x = y} + {x ≠ y}.

Instance: EqDec nat := { eq_dec := eq_nat_dec }.
Instance: EqDec bool := { eq_dec := bool_dec }.

Program Instance: ∀ A, EqDec A → EqDec (option A) := {
  eq_dec x y := match x, y with
  | None, None ⇒ in_left
  | Some x, Some y ⇒ if eq_dec x y then in_left else in_right
  | _, _ ⇒ in_right
  end }.
```

Instance inference

```
Class EqDec A := eq_dec : ∀ x y : A, {x = y} + {x ≠ y}.

Instance: EqDec nat := { eq_dec := eq_nat_dec }.
Instance: EqDec bool := { eq_dec := bool_dec }.

Program Instance: ∀ A, EqDec A → EqDec (option A) := {
  eq_dec x y := match x, y with
  | None, None ⇒ in_left
  | Some x, Some y ⇒ if eq_dec x y then in_left else in_right
  | _, _ ⇒ in_right
  end }.

Check (λ x : option (option nat), eq_dec x None).
: ∀ x : option (option nat), {x = None} + {x ≠ None}

Eval compute in (eq_dec (Some (Some 0)) (Some None)).
= in_right : {Some (Some 0) = Some None} + {Some (Some 0)
≠ Some None}
```

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 refl := apply refl.

Lemma foo `{Reflexive nat R} : R 0 0.
Proof. intros. refl. Qed.
```

Type Classes

Implicit Generalization

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

$$
\begin{array}{lcll}
x + y & = & y + x & \forall x\, y \in \mathbb{N} \\
x + 0 & = & 0 & \forall x \in \mathbb{N} \\
x + \mathsf{S}\, y & = & \mathsf{S}\,(x + y) & \forall x\, y \in \mathbb{N}
\end{array}
$$

Implicit Generalization

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

$$
\begin{array}{lcll}
x + y & = & y + x & \forall x\, y \in \mathbb{N} \\
x + 0 & = & 0 & \forall x \in \mathbb{N} \\
x + \mathsf{S}\ y & = & \mathsf{S}\ (x + y) & \forall x\, y \in \mathbb{N}
\end{array}
$$

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

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

Monomorphic binary power

The following definition is very naïve, but obviously correct:

```
Fixpoint power (a : Z) (n : nat) :=
  match n with
    | 0%nat ⇒ 1
    | S p ⇒ a × power a p
  end.

Eval vm_compute in power 2 40.
= 1099511627776 : Z
```

An efficient tail-recursive version

This one is more efficient but relies on a more elaborate property:

```
Function binary_power_mult (acc x : Z) (n : nat)
  {measure (fun i⇒i) n} : Z :=
  match n with
    | 0%nat ⇒ acc
    | _ ⇒ if Even.even_odd_dec n
      then binary_power_mult acc (x × x) (div2 n)
      else binary_power_mult (acc × x) (x × x) (div2 n)
  end.

Definition binary_power (x:Z) (n:nat) :=
  binary_power_mult 1 x n.

Eval vm_compute in binary_power 2 40.
= 1099511627776 : Z

Goal binary_power 2 234 = power 2 234.
Proof. reflexivity. Qed.
```

Questions

- Is binary_power correct (w.r.t. power)?

Questions

- Is binary_power correct (w.r.t. power)?
- Is it worth proving this correctness only for powers of integers?

Questions

- Is binary_power correct (w.r.t. power)?
- Is it worth proving this correctness only for powers of integers?
- And prove it again for powers of real numbers, matrices?

Questions

- Is binary_power correct (w.r.t. power)?
- Is it worth proving this correctness only for powers of integers?
- And prove it again for powers of real numbers, matrices?

NO!

Program with interfaces, here a monoid.

Monoids

```
Class Monoid {A:Type} (dot : A → A → A) (one : A) : Type :=
  { dot_assoc : ∀ x y z : A, dot x (dot y z) = dot (dot x y) z;
    one_left : ∀ x, dot one x = x;
    one_right : ∀ x, dot x one = x }.
```

Operations as parameters:
Monoid 0 *plus* and Monoid 1 *mult*.

Implicit Generalization

Quantification on parameters:

```
Definition two {A dot one} {M : @Monoid A dot one} :=
  dot one one.
```

Using implicit generalization:

```
Generalizable Variables A dot one.
Definition three `{Monoid A dot one} := dot two one.
```

Generic notations

Global names for parameters:

```
Definition monop `{Monoid A dot one} := dot.
Definition monunit `{Monoid A dot one} := one.
```

Generic notations:

```
Infix "×" := monop.
Notation "1" := monunit.
```

Generic power

Generic power and binary_power.

```
Section Power.
  Context `{Monoid A dot one}.

  Fixpoint power (a : A) (n : nat) :=
    match n with
      | 0%nat ⇒ 1
      | S p ⇒ a × (power a p)
    end.

  Lemma power_of_unit : ∀ n : nat, power 1 n = 1.
  Proof. ...  Qed.
```

Generic binary exponentiation

```
Function binary_power_mult (acc x : A) (n : nat)
  {measure (fun i⇒i) n} : A :=
  match n with
    | 0%nat ⇒ acc
    | _ ⇒ if Even.even_odd_dec n
      then binary_power_mult acc (x × x) (div2 n)
      else binary_power_mult (acc × x) (x × x) (div2 n)
  end.
Definition binary_power (x : A) (n : nat) :=
  binary_power_mult 1 x n.

Lemma binary_spec x n : power x n = binary_power x n.
Proof. ...  Qed.

End Power.
```

Instantiation

A Monoid instance.

```
Instance ZMult : Monoid Zmult 1%Z.
Proof. split.
subgoal 1 is:
∀ x y z : Z, x × (y × z) = x × y × z
subgoal 2 is:
∀ x : Z, 1 × x = x
subgoal 3 is:
∀ x : Z, x × 1 = x
```

... Qed.

Instantiation

Instanciation of power.

```
About power.
```

: $\forall$ (A : Type) (dot : $A \to A \to A$) (one : A), Monoid dot one $\to$ $A \to$ nat $\to A$

Arguments A, dot, one, H are implicit and maximally inserted

Instantiation

Instanciation of power.

```
About power.
```

: $\forall$ (A : Type) (dot : $A \to A \to A$) (one : A), Monoid dot one $\to$ $A \to$ nat $\to A$

Arguments A, dot, one, H are implicit and maximally inserted

```
Set Printing Implicit.
Check power 2 100.
```

@power Z *Z.mul* 1 ZMult 2 100 : Z

Instantiation

Instanciation of power.

```
About power.
```

: $\forall$ (A : Type) (dot : $A \to A \to A$) (one : A), Monoid dot one $\to$ $A \to$ nat $\to A$

Arguments A, dot, one, H are implicit and maximally inserted

```
Set Printing Implicit.
Check power 2 100.
```

@power Z *Z.mul* 1 ZMult 2 100 : Z

```
Compute power 2 100.
```

= 1267650600228229401496703205376 : Z

Live demo

Type Classes

Building hierarchies of classes: superclasses

Superclasses become **parameters**:

$$\texttt{Class}\ (\mathsf{Num}\ \alpha) \Rightarrow \mathsf{Frac}\ \alpha :=$$
$$\{\ \mathrm{div} : \alpha \to \{\ y : \alpha \mid y \neq 0\ \} \to \alpha\ \}.$$

$$\Rightarrow$$

$$\texttt{Class}\ \mathsf{Frac}\ \alpha\ \text{`}\{\mathsf{Num}\ \alpha\} :=$$
$$\{\ \mathrm{div} : \alpha \to \{\ y : \alpha \mid y \neq 0\ \} \to \alpha\ \}.$$

Building hierarchies of classes: superclasses

Superclasses become **parameters**:

$$\texttt{Class}\ (\mathsf{Num}\ \alpha) \Rightarrow \mathsf{Frac}\ \alpha :=$$
$$\{\ \mathrm{div} : \alpha \to \{\ y : \alpha \mid y \neq 0\ \} \to \alpha\ \}.$$

$$\Rightarrow$$

$$\texttt{Class}\ \mathsf{Frac}\ \alpha\ \text{`}\{\mathsf{Num}\ \alpha\} :=$$
$$\{\ \mathrm{div} : \alpha \to \{\ y : \alpha \mid y \neq 0\ \} \to \alpha\ \}.$$

$+$ Binding super-classes by implicit generalization:

$$\texttt{Program Definition}\ \mathrm{div2}\ \text{`}\{\mathsf{Frac}\ \alpha\}\ (a : \alpha) := \mathrm{div}\ a\ (1+1).$$

$$\Rightarrow$$

$$\texttt{Definition}\ \mathrm{div2}\ \{\alpha\}\ \{N : \mathsf{Num}\ \alpha\}\ \{\mathsf{Frac}\ \alpha\ N\}\ (a : \alpha) :=$$
$$\ldots$$

Building hierarchies of classes: subinstances

Substructures become **subinstances**:

$$\texttt{Class}\ \mathsf{Monoid}\ A := \{\ \mathrm{monop} : A \to A \to A\ ;\ \ldots\ \}$$

$$\texttt{Class}\ \mathsf{Group}\ A := \{\ \mathrm{grp\_mon} :> \mathrm{Monoid}\ A\ ;\ \ldots\ \}$$

Building hierarchies of classes: subinstances

Substructures become **subinstances**:

```
Class Monoid A := { monop : A → A → A ; ... }
Class Group A := { grp_mon : Monoid A ; ... }
Instance grp_mon `{Group A} : Monoid A.
```

Building hierarchies of classes: subinstances

Substructures become **subinstances**:

```
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.

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 ; ... }
```

or

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

or

```
Class Functor A (src : Category A) B (dst : Category B) := ...
```

???

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

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

$$\equiv$$

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

Sharing by parameters

```
Class (C : Category obj, D : Category obj') ⇒ Functor := ...
```

$$\equiv$$

```
Class Functor `(C : Category obj, D : Category obj') := ...
```

$$\equiv$$

```
Record Functor {obj} (C : Category obj)
    {obj'}(D : Category obj') := ...
```

Sharing by parameters

```
Class (C : Category obj, D : Category obj') ⇒ Functor := ...
```

$$\equiv$$

```
Class Functor `(C : Category obj, D : Category obj') := ...
```

$$\equiv$$

```
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.

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 Terminal `(C : Category obj hom) (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 eexists (bang _).
  split ; apply unique.
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 : ∀ prop : Prop, Reify prop → formula).

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

Reification

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

Check (@reification : ∀ 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 : ∀ 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)))).
```

Type Classes as logic programming

Implement domain-specific proof-automation:

- Discharge separation logic disjointness side-conditions (Nanevsky et al, ICFP'11)
- Generalized rewriting tactic using proof-search for morphisms (Sozeau, JFR'09)
- Derive continuity, monotonicity conditions. . .

Type Classes

Related work

Type Classes implementations:

- In Haskell by Wadler *et al.* (POPL'89, FO, second class)
- In Isabelle by Nipkow *et al.* (POPL'93, same)
- In Agda by Devriese and Piessens (ICFP'11, non-recursive proof search)

In Coq and Matita:

- Coercive Subtyping and `Canonical Structure`s (Saïbi, POPL'97). Used by Gonthier *et al.* (TPHOLs'09), Nanevski *et al.* (ICFP'11).
- Unification hints, a more general framework studied by Asperti *et al.* (TPHOLs'09).

Experiments and formalizations in Coq

- Sets, Maps etc... (Letouzey, Lescuyer ...)
- Domain theory, probability monad (Paulin, ...)
- Generalized rewriting (Sozeau, JFR'09)
- ACI rewriting (Braibant & Pous, ITP'11)
- Universal algebra, category theory and computable reals (Spitters *et al.*, ITP'10)

Exercises

1. Complete the proof of correctness of fibonacci: http://www.pps.univ-paris-diderot.fr/~sozeau/teaching/Classes/Matrices.v
2. Define a reifier for expressions in the monoid class, a reflexive tactic for simplifying monoidal expressions, and an overloaded lemma to apply it.
3. Define the state monad and prove a tree labeling algorithm with it: http://www.pps.univ-paris-diderot.fr/~sozeau/teaching/Classes/Monad.v