



CoQ with Classes

Matthieu Sozeau

Project Team πr^2
INRIA Paris &
PPS, Paris 7 University

JFLA 2012
February 4th-7th 2012
Carnac, France

- 1 Type Classes theory: from HASKELL to COQ
- 2 Demonstrations

- 1 Type Classes in theory
 - Motivation and setup
 - Type Classes from HASKELL
 - Type Classes in COQ
- 2 Type Classes in practice
 - Playing with numbers
 - Instance Inference
 - Dependent Classes
 - A programming example: Generic exponentiation
 - A structuring tool: Building hierarchies
 - Logic programming: Reification
- 3 Summary
 - Related and future work

Enhancing type inference through **overloading**:

- ▶ For generic programming with interfaces rather than concrete implementations. Akin to a first-class module system.
- ▶ For generic proof scripts: refer to proofs by semantic concept rather than name. E.g. `reflexivity` of `R` instead of `R_refl`.

In general, allows inference of arbitrary additional structure on a given type or value.

Context:

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

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).
- ▶ **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!

Existing paradigms:

- ▶ **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`).

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
```


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 Type Classes in theory
 - Motivation and setup
 - Type Classes from HASKELL
 - Type Classes in Coq
- 2 Type Classes in practice
 - Playing with numbers
 - Instance Inference
 - Dependent Classes
 - A programming example: Generic exponentiation
 - A structuring tool: Building hierarchies
 - Logic programming: Reification
- 3 Summary
 - Related and future work

- ▶ **Overloading** of names in programs, specifications and proofs.

- ▶ **Overloading** of names 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** of names 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 }.
```

- ▶ **Types and values are unified** 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** \vec{t}_n .

- ▶ Custom implicit arguments of projections

$\mathbf{f}_1 : \forall \vec{\alpha}_n : \vec{\tau}_n , \mathbf{ld} \vec{\alpha}_n \rightarrow \phi_1$

- ▶ 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

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

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

Elaboration with classes, an example

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

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

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

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}

- 1 Type Classes in theory
 - Motivation and setup
 - Type Classes from HASKELL
 - Type Classes in COQ
- 2 Type Classes in practice
 - Playing with numbers
 - Instance Inference
 - Dependent Classes
 - A programming example: Generic exponentiation
 - A structuring tool: Building hierarchies
 - Logic programming: Reification
- 3 Summary
 - Related and future work

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`.

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

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 EqDec A := {  
  eq_dec :  $\forall x y : A, \{x = y\} + \{x \neq y\}$   
}.
```

```
Instance: EqDec nat := { eq_dec := eq_nat_dec }.
```

```
Instance: EqDec bool := { eq_dec := bool_dec }.
```

```
Class EqDec A := {  
  eq_dec :  $\forall x y : A, \{x = y\} + \{x \neq y\}$   
}.
```

```
Instance: EqDec nat := { eq_dec := eq_nat_dec }.
```

```
Instance: EqDec bool := { eq_dec := bool_dec }.
```

```
Program Instance:  $\forall A, \text{EqDec } A \rightarrow \text{EqDec (option } A)$  := {  
  eq_dec x y := match x, y with  
  | None, None  $\Rightarrow$  in_left  
  | Some x, Some y  $\Rightarrow$  if eq_dec x y then in_left else in_right  
  | _, _  $\Rightarrow$  in_right  
  end }.
```

```
Check ( $\lambda x : \text{option (option nat)}, \text{eq\_dec } x \text{ None}$ ).
```

```
:  $\forall x : \text{option (option nat)}, \{x = \text{None}\} + \{x \neq \text{None}\}$ 
```

```
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 refl := apply refl.
```

```
Lemma foo '{Reflexive nat R} : R 0 0.
```

```
Proof. intros. refl. Qed.
```

- 1 Type Classes in theory
 - Motivation and setup
 - Type Classes from HASKELL
 - Type Classes in COQ
- 2 Type Classes in practice
 - Playing with numbers
 - Instance Inference
 - Dependent Classes
 - A programming example: Generic exponentiation
 - A structuring tool: Building hierarchies
 - Logic programming: Reification
- 3 Summary
 - Related and future work

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

$$x + y = y + x$$

$$x + 0 = 0$$

$$x + S y = S (x + y)$$

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

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

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

$$\begin{aligned}\Gamma : '(t) : \mathbf{Type} &\triangleq \Gamma : \Pi_{\mathcal{FV}(t)\setminus\Gamma} t \\ \Gamma : '(t) : T : \mathbf{Type} &\triangleq \Gamma : \lambda_{\mathcal{FV}(t)\setminus\Gamma} t \\ \overrightarrow{(x_i : \tau_i)} '\{y : T\} &\triangleq \overrightarrow{(x_i : \tau_i)} \{\mathcal{FV}(T) \setminus \overrightarrow{x_i}\} \{y : T\} \\ \overrightarrow{(x_i : \tau_i)} '(y : T) &\triangleq \overrightarrow{(x_i : \tau_i)} \{\mathcal{FV}(T) \setminus \overrightarrow{x_i}\} (y : T)\end{aligned}$$

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 :  $\mathbb{Z}$ ) (n : nat)
  {measure (fun i ⇒ i) n} :  $\mathbb{Z}$  :=
  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: $\mathbb{Z}$ ) (n:nat) :=
  binary_power_mult 1 x n.
```

```
Eval vm_compute in binary_power 2 40.
= 1099511627776 :  $\mathbb{Z}$ 
```

```
Goal binary_power 2 234 = power 2 234.
```

```
Proof. reflexivity. Qed.
```

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

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

- ▶ 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?

- ▶ 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!

We aim to prove the equivalence between `power` and `binary_power` for any structure consisting of a binary associative operation that admits a neutral element, i.e. any monoid.

```
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 to ease sharing, allows to specify multiple monoids on the same carrier unambiguously, e.g.

`Monoid 0 plus` and `Monoid 1 mult`.

Quantification becomes verbose:

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

Using implicit generalization:

Generalizable Variables $A \text{ dot } one.$

Definition `three` $\{Monoid A \text{ dot } one\} := \text{dot two } one.$

One can define trivial projections to recover global names for parameters:

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

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

and the corresponding generic notations:

Infix "`×`" := `monop`.

Notation "`1`" := `monunit`.

Let's redefine `power` and `binary_power` generically.

`Section Power.`

`Context '{Monoid A dot one}.`

All following definitions are overloaded over any `Monoid` structure.

```
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.`

```
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.

Let's build a `Monoid` instance.

`Instance ZMult : Monoid Zmult 1%Z.`

`Proof. split.`

`subgoal 1 is:`

$\forall x y z : \mathbb{Z}, x \times (y \times z) = x \times y \times z$

`subgoal 2 is:`

$\forall x : \mathbb{Z}, 1 \times x = x$

`subgoal 3 is:`

$\forall x : \mathbb{Z}, x \times 1 = x$

`... Qed.`

We can now use the overloaded `power` on our new `Monoid`.

About `power`.

$$: \forall (A : \text{Type}) (dot : A \rightarrow A \rightarrow A) (one : A), \text{Monoid } dot \text{ one} \rightarrow A \rightarrow \text{nat} \rightarrow A$$

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

We can now use the overloaded `power` on our new `Monoid`.

About `power`.

```
: ∀ (A : Type) (dot : A → A → A) (one : A), Monoid dot one →  
A → nat → 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
```

We can now use the overloaded `power` on our new `Monoid`.

About `power`.

```
: ∀ (A : Type) (dot : A → A → A) (one : A), Monoid dot one →  
A → nat → 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

- 1 Type Classes in theory
 - Motivation and setup
 - Type Classes from `HASKELL`
 - Type Classes in `COQ`
- 2 Type Classes in practice
 - Playing with numbers
 - Instance Inference
 - Dependent Classes
 - A programming example: Generic exponentiation
 - **A structuring tool: Building hierarchies**
 - Logic programming: Reification
- 3 Summary
 - Related and future work

Superclasses become **parameters**:

Class (Num α) \Rightarrow **Frac** α :=
{ **div** : $\alpha \rightarrow \{ y : \alpha \mid y \neq 0 \} \rightarrow \alpha$ }.
 \Rightarrow

Class **Frac** α { **Num** α } :=
{ **div** : $\alpha \rightarrow \{ y : \alpha \mid y \neq 0 \} \rightarrow \alpha$ }.

Superclasses become **parameters**:

Class (Num α) \Rightarrow **Frac** α :=
{ **div** : $\alpha \rightarrow \{ y : \alpha \mid y \neq 0 \} \rightarrow \alpha$ }.
 \Rightarrow

Class **Frac** α '{**Num** α } :=
{ **div** : $\alpha \rightarrow \{ y : \alpha \mid y \neq 0 \} \rightarrow \alpha$ }.

+ Support for binding super-classes by implicit generalization:

Program Definition **div2** '{**Frac** α } ($a : \alpha$) := **div** a (1 + 1).
 \Rightarrow

Definition **div2** { α } {**N** : **Num** α } {**Frac** α **N**} ($a : \alpha$) :=
...

Substructures become **subinstances**:

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

```
Class Group A := { grp_mon :> Monoid A ; ... }
```

Substructures become **subinstances**:

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

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

```
Instance grp_mon '{Group A} : Monoid A.
```

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 **Structures** 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) := ...
```

???

Definition adjunction $(F : \text{Functor}) (G : \text{Functor})$,
 $\text{src } F = \text{dst } G \rightarrow \text{dst } F = \text{src } G \rightarrow \dots$

Obfuscates the goals and the computations, awkward to use.

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

Class ($C : \text{Category } obj, D : \text{Category } obj'$) \Rightarrow **Functor** := ...

\equiv

Class **Functor** '($C : \text{Category } obj, D : \text{Category } obj'$) := ...

\equiv

Record **Functor** { obj } ($C : \text{Category } obj$)
{ obj' } ($D : \text{Category } obj'$) := ...

Class $(C : \text{Category } obj, D : \text{Category } obj') \Rightarrow \text{Functor} := \dots$

\equiv

Class **Functor** $(C : \text{Category } obj, D : \text{Category } obj') := \dots$

\equiv

Record **Functor** $\{obj\} (C : \text{Category } obj)$
 $\{obj'\} (D : \text{Category } obj') := \dots$

Definition **adjunction** $\{C : \text{Category } obj, D : \text{Category } obj'\}$
 $(F : \text{Functor } C D) (G : \text{Functor } D C) := \dots$

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

```
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  
    where " x 'o' y " := (compose y x) ;  
  
  id_unit_left : ∀ '(f : hom a b), f ∘ id == f ;  
  id_unit_right : ∀ '(f : hom a b), id ∘ f == f ;  
  assoc : ∀ a b c d (f : hom a b) (g : hom b c) (h : hom c d),  
    h ∘ (g ∘ f) == (h ∘ g) ∘ f }.
```

Definition `opposite` ($X : \text{Type}$) := X .

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

Definition `opposite` ($X : \text{Type}$) := X .

Program Instance `opposite_category` '(`Category` $obj\ hom$) :
`Category` (`opposite` obj) (`flip` hom).

Class `Terminal` '(`C` : `Category` $obj\ hom$) (`one` : obj) := {
`bang` : $\forall x, hom\ x\ one$;
`unique` : $\forall x (f\ g : hom\ x\ one), f == g$ }.

Definition `isomorphic` '{ Category obj hom } a b :=
{ f : hom a b & { g : hom b a |
 f o g == id ∧ g o f == id } }.

Definition `isomorphic` '{ `Category obj hom` } `a b` :=
{ `f : hom a b` & { `g : hom b a` |
 `f o g == id` & `g o f == id` } }.

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.

- 1 Type Classes in theory
 - Motivation and setup
 - Type Classes from HASKELL
 - Type Classes in COQ
- 2 Type Classes in practice
 - Playing with numbers
 - Instance Inference
 - Dependent Classes
 - A programming example: Generic exponentiation
 - A structuring tool: Building hierarchies
 - Logic programming: Reification
- 3 Summary
 - Related and future work

Inductive formula :=

| **cst** : bool → formula

| **not** : formula → formula

| **and** : formula → formula → formula

| **or** : formula → formula → formula

| **impl** : formula → formula → formula.

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.

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

```
Class Reify (prop : Prop) :=
```

```
  { reification : formula ;
```

```
    reify_correct : interp reification  $\leftrightarrow$  prop }.
```

```
Check (@reification :  $\forall$  prop : Prop, Reify prop  $\rightarrow$  formula).
```

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

```
Class Reify (prop : Prop) :=
```

```
{ reification : formula ;
```

```
  reify_correct : interp reification  $\leftrightarrow$  prop }.
```

```
Check (@reification :  $\forall$  prop : Prop, Reify prop  $\rightarrow$  formula).
```

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

```
Program Instance true_reif : Reify True :=
```

```
{ reification := cst true }.
```

```
Program Instance not_reif '(Rb : Reify b) : Reify ( $\neg$  b) :=
```

```
{ reification := not (reification b) }.
```

```
Class Reify (prop : Prop) :=
```

```
{ reification : formula ;
```

```
  reify_correct : interp reification  $\leftrightarrow$  prop }.
```

```
Check (@reification :  $\forall$  prop : Prop, Reify prop  $\rightarrow$  formula).
```

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

```
Program Instance true_reif : Reify True :=
```

```
{ reification := cst true }.
```

```
Program Instance not_reif '(Rb : Reify b) : Reify ( $\neg$  b) :=
```

```
{ reification := not (reification b) }.
```

```
Example example_prop :=
```

```
  reification ((True  $\wedge$   $\neg$  False)  $\rightarrow$   $\neg$   $\neg$  False).
```

```
Check (refl_equal _ : example_prop =
```

```
  impl (and (cst true) (not (cst false))) (not (not (cst false))))).
```


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. . .

Goal: rewrite with arbitrary relations. Based on a generic compatibility statement:

Definition `respectful {A} (R : relation A)`
`{B} (R' : relation B) : relation (A → B) :=`
`fun f g => ∀ x y : A, R x y → R' (f x) (g y).`

Notation `" R ==l R' "` := `(respectful R R')`
(right associativity, at level 80).

Morphism instances

Proper R m means m is a proper element of R .

```
Class Proper {A} (R : relation A) (m : A) :=  
  respect : R m m.
```

```
Check (iff : Prop → Prop → Prop).
```

```
Instance: Proper (iff ⇒ iff ⇒ iff) iff.
```

```
Proof. do 6 intro.
```

```
x, y : Prop
```

```
H : x ↔ y
```

```
x0, y0 : Prop
```

```
H0 : x0 ↔ y0
```

```
=====
```

```
(x ↔ x0) ↔ (y ↔ y0)
```

```
  firstorder.
```

```
Qed.
```

Instance: `Proper (iff \implies iff \implies iff)` or.

Proof. `firstorder. Qed.`

Instance `proper_trans` '{`Transitive A R`} :

`Proper (iff \implies iff \implies iff) iff.`

Proof. `firstorder. Qed.`

+ morphisms for partial applications, binding constructs...

Rewriting generates `Proper` instance constraints, which when solved prove the rewrite is correct.

- 1 Type Classes in theory
 - Motivation and setup
 - Type Classes from `HASKELL`
 - Type Classes in `COQ`
- 2 Type Classes in practice
 - Playing with numbers
 - Instance Inference
 - Dependent Classes
 - A programming example: Generic exponentiation
 - A structuring tool: Building hierarchies
 - Logic programming: Reification
- 3 Summary
 - Related and future 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 Structures** (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).

- ▶ 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)

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- ▶ Undeterministic proof-search

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- ▶ Undeterministic proof-search
 - ⇒ Determinacy inference (KRIENER and KING, ICLP'11)

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- ▶ Undeterministic proof-search
⇒ Determinacy inference (KRIENER and KING, ICLP'11)
- ▶ No forward reasoning or reordering of constraints

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- ▶ Undeterministic proof-search
⇒ Determinacy inference (KRIENER and KING, ICLP'11)
- ▶ No forward reasoning or reordering of constraints
⇒ Mode analysis (à la PROLOG, TWELF)

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- ▶ Undeterministic proof-search
⇒ Determinacy inference (KRIENER and KING, ICLP'11)
- ▶ No forward reasoning or reordering of constraints
⇒ Mode analysis (à la PROLOG, TWELF)
- ▶ Risk of non-termination

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- ▶ Undeterministic proof-search
⇒ Determinacy inference (KRIENER and KING, ICLP'11)
- ▶ No forward reasoning or reordering of constraints
⇒ Mode analysis (à la PROLOG, TWELF)
- ▶ Risk of non-termination
⇒ Termination analysis, requires modes

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- ▶ Undeterministic proof-search
⇒ Determinacy inference (KRIENER and KING, ICLP'11)
- ▶ No forward reasoning or reordering of constraints
⇒ Mode analysis (à la PROLOG, TWELF)
- ▶ Risk of non-termination
⇒ Termination analysis, requires modes
- ▶ Little sharing and intelligence in the proof-search

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- ▶ Undeterministic proof-search
 - ⇒ Determinacy inference (KRIENER and KING, ICLP'11)
- ▶ No forward reasoning or reordering of constraints
 - ⇒ Mode analysis (à la PROLOG, TWELF)
- ▶ Risk of non-termination
 - ⇒ Termination analysis, requires modes
- ▶ Little sharing and intelligence in the proof-search
 - ⇒ Focusing, strategies.

Proof search efficiency and control issues...

Prerequisite Proper formalization of unification

Hope These are all researched in the logic programming community

- ▶ Undeterministic proof-search
⇒ Determinacy inference (KRIENER and KING, ICLP'11)
- ▶ No forward reasoning or reordering of constraints
⇒ Mode analysis (à la PROLOG, TWELF)
- ▶ Risk of non-termination
⇒ Termination analysis, requires modes
- ▶ Little sharing and intelligence in the proof-search
⇒ Focusing, strategies.
- ▶ Scoping of instances... through modules only.

- ✓ A **lightweight** and **general** implementation of type classes, available since COQ v8.2.
- ✓ A type-theoretic **explanation** and **extension** of type classes concepts (TPHOLs'08, with NICOLAS OURY).

Takeaway:

- ▶ Lightweight, first-class modules
- ▶ An overloading mechanism
- ▶ A logic programming language on top of COQ