Category Theory in Coq 8.5

Amin Timany Bart Jacobs

iMinds-Distrinet KU Leuven

7th Coq Workshop – Sophia Antipolis
June 26, 2015

List of the most important formalized notions

- basic constructions:
  - terminal/initial object
  - products/sums
  - equalizers/coequalizers
  - pullbacks/pushouts
  - exponentials
  - $+ \dashv \Delta \dashv \times$ and $(- \times a) \dashv a^{-}$
- external constructions:
  - comma categories
  - product category
- for **Cat**: (`Obj` := `Category`, `Hom` := `Functor`)
  - cartesian closure
  - initial object
- for **Set**: (`Obj` := `Type`, `Hom` := `fun A B ⇒ A → B`)
  - initial object
  - sums
  - equalizers
  - coequalizers†
  - pullbacks
  - cartesian closure
  - local cartesian closure†
  - completeness
  - co-completeness†
  - sub-object classifier (`Prop : Type`)†
  - topos†

  †uses the axioms of propositional extensionality and constructive indefinite description (axiom of choice).
- the Yoneda lemma

- adjunction
  - hom-functor adjunction, unit-counit adjunction, universal morphism adjunction and their conversions
  - duality : $F \dashv G \Rightarrow G^{op} \dashv F^{op}$
  - uniqueness up to natural isomorphism
  - category of adjunctions
- kan extensions
  - global definition
  - local definition with both hom-functor and cones (along a functor)
  - uniqueness
  - preservation by adjoint functors
  - pointwise kan extensions (preserved by representable functors)
- (co)limits
  - as (left)right local kan extensions along the unique functor to the terminal category
  - (sum)product-(co)equalizer (co)limits
  - pointwise (as kan extensions)
- $T-(co)algebras$ (for an endofunctor T)

  we use proof functional extensionality

  we use proof irrelevance in many cases (mostly for proof of equality of arrows)
- This implementation can be found at:
  `https://bitbucket.org/amintimany/categories/`

- This implementation uses some features of Coq 8.5, most notably:
  - Primitive projections for records:
  - Universe polymorphism: for relative smallness/largeness

- Primitive projections for records:
  - The η rule for records: two instance of a record type are *definitionally* equal if all their respective projections are
  - E.g., for `{|x : A; y: A|}` and `f u = {|x := y u; y:= x u|}`, we have `f (f u)` $\equiv$ `u`

- Primitive projections for records:
  - The η rule for records: two instance of a record type are *definitionally* equal if all their respective projections are
  - E.g., for {|`x : A; y: A`|} and `f u` = {|`x := y u; y:= x u`|}, we have `f (f u)` ≡ `u`
  - This provides *definitional* equalities, e.g.: (similar to Coq/HoTT implementation)
    - For Categories: `(C^op)^op` ≡ `C`
    - For Functors: `(F^op)^op` ≡ `F`
    - For Natural Transformations: `(N^op)^op` ≡ `N`

- Universe polymorphism: for relative smallness/largeness

- Universe polymorphism: for relative smallness/largeness

```
Class Category : Type@{max(i+1, j+1)} :=
{
  Obj : Type@{i}
  Hom : Obj → Obj → Type@{j}
  id : forall a : Obj, Hom a a
  compose : forall a b c, (f : Hom a b) (g : Hom c d) : Hom a c
  ⋮
}
```

- Universe polymorphism: for relative smallness/largeness

```
Class Category : Type@{max(i+1, j+1)} :=
{
  Obj : Type@{i}
  Hom : Obj → Obj → Type@{j}
  id : forall a : Obj, Hom a a
  compose : forall a b c, (f : Hom a b) (g : Hom c d) : Hom a c
  ⋮
}
```

- Category is universe polymorphic

- Universe polymorphism: for relative smallness/largeness

```
Class Category : Type@{max(i+1, j+1)} :=
{
  Obj : Type@{i}
  Hom : Obj → Obj → Type@{j}
  id : forall a : Obj, Hom a a
  compose : forall a b c, (f : Hom a b) (g : Hom c d) : Hom a c
  ⋮
}
```

- Category is universe polymorphic
- For each pair of levels (m, n), `Category@{m, n}` is a copy at level (m, n)

- Universe polymorphism: for relative smallness/largeness

```
Class Category : Type@{max(i+1, j+1)} :=
{
  Obj : Type@{i}
  Hom : Obj → Obj → Type@{j}
  id : forall a : Obj, Hom a a
  compose : forall a b c, (f : Hom a b) (g : Hom c d) : Hom a c
  ⋮
}
```

- Category is universe polymorphic
- For each pair of levels (m, n), `Category@{m, n}` is a copy at level (m, n)
- Universe levels in definitions and theorems are inferred by Coq and never appear in the source code

- Universe polymorphism: for relative smallness/largeness

```
Class Category : Type@{max(i+1, j+1)} :=
{
  Obj : Type@{i}
  Hom : Obj → Obj → Type@{j}
  id : forall a : Obj, Hom a a
  compose : forall a b c, (f : Hom a b) (g : Hom c d) : Hom a c
  ⋮
}
```

- Category is universe polymorphic
- For each pair of levels (m, n), `Category@{m, n}` is a copy at level (m, n)
- Universe levels in definitions and theorems are inferred by Coq and never appear in the source code
- For each definition, theorem, etc., we get some constraints on universe levels
- The definition, theorem, etc. only works for those copies that satisfy the side constraint

- This notion of smallness/largeness using universe levels works so well that we can define **Cat**:
- `Instance` **Cat** : `Category` := {`Obj` := `Category`; `Hom` := `Functor`; ...}

- This notion of smallness/largeness using universe levels works so well that we can define **Cat**:
- `Instance` **Cat** : `Category` := {`Obj` := `Category`; `Hom` := `Functor`; ...}
- Or prove the following:

```
Theorem Complete_Preorder (C : Category) (CC : Complete C) :
  forall x y : (Obj C), Hom x y' ≃ ((Arrow C) → Hom x y)
```

- This notion of smallness/largeness using universe levels works so well that we can define **Cat**:
- `Instance` **Cat** : `Category` := {`Obj` := `Category`; `Hom` := `Functor`; ...}
- Or prove the following:

```
Theorem Complete_Preorder (C : Category) (CC : Complete C) :
   forall x y : (Obj C), Hom x y' ≃ ((Arrow C) → Hom x y)
```

- This theorem results in a contradiction as soon as there are objects a and b in C such that $|hom(a, b)| \geq 2$

- This notion of smallness/largeness using universe levels works so well that we can define **Cat**:
- `Instance` **Cat** : `Category` := {`Obj` := `Category`; `Hom` := `Functor`; ...}
- Or prove the following:

```
Theorem Complete_Preorder (C : Category) (CC : Complete C) :
   forall x y : (Obj C), Hom x y' ≃ ((Arrow C) → Hom x y)
```

- This theorem results in a contradiction as soon as there are objects a and b in C such that $|hom(a, b)| \geq 2$
- In fact, this theorem holds only for small categories

- This notion of smallness/largeness using universe levels works so well that we can define **Cat**:
- `Instance` **Cat** : `Category` := {`Obj` := `Category`; `Hom` := `Functor`; ...}
- Or prove the following:

```
Theorem Complete_Preorder (C : Category) (CC : Complete C) :
  forall x y : (Obj C), Hom x y' ≃ ((Arrow C) → Hom x y)
```

- This theorem results in a contradiction as soon as there are objects a and b in C such that $|hom(a,b)| \geq 2$
- In fact, this theorem holds only for small categories
- This can be seen in universe constraints of this theorem

- This notion of smallness/largeness using universe levels works so well that we can define **Cat**:
- `Instance` **Cat** : `Category` := {`Obj` := `Category`; `Hom` := `Functor`; ...}
- Or prove the following:

```
Theorem Complete_Preorder (C : Category) (CC : Complete C) :
   forall x y : (Obj C), Hom x y' ≃ ((Arrow C) → Hom x y)
```

- This theorem results in a contradiction as soon as there are objects a and b in C such that $|hom(a, b)| \geq 2$
- In fact, this theorem holds only for small categories
- This can be seen in universe constraints of this theorem
  - For `C : Category@{k, l}` we get the restriction that `k` ≤ `l`
  - This is in contradiction with the fact that **Set** : `Category@{m, n}` with `m` > `n`

- This notion of smallness/largeness using universe levels works so well that we can define **Cat**:
- `Instance` **Cat** : `Category` := {`Obj` := `Category`; `Hom` := `Functor`; ...}
- Or prove the following:

```
Theorem Complete_Preorder (C : Category) (CC : Complete C) :
   forall x y : (Obj C), Hom x y' ≃ ((Arrow C) → Hom x y)
```

- This theorem results in a contradiction as soon as there are objects a and b in C such that $|hom(a, b)| \geq 2$
- In fact, this theorem holds only for small categories
- This can be seen in universe constraints of this theorem
  - For `C : Category@{k, l}` we get the restriction that `k` ≤ `l`
  - This is in contradiction with the fact that **Set** : `Category@{m, n}` with `m` > `n`

```
Set@{m, n} :=
{|
  Obj := Type@{n} : Type@{m};
  Hom := fun A B ⇒ A → B : Obj → Obj → Type@{n}; ...
|} : Category@{m, n}
```

- **Cat** in Coq:

`Instance` **Cat** : `Category@{i, j}` := {`Obj` := `Category@{k, l}`; `Hom` := `Functor`; ...}

- **Cat** in Coq:

  `Instance` **Cat** : `Category@{i, j}` := {`Obj` := `Category@{k, l}`; `Hom` := `Functor`; ...}

- But according to Coq's universe polymorphism, if `C : Category@{k, l}` and `C : Category@{k', l'}`, we must have `k` = `k'` and `l` = `l'`

- **Cat** in Coq:

  `Instance` **Cat** : `Category@{i, j} := {Obj := Category@{k, l}; Hom := Functor; ...}`

- But according to Coq's universe polymorphism, if `C : Category@{k, l}` and `C : Category@{k', l'}`, we must have `k = k'` and `l = l'`
- This means **Cat**`@{i, j, k, l}` is not the category of all categories at level (k, l) or lower but *only* at level (k, l)

- **Cat** in Coq:

  `Instance` **Cat** : `Category@{i, j}` := {`Obj` := `Category@{k, l}`; `Hom` := `Functor`; ...}

- But according to Coq's universe polymorphism, if `C : Category@{k, l}` and `C : Category@{k', l'}`, we must have `k = k'` and `l = l'`
- This means **Cat**`@{i, j, k, l}` is not the category of all categories at level (k, l) or lower but *only* at level (k, l)
- We can lift category:

```
lift (C : Category@{k, l}) : Category@{k', l'} :=
{|
Obj := Obj C;
Hom := Hom C;
      ⋮
|}
```

  for $k < k'$ and $l < l'$

- **Cat** in Coq:

  `Instance` **Cat** : `Category@{i, j} := {Obj := Category@{k, l}; Hom := Functor; ...}`

- But according to Coq's universe polymorphism, if `C : Category@{k, l}` and `C : Category@{k', l'}`, we must have `k = k'` and `l = l'`
- This means **Cat**`@{i, j, k, l}` is not the category of all categories at level (k, l) or lower but *only* at level (k, l)
- We can lift category:

```
lift (C : Category@{k, l}) : Category@{k', l'} :=
{|
Obj := Obj C;
Hom := Hom C;
      ⋮
|}
```

  for $k < k'$ and $l < l'$
- But
  - We can't prove or even specify (universe inconsistency)
    `forall (C : Category), C = lift C`

- **Cat** in Coq:

  ```
  Instance Cat : Category@{i, j} := {Obj := Category@{k, l}; Hom := Functor; ...}
  ```

- But according to Coq's universe polymorphism, if `C : Category@{k, l}` and `C : Category@{k', l'}`, we must have `k = k'` and `l = l'`
- This means **Cat**`@{i, j, k, l}` is not the category of all categories at level (k, l) or lower but *only* at level (k, l)
- We can lift category:

  ```
  lift (C : Category@{k, l}) : Category@{k', l'} :=
  {|
  Obj := Obj C;
  Hom := Hom C;
        ⋮
  |}
  ```

  for $k < k'$ and $l < l'$
- But
  - We can't prove or even specify (universe inconsistency) `forall (C : Category), C = lift C`
  - We can't prove `forall (C : Category), JMeq C (lift C)`

- **Cat** in Coq:

  `Instance` **Cat** : `Category@{i, j}` := {`Obj` := `Category@{k, l}`; `Hom` := `Functor`; ...}

- But according to Coq's universe polymorphism, if `C : Category@{k, l}` and `C : Category@{k', l'}`, we must have `k = k'` and `l = l'`
- This means **Cat**`@{i, j, k, l}` is not the category of all categories at level (k, l) or lower but *only* at level (k, l)
- We can lift category:

```
lift (C : Category@{k, l}) : Category@{k', l'} :=
{|
Obj := Obj C;
Hom := Hom C;
      ⋮
|}
```

  for $k < k'$ and $l < l'$

- But
  - We can't prove or even specify (universe inconsistency) `forall (C : Category), C = lift C`
  - We can't prove `forall (C : Category), JMeq C (lift C)`
  - The equality `forall (C : Category), lift C = lift (lift C)` is not definitional

- If we show that **Cat**`@{i, j, k, l}` has exponentials, we get the constraints that $j = k = l$

- If we show that **Cat**`@{i, j, k, l}` has exponentials, we get the constraints that $j = k = l$
- Therefore, no copy of **Set** is in a copy of **Cat** in which we have exponentials

- If we show that **Cat**`@{i, j, k, l}` has exponentials, we get the constraints that $j = k = l$
- Therefore, no copy of **Set** is in a copy of **Cat** in which we have exponentials
- That means we can't define Yoneda embedding as exponential transpose (currying) of the hom functor

- If we show that **Cat**`@{i, j, k, l}` has exponentials, we get the constraints that $j = k = l$
- Therefore, no copy of **Set** is in a copy of **Cat** in which we have exponentials
- That means we can't define Yoneda embedding as exponential transpose (currying) of the hom functor
- Defining Yoneda separately, it still can only be applied in a category `C : Category@{i, j}` if $i = j$.

- If we show that **Cat**`@{i, j, k, l}` has exponentials, we get the constraints that $j = k = l$
- Therefore, no copy of **Set** is in a copy of **Cat** in which we have exponentials
- That means we can't define Yoneda embedding as exponential transpose (currying) of the hom functor
- Defining Yoneda separately, it still can only be applied in a category `C : Category@{i, j}` if $i = j$.
- We can use Yoneda to prove that in any cartesian closed category:

$$(a^b)^c \simeq a^{b \times c}$$

  but this lemma can't be applied to **Cat** or **Set**

- Consider our proof of uniqueness of adjoint functors (up to natural isomorphism)

[1] for $f : a \times b \to c$ we have $curry(f) : a \to c^b$

- Consider our proof of uniqueness of adjoint functors (up to natural isomorphism) Assume for $F, F' : C \to D : G$, we have $F \dashv G$ and $F' \dashv G$, i.e.,

$$hom_D(F, -) \simeq hom_C(-, G) \text{ and } hom_D(F', -) \simeq hom_C(-, G)$$

[1] for $f : a \times b \to c$ we have $curry(f) : a \to c^b$

- Consider our proof of uniqueness of adjoint functors (up to natural isomorphism) Assume for $F, F' : C \to D : G$, we have $F \dashv G$ and $F' \dashv G$, i.e.,

$$hom_D(F, -) \simeq hom_C(-, G) \text{ and } hom_D(F', -) \simeq hom_C(-, G)$$

Thus we have:

$$hom_D(F, -) \simeq hom_D(F', -)$$

[1] for $f : a \times b \to c$ we have $curry(f) : a \to c^b$

- Consider our proof of uniqueness of adjoint functors (up to natural isomorphism) Assume for $F, F' : C \to D : G$, we have $F \dashv G$ and $F' \dashv G$, i.e.,

$$hom_D(F, -) \simeq hom_C(-, G) \text{ and } hom_D(F', -) \simeq hom_C(-, G)$$

Thus we have:

$$hom_D(F, -) \simeq hom_D(F', -)$$

but for $H, H' : C \times C' \to D$, $H \simeq H'$ *iff* $curry(H) \simeq curry(H')$[1]

[1] for $f : a \times b \to c$ we have $curry(f) : a \to c^b$

- Consider our proof of uniqueness of adjoint functors (up to natural isomorphism) Assume for $F, F' : C \to D : G$, we have $F \dashv G$ and $F' \dashv G$, i.e.,

$$hom_D(F, -) \simeq hom_C(-, G) \text{ and } hom_D(F', -) \simeq hom_C(-, G)$$

Thus we have:

$$hom_D(F, -) \simeq hom_D(F', -)$$

but for $H, H' : C \times C' \to D$, $H \simeq H'$ *iff* $curry(H) \simeq curry(H')$[1]

so, we can assume:

$$curry(hom_D(F, -)) \simeq curry(hom_D(F', -))$$

[1] for $f : a \times b \to c$ we have $curry(f) : a \to c^b$

- Consider our proof of uniqueness of adjoint functors (up to natural isomorphism) Assume for $F, F' : C \to D : G$, we have $F \dashv G$ and $F' \dashv G$, i.e.,

$$hom_D(F, -) \simeq hom_C(-, G) \text{ and } hom_D(F', -) \simeq hom_C(-, G)$$

Thus we have:

$$hom_D(F, -) \simeq hom_D(F', -)$$

but for $H, H' : C \times C' \to D$, $H \simeq H'$ *iff* $curry(H) \simeq curry(H')$[1]
so, we can assume:

$$curry(hom_D(F, -)) \simeq curry(hom_D(F', -))$$

But according to axioms of exponentials we have

$$curry(hom_D(F, -)) = F \circ curry(hom_D)$$

[1] for $f : a \times b \to c$ we have $curry(f) : a \to c^b$

- Consider our proof of uniqueness of adjoint functors (up to natural isomorphism) Assume for $F, F' : C \to D : G$, we have $F \dashv G$ and $F' \dashv G$, i.e.,

$$hom_D(F, -) \simeq hom_C(-, G) \text{ and } hom_D(F', -) \simeq hom_C(-, G)$$

Thus we have:

$$hom_D(F, -) \simeq hom_D(F', -)$$

but for $H, H' : C \times C' \to D$, $H \simeq H'$ *iff* $curry(H) \simeq curry(H')$[1]
so, we can assume:

$$curry(hom_D(F, -)) \simeq curry(hom_D(F', -))$$

But according to axioms of exponentials we have

$$curry(hom_D(F, -)) = F \circ curry(hom_D)$$

Which means:

$$F \circ Y_D \simeq F' \circ Y_D$$

[1] for $f : a \times b \to c$ we have $curry(f) : a \to c^b$

- Consider our proof of uniqueness of adjoint functors (up to natural isomorphism) Assume for $F, F' : C \to D : G$, we have $F \dashv G$ and $F' \dashv G$, i.e.,

$$hom_D(F, -) \simeq hom_C(-, G) \text{ and } hom_D(F', -) \simeq hom_C(-, G)$$

Thus we have:

$$hom_D(F, -) \simeq hom_D(F', -)$$

but for $H, H' : C \times C' \to D$, $H \simeq H'$ *iff* $curry(H) \simeq curry(H')$[1]
so, we can assume:

$$curry(hom_D(F, -)) \simeq curry(hom_D(F', -))$$

But according to axioms of exponentials we have

$$curry(hom_D(F, -)) = F \circ curry(hom_D)$$

Which means:

$$F \circ Y_D \simeq F' \circ Y_D$$

This immediately gives $F \simeq F'$ as Y_D (the Yoneda embedding for D) is an embedding

[1] for $f : a \times b \to c$ we have $curry(f) : a \to c^b$

- Consider our proof of uniqueness of adjoint functors (up to natural isomorphism) Assume for $F, F' : C \to D : G$, we have $F \dashv G$ and $F' \dashv G$, i.e.,

$$hom_D(F, -) \simeq hom_C(-, G) \text{ and } hom_D(F', -) \simeq hom_C(-, G)$$

Thus we have:

$$hom_D(F, -) \simeq hom_D(F', -)$$

but for $H, H' : C \times C' \to D$, $H \simeq H'$ *iff* $curry(H) \simeq curry(H')$[1]
so, we can assume:

$$curry(hom_D(F, -)) \simeq curry(hom_D(F', -))$$

But according to axioms of exponentials we have

$$curry(hom_D(F, -)) = F \circ curry(hom_D)$$

Which means:

$$F \circ Y_D \simeq F' \circ Y_D$$

This immediately gives $F \simeq F'$ as Y_D (the Yoneda embedding for D) is an embedding

- But, we can't use the general fact above, as it involves both exponentials and `Set` (through *hom*) in `Cat`

[1] for $f : a \times b \to c$ we have $curry(f) : a \to c^b$

- Consider our proof of uniqueness of adjoint functors (up to natural isomorphism) Assume for $F, F' : C \to D : G$, we have $F \dashv G$ and $F' \dashv G$, i.e.,

$$hom_D(F, -) \simeq hom_C(-, G) \text{ and } hom_D(F', -) \simeq hom_C(-, G)$$

Thus we have:

$$hom_D(F, -) \simeq hom_D(F', -)$$

but for $H, H' : C \times C' \to D$, $H \simeq H'$ *iff* $curry(H) \simeq curry(H')$[1]

so, we can assume:

$$curry(hom_D(F, -)) \simeq curry(hom_D(F', -))$$

But according to axioms of exponentials we have

$$curry(hom_D(F, -)) = F \circ curry(hom_D)$$

Which means:

$$F \circ Y_D \simeq F' \circ Y_D$$

This immediately gives $F \simeq F'$ as Y_D (the Yoneda embedding for D) is an embedding

- But, we can't use the general fact above, as it involves both exponentials and `Set` (through *hom*) in `Cat` – we have proven a separate instance of this fact for `Cat`

[1] for $f : a \times b \to c$ we have $curry(f) : a \to c^b$

- Another issue that we faced is that `Set` seems to have a special place:
  - If we show that **Set** : `Category@{i, j}` has `unit` : `Set` as the terminal object, we get the restriction $j =$ `Set`

- Another issue that we faced is that `Set` seems to have a special place:
  - If we show that **Set** : `Category@{i, j}` has `unit` : `Set` as the terminal object, we get the restriction $j =$ `Set`
  - The problem occurs when we want to show that `Prop` is the subobject classifier for **Set**. As then we need a monic arrow:

$$tr : unit \rightarrow \texttt{Prop}$$

  and `unit` → `Prop` is not a term of type `Set`

- Another issue that we faced is that `Set` seems to have a special place:
  - If we show that **Set** : `Category@{i, j}` has **`unit`** : `Set` as the terminal object, we get the restriction $j =$ `Set`
  - The problem occurs when we want to show that `Prop` is the subobject classifier for **Set**. As then we need a monic arrow:

    $$tr : unit \to \texttt{Prop}$$

    and **`unit`** $\to$ `Prop` is not a term of type `Set`
  - This can be solved by defining a singleton inductive type at a level *strictly* higher than `Set`

- Another issue that we faced is that `Set` seems to have a special place:
  - If we show that **Set** : `Category@{i, j}` has **`unit`** : `Set` as the terminal object, we get the restriction $j =$ `Set`
  - The problem occurs when we want to show that `Prop` is the subobject classifier for **Set**. As then we need a monic arrow:

    $$tr : unit \to \texttt{Prop}$$

    and **`unit`** $\to$ `Prop` is not a term of type `Set`
  - This can be solved by defining a singleton inductive type at a level *strictly* higher than `Set`
  - But, that would cause a problem for the part where we show that type **`nat`** : `Set` of the library of Coq is the initial algebra for $T(X) = 1 + X$ in category **Set**

- Another issue that we faced is that `Set` seems to have a special place:
  - If we show that **Set** : `Category@{i, j}` has **`unit`** : `Set` as the terminal object, we get the restriction $j =$ `Set`
  - The problem occurs when we want to show that `Prop` is the subobject classifier for **Set**. As then we need a monic arrow:

    $$tr : unit \to \texttt{Prop}$$

    and **`unit`** $\to$ `Prop` is not a term of type `Set`
  - This can be solved by defining a singleton inductive type at a level *strictly* higher than `Set`
  - But, that would cause a problem for the part where we show that type **`nat`** : `Set` of the library of Coq is the initial algebra for $T(X) = 1 + X$ in category **Set**
  - We therefore postulate existence of a universe polymorphic singleton type:

```
Parameter UNIT : Type.
Parameter TT : UNIT.
Axiom UNIT_SINGLETON : forall x y : UNIT, x = y.
```

- Conclusion:
  - We presented an implementation of category theory covering some of the basic category theory

- Conclusion:
  - We presented an implementation of category theory covering some of the basic category theory
  - We use features of Coq 8.5: primitive projections and universe polymorphism

- Conclusion:
  - We presented an implementation of category theory covering some of the basic category theory
  - We use features of Coq 8.5: primitive projections and universe polymorphism
  - Universe polymorphism to represent smallness/largeness

- Conclusion:
  - We presented an implementation of category theory covering some of the basic category theory
  - We use features of Coq 8.5: primitive projections and universe polymorphism
  - Universe polymorphism to represent smallness/largeness
    - This works well to a degree that we don't need to mention any universe levels and can prove things like: **Cat** and `Complete_Preorder`

- Conclusion:
  - We presented an implementation of category theory covering some of the basic category theory
  - We use features of Coq 8.5: primitive projections and universe polymorphism
  - Universe polymorphism to represent smallness/largeness
    - This works well to a degree that we don't need to mention any universe levels and can prove things like: **Cat** and `Complete_Preorder`
    - It also has shortcomings: e.g., can't use Yoneda in **Cat** and **Set**