```
(*

   A formalization of Aczel's model for CZF
   in Coq by Olov Wilander and Erik Palmgren.
   Version March 2012.

   Other known formalizations of this model
   in proof assistants:

   N.P. Mendler 1990 in LEGO
   M. Takeyama mid 1990s in Agda 1


   This file requires UTF8


*)

Require Import PropasTypesUtf8Notation
PropasTypesBasics_mod.

Require Import SwedishSetoids_mod.

Delimit Scope czf_scope with czf.
Open Scope czf_scope.

(* The type of well-founded trees, that is the
universe of CZF sets in the constructed model.*)

Inductive V: Type := sup (A: Set) (f: A → V).

Definition iV (s: V): Set
  := match s with sup A f => A end.
Coercion iV : V >-> Sortclass.

Definition pV (s: V): s → V
```

```
  := match s with sup A f => f end.
Coercion pV : V >-> Funclass.

(* The equality relation is the least bisimulation on
V *)

Reserved Notation "x ≐ y" (at level 70, no
associativity).

Fixpoint eqV (x y: V): Set :=
  (∀α: x, ∃β: y, x α ≐ y β) ∧ (∀β:y, ∃α: x, x α ≐ y
β)
  where "x ≐ y" := (eqV x y): czf_scope.

Lemma eqVsplit {x y: V}: (∀α: x, ∃β: y, x α ≐ y β) →
(∀β: y, ∃α: x, x α ≐ y β) → x ≐ y.
Proof.
  destruct x; intros; split; assumption.
Defined.

Ltac eqVsplit := apply eqVsplit.

Ltac eqVassumption_canonical P0 P1 := intros [P0 P1];
fold eqV in P0, P1; simpl in P0, P1.

Lemma eqVdestruct {x y: V}: x ≐ y → (∀α: x, ∃β: y, x
α ≐ y β) ∧ (∀β: y, ∃α: x, x α ≐ y β).
Proof.
  destruct x; eqVassumption_canonical H H'; split;
assumption.
Defined.

Ltac eqVassumption P0 P1 := let P := fresh in
  intros P; apply eqVdestruct in P; destruct P as [P0
P1]; simpl in P0, P1.

(* The first thing to do is to prove that this is an
```

```
equivalence relation. *)

Theorem refV: ∀x, x ≐ x.
Proof.
  intro x; induction x.
  split; [intro x; exists x; trivial ..].
Qed.

Theorem symV: ∀x y, x ≐ y → y ≐ x.
Proof.
  intro x; induction x as [ix fx IH]. intro y.
  eqVassumption P0 P1; eqVsplit;
  match goal with [hyp: ∀_: ?a, ∃_: ?b, _ |- ∀_: ?a,
∃_: ?b, _] =>
    let x := fresh in let y := fresh in
      intro x; destruct hyp with x as [y]; exists y;
auto
  end.
Qed.

Theorem traV: ∀x y z, x ≐ y → y ≐ z → x ≐ z.
Proof.
  intros x; induction x as [ix fx IH]. intros y z.
  eqVassumption P0 P1; eqVassumption Q0 Q1; eqVsplit;
  match goal with [hyp0: ∀_: ?a, ∃b0: ?b, _, hyp1:
∀b1: ?b, ∃c0: ?c, _ |- ∀a1: ?a, ∃c1: ?c, _] =>
    let a := fresh in let b := fresh in let c :=
fresh in
      intro a; destruct hyp0 with a as [b]; destruct
hyp1 with b as [c]; exists c; eauto
  end.
Qed.

Hint Resolve refV : czf.
Hint Immediate symV : czf.
Hint Resolve traV : czf. (* The warning here is
expected *)
```

```
Ltac czf := simpl; eauto with czf.

(* Next, define the membership relation... *)

Definition memV (x y: V): Set
  := ∃idx: y, x ≐ y idx.
Infix "∈" := memV (at level 70, no associativity) :
czf_scope.
Notation "x ∉ y" := (¬(x ∈ y)) (at level 70) :
czf_scope.

Lemma membership (x y: V) (a: y): x ≐ y a → x ∈ y.
Proof.
  intro; exists a; auto.
Defined.
Hint Resolve membership : czf.

(* ... and prove that it respects the equality
relation introduced earlier. *)

Lemma ext1: ∀x y z, x ≐ y → y ∈ z → x ∈ z.
Proof.
  intros x y z H [idx eq]. czf.
Qed.

Lemma ext2: ∀x y z, x ∈ y → y ≐ z → x ∈ z.
Proof.
  intros x y z [idx eq]. eqVassumption P0 P1; clear
P1.
  destruct P0 with idx as [idx' eq']. czf.
Qed.

(* The warnings here are expected *)

Hint Resolve ext1: czf.
Hint Resolve ext2: czf.
```

```
(* Introduce some more notation *)

Notation "x ⊆ y" := (∀z: V, z ∈ x → z ∈ y) (at level
60) : czf_scope.

(* Now start proving that the axioms hold *)

Lemma ext3: ∀x y, x ⊆ y → y ⊆ x → x ≐ y.
Proof.
  intros x y xsuby ysubx. eqVsplit;
  match goal with [hyp: ?a ⊆ ?b |- ∀_: iV ?a, ∃_: iV
?b, _] =>
    let α := fresh in let β := fresh in
        intro α; destruct hyp with (a α) as [β]; czf
  end.
Qed.

Hint Resolve ext3: czf.

Theorem Extensionality:
  ∀x, ∀y, (∀z, z ∈ x ↔ z ∈ y) → x ≐ y.
Proof.
  intros x y hyp. apply ext3; intro z; apply (hyp z).
Qed.

Definition pairV (x y: V): V
  := sup bool (λ b, match b with true => x | false =>
y end).
Notation "{{  x , y  }}" := (pairV x y)
  (at level 0, x, y at level 69) : czf_scope.

Lemma pairVL1: ∀x y, x ∈ {{ x, y }}.
Proof.
  intros; exists true; czf.
Qed.
```

```
Lemma pairVL2: ∀x y, y ∈ {{ x, y }}.
Proof.
  intros; exists false; czf.
Qed.

Hint Resolve pairVL1 pairVL2: czf.

Lemma pairingchar: ∀a b y, y ∈ {{ a,b }} ↔ y ≐ a ∨ y
≐ b.
Proof.
  intros a b y; split.
  intros [i e]. destruct i; auto.
  intros [eq | eq]; czf.
Qed.

Hint Resolve pairingchar : czf.

Theorem Pairing:
  ∀a b, ∃c, ∀y, y ∈ c ↔ y ≐ a ∨ y ≐ b.
Proof.
  czf.
(*  intros; setexists {{ a, b }}; apply pairingchar.
*)
Qed.

Lemma pairingcharR: ∀a b y, y ∈ {{ a, b }} → y ≐ a ∨
y ≐ b.
Proof.
  intros a b y; eapply pairingchar.
Qed.

Lemma pairingcharL: ∀a b y, y ≐ a ∨ y ≐ b → y ∈ {{ a,
b }}.
Proof.
  intros a b y; eapply pairingchar.
Qed.
```

```
Hint Resolve pairingcharR pairingcharL: czf.

Definition unionV (x: V): V
  := sup (∃a: x, x a) (λ u, x (projT1 u) (projT2 u)).
Notation "∪ X" := (unionV X) (at level 1) :
czf_scope.

Lemma unionLm: ∀s: V, ∀x: s, s x ∈ s.
Proof.
  czf.
(*  intros s x; exists x; apply refV. *)
Qed.

Lemma unionLm1:
  ∀s: V, ∀A: Set, ∀f: A → V, ∀x: A, s ∈ f x → s ∈
∪(sup A f).
Proof.
  intros s A f x [i eq].
  exists (existT _ x i); assumption.
Qed.

Hint Resolve unionLm1 : czf.

Lemma unionLm2:
  ∀r s t, r ∈ s → s ∈ t → r ∈ ∪t.
Proof.
  intros r s t rins [idx eq]. czf.
(*  intros r s [it ft | a'] rs [idx eq].
  apply unionLm1 with idx, ext2 with s;
[assumption..].
  destruct s as [i f | a]; contradiction. *)
Qed.

Hint Resolve unionLm2 : czf.

Lemma unionchar:
  ∀a y, y ∈ ∪a ↔ ∃x, x ∈ a ∧ y ∈ x.
```

```
Proof.
  intros a y; split. intros [[idx1 idx2] eq].
  exists (a idx1). split. czf. apply membership with
idx2; auto.
  intros [x h]. apply unionLm2 with x; tauto.
Qed.

Hint Resolve unionchar : czf.

Theorem Union: ∀a, ∃b, ∀y, y∈b ↔ ∃x, x ∈ a ∧ y ∈ x.
Proof.
  czf.
Qed.

Notation "s ∪ t" := (∪{{ s, t }}) (at level 65, right
associativity) : czf_scope.

Theorem binunionchar:
  ∀x s t, x ∈ s ∪ t ↔ x ∈ s ∨ x ∈ t.
Proof.
  intros x s t; split.
  intro xinbinu. apply unionchar in xinbinu; destruct
xinbinu as [y [p q]].
  apply pairingcharR in p. destruct p; czf.
  intros [mem | mem]; czf.
(*  intros [[i i'] eq].
  destruct i; [left | right]; [ exists i'; assumption
..].
  intros [[i eq] | [i eq]].
  exists (existT _ true i); assumption.
  exists (existT _ false i); assumption. *)
Qed.

Hint Resolve binunionchar : czf.

Lemma binunioncharL: ∀x s t, x ∈ s ∪ t → x ∈ s ∨ x ∈
t.
```

```
Proof.
  intros x s t [[idx1 idx2] e]. induction idx1; simpl
in *; czf.
Qed.

Lemma binunioncharR: ∀x s t, x ∈ s ∨ x ∈ t → x ∈ s ∪
t.
Proof.
  intros x s t [mem | mem]; czf.
Qed.

Hint Resolve binunioncharL binunioncharR : czf.

Definition emptyV : V
  := sup False (False_rect V).
Notation "∅" := emptyV : czf_scope.

Theorem EmptySet:
  ∀x, x ∉ ∅.
Proof.
  intros _ [[]].
Qed.

Hint Resolve EmptySet: czf.

Theorem EmptySetAxiom:
  ∃x, ∀y, y ∉ x.
Proof.
  czf.
(*  setexists ∅; apply EmptySet. *)
Qed.

Definition extensionalV (P: V → Type): Type
  := ∀x y, P x → x ≐ y → P y.
Notation "«  P »" := (extensionalV P) : czf_scope.

Theorem SetInduction (P: V → Type) (Pext: «P»):
```

```
  (∀x, (∀y, y∈x → P y) → P x) → ∀x, P x.
Proof.
  intro IH. induction x as [I f IH']; apply IH.
  intros y [i eq]. czf.
Qed.

Definition singletonV (x: V) := sup ⊤ (λ _, x).
Notation "{{  s  }}" := (singletonV s) (at level 0, s
at level 69) : czf_scope.

(* Sanity check lemma for this notation *)
Lemma singleton_equals_pair (x: V): {{ x }} ≐ {{ x, x
}}.
Proof.
  simpl; split.
  exists true; apply refV.
  induction β; repeat progress split; apply refV.
Qed.

Lemma singleton_char (x y: V): x ∈ {{ y }} → x ≐ y.
Proof.
  firstorder.
Qed.

Hint Resolve singleton_char : czf.

Notation "s ⁺" := (s ∪ {{ s }}) (at level 1) :
czf_scope.

Lemma succL1:
  ∀s t:V, s ∈ t⁺ → s ≐ t ∨ s ∈ t.
Proof.
  intros s t H. apply binunioncharL in H. destruct H;
czf.
Qed.

Lemma succL2:
```

```
  ∀s t:V, s ≐ t ∨ s ∈ t → s ∈ t⁺.
Proof.
  intros s t [eq | mem].
  exists (existT _ false tt). trivial. czf.
Qed.

Hint Resolve succL1 succL2.

Fixpoint numeralV (n: ℕ): V
  := match n with
       | O   => ∅
       | S m => (numeralV m)⁺
     end.
Notation "⌜ n ⌝" := (numeralV n) : czf_scope.

Definition ω: V
  := sup ℕ numeralV.

(* Notation "'ω'" := (natV) : czf_scope. *)

Notation ttve x := (∀y z, z ∈ y → y ∈ x → z ∈ x).

Theorem numeralsaretransitive:
  ∀n: ℕ, ttve(⌜n⌝).
Proof.
  induction n.
  intros y z _ mem; contradiction EmptySet with y.
  intros y z mem1 mem2. apply succL1 in mem2;
destruct mem2; czf.
Qed.

Lemma eltnat:
  ∀n: ℕ, ∀x, x ∈ ⌜n⌝ → ∃m: ℕ, x ≐ ⌜m⌝.
Proof.
  induction n.
  intros _ [[]].
  intros x mem. apply succL1 in mem. firstorder.
```

```
Qed.

Lemma omegattve: ttve(ω).
Proof.
  intros z y yinz (n, eq).
  apply (eltnat n), ext2 with z; assumption.
Qed.

Lemma succext: ∀x y, x ≐ y → x⁺ ≐ y⁺.
Proof.
  intros x y eq. apply ext3;
  intros z mem; apply succL1 in mem; destruct mem;
czf.
Qed.

Theorem Infinity:
  ∃x, ∅ ∈ x ∧ ∀y, y ∈ x → y⁺ ∈ x.
Proof.
  exists ω. split. exists 0. czf.
  intros y [i e]. exists (S i). eapply traV. apply
succext, e. czf.
Qed.

Definition sepV (s: V) (P: V → Set)
  := sup (∃a: s, P (s a)) (λ u, s (projT1 u)).
Notation "{{  x ∈ s | P  }}" := (sepV s (fun x => P))
  (at level 0, x, s, P at level 69) : czf_scope.

Lemma separationchar (P: V → Set) (Pext: «P»):
  ∀s x, x ∈ {{ y ∈ s | P y }} ↔ x ∈ s ∧ P x.
Proof.
  intros s x; split.
  intros [[i e] p]; split. exists i; assumption. czf.
  intros ((i, eq), pf).
  apply (Pext x (s i)) in pf; [ | apply eq].
  exists (existT (fun u => P (s u)) i pf);
assumption.
```

```
Qed.

Theorem Separation (P: V → Set) (Pext: «P»):
  ∀s, ∃t, ∀x, x∈t ↔ x ∈ s ∧ P x.
Proof.
  intros; exists ({{ x ∈ s | P x }}).
  apply separationchar; assumption.
Qed.

Lemma sep_sub (P: V → Set) (Pext: «P»): ∀x, {{ y ∈ x
| P y }} ⊆ x.
Proof.
  intros x z [[idx pf] eq]. exists idx; assumption.
Qed.

Notation "∀ x ∈ s , P" := (∀x: iV s, (fun x: V => P)
(pV s x))
  (at level 200, x at level 0).
Notation "∃ x ∈ s , P" := (∃x: iV s, (fun x: V => P)
(pV s x))
  (at level 200, x at level 0).

Lemma Ballright:
  ∀a, ∀P: V → Set, «P» → ((∀x ∈ a, P x) ↔ ∀x, x ∈ a
→ P x).
Proof.
  intros a P E; split.
  intros bdd x (i, eq).
  apply E with (a i), symV, eq; apply bdd.
  intros full i; apply full, unionLm.
Qed.

Lemma Bexright:
  ∀a, ∀P: V → Set, «P» → ((∃x ∈ a, P x) ↔ ∃x: V, x ∈
a ∧ P x).
Proof.
  intros a P E; split.
```

```
  intros a P E; split.
  intros (i, pf);
    exists (a i); split; [apply unionLm |
assumption].
  intros (x, ((i, eq), pf)).
  exists i.
  apply E with x; [assumption..].
Qed.

Lemma SetInductionBdd (P: V → Set) (Pext: «P»):
  (∀x, (∀y ∈ x, P y) → P x) → ∀x, P x.
Proof.
  intros IH x; induction x; czf.
Qed.

Notation totalV a b R := (∀x ∈ a, ∃y ∈ b, R x y).
Notation bitotalV a b R
  := ((totalV a b R) ∧ (totalV b a (λ x y, R y x))).

Theorem StrongCollection:
  ∀R: V → V → Type,
  ∀a, (∀x ∈ a, ∃y, R x y) → ∃b, bitotalV a b R.
Proof.
  intros r a bdd. destruct (TTAC a V _ bdd) as [f p].
  exists (sup a f). czf.
Qed.

Definition sscV (a b: V): V
  := (sup (a → b) (λ f, sup a (λ x, b (f x)))).

Theorem SubsetCollection:
  ∀Q: V → V → V → Type,
  ∀a b, ∃c, ∀u,
    ((∀x ∈ a, ∃y ∈ b, Q x y u) →
      ∃d ∈ c, ((∀x ∈ a, ∃y ∈ d, Q x y u) ∧ (∀y ∈ d, ∃x
∈ a, Q x y u))).
Proof.
  intros Q a b. exists (sscV a b). intros u h.
```

```
destruct (TTAC a b _ h). czf.
Qed.

(* Now some more purely set theoretical work *)

Fixpoint TC (x: V): V :=
  match x with sup A f => x ∪ ∪(sup A (λ i, TC (f
i))) end.

Lemma TClemma: ∀x y, x ∈ y → x ∈ TC y.
Proof.
  intros x [I f] [i e]. simpl in *.
  unfold unionV; apply (sigrejig bool). exists true;
exists i; assumption.
Qed.

Lemma TC_ttve (x: V): ttve(TC x).
Proof.
  induction x as [A f IH]; intros x y mem mem2.
  apply binunioncharL in mem2; destruct mem2 as [[i
e] | [[i1 i2] e2]].
  apply binunioncharR; right. unfold unionV; apply
(sigrejig A).
  exists i. cut (y ∈ TC (f i)); trivial. apply
TClemma. apply ext2 with x; assumption.
  apply binunioncharR; right. unfold unionV; apply
(sigrejig A).
  exists i1. apply IH with x. assumption. exists
i2;  assumption.
Qed.

(* Now start developing basic mathematics inside the
model *)

Notation " 〈  x , y 〉 " := ({{ {{ x }}, {{ x, y }}
}}).
```

```
Lemma pairing_injective (x y z w: V): ⟨x, z⟩ ≐ ⟨y,
w⟩ → x ≐ y ∧ z ≐ w.
Proof.
  intros [eq1 eq2]; fold eqV in *. simpl in *.
  Local Ltac boolcases :=
    repeat match goal with [hyp: ∀_: bool, _ |- _] =>
             destruct (hyp true); destruct (hyp
false); clear hyp
           end.
  Local Ltac boolbranch := repeat match goal with
[hyp: bool |- _] => induction hyp end.
  Local Ltac dedup :=
    repeat match goal with [hyp: ⊤ |- _] => destruct
hyp end;
    repeat match goal with [hyp0: ?a ≐ ?b, hyp1: ?a ≐
?b |- _] => clear hyp1 end;
    repeat match goal with [hyp0: ?a ≐ ?b, hyp1: ?b ≐
?a |- _] => clear hyp1 end.
  Local Ltac desingleton :=
    repeat match goal with [hyp: {{ _ }} ≐ {{ _ }} |-
_] =>
             let P := fresh in
               destruct hyp as [P _]; fold eqV in P;
specialize P with tt; simpl in P; destruct P
           end.
  Local Ltac singlepaireq :=
    repeat match goal with [hyp: {{ _, _ }} ≐ {{ _ }}
|- _] => apply symV in hyp end;
    repeat match goal with [hyp: {{ _ }} ≐ {{ _, _ }}
|- _] =>
             let eq1 := fresh in destruct hyp as [_
eq1]; fold eqV in eq1; simpl in eq1; boolcases
           end.
  Local Ltac pairpaireq :=
    repeat match goal with [hyp: {{ _, _ }} ≐ {{ _, _
}} |- _] =>
             let e1 := fresh in let e2 := fresh in
```

```
                destruct hyp as [e1 e2]; simpl in e1,
e2; fold eqV in e1, e2; boolcases; boolbranch
            end.
  Local Ltac finish := eauto using symV, traV.
  Time boolcases; boolbranch; dedup; desingleton;
singlepaireq; pairpaireq; dedup; finish.
Defined.

Lemma pairing_extensional (x y z w: V): x ≐ z → y ≐ w
→ ⟨x, y⟩ ≐ ⟨z, w⟩ .
Proof.
  intros xeqz yeqw.
  split; intro a; exists a.
  destruct a. repeat (split; simpl); assumption.
  split; intro a; exists a; destruct a; assumption.
  destruct a. repeat (split; simpl); assumption.
  split; intro a; exists a; destruct a; assumption.
Defined.

Definition prodV (x y: V): V
  := sup (x ∧ y) (fun p => let (α, β) := p in ⟨x α,
y β⟩ ).
Notation "x × y" := (prodV x y) (at level 40).

Lemma productchar (x y: V): ∀z, z ∈ x × y ↔ ∃α ∈ x,
∃β ∈ y, z ≐ ⟨α, β⟩ .
Proof.
  intro z; split.
  intros [[α β] eq]; simpl in eq. exists α. exists β.
assumption.
  intros [α [β eq]]; exists (α, β); assumption.
Qed.

Lemma productchar_altproof (x y: V): ∀z, z ∈ x × y ↔
∃α ∈ x, ∃β ∈ y, z ≐ ⟨α, β⟩ .
proof.
```

```
Let z:V.
focus on (z ∈ x × y → ∃α ∈ x, ∃β ∈ y, z ≐ ⟨α, β⟩ ).
  given index such that eq: (z ≐ (x × y) index).
  claim (z ≐ (x × y) index → ∃α ∈ x, ∃β ∈ y, z ≐ ⟨α,
β⟩ ).
    consider ix: x, iy: y from index.
    assume (z ≐ (x × y) (ix, iy)). take ix. take iy.
hence thesis.
  end claim.
  hence thesis by eq.
end focus.
  given α, β such that (z ≐ ⟨x α, y β⟩ ). take (α,
β). hence thesis.
end proof.
Qed.

Theorem Products: ∀x y, ∃z, ∀w, w ∈ z ↔ ∃α ∈ x, ∃β ∈
y, w ≐ ⟨α, β⟩ .
Proof.
  intros x y. exists (x × y). apply productchar.
Qed.

Definition is_rel (x y: V) (R: V): Type
  := R ⊆ x × y.

Definition is_total (x y: V) (R: V): Type
  := ∀α ∈ x, ∃β ∈ y, ⟨α,β⟩ ∈ R.

Definition is_functional (R: V): Type
  := ∀x y y', ⟨x, y⟩ ∈ R ∧ ⟨x, y'⟩ ∈ R → y ≐ y'.

Definition is_function (x y: V) (F: V): Type
  := is_rel x y F ∧ is_total x y F ∧ is_functional F.

Definition identity (x: V): V
  := {{ p ∈ x × x | ∃y ∈ x, p ≐ ⟨y, y⟩ }}.
```

```
Lemma id_is_function {x: V}: is_function x x
(identity x).
Proof.
  split. intros a [[[ai11 ai12] ai2] eq]. exists
(ai11, ai12). assumption.
  split. intro a. exists a. unfold identity. unfold
sepV.
  apply (sigrejig (x ∧ x)). exists (a, a). apply
(sigrejig x). exists a.
  exists (refV  ⟨x a, x a⟩ ). simpl.
  split; intro y; exists y; auto using refV.
  intros y z z' [[yzi yzeq] [yz'i yz'eq]].
  repeat match goal with [hyp: ( ⟨y, ?a⟩  ≐ (?b ?c))
|- _] =>
    let i := fresh in let j := fresh in let r :=
fresh in let s := fresh in let t := fresh in
      destruct c as [[i j] r];
        change ((x × x) (i, j)) with ( ⟨x i, x j⟩ ) in
r;
          destruct r as [s t];
          change ((identity x) (existT _ (i, j) _))
with ( ⟨x i, x j⟩ ) in hyp
  end.
  repeat match goal with [hyp:  ⟨_, _⟩  ≐  ⟨_, _⟩  |-
_] =>
           apply pairing_injective in hyp; destruct
hyp
         end.
  apply traV with y;
  eauto using symV, traV.
Qed.

Definition functionspace (x y:V): V
  := sup (∃f:x→y, ∀α β:x, x α ≐ x β → y (f α) ≐ y (f
β))
        (fun p => match p with existT f _ =>
          sup x (fun α =>  ⟨x α, y (f α)⟩ ) end).
```

```
(* Triple arrow UTF-8 notation forfunctionspaces in V
*)

Notation "A ⇛ B" := (functionspace A B) (at level 55,
right associativity).

Lemma functionspacechar (x y: V)
  : ∀z, z ∈ x ⇛ y ↔ is_function x y z.
Proof.
  intros z. split.
  intros [[f pf] eq]. simpl in eq. split.
  intros w [i eq2]. eapply ext1 with (z i).
assumption.
  apply eqVdestruct in eq; simpl in eq. destruct eq
as [eq1 _]. destruct eq1 with i.
  exists (x0, f x0). assumption.
  split.
  intros α. exists (f α). apply eqVdestruct in eq;
simpl in eq; destruct eq as [_ eq].
  destruct eq with α as [x0 eq']. exists x0. finish.
  intros α β β' [[iγ eγ] [iγ' eγ']]. apply
eqVdestruct in eq; simpl in eq; destruct eq as [eq1
_].
  destruct eq1 with iγ as [γ eqγ]. destruct eq1 with
iγ' as [γ' eqγ']. clear eq1.
  assert (claim1:  〈α, β〉  ≐  〈x γ, y (f γ)〉 ); finish.
  assert (claim2:  〈α, β'〉  ≐  〈x γ', y (f γ')〉 );
finish.
  clear eγ eγ' eqγ eqγ'.
  apply pairing_injective in claim1; apply
pairing_injective in claim2.
  destruct claim1, claim2.
  assert (y (f γ) ≐ y (f γ')). apply pf. finish.
finish.
  intros [relz [totz funz]]. unfold is_total in totz;
```

```
apply TrAC in totz. destruct totz as [T p].
  unfold functionspace; apply (sigrejig (x → y)).
exists f. simpl. split.
  intros α β e; apply funz with (x α). split. apply
p. eapply ext1.
  eapply pairing_extensional. apply e. apply refV.
apply p.
  apply eqVsplit. unfold is_rel in relz. intro α.
simpl. destruct relz with (z α) as [[β γ] e]. czf.
  exists β. specialize p with β. apply traV with ( ⟨x
β, y γ⟩ ). apply e. apply pairing_extensional. czf.
  unfold is_functional in funz. apply funz with (x
β). split. exists α. finish. assumption.
  simpl. intro β. destruct p with β as [i e]. exists
i; finish.
Defined.

Theorem FunctionSpace: ∀x y, ∃z, ∀w, w ∈ z ↔
is_function x y w.
Proof.
  intros x y. exists (x ⇛ y). apply
functionspacechar.
Qed.

Definition π₁ (x y:V): V
  := sup (x × y) (λ p, let (α, β) := p in  ⟨ ⟨x α, y
β⟩ , x α⟩ ).

Lemma π₁fun (x y: V): is_function (x × y) x (π₁ x y).
Proof.
  repeat split.
  intros z [[α β] eq]. simpl in eq.
  exists ((α, β), α). assumption.
  intros [α β]. exists α. exists (α, β). apply refV.
  intros α β β'. intros [[[idx11 idx12] eq1] [[idx21
idx22] eq2]].
  repeat match goal with [hyp: _ ≐ pV (π₁ x y) (?a, ?
```

```
b) |- _] =>
            change ((π₁ x y) (a, b)) with  〈 〈x a, y
b〉 , x a〉  in hyp
          end.
  repeat match goal with [hyp:  〈_, _〉  ≐  〈_, _〉  |-
_] =>
            apply pairing_injective in hyp; destruct
hyp
          end.
  eapply traV in e1; [| apply symV; apply e].
  apply pairing_injective in e1; destruct e1 as [eq
_].
  eauto using traV, symV.
Qed.

Definition compose_relV (x y z: V) (R S: V): V
  := {{ p ∈ x × z | ∃α ∈ x, ∃β ∈ y, ∃γ ∈ z,  〈α, β〉  ∈
R ∧  〈β, γ〉  ∈ S ∧  〈α, γ〉  ≐ p }}.

Lemma composed_rel_is_relV {x y z: V} (R S: V)
  : is_rel x y R → is_rel y z S → is_rel x z
(compose_relV x y z R S).
Proof.
  intros _ _. unfold is_rel. auto. apply sep_sub.
  intros w w' [α [β [γ [αRβ [βSγ eq]]]]] weqw'.
  exists α. exists β. exists γ. split. assumption.
split. assumption. finish.
Qed.

(* Define a functor from V to the (E-)category of
setoids *)


Definition obfn: V → setoid.
  intro x.
  apply (Build_setoid x (λ a b, x a ≐ x b)); eauto
using refV, symV, traV.
Defined.
```

```
Defined.


Definition arfn (x y: V): (x ⇛ y) → (obfn x) ⇒ (obfn
y).
  intros [f fext].
  apply (Build_setoidmap (obfn x) (obfn y) f). simpl.
assumption.
Defined.

(* Must now verify functoriality *)

Definition id_index (x: V): x ⇛ x.
exists (λ α, α). auto.
Defined.

Lemma id_index_identity (x: V): (x ⇛ x) (id_index x)
≐ identity x.
Proof.
  split. intro α. unfold identity. unfold sepV. apply
(sigrejig (x × x)). exists (α, α). apply (sigrejig
x). exists α.
  exists (refV _). apply refV.
  intros [[α β] [γ eq]]. exists γ. unfold projT1.
apply symV, eq.
Qed.

Lemma functoriality_id (x: V): arfn x x (id_index x)
≈ idmap.
Proof.
  intros α. simpl. apply refV.
Qed.

Definition compose_index (x y z: V): y ⇛ z → x ⇛ y →
x ⇛ z.
  intros [f fext] [g gext].
  exists (λ α, f (g α)). intros; apply fext; apply
```

```
gext; assumption.
Defined.

Lemma functoriality_composition (x y z: V) (F: y ⇛ z)
(G: x ⇛ y)
  : arfn x z (compose_index x y z F G) ≈ (arfn y z F)
∘ (arfn x y G).
Proof.
  destruct F as [f fext]; destruct G as [g gext].
  simpl. intro; apply refV.
Qed.

(* Finally, proofs that this functor is full and
faithful *)

Lemma faithful (x y: V) (F G: x ⇛ y): arfn x y F ≈
arfn x y G → (x ⇛ y) F ≐ (x ⇛ y) G.
Proof.
  destruct F as [f fext]; destruct G as [g gext].
  unfold arfn. intro feqg. simpl in feqg.
  split. intro α. exists α. eapply traV. apply
pairing_extensional.
  apply refV. apply feqg. apply refV.
  intro β. exists β. eapply traV. apply
pairing_extensional. apply refV. apply feqg. apply
refV.
Qed.

Lemma full (x y: V): ∀f: setoidmap (obfn x) (obfn y),
∃G: x ⇛ y, arfn x y G ≈ f.
Proof.
  intros [f fext].
  unfold functionspace. apply (sigrejig (x → y)).
  exists f. exists fext. simpl. intro; apply refV.
Qed.
```

```
(* The functor maps any singleton to a terminal
setoid. This together with fact that both categories
are generated by the terminal object, and the full
faithfulness of the functors, yield that if  obfn
x  is projective in setoids, then so is x in V. In
other words if obfn x admits axiom of choice, then so
does  x. *)

Definition canon1_map (x:V) : (obfn {{ x }})  ⇒
unit_setoid.
  apply (Build_setoidmap (obfn {{ x }}) unit_setoid
(λ a, tt)).
  trivial.
Defined.

Definition canon2_map (x:V) : unit_setoid ⇒ (obfn {{
x }}).
  apply (Build_setoidmap unit_setoid (obfn {{ x }})
(λ a, tt)).
  intros.
  apply setoidrefl.
Defined.

Lemma singleton_to_terminal (x:V):
∃F:(obfn {{ x }})  ⇒ unit_setoid,
∃G:unit_setoid ⇒ (obfn {{ x }}),
F ∘ G  ≈ idmap  ∧ G ∘ F  ≈ idmap.
Proof.
  exists (canon1_map x).
  exists (canon2_map x).
  split.
  intro.
  swesetoid.
  intro.
  swesetoid.
```

```
  apply refV.
Qed.

(* A series of proofs that certain
set-theoretic expresseions are
extensional in one or another variable *)

Theorem expr_extensional_1 (z a :V):
«(fun u => z ≐ 〈 a, u 〉 )».
Proof.
  unfold extensionalV.
  intros x y P Q.
  assert ( 〈 a, x 〉 ≐ 〈 a, y 〉 ) as H.
  apply pairing_extensional.
  apply refV.
  apply Q.
  apply (traV _ _ _ P H).
Defined.

Theorem expr_extensional_2 (z a v :V):
«(fun v =>  ∃b ∈ v, z ≐ 〈a, b〉 )».
Proof.
  unfold extensionalV.
  intros x y P Q.
  assert (∃b: V,  b ∈ x ∧ z ≐ 〈a, b〉 ) as H.
  apply Bexright.
  apply expr_extensional_1.
  apply P.
  assert (∃b: V,  b ∈ y ∧ z ≐ 〈a, b〉 ) as H1.
  destruct H as [b R].
  exists b.
  split.
  destruct R as [R1 R2].
  apply (ext2 _ x _).
  apply R1.
  apply Q.
```

```
  apply R.
  apply ((snd (Bexright y (fun u => z ≐ 〈 a, u 〉 )
   (expr_extensional_1 z a))) H1).
Defined.

Theorem expr_extensional_3 (z v :V):
«(fun a =>  ∃b ∈ v, z ≐ 〈a, b〉 )».
Proof.
  unfold extensionalV.
  intros x y.
  intros P Q.
  destruct P as [b R].
  exists b.
  assert ( 〈 x, v b 〉 ≐ 〈 y, v b 〉 ) as H.
  apply pairing_extensional.
  apply Q.
  apply refV.
  apply (traV _ _ _ R H).
Defined.

(* Alternative characterization of
   cartesian product *)

Lemma productchar_2_lemma1 (x y: V):
  ∀z,  (∃a ∈ x, ∃β ∈ y, z ≐ 〈a, β〉 )
     ↔
   (∃a: V, a ∈ x ∧ ∃β ∈ y, z ≐ 〈a, β 〉 ).
Proof.
  intro z.
  apply Bexright.
  apply (expr_extensional_3 z y).
Defined.


Lemma productchar_2_lemma2 (x y: V):
  ∀z, (∃a ∈ x, ∃β ∈ y, z ≐ 〈a, β〉 )
       ↔ (∃a: V, a ∈ x ∧
```

```
                (∃β: V,  β ∈ y ∧ z ≐ ⟨α, β⟩ )).
Proof.
  intro z.
  split.
  intro H.
  assert  (∃α: V, α ∈ x ∧ ∃β ∈ y, z ≐ ⟨α, β ⟩  ) as
H1.
  apply (productchar_2_lemma1 x y).
  apply H.
  destruct H1 as  [a R].
  exists a.
  split.
  apply R.
  apply Bexright.
  apply expr_extensional_1.
  apply R.
  intro H.
  apply (productchar_2_lemma1 x y).
  destruct H as  [a R].
  exists a.
  split.
  apply R.
  apply (snd (Bexright y _ (expr_extensional_1 _
_))).
  apply R.
Defined.


Theorem productchar_2 (x y: V):
  ∀z, z ∈ x × y ↔ ∃α: V, α ∈ x ∧
                (∃β: V,  β ∈ y ∧ z ≐ ⟨α, β⟩ ).
Proof.
  intro z.
  split.
  intro P.
  assert  (∃α ∈ x, ∃β ∈ y, z ≐ ⟨α, β⟩ ) as H1.
  apply productchar.
```

```
  apply P.
  apply productchar_2_lemma2.
  apply H1.
  intro Q.
  apply productchar.
  apply productchar_2_lemma2.
  apply Q.
Defined.

Theorem product_elim (x y z w:V):
    〈z, w〉 ∈ x × y -> z ∈ x ∧ w ∈ y.
Proof.
  intro H.
  assert (∃α: V, α ∈ x ∧
                 (∃β: V,  β ∈ y ∧ 〈z, w〉 ≐ 〈α, β〉 ))
as H2.
  apply productchar_2.
  apply H.
  destruct H2 as [a [P [b [P2 P3]]]].
  assert (z ≐ a  ∧ w ≐ b) as P31.
  apply (pairing_injective _ _ _ _ P3).
  split.
  apply (ext1 _ _ _ (fst P31) P).
  apply (ext1 _ _ _ (snd P31) P2).
Qed.


Theorem product_intro (x y z w:V):
    z ∈ x ∧ w ∈ y -> 〈z, w〉 ∈ x × y.
Proof.
  intro H.
  assert (∃α: V, α ∈ x ∧
                 (∃β: V,  β ∈ y ∧ 〈z, w〉 ≐ 〈α, β〉 ))
as H2.
  exists z.
  split.
  apply H.
```

```
  exists w.
  split.
  apply H.
  apply refV.
  apply productchar_2.
  apply H2.
Qed.


Theorem prod_extensional_half (x y u v:V):
x ≐ u -> y ≐ v ->  x × y  ⊆ u × v.
Proof.
  intros P Q z R.
  assert (∃α : V, α ∈ x ∧ (∃β : V, β ∈ y ∧ z ≐ 〈 α, β
〉 )) as H.
  apply productchar_2.
  apply R.
  apply productchar_2.
  destruct H as [a [P1 [b [P2 P3]]]].
  exists a.
  split.
  apply (ext2 _ x _ P1 P).
  exists b.
  split.
  apply  (ext2 _ y _ P2 Q).
  apply P3.
Defined.

Theorem prod_extensional (x y u v:V):
x ≐ u -> y ≐ v ->  x × y  ≐ u × v.
Proof.
  intros P Q. apply ext3.
  apply prod_extensional_half.
  apply P. apply Q.
  apply prod_extensional_half.
  apply symV. apply P. apply symV. apply Q.
Defined.
```

```
Theorem is_rel_extensional (x y R x' y' R':V):
x ≐ x' -> y ≐ y' ->  R  ≐ R' -> is_rel x y R ->
is_rel x' y' R'.
Proof.
  intros H1 H2 H3 P.
  unfold is_rel in P.
  intros z Q.
  assert (x × y  ≐ x' × y') as H4.
  apply prod_extensional.
  apply H1. apply H2.
  assert (z ∈ x × y) as H5.
  apply P.
  apply (ext2 _ _ _ Q (symV _ _ H3)).
  apply (ext2 _ _ _ H5 H4).
Defined.
```

```
Theorem expr_extensional_4 (a u R :V):
«(fun u =>  〈a,u〉  ∈ R)».
Proof.
  intros x y P Q.
  assert (  〈 a, x 〉  ≐  〈 a, y 〉 ) as H.
  apply pairing_extensional.
  apply refV.
  apply Q.
  apply (ext1 _ _ _ (symV _ _ H) P).
Defined.
```

```
Theorem expr_extensional_5 (v R :V):
«(fun a => ∃b ∈ v,  〈a,b〉  ∈ R)».
Proof.
  intros x y P Q.
  destruct P as [b S].
```

```
  exists b.
  assert ( 〈 x, v b 〉 ≐ 〈 y, v b 〉 ) as H.
  apply pairing_extensional.
  apply Q.
  apply refV.
  apply (ext1 _ _ _ (symV _ _ H) S).
Defined.


Theorem is_total_char_lemma1 (x y R:V):
is_total x y R ↔
∀α:V, α ∈ x -> ∃β ∈ y,  〈α,β〉  ∈ R.
Proof.
  apply Ballright.
  apply expr_extensional_5.
Defined.

Theorem is_total_char (x y R:V):
is_total x y R ↔
∀α:V, α ∈ x -> ∃β:V,  β ∈ y ∧  〈α,β〉  ∈ R.
Proof.
  split.
  intros P a Q.
  assert (∀α:V, α ∈ x -> ∃β ∈ y,  〈α,β〉  ∈ R) as H.
  apply is_total_char_lemma1.
  apply P.
  specialize (H a Q).
  simpl in H.
  apply Bexright.
  apply (expr_extensional_4 _ y _).
  apply H.

  intro P.
  apply is_total_char_lemma1.
  intros a Q.
  specialize (P a Q).
  apply (snd (Bexright y
```

```
  apply (snd (BeXright y  _
  (expr_extensional_4 _ y _))).
  apply P.
Defined.


Theorem is_total_extensional (x y R x' y' R':V):
x ≐ x' -> y ≐ y' ->  R  ≐ R' -> is_total x y R ->
is_total x' y' R'.
Proof.
  intros P1 P2 Q1 Q2.
  assert (∀a:V, a ∈ x -> ∃β:V,  β ∈ y ∧  〈a,β〉  ∈ R)
as H1.
  apply (is_total_char x y R).
  apply Q2.
  assert (∀a:V, a ∈ x' -> ∃β:V,  β ∈ y' ∧  〈a,β〉  ∈
R') as H2.
  intros a H3.
  assert (a ∈ x) as H4.
  apply (ext2 _ _ _ H3).
  apply (symV _ _ P1).

  specialize (H1 a H4).
  destruct H1 as  [b [H5 H6 ]].
  exists b.
  split.
  apply (ext2 _ _ _ H5).
  apply P2.
  apply (ext2 _ _ _ H6 Q1).
  apply (is_total_char x' y' R').
  apply H2.
Defined.

Theorem functionspace_extensional_half (x y u v:V):
x ≐ u -> y ≐ v ->  x ⇛ y  ⊆ u  ⇛ v.
Proof.
  intros P Q z H.
  assert (is_function x y z) as H2.
```

```
  apply functionspacechar.
  apply H.
  clear H.
  assert  (is_function u v z) as H3.
  destruct H2 as [I [T F]].
  split.
  apply (is_rel_extensional _ _ _ _ _ _ P Q (refV z)
I).
  split.
  apply (is_total_extensional x y z).
  apply P.
  apply Q.
  apply refV.
  apply T.
  apply F.
  apply functionspacechar.
  apply H3.
Defined.

(* The function space construction is extensional
  as function of domain and codomain *)

Theorem functionspace_extensional (x y u v:V):
x ≐ u -> y ≐ v ->  x ⇛ y  ≐ u  ⇛ v.
Proof.
  intros P Q. apply ext3.
  apply functionspace_extensional_half.
  apply P. apply Q.
  apply functionspace_extensional_half.
  apply symV. apply P. apply symV. apply Q.
Defined.


(* A proof that certain
set-theoretic expression is
extensional in a variable *)
```

```
Theorem expr_extensional_7  (a b c f g:V):
«(fun p => ∃α ∈ a, ∃β ∈ b, ∃γ ∈ c,  〈α, β〉  ∈ f ∧  〈β,
γ〉  ∈ g ∧  〈α, γ〉  ≐ p)».
Proof.
  intros p p' H K.
  destruct H as  [ag [bg [cg [P1 [P2 P3]]]]].
  exists ag.
  exists bg.
  exists cg.
  split.
  apply P1.
  split.
  apply P2.
  apply (traV _ _ _ P3 K).
Qed.

(* A characterization of the composition
   of two relations *)

Theorem compose_relV_char (a b c f g:V)
(P: is_rel a b f)(Q: is_rel b c g):
∀ x:V, ∀ z:V,
 〈 x, z 〉  ∈ compose_relV a b c f g  ↔
  x ∈ a  ∧ z ∈ c   ∧
  ∃y:V, y ∈ b ∧  〈 x, y 〉  ∈ f  ∧  〈 y, z 〉  ∈ g.
Proof.
  intros x z.
  split.
  intro H.
  assert ( 〈 x, z 〉  ∈ a × c ∧ ∃α ∈ a, ∃β ∈ b, ∃γ ∈ c,
 〈α, β〉  ∈ f ∧  〈β, γ〉  ∈ g ∧  〈α, γ〉  ≐  〈 x, z 〉 ) as
H2.
  apply (fst (separationchar
(fun p => ∃α ∈ a, ∃β ∈ b, ∃γ ∈ c,  〈α, β〉  ∈ f ∧  〈β,
γ〉  ∈ g ∧  〈α, γ〉  ≐ p)
(expr_extensional_7 _ _ _) (a × c) ( 〈 x, z
```

```
(expr_extensional_7 _ _ _ _ _ _) (a × c) (〈 x, z
〉 ))).
  apply H.
  destruct H2 as [H3 H4].
  split.
  apply (product_elim _ _ _ _ H3).
  split.
  apply (product_elim _ _ _ _ H3).
  destruct H4 as [ag [bg [cg [H5 [H6 H7]]]]].
  exists (b bg).
  split.
  unfold memV. exists bg. apply refV.
  split.
  assert ( 〈 a ag, b bg 〉 ≐ 〈 x, b bg 〉 ) as H8.
  apply pairing_extensional.
  apply (pairing_injective _ _ _ _ H7).
  apply refV.
  apply (ext1 _ _ _ (symV _ _ H8) H5).
  assert ( 〈 b bg, z 〉 ≐ 〈 b bg, c cg 〉 ) as H9.
  apply pairing_extensional.
  apply refV.
  apply symV.
  apply (pairing_injective _ _ _ _ H7).
  apply (ext1 _ _ _ H9 H6).

  intro H.
  destruct H as [H1 [H2 [y [H3 [H4 H5]]]]].
  assert ( 〈 x, z 〉 ∈ a × c ∧ ∃α ∈ a, ∃β ∈ b, ∃γ ∈ c,
 〈α, β〉 ∈ f ∧ 〈β, γ〉 ∈ g ∧ 〈α, γ〉 ≐ 〈 x, z 〉 ) as
H6.
  split.
  apply (product_intro).
  split. apply H1. apply H2.
  unfold memV in H1.  destruct H1 as [ag H1'].
  unfold memV in H2.  destruct H2 as [cg H2'].
  unfold memV in H3.  destruct H3 as [bg H3'].
  exists ag.
  exists ba.
```

```
  exists cg.
  split.
  assert ( 〈 x, y 〉 ≐ 〈 a ag, b bg 〉 ) as E1.
  apply pairing_extensional.
  apply H1'.
  apply H3'.
  apply (ext1 _ _ _ (symV _ _ E1)  H4).
  split.
  assert ( 〈 y, z 〉 ≐ 〈 b bg, c cg 〉 ) as E2.
  apply pairing_extensional.
  apply H3'.
  apply H2'.
  apply (ext1 _ _ _ (symV _ _ E2)  H5).
  apply pairing_extensional.
  apply (symV _ _ H1').
  apply (symV _ _ H2').

  apply (snd (separationchar
(fun p => ∃α ∈ a, ∃β ∈ b, ∃γ ∈ c,  〈α, β〉 ∈ f ∧ 〈β,
γ〉 ∈ g ∧ 〈α, γ〉 ≐ p)
(expr_extensional_7 _ _ _ _ _ )  (a × c)  ( 〈 x, z
〉 ))).
  apply H6.
Qed.



Theorem compose_relV_aux (a b c f g t:V):
t∈compose_relV a b c f g -> ∃ x:V, x ∈ a  ∧ ∃ y:V,
y ∈ c  ∧ t  ≐  〈x, y〉 .
Proof.
  assert (compose_relV a b c f g ⊆ a × c) as H.
  unfold compose_relV.
  apply (sep_sub _ (expr_extensional_7 _ _ _ _ _)  (a
× c)).
  intro P.
  assert (t ∈ a × c) as H1
```

```
  assert (t ∈ a × c) as H1.
  apply H.
  apply P.
  apply productchar_2.
  apply H1.
Qed.



Theorem compose_relV_char_2 (a b c f g:V)
(P: is_rel a b f)(Q: is_rel b c g):
∀ t:V, t ∈ compose_relV a b c f g  ↔
(∃ x:V, ∃ z:V,
t ≐ 〈 x, z 〉 ∧ x ∈ a ∧ z ∈ c  ∧
  ∃y:V, y ∈ b ∧ 〈 x, y 〉 ∈ f  ∧ 〈 y, z 〉 ∈ g).
Proof.
  intro t.
  split.
  intro H.
  assert (∃ x:V, x ∈ a  ∧ ∃ y:V,
y ∈ c  ∧ t  ≐  〈x, y〉 ) as H1.
  apply (compose_relV_aux a b c f g t).
  apply H.
  destruct H1 as [x [H2 [y [H3 H4]]]].
  exists x.
  exists y.
  split.
  apply H4.
  apply compose_relV_char.
  apply P.
  apply Q.
  apply (ext1 _ _ _ (symV _ _ H4) H).

  intro H.
  destruct H as [x [z [H1 H2]]].
  assert ( 〈 x, z 〉 ∈ compose_relV a b c f g).
  apply compose_relV_char.
  apply P.
```

```
  apply P.
  apply Q.
  apply H2.
  apply (ext1 _ _ _ H1 H).
Qed.


Theorem compose_relV_extensional_half (a b c f g:V)
(P: is_rel a b f)(Q: is_rel b c g)
(a' b' c' f' g':V)
(P': is_rel a' b' f')(Q': is_rel b' c' g')
(E1:a ≐ a')(E2: b ≐ b')(E3:c ≐ c')
(E4:f ≐ f')(E5: g ≐ g'):
compose_relV a b c f g  ⊆ compose_relV a' b' c' f'
g'.
Proof.
  intros t H.
  assert (∃ x:V, x ∈ a  ∧ ∃ z:V,
z ∈ c  ∧ t  ≐  〈x, z〉 ) as H1.
  apply (compose_relV_aux a b c f g).
  apply H.
  destruct H1 as  [x [H2 [z [H3 H4]]]].
  assert ( 〈 x, z 〉  ∈ compose_relV a b c f g).
  apply (ext1 _ _ _ (symV _ _ H4) H).
  assert ( x ∈ a  ∧ z ∈ c  ∧
  ∃y:V, y ∈ b ∧ 〈 x, y 〉 ∈ f  ∧ 〈 y, z 〉 ∈ g) as
H5.
  apply compose_relV_char.
  apply P.
  apply Q.
  apply H0.
  assert ( x ∈ a'  ∧ z ∈ c'  ∧
  ∃y:V, y ∈ b' ∧ 〈 x, y 〉 ∈ f'  ∧ 〈 y, z 〉 ∈ g') as
H6.
  split.
  apply (ext2 _ _ _ H2 E1).
  split.
  apply (ext2       H3 E3)
```

```
  apply (ext2 _ _ _ H5 E5).
  destruct H5 as  [H7 [H8 [y [H9 [HA HB]]]]].
  exists y.
  split.
  apply (ext2 _ _ _ H9 E2).
  split.
  apply (ext2 _ _ _ HA E4).
  apply (ext2 _ _ _ HB E5).
  assert ( 〈 x, z 〉 ∈ compose_relV a' b' c' f' g') as
HC.
  apply compose_relV_char.
  apply P'.
  apply Q'.
  apply H6.
  apply (ext1 _ _ _ H4 HC).
Qed.

(* Extensionality of relation composition *)

Theorem compose_relV_extensional (a b c f g:V)
(P: is_rel a b f)(Q: is_rel b c g)
(a' b' c' f' g':V)
(P': is_rel a' b' f')(Q': is_rel b' c' g')
(E1:a ≐ a')(E2: b ≐ b')(E3:c ≐ c')
(E4:f ≐ f')(E5: g ≐ g'):
compose_relV a b c f g ≐ compose_relV a' b' c' f' g'.
Proof.
  apply ext3.
  apply compose_relV_extensional_half.
  assumption.
  assumption.
  assumption.
  assumption.
  assumption.
  assumption.
  assumption.
  assumption.
  assumption.
```

```
  apply compose_relV_extensional_half.
  assumption.
  assumption.
  assumption.
  assumption.
  apply symV. assumption.
  apply symV. assumption.
  apply symV. assumption.
  apply symV. assumption.
  apply symV. assumption.
Qed.


(* Corresponding to setoidmap we have .. *)

Record Typeoidmap (A B: Typeoid) :=
  {
    Typeoidmapfunction :> A → B;
    Typeoidmapextensionality : ∀x y: A, x ≈≈≈ y →
Typeoidmapfunction x ≈≈≈ Typeoidmapfunction y
  }.


(* Proof irrelevant Typeoid families *)

Record Typeoidfamily (A: Typeoid) :=
  {
    Typeoidfamilyobj :> A → Typeoid;
    Typeoidfamilymap :  ∀x y: A, ∀p: x ≈≈≈ y,
                        Typeoidmap
                        (Typeoidfamilyobj x)
                        (Typeoidfamilyobj y);
    Typeoidfamilyref :  ∀x: A, ∀y: Typeoidfamilyobj
x,
      Typeoidfamilymap x x (Typeoidrefl A x) y ≈≈≈ y;
    Typeoidfamilyirr :  ∀x y: A, ∀p q: x ≈≈≈ y, ∀z:
Typeoidfamilyobj x,
```

```
                              Typeoidfamilymap x y p z ≈≈≈
Typeoidfamilymap x y q z;
    Typeoidfamilycmp :  ∀x y z: A, ∀p: x ≈≈≈ y, ∀q: y
≈≈≈ z, ∀w: Typeoidfamilyobj x,
                              (Typeoidfamilymap y z q)
((Typeoidfamilymap x y p) w)
                                ≈≈≈ Typeoidfamilymap x z
  (Typeoidtra _ _ _ _ p q) w
  }.

(* The typeoid of V-sets *)

Definition VTypeoid:Typeoid.
  apply (Build_Typeoid V (fun x y =>  x ≐ y)).
  apply refV.
  apply symV.
  apply traV.
Defined.


(* Transform a setoid to a Typeoid *)

Definition Magnify (x:setoid): Typeoid.
  apply (Build_Typeoid x (fun u v => u ≈ v)).
  apply setoidrefl.
  apply setoidsym.
  apply setoidtra.
Defined.

(* From a proof equality of two V-sets we obtain
  two operations  push  and  pull  that tells
  how the elements of the V-sets correspond. *)

Definition push  {ax:Set}{fx:ax->V}
                 {ay:Set}{fy:ay->V}
                 (p: (sup ax fx) ≐ (sup ay fy))
(t:ax):= proiT1((fst p) t):ay.
```

```
(t:ax):= projT1((fst p) t):ay.

Theorem pushprop  {ax:Set}{fx:ax->V}
                  {ay:Set}{fy:ay->V}
                  (p: (sup ax fx) ≐ (sup ay fy)):
  (∀ t: ax, fx t  ≐ fy (push p t)).
Proof.
  intro t.
  apply (projT2 (fst p t)).
Defined.

Definition pull {ax:Set}{fx:ax->V}
                  {ay:Set}{fy:ay->V}
                  (p: (sup ax fx) ≐ (sup ay fy))
(t:ay):= projT1((snd p) t):ax.

Theorem pullprop {ax:Set}{fx:ax->V}
                  {ay:Set}{fy:ay->V}
                  (p: (sup ax fx) ≐ (sup ay fy)):
  (∀ t: ay, fx (pull p t)  ≐ fy t).
Proof.
  intro t.
  apply (projT2 (snd p t)).
Defined.

(* Rbar is V-sets as a Typeoid family over VTypeoid
*)

Definition Rbar_ob (x:VTypeoid) := Magnify (obfn x).

Definition Rbar_transp_helper:
  ∀x y: VTypeoid, ∀p: x ≈≈≈ y,
                        (Rbar_ob x) ->
                        (Rbar_ob y).
  intros [ix fx ]  [iy fy].
  intros H t.
  apply (push H t).
Defined
```

```
Defined.


Definition Rbar_transp: ∀x y: VTypeoid, ∀p: x ≈≈≈ y,
                          Typeoidmap
                        (Rbar_ob x)
                        (Rbar_ob y).
  intros x y p.
  apply (Build_Typeoidmap (Rbar_ob x) (Rbar_ob y)
           (Rbar_transp_helper x y p)).
  intros s t H.
  destruct x as [ix fx].
  destruct y as [iy fy].
  assert (fy (Rbar_transp_helper (sup ix fx) (sup iy
fy) p s) ≐
    (fy
    (Rbar_transp_helper (sup ix fx) (sup iy fy) p t)))
as H0.
  assert (fx s  ≐  fy (Rbar_transp_helper (sup ix fx)
(sup iy fy) p s)) as H1.
  apply (pushprop p s).
  assert (fx t  ≐  fy (Rbar_transp_helper (sup ix fx)
(sup iy fy) p t)) as H2.
  apply (pushprop p t).
  assert (fx s  ≐ fx t) as H3.
  apply H.
  apply (traV _ _ _ (traV _ _ _  (symV _ _ H1) H3)
H2).
  apply H0.
Defined.


Definition Rbar : Typeoidfamily VTypeoid.
  apply (Build_Typeoidfamily
      VTypeoid Rbar_ob Rbar_transp).
  intros [ix fx].
  intro t.
  apply Typeoidsym.
  assert (fx t  ≐
```

```
  assert (fx t  =
   fx ((Rbar_transp (sup ix fx) (sup ix fx)
(Typeoidrefl VTypeoid (sup ix fx))) t)) as H1.
  destruct (Typeoidrefl VTypeoid (sup ix fx)) as
[p1 p2].
  apply  (projT2 (p1 t)).
  apply H1.

  intros x y p q t.
  destruct x as [ix fx].
  destruct y as [iy fy].
  assert ( fy ((Rbar_transp (sup ix fx) (sup iy fy)
p) t)    = fy ((Rbar_transp (sup ix fx) (sup iy fy)
q) t) ) as H1.
  assert (fx t  =  fy ((Rbar_transp (sup ix fx) (sup
iy fy) p) t) ) as H2.
  apply (pushprop p t).
  assert (fx t  =  fy ((Rbar_transp (sup ix fx) (sup
iy fy) q) t) ) as H3.
  apply (pushprop q t).
  apply (traV _ _ _ (symV _ _ H2) H3).
  apply H1.

  intros x y z p q t.
  destruct x as [ix fx].
  destruct y as [iy fy].
  destruct z as [iz fz].
  assert (fx t  =  fy ((Rbar_transp (sup ix fx) (sup
iy fy) p) t) ) as H1.
  apply (pushprop p t).

  assert (fy ((Rbar_transp (sup ix fx) (sup iy fy) p)
t) = fz ((Rbar_transp (sup iy fy) (sup iz fz) q)
((Rbar_transp (sup ix fx) (sup iy fy) p) t))) as H2.
  destruct q as  [q1 q2].
  apply (projT2 (q1 ((Rbar_transp (sup ix fx) (sup iy
fy) p) t))).
  assert ( sup ix fx ~~~ { VTypeoid }sup iz fz) as
```

```
  assert ( sup ix fx ≈≈≈{ VTypeoid }sup iz fz) as
H3.

  apply (Typeoidtra VTypeoid (sup ix fx) (sup iy fy)
(sup iz fz) p q).
  assert (fx t  ≐
fz ((Rbar_transp (sup ix fx) (sup iz fz)
      (Typeoidtra VTypeoid (sup ix fx) (sup iy fy)
(sup iz fz) p q)) t)) as H4.
  destruct (Typeoidtra VTypeoid (sup ix fx) (sup iy
fy) (sup iz fz) p q) as  [r1 r2].
  apply (projT2 (r1 t)).
  apply (traV _ _ _ (traV _ _ _ (symV _ _ H2) (symV _
_ H1)) H4).
Defined.

(* Category of Type-size *)


Record Cat: Type :=
  {
    Catobj  :> Typeoid;
    Catarr  : Typeoid;
    Catcms  : Typeoid;
    Catid   : Typeoidmap Catobj Catarr;
    Catdom  : Typeoidmap Catarr Catobj;
    Catcod  : Typeoidmap Catarr Catobj;
    Catfst  : Typeoidmap Catcms Catarr;
    Catsnd  : Typeoidmap Catcms Catarr;
    Catcmp  : Typeoidmap Catcms Catarr;
    CatK1 : ∀x: Catobj, Catdom (Catid x) ≈≈≈ x;
    CatK2 : ∀x: Catobj, Catcod (Catid x) ≈≈≈ x;
    CatK3 : ∀u: Catcms, Catdom (Catcmp u) ≈≈≈
                    Catdom (Catfst u);
    CatK4 : ∀u: Catcms, Catcod (Catcmp u) ≈≈≈
                    Catcod (Catsnd u);
    CatK5 : ∀u v: Catcms, Catfst u ≈≈≈ Catfst v ->
                 Catsnd u ≈≈≈ Catsnd v -> u ≈≈≈ v;
    CatK6 : ∀f g: Catarr, Catdom f ≈≈≈ Catcod g ->
```

```
    CatK6 : ∀f g: Catarr, Catdom f ≈≈≈ Catcod g ->
     ∃u : Catcms, Catsnd u ≈≈≈ f ∧ Catfst u ≈≈≈ g;

    CatK7 : ∀u: Catcms, ∀y: Catobj, Catfst u ≈≈≈
Catid y ->
              Catcmp u ≈≈≈ Catsnd u;
    CatK8 : ∀u: Catcms, ∀x: Catobj, Catsnd u ≈≈≈
Catid x ->
              Catcmp u ≈≈≈ Catfst u;
    CatK9 : ∀u v:Catcms, ∀ w z: Catcms,
       Catfst w ≈≈≈ Catfst v -> Catsnd v ≈≈≈ Catfst u
->
       Catsnd u ≈≈≈ Catsnd z -> Catsnd w ≈≈≈ Catcmp u
->
       Catcmp v ≈≈≈ Catfst z -> Catcmp w ≈≈≈ Catcmp z
  }.

Record Functor (A B:Cat):Type :=
{Funob:  Typeoidmap (Catobj A) (Catobj B);
Funarr: Typeoidmap (Catarr A) (Catarr B);
Funcms: Typeoidmap (Catcms A) (Catcms B);
Fun_id: ∀ a: Catobj A,
         Funarr (Catid A a) ≈≈≈ (Catid B (Funob a));
Fun_dom: ∀ a: Catarr A,
         Funob (Catdom A a) ≈≈≈ (Catdom B (Funarr
a));
Fun_cod: ∀ a: Catarr A,
         Funob (Catcod A a) ≈≈≈ (Catcod B (Funarr
a));
Fun_fst: ∀ a: Catcms A,
         Funarr (Catfst A a) ≈≈≈ (Catfst B (Funcms
a));
Fun_snd: ∀ a: Catcms A,
         Funarr (Catsnd A a) ≈≈≈ (Catsnd B (Funcms
a));
Fun_cmp: ∀ a: Catcms A,
         Funarr (Catcmp A a) ≈≈≈ (Catcmp B (Funcms
a))
  }.
```

```
  ʃ.

(* TWO DIFFERENT CATEGORIES

  Two different categories of V-sets are defined
  and proved isomorphic

*)

(* Build the category of V-sets *)

Definition ObV:=VTypeoid.

(* predicate for a set to be an arrow *)

Definition is_arrow (u:V) :=
  ∃ a:V, ∃b:V, ∃f:V,
  u ≐ 〈 〈a, b〉 , f〉   ∧ is_function a b f.
Definition ArrV_under := ∃ u:V, is_arrow u.

Definition ArrV:Typeoid.
  apply (Build_Typeoid ArrV_under
       (fun u v =>
          projT1 u  ≐ projT1 v)).
  intro x. apply refV.
  intros x y. apply symV.
  intros x y z. apply traV.
Defined.

Definition make_ArrV
(a  b f : V)
(P: is_function a b f):=
  (existT _  〈 〈a, b〉 , f〉
         (existT _ a
         (existT _ b
         (existT _ f
          ((refV _), P))))): ArrV.
```

```
Theorem id_arrowpf (x:V): is_arrow 〈〈x, x〉 ,
identity x〉 .
Proof.
  exists x. exists x. exists (identity x).
  split.
  apply pairing_extensional.
  apply pairing_extensional.
  apply refV. apply refV. apply refV.
  apply id_is_function.
Defined.


(* Proofs that certain
set-theoretic expressions are
extensional in a variable *)


Theorem expr_extproof_1 (x:V):
«(fun z => ∃y : x, z ≐ 〈 x y, x y 〉 )».
Proof.
  unfold extensionalV.
  intros z z' P Q.
  destruct P as  [y P].
  exists y.
  apply (traV _ _ _ (symV _ _ Q) P).
Defined.

Theorem expr_extproof_2 (z:V):
«(fun u => z ≐ 〈 u, u 〉 )».
Proof.
  unfold extensionalV.
  intros u u' P Q.
  assert ( 〈 u, u 〉  ≐ 〈 u', u' 〉 ) as H.
  apply pairing_extensional.
  apply Q.
  apply Q.
```

```
  apply (traV _ _ _ P H).
Defined.


Theorem identity_ext_lemma_half (x y: V)(p: x ≐ y):
(identity x) ⊆ (identity y).
Proof.
  intro z.
  intro P.
  unfold identity in P.
  assert (z ∈ x × x  ∧ ∃y : x, z ≐  〈 x y, x y 〉 ) as
H.
  apply (separationchar (fun z => ∃y : x, z ≐  〈 x y,
x y 〉 )
  (expr_extproof_1 x)  (x × x) z).
  apply P.
  assert (z ∈ y × y  ∧ ∃t : y, z ≐  〈 y t, y t 〉 ) as
H2.
  destruct H as  [H0 H1].
  assert (∃w, w ∈ x ∧ z  ≐〈w, w〉 ) as H3.
  apply Bexright.
  unfold extensionalV.
  intros u v Q R.
  assert ( 〈 u, u 〉  ≐  〈 v, v 〉 ) as H4.
  apply pairing_extensional.
  apply R. apply R.
  apply (traV _ _ _ Q H4).
  apply H1.
  split.
  assert (x × x  ≐  y × y) as H4.
  apply prod_extensional.
  apply p. apply p.
  apply (ext2 _ _ _ H0 H4).
  assert (∃w : V, w ∈ y ∧ z ≐  〈 w, w 〉 ) as H5.
  destruct H3 as  [w H6].
  exists w.
  split.
```

```
  destruct H6 as  [H6a H6b].
  apply (ext2 _ x _). apply H6a. apply p.
  apply H6.
  apply ((snd (Bexright y (fun u => z ≐  〈 u, u 〉  )
   (expr_extproof_2 z))) H5).
  unfold identity.
  apply  separationchar.
  apply (expr_extproof_1).
  apply H2.
Defined.


Theorem identity_ext_lemma (x y: V)(p: x ≐ y):
(identity x) ≐ (identity y).
Proof.
  apply ext3.
  apply identity_ext_lemma_half.
  apply p.
  apply identity_ext_lemma_half.
  apply symV. apply p.
Defined.

Definition idV:Typeoidmap ObV ArrV.
apply (Build_Typeoidmap ObV ArrV
        (fun x => existT _  〈〈x, x〉 , identity x〉
(id_arrowpf x))).
  intros x y P.
  assert (  〈  〈 x, x 〉 , identity x 〉   ≐  〈  〈 y, y
〉 , identity y 〉 ) as H.
  apply pairing_extensional.
  apply pairing_extensional.
  apply P. apply P.
  apply identity_ext_lemma.
  apply P.
  apply H.
Defined.
```

```
Definition domV:Typeoidmap ArrV ObV.
  apply (Build_Typeoidmap ArrV ObV
         (fun w => projT1 (projT2 w))).
  intros w w' P.
  destruct w as [u [a [b [f [Q R]]]]].
  destruct w' as [u' [a' [b' [f' [Q' R']]]]].
  simpl.
  simpl in P.
  assert  ( 〈 〈 a, b 〉 , f 〉 ≐  〈 〈 a', b' 〉 , f'
〉 ) as H.
  apply (traV _ _ _ (symV _ _ Q) (traV _ _ _ P Q')).
  assert  ( 〈 a, b 〉 ≐  〈 a', b' 〉 ) as H2.
  apply (pairing_injective _ _ _ _ H).
  apply (pairing_injective _ _ _ _ H2).
Defined.


Definition codV: Typeoidmap ArrV ObV.
  apply (Build_Typeoidmap ArrV ObV
       (fun w => projT1 (projT2 (projT2 w)))).
  intros w w' P.
  destruct w as [u [a [b [f [Q R]]]]].
  destruct w' as [u' [a' [b' [f' [Q' R']]]]].
  simpl.
  simpl in P.
  assert  ( 〈 〈 a, b 〉 , f 〉 ≐  〈 〈 a', b' 〉 , f'
〉 ) as H.
  apply (traV _ _ _ (symV _ _ Q) (traV _ _ _ P Q')).
  assert  ( 〈 a, b 〉 ≐  〈 a', b' 〉 ) as H2.
  apply (pairing_injective _ _ _ _ H).
  apply (pairing_injective _ _ _ _ H2).
Defined.

Definition CmsV_under :=  ∃ w:V,
  ∃ u:ArrV, ∃ v:ArrV,
  w ≐ 〈projT1 u, projT1 v〉  ∧
  codV u ≐ domV v.
```

```
Definition CmsV:Typeoid.
  apply (Build_Typeoid CmsV_under
       (fun u v =>
          projT1 u  ≐ projT1 v)).
  intro x. apply refV.
  intros x y. apply symV.
  intros x y z. apply traV.
Defined.


Definition make_CmsV
  (c d g : V)
  (P : is_function c d g)
  (a b f : V)
  (Q : is_function a b f)
  (H : c ≐ b):=
(existT _ ( 〈 〈 〈 a, b 〉 , f 〉 , 〈 〈 c, d 〉 , g 〉
〉 )
  (existT _  (make_ArrV a b f Q)
    (existT _  (make_ArrV c d g P)
       ((refV _), (symV _ _ H))))) : CmsV.


Definition fstV: Typeoidmap CmsV ArrV.
  apply (Build_Typeoidmap  CmsV ArrV
       (fun z => projT1 (projT2 z))).
  intros [w [u [v Q]]] [w' [u' [v' Q']]] P.
  simpl in P.
  simpl.
  assert ( 〈 projT1 u, projT1 v 〉  ≐ 〈 projT1 u',
projT1 v' 〉 ) as H.
  apply (traV _ _ _ (symV _ _ (fst Q))
       (traV _ _ _ P (fst Q'))).
  assert (projT1 u ≐ projT1 u' ∧ projT1 v ≐ projT1
v') as H2.
  apply pairing_injective.
```

```
  apply H.
  apply H2.
Defined.


Definition sndV: Typeoidmap CmsV ArrV.
  apply (Build_Typeoidmap  CmsV ArrV
       (fun z => projT1 (projT2 (projT2 z)))).
  intros [w [u [v Q]]] [w' [u' [v' Q']]] P.
  simpl in P.
  simpl.
  assert ( 〈 projT1 u, projT1 v 〉  ≐ 〈 projT1 u',
projT1 v' 〉 ) as H.
  apply (traV _ _ _ (symV _ _ (fst Q))
       (traV _ _ _ P (fst Q'))).
  assert (projT1 u ≐ projT1 u' ∧ projT1 v ≐ projT1
v') as H2.
  apply pairing_injective.
  apply H.
  apply H2.
Defined.


Definition cmpV_helper: CmsV -> ArrV.
  intros [w [[fbl P] [[gbl Q] R]]].
  destruct P as [a [b [f [P1 P2]]]].
  destruct Q as [c [d [g [Q1 Q2]]]].
  simpl in R.
  destruct R as [R1 R2].
  exists  〈  〈 a, d 〉 , (compose_relV a b d f g) 〉 .
  exists a.
  exists d.
  exists (compose_relV a b d f g).
  split.
  apply refV.
  split.
  unfold is_rel.
```

```
  apply sep_sub.
  intros m m' [α [β [γ [αRβ [βSγ eq]]]]] meqm'.
  exists α. exists β. exists γ.
  split. apply αRβ.
  split. apply βSγ.
  apply (traV _ _ _ eq meqm').
  split.
  destruct P2 as  [P20 [P21 P22]].
  destruct Q2 as  [Q20 [Q21 Q22]].
  assert (is_total b d g) as H.
  apply (is_total_extensional c d g b d g
(symV _ _ R2) (refV _) (refV _) Q21).
  assert (∀x:V, x ∈ b -> ∃y:V,  y ∈ d ∧  〈x,y〉  ∈ g)
as H1.
  apply is_total_char.
  apply H.
  assert (∀x:V, x ∈ a -> ∃y:V,  y ∈ b ∧  〈x,y〉  ∈ f)
as H2.
  apply is_total_char.
  apply P21.
  assert (∀x:V, x ∈ a -> ∃y:V,  y ∈ d ∧  〈x,y〉  ∈
(compose_relV a b d f g) ) as H3.
  intros x N.
  specialize (H2 x N).
  destruct H2 as [y [M P]].
  specialize (H1 y M).
  destruct H1 as [z [M' P']].
  exists z.
  split.
  apply M'.
  assert (is_rel b d g) as Q20'.
  apply (is_rel_extensional c d g).
  apply (symV _ _ R2).
  apply refV.
  apply refV.
  apply Q20.
  apply (snd (compose_relV_char a b d f g P20 Q20'  x
```

```
z)).
  split.
  apply N.
  split.
  apply M'.
  exists y.
  split.
  apply M.
  split.
  apply P.
  apply P'.
  apply is_total_char.
  apply H3.
  unfold is_functional.
  intros x y y' [H1 H2].
  unfold is_function in P2.
  unfold is_function in Q2.
  assert (x ∈ a  ∧ y ∈ d   ∧
  ∃z:V, z ∈ b ∧ 〈 x, z 〉 ∈ f  ∧ 〈 z, y 〉 ∈ g) as
H1'.
  apply compose_relV_char.
  apply P2.
  apply (is_rel_extensional c d g).
  apply (symV _ _ R2).
  apply refV.
  apply refV.
  apply Q2.
  apply H1.
  destruct H1' as [H11 [H12 [z H13]]].
  assert (x ∈ a  ∧ y' ∈ d   ∧
  ∃z:V, z ∈ b ∧ 〈 x, z 〉 ∈ f  ∧ 〈 z, y' 〉 ∈ g) as
H2'.
  apply compose_relV_char.
  apply P2.
  apply (is_rel_extensional c d g).
  apply (symV _ _ R2).
  apply refV.
```

```
  apply refV.
  apply Q2.
  apply H2.
  destruct H2' as [H21 [H22 [z' H23]]].
  destruct P2 as  [_ [_ P2']].
  unfold is_functional in P2'.
  assert (z ≐ z') as H4.
  apply (P2' x).
  split.
  apply H13.
  apply H23.
  assert ( 〈 z', y' 〉  ≐  〈 z, y' 〉 ) as H5.
  apply pairing_extensional.
  apply (symV _ _ H4).
  apply refV.
  assert ( 〈 z, y' 〉  ∈ g) as H6.
  apply (ext1 _ _ _ (symV _ _ H5)).
  apply H23.
  destruct Q2 as  [_ [_ Q2']].
  unfold is_functional in Q2'.
  apply (Q2' z).
  split.
  apply H13.
  apply H6.
Defined.


Lemma singleton_extensional (x y: V):
x ≐ y ->  {{ x }} ≐ {{ y }}.
Proof.
  intro H.
  apply (Extensionality {{ x }} {{ y }} ).
  intro z.
  split.
  intro H1.
  unfold memV in H1.
  destruct H1 as  [ix P].
```

```
  exists ix.
  simpl in P.

  simpl.
  apply (traV _ _ _ P H).
  intro H1.
  unfold memV in H1.
  destruct H1 as  [iy P].
  exists iy.
  simpl in P.
  simpl.
  apply (traV _ _ _ P (symV _ _ H)).
Qed.


(* A proof that certain
set-theoretic expression is
extensional in a variable *)

Theorem expr_extensional_8 (z:V):
«(fun p => (∃v : z, p ≐  〈 z v, z v 〉 ))».
Proof.
  intros x y P Q.
  destruct P as [v P2].
  exists v.
  apply (traV _ _ _ (symV _ _ Q) P2).
Qed.


Theorem identity_lemma (x y z:V):
 〈 x, y 〉  ∈ identity z -> x ≐ y.
Proof.
  intro H.
  unfold identity in H.
  assert (( 〈 x, y 〉  ∈  z × z) ∧ (∃v : z,  〈 x, y 〉  ≐
 〈 z v, z v 〉 )) as H2.
  apply (fst (separationchar
(fun p => (∃v : z, p ≐  〈 z v, z v 〉 ))
```

```
(expr_extensional_8 z) (z × z)  〈 x, y 〉 )).
  apply H.
  destruct H2 as [_ [v H3]].
  assert (x  ≐ z v) as H4.
  apply (pairing_injective _ _ _ _ H3).
  assert (y  ≐ z v) as H5.
  apply (pairing_injective _ _ _ _ H3).
  apply (traV _ _ _ H4 (symV _ _ H5)).
Qed.


Theorem identity_lemma_2 (x y:V):
   x ∈ y ->  〈 x, x 〉  ∈ identity y.
Proof.
  intro H.
  unfold identity.
  assert ((〈 x, x 〉  ∈  y × y) ∧ (∃v : y,  〈 x, x 〉  ≐
 〈 y v, y v 〉 )) as H2.
  split.
  apply product_intro.
  split.
  apply H.
  apply H.
  unfold memV in H.
  destruct H as  [idx H2].
  exists idx.
  apply pairing_extensional.
  apply H2.
  apply H2.
  apply (snd (separationchar
(fun p => (∃v : y, p ≐  〈 y v, y v 〉 ))
(expr_extensional_8 y) (y × y)  〈 x, x 〉 )).
  apply H2.
Qed.

Theorem is_rel_pairs (a b r x:V)(P:is_rel a b r):
x ∈ r -> ∃v : V, ∃z : V, v ∈ a  ∧ z ∈ b
```

```
                    ∧ x ≐ 〈 y, z 〉 .
Proof.
  intro H.
  unfold is_rel in P.
  assert (x ∈ a × b) as H2.
  apply P.
  apply H.
  assert ((∃y : V, y ∈ a ∧ (∃z : V, z ∈ b ∧ x ≐ 〈 y,
z 〉 ))) as H3.
  apply productchar_2.
  apply H2.
  destruct H3 as [y [H4 [z [H5 H6]]]].
  exists y.
  exists z.
  split. apply H4.
  split. apply H5. apply H6.
Qed.



Definition cmpV: Typeoidmap CmsV ArrV.
apply (Build_Typeoidmap CmsV ArrV cmpV_helper).
  intros [w [[fbl P] [[gbl Q] [R1 R2]]]]
         [w' [[fbl' P'] [[gbl' Q'] [R1' R2']]]] H.
  destruct P as [a [b [f [P1 P2]]]].
  destruct P' as [a' [b' [f' [P1' P2']]]].
  destruct Q as [c [d [g [Q1 Q2]]]].
  destruct Q' as [c' [d' [g' [Q1' Q2']]]].
  simpl in R1.
  simpl in R2.
  simpl in R1'.
  simpl in R2'.
  simpl in H.
  simpl.
  assert ( 〈 fbl, gbl 〉 ≐ 〈 fbl', gbl' 〉 ) as H2.
  apply (traV _ _ _ (symV _ _ R1) (traV _ _ _ H
R1')).
```

```
  assert (fbl  ≐ fbl') as H21.
  apply (pairing_injective _ _ _ _ H2).
  assert (gbl  ≐ gbl') as H22.
  apply (pairing_injective _ _ _ _ H2).
  assert ( ⟨ ⟨ a, b ⟩ , f ⟩ ≐ ⟨ ⟨ a', b' ⟩ , f' ⟩ )
as H3.
  apply (traV _ _ _ (symV _ _ P1) (traV _ _ _ H21
P1')).
  assert ( ⟨ ⟨ c, d ⟩ , g ⟩ ≐ ⟨ ⟨ c', d' ⟩ , g' ⟩ )
as H4.
  apply (traV _ _ _ (symV _ _ Q1) (traV _ _ _ H22
Q1')).
  assert (a ≐ a') as H31.
  apply (pairing_injective _ _ _ _
      (fst (pairing_injective _ _ _ _ H3))).
  assert (b ≐ b') as H32.
  apply (pairing_injective _ _ _ _
      (fst (pairing_injective _ _ _ _ H3))).
  assert (f ≐ f') as H33.
  apply (pairing_injective _ _ _ _ H3).
  assert (c ≐ c') as H41.
  apply (pairing_injective _ _ _ _
      (fst (pairing_injective _ _ _ _ H4))).
  assert (d ≐ d') as H42.
  apply (pairing_injective _ _ _ _
      (fst (pairing_injective _ _ _ _ H4))).
  assert (g ≐ g') as H43.
  apply (pairing_injective _ _ _ _ H4).
  assert ( ⟨ ⟨ a, d ⟩ , compose_relV a b d f g ⟩
≐  ⟨ ⟨ a', d' ⟩ , compose_relV a' b' d' f' g' ⟩
) as H5.
  apply pairing_extensional.
  apply pairing_extensional.
  apply H31.
  apply H42.
  apply compose_relV_extensional.
  unfold is_function in P2
```

```
  unfold is_function in P2.
  apply P2.
  assert (is_rel b d g) as Q21.
  unfold is_function in Q2.
  apply (is_rel_extensional c d g b d g).
  apply (symV _ _ R2).
  apply refV.
  apply refV.
  apply Q2.
  apply Q21.
  unfold is_function in P2'.
  apply P2'.
  assert (is_rel b' d' g') as Q21'.
  unfold is_function in Q2'.
  apply (is_rel_extensional c' d' g' b' d' g').
  apply (symV _ _ R2').
  apply refV.
  apply refV.
  apply Q2'.
  apply Q21'.
  assumption.
  assumption.
  assumption.
  assumption.
  assumption.
  apply H5.
Defined.



Theorem Vcat: Cat.
Proof.
  apply (Build_Cat ObV ArrV CmsV idV domV codV fstV
sndV cmpV).
  intro x.
  unfold idV.
  unfold domV.
  simpl.
```

```
apply refV.


intro x.
unfold idV.
unfold codV.
simpl.
apply refV.

intro u.
destruct u as [x [arr1 [arr2 [PA PB]]]].
destruct arr1 as  [v [a [b [f [P11 P12]]]]].
destruct arr2 as  [w [c [d [g [P21 P22]]]]].
simpl in PA.
simpl in PB.
simpl.
apply refV.

intro u.
destruct u as [x [arr1 [arr2 [PA PB]]]].
destruct arr1 as  [v [a [b [f [P11 P12]]]]].
destruct arr2 as  [w [c [d [g [P21 P22]]]]].
simpl in PA.
simpl in PB.
simpl.
apply refV.

intros u u' H1 H2.
destruct u as [x [arr1 [arr2 [PA PB]]]].
destruct arr1 as  [v [a [b [f [P11 P12]]]]].
destruct arr2 as  [w [c [d [g [P21 P22]]]]].
destruct u' as [x' [arr1' [arr2' [PA' PB']]]].
destruct arr1' as  [v' [a' [b' [f' [P11' P12']]]]].
destruct arr2' as  [w' [c' [d' [g' [P21' P22']]]]].
simpl in PA.
simpl in PB.
simpl in PA'.
simpl in PB'.
```

```
  simpl in H1.
  simpl in H2.

  simpl.
  assert ( 〈 v, w 〉 ≐ 〈 v', w' 〉 ) as H3.
  apply pairing_extensional.
  apply H1.
  apply H2.
  apply (traV _ _ _ PA (traV _ _ _ H3
      (symV _ _ PA'))).

  intros arr2 arr1 H.
  destruct arr2 as  [w [c [d [g [P21 P22]]]]].
  destruct arr1 as  [v [a [b [f [P11 P12]]]]].
  unfold domV in H.
  unfold codV in H.
  simpl in H.
  exists (make_CmsV c d g P22 a b f P12 H).
  split.
  unfold make_CmsV.
  apply (symV _ _ P21).
  apply (symV _ _ P11).

  (* id *)

  intros u y P.
  destruct u as [x [arr1 [arr2 [PA PB]]]].
  destruct arr1 as  [v [a [b [f [P11 P12]]]]].
  destruct arr2 as  [w [c [d [g [P21 P22]]]]].
  simpl in P.
  simpl in PA.
  simpl in PB.
  unfold cmpV.
  simpl.
  assert ( 〈 〈 a, d 〉 , compose_relV a b d f g 〉 ≐ w)
as H.
  assert ( 〈 〈 a, b 〉 , f 〉 ≐ 〈 〈 y, y 〉 , identity
y 〉 ) as H3.
```

```
y ≐ y) as H5.
  apply (traV _ _ _ (symV _ _ P11) P).
  assert (〈 a, b 〉  ≐  〈 y, y 〉) as H4.
  apply (pairing_injective _ _ _ _ H3).
  assert (a  ≐ y) as H5.
  apply (pairing_injective _ _ _ _ H4).
  assert (b  ≐ y) as H6.
  apply (pairing_injective _ _ _ _ H4).
  assert (f ≐  identity y) as H7.
  apply (pairing_injective _ _ _ _ H3).
  assert (〈 〈 a, d 〉 , compose_relV a b d f g 〉 ≐
    〈 〈 c, d 〉 , g 〉) as H2.
  apply pairing_extensional.
  apply pairing_extensional.
  apply (traV _ _ _ H5 (traV _ _ _ (symV _ _  H6)
PB)).
  apply refV.
  assert (compose_relV a b d (identity y) g ≐
compose_relV a b d f g) as H8.
  apply compose_relV_extensional.
  assert (is_rel y y (identity y)) as H9.
  assert (is_function y y (identity y)) as HA.
  apply id_is_function.
  apply HA.
  apply (is_rel_extensional y y (identity y) a b
(identity y) (symV _ _ H5) (symV _ _ H6) (refV _)
H9).
  apply (is_rel_extensional c d g b d g
  (symV _ _ PB)
  (refV _) (refV _)).
  apply P22.
  apply P12.
  apply (is_rel_extensional c d g b d g
   (symV _ _ PB)
  (refV _) (refV _)).
  apply P22.
  apply (refV _).
  apply (refV _).
```

```
  apply (refV _).
  apply (refV _).
  apply (symV _ _ H7).

  apply (refV _).
  assert (compose_relV a b d (identity y) g ≐ g) as
HB.

  apply ext3.
  intros t Q.
  assert ((∃ x':V, ∃ z':V,
t ≐ 〈 x', z' 〉 ∧ x' ∈ a ∧ z' ∈ d ∧
  ∃y':V, y' ∈ b ∧ 〈 x', y' 〉 ∈ (identity y) ∧ 〈
y', z' 〉 ∈ g)) as X.
  apply compose_relV_char_2.
  assert (is_rel y y (identity y)) as H9.
  assert (is_function y y (identity y)) as HA.
  apply id_is_function.
  apply HA.
  apply (is_rel_extensional y y (identity y) a b
(identity y) (symV _ _ H5) (symV _ _ H6) (refV _)
H9).
  apply (is_rel_extensional c d g b d g
(symV _ _ PB)
(refV _) (refV _)).
  apply P22.
  apply Q.
  destruct X as [x' [z' [ar [PC [PD PE]]]]].
  destruct PE as [y' [PF [PG PH]]].
  assert ( 〈 x', z' 〉 ≐ 〈 y', z' 〉 ) as X2.
  apply pairing_extensional.
  apply (identity_lemma _ _ _ PG).
  apply refV.
  assert (t ≐ 〈 y', z' 〉 ) as X3.
  apply (traV _ _ _ ar X2).
  apply (ext1 _ _ _ X3 PH).

  intros t H.
  assert (∃ x':V, ∃ z':V
```

```
  assert (∃ x' :V, ∃ z' :V,
t ≐ 〈 x', z' 〉 ∧ x' ∈ a ∧ z' ∈ d  ∧
  ∃y':V, y' ∈ b ∧ 〈 x', y' 〉 ∈ (identity y) ∧ 〈
y', z' 〉 ∈ g) as G.
  unfold is_function in P22.
  assert (∃x' : V, ∃z' : V, x' ∈ c ∧ z' ∈ d
                ∧ t ≐ 〈 x', z' 〉 ) as X.
  apply (is_rel_pairs c d g t).
  apply P22.
  apply H.
  destruct X as [x' [z' [PC [PD PE]]]].
  exists x'.
  exists z'.
  split.
  apply PE.
  split.
  assert (c ≐ a) as H9.
  apply (traV _ _ _ (symV _ _ PB) (traV _ _ _  H6
(symV _ _ H5))).
  apply (ext2 _ _ _ PC H9).
  split. apply PD.
  exists x'.
  split.
  apply (ext2 _ _ _ PC (symV _ _ PB)).
  split.
  apply identity_lemma_2.
  assert (c ≐ y) as HA.
  apply (traV _ _ _ (symV _ _ PB) H6).
  apply (ext2 _ _ _ PC HA).
  apply (ext1 _ _ _ (symV _ _ PE) H).

  apply compose_relV_char_2.

  assert (is_rel y y (identity y)) as H9.
  assert (is_function y y (identity y)) as HA.
  apply id_is_function.
  apply HA.
  apply (is_rel_extensionl y y (identity y) a b
```

```
  apply (is_rel_extensional y y (identity y) a b
(identity y) (symV _ _ H5) (symV _ _ H6) (refV _)
H9).

  apply (is_rel_extensional c d g b d g
(symV _ _ PB)
(refV _) (refV _)).
  apply P22.
  apply G.
  apply (traV _ _ _ (symV _ _ H8) HB).
  apply (traV _ _ _  H2 (symV _ _ P21)).
  apply H.

  intros u y P.
  destruct u as [x [arr1 [arr2 [PA PB]]]].
  destruct arr1 as  [v [a [b [f [P11 P12]]]]].
  destruct arr2 as  [w [c [d [g [P21 P22]]]]].
  simpl in P.
  simpl in PA.
  simpl in PB.
  unfold cmpV.
  simpl.

  assert ( 〈 〈 a, d 〉 , compose_relV a b d f g 〉 ≐ v)
as H.
  assert ( 〈 〈 c, d 〉 , g 〉 ≐ 〈 〈 y, y 〉 , identity
y 〉 ) as H3.
  apply (traV _ _ _ (symV _ _ P21) P).
  assert ( 〈 c, d 〉  ≐  〈 y, y 〉 ) as H4.
  apply (pairing_injective _ _ _ _ H3).
  assert (c ≐ y) as H5.
  apply (pairing_injective _ _ _ _ H4).
  assert (d ≐ y) as H6.
  apply (pairing_injective _ _ _ _ H4).
  assert (g ≐ identity y) as H7.
  apply (pairing_injective _ _ _ _ H3).
  assert ( 〈 〈 a, d 〉 , compose_relV a b d f g 〉 ≐
    〈 〈 a, b 〉 , f 〉 ) as H2.
```

```
  apply pairing_extensional.
  apply pairing_extensional.
  apply refV.
  apply (traV _ _ _ H6 (traV _ _ _ (symV _ _ H5)
   (symV _ _ PB))).
  apply ext3.
(* apply tt.
apply P12. *)
  intros t Q.
  assert (∃ x:V, ∃ z:V,
  t ≐ 〈 x, z 〉 ∧ x ∈ a ∧ z ∈ d  ∧
    ∃y:V, y ∈ b ∧ 〈 x, y 〉 ∈ f ∧ 〈 y, z 〉 ∈ g).
  apply compose_relV_char_2.
  apply P12.
  apply (is_rel_extensional c d g b d g (symV _ _ PB)
(refV _) (refV _)).
  apply P22.
  apply Q.
  destruct X as  [x' [z' [PC [PD [PE PF]]]]].
  destruct PF as [y' [PG [PH PI]]].
  assert ( 〈 y', z' 〉 ∈ identity y) as H8.
  apply (ext2 _ _ _ PI H7).
  assert (y'≐ z') as H9.
  apply (identity_lemma _ _ y).
  apply H8.
  assert ( 〈 x', y' 〉 ≐ 〈 x', z' 〉 ) as HA.
  apply pairing_extensional.
  apply refV.
  apply H9.
  assert  ( 〈x', y'〉 ≐ t) as HB.
  apply (traV _ _ _ HA (symV _ _ PC)).
  apply (ext1 _ _ _ (symV _ _ HB) PH).

  intros t Q.
  assert (∃ x':V, ∃ z':V,
t ≐ 〈 x', z' 〉 ∧ x' ∈ a ∧ z' ∈ d  ∧
  ∃y':V, y' ∈ b ∧ 〈 x', y' 〉 ∈ f ∧ 〈 y', z' 〉 ∈ g)
```

```
as H8.
  assert (∃x' : V, ∃z' : V, x' ∈ a  ∧ z' ∈ b
                  ∧ t ≐ 〈 x', z' 〉 ) as X.
  apply (is_rel_pairs a b f t).
  apply P12.
  apply Q.
  destruct X as  [x' [z' [PC [PD PE ]]]].
  exists x'.
  exists z'.
  split.
  apply PE.
  split.
  apply PC.
  split.
  assert (b  ≐ d) as H8.
  apply (traV _ _ _ PB (traV _ _ _ H5 (symV _ _
H6))).
  apply (ext2 _ _ _ PD H8).
  exists z'.
  split.
  apply PD.
  split.
  apply (ext1 _ _ _ (symV _ _ PE) Q).
  assert ( 〈 z', z' 〉  ∈ identity y) as H8.
  apply identity_lemma_2.
  apply (ext2 _ _ _ PD (traV _ _ _ PB H5)).
  apply (ext2 _ _ _ H8 (symV _ _ H7)).
  apply compose_relV_char_2.
  apply P12.
  apply (is_rel_extensional c d g b d g).
  apply (symV _ _ PB).
  apply refV.
  apply refV.
  apply P22.
  apply H8.
  apply (traV _ _ _ H2 (symV _ _ P11)).
  apply H.
```

```
(* assoc *)


  intros u u' u'' u'''.
  destruct u as [x [arr1 [arr2 [PA PB]]]].
  destruct arr1 as  [v [c [d [g [P11 P12]]]]].
  destruct arr2 as  [w [i [j [h [P21 P22]]]]].
  destruct u' as [x' [arr1' [arr2' [PA' PB']]]].
  destruct arr1' as  [v' [a [b [f [P11' P12']]]]].
  destruct arr2' as  [w' [c' [d' [g' [P21' P22']]]]].

  destruct u'' as [x'' [arr1'' [arr2'' [PA''
PB'']]]].
  destruct arr1'' as  [v'' [a' [b' [f' [P11''
P12'']]]]].
  destruct arr2'' as  [w'' [c'' [j' [hg [P21''
P22'']]]]].

  destruct u''' as [x''' [arr1''' [arr2''' [PA'''
PB''']]]].
  destruct arr1''' as  [v''' [a''' [d''' [gf''
[P11''' P12''']]]]].
  destruct arr2''' as  [w''' [i''' [j''' [h' [P21'''
P22''']]]]].
  simpl in PA.
  simpl in PB.
  simpl in PA'.
  simpl in PB'.
  simpl in PA''.
  simpl in PB''.
  simpl in PA'''.
  simpl in PB'''.
  intros H1 H2 H3.
  simpl in H1.
  simpl in H2.
  simpl in H3.
  intros H4 H5.
```

```
  intros H4 H5.
  simpl in H4.
  simpl in H5.

  simpl.
  assert ( ⟨ ⟨ a', j' ⟩ , compose_relV a' b' j' f' hg
⟩ ≐  ⟨ ⟨ a''', j''' ⟩ , compose_relV a''' d''' j'''
gf'' h'⟩ ) as G.
  assert ( ⟨ ⟨ a, d' ⟩ , compose_relV a b d' f g' ⟩
≐ v''') as H5'.
  apply H5.
  clear H5.
  assert ( ⟨ ⟨ a, b ⟩ , f ⟩ ≐ ⟨ ⟨ a', b' ⟩ , f' ⟩ )
as E1.
  apply (traV _ _ _ (symV _ _  P11') (traV _ _ _
(symV _ _ H1) P11'')).
  assert ( ⟨ a, b ⟩  ≐  ⟨ a', b' ⟩ ) as E11.
  apply (pairing_injective _ _ _ _ E1).
  assert (a ≐ a') as E111.
  apply (pairing_injective _ _ _ _ E11).
  assert (b ≐ b') as E112.
  apply (pairing_injective _ _ _ _ E11).
  assert (f ≐ f') as E12.
  apply (pairing_injective _ _ _ _ E1).
  assert ( ⟨ ⟨ i, j ⟩ , h ⟩  ≐ ⟨ ⟨ i''', j''' ⟩ , h'
⟩ ) as E2.
  apply (traV _ _ _ (symV _ _ P21) (traV _ _ _ H3
P21''')).
  assert ( ⟨ i, j ⟩ ≐ ⟨ i''', j''' ⟩ ) as E21.
  apply (pairing_injective _ _ _ _ E2).
  assert (i ≐ i''') as E211.
  apply (pairing_injective _ _ _ _ E21).
  assert (j ≐ j''' ) as E212.
  apply (pairing_injective _ _ _ _ E21).
  assert (h ≐ h') as E22.
  apply (pairing_injective _ _ _ _ E2).
  assert ( ⟨ ⟨ a, d' ⟩ , compose_relV a b d' f g' ⟩
≐ ⟨ ⟨ a''', d''' ⟩ , gf'' ⟩ ) as E3.
```

```
  apply (traV _ _ _ H5' P11''').
  assert ( ⟨ a, d' ⟩ ≐ ⟨ a''', d''' ⟩ ) as E31.
  apply (pairing_injective _ _ _ _ E3).
  assert (a ≐ a''') as E311.
  apply (pairing_injective _ _ _ _ E31).
  assert (d' ≐ d''') as E312.
  apply (pairing_injective _ _ _ _ E31).
  assert (compose_relV a b d' f g' ≐ gf'') as E32.
  apply (pairing_injective _ _ _ _ E3).

  assert ( ⟨ ⟨ c, j ⟩ , compose_relV c d j g h ⟩ ≐ ⟨
 ⟨ c'', j' ⟩ , hg ⟩ ) as E4.
  apply (traV _ _ _ (symV _ _ H4) P21'').
  assert ( ⟨ c, j ⟩ ≐ ⟨ c'', j' ⟩ ) as E41.
  apply (pairing_injective _ _ _ _ E4).
  assert ( c ≐ c'') as E411.
  apply (pairing_injective _ _ _ _ E41).
  assert ( j ≐ j') as E412.
  apply (pairing_injective _ _ _ _ E41).
  assert ( compose_relV c d j g h ≐ hg ) as E42.
  apply (pairing_injective _ _ _ _ E4).
  assert ( ⟨ ⟨ c, d ⟩ , g ⟩ ≐ ⟨ ⟨ c', d' ⟩ , g' ⟩ )
as E5.
  apply (traV _ _ _ (symV _ _ P11) (traV _ _ _ (symV
_ _ H2) P21')).
  assert (g ≐ g') as E52.
  apply (pairing_injective _ _ _ _ E5).
  assert ( ⟨ c, d ⟩ ≐ ⟨ c', d' ⟩ ) as E51.
  apply (pairing_injective _ _ _ _ E5).
  assert (c ≐ c') as E511.
  apply (pairing_injective _ _ _ _ E51).
  assert (d ≐ d') as E512.
  apply (pairing_injective _ _ _ _ E51).

  apply pairing_extensional.
  apply pairing_extensional.
  apply (traV _ _ _ (symV _ _ E111) E311).
```

```
  apply (traV _ _ _ (symV _ _ E412) E212).

  apply ext3.


  intros t Q.
  assert (
  (∃ x:V, ∃ z:V,
t ≐ 〈 x, z 〉 ∧ x ∈ a' ∧ z ∈ j'  ∧
  ∃y:V, y ∈ b' ∧ 〈 x, y 〉 ∈ f' ∧ 〈 y, z 〉 ∈ hg)
) as Q'.
  apply compose_relV_char_2.
  apply P12''.
  assert (is_rel c'' j' hg) as Q''.
  apply P22''.
  apply (is_rel_extensional c'' j' hg b' j' hg
  (symV _ _ PB'') (refV _) (refV _) Q'').
  apply Q.
  assert ((∃ x:V, ∃ z:V,
   t ≐ 〈 x, z 〉 ∧ x ∈ a''' ∧ z ∈ j'''  ∧
  ∃y:V, y ∈ d''' ∧ 〈 x, y 〉 ∈ gf'' ∧ 〈 y, z 〉 ∈
h')) as G.
  destruct Q' as  [x0 [z0 [R1 [R2 R3]]]].
  exists x0.
  exists z0.
  split.
  apply R1.
  split.
  apply (ext2 _ _ _  R2 (traV _ _ _ (symV _ _ E111)
E311)).
  destruct R3 as [R4 R5].
  split.
  apply (ext2 _ _ _  R4 (traV _ _ _ (symV _ _ E412)
E212)).
  destruct R5 as [y0 [R6 [R7 R8]]].
  assert ( 〈 y0, z0 〉 ∈ compose_relV c d j g h ) as
R9.
  apply (ext2 _ _ _ R8 (symV _ _ E42)).
```

```
  assert ( y0 ∈ c  ∧ z0 ∈ j   ∧
  ∃y:V, y ∈ d ∧ 〈 y0, y 〉 ∈ g  ∧ 〈 y, z0 〉 ∈ h) as
RA.
  apply (compose_relV_char c d j g h).
  apply P12.
  assert (is_rel d j h) as RB.
  apply (is_rel_extensional i j h d j ).
  apply (symV _ _ PB).
  apply (refV _).
  apply (refV _).
  apply P22.
  apply RB.
  apply R9.
  destruct RA as [RB [RC [y1 RD]]].
  exists y1.
  split.
  assert (d ≐ d''') as RE.
  apply (traV _ _ _ PB (traV _ _ _ E211 (symV _ _
PB'''))).
  apply (ext2 _ _ _ (fst RD) RE).
  split.
  assert (  〈 x0, y1 〉 ∈ compose_relV a b d' f g') as
RF.
  assert ( x0 ∈ a  ∧ y1 ∈ d'   ∧
  ∃y:V, y ∈ b ∧ 〈 x0, y 〉 ∈ f  ∧ 〈 y, y1 〉 ∈ g') as
RG.
  split.
  apply (ext2 _ _ _ R2 (symV _ _ E111)).
  split.
  assert (d  ≐ d') as RH.
  apply (traV _ _ _ PB (traV _ _ _ E211
(traV _ _ _ (symV _ _ PB''') (symV _ _ E312)))).
  apply (ext2 _ _ _ (fst RD) RH).
  exists y0.
  split.
  apply (ext2 _ _ _ R6 (symV _ _ E112)).
  split.
```

```
  apply (ext2 _ _ _ R7 (symV _ _ E12)).
  apply (ext2 _ _ _ (fst (snd RD)) E52).
  apply compose_relV_char.

  apply P12'.
  assert (is_rel c' d' g') as RH.
  apply P22'.
  apply (is_rel_extensional c' d' g' b d' g').
  apply (symV _ _ PB').
  apply refV.
  apply refV.
  apply RH.
  apply RG.
  apply (ext2 _ _ _ RF E32).
  apply (ext2 _ _ _ (snd (snd RD)) E22).
  apply compose_relV_char_2.
  apply P12'''.
  assert (is_rel i''' j''' h') as RJ.
  apply P22'''.
  apply (is_rel_extensional i''' j''' h' d''' j'''
h').
  apply (symV _ _ PB''').
  apply refV.
  apply refV.
  apply RJ.
  apply G.

  intros t Q.
  assert (∃ x:V, ∃ z:V,
   t ≐ 〈 x, z 〉 ∧ x ∈ a''' ∧ z ∈ j'''  ∧
  ∃y:V, y ∈ d''' ∧ 〈 x, y 〉 ∈ gf'' ∧ 〈 y, z 〉 ∈
h') as Q'.
  apply compose_relV_char_2.
  apply P12'''.
  assert (is_rel i''' j''' h') as R2.
  apply P22'''.
  apply (is_rel_extensional i''' j''' h' d''' j'''
h').
```

```
  apply (symV _ _ PB''').
  apply refV.
  apply refV.
  apply R2.
  apply Q.
  assert (∃ x:V, ∃ z:V,
t ≐ 〈 x, z 〉 ∧ x ∈ a' ∧ z ∈ j'  ∧
  ∃y:V, y ∈ b' ∧ 〈 x, y 〉 ∈ f' ∧ 〈 y, z 〉 ∈ hg) as
G.
  destruct Q' as  [x0 [z0 [R4 [R5 [R6 R7]]]]].
  destruct R7 as [y0 [R8 [R9 RA]]].
  exists x0.
  exists z0.
  split.
  apply R4.
  split.
  apply (ext2 _ _ _ R5).
  apply (traV _ _ _ (symV _ _ E311) E111).
  split.
  apply (ext2 _ _ _ R6).
  apply (traV _ _ _ (symV _ _ E212) E412).
  assert ( 〈 x0, y0 〉 ∈ compose_relV a b d' f g') as
RB.
  apply (ext2 _ _ _ R9).
  apply (symV _ _ E32).
  assert (
x0 ∈ a  ∧ y0 ∈ d'  ∧
  ∃y:V, y ∈ b ∧ 〈 x0, y〉 ∈ f ∧ 〈 y, y0〉 ∈ g') as
RB'.
  apply compose_relV_char.
  apply P12'.
  assert (is_rel c' d' g') as RC.
  apply P22'.
  apply (is_rel_extensional c' d' g' b d' g').
  apply (symV _ _ PB').
  apply refV.
  apply refV.
```

```
  apply RC.
  apply RB.
  destruct RB' as [RC [RD [y1 [RE [RF RG]]]]].
  exists y1.
  split.
  apply (ext2 _ _ _ RE E112).
  split.
  apply (ext2 _ _ _ RF E12).
  assert ( 〈 y1, z0 〉  ∈ compose_relV c d j g h) as
RH.
  assert (
y1 ∈ c  ∧ z0 ∈ j   ∧
  ∃y:V, y ∈ d ∧  〈 y1, y〉  ∈ g  ∧  〈 y, z0〉  ∈ h) as
RB'.
  split.
  apply (ext2 _ _ _ RE (traV _ _ _ PB' (symV _ _
E511))).
  split.
  apply (ext2 _ _ _ R6 (symV _ _ E212)).
  exists y0.
  split.
  apply (ext2 _ _ _ RD (symV _ _ E512)).
  split.
  apply (ext2 _ _ _ RG (symV _ _ E52)).
  apply (ext2 _ _ _ RA (symV _ _ E22)).
  apply compose_relV_char.
  apply P12.
  assert (is_rel i j h) as RJ.
  apply P22.
  apply (is_rel_extensional i j h d j h).
  apply (symV _ _ PB).
  apply refV.
  apply refV.
  apply RJ.
  apply RB'.
  apply (ext2 _ _ _ RH E42).
```

```
  apply compose_relV_char_2.
  apply P12''.
  assert (is_rel c'' j' hg) as R3.
  apply P22''.
  apply (is_rel_extensional c'' j' hg b' j' hg).
  apply (symV _ _ PB'').
  apply refV.
  apply refV.
  apply R3.
  apply G.
  apply G.
Defined.

(* Another category is built from Rbar using
  the followoing general construction. *)



(*

       The construction of  a category from a
       proof-irrelevant big family of setoids.

*)


(* Ad hoc definitions Big families *)

Notation "p ⁻¹" := (Typeoidsym _ _ _ p) (at level 3,
no associativity).
Notation "p ⊙ q" := (Typeoidtra _ _ _ _ q p) (at
level 34, right associativity).
Notation "F ↱ p" := (Typeoidmapextensionality _ _ F
_ _ p) (at level 100).
```

```
Notation "F • p" := (Typeoidfamilymap _ F _ _ p)
  (at level 9, right associativity).


Lemma Typeoidfamilyrefgeneral {A: Typeoid} (F:
Typeoidfamily A):
  ∀x: A, ∀p: x  ≈≈≈ x, ∀y: F x, F•p y  ≈≈≈ y.
Proof.
  intros x p y.
  apply Typeoidtra with (F•(Typeoidrefl A x) y);
  eauto using Typeoidtra, Typeoidfamilyirr,
Typeoidfamilyref, Typeoidrefl.
Defined.

Lemma Typeoidfamilycmpgeneral {A: Typeoid} (F:
Typeoidfamily A):
  ∀x y z: A, ∀p: x  ≈≈≈ y, ∀q: y  ≈≈≈ z, ∀r: x  ≈≈≈
z, ∀w: F x, F•q (F•p w)  ≈≈≈ F•r w.
Proof.
  intros x y z p q r w.
  assert (F • q (F • p w) ≈≈≈ (F • (q ⊙ p) w)) as H.
  apply Typeoidfamilycmp.
  eauto using Typeoidtra, Typeoidfamilyirr,
Typeoidfamilycmp.
Defined.


Lemma Typeoidfamilycmp3general {A: Typeoid} (F:
Typeoidfamily A):
  ∀x y z, ∀u: A, ∀p: x  ≈≈≈ y, ∀q: y  ≈≈≈ z, ∀s:
z  ≈≈≈ u,
  ∀r: x  ≈≈≈ u,
∀w: F x, F•s (F•q (F•p w))  ≈≈≈ F•r w.
Proof.
  intros x y z u p q s r w.
  assert (F • s (F • q (F • p w)) ≈≈≈
          F • (s ⊙ q) (F • p w)) as H1.
  apply Typeoidfamilycmp.
```

```
  assert ( F ● (s ⊙ q) (F ● p w)  ≈≈≈ F ● r w) as H2.
  apply Typeoidfamilycmpgeneral.
      eauto using Typeoidtra, Typeoidfamilyirr,
Typeoidfamilycmp.
Defined.


Lemma Typeoidfamilycmpinvert
{A: Typeoid} (F: Typeoidfamily A):
  ∀x y: A, ∀p: x ≈≈≈ y, ∀q: y ≈≈≈ x, ∀w: F x, F●q
(F●p w) ≈≈≈ w.
Proof.
  intros x y p q w.
  assert  (F ● q (F ● p w) ≈≈≈,{ F x } F ●
(Typeoidrefl A x) w) as H1.
  apply Typeoidfamilycmpgeneral.
  assert (F ● (Typeoidrefl A x) w ≈≈≈,{ F x } w) as
H2.
  apply Typeoidfamilyrefgeneral.
  apply (Typeoidtra _ _ _ _ H1 H2).
Defined.

(* Here are some tactics for Typeoids
  similar to those for setoids *)

Hint Resolve Typeoidrefl : Swesetoid.
Hint Immediate Typeoidsym : Swesetoid.
Hint Resolve Typeoidtra : Swesetoid. (* The warning
here is expected *)

Hint Resolve Typeoidmapextensionality : swesetoid.


Ltac Swesetoid := simpl; eauto with Swesetoid.
```

```
(* Some lemmas about proof-irrelevant families *)

Lemma Typeoidfamilyirrgeneral
  {A: Typeoid} (F: Typeoidfamily A):
  ∀ x y: A,  ∀ p q: x ≈≈≈ y, ∀u v: F x,
  u  ≈≈≈ v ->  F●p u ≈≈≈ F●q v.
Proof.
  intros x y p q u v H.
  assert (F ● p u ≈≈≈,{ F y } F ● q u) as H1.
  apply Typeoidfamilyirr.
  assert (F ● q u ≈≈≈,{ F y } F ● q v) as H2.
  apply Typeoidmapextensionality.
  apply H.
  apply (Typeoidtra _ _ _ _ H1 H2).
Defined.

Lemma Typeoidfamilyirrgeneraldouble
  {A: Typeoid} (F: Typeoidfamily A):
  ∀ x y: A, ∀ z w: A,
  ∀ p: x ≈≈≈ z,  ∀ p': z ≈≈≈ y,
  ∀ q: x ≈≈≈ w,  ∀ q': w ≈≈≈ y,
  ∀u v: F x,
     u  ≈≈≈ v ->  F●p' (F●p u) ≈≈≈  F●q' (F●q v).
Proof.
  intros x y z w p p' q q' u v H.
  assert (F ● p' (F ● p u) ≈≈≈  F ● (p'⊙ p) u).
  apply Typeoidfamilycmp.
  assert (F ● q' (F ● q v) ≈≈≈  F ● (q'⊙ q) v).
  apply Typeoidfamilycmp.
  assert (F ● (p'⊙ p) u ≈≈≈  F ● (q'⊙ q) v ).
  apply Typeoidfamilyirrgeneral.
  exact H.
  Swesetoid.
Defined.


Lemma Typeoidfamilycmpgeneral_3
```

```
{A: Typeoid} (F: Typeoidfamily A):
∀ x: A,  ∀ v z: A, ∀ r: x ≈≈≈ v, ∀ s: v ≈≈≈ z,
   ∀u:F z , ∀w: F x, u ≈≈≈  F●(s⊙r) w -> u ≈≈≈  F●s
(F●r w).
Proof.
  intros x v z r s u w H.
  assert (F●s (F●r w) ≈≈≈  F●(s⊙r) w).
  apply Typeoidfamilycmp.
  Swesetoid.
Defined.


Lemma Typeoidfamilycmptactical
  {A: Typeoid} (F: Typeoidfamily A):
  ∀ x y z: A,
  ∀ p: x ≈≈≈ z,  ∀ p': z ≈≈≈ y,
  ∀u: F x, ∀v: F y,
     F ● (p'⊙ p) u  ≈≈≈ v ->  F●p' (F●p u) ≈≈≈  v.
intros x y z p p' u v H.
assert (F●p' (F●p u)  ≈≈≈ F ● (p' ⊙ p) u) as H2.
apply Typeoidfamilycmp.
Swesetoid.
Defined.



(* Build the set or arrows from a family *)

Definition arr_from_family {A:Typeoid} (F:
Typeoidfamily A): Typeoid.
  apply
  (Build_Typeoid (∃a: A, (∃b: A, Typeoidmap (F a) (F
b)))
   (λ p q, match (p, q) with
    (existT a (existT b f), existT a' (existT b' f'))
=>
```

```
          ∃e: a ≈≈≈ a',  ∃d: b ≈≈≈ b',  ∀x:(F a),
          F●d (f x)  ≈≈≈ f' (F●e x)
               end)).

  intros [a [b f]].
  exists (Typeoidrefl _ _).
  exists (Typeoidrefl _ _).
  intro x.
  assert ( f x ≈≈≈,{ F b }f (F ● (Typeoidrefl A a)
x)).
  apply Typeoidmapextensionality.
  apply Typeoidsym.
  apply Typeoidfamilyrefgeneral.
  assert (F ● (Typeoidrefl A b) (f x) ≈≈≈,{ F b } (f
x)).
  apply Typeoidfamilyrefgeneral.
  Swesetoid.
  intros [a [b f]].
  intros [a' [b' f']].
  intros  [e [d H2]].
  exists e⁻¹.
  exists d⁻¹.
  intro x.
  apply Typeoidsym.
  assert (F ● d (f (F ● e ⁻¹ x)) ≈≈≈,{ F b' }f'
                              (F ● e (F ● e ⁻¹ x))).
  apply H2.
  assert (F ● d⁻¹ (F ● d (f (F ● e ⁻¹ x))) ≈≈≈,{ F b
}
                        F ● d⁻¹ (f' (F ● e (F ● e ⁻¹
x)))).
  apply Typeoidmapextensionality.
  assumption.
  assert (F ● d ⁻¹ (F ● d (f (F ● e ⁻¹ x))) ≈≈≈,{ F b
}  f (F ● e ⁻¹ x)).
  assert (F ● d ⁻¹ (F ● d (f (F ● e ⁻¹ x))) ≈≈≈,{ F b
}  F ● (Typeoidrefl A b) (f (F ● e ⁻¹ x))).
  apply Typeoidfamilycmpgeneral.
```

```
  assert (F ● (Typeoidrefl A b) (f (F ● e ⁻¹ x)) ≈≈≈,
{ F b } f (F ● e ⁻¹ x)).
  apply Typeoidfamilyref.

  Swesetoid.
  assert  (f (F ● e ⁻¹ x) ≈≈≈,{ F b } F ● d ⁻¹ (f' (F
● e (F ● e ⁻¹ x)))).
  Swesetoid.
  assert (F ● d ⁻¹ (f' (F ● e (F ● e ⁻¹ x))) ≈≈≈,{ F
b }F ● d ⁻¹ (f' x)).
  apply Typeoidmapextensionality.
  apply Typeoidmapextensionality.
  assert (F ● e (F ● e ⁻¹ x) ≈≈≈,{ F a' }  F ●
(Typeoidrefl A a') x).
  apply Typeoidfamilycmpgeneral.
  assert (F ● (Typeoidrefl A a') x  ≈≈≈,{ F a' } x).
  apply Typeoidfamilyref.
  Swesetoid. Swesetoid.
  intros [a [b f]].
  intros [a' [b' f']].
  intros [a'' [b'' f'']].
  intros [e [d H]].
  intros [e' [d' H']].

  exists (e'⊙e).
  exists (d'⊙d).
  intro x.
  apply Typeoidtra with  (F ● d' ((F ● d) (f x))).
  apply Typeoidsym.
  apply Typeoidfamilycmp.
  apply Typeoidtra with (f'' (F ● e' ((F ● e) x))).

  specialize (H' ((F ● e) x)).
  specialize (H x).
  assert (F ● d' (f' (F ● e x)) ≈≈≈ F ● d' ( F ● d (f
x))) as H2.
  apply Typeoidmapextensionality.
  apply (Typeoidsym       H).
```

```
  apply (Typeoidtra _ _ _ _ (Typeoidsym _ _ _ H2)
H').


  apply Typeoidmapextensionality.
  apply Typeoidfamilycmp.
Defined.


(* The id-map on a Typeoid *)

Definition Idmapp (A: Typeoid): Typeoidmap A A.
  apply (Build_Typeoidmap A A (λ x: A, x));
Swesetoid.
Defined.


(* The id-map on a setoid as an arrow *)

Definition catid_from_family_helper {A:Typeoid}
(F: Typeoidfamily A)(a: A) : arr_from_family F.
  exists a.
  exists a.
  apply (Idmapp (F a)).
Defined.

(* The id creating map in the category *)

Definition catid_from_family
  {A:Typeoid} (F: Typeoidfamily A):
          Typeoidmap A  (arr_from_family F).
  apply (Build_Typeoidmap A (arr_from_family F)
  (λ a:A, catid_from_family_helper F a)).
  intros x y H.
  exists H. exists H.
  intro t.
  Swesetoid.
Defined.
```

```
Defined.

(* The domain map in the category *)

Definition catdom_from_family_helper
{A:Typeoid}(F: Typeoidfamily A)(f:arr_from_family F):
A.
  destruct f as [a s].
  exact a.
Defined.

Definition catdom_from_family
  {A:Typeoid} (F: Typeoidfamily A):
  Typeoidmap (arr_from_family F) A.
  apply (Build_Typeoidmap  (arr_from_family F) A
        (λ a, catdom_from_family_helper F a)).
  intros  [a s] [b t] H.
  destruct s.
  destruct t.
  destruct H.
  Swesetoid.
Defined.

(* The codomain map in the category *)

Definition catcod_from_family_helper
  {A:Typeoid} (F: Typeoidfamily A)(f:arr_from_family
F): A.
  destruct f as [a  [b t]].
  exact b.
Defined.

Definition catcod_from_family
  {A:Typeoid} (F: Typeoidfamily A):
  Typeoidmap (arr_from_family F)  A.
  apply (Build_Typeoidmap  (arr_from_family F) A
        (λ a, catcod_from_family_helper F a)).
  intros [a s] [b t] H
```

```
  intros [a s] [b t] H.
  destruct s.
  destruct t.
  destruct H as [p [q H]].

  Swesetoid.
Defined.


(* Construction of the setoid of composable arrows
   These are pairs

   (g f) : s.t. cod g =  dom f

*)

Definition cms_from_family
  {A:Typeoid} (F: Typeoidfamily A): Typeoid.
  apply (Build_Typeoid (∃g: arr_from_family F,
                        ∃f: arr_from_family F,
                        catcod_from_family F g ≈≈≈
                        catdom_from_family F f)
      (λ p q, match (p, q) with
            (existT g (existT f p),
             existT g' (existT f' p'))
                => g ≈≈≈ g' ∧ f ≈≈≈ f'
             end)).
  intros [g [f H]].
  split.
  apply Typeoidrefl.
  apply Typeoidrefl.

  intros [g [f H]].
  intros [g1 [f1 H1]].
  intros [H21 H22].
  split.
  apply Typeoidsym. assumption.
  apply Typeoidsym. assumption.
```

```
  intros  [g1 [f1 H1]].
  intros  [g2 [f2 H2]].

  intros  [g3 [f3 H3]].
  intros  [H41 H42].
  intros  [H51 H52].
  split.
  apply Typeoidtra with g2. assumption. assumption.
  apply Typeoidtra with f2. assumption. assumption.
Defined.

(* Map picking out the first map of a composable pair
*)

Definition catfst_from_family_helper
  {A:Typeoid} (F: Typeoidfamily A)(u: cms_from_family
F):
    arr_from_family F.
  destruct u as  [g [f u]].
  exact g.
Defined.

Definition catfst_from_family
  {A:Typeoid} (F: Typeoidfamily A):
    Typeoidmap (cms_from_family F)
               (arr_from_family F).
  apply (Build_Typeoidmap
        (cms_from_family F)
        (arr_from_family F)
        (λ u,  catfst_from_family_helper F u)).
intros [g [f u]] [g' [f' v]] [H1 H2].
  exact H1.
Defined.

(* Map picking out the second map of a composable
pair *)
```

```
Definition catsnd_from_family_helper
  {A:Typeoid} (F: Typeoidfamily A)(u: cms_from_family
F):

     arr_from_family F.
  destruct u as  [g [f u]].
  exact f.
Defined.

Definition catsnd_from_family
  {A:Typeoid} (F: Typeoidfamily A):
    Typeoidmap (cms_from_family F)
               (arr_from_family F).
  apply (Build_Typeoidmap
        (cms_from_family F)
        (arr_from_family F)
        (λ u,  catsnd_from_family_helper F u)).
  intros [g [f u]]  [g' [f' v]] [H1 H2].
  exact H2.
Defined.

(* Composition for Typeoidmaps *)

Definition Typeoidmaps (A B: Typeoid): Typeoid.
apply (Build_Typeoid (Typeoidmap A B) (λ f g, ∀x:A, f
x ≈≈≈ g x)); Swesetoid.
Defined.

Lemma bintypeoidmap_helper {A B C: Typeoid} (f:A → B
→ C)
  (p: ∀a a', a ≈≈≈ a' → ∀b, f a b ≈≈≈ f a' b)
  (q: ∀a b b', b ≈≈≈ b' → f a b ≈≈≈ f a b')
  : (Typeoidmap A (Typeoidmaps B C)).
Proof.
  apply (Build_Typeoidmap A (Typeoidmaps B C)
   (λ a, (Build_Typeoidmap B C (f a) (q a))) p).
Defined.
```

```
Lemma trintypeoidmap_helper {A B C D: Typeoid} (f: A
→ B → C → D)
  (p: ∀a a', a ≈≈≈ a' → ∀b c, f a b c ≈≈≈ f a' b c)
  (q: ∀a b b', b ≈≈≈ b' → ∀c, f a b c ≈≈≈ f a b' c)
  (r: ∀a b c c', c ≈≈≈ c' → f a b c ≈≈≈ f a b c')
  : (Typeoidmap A (Typeoidmaps B (Typeoidmaps C D))).
Proof.
  apply (Build_Typeoidmap A (Typeoidmaps B
(Typeoidmaps C D))) with (λ a, bintypeoidmap_helper
(f a) (q a) (r a));
    assumption.
Defined.


Definition Comp {A B: Typeoid} (C: Typeoid):
Typeoidmap (Typeoidmaps B C)
           (Typeoidmaps (Typeoidmaps A B)
                        (Typeoidmaps A C)).
apply trintypeoidmap_helper with (λ f: (Typeoidmap B
C), λ g: (Typeoidmap A B), λ a, f (g a)).
  Swesetoid.
  intros.
  apply Typeoidmapextensionality.
  Swesetoid.
  intros.
  apply Typeoidmapextensionality.
  apply Typeoidmapextensionality.
  Swesetoid.
Defined.

(* The composition map in the category *)

Definition catcmp_from_family_helper
  {A:Typeoid} (F: Typeoidfamily A) (u:
cms_from_family F):
    arr_from_family F.
  destruct u as [[a [b g]] [[c [d f]] p]].
```

```
  destruct u as [[a [b g]] [[c [d f]] p]].
  exists a.
  exists d.
  apply (Comp _ f (Comp _ (F•p) g)).
Defined.

Definition catcmp_from_family
  {A:Typeoid} (F: Typeoidfamily A):
    Typeoidmap (cms_from_family F)
               (arr_from_family F).
  apply (Build_Typeoidmap
        (cms_from_family F)
        (arr_from_family F)
        (λ u,  catcmp_from_family_helper F u)).
  intros [[c [d g]] [[a [b f]] p]]
         [[c'[d' g']] [[a' [b' f']] p']].
  simpl.
  intros [[q [r H1]] [s [t H2]]].

  exists q. exists t.
  intro x.
  assert (F • t (f (F • p (g x))) ≈≈≈,{ F b' }
         f' (F • s (F • p ( g x )))).
  apply H2.
  assert  (f' (F • p' ( F • r (g x)))  ≈≈≈,{ F b' }
         f' (F • p' (g' (F • q x)))).
  apply Typeoidmapextensionality.
  apply Typeoidmapextensionality.
  apply H1.
  assert (f' (F • s (F • p (g x)))  ≈≈≈,{ F b' }
          f' (F • p' (F • r (g x))) ).
  apply Typeoidmapextensionality.
  assert ((F • s (F • p (g x))) ≈≈≈,{ F a' } F •
(s⊙p) (g x)).
  apply Typeoidfamilycmp.
  assert ((F • p' (F • r (g x))) ≈≈≈,{ F a' } F •
(p'⊙r) (g x)).
  apply Typeoidfamilycmp.
```

```
  apply Typeoidfamilycmp.
  assert (F ● (p'⊙r) (g x) ≈≈≈,{ F a' } F ● (s⊙p) (g
x)).
  apply Typeoidfamilyirr.

  Swesetoid.
  Swesetoid.
Defined.

Definition cms_from_family_builder
  {A:Typeoid} (F: Typeoidfamily A)
  (g f : arr_from_family F)
  (p: catcod_from_family F g ≈≈≈ catdom_from_family F
f):
  cms_from_family F.
  exists g.
  exists f.
  assumption.
Defined.

(*

  Now combine all the above components
  to build the category and prove that they
  satisfies the laws.

  *)

Definition Cat_from_family
  {A:Typeoid} (F: Typeoidfamily A): Cat.
  apply (Build_Cat A (arr_from_family F)
                     (cms_from_family F)
                     (catid_from_family F)
                     (catdom_from_family F)
                     (catcod_from_family F)
                     (catfst_from_family F)
                     (catsnd_from_family F)
                     (catcmp_from_family F)
```

```
      ).

  (* K1 *)
  intro x. apply Typeoidrefl.

  (* K2 *)
  intro x. apply Typeoidrefl.
  (* K3 *)
  intros [g  [f p]].
  destruct g as [c [d g]].
  destruct f as [a [b f]].
  simpl.
  apply Typeoidrefl.
  (* K4 *)
   intros [g  [f p]].
  destruct g as [c [d g]].
  destruct f as [a [b f]].
  simpl.
  apply Typeoidrefl.
  (* K5 *)
  intros u u'.
  intros H1 H2.
  destruct u as  [g  [f p]].
  destruct u' as [g' [f' p']].
  Swesetoid.
  (* K6 *)
  intros f g.
  intro H.
  assert ((catcod_from_family F) g ≈≈≈,{ A }
(catdom_from_family F) f) as H1.
  apply Typeoidsym.
  apply H.
  exists (cms_from_family_builder F g f H1).
  split. apply Typeoidrefl. apply Typeoidrefl.
  (* K7 *)
  intro u.
  intro y.
  intro H.
  destruct u as [a [f p]].
```

```
  destruct u as  [g [t p]].
  destruct g as  [c [d g]].
  destruct f as  [a [b f]].
  (* p: d = a *)
  destruct H as [e [k H]].
  simpl.
  assert ( c ≈≈≈,{ A }a) as H2.
  Swesetoid.
  exists H2.
  assert (b  ≈≈≈,{ A } b) as H3.
  apply Typeoidrefl.
  exists H3.
  intro x.
  assert (∀x : F c, F • k (g x) ≈≈≈,{ F y } F • e x)
as H5.
  apply H.
  clear H.
  assert  (F • H3 (f (F • p (g x)))  ≈≈≈,{ F b }
  f (F • p (g x))) as H6.
  apply Typeoidfamilyrefgeneral.
  assert (f (F • p (g x))  ≈≈≈,{ F b } f (F • H2 x))
as H7.
  apply Typeoidmapextensionality.
  assert (y ≈≈≈,{ A } a) as q.
  Swesetoid.
  assert (F • q ( F • k (g x)) ≈≈≈,{ F a } F • q (F •
e x)) as H8.
  apply Typeoidmapextensionality.
  apply H5.
  clear H5.
  assert (F • q (F • k (g x)) ≈≈≈,{ F a }  F • p (g
x)).
  apply Typeoidfamilycmpgeneral.
  assert (F • q (F • e x) ≈≈≈,{ F a } F • H2 x).
  apply Typeoidfamilycmpgeneral.
  Swesetoid.
  Swesetoid.
```

```
  (* K8 *)
  intro u.
  intro y.
  intro H.

  destruct u as  [g  [f p]].
  destruct g as  [c [d g]].
  destruct f as  [a [b f]].
  (* p: d = a *)
  destruct H as [e [k H]].
  simpl.
  assert ( c ≈≈≈,{ A }c) as H2.
  apply Typeoidrefl.
  exists H2.
  assert (b  ≈≈≈,{ A } d) as H3.
  apply Typeoidtra with y.
  assumption.
  apply Typeoidsym.
  Swesetoid.
  exists H3.
  intro x.
  assert (g (F • H2 x) ≈≈≈,{ F d } g x) as H0.
  apply Typeoidmapextensionality.
  apply Typeoidfamilyrefgeneral.
  assert (F • H3 (f (F • p (g x))) ≈≈≈,{ F d } g x)
as H4.
  assert (F • k (f (F • p (g x))) ≈≈≈,{ F y }(F • e
(F • p (g x)))) as H5.
  apply H.
  clear H.
  assert (y ≈≈≈,{ A } d) as H6.
  Swesetoid.
  assert ( F • H3 (f (F • p (g x))) ≈≈≈,{ F d }
   F • H6 (F • k (f (F • p (g x))))) as H7.
  apply Typeoidsym.
  apply Typeoidfamilycmpgeneral.
  assert (F • H6 (F • k (f (F • p (g x)))) ≈≈≈,{ F d
}
```

```
   (F ● H6 (F ● e (F ● p (g x))))) as H8.
  apply Typeoidmapextensionality.
  assumption.
  assert ( F ● H6 (F ● e (F ● p (g x))) ≈≈≈,{ F d } g
x ) as H9.
  assert ( F ● H6 (F ● e (F ● p (g x)))  ≈≈≈,{ F d }
   F ● (H6  ⊙ e) (F ● p (g x))) as HA.
  apply Typeoidfamilycmpgeneral.
  assert (F ● (H6 ⊙ e) (F ● p (g x)) ≈≈≈,{ F d }g x).
  apply Typeoidfamilycmpinvert.
  Swesetoid.
  Swesetoid.
  Swesetoid.
  (* K9 *)
  intros u v w z.
  destruct u as  [u1 [u2 up]].
  destruct u1 as  [u1d [u1c u1]].
  destruct u2 as  [u2d [u2c u2]].
  destruct v as  [v1 [v2 vp]].
  destruct v1 as  [v1d [v1c v1]].
  destruct v2 as  [v2d [v2c v2]].
  destruct w as  [w1 [w2 wp]].
  destruct w1 as  [w1d [w1c w1]].
  destruct w2 as  [w2d [w2c w2]].
  destruct z as  [z1 [z2 zp]].
  destruct z1 as  [z1d [z1c z1]].
  destruct z2 as  [z2d [z2c z2]].
  simpl.
  intros H1 H2 H3 H4 H5.
  destruct H1 as [p1 [q1 H1]].
  destruct H2 as [p2 [q2 H2]].
  destruct H3 as [p3 [q3 H3]].
  destruct H4 as [p4 [q4 H4]].
  destruct H5 as [p5 [q5 H5]].
  assert (w1d ≈≈≈,{ A }z1d) as p6.
  Swesetoid.
  exists p6.
```

```
  assert (w2c ≈≈≈,{ A }z2c) as q6.
  Swesetoid.
  exists q6.
  intro x.


  assert  (∀x, F • q5 (v2 (F • vp (v1  (F • p5 ⁻¹ x)
))) ≈≈≈,{ F z1c }z1 x) as H5'.
  intro x0.
  (* Fin *)
  assert (F • q5 (v2 (F • vp (v1 (F • p5 ⁻¹ x0))))
≈≈≈,{ F z1c } z1 (F • p5 (F • p5 ⁻¹ x0))) as H5''.
  apply H5.
  assert (z1 (F • p5 (F • p5 ⁻¹ x0))  ≈≈≈,{ F z1c }z1
x0).
  apply Typeoidmapextensionality.
  apply Typeoidfamilycmpinvert.
  Swesetoid.
  clear H5.

  assert ( ∀x : F w2d, w2 x ≈≈≈
                   F • q4 ⁻¹ ( u2 (F • up (u1 (F • p4
x)))) ) as H4'.
  intro x0.
  assert ( ∀x : F w2d, F • q4 ⁻¹ (F • q4 (w2 x)) ≈≈≈
     F • q4 ⁻¹ ( u2 (F • up (u1 (F • p4 x)))) ).
  intro x1.
  apply Typeoidmapextensionality.
  apply H4.
  assert (w2 x0 ≈≈≈,{ F w2c } F • q4 ⁻¹ (F • q4 (w2
x0))).
  apply Typeoidsym.
  apply Typeoidfamilycmpinvert.
  Swesetoid.
  clear H4.

  assert (∀x, F • q3 (u2 (F • p3 ⁻¹ x)) ≈≈≈,{ F z2c
}z2 x  ) as H3'.
```

```
  assert (∀x, F ● q3 (u2 (F ● p3 ⁻¹ x)) ≈≈≈,{ F z2c
}z2  (F ● p3 (F ● p3 ⁻¹ x))).
  intro x0.
  apply H3.
  intro x0.
  assert ( z2 (F ● p3 (F ● p3 ⁻¹ x0)) ≈≈≈ z2 x0).
  apply Typeoidmapextensionality.
  apply Typeoidfamilycmpinvert.
  Swesetoid.
  clear H3.
  assert (∀x : F w1d, w1 x ≈≈≈ F ● q1 ⁻¹ (v1 (F ● p1
x))) as H1'.
  intro x0.
  assert (w1 x0 ≈≈≈,{ F w1c }F ● q1 ⁻¹ (F ● q1 ( w1
x0))).
  apply Typeoidsym.
  apply Typeoidfamilycmpinvert.
    specialize (H1 x0).
    assert (F ● q1 ⁻¹ (F ● q1 (w1 x0))  ≈≈≈ F ● q1 ⁻¹
(v1 (F ● p1 x0))) as H99.
    apply Typeoidmapextensionality.
    apply H1.
  Swesetoid.
  clear H1.
  assert (F ● q6 (w2 (F ● wp (w1 x))) ≈≈≈,{ F z2c }
F ● q6 (F ● q4 ⁻¹ (u2 (F ● up (u1 (F ● p4 (F ● wp (w1
x)))))))) as H6.
  apply Typeoidmapextensionality.
  apply H4'.
  clear H4'.
  assert (z2 (F ● zp (z1 (F ● p6 x)))
≈≈≈,{ F z2c }  F ● q3 (u2 (F ● p3 ⁻¹ (F ● zp (z1 (F ●
p6 x))))))
as H7.
  apply Typeoidsym.
  apply H3'.
```

```
  clear H3'.
  assert ((F • up (u1 (F • p4 (F • wp (w1 x)))))
≈≈≈,{ F u2d }  (F • p3 ⁻¹ (F • zp (z1 (F • p6 x)))))
as H8.


  assert (
F • p3 ⁻¹ (F • zp (F • q5 (v2 (F • vp (v1 (F • p5
⁻¹  (F • p6 x))))))
≈≈≈,{ F u2d }
   F • p3 ⁻¹ (F • zp (z1 (F • p6 x)))) as H9.
  apply Typeoidmapextensionality.
  apply Typeoidmapextensionality.
  simpl.
  apply H5'.

  assert ( F • up (u1 (F • p4 (F • wp (w1 x)))) ≈≈≈,{
F u2d }
F • p3 ⁻¹ (F • zp (F • q5 (v2 (F • vp (v1 (F • p5 ⁻¹
(F • p6 x)))))))).
  clear H9.
  assert (
F • up (u1 (F • p4 (F • wp (w1 x)))) ≈≈≈,{ F u2d }
F • up (u1 (F • p4 (F • wp (F • q1 ⁻¹ (v1 (F • p1
x)))))) as
HA.
  apply Typeoidmapextensionality.
  apply Typeoidmapextensionality.
  apply Typeoidmapextensionality.
  apply Typeoidmapextensionality.
  Swesetoid.
  assert (
F • up (u1 (F • p4 (F • wp (F • q1 ⁻¹ (v1 (F • p1
x))))))
≈≈≈,{ F u2d }
   F • p3 ⁻¹ (F • zp (F • q5 (v2 (F • vp (v1 (F • p5
⁻¹ (F • p6 x)))))))) as HB.
  assert (  ∀x : F v2d,  v2 x ≈≈≈  F • q2 ⁻¹( u1 (F •
```

```
p2 x)) ) as H2'.
  intro x0.
  assert (v2 x0 ≈≈≈ F • q2 ⁻¹(F • q2 (v2 x0))) as HX.
  apply Typeoidsym.

  apply Typeoidfamilycmpinvert.
  assert (
F • q2 ⁻¹ (F • q2 (v2 x0))
≈≈≈,{ F v2c }F • q2 ⁻¹ (u1 (F • p2 x0)))  as HY.
  apply Typeoidmapextensionality.
  apply H2.
  Swesetoid.
  assert ( F • up (u1 (F • p4 (F • wp (F • q1 ⁻¹ (v1
(F • p1 x)))))) ≈≈≈,{
   F u2d }
F • p3 ⁻¹ (F • zp (F • q5 (F • q2 ⁻¹ (u1 (F • p2 (F •
vp (v1 (F • p5 ⁻¹ (F • p6 x)))))))))) as HZ.

  apply Typeoidfamilycmpgeneral_3.
  apply Typeoidfamilycmpgeneral_3.
  apply Typeoidfamilycmpgeneral_3.
  apply Typeoidfamilyirrgeneral.
  apply Typeoidmapextensionality.
  apply Typeoidfamilycmpgeneral_3.
  apply Typeoidsym.
  apply Typeoidfamilycmpgeneral_3.
  apply Typeoidfamilycmpgeneral_3.
  apply Typeoidfamilyirrgeneral.
  apply Typeoidmapextensionality.
  apply Typeoidfamilycmpgeneral.

  specialize (H2'(F • vp (v1 (F • p5 ⁻¹ (F • p6
x))))).
  assert (F • p3 ⁻¹
         (F • zp
            (F • q5
               (F • q2 ⁻¹ (u1 (F • p2 (F • vp (v1 (F
• p5 ⁻¹ (F • p6 x)))))))))  ≈≈≈
```

```
(F • p3 ⁻¹
         (F • zp
            (F • q5
      (v2 (F • vp (v1 (F • p5 ⁻¹ (F • p6 x)))) )))))

as HW.
  apply Typeoidmapextensionality.
  apply Typeoidmapextensionality.
  apply Typeoidmapextensionality.
  apply Typeoidsym.
  apply H2'.
  Swesetoid.
  Swesetoid.
  Swesetoid.
  assert (F • q6 (F • q4 ⁻¹ (u2 (F • up (u1 (F • p4
(F • wp (w1 x)))))) ≈≈≈
F • q3 (u2 (F • p3 ⁻¹ (F • zp (z1 (F • p6 x)))))) as
HU.
  apply Typeoidsym.
  apply Typeoidfamilycmpgeneral_3.
  apply Typeoidfamilyirrgeneral.
  apply Typeoidmapextensionality.
  Swesetoid.
  Swesetoid.
Defined.



Definition isofunctor_ob:= Idmapp ObV:
Typeoidmap (Catobj (Cat_from_family Rbar))
(Catobj Vcat).


Definition isofunctor_arr_helper:  (Catarr
(Cat_from_family Rbar)) -> ArrV.
  intros  [u [v [hmap hext]]].

  simpl in hmap.
```

```
  exists  〈 〈u, v〉 ,
        (sup (iV u) (fun x=> 〈(pV u) x, (pV v) (hmap
x)〉 )) 〉 .
  unfold is_arrow.
  exists u.
  exists v.
  exists(sup (iV u) (fun x=> 〈(pV u) x, (pV v) (hmap
x)〉 )).
  split.
  apply refV.
  apply functionspacechar.
  unfold memV.
  exists (existT _ hmap hext).
  apply refV.
Defined.


Definition isofunctor_arr:
Typeoidmap  (Catarr (Cat_from_family Rbar)) ArrV.
apply (Build_Typeoidmap (Catarr (Cat_from_family
Rbar)) ArrV isofunctor_arr_helper).
  intros  [u [v [hmap hext]]]
  [u' [v' [hmap' hext']]] H.
  destruct H as  [phi  [psi eq]].
  simpl in eq.
  destruct u as  [a f].
  destruct v as  [b g].
  destruct u' as  [a' f'].
  destruct v' as  [b' g'].
  simpl in eq.
  simpl.
  assert ( 〈 〈 sup a f, sup b g 〉 , sup a (λ x : a,  〈
f x, g (hmap x) 〉 )〉   ≐
 〈 〈 sup a' f', sup b' g' 〉 , sup a' (λ x : a',
 〈 f' x, g' (hmap' x) 〉 )〉 ) as L.
  apply pairing_extensional.
  apply pairing_extensional
```

```
  apply pairing_extensional.
  apply phi.
  apply psi.
  apply Extensionality.
  intro z.
  split.
  intro Q.
  destruct Q as [x xeq].
  simpl in x.
  simpl in xeq.
  exists (push phi x).
  simpl.
  assert ( 〈 f x, g (hmap x) 〉  ≐ 〈 f' (push phi x),
g' (hmap' (push phi x)) 〉 ) as H.
  apply pairing_extensional.
  apply pushprop.
  assert (g (hmap x) ≐ g' (push psi (hmap x))) as H2.
  apply pushprop.
  apply (traV _ _ _ H2 (eq x)).
  apply (traV _ _ _ xeq H).
  intro Q2.
  destruct Q2 as [x xeq].
  simpl in x.
  simpl in xeq.
  exists (pull phi x).
  simpl.
  assert (  〈 f' x, g' (hmap' x) 〉  ≐
 〈 f (pull phi x), g (hmap (pull phi x)) 〉  ) as H2.
  apply pairing_extensional.
  apply symV.
  apply pullprop.
  simpl in hmap.
  simpl in hmap'.
  simpl in hext.
  simpl in hext'.
  assert (forall t: b, g t ≐ g' (push psi t)) as L1.
  intro t.
  apply pushprop.
```

```
  apply pushprop.
  assert (forall t: b', g (pull psi t) ≐ g' t) as L2.
  intro t.
  apply pullprop.
  specialize (L1 (hmap (pull phi x))).
  specialize (eq (pull phi x)).
  specialize (L2 (hmap' x)).
  assert ( g' (hmap' x)  ≐ g' (hmap' (push phi (pull
phi x)))) as L3.
  apply hext'.
  assert (f' x  ≐ f (pull phi x)) as L4.
  apply symV.
  apply pullprop.
  assert (f (pull phi x)  ≐ f' (push phi (pull phi
x))) as L5.
  apply pushprop.
  apply (traV _ _ _ L4 L5).
  apply (traV _ _ _ L3 (traV _ _ _ (symV _ _ eq)
(symV _ _ L1))).
  apply (traV _ _ _ xeq H2).
  apply L.
Defined.

Theorem isofunctor_arr_lemma (u v: VTypeoid)(h:
Typeoidmap (Rbar u) (Rbar v)):
projT1 (isofunctor_arr (existT _ u (existT _ v h)))
≐
 〈 〈u, v〉 ,
       (sup (iV u) (fun x=> 〈(pV u) x, (pV v) (h
x)〉 )) 〉 .
Proof.
  destruct h.
  simpl.
  assert (
 〈 〈 u, v 〉 , sup u (λ x : u,  〈 u x, v
(Typeoidmapfunction0 x) 〉 ) 〉   ≐
 〈 〈 u, v 〉 , sup u (λ x : u,  〈 u x, v
(Typeoidmapfunction0 x) 〉 ) 〉 ) as G
```

```
(typeoidmapfunctionb x) ) ) ) as G.
  apply refV.
apply G.
Defined.

Definition Injective (A B:Typeoid)
(f: Typeoidmap A B):=
(∀ x y: A,  f x  ≈≈≈ f y -> x  ≈≈≈ y).


Definition Surjective (A B:Typeoid)
(f: Typeoidmap A B):=
(∀ u: B, ∃ x:A,  f x ≈≈≈ u).


Theorem isofunctor_arr_is_injective:
Injective (Catarr (Cat_from_family Rbar)) ArrV
isofunctor_arr.
Proof.
  intros  [u [v [hmap hext]]] [u' [v' [hmap'
hext']]].
  destruct u as  [a f].
  destruct v as  [b g].
  destruct u' as  [a' f'].
  destruct v' as  [b' g'].
  intro H.
  simpl.
  assert ( ⟨ ⟨ sup a f, sup b g ⟩ , sup a (λ x : a,  ⟨
f x, g (hmap x) ⟩ )⟩  ≐
 ⟨ ⟨ sup a' f', sup b' g' ⟩ , sup a' (λ x : a',
 ⟨ f' x, g' (hmap' x) ⟩ )⟩ ) as L.
  apply H.
  clear H.
  assert ( ⟨ sup a f, sup b g ⟩  ≐  ⟨ sup a' f', sup
b' g' ⟩ ) as L1.
  apply (pairing_injective _ _ _ _ L).
  assert (sup a f ≐ sup a' f') as L11.
  apply (pairing_injective _ _ _ _ L1).
```

```
  apply  (pairing_injective _ _ _ _ L1).
  assert (sup b g ≐  sup b' g') as L12.
  apply (pairing_injective _ _ _ _ L1).
  simpl in hmap.
  simpl in hmap'.
  simpl in hext.
  simpl in hext'.

  exists L11.
  exists L12.
  assert (sup a (λ x : a,  〈 f x, g (hmap x) 〉 ) ≐
        sup a' (λ x : a',  〈 f' x, g' (hmap' x) 〉 ))
as L2.
  apply (pairing_injective _ _ _ _ L).
  intro x.
  assert (  〈 f x, g (hmap x) 〉  ≐
 〈 f'  (push L2 x), g' (hmap'  (push L2 x)) 〉 ) as L5.
  apply (pushprop L2).
  assert ( f x ≐ f' (push L2 x)) as H1.
  apply (pairing_injective _ _ _ _ L5).
  assert (g (hmap x) ≐ g' (hmap' (push L2 x))) as H2.
  apply (pairing_injective _ _ _ _ L5).
  clear L.
  clear L1.

  assert (g (hmap x) ≐ g' (push L12 (hmap x))) as L3.
  apply pushprop.
  assert (g' (hmap' (push L2 x)) ≐ g' (hmap' (push
L11 x)) ) as L4.
  apply hext'.
  assert (f x ≐  f' (push L11 x)) as L6.
  apply pushprop.
  apply (traV _ _ _ (symV _ _ H1) L6).
  apply (traV _ _ _ (symV _ _ L3)
       (traV _ _ _ H2 L4)).
Qed.
```

```
Definition arFN (x y: V): (x ⇛ y) →
Typeoidmap  (Rbar_ob x) (Rbar_ob y).
  intros [f fext].
  apply (Build_Typeoidmap (Rbar_ob x) (Rbar_ob y) f).
simpl. assumption.
Defined.


Theorem function_extract (x y z:V)
(p:is_function x y z):  x ⇛ y.
Proof.
  assert  (z ∈ x ⇛ y) as H.
  destruct x.
  destruct y.
  destruct z.
  apply functionspacechar.
  assumption.
  apply (projT1 H).
Defined.


Definition make_an_arr (a b phi: V)(p: is_function a
b phi) : ∃a : V, ∃b : V, Typeoidmap (Rbar_ob a)
(Rbar_ob b).
  exists a.
  exists b.
  apply arFN.
  apply  (function_extract a b phi p).
Defined.

Theorem isofunctor_arr_is_surjective:
Surjective (Catarr (Cat_from_family Rbar)) ArrV
isofunctor_arr.
Proof.
  intros  [u [v [w [phi [q1 q2]]]]].
  simpl
```

```
  simpl.
  exists (make_an_arr v w phi q2).
  assert ( projT1 (isofunctor_arr_helper (make_an_arr
v w phi q2)) ≐ 〈 〈 v, w 〉 , phi 〉 ) as H.
  unfold make_an_arr.
  unfold arFN.

  unfold function_extract.

  unfold isofunctor_arr_helper.
  simpl.
  destruct u as [a f].
  destruct v as [a0 f0].
  destruct w as [a1 f1].
  destruct phi as [a2 f2].

  simpl.

  unfold functionspacechar.
  unfold pairing_extensional.
  unfold pairing_injective.
  simpl.
  destruct q2 as  [i1 i0].
  destruct i0  as  [i0 i2].

  assert (
 〈 〈 sup a0 f0, sup a1 f1 〉 ,
        sup a0 (λ x : a0,  〈 f0 x, f1 (projT1 (i0 x))
〉 ) 〉
≐  〈 〈 sup a0 f0, sup a1 f1 〉 ,  sup a2 f2〉 ) as Q.
  apply pairing_extensional.
  apply refV.

  apply Extensionality.
  intro z.
  split.
  intro P.
  unfold memV in P.
```

```
  unfold memV in P.
  destruct P as  [idx P].
  simpl in idx.
  simpl in P.
  unfold is_total in i0.
  assert ( 〈 f0 idx, f1 (projT1 (i0 idx)) 〉   ∈ sup a2
f2) as P1.
  apply (projT2 (i0 idx)).
  apply (ext1 _ _ _ P P1).
  intro P.
  unfold is_rel in i1.
  assert (  z ∈ sup a0 f0 × sup a1 f1) as P2.
  apply (i1 z).
  apply P.
  assert (∃ x:a0, ∃ y:a1, z ≐  〈f0 x, f1 y〉 ) as P3.
  assert ((z ∈ sup a0 f0 × sup a1 f1
    -> (∃α : sup a0 f0, ∃β : sup a1 f1, z ≐  〈 (sup a0
f0) α, (sup a1 f1) β 〉 ))) as P4.
  apply productchar.
  apply P4.
  apply P2.
  destruct P3 as [x [y P5]].
  exists x.
  simpl.
  assert ( 〈 f0 x, f1 (projT1 (i0 x)) 〉   ∈ sup a2 f2)
as P1.
  apply (projT2 (i0 x)).
  assert ( 〈 f0 x, f1 y 〉   ∈ sup a2 f2) as P3.
  apply (ext1 _ _ _ (symV _ _ P5) P).
  assert (  〈 f0 x, f1 (projT1 (i0 x)) 〉  ≐
 〈 f0 x, f1 y 〉 ) as P6.
  apply pairing_extensional.
  apply refV.
  unfold is_functional in i2.
  apply (i2 (f0 x)).
  split.
  assumption.
```

```
  assumption.
  apply (traV _ _ _ P5 (symV _ _ P6)).
  apply Q.
  apply (traV _ _ _ H (symV _ _ q1)).
Defined.



Definition isofunctor_cms_helper:
(Catcms (Cat_from_family Rbar)) -> CmsV.
  intros  [arr1 [arr2 P]].
  exists  ⟨projT1 (isofunctor_arr arr1),
  projT1  (isofunctor_arr arr2)⟩ .
  exists (isofunctor_arr arr1).
  exists (isofunctor_arr arr2).
  split.
  apply refV.
  destruct arr1 as  [d1 [c1 f1]].
  destruct arr2 as  [d2 [c2 f2]].
  simpl in P.
  destruct f1.
  destruct f2.
  simpl.
  apply P.
Defined.

Definition isofunctor_cms:
Typeoidmap  (Catcms (Cat_from_family Rbar)) CmsV.
apply (Build_Typeoidmap  (Catcms (Cat_from_family
Rbar)) CmsV isofunctor_cms_helper).
  intros [arr1 [arr2 P]] [arr1' [arr2' P']].
  intro Q.
  assert (arr1 ≈≈≈ arr1' ∧ arr2  ≈≈≈ arr2') as Q'.
  apply Q.
  clear Q.

  assert ( ⟨ projT1 (isofunctor_arr_helper arr1)
```

```
  assert ( 〈 projT1 (isofunctor_arr_helper arr1),
      projT1 (isofunctor_arr_helper arr2) 〉  ≐
 〈 projT1 (isofunctor_arr_helper arr1'),
      projT1 (isofunctor_arr_helper arr2') 〉 ) as
G1.
  apply pairing_extensional.
  assert ( isofunctor_arr_helper arr1 ≈≈≈,{ ArrV }
isofunctor_arr_helper arr1') as G11.
  apply (Typeoidmapextensionality
(arr_from_family Rbar) ArrV isofunctor_arr).
  apply Q'.
  apply G11.
  assert ( isofunctor_arr_helper arr2 ≈≈≈,{ ArrV }
isofunctor_arr_helper arr2') as G12.
  apply (Typeoidmapextensionality
(arr_from_family Rbar) ArrV isofunctor_arr).
  apply Q'.
  apply G12.
  apply G1.
Defined.


Theorem isofunctor_cms_is_injective:
Injective (Catcms (Cat_from_family Rbar)) CmsV
isofunctor_cms.
Proof.
  unfold Injective.
  intros [arr1 [arr2 P]] [arr1' [arr2' P']].
intro Q.
  assert ( 〈 projT1 (isofunctor_arr_helper arr1),
      projT1 (isofunctor_arr_helper arr2) 〉  ≐
 〈 projT1 (isofunctor_arr_helper arr1'),
      projT1 (isofunctor_arr_helper arr2') 〉 ) as
Q'.
  apply Q.
  clear Q.
  assert (arr1 ≈≈≈ arr1' ∧ arr2  ≈≈≈ arr2') as G.
```

```
  split.
  assert (projT1 (isofunctor_arr_helper arr1)
≐  projT1 (isofunctor_arr_helper arr1')) as Q1.
  apply (pairing_injective _ _ _ _ Q').
  apply  isofunctor_arr_is_injective.
  apply Q1.
  assert (projT1 (isofunctor_arr_helper arr2)
≐  projT1 (isofunctor_arr_helper arr2')) as Q2.
  apply (pairing_injective _ _ _ _ Q').
  apply  isofunctor_arr_is_injective.
  apply Q2.
  apply G.
Defined.


(* The functor that gives the isomorphism of
the two categories *)


Theorem isofunctor: Functor (Cat_from_family Rbar)
Vcat.
apply (Build_Functor (Cat_from_family Rbar) Vcat
isofunctor_ob isofunctor_arr isofunctor_cms).
Proof.
(* Prop1 *)
  intro a.
  assert ( ⟨ ⟨ a, a ⟩ , sup a (λ x : a,  ⟨ a x, a x
⟩ )⟩  ≐
 ⟨ ⟨ a, a ⟩ , identity a ⟩ ) as G.
  apply pairing_extensional.
apply refV.
  unfold identity.
  apply Extensionality.
  intro z.
  assert (z ∈ {{ p ∈ a × a | (∃y : a, p ≐  ⟨ a y, a y
⟩ ) }} ↔  z ∈ a × a ∧ (∃y : a, z ≐  ⟨ a y, a y ⟩ )) as
L.
```

```
apply (separationchar _ (expr_extproof_1 a)).
split.
intro P.
apply L.
unfold memV in P.
destruct P as [x P].
simpl in x.
simpl in P.
split.
assert ( 〈 a x, a x 〉  ∈ a × a).
unfold memV.
exists (x,x).
apply refV.
apply (ext1 _ _ _ P H).
exists x.
apply P.
intro P.
unfold memV.
simpl.
apply L.
apply P.
apply G.

(* Prop2 *)
intros [a [b f]].
unfold isofunctor_arr.
simpl.
unfold pairing_injective.
simpl.
destruct f.
unfold is_arrow.
simpl.
apply refV.

(* Prop3 *)
intros [a [b f]].
unfold isofunctor_arr.
simpl
```

```
  simpl.
  unfold pairing_injective.
  simpl.
  destruct f.
  unfold is_arrow.

  simpl.
  apply refV.

  (* Prop4 *)
  intros [[a [b f]] [[a' [b' f']] P]].
  destruct f.
  destruct f'.
  simpl.
  assert ( ⟨ ⟨ a, b ⟩ , sup a (λ x : a,  ⟨ a x, b
(Typeoidmapfunction0 x) ⟩ )⟩
≐  ⟨ ⟨ a, b ⟩ , sup a (λ x : a,  ⟨ a x, b
(Typeoidmapfunction0 x) ⟩ )⟩ ) as G.
  apply refV.
  apply G.

  (* Prop5 *)

  intros [[a [b f]] [[a' [b' f']] P]].
  destruct f.
  destruct f'.
  simpl.
  assert (
 ⟨ ⟨ a', b' ⟩ , sup a' (λ x : a',  ⟨ a' x, b'
(Typeoidmapfunction1 x) ⟩ )⟩
≐
 ⟨ ⟨ a', b' ⟩ , sup a' (λ x : a',  ⟨ a' x, b'
(Typeoidmapfunction1 x) ⟩ )⟩
) as G.
  apply refV.
  apply G.

(* Prop6 *)
```

```
  intros [[a [b f]] [[a' [b' f']] P]].
  destruct f.
  destruct f'.
  simpl in P.
  simpl.
  assert (
〈 〈 a, b' 〉 ,
     sup a
       (λ x : a,
         〈 a x,
          b'
            (Typeoidmapfunction1
               (Rbar_transp_helper b a' P
(Typeoidmapfunction0 x))) 〉 )〉  ≐
 〈 〈 a, b' 〉 ,
        compose_relV a b b'
          (sup a (λ x : a,  〈 a x, b
(Typeoidmapfunction0 x) 〉 ))
          (sup a' (λ x : a',  〈 a' x, b'
(Typeoidmapfunction1 x) 〉 ))〉 ) as G.
  apply pairing_extensional.
  apply refV.
  apply Extensionality.
  intro z.
  split.
  intro H.
  apply compose_relV_char_2.

  unfold is_rel.
  intro s.
  intro H2.
  unfold memV in H2.
  destruct H2 as  [ix H3].
  simpl in ix.
  simpl in H3.
  exists (ix, (Typeoidmapfunction0 ix)).
  simpl.
```

```
  apply H3.

  unfold is_rel.
  intro s.
  intro H2.
  unfold memV in H2.
  destruct H2 as  [ix H3].
  simpl in ix.
  simpl in H3.
  assert (s ∈ a' × b') as H4.
  exists (ix, (Typeoidmapfunction1 ix)).
  simpl.
  apply H3.
  apply (prod_extensional_half a' b' b b').
  apply (symV _ _ P).
  apply (refV b').
  apply H4.

  unfold memV in H.
  destruct H as  [ix H2].
  simpl in ix.
  simpl in H2.
  exists (a ix).
  exists (b'
           (Typeoidmapfunction1
              (Rbar_transp_helper b a' P
(Typeoidmapfunction0 ix)))).
  split.
  apply H2.
  split.
  exists ix.
  apply refV.
  split.
  exists  (Typeoidmapfunction1
              (Rbar_transp_helper b a' P
(Typeoidmapfunction0 ix))).
  apply refV.
```

```
  exists (b (Typeoidmapfunction0 ix)).
  split.
  exists (Typeoidmapfunction0 ix).
  apply refV.

  split.
  exists ix.
  apply refV.
  unfold memV.
  destruct a.
  destruct b.
  destruct a'.
  destruct b'.
  exists (push P (Typeoidmapfunction0 ix)).
  simpl.
  assert (  〈f0 (Typeoidmapfunction0 ix),
      f2 (Typeoidmapfunction1 (push P
(Typeoidmapfunction0 ix))) 〉
≐  〈 f1 (push P (Typeoidmapfunction0 ix)),
       f2 (Typeoidmapfunction1 (push P
(Typeoidmapfunction0 ix))) 〉 )
as G.
  apply pairing_extensional.
  apply pushprop.
  apply refV.
  apply G.

  intro Q.
  assert ((∃ x:V, ∃ z':V,
z ≐  〈 x, z' 〉  ∧ x ∈ a ∧ z' ∈ b'   ∧
  ∃y:V, y ∈ b ∧  〈 x, y 〉  ∈  (sup a (λ x : a,  〈 a x,
b (Typeoidmapfunction0 x) 〉 ))
∧  〈 y, z' 〉  ∈  (sup a' (λ x : a',  〈 a' x, b'
(Typeoidmapfunction1 x) 〉 )))) as Q'.
  apply compose_relV_char_2.

  unfold is_rel.
```

```
  intro t.
  intro H.
  unfold memV in H.
  destruct H as  [ix H1].
  simpl in ix.
  simpl in H1.
  exists (ix, (Typeoidmapfunction0 ix)).
  simpl.
  apply H1.

  unfold is_rel.
  intro s.
  intro H2.
  unfold memV in H2.
  destruct H2 as  [ix H3].
  simpl in ix.
  simpl in H3.
  assert  (s ∈ a' × b') as G.
  exists (ix, (Typeoidmapfunction1 ix)).
  simpl.
  apply H3.
  apply (prod_extensional_half a' b' b b').
  apply (symV _ _ P).
  apply (refV b').
  apply G.
  apply Q.

  destruct Q' as [x [z' [Q2 [Q3 [Q4 [u [Q6 [Q7
Q8]]]]]]]].
  unfold memV in Q3.
  destruct Q3 as [ix Q3].
  unfold memV in Q4.
  destruct Q4 as [ix'' Q4].
  unfold memV in Q6.
  destruct Q6 as [ix' Q6].
  unfold memV in Q7.
  destruct Q7 as [ix4 Q7].
```

```
  simpl in ix4.
  unfold memV in Q8.
  destruct Q8 as [ix5 Q8].
  simpl in ix5.
  assert (  〈 x, u 〉  ≐  〈 a ix4, b
(Typeoidmapfunction0 ix4) 〉 ) as Q7'.
  apply Q7.
  clear Q7.
  assert (  〈 u, z' 〉
       ≐  〈 a' ix5, b' (Typeoidmapfunction1 ix5) 〉 )
as Q8'.
  apply Q8.
  clear Q8.
  unfold memV.
  assert ( ∃idx
   : a,
   z
   ≐   〈 a idx, b'
           (Typeoidmapfunction1
              (Rbar_transp_helper b a' P
(Typeoidmapfunction0 idx))) 〉 ) as G.

  exists ix4.
  assert (x ≐ a ix4) as H1.
  apply (pairing_injective _ _ _ _ Q7').
  assert (z' ≐ b' (Typeoidmapfunction1 ix5)) as H2.
  apply (pairing_injective _ _ _ _ Q8').
  assert (b' (Typeoidmapfunction1 ix5)  ≐ b'
       (Typeoidmapfunction1
          (Rbar_transp_helper b a' P
(Typeoidmapfunction0 ix4)))) as H3.
  apply Typeoidmapextensionality1.
  assert (u  ≐  b (Typeoidmapfunction0 ix4)) as Q9.
  apply (pairing_injective _ _ _ _ Q7').
  assert ( u ≐  a' ix5) as QA.
  apply (pairing_injective _ _ _ _ Q8').
  assert (a' ix5  ≐   b (Typeoidmapfunction0 ix4)) as
```

```
QB.
  apply (traV _ _ _ (symV _ _ QA) Q9).
  assert (a' (Rbar_transp_helper b a' P
(Typeoidmapfunction0 ix4))  ≐ b (Typeoidmapfunction0
ix4)) as QC.
  apply symV.
  destruct a'.
  destruct b.
  apply pushprop.
  apply (traV _ _ _ QB (symV _ _ QC)).
  assert ( 〈 x, z' 〉   ≐  〈a ix4,
b'
           (Typeoidmapfunction1
              (Rbar_transp_helper b a' P
(Typeoidmapfunction0 ix4))) 〉   ) as QD.
  apply pairing_extensional.
  apply H1.
  apply (traV _ _ _ H2 H3).
  apply (traV _ _ _ Q2 QD).
  apply G.
  apply G.
Defined.


(* Finally we establish that  isofunctor  is
surjective also for the Cms component. This
concludes the proof that  isofunctor  is
an isomorphism of categories as was to be proved *)

Definition isofunctor_arr_inverse (u:ArrV):=
(projT1 (isofunctor_arr_is_surjective u)):
(Catarr (Cat_from_family Rbar)).

Theorem isofunctor_arr_inverse_prop (u:ArrV):
isofunctor_arr (isofunctor_arr_inverse u)  ≈≈≈,{ ArrV
} u.
Proof.
```

```
  apply (projT2 (isofunctor_arr_is_surjective u)).
Defined.


Theorem isofunctor_cms_is_surjective:
Surjective (Catcms (Cat_from_family Rbar)) CmsV
isofunctor_cms.
Proof.
  unfold Surjective.
  intros [w [arr1 [arr2 P]]].
  assert (isofunctor_arr (isofunctor_arr_inverse
arr1)  ≈≈≈,{ ArrV } arr1) as L1.
  apply isofunctor_arr_inverse_prop.
  assert (isofunctor_arr (isofunctor_arr_inverse
arr2)  ≈≈≈,{ ArrV } arr2) as L2.
  apply isofunctor_arr_inverse_prop.
  assert (codV (isofunctor_arr
(isofunctor_arr_inverse arr1)) ≈≈≈,{ ObV } codV arr1)
as L3.
  apply Typeoidmapextensionality.
  apply L1.
  assert (domV (isofunctor_arr
(isofunctor_arr_inverse arr2)) ≈≈≈,{ ObV } domV arr2)
as L4.
  apply Typeoidmapextensionality.
  apply L2.
  assert (codV (isofunctor_arr
(isofunctor_arr_inverse arr1))  ≈≈≈,{ ObV }  (Catcod
(Cat_from_family Rbar)) (isofunctor_arr_inverse
arr1)) as L5.
  apply Typeoidsym.
  simpl.
  apply (Fun_cod (Cat_from_family Rbar)
Vcat  isofunctor).
  assert (domV (isofunctor_arr
(isofunctor_arr_inverse arr2))  ≈≈≈,{ ObV }  (Catdom
(Cat_from_family Rbar)) (isofunctor_arr_inverse
```

```
arr2)) as L6.
  apply Typeoidsym.
  simpl.
  apply (Fun_dom (Cat_from_family Rbar)
Vcat  isofunctor).

  assert ((Catcod (Cat_from_family Rbar))
(isofunctor_arr_inverse arr1) ≈≈≈,{ ObV }
(Catdom (Cat_from_family Rbar))
(isofunctor_arr_inverse arr2)
) as L7.
  destruct P as [P1 P2].
  apply (traV _ _ _ (symV _ _ L5) (traV _ _ _ L3
   (traV _ _ _ P2 (traV _ _ _ (symV _ _ L4) L6)))).
  exists (cms_from_family_builder Rbar
(isofunctor_arr_inverse arr1)
(isofunctor_arr_inverse arr2)
L7).
  assert ( ⟨projT1 (isofunctor_arr_helper
(isofunctor_arr_inverse arr1)),
      projT1 (isofunctor_arr_helper
(isofunctor_arr_inverse arr2)) ⟩  ≐ w) as G.
  assert ( ⟨ projT1 arr1, projT1 arr2 ⟩  ≐
   ⟨ projT1 (isofunctor_arr_helper
(isofunctor_arr_inverse arr1)),
   projT1 (isofunctor_arr_helper
(isofunctor_arr_inverse arr2)) ⟩ ) as L8.
  apply pairing_extensional.
  apply symV.
  apply L1.
  apply symV.
  apply L2.
  apply symV.
  destruct P as [P1 P2].
  apply (traV _ _ _ P1 L8).
  simpl.
  apply G.
Defined.
```