```
(*

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


   Written in
   Coq 8.3pl2/Coq 8.3pl3 with UTF8-encoding.


*)

Require Import PropasTypesUtf8Notation
PropasTypesBasics_mod.

Require Import SwedishSetoids_mod.

Delimit Scope czf_scope with czf.
Open Scope czf_scope.


Section CZF.


(* 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) (α: y): x ≐ y α → x ∈ y.
Proof.
  intro; exists α; 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 binuncharL: ∀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 binuncharR: ∀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 binuncharL 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
       | 0   => ∅
       | 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 (i, pf);
```

```
  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 binuncharL in mem2; destruct mem2 as [[i
e] | [[i1 i2] e2]].
  apply binuncharR; right. unfold unionV; apply
(sigrejig A).
  exists i. cut (y ∈ TC (f i)); trivial. apply
TClemma. apply ext2 with x; assumption.
  apply binuncharR; 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 α [[[ai11 ai12] ai2] eq]. exists
(ai11, ai12). assumption.
  split. intro α. exists α. unfold identity. unfold
sepV.
  apply (sigrejig (x ∧ x)). exists (α, α). apply
(sigrejig x). exists α.
  exists (refV  ⟨x α, x α⟩ ). 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 a 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 TTAC in totz. destruct totz as [f 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.


End CZF.
```