Contents

Preamble

```
Require Import Basics.
Require Import PolyWellFounded.

Set Implicit Arguments.

Notation " g `o` f " := (compose g f)
  (at level 40, left associativity).

Reserved Notation "a <*> b"
  (at level 41, left associativity).

Definition flip_apply {A B : Type}  := flip (@apply A B).

Definition prod_curry {A B C:Type} (f:A -> B -> C)
  (p:prod A B) : C := f (fst p) (snd p).
```

1 Functor

```
Section Functor.
  Variable f : Type -> Type.
  Variable fmap : forall {a b : Type}, (a -> b) -> f a -> f b.

  Class Functor : Prop := {
    fmap_id : forall {a : Type} (x : f a), fmap id x = x;
    fmap_comp: forall (a b c : Type) (x : f a) (g : b -> c) (h : a -> b),
      fmap (g `o` h) x = fmap g (fmap h x)
  }.

End Functor.

Arguments Functor {f} fmap.
```

2 Applicative

```
Section Applicative.
  Variable f : Type -> Type.
  Variable pure : forall {a : Type}, a -> f a.
  Variable ap : forall {a b : Type}, f (a -> b) -> f a -> f b.

  Notation "g <*> x" := (ap g x).

  Class Applicative : Prop := {
    applicative_identity : forall (a : Type) (x : f a), pure id <*> x = x;
    applicative_homomorphism : forall (a b : Type) (g : (a -> b)) (x : a),
      pure g <*> pure x = pure (g x);
    applicative_interchange : forall (a b : Type) (u : f (a -> b)) (y : a),
      u <*> pure y = pure (flip_apply y) <*> u;
    applicative_composition : forall (a b c : Type) (u : f (b -> c)) (v : f (a -> b)) (w : f a),
      u <*> (v <*> w) = pure compose <*> u <*> v <*> w
  }.
```

2.1 Applicative is a Functor

```
  Definition applicative_map {a b : Type} (g : a -> b) (x : f a) : f b :=
    pure g <*> x.

  Instance applicative_functor (instance : Applicative) : Functor (@applicative_map).
  Proof.
    split.

(* functor identity *)

    apply applicative_identity.

(* functor composition *)

    intros. unfold applicative_map.
    rewrite applicative_composition.
    rewrite applicative_homomorphism.
    rewrite applicative_homomorphism.
    reflexivity. Qed.
End Applicative.

Arguments Applicative {f} pure ap.
```

3 Free applicative

http://ro-che.info/articles/2013-03-31-flavours-of-free-applicative-functors.html

3.1 Paolo Capriotti's free applicative

```
data Free f a where
  Pure :: a -> Free f a
  Ap :: f (a -> b) -> Free f a -> Free f b

instance Functor f => Functor (Free f) where
  fmap f (Pure x) = Pure $ f x
  fmap f (Ap ax ty) = Ap (fmap (f.) ax) ty
```

```
instance Functor f => Applicative (Free f) where
  pure = Pure
  Pure f <*> tx = fmap f tx
  Ap ax ty <*> tz = Ap (fmap uncurry ax) ((,) <$> ty <*> tz)



Section FreeA.

Variable f : Type -> Type.

Inductive FreeA (a : Type) : Type :=
  | Pure : a -> FreeA a
  | Fm   : forall (b : Type), f (b -> a) -> FreeA b -> FreeA a.

Fixpoint freeA_measure {a : Type} (x : FreeA a) : nat :=
  match x with
  | Pure _      => 0
  | Fm _ _ x  => S (freeA_measure x)
  end.
```

We use polymorphic well founded relation to define and proof things below. Definition of polymorphic well foundness is in Appendix A

```
Definition freeA_measureR X x Y y := @ltofp Type FreeA (@freeA_measure) X x Y y.

Definition freeA_measure2R X x Y y :=
  @ltofp (Type * Type) (fun T2 => FreeA (fst T2 -> snd T2))
    (fun T2 x => freeA_measure x) X x Y y.

Lemma freeA_measure2R_pwf :
  @poly_well_founded (Type*Type) (fun T2 => FreeA (fst T2 -> snd T2)) freeA_measure2R.
Proof. apply ltofp_pwf. Defined.

Definition freeA_measure3R X x Y y :=
 @ltofp (Type * Type) (fun T2 => prod (FreeA (fst T2 -> snd T2)) (FreeA (fst T2)))
    (fun T2 x => freeA_measure (fst x)) X x Y y.

Lemma freeA_measure3R_pwf :
  @poly_well_founded
    (Type*Type)
    (fun T2 => prod (FreeA (fst T2 -> snd T2)) (FreeA (fst T2)))
    freeA_measure3R.
Proof. apply ltofp_pwf. Defined.

Variable fmap : forall {a b : Type}, (a -> b) -> f a -> f b.

Hypothesis functor : Functor fmap.

Fixpoint freeA_fmap {a b : Type} (g : a -> b) (x : FreeA a) : FreeA b :=
  match x with
  | Pure x'      => Pure (g x')
  | Fm _ h x'    => Fm (fmap (compose g) h) x'
  end.

Instance freeA_functor : Functor (@freeA_fmap).
Proof.
  destruct functor as [fmap_id fmap_comp].
  split.
```

```
(* identity *)

  intros.
  induction x. reflexivity.
  simpl. replace f0 with (fmap id f0) by (apply fmap_id).
  rewrite <- fmap_comp. reflexivity.

(* composition *)

  intros.
  generalize dependent b. generalize dependent c.
  induction x. reflexivity.
  simpl. intros.
  rewrite <-fmap_comp. reflexivity. Qed.

Lemma freeA_fmap_measure : forall {a b : Type} (g : a -> b) (x : FreeA a),
  freeA_measure (freeA_fmap g x) = freeA_measure x.
Proof.
  intros. destruct x. reflexivity. reflexivity. Qed.

Definition freeA_pure {a : Type} (x : a) : FreeA a := Pure x.

Program Definition freeA_ap (a b : Type) (g : FreeA (a -> b)) (y : FreeA a) : FreeA b :=
  PFix
    (R:=freeA_measure3R)
    (A:=fun T2 => prod (FreeA (fst T2 -> snd T2)) (FreeA (fst T2)))
    freeA_measure3R_pwf
    (fun T2 g => FreeA (snd T2))
    (fun X2 g freeA_ap => _)
    (a,b) (g,y).

Next Obligation.
  simpl in *.
  destruct f0 as [g' | c h x'].
  exact (freeA_fmap g' f1).
  refine (Fm  (fmap prod_curry h)
              (freeA_ap (T, prod c T) ((freeA_fmap (@pair c T) x'), f1) _)).
  unfold freeA_measure3R, ltofp. simpl. rewrite freeA_fmap_measure. apply le_n. Defined.

Notation "g <*> x" := (freeA_ap g x).

Lemma freeA_ap_ap : forall (a b c : Type) (h : f (c -> a -> b)) (x : FreeA c) (y : FreeA a),
  Fm h x <*> y = Fm (fmap prod_curry h) (freeA_fmap pair x <*> y).
Proof.
  intros. unfold freeA_ap.
  rewrite PFix_eq.
  unfold freeA_ap_obligation_1.
  f_equal.
  intros.
  unfold freeA_ap_obligation_1.
  destruct X, x0. destruct f1.
  reflexivity.
  rewrite H. reflexivity. Qed.

Lemma freeA_ap_pure : forall (a b : Type) (g : a -> b) (x : FreeA a),
  (Pure g) <*> x = freeA_fmap g x.
Proof. reflexivity. Qed.

Lemma freeA_fmap_fmap : forall a b c (g : a -> b) (h : f (c -> a)) (x : FreeA c),
  freeA_fmap g (Fm h x) = Fm (fmap (compose g) h) x.
Proof. reflexivity. Qed.
```

```
Lemma freeA_lemma1 :
  forall {a b c : Type},
  forall (u : b -> c) (v : FreeA (a -> b)) (x : FreeA a),
  freeA_fmap u (v <*> x) = freeA_fmap (compose u) v <*> x.
Proof.
  intros.
  destruct freeA_functor as [freeA_fmap_id freeA_fmap_comp].
  destruct functor as [f_fmap_id f_fmap_comp].
  destruct v as [v | c' g y].

(* pure *)

  simpl.
  rewrite freeA_ap_pure. rewrite freeA_ap_pure.
  rewrite freeA_fmap_comp. reflexivity.

(* fmap *)

  rewrite freeA_ap_ap.
  simpl. rewrite <- f_fmap_comp.
  rewrite freeA_ap_ap.
  f_equal.
  rewrite <- f_fmap_comp.
  f_equal. Qed.

Program Definition freeA_measure2_ind
  (P : forall a b : Type, FreeA (a -> b) -> Prop)
  (a b : Type)
  (x : FreeA (a -> b))
  (Proof: forall (a1 b1 : Type) (x : FreeA (a1 -> b1)),
    (forall (a2 b2 : Type) (y : FreeA (a2 -> b2)),
      freeA_measure2R (a2,b2) y (a1,b1) x -> P a2 b2 y) -> P a1 b1 x)
  : P a b x := @polymorphic_well_founded_ind (Type * Type)
    (fun T2 => FreeA (fst T2 -> snd T2))
    freeA_measure2R
    (fun T2 u =>
      P (fst T2) (snd T2) u) _ (a, b) x _.

Next Obligation.
  apply freeA_measure2R_pwf. Defined.

Definition t (w x y : Type) (p : ((w * (x -> y)) * x)) : (w * y) :=
  pair (fst (fst p)) ((snd (fst p)) (snd p)).

(*
  match p with
  | ((w, v), x) => (w, v x)
  end.
*)


Definition freeA_free_theorem_type : Prop :=
  forall {x y : Type} (h : x -> y),
  forall (a : Type) (g : f (y -> a)) (u : FreeA x),
    Fm (fmap (flip compose h) g) u = Fm g (freeA_fmap h u).

Hypothesis freeA_free_theorem :  freeA_free_theorem_type.

Instance freeA_applicative : Applicative (@freeA_pure) (@freeA_ap).
Proof.
  split.
```

```
(* identity *)

  intros. unfold freeA_pure. rewrite freeA_ap_pure.
  apply fmap_id. exact (freeA_functor).

(* homomorphism *)

  intros.
  unfold freeA_pure. reflexivity.

(* interchange *)

  intros.


(*
  Check polymorphic_well_founded_ind.
  Check polymorphic_well_founded_ind (Type * Type)
    (fun T2 => FreeA f (fst T2 -> snd T2)) _.
  Check (polymorphic_well_founded_ind (Type * Type)
    (fun T2 => FreeA f (fst T2 -> snd T2)) freeA_measure2R
    (fun T2 u =>
       forall y : fst T2, u <*> freeA_pure y = freeA_pure (flip_apply y) <*> u)
  ) _ (a, b) u _ _.

  refine ((polymorphic_well_founded_ind (Type * Type)
    (fun T2 => FreeA f (fst T2 -> snd T2)) freeA_measure2R
    (fun T2 u =>
       forall y : fst T2, u <*> freeA_pure y = freeA_pure (flip_apply y) <*> u)
  ) _ (a, b) u _ _).
*)


  apply (freeA_measure2_ind
    (fun a b u => forall y :a, u <*> freeA_pure y = freeA_pure (flip_apply y) <*> u)
    u).

  intros.
  destruct x. reflexivity.

  simpl. rewrite freeA_ap_ap. simpl in f0.
  rewrite H.

  unfold freeA_pure.
  rewrite freeA_ap_pure. rewrite freeA_ap_pure. simpl.
  rewrite <- fmap_comp.
  rewrite <- freeA_free_theorem.
  rewrite <- fmap_comp.
  f_equal.

  auto. apply freeA_functor.
  unfold freeA_measure2R, ltofp. simpl. rewrite freeA_fmap_measure. apply le_n.


(* composition *)

  intros.

  apply (freeA_measure2_ind
    (fun b c u =>  forall (v : FreeA (a -> b)),
                   u <*> (v <*> w) = freeA_pure compose <*> u <*> v <*> w)
```

```
    u).

  intros.

  destruct x.


(* pure *)

  unfold freeA_pure.
  rewrite freeA_ap_pure.

  rewrite freeA_ap_pure.
  simpl.
  apply freeA_lemma1.


(* fmap *)

  rewrite freeA_ap_ap.
  rewrite H.

  unfold freeA_pure.
  rewrite freeA_ap_pure.

  rewrite <- fmap_comp.

  rewrite freeA_ap_pure.
  simpl.
  rewrite freeA_ap_ap.
  rewrite <- fmap_comp.
  rewrite freeA_ap_ap.
  rewrite <- fmap_comp.

  Eval compute in (@prod_curry (b0 * (a -> a1)) a b1
      `o` (@prod_curry b0 (a -> a1) (a -> b1)
         `o` @compose b0 (a1 -> b1) ((a -> a1) -> a -> b1) (@compose a a1 b1))).

  Eval compute in (compose
    (flip (@compose (b0 * (a -> a1) * a) (b0 * a1) b1) (@t b0 a a1))
    (@prod_curry b0 a1 b1)).

  replace
    (@prod_curry (b0 * (a -> a1)) a b1
      `o` (@prod_curry b0 (a -> a1) (a -> b1)
         `o` @compose b0 (a1 -> b1) ((a -> a1) -> a -> b1) (@compose a a1 b1)))
  with
    (compose
      (flip (@compose (b0 * (a -> a1) * a) (b0 * a1) b1) (@t b0 a a1))
      (@prod_curry b0 a1 b1)).

  pattern (fmap (flip compose (t (x:=a) (y:=a1)) `o` prod_curry) f0).
  rewrite fmap_comp.
  rewrite freeA_free_theorem.
  f_equal.
  rewrite freeA_lemma1.
  rewrite <- fmap_comp.
  rewrite freeA_lemma1.
  rewrite <- fmap_comp.
  f_equal.

  exact freeA_functor.
```

```
  exact freeA_functor.
  auto.


(* replace obligation *)

  reflexivity.

  auto.
  auto.
  exact freeA_functor.


(* induction obligations *)

  unfold freeA_measure2R, ltofp. simpl. rewrite freeA_fmap_measure.
  apply le_n. Qed.

End FreeA.
```

3.2 Twan van Laarhovens free applicative

```
data Free f a where
  Pure :: a -> Free f a
  Ap :: forall b. f b -> Free f (b -> a)  -> Free f a

instance Functor f => Functor (Free f) where
  fmap f (Pure x) = Pure $ f x
  fmap f (Ap tx ay) = Ap ((f .) <$> tx) ay

instance Functor f => Applicative (Free f) where
  pure = Pure
  Pure f <*> tx = fmap f tx
  Ap tx ay <*> tz = Ap (flip <$> tx <*> tz) ay



Section FreeAL.

Variable f : Type -> Type.

Inductive FreeAL (a : Type) : Type :=
  | PureL  : a -> FreeAL a
  | Ap     : forall (b : Type), f b -> FreeAL (b -> a) -> FreeAL a.

Fixpoint freeAL_measure {a : Type} (x : FreeAL a) : nat :=
  match x with
  | PureL _    => 0
  | Ap _ _ x  => S (freeAL_measure x)
  end.

Definition freeAL_measureR X x Y y := @ltofp Type FreeAL (@freeAL_measure) X x Y y.

Lemma freeAL_measureR_pwf :
  @poly_well_founded Type FreeAL freeAL_measureR.
Proof. apply ltofp_pwf. Defined.

Definition freeAL_measure2R X x Y y :=
  @ltofp (Type * Type) (fun T2 => FreeAL (fst T2 -> snd T2))
    (fun T2 x => freeAL_measure x) X x Y y.
```

```
Lemma freeAL_measure2R_pwf :
  @poly_well_founded (Type*Type) (fun T2 => FreeAL (fst T2 -> snd T2)) freeAL_measure2R.
Proof. apply ltofp_pwf. Defined.

Definition freeAL_measure3R X x Y y :=
 @ltofp (Type * Type) (fun T2 => prod (FreeAL (fst T2 -> snd T2)) (FreeAL (fst T2)))
    (fun T2 x => freeAL_measure (fst x)) X x Y y.

Lemma freeAL_measure3R_pwf :
  @poly_well_founded
    (Type*Type)
    (fun T2 => prod (FreeAL (fst T2 -> snd T2)) (FreeAL (fst T2)))
    freeAL_measure3R.
Proof. apply ltofp_pwf. Defined.

Variable fmap : forall {a b : Type}, (a -> b) -> f a -> f b.

Hypothesis functor : Functor fmap.

Fixpoint freeAL_fmap {a b : Type} (g : a -> b) (x : FreeAL a) : FreeAL b :=
  match x with
  | PureL x'    => PureL (g x')
  | Ap _ x' y'  => Ap x' (freeAL_fmap (compose g) y')
  end.

Lemma freeAL_fmap_comp : forall (a b c : Type) (x : FreeAL a) (g : b -> c) (h : a -> b),
  freeAL_fmap (g `o` h) x = freeAL_fmap g (freeAL_fmap h x).
Proof.
  destruct functor as [fmap_id fmap_comp].
  intros. generalize dependent b. generalize dependent c.
  induction x. reflexivity.
  simpl. intros. rewrite <- IHx. reflexivity. Qed.

Lemma freeAL_fmap_ap : forall (a b c : Type) (g : a -> b) (x : f c) (y : FreeAL (c -> a)),
  freeAL_fmap g (Ap x y) = Ap x (freeAL_fmap (compose g) y).
Proof. reflexivity. Qed.

Instance freeAL_functor : Functor (@freeAL_fmap).
Proof.
  destruct functor as [fmap_id fmap_comp]. split.

(* identity *)

  intros.
  induction x. reflexivity.
  simpl. rewrite <- IHx. rewrite <- freeAL_fmap_comp. reflexivity.

(* composition *)

  exact freeAL_fmap_comp. Qed.

Lemma freeAL_fmap_measure : forall {a b : Type} (g : a -> b) (x : FreeAL a),
  freeAL_measure (freeAL_fmap g x) = freeAL_measure x.
Proof.
  intros. generalize dependent b. induction x. reflexivity.
  simpl. intros. apply f_equal. apply IHx. Qed.

Definition freeAL_pure {a : Type} (x : a) : FreeAL a := PureL x.

Program Definition freeAL_ap (a b : Type) (g : FreeAL (a -> b)) (z : FreeAL a) : FreeAL b :=
  PFix
    (R:=freeAL_measure3R)
```

```
    (A:=fun T2 => prod (FreeAL (fst T2 -> snd T2)) (FreeAL (fst T2)))
    freeAL_measure3R_pwf
    (fun T2 gz => FreeAL (snd T2))
    (fun X2 gz freeAL_ap => _)
    (a,b) (g,z).

Next Obligation.
  simpl in *.
  destruct f0 as [g' | c x' y'].
  exact (freeAL_fmap g' f1).
  refine (Ap x' (freeAL_ap (T, c -> T0) ((freeAL_fmap flip y'), f1) _)).
  unfold freeAL_measure3R, ltofp. simpl. rewrite freeAL_fmap_measure. apply le_n. Defined.

Notation "g <*> x" := (freeAL_ap g x).

Lemma freeAL_ap_pure : forall (a b : Type) (g : a -> b) (x : FreeAL a),
  PureL g <*> x = freeAL_fmap g x.
Proof.
  intros. reflexivity. Qed.

Lemma freeAL_ap_ap : forall (a b c : Type) (x :f b) (y : FreeAL (b -> c -> a)) (z : FreeAL c),
  Ap x y <*> z = Ap x (freeAL_fmap flip y <*> z).
Proof.
  intros.
  unfold freeAL_ap.
  rewrite PFix_eq.
  unfold freeAL_ap_obligation_1.
  f_equal.
  intros.
  unfold freeAL_ap_obligation_1.
  destruct X. destruct x0. destruct f1. reflexivity.
  rewrite H. reflexivity. Qed.

Program Definition freeAL_measure2_ind
  (P : forall a b : Type, FreeAL (a -> b) -> Prop)
  (a b : Type)
  (x : FreeAL (a -> b))
  (Proof: forall (a1 b1 : Type) (x : FreeAL (a1 -> b1)),
    (forall (a2 b2 : Type) (y : FreeAL (a2 -> b2)),
      freeAL_measure2R (a2,b2) y (a1,b1) x -> P a2 b2 y) -> P a1 b1 x)
  : P a b x := @polymorphic_well_founded_ind (Type * Type)
    (fun T2 => FreeAL (fst T2 -> snd T2))
    freeAL_measure2R
    (fun T2 u =>
      P (fst T2) (snd T2) u) _ (a, b) x _.

Next Obligation.
  apply freeAL_measure2R_pwf. Qed.

Lemma freeAL_lemma1 :
  forall {a b c : Type},
  forall (u : b -> c) (v : FreeAL (a -> b)) (x : FreeAL a),
  freeAL_fmap u (v <*> x) = freeAL_fmap (compose u) v <*> x.
Proof.
  intros.
  destruct freeAL_functor as [freeAL_fmap_id freeAL_fmap_comp].
  destruct functor as [fmap_id fmap_comp].

  apply (freeAL_measure2_ind
    (fun a b v => forall c (u : b -> c) (x : FreeAL a),
      freeAL_fmap u (v <*> x) = freeAL_fmap (compose u) v <*> x)
    v).
```

```
  intros.

  destruct x0 as [x0 | c' g y].

(* pure *)

  simpl.
  rewrite freeAL_ap_pure. rewrite freeAL_ap_pure.
  rewrite freeAL_fmap_comp. reflexivity.

(* fmap *)

  rewrite freeAL_ap_ap.
  simpl. rewrite freeAL_ap_ap.
  f_equal. rewrite <- freeAL_fmap_comp.
  rewrite H.
  rewrite <- freeAL_fmap_comp.
  f_equal.

  unfold freeAL_measure2R, ltofp. simpl. rewrite freeAL_fmap_measure.
  apply le_n. Qed.


Instance freeAL_applicative : Applicative (@freeAL_pure) (@freeAL_ap).
Proof.
  destruct functor as [fmap_id fmap_comp].
  destruct freeAL_functor as [freeAL_fmap_id freeAL_fmap_comp].
  split.

(* identity *)

  intros.
  induction x. reflexivity.
  unfold freeAL_pure. rewrite freeAL_ap_pure. apply freeAL_fmap_id.

(* homomorphism *)

  unfold freeAL_pure. reflexivity.

(* interchange *)

  intros.
  unfold freeAL_pure.
  rewrite freeAL_ap_pure.

  apply (freeAL_measure2_ind
    (fun a b u => forall y :a, u <*> PureL y = freeAL_fmap (flip_apply y) u)
    u).
  intros.

  destruct x.
  reflexivity.

  rewrite freeAL_ap_ap.
  simpl.
  rewrite H.
  rewrite <- freeAL_fmap_comp.
  reflexivity.

  unfold freeAL_measure2R, ltofp. simpl. rewrite freeAL_fmap_measure.
  apply le_n.
```

```
(* composition *)

  intros.

  apply (freeAL_measure2_ind
    (fun b c u =>  forall (v : FreeAL (a -> b)),
                   u <*> (v <*> w) = freeAL_pure compose <*> u <*> v <*> w)
    u).
  intros.

  destruct x. clear H.
  unfold freeAL_pure.
  repeat (rewrite freeAL_ap_pure).
  simpl.
  repeat (rewrite freeAL_ap_pure).

  apply freeAL_lemma1.

  simpl.
  unfold freeAL_pure.
  rewrite freeAL_ap_pure.
  rewrite freeAL_fmap_ap.
  repeat (rewrite freeAL_ap_ap).
  f_equal.
  rewrite H.
  unfold freeAL_pure.
  rewrite freeAL_ap_pure.
  repeat (rewrite <- freeAL_fmap_comp).
  rewrite freeAL_lemma1.
  rewrite <- freeAL_fmap_comp.
  f_equal.

  unfold freeAL_measure2R, ltofp. simpl. rewrite freeAL_fmap_measure. apply le_n. Qed.

End FreeAL.
```

3.3 Summary

```
Check freeA_applicative.

freeA_applicative
     : forall (f : Type -> Type)
         (fmap : forall a b : Type, (a -> b) -> f a -> f b),
       Functor fmap ->
       freeA_free_theorem_type f fmap ->
       Applicative (freeA_pure f) (freeA_ap fmap)


Print Assumptions freeA_applicative.

Closed under the global context



Check freeAL_applicative.
```

```
freeAL_applicative
     : forall (f : Type -> Type)
         (fmap : forall a b : Type, (a -> b) -> f a -> f b),
       Functor fmap -> Applicative (freeAL_pure f) (freeAL_ap (f:=f))



Print Assumptions freeAL_applicative.

Closed under the global context
```

A Polymorphic well founded relation

```
Require Import Arith.

Set Implicit Arguments.


Section PWF.
  Variable T: Type.

  Variable A : T -> Type.

  Variable R : forall (X : T), A X -> forall Y : T, A Y -> Prop.

 Inductive PAcc {X : T} (x : A X) : Prop :=
     PAcc_intro : (forall (Y : T) (y : A Y) , R y x -> @PAcc Y y) -> PAcc x.

   Lemma PAcc_inv : forall (X : T) (x : A X), @PAcc X x -> forall (Y : T) (y : A Y), R y x -> @PAcc
     destruct 1; trivial.
   Defined.

  Definition poly_well_founded := forall (X : T) (a : A X), PAcc a.

  Hypothesis PRwf : poly_well_founded.

 Theorem polymorphic_well_founded_induction_type :
   forall P : forall X : T, A X -> Type,
     (forall (X : T) (x : A X),
        (forall (Y : T) (y : A Y), R y x -> P Y y) -> P X x) ->
     forall (X : T) (x : A X), PAcc x -> P X x.
 Proof.
   intros. apply PAcc_rect; auto. Qed.

 Theorem polymorphic_well_founded_induction :
   forall P : forall X : T, A X -> Set,
     (forall (X : T) (x : A X),
        (forall (Y : T) (y : A Y), R y x -> P Y y) -> P X x) ->
     forall (X : T) (x : A X), PAcc x -> P X x.
 Proof.
   intros. apply PAcc_rect; auto. Qed.

 Theorem polymorphic_well_founded_ind :
   forall P : forall X : T, A X -> Prop,
     (forall (X : T) (x : A X),
        (forall (Y : T) (y : A Y), R y x -> P Y y) -> P X x) ->
     forall (X : T) (x : A X), PAcc x -> P X x.
 Proof.
   intros. apply PAcc_rect; auto. Qed.

Section PFixPoint.
  Variable P : forall (X : T), A X -> Type.
  Variable F : forall (X : T) (x : A X), (forall (Y : T) (y : A Y), R y x -> P y) -> P x.

  Fixpoint PFix_F (X : T) (x : A X) (a : @PAcc X x) : P x :=
    F (fun (Y : T) (y : A Y) (H : R y x) =>
       PFix_F (PAcc_inv a H)).

  Scheme PAcc_inv_dep := Induction for PAcc Sort Prop.
```

```
  Lemma PFix_F_eq :
    forall (X : T) (x : A X) (r : PAcc x),
      F (fun (Y : T) (y : A Y) (p : R y x) => PFix_F (x:=y) (PAcc_inv r p)) = PFix_F (x:=x) r.
  Proof.
   destruct r using PAcc_inv_dep; auto.
  Qed.

  Definition PFix (X : T) (x : A X) := PFix_F (PRwf x).
```

Proof that well-founded induction satisfies the fixpoint equation. It requires an extra property of the functional

```
  Hypothesis
    PF_ext :
      forall (X : T) (x : A X) (f g : forall (Y : T) (y : A Y), R y x -> P y),
        (forall (Y : T) (y:A Y) (p : R y x), f Y y p = g Y y p) -> F f = F g.

  Lemma PFix_F_inv : forall (X : T) (x : A X) (r s : PAcc x), PFix_F r = PFix_F s.
  Proof.
   intros X x; induction (PRwf x); intros.
   rewrite <- (PFix_F_eq r). rewrite <- (PFix_F_eq s).
   apply PF_ext; auto.
  Qed.

  Lemma PFix_eq : forall (X : T) (x : A X), PFix x = F (fun (Y : T) (y : A Y) (p : R y x) => PFix y)
  Proof.
    intros x X. unfold PFix in |- *.
    rewrite <- PFix_F_eq.
    apply PF_ext; intros.
    apply PFix_F_inv.
  Qed.

End PFixPoint.
End PWF.
```

A.1 Measure less-than relation analogue

```
Section poly_well_founded_Nat.
  Variable T : Type.
  Variable A : T -> Type.
  Variable f : forall X : T, A X -> nat.

  Definition ltofp {X : T} (a : A X) {Y : T} (b : A Y) := f a < f b.

  Theorem ltofp_pwf : poly_well_founded A (@ltofp).
  Proof.
    red.
    cut (forall n (X : T) (a : A X), f a < n -> PAcc A (@ltofp) a).
    intros H X a. apply (H (S (f a))). apply le_n.

    induction n. intros X a H. inversion H.

    intros X a ltSma. apply PAcc_intro.
    unfold ltofp. intros Y b ltfafy.
    apply IHn. apply lt_le_trans with (f a); auto with arith. Defined.

End poly_well_founded_Nat.
```

A.2 Example

```
Require Import List.

Notation " [ ] " := nil : list_scope.
Notation " [ x ] " := (cons x nil) : list_scope.
Notation " [ x ; .. ; y ] " := (cons x .. (cons y nil) ..) : list_scope.

Example list_example : ltofp list length [true] [1; 2; 3].
Proof. compute. auto with arith. Qed.
```