Utils.v

```
Require Import Coq.Classes.EquivDec.
Require Import Coq.Relations.Relation_Operators.
Require Import Coq.Lists.List.
Require Import Coq.omega.Omega.
Require Import Datatypes.
Require Import Omega.

Import ListNotations.
Import EqNotations.

Lemma in_tuple_snd {A} {eq_dec : EqDec A eq} : forall (l : list (A*A)) a, {b & In (a, b) l} + {forall b, ~In (a,b) l}.
Proof.
  intros.
  pose proof prod_eqdec eq_dec eq_dec as tuple_eq_dec.
  induction l.
  - right. auto.
  - destruct IHl.
    + destruct s. left. intros. exists x. constructor 2. assumption.
    + destruct a0. destruct (a == a0).
      * rewrite e. left. exists a1. constructor. auto.
      * right. intros. intros F. inversion F.
        ** apply c. injection H. intros. subst. reflexivity.
        ** pose proof n b. contradiction.
Qed.


Lemma ge_0_eq : forall m, 0 >= m -> 0 = m. Proof. intros. omega. Qed.

Lemma Odd_plus_Even_is_Odd : forall n m, Nat.Odd n -> Nat.Even m -> Nat.Odd (n + m).
Proof.
  intros n m [n' Hodd] [m' Heven].
  unfold Nat.Odd.
  exists (n' + m').
  omega.
Qed.


Lemma count_occ_split {A} : forall eq_dec (x : A) l1 l2, count_occ eq_dec (l1 ++ l2) x = count_occ eq_dec l1 x + count_occ eq_dec l2 x.
Proof.
  induction l1.
  - reflexivity.
  - intros. rewrite <- app_comm_cons. simpl. destruct (eq_dec a x) eqn:Heq.
    + rewrite IHl1. firstorder.
    + apply IHl1.
Qed.

Lemma count_occ_head_last {A} : forall eq_dec (x: A) y l, count_occ eq_dec (y :: l) x = count_occ eq_dec (l ++ [y]) x.
Proof.
  intros.
  rewrite count_occ_split.
```

```
  simpl.
  destruct (eq_dec y x) eqn:Heq.
  - firstorder.
  - firstorder.
Qed.

Lemma count_occ_last_neq {A} : forall eq_dec (x: A) y l, y <> x -> count_occ eq_dec (l ++ [y]) x = count_occ eq_dec l x.
Proof.
  intros.
  rewrite <- count_occ_head_last.
  apply count_occ_cons_neq.
  assumption.
Qed.

Lemma In_head_set {A} {eqdec : EqDec A eq} : forall l (a x : A), In x (a :: l) -> {a = x} + {In x l}.
Proof.
  intros.
  destruct (a == x).
  - left. assumption.
  - right. destruct H.
    + contradiction.
    + assumption.
Qed.

Lemma In_app_sumbool {A} {eqdec : EqDec A eq}: forall (a : A) l1 l2,  In a (l1 ++ l2) -> {In a l1} + {In a l2}.
Proof.
  intros. apply in_app_or in H. destruct (in_dec eqdec a l1).
  left. assumption. right. destruct H. contradiction. assumption.
Qed.

Lemma in_map_cons {A} : forall (x : A) xs ys, In (x::xs) (map (cons x) ys) -> In xs ys.
Proof.
  induction ys.
  - auto.
  - simpl in *. intros. destruct H.
    + left. inversion H. reflexivity.
    + right. apply IHys. assumption.
Qed.

Lemma in_map_cons_not {A} : forall (x y : A) xs ys, x <> y -> ~(In (x::xs) (map (cons y) ys)).
Proof.
  induction ys.
  - intros Heq F. exact F.
  - intros Heq F. destruct F.
    + inversion H. symmetry in H1. contradiction.
    + apply IHys in Heq. contradiction.
Qed.

Lemma in_map_cons_iff {A} : forall (x : A) xs ys, In (x::xs) (map (cons x) ys) <-> In xs ys.
Proof.
```

```
  intros. split.
  - apply in_map_cons.
  - intros. induction ys.
    + inversion H.
    + simpl. destruct H.
      * left. subst. reflexivity.
      * right. apply IHys. apply H.
Qed.

Lemma some_eq : forall (T : Type) (a b : T), a = b <-> Some a = Some b.
Proof. intros. split.
  - intros Heq. subst. reflexivity.
  - intros Heq. inversion Heq. reflexivity.
Qed.

Inductive Forall_T {A : Type} (P : A -> Type) : list A -> Type :=
| Forall_T_nil : Forall_T P nil
| Forall_T_cons : forall (x : A) (l : list A), P x -> Forall_T P l -> Forall_T P (x :: l)
.

Lemma Forall_T_forall {A P} {eqdec : EqDec A eq} {l: list A} : Forall_T P l -> (forall x : A, In x l -> P x).
Proof.
  induction 1;intros.
  - inversion H.
  - destruct (x == x0).
    + rewrite e in p. assumption.
    + apply IHX.
      inversion H.
      * pose proof (Equivalence.equiv_reflexive_obligation_1 _ x0). contradiction.
      * assumption.
Qed.

Inductive Forall2_T {A B : Type} (R : A -> B -> Type) : list A -> list B -> Type :=
  | Forall2_T_nil : Forall2_T R [] []
  | Forall2_T_cons : forall (x : A) (y : B) (l : list A) (l' : list B),
                     R x y -> Forall2_T R l l' -> Forall2_T R (x :: l) (y :: l').

Inductive Forall2_T_idx {A B : Type} (R : nat -> A -> B -> Type) : nat -> list A -> list B -> Type :=
  | Forall2_T_idx_nil : Forall2_T_idx R 0 [] []
  | Forall2_T_idx_cons : forall (x : A) (y : B) (l : list A) (l' : list B) (n : nat),
                     R n x y -> Forall2_T_idx R n l l' -> Forall2_T_idx R (S n) (x :: l) (y :: l').

Ltac dec_eq_one :=
    match goal with
    | H : ?x =/= ?x |- _ => elimtype False; apply H; reflexivity
    | H : ?x === ?y |- _ => red in H; subst
    | |- { ?x === ?x } + { _ } => left; reflexivity
    | |- { _ } + { ?x =/= ?x } => left; f_equal
    | |- { ?x === ?y } + { _ } => right; let H := fresh in intro H; red in H;
```

```
            (injection H || discriminate); intros; subst
    end.

Ltac dec_eq := try solve [ repeat dec_eq_one ];
    repeat match goal with H : _ === _ |- _ => red in H; subst end.

Ltac isfalse := intros F; inversion F; try contradiction.


Definition kleisli_option {A B C : Type} (f : A -> (option B)) (g : B -> option C) x :=
    match f x with
    | None => None
    | Some y => g(y)
    end.

Definition fmap_option {A B : Type} (f : A -> B) (a : option A) : option B :=
    match a with
    | None => None
    | Some x => Some (f x)
    end.

Definition bind_option {A B : Type} (m : option A) (f : A -> option B) : option B :=
    match m with
    | None => None
    | Some x => f x
    end.

Notation "A >=> B" := (kleisli_option A B) (at level 50, left associativity).
Notation "m >>= f" := (bind_option m f) (at level 50, left associativity).

Lemma kleisli_to_bind_option {A B C : Type} :
    forall (m : A -> option B) (n : B -> option C) x,
        (m >=> n) x = m x >>= (fun y => n y).
Proof.
    intros. unfold kleisli_option. destruct (m x); reflexivity.
Qed.

Lemma monad_law_option_1 {A B: Type} : forall (f: A -> option B) a , Some a >>= f = f a.
Proof. reflexivity. Qed.

Lemma monad_law_option_2 {A : Type} : forall (m : option A) , m >>= Some = m.
Proof.
    destruct m; reflexivity.
Qed.

Lemma monad_law_option_3 {A B C: Type} :
    forall m (f : A -> option B) (g : B -> option C),
        (m >>= f) >>= g = m >>= (fun x => f x >>= g).
Proof.
    destruct m; reflexivity.
Qed.
```

```
Lemma split_rev : forall A (ts1:list A) ts2, rev (ts1 ++ ts2) = rev ts2 ++ rev ts1.
Proof.
  induction ts1.
  - intros. simpl. rewrite app_nil_r. reflexivity.
  - intros. simpl. rewrite IHts1. rewrite app_assoc. reflexivity.
Qed.

Lemma rev_head_last : forall A (ts : list A) a, rev (a :: ts) = rev ts ++ [a].
Proof.
  intros.
  assert (a :: ts = [a] ++ ts). reflexivity.
  rewrite H. apply split_rev.
Qed.


Inductive Forall2_rev {A B} R : list A -> list B -> Prop :=
| Forall2_rev_nil : Forall2_rev R [] []
| Forall2_rev_cons : forall x y l l',
              R x y -> Forall2_rev R l l' -> Forall2_rev R (rev (x :: l)) (rev (y :: l')).

Hint Constructors Forall2_rev.

Lemma Forall2_head : forall A B R (l1 : list A) (l2 : list B) a b ,
  Forall2 R (a :: l1) (b :: l2) -> R a b.
Proof.
  intros. inversion H. assumption.
Qed.

Lemma Forall2_head_T : forall A B R (l1 : list A) (l2 : list B) a b ,
  Forall2_T R (a :: l1) (b :: l2) -> R a b.
Proof.
  intros. inversion X. assumption.
Qed.

Lemma app_eq_length_eq : forall A (l1 l2 : list A),
  l1 = l2 -> length l1 = length l2.
Proof.
  intros. subst. reflexivity.
Qed.

Lemma app_singl_eq_singl_nil : forall A (l: list A) a b,
  [a] = l ++ [b] -> l = [] /\ a = b.
Proof.
  intros.
  induction l.
  - simpl in H. inversion H. auto.
  - apply app_eq_length_eq in H. simpl in H. inversion H.
    rewrite app_length in H1. simpl in H1. rewrite <- plus_n_Sm in H1. inversion H1.
Qed.

Lemma l1_le_length_split : forall A (l: list A),
  1 <= length l -> exists a l', l = a::l'.
Proof.
```

```
  destruct l.
  - isfalse.
  - intros. exists a. exists l. reflexivity.
Qed.

Lemma Sn_impl_1_lt_m : forall m n, S n = m → 1 ⩽ m.
Proof.
  intros. omega.
Qed.

Lemma Forall2_idx {A B: Type} : forall (n : A) ns (m : B) ms R , Forall2 R ns ms → forall i, nth_error ns i = Some n →
  nth_error ms i = Some m → R n m.
Proof.
  induction 1.
  - intros. destruct i; inversion H.
  - intros. destruct i.
    + simpl in *. inversion H1. inversion H2. subst. assumption.
    + simpl in *. eapply IHForall2.
      { apply H1. }
      { apply H2. }
Qed.

Lemma Forall2_idx_T {A B: Type} : forall (n : A) ns (m : B) ms R , Forall2_T R ns ms → forall i, nth_error ns i = Some n →
  nth_error ms i = Some m → R n m.
Proof.
  induction 1.
  - intros. destruct i; inversion H.
  - intros. destruct i.
    + simpl in *. inversion H. inversion H0. subst. assumption.
    + simpl in *. eapply IHX.
      { apply H. }
      { apply H0. }
Qed.

Lemma Forall2_length : forall A B R (l : list A) (l' : list B), Forall2 R l l' → length l = length l'.
Proof.
  induction 1.
  - reflexivity.
  - simpl. rewrite IHForall2. reflexivity.
Qed.

Lemma Forall2_length_T : forall A B R (l : list A) (l' : list B), Forall2_T R l l' → length l = length l'.
Proof.
  induction 1.
  - reflexivity.
  - simpl. rewrite IHX. reflexivity.
Qed.

Lemma nth_error_last_length {A: Type} : forall ls (x : A), nth_error (ls ++ [x]) (length ls) = Some x.
Proof.
  intros.
```

```
  induction ls.
  - reflexivity.
  - simpl. assumption.
Qed.

Lemma Forall2_last_length {A B: Type} : forall (n : A) ns (m : B) ms R , Forall2 R ns ms →
  nth_error ns ((length ns) - 1) = Some n →
  nth_error ms ((length ms) - 1) = Some m → R n m.
Proof.
  intros.
  apply Forall2_idx with ns ms (length ns - 1); try assumption.
  - apply Forall2_length in H. rewrite H. assumption.
Qed.

Lemma Forall2_last_length_T {A B: Type} : forall (n : A) ns (m : B) ms R , Forall2_T R ns ms →
  nth_error ns ((length ns) - 1) = Some n →
  nth_error ms ((length ms) - 1) = Some m → R n m.
Proof.
  intros.
  apply Forall2_idx_T with ns ms (length ns - 1); try assumption.
  - apply Forall2_length_T in X. rewrite X. assumption.
Qed.

Lemma Forall2_last {A B: Type} : forall (n : A) ns (m : B) ms R , Forall2 R (ns ++ [n]) (ms ++ [m]) →
  R n m.
Proof.
  intros.
  apply Forall2_last_length with (ns ++ [n]) (ms ++ [m]).
  - assumption.
  - rewrite app_length. rewrite plus_comm. simpl. rewrite ← minus_n_O.
    apply nth_error_last_length.
  - rewrite app_length. rewrite plus_comm. simpl. rewrite ← minus_n_O.
    apply nth_error_last_length.
Qed.

Lemma Forall2_last_T {A B: Type} : forall (n : A) ns (m : B) ms R , Forall2_T R (ns ++ [n]) (ms ++ [m]) →
  R n m.
Proof.
  intros.
  apply Forall2_last_length_T with (ns ++ [n]) (ms ++ [m]).
  - assumption.
  - rewrite app_length. rewrite plus_comm. simpl. rewrite ← minus_n_O.
    apply nth_error_last_length.
  - rewrite app_length. rewrite plus_comm. simpl. rewrite ← minus_n_O.
    apply nth_error_last_length.
Qed.

Lemma Forall2_firstn {A B: Type} : forall ns  ms (R : A → B → Prop), Forall2 R ns ms →
  forall n ans ams, firstn n ns = ans → firstn n ms = ams → Forall2 R ans ams.
Proof.
  induction 1.
```

```
  - intros. destruct n; simpl in *; subst; constructor.
  - intros. destruct n.
    + subst. constructor.
    + simpl in *. destruct ans, ams.
      { constructor. }
      { inversion H1. }
      { inversion H2. }
      { inversion H1. inversion H2. constructor.
        - subst. assumption.
        - eapply IHForall2; try reflexivity. }
Qed.

Lemma Forall2_firstn_T {A B: Type} : forall ns  ms (R : A → B → Type), Forall2_T R ns ms →
  forall n ans ams, firstn n ns = ans → firstn n ms = ams → Forall2_T R ans ams.
Proof.
  induction 1.
  - intros. destruct n; simpl in *; subst; constructor.
  - intros. destruct n.
    + subst. constructor.
    + simpl in *. destruct ans, ams.
      { constructor. }
      { inversion H. }
      { inversion H0. }
      { inversion H. inversion H0. constructor.
        - subst. assumption.
        - eapply IHX; try reflexivity. }
Qed.

Lemma firstn_init_length {A : Type} : forall init (x : A), firstn (length init) (init ++ [x]) = init.
Proof.
  intros.
  induction init.
  - reflexivity.
  - simpl. rewrite IHinit. reflexivity.
Qed.

Lemma Forall2_init_length {A B: Type} : forall (ans : list A) ns (ams : list B) ms R , Forall2 R ns ms →
  firstn ((length ms) - 1) ms = ams →
  firstn ((length ns) - 1) ns = ans → Forall2 R ans ams.
Proof.
  intros.
  eapply Forall2_firstn.
  - apply H.
  - apply H1.
  - assert (length ns = length ms).
    { eapply Forall2_length. apply H. } rewrite H2. apply H0.
Qed.

Lemma Forall2_init_length_T {A B: Type} : forall (ans : list A) ns (ams : list B) ms R , Forall2_T R ns ms →
  firstn ((length ms) - 1) ms = ams →
  firstn ((length ns) - 1) ns = ans → Forall2_T R ans ams.
```

```
Proof.
  intros.
  eapply Forall2_firstn_T.
  - apply X.
  - apply H0.
  - assert (length ns = length ms).
    { eapply Forall2_length_T. apply X. } rewrite H1. apply H.
Qed.

Lemma Forall2_init {A B: Type} : forall (n : A) ns (m : B) ms R , Forall2 R (ns
 ↪ ++ [n]) (ms ++ [m]) →
  Forall2 R ns ms.
Proof.
  intros.
  eapply Forall2_init_length.
  - apply H.
  - rewrite app_length. rewrite plus_comm. simpl. rewrite ← minus_n_O. apply
   ↪ firstn_init_length.
  - rewrite app_length. rewrite plus_comm. simpl. rewrite ← minus_n_O. apply
   ↪ firstn_init_length.
Qed.

Lemma Forall2_init_T {A B: Type} : forall (n : A) ns (m : B) ms R , Forall2_T R
 ↪ (ns ++ [n]) (ms ++ [m]) →
  Forall2_T R ns ms.
Proof.
  intros.
  eapply Forall2_init_length_T.
  - apply X.
  - rewrite app_length. rewrite plus_comm. simpl. rewrite ← minus_n_O. apply
   ↪ firstn_init_length.
  - rewrite app_length. rewrite plus_comm. simpl. rewrite ← minus_n_O. apply
   ↪ firstn_init_length.
Qed.

Lemma Forall2_split_last {A B: Type} : forall (n : A) ns (m : B) ms R , Forall2
 ↪ R (ns ++ [n]) (ms ++ [m]) ↣
  Forall2 R ns ms /\ R n m.
Proof.
  intros.
  split.
  - split.
    + eapply Forall2_init. apply H.
    + eapply Forall2_last. apply H.
  - intros. destruct H.
    induction H.
    + simpl. constructor.
      * assumption.
      * constructor.
    + simpl. constructor; assumption.
Qed.

Lemma Forall2_split_last_T {A B: Type} : forall (n : A) ns (m : B) ms R ,
 ↪ Forall2_T R (ns ++ [n]) (ms ++ [m]) →
  prod (Forall2_T R ns ms)  (R n m).
Proof.
```

```
  intros.
  split.
    + eapply Forall2_init_T. apply X.
    + eapply Forall2_last_T. apply X.
Qed.

Lemma Forall2_split_last_T_r {A B: Type} : forall (n : A) ns (m : B) ms R , prod (Forall2_T R ns ms)  (R n m) -> Forall2_T R (ns ++ [n]) (ms ++ [m]).
Proof.
 intros. destruct X.
 induction f.
    + simpl. constructor.
      * assumption.
      * constructor.
    + simpl. constructor; assumption.
Qed.

Lemma Forall2_head_to_last {A B : Type} : forall n m ns ms (R : A -> B -> Prop), Forall2 R (n :: ns) (m :: ms) <-> Forall2 R (ns ++ [n]) (ms ++ [m]).
Proof.
  intros.
  split.
  - intros. apply Forall2_split_last. inversion H. split; assumption.
  - intros. apply Forall2_split_last in H. destruct H. constructor; assumption.
Qed.

Lemma Forall2_head_to_last_T {A B : Type} : forall n m ns ms (R : A -> B -> Type), Forall2_T R (n :: ns) (m :: ms) -> Forall2_T R (ns ++ [n]) (ms ++ [m]).
Proof.
  intros. apply Forall2_split_last_T_r. inversion X. split; assumption.
Qed.

Lemma Forall2_head_to_last_T_r {A B : Type} : forall n m ns ms (R : A -> B -> Type), Forall2_T R (ns ++ [n]) (ms ++ [m]) -> Forall2_T R (n :: ns) (m :: ms).
Proof.
  intros. apply Forall2_split_last_T in X. destruct X. constructor; assumption.
Qed.

Lemma rev_nil_iff_nil {A : Type} : forall (ms : list A), [] = rev ms <-> ms = [].
Proof.
  intros.
  split.
  - intros. destruct ms.
    + reflexivity.
    + inversion H. apply app_eq_length_eq in H1. rewrite app_length in H1. simpl in H1.
      rewrite plus_comm in H1. inversion H1.
  - intros. subst. reflexivity.
Qed.

Lemma app_last_eq {A : Type} : forall (ms ns: list A) a b, ms ++ [a] = ns ++ [b] ->
  ms = ns /\ a = b.
```

```
Proof.
  induction ms.
  - induction ns.
    + simpl. intros. split. reflexivity. inversion H. reflexivity.
    + simpl. intros. inversion H. symmetry in H2. apply app_eq_nil in H2.
      inversion H2. inversion H3.
  - induction ns.
    + simpl. intros. inversion H. apply app_eq_nil in H2.
      inversion H2. inversion H3.
    + simpl in *. intros. split.
      { inversion H. apply IHms in H2. inversion H2. subst. reflexivity. }
      { inversion H. apply IHms in H2. inversion H2. assumption. }
Qed.

Lemma rev_eq {A : Type} : forall (ms ns: list A), rev ms = rev ns <-> ms = ns.
Proof.
  intros.
  split.
  - intros. remember (length ms) as lms. generalize dependent ms. generalize dependent ns. induction (lms).
    + intros. apply app_eq_length_eq in H. rewrite rev_length in H. rewrite rev_length in H.
      symmetry in Heqlms. apply length_zero_iff_nil in Heqlms. subst. simpl in H.
      symmetry in H. apply length_zero_iff_nil in H. subst. reflexivity.
    + intros. assert (length ms = length ns).
      { apply app_eq_length_eq in H. rewrite rev_length in H. rewrite rev_length in H. assumption. }
      assert (1 <= length ms).
      { rewrite <- Heqlms. firstorder. }
      assert (1 <= length ns). { rewrite <- H0. assumption. }
      apply  l1_le_length_split in H1.
      apply  l1_le_length_split in H2.
      inversion H1. inversion H2. inversion H3. inversion H4.
      subst. simpl in H. apply app_last_eq in H.
      destruct H. apply IHn in H.
      { subst. reflexivity. }
      { inversion Heqlms. reflexivity. }
  - intros. subst. reflexivity.
Qed.

Lemma rev_cons_iff_last {A : Type} : forall (ms : list A) x l , x :: l = rev ms <->
  ms = rev l ++ [x].
Proof.
  intros.
  split.
  - intros.
    rewrite <- (rev_involutive ms). rewrite <- rev_head_last.
    apply rev_eq. symmetry. assumption.
  - intros. subst. rewrite <- rev_head_last. rewrite rev_involutive.
    reflexivity.
Qed.

Lemma Forall2_is_rev {A B: Type} : forall ns ms {R : A -> B -> Prop}, Forall2 R ns ms <-> Forall2 R (rev ns) (rev ms).
```

```
Proof.
  intros.
  split.
  - intros. induction H.
    + simpl. constructor.
    + simpl. apply Forall2_head_to_last. constructor; assumption.
  - intros. remember (rev ns) as rns. remember (rev ms) as rms.
    generalize dependent ms. generalize dependent ns.
    induction H.
    + intros. apply rev_nil_iff_nil in Heqrns.
      apply rev_nil_iff_nil in Heqrms. subst. constructor.
    + intros.
      apply rev_cons_iff_last in Heqrns.
      apply rev_cons_iff_last in Heqrms.
      subst. eapply Forall2_split_last. split.
      { apply IHForall2.
        - symmetry. apply rev_involutive.
        - symmetry. apply rev_involutive. }
      assumption.
Qed.

Lemma Forall2_T_is_rev {A B: Type} : forall ns ms {R : A → B → Type},
 ↪  Forall2_T R ns ms → Forall2_T R (rev ns) (rev ms).
Proof.
  induction 1.
    + simpl. constructor.
    + simpl. apply Forall2_head_to_last_T. constructor; assumption.
Qed.

Lemma Forall2_T_is_rev_r {A B: Type} : forall ns ms {R : A → B → Type},
 ↪  Forall2_T R (rev ns) (rev ms) → Forall2_T R ns ms .
Proof.
  intros. remember (rev ns) as rns. remember (rev ms) as rms.
    generalize dependent ms. generalize dependent ns.
    induction X.
    + intros. apply rev_nil_iff_nil in Heqrns.
      apply rev_nil_iff_nil in Heqrms. subst. constructor.
    + intros.
      apply rev_cons_iff_last in Heqrns.
      apply rev_cons_iff_last in Heqrms.
      subst. eapply Forall2_split_last_T_r. split.
      { apply IHX.
        - symmetry. apply rev_involutive.
        - symmetry. apply rev_involutive. }
      assumption.
Qed.


Lemma hd_none_impl_nil {T: Type} : forall (ms : list T), hd_error ms = None →
 ↪  ms = nil.
Proof.
    intros.
    induction ms.
    - reflexivity.
    - simpl in H. discriminate H.
Qed.
```

```
Lemma list_nonmt_split {T: Type} : forall (ms : list T), ms <> nil <-> exists head tail, ms = head :: tail.
Proof.
    intros. split.
    - intros. destruct (hd_error ms) eqn:he.
      + exists t. exists (tl ms). remember (tl ms) as tail.
      assert (hd_error ms = Some t /\ tl ms = tail). auto.
      apply hd_error_tl_repr in H0. assumption.
      + eapply hd_none_impl_nil in he. contradiction.
    - intros.
      destruct H as [head [tail Heq]].
      intros F. subst. discriminate F.
Qed.

Fixpoint enumerate_aux {T : Type} (ls : list T) (start : nat) : list (nat * T) :=
    match ls with
    | [] => []
    | x :: xs => (start, x) :: enumerate_aux xs (Datatypes.S start)
    end.

Definition enumerate {T: Type} (ls: list T) : list (nat * T) :=
    enumerate_aux ls 0.


Lemma prooflater : False.
Proof.
  Admitted.

Ltac prooflater := exfalso; apply prooflater.

Lemma list_split_rev {A}: forall (ms : list A), (exists mshead mstail, ms = mshead :: mstail) <-> exists msinit mslast, ms = msinit ++ [mslast].
Proof.
  intros. split.
  + intros. destruct H as [hd [tl mssplit]]. assert (ms <> nil).
    { intros F. subst. discriminate F. }
    apply exists_last in H. destruct H as [x  [x0 Heq]]. exists x. exists x0. assumption.
  + intros. destruct H as [init [tail mssplit]]. destruct ms.
    - symmetry in mssplit. apply app_eq_nil in mssplit. destruct mssplit. subst. discriminate H0.
    - exists a. exists ms. reflexivity.
Qed.

Definition eq_ind_T {A : Type } :=
fun (x : A) (P : A -> Type) (f : P x) (y : A) (e : x = y) =>
match e in (_ = y0) return (P y0) with
| eq_refl => f
end.

Definition list_ind_T {A : Type} :=
fun (P : list A -> Type) (f : P []) (f0 : forall (a : A) (l : list A), P l -> P (a :: l)) =>
fix F (l : list A) : P l := match l as l0 return (P l0) with
```

```
                                | [] ⇒ f
                                | y :: l0 ⇒ f0 y l0 (F l0)
                                end.

Definition rev_list_ind_T {A : Type} :=
fun (P : list A → Type) (H : P [])
  (H0 : forall (a : A) (l : list A), P (rev l) → P (rev (a :: l))) (l : list A) ⇒
  list_ind_T (fun l0 : list A ⇒ P (rev l0)) H (fun (a : A) (l0 : list A) (IHl : P (rev l0)) ⇒ H0 a l0 IHl) l.

Definition rev_ind_T {A : Type} :=
fun (P : list A → Type) (H : P []) (H0 : forall (x : A) (l : list A), P l → P (l ++ [x]))
  (l : list A) ⇒
(fun E : rev (rev l) = l ⇒
 eq_ind_T (rev (rev l)) (fun l0 : list A ⇒ P l0)
   (rev_list_ind_T P H (fun (a : A) (l0 : list A) (H1 : P (rev l0)) ⇒ H0 a (rev l0) H1) (rev l)) l E)
  (rev_involutive l).

Lemma nth_error_nil {A: Type} :forall x, @nth_error A [] x = None.
Proof.
  induction x; reflexivity.
Qed.



Lemma nth_error_map {A B} : forall x (l : list A) (f: A → B),
  nth_error ((map f) l) x = match (nth_error l x) with
                                                                      | None ⇒ None
                                                                      | Some a ⇒ Some (f a)
                                                                      end.
Proof.
  induction x.
  - intros. destruct l; reflexivity.
  - intros. destruct l.
    + reflexivity.
    + simpl. apply IHx.
Qed.
Fixpoint nth_ok {A} (l : list A) (n : nat) : (n < length l) → A :=
  match n with
  | 0 ⇒ match l with
        | [] ⇒ fun proof ⇒ False_rect A (Nat.nlt_0_r _ proof)
        | (a :: _ ) ⇒ fun _ ⇒ a
        end
  | S m ⇒ match l with
        | [] ⇒ fun proof ⇒ False_rect A (Nat.nlt_0_r _ proof)
        | (a :: aa) ⇒ fun proof ⇒ nth_ok aa m (Lt.lt_S_n _ _ proof)
        end
```

```
  end.

Lemma nth_ok_nth_error {A} : forall n (l : list A) p a,
    nth_ok l n p = a <-> nth_error l n = Some a.
Proof.
  intros.
  split.
  - revert n l p a.
    induction n.
    + intros [|h l] p a; try solve [inversion p].
      simpl. intros. subst. reflexivity.
    + simpl. intros [|h l] p a; try solve [inversion p].
      simpl in *. apply IHn.
  - revert n l p a. induction n.
    + simpl. intros [|h l] p a; intros H; try solve [inversion p]; try (inversion H; subst); try reflexivity.
    + simpl. intros [|h l] p a; intros H; try solve [inversion p]. simpl in *. apply IHn. assumption.
Qed.

Lemma nth_ok_map {A B} : forall n l (f: A -> B) p, nth_ok (map f l) n p = f (nth_ok l n (rew (map_length _ _ ) in p)).
Proof.
  intros.
  remember (nth_ok (map f l) n p) as Ha.
  remember (nth_ok l n (rew [lt n] map_length f l in p)) as Hb.
  symmetry in HeqHa.
  symmetry in HeqHb.
  rewrite nth_ok_nth_error in HeqHa.
  rewrite nth_ok_nth_error in HeqHb.
  rewrite nth_error_map in HeqHa. destruct (nth_error l n).
  - inversion HeqHb. inversion HeqHa. subst. reflexivity.
  - discriminate HeqHa.
Qed.

Definition tuple_dec {A} {eqdec : EqDec A eq} := prod_eqdec eqdec eqdec.

Fixpoint Inb {A} {eqdec : EqDec A eq} (x : A) l {struct l} :=
  match l with
  | [] => false
  | a :: ms => orb (x ==b a) (Inb x ms)
  end.

Fixpoint get_adj {A} {eqdec : EqDec A eq} (a : A) (l : list (A * A)) : list A :=
  match l with
  | [] => []
  | (a', b') :: l' => if (a == a') then b' :: get_adj a l' else get_adj a l'
  end.

Inductive in_range_dir {A} (R : list (A * A)) : nat -> A -> A -> Type :=
| in_range_base a b : In (a, b) R -> in_range_dir R 1 a b
| in_range_refl a : in_range_dir R 0 a a
| in_range_follow a b n : in_range_dir R n a b -> in_range_dir R (S n) a b
| in_range_trans a b c n : In (b, c) R -> in_range_dir R n a b -> in_range_dir R (S n) a c.
```

```
Lemma in_range_dir_le {A R} {a b : A} : forall m n, n <= m -> in_range_dir R n a b -> in_range_dir R m a b.
Proof.
  intros.
  remember (m - n) as diff.
  revert n m H X Heqdiff.
  induction diff; intros.
  - assert (n = m). { omega. } subst. assumption.
  - assert (diff = m - S n).
    { omega. } apply (IHdiff (S n) m). omega. apply in_range_follow. assumption. assumption.
Qed.

Lemma in_ex_dec {A} {eqdec : EqDec A eq} (a : A) R : {c & In (a, c) R} + {forall c :A, In (a, c) R -> False}.
Proof.
  induction R.
  - right. intros. inversion H.
  - destruct IHR.
    + destruct s. left. exists x. constructor 2. assumption.
    + destruct (a == fst a0).
      * left. exists (snd a0). constructor. rewrite e. apply surjective_pairing.
      * right. intros. inversion H.
        -- assert (a = fst a0). subst. reflexivity. contradiction.
        -- pose proof f c0. contradiction.
Defined.

Inductive ts_cl_list {A} (R: list (A * A)) : A -> A -> Type :=
  | ts_R_list : forall a b, In (a, b) R -> ts_cl_list R a b
  | ts_symm_list : forall a b, ts_cl_list R a b -> ts_cl_list R b a
  | ts_trans_list : forall a b c, ts_cl_list R a b -> ts_cl_list R b c -> ts_cl_list R a c
.

Inductive eq_cl_list {A} (R: list (A * A)) : A -> A -> Type :=
  | eq_R_list : forall a b, In (a, b) R -> eq_cl_list R a b
  | eq_refl_list_l : forall a b, In (a, b) R -> eq_cl_list R a a
  | eq_refl_list_r : forall a b, In (b, a) R -> eq_cl_list R a a
  | eq_symm_list : forall a b, eq_cl_list R a b -> eq_cl_list R b a
  | eq_trans_list : forall a b c, eq_cl_list R a b -> eq_cl_list R b c -> eq_cl_list R a c
.

Lemma eq_cl_list_pump {A} {R} {a b : A} : eq_cl_list R a b -> forall r, eq_cl_list (r::R) a b.
Proof.
  intros.
  induction X.
  - constructor. constructor 2. assumption.
  - econstructor 2. constructor 2. exact i.
  - econstructor 3. constructor 2. exact i.
  - constructor 4. assumption.
  - econstructor 5. apply IHX1. apply IHX2.
Qed.
```

```
Inductive sym_hull {A} (R: A → A → Type) : A → A → Type :=
| sym_R : forall a b, R a b → sym_hull R a b
| sym_sym : forall a b, R a b → sym_hull R b a.

Definition flip_tuple {A B} : (A * B) → (B * A) := fun tpl ⇒ (snd tpl, fst tpl).

Lemma flip_tuple_invol {A B} (t : A * B) : flip_tuple (flip_tuple t) = t.
Proof.
  unfold flip_tuple. simpl.
  destruct t.
  simpl. reflexivity.
Qed.

Definition flipped {A B} (R: list (A * B)) : list (B * A) := map flip_tuple R.

Definition sym_hull_list {A} (R: list (A * A)) : list (A * A) :=
  R ++ flipped R.

Lemma In_flipped {A B}: forall (R : list (A * B)) a b, In (a, b) R → In (b, a) (flipped R).
Proof.
  induction R.
  - intros. inversion H.
  - intros. destruct H.
    + simpl. left. subst. reflexivity.
    + constructor 2. apply IHR. assumption.
Qed.

Lemma flipped_invol {A B} : forall (R : list (A * B)), flipped (flipped R) = R.
Proof.
  induction R.
  - reflexivity.
  - simpl. rewrite flip_tuple_invol. rewrite IHR. reflexivity.
Qed.

Lemma sym_hull_dec {A} : forall (R : A → A → Type), (forall a b , R a b + (R a b → False)) → forall a b, sym_hull R a b + (sym_hull R a b → False).
Proof.
  intros.
  destruct (X a b).
  - left. constructor. assumption.
  - destruct (X b a).
    + left. constructor 2. assumption.
    + right. intros F. inversion F; contradiction.
Defined.

Definition diag_dom {A} (R: list (A * A)) : list (A * A) :=
  map (fun a ⇒ (fst a, fst a)) R.

Definition diag_codom {A} (R: list (A * A)) : list (A * A) :=
  map (fun a ⇒ (snd a, snd a)) R.

Definition refl_hull {A} (R: list (A * A)) : list (A * A) :=
  R ++ diag_dom R ++ diag_codom R.
```

```
Lemma diag_codom_eq {A} (R : list (A * A)) a b : In (a, b) (diag_codom R) →
                                         a = b.
Proof.
  intros. unfold diag_codom in H.
  induction R.
  - inversion H.
  - simpl in H. destruct H.
    + inversion H. reflexivity.
    + apply IHR. apply H.
Qed.

Lemma diag_dom_in_dec {A} {eqdec: EqDec A eq} (R: list (A * A))
  : forall a b, In (a, b) (diag_dom R) → {c & In (a, c) R}.
Proof.
  induction R.
  - intros. inversion H.
  - intros. unfold diag_dom in H. simpl map in H. apply In_head_set in H.
   ↪  destruct H.
    + inversion e. subst. exists (snd a). constructor. apply surjective_pairing.
    + apply IHR in i. destruct i. eexists. constructor 2. exact i.
Qed.

Lemma diag_codom_in_dec {A} {eqdec: EqDec A eq} (R: list (A * A))
  : forall a b, In (a, b) (diag_codom R) → {c & In (c, a) R}.
Proof.
  induction R.
  - intros. inversion H.
  - intros. unfold diag_codom in H. simpl map in H. apply In_head_set in H.
   ↪  destruct H.
    + inversion e. exists (fst a). constructor. apply surjective_pairing.
    + apply IHR in i. destruct i. eexists. constructor 2. exact i.
Qed.

Lemma diag_dom_eq {A} (R : list (A * A)) a b : In (a, b) (diag_dom R) →
                                         a = b.
Proof.
  intros. unfold diag_dom in H.
  induction R.
  - inversion H.
  - simpl in H. destruct H.
    + inversion H. reflexivity.
    + apply IHR. apply H.
Qed.

Lemma diag_dom_codom_flipped {A} (R : list (A * A)) a:
  In a (diag_dom (R)) ↣ In a (diag_codom (flipped R)).
Proof.
  revert a.
  induction R.
  - reflexivity.
  - intros. split.
    + simpl. intros. destruct H.
      * left. assumption.
      * right. apply IHR. apply H.
    + simpl. intros. destruct H.
      * left. assumption.
```

```
      * right. apply IHR. apply H.
Qed.

Lemma In_refl {A} {eqdec : EqDec A eq} R : forall (a b : A), In (a, b) (refl_hull R) ->
                                                        {c& prod (a = b) (In (a, c) R)} +
                                                        {c& prod (a = b) (In (c, a) R)} + {In (a, b) R}.
Proof.
  intros. unfold refl_hull in H. apply In_app_sumbool in H. destruct H.
  - right. exact i.
  - apply In_app_sumbool in i. destruct i.
    + left. left. pose proof diag_dom_eq _ _ _ i.
      apply diag_dom_in_dec in i. destruct i. eexists. split. assumption. exact i.
    + left. right. pose proof diag_codom_eq _ _ _ i.
      apply diag_codom_in_dec in i. destruct i. eexists. split. assumption. exact i.
Qed.

Inductive trans_hull {A} (R : list (A * A)) : A -> A -> Type :=
| trans_R : forall a b, In (a, b) R -> trans_hull R a b
| trans_trans : forall a b c, trans_hull R a b -> trans_hull R b c -> trans_hull R a c.


Inductive trans_refl_hull {A} (R: list (A * A)) : A -> A -> Type :=
| trans_refl_R : forall a b, In (a, b) R -> trans_refl_hull R a b
| trans_refl_refl_l : forall a b, In (a, b) R -> trans_refl_hull R a a
| trans_refl_refl_r : forall a b, In (b, a) R -> trans_refl_hull R a a
| trans_refl_trans : forall a b c, trans_refl_hull R a b -> trans_refl_hull R b c ->
                              trans_refl_hull R a c.

Inductive list_rel {A} : list (A * A) -> A -> A -> Type :=
| list_rel_head : forall a b x R, x = (a, b) -> list_rel (x::R) a b
| list_rel_tail : forall a b x R, list_rel R a b ->  list_rel (x::R) a b.

Lemma In_sym_list_dec {A} {eqdec: EqDec A eq} (a b : A) R : In (a, b) (sym_hull_list R) -> {In (a, b) R} + {In (b, a) R}.
Proof.
  intros. unfold sym_hull_list in H. apply In_app_sumbool in H. destruct H.
  - left. auto.
  - right. apply In_flipped in i. rewrite flipped_invol in i. auto.
Defined.

Lemma In_sym_list_sym {A} (a b : A) R : In (a, b) (sym_hull_list R) -> In (b, a) (sym_hull_list R).
Proof.
  intros.
  unfold sym_hull_list in *. apply in_app_or in H. apply in_or_app.
  destruct H. right. apply In_flipped. assumption.
  left. apply In_flipped in H. rewrite flipped_invol in H.
  assumption.
```

```
Qed.

Lemma trans_refll_syml_eq_cl_list {A} {eqdec : EqDec A eq}: forall R (a b : A),
    trans_hull (refl_hull (sym_hull_list R)) a b → eq_cl_list R a b.
Proof.
  intros.
  induction X.
  - apply In_refl in i. destruct i as [[[c [Heq HIn]] | [c [Heq HIn]]] | HIn].
    + subst. unfold sym_hull_list in HIn. apply In_app_sumbool in HIn. destruct HIn.
      * econstructor 2. exact i.
      * econstructor 3. apply In_flipped in i. rewrite flipped_invol in i. exact i.
    + subst. unfold sym_hull_list in HIn. apply In_app_sumbool in HIn. destruct HIn.
      * econstructor 3. exact i.
      * econstructor 2. apply In_flipped in i. rewrite flipped_invol in i. exact i.
    + unfold sym_hull_list in HIn. apply In_app_sumbool in HIn. destruct HIn.
      * constructor. assumption.
      * constructor 4. constructor. apply In_flipped in i. rewrite flipped_invol in i. assumption.
  - econstructor 5. exact IHX1. assumption.
Qed.

Lemma trans_refl_sym_eq_cl_list {A} {eqdec : EqDec A eq}: forall R (a b : A), trans_refl_hull (sym_hull_list R) a b → eq_cl_list R a b.
Proof.
  intros.
  induction X.
  - induction R.
    + inversion i.
    + unfold sym_hull_list in i. apply In_app_sumbool in i. destruct i.
      * apply In_head_set in i. destruct i.
        -- subst. constructor. constructor. reflexivity.
        -- constructor. constructor 2. assumption.
      * unfold flipped in i. simpl map in i. apply In_head_set in i. destruct i.
        -- constructor 4. constructor. inversion e. subst. constructor. destruct a0. reflexivity.
        -- apply eq_cl_list_pump. apply IHR. unfold sym_hull_list. apply in_or_app. right. assumption.
  - apply In_sym_list_dec in i. destruct i.
    + econstructor 2. apply i.
    + econstructor 3. apply i.
  - apply In_sym_list_dec in i. destruct i.
    + econstructor 3. apply i.
    + econstructor 2. apply i.
  - econstructor 5. exact IHX1. assumption.
Qed.

Lemma trans_refl_sym_is_sym {A} : forall R (a b : A),
  trans_refl_hull (sym_hull_list R) b a →
  trans_refl_hull (sym_hull_list R) a b.
Proof.
  intros.
  induction X.
```

```
  - constructor. unfold sym_hull_list in *. apply in_app_or in i. apply in_or_app.
    destruct i.
    + right. apply In_flipped. assumption.
    + left. apply In_flipped in H. rewrite flipped_invol in H. assumption.
  - econstructor 2. exact i.
  - econstructor 3. exact i.
  - econstructor 4. apply IHX2. assumption.
Qed.

Lemma refl_skip {A} {eqdec : EqDec A eq} R : forall (a b: A), a <> b -> In (a, b) (refl_hull R) -> In (a, b) R.
Proof.
  intros. apply In_refl in H0. destruct H0 as [[[_ [F _]] | [_ [F _]]]]].
  - contradiction.
  - contradiction.
  - assumption.
Qed.

Lemma refl_sym_is_sym {A} : forall R (a b : A),
    In (a, b) (refl_hull (sym_hull_list R)) -> In (b, a) (refl_hull (sym_hull_list R)).
Proof.
  intros.
  unfold refl_hull in *.
  apply in_app_or in H. destruct H.
  - apply in_or_app. left. unfold sym_hull_list in *. apply in_app_or in H. destruct H.
    + apply in_or_app. right. apply In_flipped. assumption.
    + apply in_or_app. left. apply In_flipped in H. rewrite flipped_invol in H. assumption.
  - apply in_app_or in H. destruct H.
    + apply in_or_app. right. apply in_or_app. epose proof diag_dom_eq _ _ _ H. subst. left. assumption.
    + apply in_or_app. right. apply in_or_app. epose proof diag_codom_eq _ _ _ H. subst. right. assumption.
Qed.

Lemma trans_sym_is_sym {A} : forall R (a b : A),
    trans_hull (sym_hull_list R) a b ->
    trans_hull (sym_hull_list R) b a.
Proof.
  intros.
  induction X.
  - constructor. apply In_sym_list_sym. assumption.
  - econstructor 2. exact IHX2. assumption.
Qed.

Lemma trans_refl_sym_is_sym2 {A} : forall R (a b : A),
    trans_hull (refl_hull (sym_hull_list R)) a b ->
    trans_hull (refl_hull (sym_hull_list R)) b a.
Proof.
  intros.
  induction X.
  - constructor. apply refl_sym_is_sym. assumption.
  - econstructor 2. exact IHX2. assumption.
```

```
Qed.

Lemma ts_cl_list_trans_sym {A} {eqdec: EqDec A eq} : forall R (a b : A),
    ts_cl_list R a b →
    trans_hull (sym_hull_list R) a b.
Proof.
  intros.
  induction X.
  - constructor. unfold sym_hull_list. apply in_or_app. left. assumption.
  - apply trans_sym_is_sym. assumption.
  - econstructor 2. exact IHX1. assumption.
Qed.

Lemma trans_hull_nil {A} : forall (a b : A), trans_hull [] a b → False.
Proof.
  intros.
  induction X.
  - inversion i.
  - auto.
Qed.

Lemma trans_sym_ts_cl_list {A} {eqdec: EqDec A eq} : forall R (a b : A),
    trans_hull (sym_hull_list R) a b →
    ts_cl_list R a b.
Proof.
  intros.
  induction X.
  - unfold sym_hull_list in i. apply In_app_sumbool in i. destruct i.
    + constructor. assumption.
    + apply In_flipped in i. rewrite flipped_invol in i. constructor 2. constructor. assumption.
  - econstructor 3. exact IHX1. exact IHX2.
Qed.


Lemma eq_cl_list_trans_refl_sym2 {A} {eqdec: EqDec A eq} : forall R (a b : A),
    eq_cl_list R a b →
    trans_hull (refl_hull (sym_hull_list R)) a b.
Proof.
  intros.
  induction X.
  - constructor. unfold refl_hull. apply in_or_app. left. unfold sym_hull_list. apply in_or_app. left.
    assumption.
  - constructor. unfold refl_hull. apply in_or_app. right. apply in_or_app.
    left. unfold diag_dom. induction R.
    + inversion i.
    + destruct i.
      * simpl. left. subst. reflexivity.
      * right. rewrite map_app. apply in_or_app. left. clear IHR. induction R.
        -- inversion H.
        -- simpl. destruct H.
           ++ left. destruct a1. injection H. intros. subst. reflexivity.
           ++ right. apply IHR. assumption.
  - constructor. unfold refl_hull. apply in_or_app. right. apply in_or_app.
    right. unfold diag_codom. induction R.
```

```
    + inversion i.
    + destruct i.
      * simpl. left. subst. reflexivity.
      * right. rewrite map_app. apply in_or_app. left. clear IHR. induction R.
        -- inversion H.
        -- simpl. destruct H.
           ++ left. destruct a1. inversion H. reflexivity.
           ++ right. apply IHR. assumption.
  - apply trans_refl_sym_is_sym2. assumption.
  - econstructor 2. apply IHX1. assumption.
Qed.


Lemma eq_cl_list_trans_refl_sym {A} {eqdec: EqDec A eq} : forall R (a b : A),
    eq_cl_list R a b ->
    trans_refl_hull (sym_hull_list R) a b.
Proof.
  intros.
  induction X.
  - repeat constructor. induction R.
    + inversion i.
    + apply In_head_set in i. destruct i. constructor. assumption. constructor 2.
      simpl. apply in_or_app. left. assumption.
  - econstructor 2. unfold sym_hull_list. apply in_or_app. left. exact i.
  - econstructor 3. unfold sym_hull_list. apply in_or_app. left. exact i.
  - apply trans_refl_sym_is_sym. assumption.
  - econstructor 4. exact IHX1. assumption.
Qed.

Lemma trs_eq_cl_dec {A} {eqdec : EqDec A eq} (R : list (A * A)) : (forall a b, trans_refl_hull (sym_hull_list R) a b +
                              (trans_refl_hull (sym_hull_list R) a b -> False)) ->
                              (forall a b, eq_cl_list R a b + (eq_cl_list R a b -> False)).
Proof.
  intros.
  destruct (X a b).
  - left. apply trans_refl_sym_eq_cl_list. assumption.
  - right. intros. apply f. apply eq_cl_list_trans_refl_sym. assumption.
Defined.

Inductive t_path {A} (R: list (A *A)) : list (A * A) -> A -> A -> Type :=
| t_path_R : forall a b, In (a, b) R -> t_path R [(a, b)] a b
| t_path_trans : forall a b c p, In (a, b) R -> t_path R p b c -> t_path R ((a,b) :: p) a c.

Inductive tr_path {A} (R: list (A *A)) : list (A * A) -> A -> A -> Type :=
| tr_path_R : forall a b, In (a, b) R -> tr_path R [(a, b)] a b
| tr_path_refl_l : forall a b, In (a, b) R -> tr_path R [] a a
| tr_path_refl_r : forall a b, In (b, a) R -> tr_path R [] a a
| tr_path_trans : forall a b c p, In (a, b) R -> tr_path R p b c -> tr_path R ((a,b) :: p) a c.

Lemma t_path_nil {A} : forall (a b : A) p, t_path [] p a b -> False.
Proof.
```

```
  intros.
  induction X; inversion i.
Qed.

Lemma t_path_0 {A} : forall (a b : A) R, t_path R [] a b -> False.
Proof.
  intros.
  inversion X.
Qed.

Lemma t_path_start {A} R : forall P (a b a' b': A), t_path R ((a, b)::P) a' b' -> a = a'.
Proof.
  intros.
  inversion X;reflexivity.
Qed.


Lemma t_path_Sn {A} (R : list (A * A)) P : forall (a b x: A),
    t_path R ((a, x) :: P) a b ->  prod (In (a, x) R) (t_path R P x b) + {prod (P = []) (prod (b = x) (In (a, b) R))}.
Proof.
  intros.
  inversion X.
  - subst. right. split. reflexivity. split. reflexivity. assumption.
  - subst. left. split; assumption.
Qed.

Lemma t_path_P_in_R {A} (R : list (A * A)) P (a b : A): t_path R P a b -> forall a' b', In (a', b') P -> In (a', b') R.
Proof.
  induction 1.
  - intros. inversion H. injection H0. intros. subst. assumption. inversion H0.
  - intros. destruct H. injection H. intros. subst. assumption. apply IHX. assumption.
Qed.

Lemma t_path_trans2 {A} : forall P P' (a b c: A) R , t_path R P a b -> t_path R P' b c -> t_path R (P ++ P') a c.
Proof.
  induction P; intros.
  - inversion X.
  - destruct a. pose proof t_path_start _ _ _ _ _ _ X. subst.
    pose proof t_path_P_in_R _ _ _ _ X a0 a1 (in_eq _ _).
    simpl. pose proof t_path_Sn _ _ _ _ _ X as [[Hin Htr]|[Heq [Hnil Hin]]].
    + constructor 2.
      * assumption.
      * eapply IHP. apply Htr. assumption.
    + subst. simpl. constructor 2. assumption. assumption.
Qed.

Lemma t_path_ex {A} R (a b : A) : trans_hull R a b ->
                                  {P & t_path R P a b}.
Proof.
  induction 1.
  - eexists. constructor. assumption.
```

```
  - destruct IHX1. destruct IHX2. exists (x ++ x0). eapply t_path_trans2.
    + apply t.
    + assumption.
Qed.

Lemma t_path_trh {A} R (a b: A) P : t_path R P a b -> trans_hull R a b.
Proof.
  induction 1.
  - constructor. assumption.
  - econstructor 2. constructor. apply i. exact IHX.
Qed.

Lemma t_path_pump {A} R (a b : A) P : t_path R P a b -> forall ab, t_path
 ↪ (ab::R) P a b.
Proof.
  induction 1.
  - intros. constructor. constructor 2. assumption.
  - intros. constructor 2. constructor 2. assumption. apply IHX.
Qed.

Lemma t_path_trans_dec {A} {eqdec : EqDec A eq} R (a b a' b':A)
      (IHR : forall a b : A,
          {P : list (A * A) & t_path R P a b} + {forall P : list (A * A), t_path
           ↪ R P a b -> False}) :
  {P1 & {P2 & ((t_path R P1 a a') * (t_path R P2 b' b))%type }} +
            {forall P1 P2, ((t_path R P1 a a') * (t_path R P2 b' b))%type ->
             ↪ False}.
Proof.
  destruct (IHR a a') as [[P Htr] | Htr].
  - destruct (IHR b' b) as [[P' Htr'] | Htr'].
    + left. eexists. eexists. split. apply Htr. apply Htr'.
    + right. intros P1 P2 [t1 t2]. eapply Htr'. apply t2.
  - right. intros P1 P2 [t1 t2]. eapply Htr. apply t1.
Defined.

Definition ex_in_rel {A} (a : A) R : Type := ({c & In (a,c) R} + {c & In (c, a)
 ↪ R})%type.

Lemma ex_in_rel_dec {A} {eqdec : EqDec A eq} : forall R (a : A), ex_in_rel a R +
 ↪ (ex_in_rel a R -> False).
Proof.
  induction R.
  - intros. right. intros. unfold ex_in_rel in X. destruct X as [[c Hin] | [c
    ↪ Hin]]; inversion Hin.
  - intros. destruct (IHR a0).
    + left. unfold ex_in_rel. destruct e as [[c H]|[c H]].
      * left. exists c. constructor 2. assumption.
      * right. exists c. constructor 2. assumption.
    + destruct a. destruct (a0 == a1).
      * left. unfold  ex_in_rel. rewrite e. right. exists a. constructor.
        ↪ reflexivity.
      * destruct (a0 == a).
        -- rewrite e. left. left. exists a1. constructor. reflexivity.
        -- right. intros. destruct X as [[c' H]|[c' H]].
           ++ apply In_head_set in H. destruct H.
              ** inversion e. subst. apply c0. reflexivity.
```

```
        ** apply f. left. exists c'. assumption.
      ++ apply In_head_set in H. destruct H.
        ** inversion e. subst. apply c. reflexivity.
        ** apply f. right. exists c'. assumption.
Defined.

Lemma t_path_trans_R_and {A} {eqdec: EqDec A eq} : forall R (a b a' b' : A) P',
    t_path ((a', b')::R) P' a b → {P & t_path R P a b} + {(a', b') = (a, b)} +
                                  {P & prod (t_path R P a a') (b = b')} +
                                  {P & prod (t_path R P b' b) (a = a')} +
                                  {P1 & {P2 & prod (t_path R P1 a a') (t_path R P2 b' b)}}.
Proof.
  intros.
  induction X.
  - apply In_head_set in i. destruct i.
    + inversion e. subst. left. left. left. right. auto.
    + left. left. left. left. eexists. constructor. assumption.
  - apply In_head_set in i. destruct i.
    + destruct IHX as [[[[[P Htr]| Heq]| [P [Htr Heq]]]| [P [Htr Heq]]]| [P1 [P2 [Htr1 Htr2]]]]; inversion e; subst.
      * left. right. eexists. split. apply Htr. reflexivity.
      * left. left. left. right. inversion Heq. subst. assumption.
      * left. left. left. right. reflexivity.
      * left. right. eexists. split. apply Htr. reflexivity.
      * left. right. eexists. split. apply Htr2. reflexivity.
    + destruct IHX as [[[[[P Htr]| Heq]| [P [Htr Heq]]]| [P [Htr Heq]]]| [P1 [P2 [Htr1 Htr2]]]].
      * left. left. left. left. eexists. econstructor 2. apply i. apply Htr.
      * inversion Heq. subst. left. left. right. eexists. split. constructor. assumption. reflexivity.
      * left. left. right. eexists. split. econstructor 2. apply i. apply Htr. assumption.
      * right. eexists. eexists. split. constructor. subst. assumption. apply Htr.
      * right. eexists. eexists. split. constructor 2. apply i. apply Htr1. apply Htr2.
Qed.

Lemma prod_dec_ex {A} P Q {pdec : {p : A & P p} + {forall p, P p → False}} {qdec : Q + (Q → False)} :
  {p & (P p * Q)%type} + {forall p, (P p * Q)%type → False}.
Proof.
  destruct pdec as [[p HP]| HP].
  - destruct qdec.
    + left. exists p. split; assumption.
    + right. intros. apply f. destruct X. assumption.
  - right. intros. eapply HP. destruct X. apply p0.
Defined.

Lemma t_path_dec {A} {eqdec : EqDec A eq} : forall R (a b : A), {P & t_path R P a b} + {forall P, (t_path R P a b → False)}.
Proof.
  induction R; intros.
  - right. apply t_path_nil.
  - destruct (IHR a0 b) as [[P IH] | H1].
```

```
    {left. eexists. apply t_path_pump. apply IH. }
    destruct a. destruct ((a, a1) == (a0, b)).
    {left. inversion e. subst. eexists. constructor. constructor. reflexivity. }
    destruct (@prod_dec_ex _ (fun P => t_path R P a0 a) (a1 = b) (IHR a0 a)).
    { destruct (a1 == b). rewrite e. left. reflexivity. right. intros.
     ↪ contradiction. }
    { destruct s as [p [Htr Heq]]. subst. left. eexists. eapply t_path_trans2.
      - apply t_path_pump. apply Htr.
      - constructor. constructor. reflexivity. }
    destruct (@prod_dec_ex _ (fun P => t_path R P a1 b) (a0 = a) (IHR a1 b)).
    { destruct (a0 == a). rewrite e. left. reflexivity. right. intros.
     ↪ contradiction. }
    { destruct s as [p [Htr Heq]]. subst. left. eexists. eapply t_path_trans2.
      - constructor. constructor. reflexivity.
      - apply t_path_pump. apply Htr. }
    destruct (t_path_trans_dec _ a0 b a a1 IHR) as [[P1 [P2 [Ht1 Ht2]]]]. left.
    eexists. eapply t_path_trans2. eapply t_path_trans2. apply t_path_pump.
     ↪ exact Ht1. constructor.
    constructor. reflexivity. apply t_path_pump. apply Ht2.
    right. intros.
    pose proof t_path_trans_R_and _ _ _ _ _ _ X as [[[[[P' Htr]| Heq]| [P' [Htr
     ↪ Heq]]]| [P' [Htr Heq]]]| [P1 [P2 [Htr1 Htr2]]]].
    + eapply H1. apply Htr.
    + contradiction.
    + eapply f. split. apply Htr. subst. reflexivity.
    + eapply f0. split. apply Htr. assumption.
    + eapply f1. split. apply Htr1. apply Htr2.
Defined.

Lemma trans_hull_dec {A} {eqdec: EqDec A eq}: forall R (a b :A), trans_hull R a
 ↪ b + (trans_hull R a b -> False).
Proof.
  intros.
  destruct (t_path_dec R a b) as [[P Htr]|Htr].
  - left. eapply t_path_trh. apply Htr.
  - right. intros. apply t_path_ex in X. destruct X. eapply Htr. apply t.
Defined.

Lemma ts_cl_list_dec {A} {eqdec: EqDec A eq}: forall R (a b: A), ts_cl_list R a
 ↪ b + (ts_cl_list R a b -> False).
Proof.
  intros.
  destruct (trans_hull_dec (sym_hull_list R) a b).
  - left. apply trans_sym_ts_cl_list. assumption.
  - right. intros. apply f. apply ts_cl_list_trans_sym. assumption.
Defined.

Lemma eq_cl_nil_impl_refl {A} : forall (a b :A), eq_cl_list [] a b -> a = b.
Proof.
  induction 1; try inversion i; subst; reflexivity.
Qed.

Inductive eq_cl_bool {A} (R: A -> A -> bool) : A -> A -> Type :=
  | eq_R_bool : forall a b, R a b = true -> eq_cl_bool R a b
  | eq_refl_bool : forall a b, eq_cl_bool R a b -> eq_cl_bool R a a
  | eq_symm_bool : forall a b, eq_cl_bool R a b -> eq_cl_bool R b a
```

```
  | eq_trans_bool : forall a b c, eq_cl_bool R a b → eq_cl_bool R b c → eq_cl_bool R a c
.

Inductive eq_cl_prop {A} (R: A → A → Prop) : A → A → Type :=
  | eq_R_prop : forall a b, R a b → eq_cl_prop R a b
  | eq_refl_prop : forall a b, eq_cl_prop R a b → eq_cl_prop R a a
  | eq_symm_prop : forall a b, eq_cl_prop R a b → eq_cl_prop R b a
  | eq_trans_prop : forall a b c, eq_cl_prop R a b → eq_cl_prop R b c → eq_cl_prop R a c
.
Inductive eq_cl_type {A} (R: A → A → Type) : A → A → Type :=
  | eq_R_type : forall a b, R a b → eq_cl_type R a b
  | eq_refl_type : forall a b, eq_cl_type R a b → eq_cl_type R a a
  | eq_symm_type : forall a b, eq_cl_type R a b → eq_cl_type R b a
  | eq_trans_type : forall a b c, eq_cl_type R a b → eq_cl_type R b c → eq_cl_type R a c
.

Definition range (n: nat) : list nat := seq 0 n.

Lemma app_head {A} : forall (a : A) aa, a :: aa = [a] ++ aa.
Proof.
  auto.
Qed.


Definition combine_with {A B C} (f: A → B → C) := fix dummy (l:list A) (l' :list B) : list C :=
    match l with
      | [] ⇒ []
      | a :: t ⇒ match l' with
                 | [] ⇒ []
                 | b :: t' ⇒ (f a b) :: (dummy t t')
                 end
    end.


Lemma nth_error_combine {A B} : forall n a b (x : A) (y : B), nth_error (combine a b) n = Some (x, y) →
      nth_error a n = Some x /\ nth_error b n = Some y.
  induction n; intros.
  - inversion H. destruct a; destruct b; try discriminate H1.
    simpl in *. inversion H. subst. split; reflexivity.
  - inversion H. destruct a; destruct b; try discriminate H1.
    simpl in *. apply IHn in H1. assumption.
Qed.


Fixpoint all_some {A} (l : list (option A)) : option (list A) :=
  match l with
  | [] ⇒ Some []
  | Some a :: xs ⇒ match all_some xs with
                   | Some xxs ⇒ Some (a :: xxs)
                   | None ⇒ None
                   end
```

```
  | None :: xs ⇒ None
  end.

Definition option_concat {A} (l: list (option (list A))) : option (list A) :=
  match all_some l with
  | Some xs ⇒ Some (concat xs)
  | None ⇒ None
  end.

Instance option_eqdec {A: Type} {equiv0 : Equivalence eq} {innereqdec: EqDec A eq} : EqDec (option A) eq.
Proof.
  intros.
  unfold EqDec.
  intros.
  destruct x; destruct y; try destruct (a == a0) eqn:Haa.
  - left. inversion e. reflexivity.
  - right. intros F. apply some_eq in F. contradiction.
  - right. intros F. discriminate F.
  - right. intros F. discriminate F.
  - left. reflexivity.
Defined.

Lemma in_not_first {A} : forall b a (x : A), In x (a :: b) → a <> x → In x b.
Proof.
  intros.
  inversion H.
  - contradiction.
  - assumption.
Qed.

Lemma all_some_some {A} {eq_dec : EqDec A eq} : forall ls l (x : option (A)), all_some ls = Some l → In x ls →
                                                                       { y & x = Some y /\ In y l }.
Proof.
  induction ls.
  - intros. inversion H0.
  - intros. simpl in H.
    destruct a eqn:Ha; try discriminate H.
    destruct (all_some ls) eqn:Hls; try discriminate H.
    apply some_eq in H.
    destruct (option_eqdec x a).
    + subst. exists a0. split. rewrite e. reflexivity. constructor. reflexivity.
    + apply in_not_first in H0; try (rewrite ← Ha; intros F; symmetry in F; contradiction).
      pose proof (IHls l0 x eq_refl H0). subst. destruct X as [y [Hx Hin]].
      exists y. split. assumption. constructor 2. assumption.
Qed.

Lemma all_some_none_head {A} (a : A) ls : all_some (Some a :: ls) = None → all_some ls = None.
Proof.
  simpl.
  intros.
  destruct (all_some ls); try discriminate H. reflexivity.
```

```
Qed.

Lemma all_some_none_last {A} (a : A) ls : all_some (ls ++ [Some a]) = None → all_some ls = None.
Proof.
  intros.
  induction ls.
  - simpl in H. discriminate H.
  - simpl in H. destruct a0.
    destruct (all_some (ls ++ [Some a])).
    + discriminate H.
    + apply IHls in H.
      simpl. rewrite H. reflexivity.
    + reflexivity.
Qed.

Lemma all_some_some_app_l {A} (l1 l2 : list (option A)) l3 : all_some (l1 ++ l2) = Some l3 →
                                                                exists l4, all_some l1 = Some l4.
Proof.
  revert l3.
  induction l1.
  - simpl. exists []. reflexivity.
  - intros. simpl. rewrite ← app_comm_cons in H. destruct a  eqn:Ha.
    + simpl in H.
      destruct (all_some (_ ++ _)); try discriminate H. pose proof (IHl1 l eq_refl). destruct H0.
      rewrite H0. eexists. reflexivity.
    + simpl in H. discriminate H.
Qed.

Lemma all_some_forall_eq {A} {eqdec : EqDec A eq} (l : list (option A)) sl : all_some l = Some sl →
                                                            forall i, In i l → { s & i = Some s /\ In s sl}.
Proof.
  revert sl.
  induction l.
  - intros. inversion H0.
  - intros. unfold all_some in H at 1. destruct a; try discriminate H. fold (@all_some A) in H.
    destruct (all_some l) eqn:Hall; try discriminate H. apply some_eq in H.
    rewrite app_head in H0. apply In_app_sumbool in H0. destruct H0.
    + exists a. subst. split. symmetry. inversion i0. assumption. exfalso. inversion H. constructor. reflexivity.
    + subst. pose proof (IHl _ eq_refl i i0) as [s [Hisome Hin]].
      exists s. split. assumption. constructor 2. assumption.
Qed.

Lemma le_tail {A n ms} {n0 : A}: S n < length (n0 :: ms) → n < length ms.
Proof.
  intros. simpl in H. apply lt_S_n in H. assumption.
Qed.
```

```
Lemma nth_ok_skip {A}: forall (n0 :A) n ms p1, nth_ok (n0 :: ms) (S n) p1 =
  nth_ok ms n (le_tail p1).
Proof.
  intros.
  simpl.
  remember (nth_ok ms n _) as n1.
  symmetry in Heqn1. symmetry. apply nth_ok_nth_error. apply nth_ok_nth_error in
    Heqn1. assumption.
Qed.

Lemma in_concat {A} : forall l (x : A), In [x] l -> In x (concat l).
Proof.
 induction l.
 - intros. inversion H.
 - simpl. intros x [Heq | Hin].
   + subst. constructor. reflexivity.
   + apply in_or_app. right. apply IHl. assumption.
Qed.

Lemma combine_with_map {A B C}: forall ms ns (f : A -> B -> C),
                combine_with f ms ns = map (fun mi => f (fst mi) (snd mi))
                  (combine ms ns).
Proof.
  induction ms.
  - reflexivity.
  - intros. destruct ns. { reflexivity. }
    simpl. rewrite IHms. reflexivity.
Qed.

Lemma length_combine_with {A} {B}: forall (ms : list A) (f : A -> nat -> B),
  length (combine_with f ms (range (length ms))) = length ms.
Proof.
  intros.
  rewrite combine_with_map.
  rewrite map_length.
  rewrite combine_length.
  unfold range.
  rewrite seq_length. apply Nat.min_id.
Qed.

Lemma lt_comb {A B n} {ms : list A} : forall (f : A -> nat -> B),  n < length ms
  -> n <
  length ( combine_with f ms (range (length ms))).
Proof.
  intros.
  rewrite (length_combine_with ms).
  assumption.
Qed.


Lemma seq_nth_error : forall len start n,
    n < len -> nth_error (seq start len) n = Some (start+n).
  Proof.
    induction len; intros.
    inversion H.
```

```
    simpl seq.
    destruct n; simpl.
    auto with arith.
    rewrite IHlen;simpl; auto with arith.
Qed.

Lemma pair_eq {A B} : forall (a: A) (b: B) a' b', a = a' /\ b = b' <-> (a, b) = (a', b').
Proof.
  intros.
  split.
  - intros [HA HB]. subst. reflexivity.
  - intros. inversion H. subst. split; reflexivity.
Qed.

Lemma equivb_prop {A} {eqdec : EqDec A eq} : forall (a b : A), a ==b b = true <-> a = b.
Proof.
  intros. split; intros.
  - unfold equiv_decb in H. destruct (a == b). inversion e. subst. reflexivity. discriminate H.
  - subst. unfold equiv_decb. destruct (equiv_dec b b).
    + reflexivity.
    + unfold complement in c. exfalso. apply c. reflexivity.
Qed.

Lemma nequivb_prop {A} {eqdec : EqDec A eq} : forall (a b : A), a <>b b = true <-> a <> b.
Proof.
  intros. split; intros; unfold nequiv_decb in *.
  - apply Bool.negb_true_iff in H. unfold equiv_decb in H. destruct (a == b). inversion H. assumption.
  - apply Bool.negb_true_iff. unfold equiv_decb. destruct (a == b).
    + contradiction.
    + reflexivity.
Qed.

Definition Forall2_T_map {A B} {P : A -> B -> Type} {Q aa bb} (forall2t : Forall2_T (fun a b => {p : P a b & Q a b p}) aa bb) : Forall2_T P aa bb :=
  Forall2_T_rect _ _ _ (fun l l' _ => Forall2_T P l l') (Forall2_T_nil _) (
                  fun a b aa bb head tail result => Forall2_T_cons _ _ _ _ _ (projT1 head) result
                 ) aa bb forall2t.

Definition Forall2_T_pair {A B} (P : A -> B -> Type) Q F ms pis f3 :=
  Forall2_T_rect _ _ _ (fun l l' _ => Forall2_T (fun m pi => { p: P m pi & Q m pi p } ) l l') (Forall2_T_nil _) (
                  fun a b aa bb head tail result =>
                    Forall2_T_cons _ a b aa bb (existT (Q a b) head (F a b head)) result
                 ) ms pis f3.

Lemma Forall2_T_map_inv {A B} (P : A -> B -> Type) Q aa bb f3 : forall F, Forall2_T_map (Forall2_T_pair P Q F aa bb f3) = f3.
Proof.
  intro F.
```

```
  induction f3.
  - constructor.
  - simpl. apply f_equal. assumption.
Qed.

Definition Rsub {A} (R R' : A-> A-> Type) := forall (pi pi' : A), R pi pi' -> R' pi pi'.

Definition Rsub_list {A} R R' := forall (pi pi' : A), In (pi, pi') R -> In (pi, pi') R'.

Lemma Rsub_list_ts {A} R R' : @Rsub_list A R R' -> @Rsub A (ts_cl_list R) (ts_cl_list R').
Proof.
  unfold Rsub_list.
  unfold Rsub.
  intros.
  induction X.
  - constructor. apply H. assumption.
  - constructor 2. assumption.
  - econstructor 3.
    + apply IHX1.
    + apply IHX2.
Qed.

Lemma seq_head : forall b a, seq a (S b) = a :: seq (S a) b.
Proof.
  induction b.
  - reflexivity.
  - intros. rewrite IHb.
    + reflexivity.
Qed.

Lemma list_cons_eq {A} : forall (x : A) l1 l2, l1 = l2 <-> x::l1 = x::l2.
Proof.
  intros.
  split; intros; subst. reflexivity. inversion H. reflexivity.
Qed.

Lemma cons_app {A} : forall (a : A) l, a :: l = [a] ++ l.
Proof.
  intros.
  auto.
Qed.

Lemma forallb_existsb {A} : forall P (ls : list A), forallb P ls = false -> existsb (fun x => negb (P x)) ls = true.
Proof.
  intros.
  induction ls.
  - inversion H.
  - simpl. apply Bool.orb_true_intro.
    simpl in H. apply Bool.andb_false_iff in H as [H1 | H2].
    + left. rewrite H1. reflexivity.
    + right. apply IHls. apply H2.
Qed.
```

```
Fixpoint ntimes {A} (n : nat) (f :forall m, A) : list A :=
  match n with
  | 0 => []
  | S n => f n :: ntimes n f
  end.

Definition ntimes_proof {A} (n: nat) : (forall m, (m < n) -> A) -> list A :=
  (fix ntimes_proof_inner (n': nat) (proof : n' < S n) (f : forall m, (m < n) -> A)  {struct n'} : list A :=
    (match n' as n'' return (n' = n'') -> list A with
    | 0 => fun _ => []
    | S n'' => fun Heq => let newproof := (Lt.lt_S_n n'' n (rew [fun _ => _] Heq in proof)) in
               f n'' newproof :: ntimes_proof_inner n'' (Nat.lt_lt_succ_r _ _ newproof) f
    end) eq_refl) n (Nat.lt_succ_diag_r n).

Definition Forall_with_proof (n: nat) : (forall m, (m < n) -> Prop) -> Prop  :=
  (fix ntimes_proof_inner (n': nat) (proof : n' < S n) (f : forall m, (m < n) -> Prop)  {struct n'} : Prop :=
    (match n' as n'' return (n' = n'') -> Prop with
    | 0 => fun _ => True
    | S n'' => fun Heq => let newproof := (Lt.lt_S_n n'' n (rew [fun _ => _] Heq in proof)) in
               f n'' newproof /\ ntimes_proof_inner n'' (Nat.lt_lt_succ_r _ _ newproof) f
    end) eq_refl) n (Nat.lt_succ_diag_r n).

Ltac bsplit := apply andb_true_intro; split.

Lemma seq_skip : forall n len, seq n (S len) = n :: seq (S n) len.
Proof.
  reflexivity.
Qed.

Lemma nth_error_map_range {A} : forall n (f : nat -> A) lms, n < lms ->
    nth_error (map f (range lms)) n = Some (f n).
Proof.
  induction n.
  - intros. unfold range. destruct lms. {inversion H. } reflexivity.
  - intros. unfold range. simpl. destruct lms. {inversion H. } simpl.
    rewrite <- seq_shift. rewrite map_map. rewrite IHn.
    + reflexivity.
    + auto with arith.
Qed.

Fixpoint map_i {A B} (f : nat -> A -> B) (start : nat) (ls : list A) : list B :=
  match ls with
  | [] => []
  | (hd :: tl) => f start hd :: map_i f (S start) tl
  end.

Lemma nth_ok_proof_irel {A} : forall n (ms : list A) p1 p2, nth_ok ms n p1 =
  nth_ok ms n p2.
Proof.
```

```
  intros.
  remember (nth_ok _ _ _) as n1.
  symmetry. symmetry in Heqn1. apply nth_ok_nth_error. apply nth_ok_nth_error in Heqn1.
  rewrite <- Heqn1. reflexivity.
Qed.

Lemma fold_left_max_acc : forall ls i j, fold_left Nat.max ls i < j -> i < j.
Proof.
  induction ls; intros.
  - assumption.
  - simpl in H. apply IHls in H. apply Nat.max_lub_lt_iff in H. destruct H. assumption.
Qed.

Lemma fold_left_max_in : forall ls i j, fold_left Nat.max ls i < j -> forall k, In k ls -> k < j.
Proof.
  induction ls; intros.
  - inversion H0.
  - simpl in H0. destruct H0.
    + simpl in H. apply fold_left_max_acc in H. apply Nat.max_lub_lt_iff in H. destruct H. subst. assumption.
    + eapply IHls. simpl in H. apply H. assumption.
Qed.

Lemma in_fold_left_max : forall ls j i, (forall k, In k ls -> k < j) -> i < j -> fold_left Nat.max ls i < j.
Proof.
  induction ls; intros.
  - simpl. assumption.
  - simpl.
    destruct (dec_le i a).
    + rewrite (Nat.max_r _ _ H1).
      eapply IHls.
      * intros. apply H. constructor 2. assumption.
      * apply H. constructor. reflexivity.
    + apply not_le in H1. assert (a <= i). {
        omega.
      } rewrite (Nat.max_l _ _ H2).
      apply IHls. intros. apply H. constructor 2. assumption. assumption.
Qed.

Lemma all_some_none_exists {A} : forall (ls : list (option A)), all_some ls = None -> In None ls.
Proof.
  induction ls.
  - intros F. inversion F.
  - intros. destruct a.
    + apply all_some_none_head in H. apply IHls in H. constructor 2. assumption.
    + constructor. reflexivity.
Qed.

Lemma nth_error_Some2 {A} : forall (ms : list A) n, n < length ms -> {x & nth_error ms n = Some x}.
Proof.
```

```
  intros.
  apply nth_error_Some in H.
  destruct (nth_error ms n).
  - exists a. reflexivity.
  - exfalso. apply H. reflexivity.
Qed.

Lemma nth_error_nth_ok {A} : forall ms (x : A) n, nth_error ms n = Some x -> { lp & nth_ok ms n lp = x }.
Proof.
  intros.
  assert (nth_error ms n <> None).
  {
    intros F. rewrite H in F. discriminate F.
  }
  rewrite nth_error_Some in H0. exists H0.
  revert ms H H0.
  induction n; intros.
  - destruct ms. inversion H0. simpl in H0.
    simpl. simpl in H. apply some_eq in H. assumption.
  - destruct ms. inversion H0. simpl in H. pose proof (IHn ms H (lt_S_n _ _ H0)).
    simpl. assumption.
Qed.

Lemma In_nth_error_set {A} {eqdec : EqDec A eq} l (x : A) : In x l -> { n & nth_error l n = Some x}.
Proof.
  induction l.
  - intros. inversion H.
  - intros. rewrite app_head in H. apply In_app_sumbool in H. destruct H.
    + exists 0. simpl. inversion i. subst. reflexivity. inversion H.
    + apply IHl in i. destruct i. exists (S x0). rewrite <- e. reflexivity.
Qed.

Lemma in_map_set {A B} {eqdec : EqDec B eq} {f : A -> B} : forall (l : list A) y, In y (map f l) -> { x & f x = y /\ In x l}.
Proof.
  induction l.
  - intros. inversion H.
  - intros. rewrite app_head in H. rewrite map_app in H. apply In_app_sumbool in H. destruct H.
    + exists a. split. inversion i. assumption. inversion H. constructor. reflexivity.
    + apply IHl in i. destruct i as [x [Hf Hin]]. exists x. split. assumption. constructor 2. assumption.
Qed.

Lemma forall_length_in {A} : forall ms (Pr : A -> Prop), (forall n pr, Pr (nth_ok ms n pr)) -> (forall m, In m ms -> Pr m).
Proof.
  induction ms; intros.
  - inversion H0.
  - destruct H0.
    + subst. pose proof (H 0 (Nat.lt_0_succ _)).
      simpl in H0. assumption.
```

```
    + apply IHms.
      * intros. pose proof (H (S n) (lt_n_S _ _ pr)).
        rewrite nth_ok_skip in H1.
        erewrite nth_ok_proof_irel in H1.
        apply H1.
      * assumption.
Qed.

Lemma option_concat_head {A} : forall (m : list (option (list A))) a oms, option_concat (a :: m) = Some oms →
                                                              exists omms, option_conca m = Some omms.
Proof.
  unfold option_concat.
  simpl.
  intros.
  destruct a; try discriminate H.
  destruct (all_some m); try discriminate H.
  exists (concat l0).
  reflexivity.
Qed.

Lemma all_some_app_l {A} : forall (m1 : list (option A)) m2 ams, all_some (m1 ++ m2) = Some ams →
                                                  { amms & all_some m1 = Some amms}.
Proof.
  induction m1.
  - intros. exists []. reflexivity.
  - intros. simpl in H. destruct a eqn:Hl; try discriminate H.
    simpl. destruct (all_some (m1 ++ m2)) eqn:Hb; try discriminate H.
    apply IHm1 in Hb. destruct Hb. rewrite e. eexists. reflexivity.
Qed.

Lemma all_some_app_l_sub {A} : forall (m1 : list (option A)) m2 ams, all_some (m1 ++ m2) = Some ams →
                                                  { amms & all_some m1 = Some amms /\ forall i, In i amms → In i ams}.
Proof.
  induction m1.
  - intros. exists []. split. reflexivity. intros. inversion H0.
  - intros. simpl in H. destruct a eqn:Hl; try discriminate H.
    simpl. destruct (all_some (m1 ++ m2)) eqn:Hb; try discriminate H.
    apply IHm1 in Hb. destruct Hb as [amms [Hall Hin]]. destruct (all_some m1); try discriminate Hall. eexists. split.
    + reflexivity.
    + intros. apply some_eq in H. apply some_eq in Hall. rewrite ← H. destruct H0.
      * constructor. assumption.
      * constructor 2. apply Hin. subst. assumption.
Qed.
```

```
Lemma all_some_app {A} : forall (m1 : list (option A)) m2 ams,
    all_some (m1 ++ m2) = Some ams
    -> { ams1 & { ams2 & all_some m1 = Some ams1 /\ all_some m2 = Some ams2 /\ ams = ams1 ++ ams2}}.
Proof.
  induction m1.
  - intros. exists []. exists ams. split.
    + reflexivity.
    + split.
      * assumption.
      * reflexivity.
  - intros. rewrite <- app_comm_cons in H. destruct a; try (simpl in H; discriminate H). simpl in H.
    destruct (all_some (m1 ++ m2)) eqn:Hamm; try (simpl in H; discriminate H).
    apply IHm1 in Hamm. destruct Hamm as [ams1 [ams2 [H1 [H2 H3]]]].
    eexists. eexists.
    split.
    + simpl. rewrite H1. reflexivity.
    + split.
      * apply H2.
      * apply some_eq in H. subst. reflexivity.
Qed.



Lemma option_concat_app_l {A} : forall (m1 : list (option (list A))) m2 oms, option_concat (m1 ++ m2) = Some oms ->
                                                { omms & option_concat m1 = Some omms}.
Proof.
  unfold option_concat.
  intros.
  destruct (all_some (m1 ++ m2)) eqn:Hb; try discriminate H.
  apply all_some_app_l in Hb. destruct Hb. rewrite e. eexists. reflexivity.
Qed.

Lemma in_l_in_concat {A} : forall (x : list A) l, In x l -> (forall i, In i x -> In i (concat l)).
Proof.
  induction l.
  - intros F. inversion F.
  - intros. simpl in *. destruct H.
    + apply in_or_app. left. subst. assumption.
    + apply in_or_app. right. apply IHl.
      * assumption.
      * assumption.
Qed.

Lemma concat_in {A} : forall l1 l2 (x :A), (forall i, In i l1 -> In i l2) -> In x (concat l1) -> In x (concat l2).
Proof.
  induction l1.
  - intros. inversion H0.
  - intros.
    simpl in H0.
    apply in_app_or in H0.
```

```
    destruct H0.
    + eapply in_l_in_concat in H0.
      * apply H0.
      * apply H. constructor. reflexivity.
    + apply IHl1.
      intros. apply H. constructor 2. assumption. assumption.
Qed.

Lemma option_concat_app_l_sub {A} : forall (m1 : list (option (list A))) m2 oms,
 ↪  option_concat (m1 ++ m2) = Some oms → { omms & option_concat m1 = Some omms
 ↪  /\ forall x, In x omms → In x oms}.
Proof.
  unfold option_concat.
  intros.
  destruct (all_some (m1 ++ m2)) eqn:Hb; try discriminate H.
  apply all_some_app_l_sub in Hb. destruct Hb as [amms [Hall Hin]].
  rewrite Hall. exists (concat amms). split. reflexivity. intros.
  apply some_eq in H. subst.
  eapply concat_in.
  - intros. apply Hin. apply H.
  - assumption.
Qed.

Lemma option_concat_app {A} : forall (m1 : list (option (list A))) m2 oms,
 ↪  option_concat (m1 ++ m2) = Some oms → { oms1 & {oms2 & option_concat m1 =
 ↪  Some oms1 /\  option_concat m2 = Some oms2 /\ oms = oms1 ++ oms2}}.
Proof.
  unfold option_concat.
  intros.
  destruct (all_some (m1 ++ m2)) eqn:Hb; try discriminate H.
  apply all_some_app in Hb. destruct Hb as [ams1 [ams2 [H1 [H2 Heq]]]].
  rewrite H1. rewrite H2. rewrite ← some_eq in H. rewrite Heq in H.
  rewrite concat_app in H. exists (concat ams1). exists (concat ams2).
   ↪  firstorder.
Qed.

Lemma ts_cl_list_sub {A} : forall (R: list (A * A)) R', Rsub_list R R' → Rsub
 ↪  (ts_cl_list R) (ts_cl_list R').
Proof.
  unfold Rsub.
  intros.
  induction X.
  - constructor. apply H. assumption.
  - constructor 2. assumption.
  - econstructor 3. apply IHX1.  assumption.
Qed.

Lemma Rsub_trans {t} : forall (A B C : t → t → Type) , Rsub A B → Rsub B C →
 ↪  Rsub A C.
Proof.
  firstorder.
Qed.

Lemma Rsublist_app {A} : forall ls1 ls2 (ms : list (A * A)), Rsub_list ls1 ms →
 ↪  Rsub_list ls2 ms → Rsub_list (ls1 ++ ls2) ms.
Proof.
```

```
  unfold Rsub_list.
  intros.
  apply in_app_or in H1.
  destruct H1.
  + apply H. assumption.
  + apply H0. assumption.
Qed.

Lemma Rsub_in_app {A:Type} {eqdec : EqDec A eq} : forall (R : A → A → Type) oms1 oms2 , (Rsub (fun p p' ⇒ In (p, p') oms1) R) → Rsub (fun p p' ⇒ In (p, p') oms2) R → Rsub (fun p p' ⇒ In (p, p') (oms1 ++ oms2)) R.
Proof.
  unfold Rsub. intros.
  apply In_app_sumbool in H.
  destruct H.
  + apply X. assumption.
  + apply X0. assumption.
Qed.

Lemma Rsub_in_concat {A:Type} {eqdec : EqDec A eq} : forall (R : A → A → Type) l ,
    (forall m, In m l →  (Rsub (fun p p' ⇒ In (p, p') m) R))  → Rsub (fun p p' ⇒ In (p, p') (concat l)) R.
Proof.
  induction l.
  - intros. unfold Rsub. intros. inversion H.
  - intros. simpl. apply Rsub_in_app.
    + apply X. constructor. reflexivity.
    + apply IHl. intros. apply X. constructor 2. assumption.
Qed.

Lemma all_some_map2 {A B} {eqdec : EqDec A eq} : forall ms (f: A → option B) l, all_some (map f ms) = Some l → forall m, In m ms → {x & f m = Some x}.
Proof.
  induction ms.
  - intros. inversion H0.
  - intros. simpl in H. destruct (f a) eqn:Hfa; try discriminate H.
    destruct (all_some _) eqn:Hall; try discriminate H. apply In_head_set in H0. destruct H0.
    + subst. exists b. apply Hfa.
    + eapply IHms.
      * apply Hall.
      * assumption.
Qed.

Lemma all_some_map {A B} {eqdec : EqDec B eq} : forall ms (f : A → option B) l,
    all_some (map f ms) = Some l → (forall m, In m l → {n & In n ms /\ f n = Some m}).
Proof.
  induction ms.
  - intros. inversion H. subst. exfalso. inversion H0.
  - intros. simpl in *. destruct (f a) eqn:Hfa; try discriminate H.
    destruct (all_some _) eqn:Hall; try discriminate H. apply some_eq in H.
    pose proof (IHms f _ Hall m).
    rewrite ← H in H0. rewrite app_head in H0. apply In_app_sumbool in H0. destruct H0.
```

```
    + assert (m = b). subst. inversion i. subst. reflexivity. inversion H. subst. exists a. split.
      * left. reflexivity.
      * assumption.
    + apply X in i. destruct i. destruct a0. exists x.
      split.
      * right. assumption.
      * assumption.
Qed.

Lemma in_combine_range {A} : forall (ls : list A) a n start pr, (In (a, start + n) (combine ls (seq start (length ls)))) -> nth_ok ls n pr = a.
Proof.
  intros. rewrite nth_ok_nth_error. apply In_nth_error in H. destruct H as [n0 H]. apply nth_error_combine in H. destruct H.
  assert (nth_error ls n0 <> None). { intros F. rewrite H in F. discriminate F. }
  apply nth_error_Some in H1.
  pose proof (seq_nth_error (length ls) start n0 H1). rewrite H0 in H2. apply some_eq in H2. assert (n = n0). omega.
  subst. assumption.
Qed.


Inductive Exists_T {A : Type} (P : A -> Type) : list A -> Type :=
| Exists_T_cons_hd : forall (x : A) (l : list A), P x -> Exists_T P (x :: l)
| Exists_T_cons_tl : forall (x : A) (l : list A), Exists_T P l -> Exists_T P (x :: l).

Lemma nth_error_Some3 {A} : forall ms n (a :A), nth_error ms n = Some a -> n < length ms.
Proof.
  intros.
  apply nth_error_Some.
  intros F. rewrite H in F. discriminate F.
Qed.

Lemma all_some_length {A} : forall ls (ls' : list A), all_some ls = Some ls' -> length ls = length ls'.
Proof.
  induction ls.
  - intros. inversion H. reflexivity.
  - intros. destruct a; try discriminate H. simpl in *. destruct (all_some ls) eqn:Hall; try discriminate H.
    destruct ls'; try discriminate H. apply some_eq in H. rewrite <- H. simpl. apply eq_S.
    apply IHls. reflexivity.
Qed.

Lemma map_in {A B}: forall ls (f: A -> B) x,  In x ls -> In (f x) (map f ls).
Proof.
  induction ls.
  - intros. inversion H.
  - intros. destruct H.
    + simpl. subst. left. reflexivity.
    + simpl. right. apply IHls. assumption.
```

```
Qed.

Lemma nth_ok_in {A} : forall (ls : list A) x pr, In (nth_ok ls x pr) ls.
Proof.
  intros.
  remember (nth_ok ls x pr). symmetry in Heqa. apply nth_ok_nth_error in Heqa. eapply nth_error_In. apply Heqa.
Qed.

Lemma all_some_some_head {A} : forall a (a0: A) ls ls0,
    all_some (a :: ls) = Some (a0 :: ls0) -> a = Some a0 /\ all_some ls = Some ls0.
Proof.
  intros.
  simpl in H. destruct a eqn:Ha; try discriminate H. destruct (all_some ls) eqn:Hall; try discriminate H.
  apply some_eq in H. inversion H. split; reflexivity.
Qed.

Lemma all_some_nth {A} : forall (ls : list (option A)) ls' (Hall : all_some ls = Some ls') x (pr : x < (length ls)),
    nth_ok ls x pr = Some (nth_ok ls' x (rew (all_some_length _ _ Hall) in pr)).
Proof.
  induction ls.
  - intros. inversion pr.
  - intros. destruct ls'.
    + pose proof (all_some_length _ _ Hall). inversion H.
    + pose proof (all_some_some_head _ _ _ _ Hall) as [Ha Hall2]. destruct x.
      * simpl. assumption.
      * simpl.
        erewrite nth_ok_proof_irel.
        erewrite (nth_ok_proof_irel _ ls').
        apply IHls. Unshelve.
        ** firstorder.
        ** assumption.
Qed.

Lemma repeat_rev {A} : forall (a :A) n, repeat a (S n) = repeat a n ++ [a].
Proof.
  induction n.
  - reflexivity.
  - simpl. rewrite <- IHn. reflexivity.
Qed.
```