SfC.v

```
Require Import Coq.Arith.PeanoNat.
Require Import Coq.Lists.List.
Require Import Coq.Bool.Sumbool.
Require Import Coq.Classes.EquivDec.
Require Import Autosubst.Autosubst.
Require Import Datatypes.
Require Import Omega.

Require Import PrincInh.Types.
Require Import PrincInh.NFTerms.
Require Import PrincInh.LongTyping.
Require Import PrincInh.Utils.
Require Import PrincInh.Paths.

Import ListNotations.
Import EqNotations.

Open Scope bool_scope.

Inductive SfC (Delta : list path) (R: path → path → Type) : nfterm → path → Type :=
| SfC_I s pi : SfC ((pi ++ [Src]) :: Delta) R s (pi ++ [Tgt]) →
               SfC Delta R (\__ s) pi
| SfC_E ms pi pi' x : nth_error Delta x = Some pi → R (pi ++ repeat Tgt (length ms)) pi' →
                  (forall n (p: n < length ms), SfC Delta R (nth_ok ms n p) (make_tgt_path pi n) )
                  → SfC Delta R ( !! x @@ ms) pi'.

Fixpoint proof_length_SfC {Delta R m pi} (proof : SfC Delta R m pi) : nat :=
  match proof with
  | SfC_I _ _ _ _ proof' ⇒ S (proof_length_SfC proof')
  | SfC_E _ _ ms pi pi' x Deltaok Rok proofs ⇒
    (fix dummy n (pr : n ≤ length ms) : nat :=
       match n with
       | 0 ⇒ fun _ ⇒ 0
       | S n' ⇒ fun pr ⇒ S (Nat.max (proof_length_SfC (proofs n' (proj1 (Nat.le_succ_l n' (length ms)) pr)))
                                   (dummy n' (le_Sn_le _ _ pr)))
       end pr) (length ms) (Nat.le_refl _)
  end.

Lemma proof_length_zero : forall Delta R m pi (proof : SfC Delta R m pi), 0 = proof_length_SfC proof →
                                                                   {x & m = !! x @@ []}.
Proof.
  intros.
  destruct proof.
  - discriminate H.
  - destruct ms.
    + exists x. reflexivity.
    + discriminate H.
```

```
Qed.

Definition get_subproof_app_pi
           {Delta R x ms pi'} (proof: SfC Delta R ( !! x @@ ms) pi') : path.
  destruct proof; exact pi.
Defined.

Definition get_subproof_app_deltaok
           {Delta R x ms pi'} (proof: SfC Delta R ( !! x @@ ms) pi') : nth_error Delta x = Some (get_subproof_app_pi proof).
  unfold get_subproof_app_pi.
  revert proof. remember ( !! x @@ ms) as m.
  intro proof.
  revert Heqm.
  destruct proof.
  - intros. discriminate Heqm.
  - intros. injection Heqm. intros Hx _. rewrite ← Hx. exact e.
Defined.

Definition get_subproof_app_Rok
           {Delta R x ms pi'} (proof: SfC Delta R ( !! x @@ ms) pi') : R ((get_subproof_app_pi proof) ++ repeat Tgt (length ms)) pi'.
  unfold get_subproof_app_pi.
  revert proof. remember ( !! x @@ ms) as m.
  intro proof.
  revert Heqm.
  destruct proof.
  - intros. discriminate Heqm.
  - intros. injection Heqm. intros _ Hms. rewrite ← Hms. exact r.
Defined.

Definition get_subproof_app
           {Delta R x ms pi'} (proof: SfC Delta R ( !! x @@ ms) pi') :
  (forall n (p: n < length ms), SfC Delta R (nth_ok ms n p)
                                (make_tgt_path (get_subproof_app_pi proof) n) ).
  unfold get_subproof_app_pi.
  revert proof. remember ( !! x @@ ms) as m.
  intro proof.
  revert Heqm.
  destruct proof.
  - intros. discriminate Heqm.
  - intro. injection Heqm. intros _ Hms. rewrite ← Hms. exact s.
Defined.

Lemma SfC_gen_app : forall R x ms pi' Delta (proof : SfC Delta R ( !! x @@ ms) pi'),
    SfC_E Delta R ms
                 (get_subproof_app_pi proof) pi' x
                 (get_subproof_app_deltaok proof)
                 (get_subproof_app_Rok proof)
                 (get_subproof_app proof) = proof.
Proof.
  intros.
  unfold get_subproof_app_pi.
  unfold get_subproof_app_deltaok.
  unfold get_subproof_app_Rok.
```

```
unfold get_subproof_app.
generalize (eq_refl ( !! x @@ ms)).
revert proof.
set (P := fun m' ⇒ forall (proof : SfC Delta R m' pi') (e : m' = !! x @@ ms),
SfC_E Delta R ms match proof with
                 | SfC_I _ _ _ pi _ | SfC_E _ _ _ pi _ _ _ _ _ ⇒ pi
                 end pi' x
  (match
     proof as s in (SfC _ _ n p)
     return
       (n = !! x @@ ms →
        nth_error Delta x = Some match s with
                                 | SfC_I _ _ _ pi _ | SfC_E _ _ _ pi _ _ _ _ _
                                   ↪ ⇒ pi
                                 end)
   with
   | SfC_I _ _ s pi _ ⇒
       fun Heqm : \__ s = !! x @@ ms ⇒
       False_ind (nth_error Delta x = Some pi)
         (eq_ind (\__ s) (fun e0 : nfterm ⇒ match e0 with
                                             | !! _ @@ _ ⇒ False
                                             | \__ _ ⇒ True
                                             end) I ( !! x @@ ms) Heqm)
   | SfC_E _ _ ms0 pi _ x0 e0 _ _ ⇒
       fun Heqm : !! x0 @@ ms0 = !! x @@ ms ⇒
       eq_ind x0 (fun x1 : var ⇒ nth_error Delta x1 = Some pi) e0 x
         (f_equal (fun e1 : nfterm ⇒ match e1 with
                                      | !! x1 @@ _ ⇒ x1
                                      | \__ _ ⇒ x0
                                      end) Heqm)
   end e)
  (match
     proof as s in (SfC _ _ n p)
     return
       (n = !! x @@ ms →
        R (match s with
           | SfC_I _ _ _ pi _ | SfC_E _ _ _ pi _ _ _ _ _ ⇒ pi
           end ++ repeat Tgt (length ms)) p)
   with
   | SfC_I _ _ s pi _ ⇒
       fun Heqm : \__ s = !! x @@ ms ⇒
       False_rect (R (pi ++ repeat Tgt (length ms)) pi)
         (eq_ind (\__ s) (fun e0 : nfterm ⇒ match e0 with
                                             | !! _ @@ _ ⇒ False
                                             | \__ _ ⇒ True
                                             end) I ( !! x @@ ms) Heqm)
   | SfC_E _ _ ms0 pi pi'0 x0 _ r _ ⇒
       fun Heqm : !! x0 @@ ms0 = !! x @@ ms ⇒
       rew [fun ms1 : list nfterm ⇒ R (pi ++ repeat Tgt (length ms1)) pi'0]
           f_equal (fun e1 : nfterm ⇒ match e1 with
                                       | !! _ @@ ms1 ⇒ ms1
                                       | \__ _ ⇒ ms0
                                       end) Heqm in r
   end e)
  (match
```

```
        proof as s in (SfC _ _ n p)
        return
          (n = !! x @@ ms →
           forall (n0 : nat) (p0 : n0 < length ms),
           SfC Delta R (nth_ok ms n0 p0)
             (make_tgt_path match s with
                            | SfC_I _ _ _ pi _ | SfC_E _ _ _ pi _ _ _ _ _ ⇒ pi
                            end n0))
      with
      | SfC_I _ _ s pi _ ⇒
          fun (Heqm : \__ s = !! x @@ ms) (n : nat) (p : n < length ms) ⇒
          False_rect (SfC Delta R (nth_ok ms n p) (make_tgt_path pi n))
            (eq_ind (\__ s) (fun e0 : nfterm ⇒ match e0 with
                                                | !! _ @@ _ ⇒ False
                                                | \__ _ ⇒ True
                                                end) I ( !! x @@ ms) Heqm)
      | SfC_E _ _ ms0 pi _ x0 _ _ s ⇒
          fun Heqm : !! x0 @@ ms0 = !! x @@ ms ⇒
          rew [fun ms1 : list nfterm ⇒
               forall (n : nat) (p : n < length ms1), SfC Delta R (nth_ok ms1 n
            ↪  p) (make_tgt_path pi n)]
            f_equal (fun e1 : nfterm ⇒ match e1 with
                                        | !! _ @@ ms1 ⇒ ms1
                                        | \__ _ ⇒ ms0
                                        end) Heqm in s
      end e) = rew [fun _ ⇒ _] e in proof).
   assert (forall m', P m').
   {
     unfold P. intros m' proof' Heq.
     destruct proof'.
     - discriminate Heq.
     - revert e r s.
       injection Heq. intros Hx Hms.
       revert Heq.
       rewrite Hx. rewrite Hms.
       intros Heq.
       assert (Heq = eq_refl).
       {
         apply Coq.Logic.Eqdep_dec.UIP_dec.
         apply eqdec_nfterm.
       }
       rewrite H. simpl.
       reflexivity.
   }
   intros proof Heqm.
   unfold P in H. generalize (H ( !! x @@ ms) proof Heqm).
   clear ...
   match goal with
   | [⊢ ?lhs = _ → ?lhs' = _] ⇒ assert (lhs = lhs'); [apply f_equal;
    ↪ reflexivity ]]
   end.
   rewrite H. intros. rewrite H0.
   rewrite (Coq.Logic.Eqdep_dec.UIP_dec eqdec_nfterm Heqm eq_refl). reflexivity.
 Qed.
```

```
Definition get_subproof_lam {Delta R m pi}
           (proof: SfC Delta R (\__m) pi) : SfC ((pi ++ [Src]) :: Delta) R m (pi ++ [Tgt]).
inversion proof. assumption.
Defined.

Lemma SfC_gen_lam R m pi Delta (proof : SfC Delta R (\__ m) pi) :
  SfC_I Delta R m pi (get_subproof_lam proof) = proof .
Proof.
  unfold get_subproof_lam.
  match goal with
  | [ ⊢ SfC_I _ _ _ _ (_ ?eq1 ?eq2) = _] ⇒ generalize eq1
  end.
  revert proof.
  set (P := fun m' : nfterm ⇒ (
  forall (proof : SfC Delta R (m') pi) (e : m' = \__ m),
  SfC_I Delta R m pi
    (match
       proof in (SfC _ _ n p) return (n = \__ m → p = pi → SfC ((pi ++ [Src]) :: Delta) R m (pi ++ [Tgt]))
     with
     | SfC_I _ _ s pi0 X ⇒
         fun (H : \__ s = \__ m) (H0 : pi0 = pi) ⇒
         (rew ← [fun n : nfterm ⇒
                   pi0 = pi →
                   SfC ((pi0 ++ [Src]) :: Delta) R n (pi0 ++ [Tgt]) →
                   SfC ((pi ++ [Src]) :: Delta) R m (pi ++ [Tgt])]
              f_equal (fun e0 : nfterm ⇒ match e0 with
                                         | !! _ @@ _ ⇒ s
                                         | \__ s0 ⇒ s0
                                         end) H in
           (fun H1 : pi0 = pi ⇒
            rew ← [fun l : list dir ⇒
                     SfC ((l ++ [Src]) :: Delta) R m (l ++ [Tgt]) →
                     SfC ((pi ++ [Src]) :: Delta) R m (pi ++ [Tgt])] H1 in
            (fun X0 : SfC ((pi ++ [Src]) :: Delta) R m (pi ++ [Tgt]) ⇒ X0))) H0 X
     | SfC_E _ _ ms pi0 pi' x H X X0 ⇒
         fun (H0 : !! x @@ ms = \__ m) (H1 : pi' = pi) ⇒
         False_rect
           (pi' = pi →
            nth_error Delta x = Some pi0 →
            R (pi0 ++ repeat Tgt (length ms)) pi' →
            (forall (n : nat) (p : n < length ms), SfC Delta R (nth_ok ms n p) (make_tgt_path pi0 n)) →
            SfC ((pi ++ [Src]) :: Delta) R m (pi ++ [Tgt]))
           (eq_ind ( !! x @@ ms) (fun e0 : nfterm ⇒ match e0 with
                                                    | !! _ @@ _ ⇒ True
                                                    | \__ _ ⇒ False
                                                    end) I (\__ m) H0) H1 H X X0
     end e eq_refl) = rew [fun _ ⇒ _] e in proof
              )).
  assert (forall m',  P m').
  {
```

```
    unfold P. intros m' proof' Heq.
    destruct proof'.
    - revert proof'.
      remember (f_equal (fun e0 : nfterm => match e0 with
                                            | !! _ @@ _ => s
                                            | \__ s0 => s0
                                            end) Heq) as Hfeq.
      assert (m = s).
      { ainv. }
      revert Heq Hfeq HeqHfeq.
      rewrite H.
      intros Heq.
      assert (Heq = eq_refl).
      {
        apply Coq.Logic.Eqdep_dec.UIP_dec.
        apply eqdec_nfterm.
      }
      rewrite H0. simpl.
      intros Hfeq Hfeq_eqrefl.
      rewrite Hfeq_eqrefl.
      unfold eq_rect_r. simpl. auto.
    - ainv.
  }
  intros proof Heqm.
  unfold P in H. generalize (H (\__ m) proof Heqm).
  clear ...
  match goal with
  | [|- ?lhs = _ -> ?lhs' = _] => assert (lhs = lhs'); [apply f_equal; reflexivity]
  end.
  rewrite H. intros. rewrite H0.
  rewrite (Coq.Logic.Eqdep_dec.UIP_dec eqdec_nfterm Heqm eq_refl). reflexivity.
Qed.


Inductive SfC_prev R : forall Delta Delta' m m' pi pi', SfC Delta R m pi -> SfC Delta' R m' pi' -> Type :=
| SfC_prev_I : forall Delta m pi (proof: SfC ((pi ++ [Src]) :: Delta) R m (pi ++ [Tgt])),
    SfC_prev _ _ _ _ _ _ _ _ proof (SfC_I _ _ _ _ proof)
| SfC_prev_E : forall Delta pi pi' ms x
    (deltaok: nth_error Delta x = Some pi)
      (Rproof: R (pi ++ repeat Tgt (length ms)) pi')
      (proofs: (forall n (p: n < length ms), SfC Delta R (nth_ok ms n p) (make_tgt_path pi n) ))
                n (ltproof: (n < length ms)),
              SfC_prev _ _ _ _ _ _ _ _ (proofs n ltproof) (SfC_E _ _ _ _ _ _ deltaok Rproof proofs)
.

Inductive SfC_subj R : forall Delta Delta' m m' pi pi', SfC Delta R m pi -> SfC Delta' R m' pi' -> Type :=
| SfC_subj_refl : forall Delta m pi (proof: SfC Delta R m pi), SfC_subj _ _ _ _ _ _ proof proof
```

```
| SfC_subj_trans : forall Delta Delta' Delta'' m m' m'' pi pi' pi'' (proof1 : SfC Delta R m pi) (proof2: SfC Delta' R m' pi') (proof3: SfC Delta'' R m'' pi''),
    SfC_subj _ _ _ _ _ _ _ _ proof1 proof2 ->
    SfC_subj _ _ _ _ _ _ _ _ proof2 proof3 ->
    SfC_subj _ _ _ _ _ _ _ _ proof1 proof3
| SfC_subj_I : forall Delta m pi (proof: SfC ((pi ++ [Src]) :: Delta) R m (pi ++ [Tgt])),
    SfC_subj _ _ _ _ _ _ _ _ proof (SfC_I _ _ _ _ proof)
| SfC_subj_E : forall Delta pi pi' ms x
    (deltaok: nth_error Delta x = Some pi)
      (Rproof: R (pi ++ repeat Tgt (length ms)) pi')
      (proofs: (forall n (p: n < length ms), SfC Delta R (nth_ok ms n p) (make_tgt_path pi n) ))
                n (ltproof: (n < length ms)),
              SfC_subj _ _ _ _ _ _ _ _ (proofs n ltproof) (SfC_E _ _ _ _ _ _ deltaok Rproof proofs)
.


Lemma sfc_monotone_aux : forall m R R' pi Delta, Rsub R R' ->
                                          SfC Delta R m pi -> SfC Delta R' m pi.
Proof.
  induction 2.
  - constructor. apply IHX0. assumption.
  - econstructor.
    + apply e.
    + auto.
    + ainv.
      apply X0.
      assumption.
Qed.

Lemma sfc_monotone : forall m R R', Rsub R R' -> SfC [] R m [] -> SfC [] R' m [].
Proof.
  intros.
  eapply sfc_monotone_aux.
  - apply X.
  - assumption.
Qed.

Lemma sfc_monotone_aux_list : forall m R R' pi Delta, Rsub_list R R' ->
                                               SfC Delta (ts_cl_list R) m pi ->
                                               SfC Delta (ts_cl_list R') m pi.
Proof.
  intros.
  pose proof (Rsub_list_ts R R' H).
  eapply sfc_monotone_aux.
  - apply X0.
  - apply X.
Qed.

Definition evenodd_cond {Delta R m pi} (proof : SfC Delta R m pi) :=
  (even_ones pi) /\ (odd_repo (Delta)).
```

```
Fixpoint each_judg_SfC {Delta R m pi} (P : forall Delta' R' m' pi', SfC Delta' R' m' pi' → Prop) (proof : SfC Delta R m pi) : Prop :=
  P Delta R m pi proof /\
  match proof with
  | SfC_I _ _ s' pi' proof' ⇒ each_judg_SfC (P) proof'
  | SfC_E _ _ ms pi pi' x DeltaProof Rproof proof' ⇒
    forall (n : nat) (p : n < length ms),
      each_judg_SfC (P) (proof' n p)
  end.

Lemma each_judg_subj_SfC : forall Delta R t pi (m : SfC Delta R t pi) Pr,
    each_judg_SfC Pr m →
    forall Delta' t' pi' (n : SfC Delta' R t' pi'), SfC_subj _ _ _ _ _ _ _ n m →
                                                   (each_judg_SfC Pr n).
Proof.
  intros.
  generalize dependent H.
  generalize dependent Pr.
  induction X; intros.
  - assumption.
  - apply IHX1. apply IHX2. assumption.
  - destruct H. assumption.
  - destruct H. apply H0.
Qed.

Lemma each_judg_subj_SfC_P : forall Delta R t pi (m : SfC Delta R t pi) P,
    each_judg_SfC P m →
    forall Delta' t' pi' (n : SfC Delta' R t' pi'), SfC_subj _ _ _ _ _ _ _ n m →
                                                   P _ _ _ _ n.
Proof.
  intros.
  pose proof (each_judg_subj_SfC _ _ _ _ _ _ H _ _ _ _ X).
  destruct n; destruct H0; assumption.
Qed.

Fixpoint any_judg_SfC {Delta R m pi} (P : forall Delta' R' m' pi', SfC Delta' R' m' pi' → Prop) (proof : SfC Delta R m pi) : Prop :=
  P Delta R m pi proof \/
  match proof with
  | SfC_I _ _ s' pi' proof' ⇒ any_judg_SfC (P) proof'
  | SfC_E _ _ ms pi pi' x DeltaProof Rproof proof' ⇒
    exists (n : nat) (p : n < length ms),
      any_judg_SfC (P) (proof' n p)
  end.

Lemma evenodd_aux {Delta R m pi} (proof : SfC Delta R m pi) : evenodd_cond proof → each_judg_SfC (@evenodd_cond) proof .
Proof.
  induction proof.
  - intros. unfold each_judg_SfC. split. { assumption. }
    apply IHproof.
    unfold evenodd_cond in *. destruct H as [Hev Hodd].
    split.
```

```
    + rewrite ← even_ones_pump. assumption.
    + rewrite odd_repo_head_tail. unfold odd_repo.
      constructor.
      * simpl. apply Nat.Odd_succ. exact Hev.
      * assumption.
  - split. {assumption. }
    intros. apply H.
    destruct H0 as [Hev Hodd].
    split.
    + revert Hev e Hodd. clear ... intros.
      eapply tgt_path_even_if_delta_odd.
      * apply Hodd.
      * eapply nth_error_In. apply e.
    + assumption.
Qed.

Lemma evenodd {R m} (proof : SfC [] R m []) : each_judg_SfC (@evenodd_cond) proof .
Proof.
 apply evenodd_aux.
 unfold evenodd_cond.
 split.
 - unfold even_ones. simpl. exists 0. auto.
 - unfold odd_repo. constructor.
Qed.

Definition r_not_refl_cond {Delta R m pi} (proof : SfC Delta R m pi)  :=
  match proof with
  | SfC_I _ _ _ _ _ ⇒ True
  | SfC_E _ _ ms pi pi' _ _ _ _ ⇒ ~(pi ++ repeat Tgt (length ms) = pi')
  end.

Lemma evenodd_2_r_not_refl {Delta R m pi} (proof : SfC Delta R m pi) : evenodd_cond proof → r_not_refl_cond proof.
Proof.
  destruct proof.
  - simpl. auto.
  - unfold r_not_refl_cond. unfold evenodd_cond.
    intros. destruct H as [Hev Hodd].
    unfold odd_repo in Hodd. rewrite Forall_forall in Hodd.
    pose proof (Hodd pi). apply nth_error_In in e. apply H in e.
    assert (~even_ones(pi ++ repeat Tgt (length ms))).
    {
      unfold even_ones. intros F. eapply Nat.Even_Odd_False. exact F.
      rewrite count_occ_split. apply Odd_plus_Even_is_Odd.
      - assumption.
      - clear ... remember (length ms) as lms. clear ... induction lms.
        + simpl. unfold Nat.Even. exists 0. auto.
        + simpl. assumption.
    }
    unfold even_ones in H0. intros F. subst.
    contradiction.
Qed.

Lemma each_judg_impl :forall (P : forall Delta R m pi, SfC Delta R m pi → Prop)
                             (Q : forall Delta R m pi, SfC Delta R m pi → Prop)
```

```
  , (forall Delta R m pi (proof : SfC Delta R m pi), P _ _ _ _ proof → Q _ _ _
   ↪ _ proof)
    → forall Delta R m pi (proof : SfC Delta R m pi), each_judg_SfC (P) proof
     ↪ → each_judg_SfC Q proof.
Proof.
  induction proof.
  - simpl. intros [Hl Hr]. split.
    + apply H. assumption.
    + apply IHproof. assumption.
  - simpl. intros [Hl Hr]. split.
    + apply H. assumption.
    + intros. apply H0. apply Hr.
Qed.

Lemma r_not_refl {R m} (proof : SfC [] R m []) : each_judg_SfC
 ↪ (@r_not_refl_cond) proof.
Proof.
  apply each_judg_impl with @evenodd_cond.
  intros.
  apply evenodd_2_r_not_refl. assumption.
  apply evenodd.
Qed.

Definition is_minimal_R (m : nfterm) (R : path → path → Type) :=
  forall R', SfC [] R' m [] → Rsub R' R.

Lemma SfC_Delta_x : forall Delta x ms pi' R, SfC Delta R (!! x @@ ms) pi' →
 ↪ nth_error Delta x <> None.
Proof.
  intros.
  inv X.
  intros F. rewrite H2 in F. inv F.
Qed.

Fixpoint R_m_aux (Delta: list path) (pi': path) (m : nfterm) {struct m} : option
 ↪ (list (path * path)) :=
  match m with
  | \__ s ⇒ R_m_aux ((pi' ++ [Src]) :: Delta) (pi' ++ [Tgt]) s
  | !! x @@ ms ⇒ match nth_error Delta x with
                 | None ⇒ None
                 | Some pi ⇒ option_concat
                              (((fix combine_with ms ns :=
                                   match ms with
                                   | [] ⇒ []
                                   | x :: xs ⇒
                                     match ns with
                                     | n :: ns ⇒ (R_m_aux Delta (make_tgt_path
                                      ↪ pi n) x) :: combine_with xs ns
                                     | [] ⇒ []
                                     end
                                   end) ms (range( length ms))) ++ [Some [((pi
                                    ↪ ++ repeat Tgt (length ms), pi'))]])
                 end
  end.
```

```
Definition R_m m := R_m_aux [] [] m.
Hint Unfold R_m R_m_aux.
Hint Immediate app_nil_l app_nil_r.

Definition R_m_ts m := match R_m m with
                       | None => fun pi pi2 => False
                       | Some Rmm => ts_cl_list (Rmm)
                       end.

Lemma combine_with_Rstuff : forall pi Delta, (fix combine_with (ms : list nfterm) (ns : list nat) {struct ms} :
                   list (option (list (path * path))) :=
                   match ms with
                   | [] => []
                   | x :: xs =>
                      match ns with
                      | [] => []
                      | n :: ns0 =>
                         R_m_aux Delta (make_tgt_path pi n) x :: combine_with xs ns0
                      end
                   end)
                 = combine_with (fun x n => R_m_aux Delta (make_tgt_path pi n) x).
Proof.
  reflexivity.
Qed.

Lemma Rm_nth_ok : forall n ms Delta pi p1 p2, nth_ok
                        (combine_with (fun x n => R_m_aux Delta (make_tgt_path pi n) x) ms (range (length ms))) n p1 = R_m_aux Delta (make_tgt_path pi n) (nth_ok ms n p2).
Proof.
  intros.
  rewrite nth_ok_nth_error.
  rewrite combine_with_map .
  rewrite nth_error_map.
  pose proof (combine_length ms (range (length ms))).
  unfold range in H at 2. rewrite seq_length in H. rewrite Nat.min_id in H. pose proof p2. rewrite <- H in H0.
  destruct (nth_error (combine ms (range (length ms)))) eqn:Ha.
  + destruct p.
    apply nth_error_combine in Ha.
    destruct Ha.
    unfold range in H2.
    rewrite seq_nth_error in H2; try apply p2. asimpl in *. apply some_eq in H2. subst.
    rewrite <- (nth_ok_nth_error _ _ p2) in H1. subst. reflexivity.
  + pose proof p2.
    rewrite <- H in H0.
    rewrite <- nth_error_Some in H0. rewrite H in H0. contradiction.
Qed.

Lemma Rm_in_list : forall n ms p0 (p : n < length ms) Delta,
    In (R_m_aux Delta (p0 ++ repeat Tgt n ++ [Src]) (nth_ok ms n p))
```

```
      (combine_with (fun x n => R_m_aux Delta (p0 ++ repeat Tgt n ++ [Src]) x) ms (range (length ms))).
Proof.
  intros.
  pose proof (Rm_nth_ok n ms Delta p0 (lt_comb _ p) p).
  rewrite nth_ok_nth_error in H.
  eapply nth_error_In.
  apply H.
Qed.

Lemma R_m_ts_correct : forall m Rm Delta pi, R_m_aux Delta pi m = Some Rm  -> SfC Delta (ts_cl_list Rm) m pi.
Proof.
  induction m using nfterm_rect'.
  - intros. simpl in H.
    destruct (nth_error Delta x) eqn:Hx; try discriminate H.
    rewrite (combine_with_Rstuff p0 Delta) in H.
    unfold option_concat in H.
    destruct (
        all_some
          (combine_with (fun (x : nfterm) (n : nat) => R_m_aux Delta (make_tgt_path p0 n) x) ms
            (range (length ms)) ++ [Some [(p0 ++ repeat Tgt (length ms), pi)]])
      ) eqn:Hallsome; try discriminate H.
    apply some_eq in H.
    econstructor.
    + apply Hx.
    + constructor.
       apply (all_some_some _ _ (Some [(p0 ++ repeat Tgt (length ms), pi)])) in Hallsome.
        ** destruct Hallsome as [y [Hsomey Hinyl]].
           apply some_eq in Hsomey.
           subst. apply in_concat. assumption.
        ** apply in_or_app. right. constructor. reflexivity.
    + intros.
      remember (R_m_aux Delta (p0 ++ repeat Tgt n ++ [Src]) (nth_ok ms n p1)) as Rmm.
      pose proof (Rm_in_list n ms p0 p1 Delta).
      apply (all_some_some _ _ (R_m_aux Delta (p0 ++ repeat Tgt n ++ [Src]) (nth_ok ms n p1))) in Hallsome.
      * destruct Hallsome.
        eapply (sfc_monotone_aux_list _ x0).
        ** unfold Rsub_list. intros. destruct a.
           rewrite <- H.
           revert H1 H3. clear ...
           intros. induction l.
           {
             inversion H3.
           }
           {
             simpl.
             apply in_or_app.
             destruct H3.
             - left. subst. assumption.
             - right. apply IHl. assumption.
           }
```

```
        ** apply p.
           destruct a. assumption.
      * apply in_or_app. left. apply H0.
  - intros. constructor.
    apply IHm. rewrite <- H.
    reflexivity.
Qed.

Lemma SfC_var_size : forall x ms Delta R pi, SfC Delta R (!!x @@ ms) pi -> x < length Delta.
Proof.
  intros.
  ainv.
  apply nth_error_Some.
  intros F. rewrite H0 in F. discriminate F.
Qed.

Lemma SfC_var_in_some x ms {Delta pi} : { R & SfC Delta R (!!x @@ ms) pi } -> nth_error Delta x <> None.
Proof.
  intros.
  destruct X.
  apply SfC_var_size in s.
  apply nth_error_Some.
  assumption.
Qed.

Lemma SfC_lam_proof s {Delta pi}: {R & SfC Delta R (\__ s) pi} -> {R & SfC ((pi ++ [Src])::Delta) R s (pi ++ [Tgt])}.
Proof.
  intros.
  destruct X.
  inversion s0.
  exists x. assumption.
Qed.

Lemma SfC_var_proof x ms n m pi' {Delta pi} p ltp: {R & SfC Delta R (!!x @@ ms) pi} -> m = (nth_ok ms n p) -> pi' = nth_ok Delta x ltp -> {R & SfC Delta R m (make_tgt_path pi' n)}.
Proof.
  intros.
  destruct X.
  inversion s.
  remember (nth_ok Delta x ltp).
  symmetry in Heqp0.
  rewrite nth_ok_nth_error in Heqp0.
  rewrite H4 in Heqp0. apply some_eq in Heqp0. subst.
  eexists.
  apply X0.
Qed.

Lemma exists_R_m m : forall Delta, all_var_in_repo m Delta -> forall pi, { R' & R_m_aux Delta pi m = Some R'}.
Proof.
  induction m using nfterm_rect'.
```

```
  - intros. asimpl in *. unfold all_var_in_repo in H.
    pose proof H as Hx. apply fold_left_max_acc in Hx.
    apply lt_S_n in Hx.
    pose proof (in_seq (length ms) 0).
    assert (forall n, In n (seq 0 (length ms)) → exists lp m, nth_ok ms n lp = m) as Hseqlen.
    {
      clear ...
      intros.
      apply in_seq in H. destruct H. asimpl in H0.
      apply nth_error_Some2 in H0.
      destruct H0. apply nth_error_nth_ok in e.
      destruct e. exists x0. exists x. assumption.
    }
    assert (forall (n : nat) (lp : n < length ms), max_fvar (nth_ok ms n lp) < S (length Delta)) as Hmaxfvar.
    {
      revert H Hx.
      clear ...
      induction n.
      - intros. destruct ms.
        + inversion lp.
        + asimpl. asimpl in H. apply fold_left_max_acc in H. destruct (max_fvar n).
          * apply Nat.lt_0_succ.
          * apply lt_S_n in H. apply Nat.max_lub_lt_iff in H. destruct H. apply lt_n_S. assumption.
      - intros. destruct ms.
        + inversion lp.
        + remember (nth_ok _ _ _) as n1. symmetry in Heqn1. apply nth_ok_nth_error in Heqn1.
          asimpl in Heqn1. eapply fold_left_max_in in H.
          * apply H.
          * apply nth_error_In in Heqn1. constructor 2. apply in_map. assumption.
    }
    assert (forall (n : nat) (lp : n < length ms) (pi : path),
               { R' : list (path * path) & R_m_aux Delta pi (nth_ok ms n lp) = Some R'}) as p'.
    {
      intros.
      apply p.
      apply Hmaxfvar.
    }
    assert (forall m pi, In m ms →
                { R' : list (path * path) & R_m_aux Delta pi m = Some R'}) as p''.
    {
      intros. pose proof (H1). apply In_nth_error_set in H2. destruct H2.
      assert (x0 < length ms). {
        apply nth_error_Some. intros F. rewrite e in F. discriminate F.
      }
      pose proof (p' x0 H2 pi0).
      destruct X. exists x1. rewrite ← nth_ok_nth_error in e. rewrite (nth_ok_proof_irel _ _ _ H2) in e.
      subst. assumption.
```

```
      Unshelve.
      apply H2.
    }
    apply nth_error_Some in Hx. destruct (nth_error Delta x); try (exfalso; apply Hx; reflexivity).
    rewrite (combine_with_Rstuff p0 Delta).
    unfold option_concat.
    destruct (all_some _) eqn:Hall.
    + eexists. reflexivity.
    + apply all_some_none_last in Hall. apply all_some_none_exists in Hall.
      rewrite combine_with_map in Hall. apply in_map_set in Hall. destruct Hall as [[m b] [HRNone HInc]].
      asimpl in HRNone. apply in_combine_l in HInc. pose proof (p'' m (make_tgt_path p0 b) HInc).
      destruct X. rewrite HRNone in e. discriminate e.
  - intros. simpl in *. apply IHm. unfold all_var_in_repo in *.
    asimpl in *. omega.
Qed.

Definition closed m := max_fvar m = 0.

Hint Unfold closed.

Lemma closed_Rm : forall m, closed m -> SfC [] (R_m_ts m) m [].
Proof.
  intros.
  unfold R_m_ts.
  destruct (R_m m) eqn:HRm.
  - apply R_m_ts_correct. exact HRm.
  - unfold R_m in HRm.
    pose proof (exists_R_m m []) as H0.
    unfold all_var_in_repo in H0. rewrite H in H0.
    asimpl in H0. pose proof (H0 Nat.lt_0_1). exfalso.
    destruct (X []). rewrite HRm in e. discriminate e.
Qed.

Lemma SfC_repo_size : forall m R Delta pi, SfC Delta R m pi -> max_fvar m < S (length Delta).
Proof.
  induction 1.
  - simpl.
    asimpl in *.
    apply lt_S_n.
    unfold pred.
    destruct (max_fvar s) eqn:Hmf.
    + omega.
    + assumption.
  - asimpl.
    assert (nth_error Delta x <> None). { intros F. rewrite e in F. discriminate F. }
                                           apply nth_error_Some in H0.
    assert (forall m, In m ms -> max_fvar m < S (length Delta)).
      {
        intros. eapply (forall_length_in ms).
        - intros. apply In_nth_error in H1. destruct H1. apply nth_error_nth_ok in H1.
```

```
          destruct H1. rewrite <- e0. apply H.
        - apply H1.
      }
      apply in_fold_left_max.
    + intros. apply in_map_iff in H2. destruct H2 as [x0 [Hmfx Hinx]]. pose proof (H1 x0 Hinx).
      rewrite Hmfx in H2. assumption.
    + apply lt_n_S. assumption.
Qed.

Lemma SfC_closed : forall R m, SfC [] R m [] -> closed m.
Proof.
  intros.
  unfold closed.
  apply SfC_repo_size in X.
  asimpl in X.
  omega.
Qed.

Lemma Long_repo_size : forall m Delta tau, nfty_long Delta m tau -> max_fvar m < S (length Delta).
Proof.
  induction 1.
  - simpl.
    asimpl in *.
    apply lt_S_n.
    unfold pred.
    destruct (max_fvar s) eqn:Hmf.
    + omega.
    + assumption.
  - asimpl in *.
    assert (nth_error Gamma x <> None). { intros F. rewrite Gammaok in F. discriminate F. }
                                       apply nth_error_Some in H0.
    assert (forall m, In m ms -> max_fvar m < S (length Gamma)).
      {
        intros. eapply (forall_length_in ms).
        - intros. apply In_nth_error in H1. destruct H1.
          apply nth_error_nth_ok in H1. destruct H1.
          rewrite <- e. apply H.
        - apply H1.
      }
      apply in_fold_left_max.
    + intros. apply in_map_iff in H2. destruct H2 as [x0 [Hmfx Hinx]]. rewrite <- Hmfx.
      apply H1. assumption.
    + apply lt_n_S. assumption.
Qed.

Lemma Long_closed : forall m tau, nfty_long [] m tau -> closed m.
Proof.
  intros. unfold closed. apply Long_repo_size in X. asimpl in X. omega.
Qed.
```

```
Lemma R_m_ts_minimal : forall m Rm Delta pi,  SfC Delta Rm m pi → forall Rm', R_m_aux Delta pi m = Some Rm' →
                                                      Rsub (fun p p' ⇒ In (p, p') Rm') Rm.
Proof.
  induction 1.
  - intros. apply IHX. simpl in H. assumption.
  - intros.
    assert (forall skip, (forall (n : nat) (p : n < length ms), SfC Delta R (nth_ok ms n p) (make_tgt_path pi (skip + n))) →
        Forall2_T (SfC Delta R) ms (map (make_tgt_path pi) (seq skip (length ms)))).
    {
      clear ...
      induction ms.
      - intros. constructor.
      - intros. simpl. constructor.
        + pose proof (X 0 (Nat.lt_0_succ _)).
          asimpl in X0. apply X0.
        + assert (forall (n : nat) (p : n < length ms), SfC Delta R (nth_ok ms n p) (make_tgt_path pi (S skip + n))
                ). {
            intros.
            pose proof (X (S n) (Lt.lt_n_S _ _ p)).
            asimpl in X0.
            erewrite (nth_ok_proof_irel) in X0.
            apply X0.
          }
          apply IHms in X0. assumption.
    }
    apply (X0 0) in s.
    clear X0.
    asimpl in H. rewrite e in H. rewrite (combine_with_Rstuff pi Delta) in H. rewrite combine_with_map in H.
    apply option_concat_app in H. destruct H as [oms1 [oms2 [Hom1 [Hom2 Heq]]]]. asimpl in Hom2.
    apply some_eq in Hom2.
    rewrite Heq. apply Rsub_in_app.
    + unfold option_concat in Hom1.
      destruct (all_some _) eqn:Hall;try discriminate Hom1.
      apply some_eq in Hom1. subst. apply Rsub_in_concat. intros.
      apply all_some_map with (m0:=m) in Hall.
      destruct Hall as [com [Hinc HRm]]. destruct com as [t n].
      pose proof (in_combine_r ms _ _ _ Hinc) as Hinran.
      apply in_seq in Hinran. destruct Hinran as [_ pr]. simpl in pr.
      apply (X n pr). asimpl in HRm. apply in_combine_range with (start := 0) (pr0 := pr) in Hinc. ainv. assumption.
    + subst. unfold Rsub.
      intros. assert ((pi0, pi'0) = (pi ++ repeat Tgt (length ms), pi')).
      { inversion H. symmetry. assumption. inversion H0. }
      inversion H0. subst. assumption.
Qed.

Definition R_tau_cond (tau: type) (pipi' : path * path) : bool :=
  let pi := fst pipi' in
  let pi' := snd pipi' in
```

```
  (pi <>b pi') && match P tau pi with
                  | None => false
                  | Some (sigma ~> tau) => false
                  | Some (? a) => match P tau pi' with
                                 | None => false
                                 | Some a' => (? a) ==b a'
                                 end
                  end
.


Definition R_tau_list tau :=
  filter (R_tau_cond tau)
  (list_prod (dom_P tau) (dom_P tau)).

(* Another Definition of R_tau *)
Inductive R_tau (tau: type) : path -> path -> Type :=
| R_tau_in pi pi' :
    In pi (dom_P tau) -> In pi' (dom_P tau) -> pi <> pi' -> {a & P tau pi = Some (? a) } ->
    P tau pi = P tau pi' -> R_tau tau pi pi'.

(* Equivalence closure over R_tau *)
Definition R_tau_ts (tau: type) := ts_cl_list (R_tau_list tau).

(* Equivalence between R_tau_list and R_tau *)
Lemma R_tau_list_type : forall tau pi pi', In (pi, pi') (R_tau_list tau) <-> inhabited (R_tau tau pi pi').
Proof.
split.
- induction tau.
  + ainv.
  + intros. unfold R_tau_list in H. rewrite filter_In in H. destruct H as [H1 H2]. rewrite in_prod_iff in H1.
    destruct H1 as [Hpi Hpi']. unfold R_tau_cond in H2. simpl in H2.  apply andb_prop in H2 as [H2 H3].
    apply nequivb_prop in H2.  constructor.
    constructor; try assumption; destruct P eqn:HP; try discriminate H3;
      destruct t; try discriminate H3.
      * exists x. reflexivity.
      * destruct (P (tau1 ~> tau2) pi'); try discriminate H3.
        rewrite equivb_prop in H3. subst. reflexivity.
- intros. inversion H. inversion X.
  unfold R_tau_list. rewrite filter_In. split.
  + rewrite in_prod_iff.
    split; assumption.
  + unfold R_tau_cond. simpl.
    apply andb_true_intro. split.
    * apply nequivb_prop. assumption.
    * destruct P; ainv; try apply equivb_prop; ainv.
Qed.

Definition Delta2Gamma (tau: type) Delta : option (list type) :=
  all_some (map (P tau) Delta).
```

```
Lemma subterm_long_ty : forall m tau Delta, nfty_long Delta m tau → forall n,
  subterm_nf n m → {Delta' & { tau' & nfty_long Delta' n tau'}}.
Proof.
  induction 1.
  - intros. inversion X0.
    + subst. exists Gamma. exists (sigma ⇝ tau). constructor. assumption.
    + subst. apply IHX. assumption.
  - intros. inversion X0.
    + subst. exists Gamma. eexists. econstructor. eapply Gammaok. apply n.
    + subst. apply In_nth_error_set in H1. destruct H1 as [nr H1]. apply
      nth_error_nth_ok in H1.
      destruct H1 as [Hlen Hnth]. rewrite ← Hnth in X1.
      pose proof (X nr Hlen n0 X1). assumption.
Qed.

Fixpoint wrap_lam (n: nat) m :=
  match n with
  | 0 ⇒ m
  | S n ⇒ \__ (wrap_lam n m)
  end.

Lemma Delta2Gamma_pump : forall Delta rho rho' pi, P rho pi = Some rho' →
  forall Gamma, Delta2Gamma rho Delta = Some Gamma →
                           Delta2Gamma rho (pi::Delta) = Some (rho' :: Gamma).
Proof.
  intros. asimpl. rewrite H. unfold Delta2Gamma in H0. rewrite H0. reflexivity.
Qed.

Lemma P_lam_proof_Src {rho pi m pr Gamma} :
  nfty_long Gamma (\__ m) (P_ok rho pi pr) → In (pi ++ [Src]) (dom_P rho).
Proof.
  intros. inversion X. subst. symmetry in H. rewrite P_ok_P in H. apply P_src in
    H.
  rewrite ← P_ok_P_ex in H. destruct H. assumption.
Qed.

Lemma P_lam_proof_Tgt {rho pi m pr Gamma} :
  nfty_long Gamma (\__ m) (P_ok rho pi pr) → In (pi ++ [Tgt]) (dom_P rho).
Proof.
  intros. inversion X. subst. symmetry in H. rewrite P_ok_P in H. apply P_tgt in
    H.
  rewrite ← P_ok_P_ex in H. destruct H. assumption.
Qed.

Lemma P_lam_step : forall m pi Delta R,
    SfC Delta R (\__ m) pi → forall rho Gamma,
      Delta2Gamma rho Delta = Some Gamma →
      forall pr (nfpr : nfty_long Gamma (\__ m) (P_ok rho pi pr)),
        nfty_long
          (P_ok rho (pi ++ [Src]) (P_lam_proof_Src nfpr) :: Gamma)
          m (P_ok rho (pi ++ [Tgt]) (P_lam_proof_Tgt nfpr)).
Proof.
  intros. asimpl. inv nfpr. inv X. symmetry in H2. apply P_ok_P in H2.
  assert (P_ok rho (pi ++ [Src]) (P_lam_proof_Src nfpr) = sigma).
  {
```

```
    apply P_ok_P. eapply P_src. apply H2.
  }
  assert (P_ok rho (pi ++ [Tgt]) (P_lam_proof_Tgt nfpr) = tau).
  {
    apply P_ok_P. eapply P_tgt. apply H2.
  }
  subst. assumption.
Qed.

Lemma nfty_app_x_in_Gamma {x ms a Gamma} : nfty_long Gamma ( !! x @@ ms) a →
                           {ts & nth_error Gamma x = Some (make_arrow_type ts
                            ↪ a) }.
Proof.
  intros. inversion X.  exists ts. assumption.
Qed.

Lemma P_app_step {Delta pi' x ms R}:
  SfC Delta R ( !! x @@ ms) pi' →
  forall rho Gamma pi pr' ts a,
    nth_error Delta x = Some pi →
    nth_error Gamma x = Some (make_arrow_type ts (? a)) →
    forall (nfpr : nfty_long Gamma ( !! x @@ ms) (P_ok rho pi' pr'))
       (prp: P rho pi = Some (make_arrow_type ts (P_ok rho pi' pr'))) (leneq :
        ↪ length ms = length ts)
    n (lenpr: n < length ms),
      nfty_long Gamma (nth_ok ms n lenpr) (P_ok rho (pi ++ repeat Tgt n ++
       ↪ [Src])
                                                     (P_app_proof_in prp n (rew leneq
                                                      ↪ in lenpr))).
Proof.
  intros. inv nfpr. inv X. assert (ts0 = ts /\ a = a0) as [Hts Ha].
  {
    revert Gammaok H0. clear ...
    intros. rewrite H0 in Gammaok. clear H0. apply some_eq in Gammaok.
    unfold make_arrow_type in Gammaok.
    generalize dependent ts0. induction ts; intros.
    - simpl in Gammaok.
      destruct ts0.
      + split. reflexivity. ainv.
      + ainv.
    - destruct ts0.
      + ainv.
      + pose proof (IHts ts0).
        assert (t = a1).
        {
          ainv.
        }
        subst. ainv. apply H in H2. destruct H2. split. ainv. assumption.
  }
  subst.
  pose proof (X0 n lenpr).
  assert (P rho (pi ++ repeat Tgt n ++ [Src]) = Some (nth_ok ts n (rew [lt n]
   ↪ Lenproof in lenpr))).
  {
    eapply P_app_proof.
    apply prp.
```

```
  }
  apply P_P_ok_set in H1. destruct H1 as [pr'' H1]. erewrite P_ok_proof_irl. rewrite H1.
  assumption.
Qed.

Lemma Delta2Gamma_length {rho Delta Gamma} : Delta2Gamma rho Delta = Some Gamma → length Delta = length Gamma.
Proof.
  unfold Delta2Gamma. intros. apply all_some_length in H. rewrite map_length in H. assumption.
Qed.

Lemma Delta2Gamma_nth {rho Delta Gamma}: forall (HD2G: Delta2Gamma rho Delta = Some Gamma)
                                             x (pr : x < length Delta),
    P rho (nth_ok Delta x pr) = Some (nth_ok Gamma x (rew (Delta2Gamma_length HD2G) in pr)).
Proof.
  intros.
  intros. unfold Delta2Gamma in HD2G.
  assert (In (P rho (nth_ok Delta x pr)) (map (P rho) Delta)).
  {
    apply map_in. apply nth_ok_in.
  }
  pose proof (all_some_nth _ _ HD2G).
  erewrite (nth_ok_proof_irel _ Gamma).
  rewrite ← H0. rewrite nth_ok_map.
  erewrite nth_ok_proof_irel. reflexivity. Unshelve.
  rewrite map_length. assumption.
Qed.

Lemma sfc_to_long_subj {rho} : forall Delta R m pi (base_sfc : SfC Delta R m pi) Gamma,
               Delta2Gamma rho Delta = Some Gamma →
               forall pr (base_long : nfty_long Gamma m (P_ok rho pi pr))
               Delta' m' pi' (subj_sfc : SfC Delta' R m' pi'),
                 SfC_subj _ _ _ _ _ _ _ subj_sfc base_sfc →
                 { Gamma' & prod (Delta2Gamma rho Delta' = Some Gamma')
                          { pr' & nfty_long Gamma' m' (P_ok rho pi' pr')}}.
Proof.
  intros.
  generalize dependent Gamma.
  generalize dependent pr.
  induction X.
  - intros. exists Gamma. split. assumption. exists pr. assumption.
  - intros. pose proof (IHX2 pr Gamma H base_long) as [Gamma' [HD2G [pr' nfty1]]].
    pose proof (IHX1 pr' Gamma' HD2G nfty1) as [Gamma'' [HD2G' [pr'' nfty2]]].
    exists Gamma''. split. assumption. exists pr''. assumption.
  - intros.
    assert (In (pi ++ [Src]) (dom_P rho)) as prSrc.
    {
      eapply P_lam_proof_Src. exact base_long.
    }
    exists (P_ok rho (pi ++ [Src]) prSrc :: Gamma). split.
```

```
    { apply Delta2Gamma_pump.
      - remember (P_ok rho (pi ++ [Src]) prSrc). eapply P_ok_P. symmetry. exact
        ↪ Heqt.
      - assumption.
    }
    pose proof (P_lam_step m pi Delta R (SfC_I _ _ _ _ proof) rho Gamma H pr) as
      ↪ HlamStep.
    pose proof (dom_P_Src_to_Tgt _ _ _ Tgt prSrc) as prTgt.
    exists prTgt.
    assert (forall prSrc' pr'', nfty_long (P_ok rho (pi ++ [Src]) prSrc' ::
      ↪ Gamma) m (P_ok rho (pi ++ [Tgt]) pr'')
                  → nfty_long (P_ok rho (pi ++ [Src]) prSrc
                    ↪ :: Gamma) m (P_ok rho (pi ++ [Tgt])
                    ↪ prTgt)).
    {
      intros. rewrite P_ok_proof_irl with (p2 := prSrc').
      rewrite P_ok_proof_irl with (p2 := pr''). assumption.
    }
    eapply X. clear X.
    eapply HlamStep. Unshelve. erewrite (P_ok_proof_irl). apply base_long.
  - intros. inversion base_long.
    pose proof (SfC_E Delta R ms pi pi' x deltaok Rproof proofs) as SfC_base.
    exists Gamma.
    assert (P rho pi = Some (make_arrow_type ts (P_ok rho pi' pr))).
    {
      rewrite ← H3. rewrite ← Gammaok.
      revert H. revert deltaok. clear ...
      intros. apply nth_error_nth_ok in deltaok. destruct deltaok as [pr
        ↪ deltaok].
      pose proof (Delta2Gamma_nth H x pr). remember (nth_ok Gamma x _). symmetry
        ↪ in Heqt.
      apply nth_ok_nth_error in Heqt. subst. rewrite ← Heqt in H0. exact H0.
    } subst.
    pose proof (
      P_app_step SfC_base rho Gamma pi pr ts a deltaok Gammaok base_long H0
        ↪ Lenproof n ltproof
        ). split. assumption.
    eexists. apply X0.
Qed.

Lemma Delta2Gamma_nth_error : forall Delta Gamma rho, Delta2Gamma rho Delta =
  ↪ Some Gamma →
               forall x pi tau, nth_error Delta x = Some pi →
                ↪ nth_error Gamma x = Some (tau) →
                  {pr & P_ok rho pi pr = tau}.
Proof.
  intros.
  pose proof (nth_error_nth_ok _ _ _ H0) as [pr H0_ok].
  pose proof (nth_error_nth_ok _ _ _ H1) as [pr' H1_ok].
  pose proof (Delta2Gamma_nth H x pr). rewrite H0_ok in H2.
  apply P_P_ok_set in H2. destruct H2 as [pr0 H2].
  rewrite (nth_ok_proof_irel _ _ _ pr') in H2. rewrite H1_ok in H2.
  exists pr0. assumption.
Qed.
```

```
Lemma sfc_app_subj_types_atomic {rho} : forall R m someDelta somepi (base_sfc :
 ↪  SfC someDelta R m somepi)
      someGamma somepr (base_long : nfty_long someGamma m (P_ok rho somepi
       ↪  somepr))
      (someD2G: Delta2Gamma rho someDelta = Some someGamma)
      Delta x pi (Deltaok : nth_error Delta x = Some pi)
                 ms pi' (subj_sfc : SfC Delta R (!! x @@ ms) pi'),
                 SfC_subj _ _ _ _ _ _ _ _ subj_sfc base_sfc →
                 { a & P rho (pi ++ repeat Tgt (length ms)) = Some (? a) /\ P
                  ↪  rho (pi') = Some (? a) }.
Proof.
  intros.
  pose proof (sfc_to_long_subj _ R m _ base_sfc _ someD2G somepr base_long Delta
   ↪  (!! x @@ ms) pi' subj_sfc X)
    as [Gamma [D2G [pr nfty]]].
  inversion nfty. rewrite Lenproof in *. subst. exists a. split.
  - rewrite H.
    pose proof (Delta2Gamma_nth_error _ _ _ D2G _ _ _ Deltaok Gammaok) as [D2Gpr
     ↪  Hpirho].
    rewrite ← H.
    pose proof (P_ok_make_arrow ts (? a)) as [mkpr HPmk].
    eapply P_app_split.
    + eapply P_ok_P. exact Hpirho.
    + eapply P_ok_P. exact HPmk.
  - eapply P_ok_P. symmetry. exact H.
Qed.

Lemma sfc_app_subj_R_tau_cond {rho m R Delta x pi ms pi'} :
  forall (base_sfc : SfC [] R m []) (base_long : nfty_long [] m rho)
    (subj_sfc : SfC Delta R (!! x @@ ms) pi')
    (Deltaok : nth_error Delta x = Some pi),
    SfC_subj _ _ _ _ _ _ _ _ subj_sfc base_sfc →
    R_tau_cond rho (pi ++ repeat Tgt (length ms), pi') = true.
Proof.
  intros.
  pose proof (sfc_app_subj_types_atomic _ _ _ _ base_sfc _ (dom_P_nil _)
   ↪  base_long eq_refl _ _ _ Deltaok _ _ subj_sfc X) as [a [Hpi Hpi']].
  unfold R_tau_cond. simpl. rewrite Hpi. rewrite Hpi'. apply andb_true_intro.
  split; try (apply equivb_prop; reflexivity).
  pose proof (r_not_refl base_sfc).
  pose proof (each_judg_subj_SfC_P _ _ _ _ _ _ _ H _ _ _ _ X). unfold
   ↪  r_not_refl_cond in H0. simpl in H0.
  pose proof SfC_gen_app _ _ _ _ _ subj_sfc.
  rewrite ← H1 in H0.
  pose proof (get_subproof_app_deltaok subj_sfc).
  rewrite H2 in Deltaok. apply some_eq in Deltaok. subst.
  apply nequivb_prop. assumption.
Qed.

Lemma sfc_app_subj_in_R_tau {rho m R} :
  forall (base_sfc : SfC [] R m []) (base_long : nfty_long [] m rho) Delta x pi
   ↪  (Deltaok : nth_error Delta x = Some pi) ms pi'
    (subj_sfc : SfC Delta R (!! x @@ ms) pi'),
    SfC_subj _ _ _ _ _ _ _ _ subj_sfc base_sfc →
    R_tau_ts rho (pi ++ repeat Tgt (length ms)) pi'.
Proof.
```

```
  intros.
  pose proof (sfc_app_subj_R_tau_cond base_sfc base_long subj_sfc Deltaok X).
  pose proof (sfc_app_subj_types_atomic _ _ _ _ base_sfc _ (dom_P_nil _) base_long eq_refl _ _ _ Deltaok _ _ subj_sfc X) as [a [Hpi Hpi']].
  unfold R_tau_ts.
  unfold R_tau_list.
  constructor.
  apply filter_In. split.
  - apply in_prod.
    + apply P_P_ok_set in Hpi as [pr _]. assumption.
    + apply P_P_ok_set in Hpi' as [pr _]. assumption.
  - assumption.
Qed.

Lemma sfc_replace_R {R m Delta' pi''}: forall (base_sfc: SfC Delta' R m pi'') R',
    (forall Delta x pi, nth_error Delta x = Some pi -> forall ms pi' (subj_sfc: SfC Delta R ( !! x @@ ms) pi'),
          SfC_subj _ _ _ _ _ _ _ _ subj_sfc base_sfc -> R' (pi ++ repeat Tgt (length ms)) pi')
    -> SfC Delta' R' m pi''.
Proof.
  intros base_sfc.
  induction base_sfc.
  - intros.
    constructor. apply IHbase_sfc.
    intros.
    eapply X.
    + exact H.
    + remember (SfC_subj_I R Delta s pi base_sfc).
      remember (SfC_subj_trans _ _ _ _ _ _ _ _ _ _ _ _ _ X0 s0).
      apply s1.
  - intros. econstructor.
    + apply e.
    + eapply X0.
      * apply e.
      * constructor.
    + intros. apply X. intros.
      eapply X0.
       ** apply H.
       ** remember (SfC_subj_E R Delta pi pi' ms x e r s n p).
          remember (SfC_subj_trans _ _ _ _ _ _ _ _ _ _ _ _ _ X1 s0).
          apply s1.
Qed.

Lemma long_to_sfc_tau {rho m} : nfty_long [] m rho ->
    SfC [] (R_tau_ts rho) m [].
Proof.
  intro base_long.
  pose proof (Long_closed _ _ base_long) as Hclosed.
  pose proof (closed_Rm _ Hclosed) as base_sfc.
  pose proof (sfc_app_subj_in_R_tau base_sfc base_long).
  pose proof sfc_replace_R base_sfc _ X.
  assumption.
Qed.
```

```
Lemma sfc_tau_to_Rsub_m_tau {m tau} : SfC [] (R_tau_ts tau) m [] → Rsub (R_m_ts m) (R_tau_ts tau).
Proof.
  intros. unfold R_m_ts. destruct (R_m m) eqn:HRm.
  - apply R_m_ts_minimal with (Rm':=l) in X.
    + unfold Rsub in *. intros. unfold R_tau_ts.
      induction X0.
      * exact (X _ _ i).
      * econstructor 2. exact IHX0.
      * econstructor 3. exact IHX0_1. exact  IHX0_2.
    + exact HRm.
  - unfold Rsub. intros. inversion H.
Qed.

Lemma pi_in_R : forall m Delta pi R, SfC Delta R m pi → {pi' & {app & R pi' (pi ++ app)}}.
Proof.
  induction 1.
  - destruct IHX as [pi' [app IHX]]. exists pi'. exists ([Tgt] ++ app). rewrite app_assoc. assumption.
  - exists (pi ++ repeat Tgt (length ms)). exists []. rewrite app_nil_r. assumption.
Qed.

Lemma R_tau_ts_dom_P {tau pi pi'} : R_tau_ts tau pi pi' → {a & P tau pi = Some (? a) /\ P tau pi' = Some (? a)}.
Proof.
  intros.
  induction X.
  - unfold R_tau_list in i. apply filter_In in i as [i x]. unfold R_tau_cond in x.
    simpl in x. apply andb_prop in x. destruct x as [nab pt]. destruct (P tau a) eqn:HPa; try discriminate pt.
    destruct t eqn:Htl ; try discriminate pt. destruct (P tau b) eqn:HPb; try discriminate pt.
    exists x. rewrite equivb_prop in pt. subst. auto.
  - ainv. exists x. split; assumption.
  - destruct IHX1 as [x1 [Pa Pb]]. exists x1.
    destruct IHX2 as [x2 [Pa1 Pb2]]. assert (x1 = x2). rewrite Pa1 in Pb. injection Pb. auto. subst. auto.
Qed.

Lemma Delta2Gamma_x {rho Delta Gamma x pi}: Delta2Gamma rho Delta = Some Gamma →
                     nth_error Delta x = Some pi →
                     {pr & nth_error Gamma x = Some (P_ok rho pi pr)}.
Proof.
  intros.
  pose proof nth_error_Some3 _ _ _ H0 as lenpr.
  pose proof Delta2Gamma_nth H x lenpr as Prhoeq.
  pose proof nth_error_nth_ok _ _ _ H0 as [pr Hnthok].
  erewrite nth_ok_proof_irel in Hnthok.
  rewrite Hnthok in Prhoeq.
  pose proof P_P_ok_set Prhoeq as [pr0 HPok].
  exists pr0. rewrite HPok. remember (nth_ok Gamma x _).
  symmetry in Heqt.
```

```
  rewrite nth_ok_nth_error in Heqt. assumption.
Qed.

Lemma Rsub_m_tau_to_Long_aux {m tau} : forall Delta pi R, SfC Delta R m pi →
    forall Gamma, Delta2Gamma tau Delta = Some Gamma →
      Rsub R (R_tau_ts tau) →
      {pr & nfty_long Gamma m (P_ok tau pi pr)}.
Proof.
  induction 1.
  - intros. unfold Rsub in X0.
    pose proof pi_in_R _ _ _ _ X as [pi' [appl HR]].
    pose proof X0 _ _ HR as HRtau.
    assert ({rho & {pr & P_ok tau (pi ++ [Src]) pr = rho}}) as [rho [pr HPok]]. {
      pose proof R_tau_ts_dom_P HRtau. destruct H0 as [a [HPpi HPpi']].
       pose proof P_prefix HPpi' as [tau' HPpi_Tgt].
        pose proof P_P_ok_set HPpi_Tgt as [pr HPok].
        pose proof P_ok_Src_to_Tgt _ _ _ Src _ _ HPok as [pr' [rho HP]].
        exists rho. exists pr'. assumption.
    }
    pose proof proj1 P_ok_P HPok as HP.
    pose proof IHX (rho :: Gamma) (Delta2Gamma_pump _ _ _ _ HP _ H) X0 as [prTgt nfbase].
    pose proof P_Src2 _ _ _ HP as [tau' [HPbase HPtgt]].
    pose proof P_P_ok_set HPbase as [pr' HPokbase].
    exists pr'. rewrite HPokbase.
    constructor.
    pose proof P_P_ok_set HPtgt as [pr'' HPoktgt].
    rewrite ← HPoktgt. erewrite P_ok_proof_irl. exact nfbase.
  - intros. unfold Rsub in X0.
    pose proof X0 _ _ r as HRtau.
    pose proof R_tau_ts_dom_P HRtau as [a [Ppi Ppi']].
    pose proof P_P_ok_set Ppi' as [pr Pokpi'].
    exists pr. rewrite Pokpi'. pose proof P_path_make_arrow_type Ppi as [ts [HPts HLeneq]].
    econstructor.
    + pose proof Delta2Gamma_x H e as [pr' HGamma].
      erewrite ← P_ok_P in HPts. rewrite HPts in HGamma. exact HGamma.
    + intros.
      pose proof X n pms Gamma H X0 as [pr' Hnft].
      erewrite (nth_ok_proof_irel n ts).
      assert (P_ok tau (make_tgt_path pi n) pr' = nth_ok ts n (rew ← HLeneq in pms)). {
        remember (make_arrow_type ts (? a)) as rho.
        symmetry in Heqrho.
        pose proof make_arrow_type_dirs Heqrho (n:=n).
        destruct (nth_error ts n) eqn:Hnthts.
        - epose proof P_app_split HPts H0.
          rewrite P_ok_P. unfold make_tgt_path. rewrite H1.
          rewrite ← some_eq. symmetry. apply nth_ok_nth_error. assumption.
        - apply nth_error_None in Hnthts.
```

```
          pose proof lt_not_le _ _ (rew ← HLeneq in pms). contradiction.

      }
      rewrite ← H0.
      assumption. Unshelve. symmetry. assumption.
Qed.

Lemma Rsub_m_tau_to_Long {m tau} : closed m → Rsub (R_m_ts m) (R_tau_ts tau) → nfty_long [] m tau.
Proof.
  intros Hclosed.
  pose proof (closed_Rm _ Hclosed) as base_sfc.
  intros.
  pose proof (Rsub_m_tau_to_Long_aux _ _ _ base_sfc [] eq_refl X) as [pr nf].
  simpl in nf. assumption.
Qed.

Lemma long_to_Rsub_m_tau {m tau} : nfty_long [] m tau → Rsub (R_m_ts m) (R_tau_ts tau).
Proof.
  intros.
  apply sfc_tau_to_Rsub_m_tau.
  apply long_to_sfc_tau.
  assumption.
Qed.

Lemma R_m_ts_dec : forall m pi pi', (R_m_ts m pi pi') + (R_m_ts m pi pi' → False).
Proof.
  intros.
  unfold R_m_ts.
  destruct (R_m m).
  + apply ts_cl_list_dec.
  + right. intros F. inversion F.
Defined.

Lemma R_tau_ts_dec : forall m pi pi', (R_tau_ts m pi pi') + (R_tau_ts m pi pi' → False).
Proof.
  intros.
  unfold R_tau_ts.
  apply ts_cl_list_dec.
Defined.

Definition replaceable_paths_cond m pi pi' : bool :=
  if (R_m_ts_dec m pi pi') then true else
    if (pi == pi') then true else false.

Definition replaceable_paths tau m pi : list path :=
  filter (replaceable_paths_cond m pi) (dom_P tau).

Definition replace_all_paths (tau: type)(pis : list path) (b : type) :=
  fold_left (replace_at_path b) pis tau.

Lemma replace_at_nil : forall b tau, replace_at_path b tau [] = b.
Proof.
```

```
  induction tau; reflexivity.
Qed.

Lemma replace_all_var: forall pis b,
  replace_all_paths (? b) pis (? b) = ? b.
Proof.
  induction pis.
  - reflexivity.
  - intros. induction a.
    + simpl. apply IHpis.
    + destruct a.
      * simpl. apply IHpis.
      * simpl. apply IHpis.
Qed.

Lemma replace_all_var_is_var: forall pis b c,
  {d & replace_all_paths (? c) pis (? b) = ? d}.
Proof.
  induction pis.
  - eexists. reflexivity.
  - intros. induction a.
    + simpl. apply IHpis.
    + destruct a.
      * simpl. apply IHpis.
      * simpl. apply IHpis.
Qed.
(*
Lemma nil_in_replace_paths2 : forall pis tau b, replace_all_paths tau pis (? b) = (? b) -> {In [] pis} + {tau = (? b)}.
Proof.
   induction pis.
   - intros. destruct tau.
     + simpl in H. right. assumption.
     + simpl in H. discriminate H.
   - intros. simpl in H. destruct (tau == ? b). right. assumption.
     destruct a. left. constructor. reflexivity.
     apply IHpis in H.
     destruct H.
     + left. constructor 2. assumption.
     + simpl in e.

     destruct ((replace_at_path (? b) tau a)) eqn:Hr.
       * rewrite e in *. left. constructor 2.

     destruct (IHpis tau b).
     +

    + intros. destruct H.
      * subst. simpl. apply replace_all_var.
      * simpl. destruct a.
        -- simpl. apply replace_all_var.
        -- destruct d.
           ++ simpl. apply IHpis. assumption.
           ++ simpl. apply IHpis. assumption.
Qed.*)

```

```
Lemma nil_in_replace_paths : forall pis tau b, In [] pis → replace_all_paths
 ↪ tau pis (? b) = (? b).
Proof.
    induction pis.
    + ainv.
    + intros. destruct H.
      * subst. simpl. apply replace_all_var.
      * simpl. destruct a.
        -- simpl. apply replace_all_var.
        -- destruct d.
           ++ simpl. apply IHpis. assumption.
           ++ simpl. apply IHpis. assumption.
Qed.

Lemma lt_S_l_max x y: x ⩽ S (Init.Nat.max x y).
Proof.
  destruct (Nat.max_dec x y).
  - rewrite e. omega.
  - rewrite e. apply Nat.max_r_iff in e. omega.
Qed.

Lemma lt_S_r_max x y: x ⩽ S (Init.Nat.max y x).
Proof.
  destruct (Nat.max_dec y x).
  - rewrite e. apply Nat.max_l_iff in e. omega.
  - rewrite e. omega.
Qed.

Lemma no_path_to_fresh : forall pi x tau, first_fresh_type tau ⩽ x → P tau pi
 ↪ = Some (? x) → False.
Proof.
  induction pi.
  - ainv. asimpl in H. omega.
  - ainv. destruct a; destruct tau; ainv.
    + assert (first_fresh_type tau1 ⩽ x).
      { transitivity (first_fresh_type (tau1 ↝ tau2)).
        - apply lt_S_l_max.
        - assumption.
      }
      apply (IHpi _ _ H0 H1).
    + assert (first_fresh_type tau2 ⩽ x).
      { transitivity (first_fresh_type (tau1 ↝ tau2)).
        - apply lt_S_r_max.
        - assumption.
      }
      apply (IHpi _ _ H0 H1).
Qed.

Lemma no_path_to_first_fresh : forall pi tau, P tau pi = Some (fresh_type tau)
 ↪ → False.
Proof.
  intros.
  destruct (fresh_type tau) eqn:Hft.
  - unfold fresh_type in Hft. ainv.
    apply (no_path_to_fresh pi (first_fresh_type tau) tau (Nat.le_refl _) H1).
  - ainv.
```

```
Qed.

Lemma ts_cl_list_nil {A} : forall (pi pi' :A), ts_cl_list [] pi pi' -> False.
Proof.
  intros.
  induction X;ainv.
Qed.

Lemma R_tau_replace_one tau pi tau' a pr :
  P_ok tau pi pr = ? a ->
  replace_at_path (fresh_type tau) tau pi = tau' ->
  R_tau_list tau = R_tau_list tau'.
Proof.
  intros.
  unfold R_tau_list. Abort.

Lemma P_replace_at_path : forall tau pi b pr,  P_ok (replace_at_path b tau pi) pi pr = b.
Proof.
  induction tau.
  - intros. asimpl in pr. destruct pi.
    + reflexivity.
    + destruct d; ainv.
  - asimpl. intros. destruct pi.
    + reflexivity.
    + destruct d.
      * simpl. apply IHtau1.
      * simpl. apply IHtau2.
Qed.

(* General *)
Lemma filter_NoDup {A} : forall (l : list A) fb, NoDup l -> NoDup (filter fb l).
Proof.
  induction l.
  - intros. simpl. assumption.
  - intros. simpl. destruct (fb a).
    + inversion H. subst. constructor.
      * intros F.  apply filter_In in F. destruct F. contradiction.
      * apply IHl. assumption.
    + inversion H. apply IHl. assumption.
Qed.

Lemma NoDup_map_cons_iff {A} : forall l (x : A), NoDup (map (cons x) l) <-> NoDup l.
Proof.
  split.
  - intros. eapply NoDup_map_inv. exact H.
  - intros. induction l.
    + constructor.
    + inv H. simpl. constructor.
      * intros F. apply in_map_iff in F. destruct F as [x' [Heq HIn]].
        ainv. apply H2. constructor. reflexivity.
      * apply IHl. assumption.
Qed.
```

```
Lemma NoDup_app_disjoint {A} : forall l l' , (forall (a :A), In a l → ~(In a l')) /\ (forall a, In a l' → ~(In a l)) →
                                NoDup l → NoDup l' → NoDup (l ++ l').
Proof.
  induction l.
  - ainv.
  - intros. destruct H. simpl. pose proof H a (in_eq _ _).
    constructor.
    + intros F. inv H0. apply in_app_or in F. destruct F.
      * contradiction.
      * contradiction.
    + apply IHl.
      * split.
        -- intros. apply H. constructor 2. assumption.
        -- intros. apply H2 in H4. intros F. apply H4. constructor 2. assumption.
      * inv H0. assumption.
      * assumption.
Qed.


(* General Paths *)
Lemma dom_P_NoDup tau : NoDup (dom_P tau).
Proof.
  induction tau.
  - simpl. constructor.
    + intros F. inv F.
    + constructor.
  - simpl. constructor.
    + intros F. apply in_app_or in F. destruct F.
      * apply in_map_iff in H. destruct H as [x [F _]].
        discriminate F.
      * apply in_map_iff in H. destruct H as [x [F _]].
        discriminate F.
    + apply NoDup_app_disjoint.
      * split; intros.
        -- apply in_map_iff in H. destruct H as [x [Heq HIn]]. subst. intros F. apply in_map_iff in F.
           destruct F as [x0[Heq Hin]]. discriminate Heq.
        -- apply in_map_iff in H. destruct H as [x [Heq HIn]]. subst. intros F. apply in_map_iff in F.
           destruct F as [x0[Heq Hin]]. discriminate Heq.
      * apply NoDup_map_cons_iff. assumption.
      * apply NoDup_map_cons_iff. assumption.
Qed.

Lemma replaceable_nodup : forall tau m pi, NoDup (replaceable_paths tau m pi).
Proof.
  intros.
  apply filter_NoDup.
  apply dom_P_NoDup.
Qed.

Lemma replace_all_arrow : forall pis tau sigma b,  (~ In [] pis) →
```

```
                                        {tau' & {sigma' & replace_all_paths (tau ⇝ sigma) pis (? b) = tau' ⇝ sigma'}}.
Proof.
  induction pis.
  - intros. simpl. eexists. eexists. reflexivity.
  - intros. induction a.
    + exfalso. apply H. constructor. reflexivity.
    + destruct a.
      * simpl. apply IHpis. intros F. apply H.
        constructor 2. assumption.
      * simpl. apply IHpis. intros F. apply H.
        constructor 2. assumption.
Qed.
(*
Lemma replace_all_arrow2 : forall pis tau sigma b, {tau' & {sigma' & replace_all_paths (tau ⇝ sigma) pis (? b) = tau' ⇝ sigma'}} +
                                        {replace_all_paths (tau ⇝ sigma) pis (? b) = tau ⇝ sigma} +
                                        {replace_all_paths (tau ⇝ sigma) pis (? b) = (? b)}.
Proof.
  induction pis.
  - intros. simpl. left. right. reflexivity.
  - intros. destruct a.
    + asimpl. right. pose proof replace_all_var. unfold replace_all_paths in H. apply H.
    + destruct (IHpis tau sigma b).
      * admit.
      * right. simpl.

    destruct d.
      * simpl. edestruct IHpis.
        apply IHpis. intros F. apply H.
        constructor 2. assumption.
      * simpl. apply IHpis. intros F. apply H.
        constructor 2. assumption.
Qed.*)

Inductive is_prefix {A} : list A → list A → Type :=
| is_prefix_direct l a : is_prefix [] (a::l)
| is_prefix_cons l l' a : is_prefix l l' → is_prefix (a::l) (a::l').

Lemma is_prefix_nil_l {A} (l : list A) : is_prefix [] l + {l = []}.
Proof.
  destruct l.
  - right. reflexivity.
  - left. constructor.
Defined.

Lemma is_prefix_dec {A} {eqdec : EqDec A eq} : forall (l l' : list A), is_prefix l l' + (is_prefix l l' → False).
Proof.
  induction l.
  - intros. destruct (is_prefix_nil_l l').
    + left. assumption.
```

```
    + subst. right. intros F. inv F.
  - intros. destruct l'.
    + right. intros. inv X.
    + destruct (a == a0).
      * rewrite e. destruct (IHl l').
        -- left. constructor. assumption.
        -- right. intros. inv X. contradiction.
      * right. intros. inv X. apply c. reflexivity.
Defined.

Lemma prefix_in_list_dec {A} {eqdec : EqDec A eq} : forall (l : list (list A)) pi,
    {pi' & (In pi' l * is_prefix pi' pi)%type} + {forall pi', (In pi' l * is_prefix pi' pi)%type -> False}.
Proof.
  induction l.
  - intros. right. intros pi' [Hin Hpr]. inv Hin.
  - intros. destruct (IHl pi) as [[pi' [Hin Hpr]]|Hpr].
    + left. exists pi'. split. constructor 2. assumption. assumption.
    + destruct (is_prefix_dec a pi).
      * left. exists a. split. constructor. reflexivity. assumption.
      * right. intros pi' [[Heq|Hin] Hpr'].
        -- subst. contradiction.
        -- apply Hpr with (pi' := pi'). split; assumption.
Qed.

Lemma replace_prefix : forall pi' pi tau b, is_prefix pi' pi -> P (replace_at_path (? b) tau pi') pi = None.
Proof.
  induction pi'.
  - intros. inv H. simpl. destruct a; reflexivity.
  - intros. simpl. destruct a.
    + destruct tau.
      * inv H. reflexivity.
      * inv H. simpl. apply IHpi'. assumption.
    + destruct tau.
      * inv H. reflexivity.
      * inv H. simpl. apply IHpi'. assumption.
Qed.

Lemma replace_not_in_dom : forall pi tau,
  ~ In pi (dom_P tau) -> forall pi' b,  ~ In pi (dom_P (replace_at_path (? b) tau pi')).
Proof.
  intros.
  intros F.
  apply H.
  clear H.
  generalize dependent tau.
  revert pi.
  induction pi'.
  - intros. simpl in F. destruct F. subst. apply dom_P_nil. ainv.
  - intros. asimpl in F. destruct a.
    + destruct tau.
      * assumption.
      * simpl. simpl in F. destruct F.
```

```
        -- left. assumption.
        -- right. apply in_app_or in H. apply in_or_app.
           destruct H.
           ++ left. destruct pi.
              ** exfalso. induction (dom_P (_)).
                 --- ainv.
                 --- simpl in H. destruct H.
                     +++ discriminate H.
                     +++ contradiction.
              ** destruct d.
                 --- apply in_map_cons_iff. apply in_map_cons_iff in H. apply
                  ↪  IHpi'. assumption.
                 --- induction (dom_P (_)).
                     +++ ainv.
                     +++ simpl in H. destruct H.
                         *** discriminate H.
                         *** apply IHl. assumption.
           ++ right. assumption.
    + destruct tau.
      * assumption.
      * simpl. simpl in F. destruct F.
        -- left. assumption.
        -- right. apply in_app_or in H. apply in_or_app.
           destruct H.
           ++ left. assumption.
           ++ right. destruct pi.
              ** exfalso. induction (dom_P (_)).
                 --- ainv.
                 --- simpl in H. destruct H.
                     +++ discriminate H.
                     +++ contradiction.
              ** destruct d.
                 --- induction (dom_P (_)).
                     +++ ainv.
                     +++ simpl in H. destruct H.
                         *** discriminate H.
                         *** apply IHl. assumption.
                 --- apply in_map_cons_iff. apply in_map_cons_iff in H. apply
                  ↪  IHpi'. assumption.
Qed.

Lemma replace_at_path_var : forall pi a b, {c & replace_at_path (? a) (? b) pi =
 ↪  ? c}.
Proof.
  destruct pi.
  - intros. eexists. simpl. reflexivity.
  - intros. eexists. simpl. destruct d; reflexivity.
Qed.

Definition replace_all_paths2 tau pis b := fold_right (fun pi tau ⇒
 ↪  replace_at_path b tau pi) tau pis.
Lemma sz_subst_is_fresh : forall pi' tau m pi pr,
    R_m_ts m pi pi' →
    P_ok (replace_all_paths tau (replaceable_paths tau m pi) (fresh_type tau))
     ↪  pi' pr = fresh_type tau.
Proof.
```

```
  (*induction pi'.
  - intros. asimpl. unfold replace_all_paths.
    assert (In [] (replaceable_paths tau m pi)). {
      unfold replaceable_paths. apply filter_In. split.
      - apply dom_P_nil.
      - unfold replaceable_paths_cond. destruct (R_m_ts_dec m pi []).
        + reflexivity.
        + contradiction.
    }
    eapply nil_in_replace_paths in H. exact H.
  - intros. unfold fresh_type in *. destruct a.
    + destruct tau.
      * pose proof replace_all_var_is_var (replaceable_paths (? x) m pi) (first_fresh_type (? x)) x as [d HR].
        pose proof pr as pr2. rewrite HR in pr2. simpl in pr2. destruct pr2; ainv.
      * destruct (in_dec (list_eqdec dir_eqdec) [] (replaceable_paths (tau1 ⇝ tau2) m pi)).
        -- exfalso. eapply nil_in_replace_paths in i.  rewrite i in pr. simpl in pr. destruct pr; ainv.
        --
          epose proof replace_all_arrow _ tau1 tau2 _ n as [tau' [sigma' Heq]]. rewrite P_ok_P.
           rewrite Heq. simpl.
           destruct (replaceable_paths (tau1 ⇝ tau2) m pi == []).
           ++ rewrite e in *. simpl in Heq. injection Heq. intros. subst.
        simpl.

             ainv.
      unfold replace_all_paths.

    unfold replaceable_paths.
    unfold replaceable_paths_cond.
    asimpl. ainv. destruct a.
    + simpl. iapply IHpi'.
    asimpl.*)
  Admitted.




Lemma sechsundzwanzig_aux : forall tau pi m,
  Rsub (R_m_ts m) (R_tau_ts tau) →
  Rsub (R_m_ts m) (R_tau_ts (replace_all_paths tau (replaceable_paths tau m pi) (fresh_type tau))).
Proof.
  (*
  unfold Rsub.
  intros.
  unfold R_m_ts in *.
  destruct (R_m m) eqn:HRmm.
  - pose proof (ts_cl_list_dec l pi).
    destruct (X1 pi0).
    + assert (ts_cl_list l pi pi').
      {
```

```
        econstructor 3. apply t. assumption.
      }
      assert (P (replace_all_paths tau (replaceable_paths tau m pi) (fresh_type tau)) pi0 =
              P (replace_all_paths tau (replaceable_paths tau m pi) (fresh_type tau)) pi').
      {

      }
  - ainv.

  pose proof (rdec pi).
  destru*)
  Admitted.
Lemma sechsundzwanzig : forall m tau pi pr a, nfty_long [] m tau → P_ok tau pi pr = ? a →
                                          nfty_long [] m (replace_all_paths tau (replaceable_paths tau m pi) (fresh_type tau)).
Proof.
  (*
  intros.
  pose proof long_to_sfc_taui X.
  assert (Rsub (R_m_ts m) (R_tau_ts (replace_all_paths tau (replaceable_paths tau m pi) (fresh_type tau)))).
  {
    ainv.
    revert pi pr a H X0.
    induction X.
    - ainv. asimpl. unfold Rsub. intros. unfold replaceable_paths_cond.
      unfold Rsub in X0.


    apply Long_closed in X.
    revert m X0 X.
    induction tau.
    - asimpl. ainv.
      destruct m.
      + asimpl in H1. ainv. prooflater.
      + prooflater.
    -

        unfold max_fvar in H1. inversion H1. unfold R_tau_ts in X0. asimpl in X0.
      unfold Rsub in X0. unfold fresh_type. asimpl.
      unfold replaceable_paths_cond. simpl. unfold R_m_ts_dec.
      unfold R_m_ts in X0.
      destruct (R_m m).
      destruct m.
      simpl dom_P in pr.
      pose proof pr.
      apply In_head_set in H.
      destruct H.
      + subst. ainv. asimpl. unfold R_m_ts in X0.
        epose proof (fun pi pi' X ⇒ ts_cl_list_nil pi pi' (X0 pi pi' X)).

```

```
      + admit.
      + asimpl in *.

      assert ({x1 & m = !! x1 @@ []}).
      {
        destruct m.
        - unfold R_m_ts in X0. asimpl in X0. eexists.
      }

      rewrite nth_error_nil in H1. discriminate H1.
    - intros.

    ainv.
    revert H. admit. (*
    clear ...
    revert pi pr a.
    induction tau.
    - intros. asimpl. ainv.
    - intros. unfold R_tau_list. *)}
  pose proof (Long_closed _ _ X) as Hclosed.
  pose proof (Rsub_m_tau_to_Long Hclosed X1).
  assumption.*)
Admitted.
```