Paths.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 Omega.

Require Import PrincInh.Types.
Require Import PrincInh.Utils.

Import ListNotations.
Import EqNotations.

Inductive dir :Type :=
| Src
| Tgt
.

Instance dir_eqdec : EqDec dir eq.
Proof.
  unfold EqDec.
  intros.
  destruct x; destruct y;
    try (left; reflexivity);
    try (right; intros F; discriminate F).
Defined.

Definition path :Type := list dir.

Hint Rewrite (@nth_error_nil path) app_nil_l app_nil_r.

Fixpoint P (rho:type) (pi: path) {struct pi} : option type :=
  match pi with
  | [] ⇒ Some rho
  | Src :: pi' ⇒ match rho with
               | (? x) ⇒ None
               | sigma ⇝ _ ⇒ P sigma pi'
               end
  | Tgt :: pi' ⇒ match rho with
               | (? x) ⇒ None
               | _ ⇝ tau ⇒ P tau pi'
               end
  end.

Fixpoint P_default (rho:type) (def : type) (pi: path) {struct pi} : type :=
  match pi with
  | [] ⇒ rho
  | Src :: pi' ⇒ match rho with
               | (? x) ⇒ def
               | sigma ⇝ _ ⇒ P_default sigma def pi'
               end
  | Tgt :: pi' ⇒ match rho with
               | (? x) ⇒ def
               | _ ⇝ tau ⇒ P_default tau def pi'
```

```
          end
  end.

Fixpoint dom_P (rho: type) : list path :=
  match rho with
  | ? x => [[]]
  | sigma ~> tau => [] :: map (cons Src) (dom_P sigma) ++ map (cons Tgt) (dom_P tau)
  end.

Lemma dom_P_some : forall pi rho, In pi (dom_P rho) ->
                             { tau & P rho pi = Some tau}.
Proof.
  induction pi.
  - intros. exists rho. destruct rho; reflexivity.
  - intros. destruct a.
    + destruct rho.
      * simpl in H. exfalso. ainv.
      * simpl in H. simpl. apply IHpi.  destruct H as [F | H]. {inversion F. } apply in_app_or in H as [H|H];
        apply in_map_iff in H as [pi' [H1 H2]]; ainv.
    + destruct rho.
      * simpl in H. exfalso. ainv.
      * simpl in H. simpl. apply IHpi. destruct H as [F | H]. {inversion F. } apply in_app_or in H as [H|H];
        apply in_map_iff in H as [pi' [H1 H2]]; ainv.
Qed.

Lemma dom_P_none : forall pi rho, ~ In pi (dom_P rho) -> P rho pi = None.
Proof.
  induction pi.
  - intros. exfalso. apply H. destruct rho; simpl; left; reflexivity.
  - destruct a.
    + intros. simpl. destruct rho.
      * reflexivity.
      * apply IHpi. simpl in H. intros H1. apply H. right. apply in_or_app. left.
        apply in_map. assumption.
    + intros. simpl. destruct rho.
      * reflexivity.
      * apply IHpi. simpl in H. intros H1. apply H. right. apply in_or_app. right.
        apply in_map. assumption.
Qed.

Lemma dom_P_false : forall pi' d x, In (d :: pi') (dom_P (? x)) -> False.
Proof.
 ainv.
Qed.

Lemma dom_P_Src {pi sigma tau} : In (Src :: pi) (dom_P (sigma ~> tau)) -> In pi (dom_P sigma).
  intros. asimpl in H. destruct H. discriminate H. apply In_app_sumbool in H. destruct H.
    + apply in_map_cons in i. assumption.
    + exfalso. apply in_map_cons_not in i. apply i. intros F. discriminate F.
```

```
Qed.

Lemma dom_P_Src_iff {pi sigma tau} : In (Src :: pi) (dom_P (sigma ~> tau)) <-> In pi (dom_P sigma).
Proof.
  split; intros.
  - apply dom_P_Src in H. assumption.
  - asimpl. right. apply in_or_app. left. apply in_map_cons_iff. assumption.
Qed.

Lemma dom_P_Tgt {pi sigma tau} : In (Tgt :: pi) (dom_P (sigma ~> tau)) -> In pi (dom_P tau).
   intros. asimpl in H. destruct H. discriminate H. apply In_app_sumbool in H. destruct H.
    + exfalso. apply in_map_cons_not in i. apply i. intros F. discriminate F.
    + apply in_map_cons in i. assumption.
Qed.

Lemma dom_P_Tgt_iff {pi sigma tau} : In (Tgt :: pi) (dom_P (sigma ~> tau)) <-> In pi (dom_P tau).
Proof.
  split; intros.
  - apply dom_P_Tgt in H. assumption.
  - asimpl. right. apply in_or_app. right. apply in_map_cons_iff. assumption.
Qed.

Lemma dom_P_last : forall rho pi d, In (pi ++ [d]) (dom_P rho) -> In pi (dom_P rho).
Proof.
  induction rho; intros.
  - pose proof (app_cons_not_nil pi [] d). inversion H; try contradiction.
  - asimpl. destruct pi.
    + left. reflexivity.
    + right. apply in_or_app. destruct d0.
      * left. apply in_map_cons_iff. eapply IHrho1. eapply dom_P_Src. exact H.
      * right. apply in_map_cons_iff. eapply IHrho2. eapply dom_P_Tgt. exact H.
Qed.

Lemma dom_P_prefix : forall pi' pi rho, In (pi ++ pi') (dom_P rho) -> In pi (dom_P rho).
Proof.
  induction pi' using rev_ind.
  - intros. rewrite app_nil_r in H. assumption.
  - intros. rewrite app_assoc in H. apply dom_P_last in H. apply IHpi'. assumption.
Qed.

Lemma P_prefix {rho pi pi' tau}: P rho (pi ++ pi') = Some tau -> {tau' & P rho pi = Some tau'}.
Proof.
  intros.
  revert rho tau pi' H.
  induction pi.
  - intros. exists rho. reflexivity.
  - intros. simpl. destruct a; destruct rho; try discriminate H;
    simpl in H; eapply IHpi; exact H.
```

```
Qed.

Lemma dom_P_nil : forall rho, In [] (dom_P rho).
Proof.
  destruct rho; simpl; left; reflexivity.
Qed.

Definition P_ok rho pi (proof : In pi (dom_P rho)) : type.
  revert rho pi proof.
  fix dummy 2. intros.
  destruct pi.
  - exact rho.
  - destruct rho.
    + exfalso. exact (dom_P_false _ _ _ proof).
    + destruct d.
      * exact (dummy rho1 pi (dom_P_Src proof)).
      * exact (dummy rho2 pi (dom_P_Tgt proof)).
Defined.

Lemma P_ok_Src : forall sigma tau pi pr, P_ok (sigma ⇝ tau) (Src::pi) pr = P_ok (sigma) pi (dom_P_Src pr).
Proof.
  reflexivity.
Qed.

Lemma P_ok_Tgt : forall sigma tau pi pr, P_ok (sigma ⇝ tau) (Tgt::pi) pr = P_ok tau pi (dom_P_Tgt pr).
Proof.
  reflexivity.
Qed.

Lemma P_ok_proof_irl : forall rho pi p1 p2, P_ok rho pi p1 = P_ok rho pi p2.
Proof.
  induction rho.
  - intros. destruct pi.
    + reflexivity.
    + inversion p1. discriminate H. inversion H.
  - intros. destruct pi.
    + reflexivity.
    + destruct d.
      * rewrite P_ok_Src. rewrite  P_ok_Src. apply IHrho1.
      * rewrite P_ok_Tgt. rewrite  P_ok_Tgt. apply IHrho2.
Qed.

Lemma P_ok_P {rho pi tau pr}: P_ok rho pi pr = tau ↣ P rho pi = Some tau.
Proof.
  split.
  - revert rho tau pr. induction pi.
    + simpl. intros rho tau _. apply some_eq.
    + simpl. intros. destruct rho.
      * inversion pr. discriminate H0. inversion  H0.
      * destruct a; eapply IHpi; exact H.
  - revert rho tau pr. induction pi.
    + simpl. intros rho tau _ eq. apply some_eq. exact eq.
    + simpl. intros. destruct rho.
      * destruct a; discriminate H.
```

```
      * destruct a.
        ** eapply IHpi in H. apply H.
        ** eapply IHpi in H. apply H.
Qed.

Lemma P_ok_P_ex {rho pi tau}: (exists pr, P_ok rho pi pr = tau) <-> P rho pi = Some tau.
Proof.
  split.
  - revert rho tau. induction pi.
    + simpl. intros. destruct H. subst. reflexivity.
    + simpl. intros. destruct rho.
      * destruct H as [pr H]. inversion pr. discriminate H0. inversion  H0.
      * destruct a.
        ** eapply IHpi. destruct H as [pr H]. exists (dom_P_Src pr). exact H.
        ** eapply IHpi. destruct H as [pr H]. exists (dom_P_Tgt pr). exact H.
  - revert rho tau. induction pi.
    + simpl. intros rho tau eq. exists (dom_P_nil rho). apply some_eq. exact eq.
    + simpl. intros. destruct rho.
      * destruct a; discriminate H.
      * destruct a.
        ** apply IHpi in H.
           destruct H as [pr H]. assert (In (Src::pi) (dom_P (rho1 ~> rho2))).
           {
             apply dom_P_Src_iff. assumption.
           }
           exists H0. rewrite (P_ok_proof_irl _ _ _ pr). assumption.
        ** apply IHpi in H.
           destruct H as [pr H]. assert (In (Tgt::pi) (dom_P (rho1 ~> rho2))).
           {
             apply dom_P_Tgt_iff. assumption.
           }
           exists H0. rewrite (P_ok_proof_irl _ _ _ pr). assumption.
Qed.

Lemma P_P_ok_set {rho pi tau}: P rho pi = Some tau -> { pr & P_ok rho pi pr = tau}.
Proof.
  - revert rho tau. induction pi.
    + simpl. intros rho tau eq. exists (dom_P_nil rho). apply some_eq. exact eq.
    + simpl. intros. destruct rho.
      * destruct a; discriminate H.
      * destruct a.
        ** apply IHpi in H.
           destruct H as [pr H]. assert (In (Src::pi) (dom_P (rho1 ~> rho2))).
           {
             apply dom_P_Src_iff. assumption.
           }
           exists H0. rewrite (P_ok_proof_irl _ _ _ pr). assumption.
        ** apply IHpi in H.
           destruct H as [pr H]. assert (In (Tgt::pi) (dom_P (rho1 ~> rho2))).
           {
             apply dom_P_Tgt_iff. assumption.
           }
           exists H0. rewrite (P_ok_proof_irl _ _ _ pr). assumption.
Qed.
```

```
Definition make_tgt_path (pi: path) (n : nat) :=
  pi ++ (repeat Tgt n) ++ [Src].

Definition even_ones pi := Nat.Even (count_occ dir_eqdec pi Src).

Lemma even_ones_pump pi : even_ones pi = even_ones (pi ++ [Tgt]).
Proof.
  unfold even_ones.
  rewrite count_occ_last_neq.
  - reflexivity.
  - isfalse.
Qed.

Definition odd_repo (Delta : list path) := Forall (fun pi => Nat.Odd (count_occ dir_eqdec pi Src)) Delta.

Lemma odd_repo_head (Delta : list path) : forall pi, odd_repo (pi :: Delta) -> Nat.Odd (count_occ dir_eqdec pi Src).
Proof.
  intros.
  unfold odd_repo in H.
  inv H.
  assumption.
Qed.

Lemma odd_repo_split Delta : forall pi,
  ~ odd_repo (pi :: Delta) ->
  odd_repo Delta ->
  ~ Nat.Odd (count_occ dir_eqdec pi Src).
Proof.
  intros.
  unfold odd_repo in *.
  intros F.
  apply H.
  constructor; assumption.
Qed.

Lemma odd_repo_comb Delta : forall pi,
    odd_repo Delta ->
    Nat.Odd (count_occ dir_eqdec pi Src) ->
    odd_repo (pi :: Delta).
Proof.
  intros. unfold odd_repo. constructor; assumption.
Qed.

Lemma odd_repo_head_eq (Delta : list path) : forall pi pi', (odd_repo (pi :: Delta) -> odd_repo (pi' :: Delta)) ->
odd_repo Delta -> (Nat.Odd (count_occ dir_eqdec pi Src) -> Nat.Odd (count_occ dir_eqdec pi' Src)).
Proof.
  intros.
  eapply odd_repo_head.
  apply H.
  apply odd_repo_comb; assumption.
Qed.
```

```
Lemma odd_repo_head_eq2 Delta : forall pi pi',
    (Nat.Odd (count_occ dir_eqdec pi Src) → Nat.Odd (count_occ dir_eqdec pi' Src)) →
                                             odd_repo (pi :: Delta) → odd_repo (pi' :: Delta).
Proof.
  intros.
  unfold odd_repo in H0.
  unfold odd_repo .
  constructor.
  - apply H. ainv.
  - inversion H0. assumption.
Qed.

Lemma odd_repo_head_tail (Delta : list path) : forall pi, odd_repo ((pi ++ [Src]) :: Delta) ↣ odd_repo ((Src :: pi) :: Delta).
Proof.
  intros.
  split.
  - apply odd_repo_head_eq2. simpl.
    rewrite count_occ_split. simpl. rewrite Nat.add_comm. simpl. auto.
  - apply odd_repo_head_eq2. simpl.
    rewrite count_occ_split. simpl. rewrite Nat.add_comm. simpl. auto.
Qed.

Lemma tgt_path_even_if_pi_odd : forall n pi, Nat.Odd (count_occ dir_eqdec pi Src) →
  Nat.Even (count_occ dir_eqdec (make_tgt_path pi n) Src).
Proof.
  unfold make_tgt_path. simpl. intros n pi.
  repeat rewrite count_occ_split. simpl. rewrite Nat.add_comm. revert n pi.
  induction n.
  - intros. simpl.
    apply Nat.Even_succ. assumption.
  - intros. simpl. apply IHn. assumption.
Qed.

Lemma tgt_path_even_if_delta_odd: forall (Delta : list path) pi n,
    odd_repo Delta → In pi Delta →
    even_ones (make_tgt_path pi n).
Proof.
  intros.
    apply tgt_path_even_if_pi_odd.
    unfold odd_repo in H.
    rewrite Forall_forall in H. apply H. assumption.
Qed.

Lemma P_src {rho pi sigma tau} : P rho pi = Some (sigma ⇝ tau) → P rho (pi ++ [Src]) = Some sigma.
Proof.
  revert pi rho sigma tau. induction pi; intros.
  - simpl in H. apply some_eq in H. subst. reflexivity.
  - destruct a; simpl in H; destruct rho; try discriminate H; simpl; eapply IHpi; apply H.
Qed.
```

```
Lemma P_tgt {rho pi sigma tau} : P rho pi = Some (sigma ~> tau) -> P rho (pi ++ [Tgt]) = Some tau.
Proof.
  revert pi rho sigma tau. induction pi; intros.
  - simpl in H. apply some_eq in H. subst. reflexivity.
  - destruct a; simpl in H; destruct rho; try discriminate H; simpl; eapply IHpi; apply H.
Qed.

Lemma P_app_split {pi pi' rho rho' rho''}: P rho pi = Some rho' -> P rho' pi' = Some rho'' -> P rho (pi ++ pi') = Some rho''.
Proof.
  revert pi' rho rho' rho''.
  induction pi.
  - ainv.
  - intros. asimpl. destruct a.
    + asimpl in *. destruct rho; try discriminate H. eapply IHpi. apply H. apply H0.
    + asimpl in *. destruct rho; try discriminate H. eapply IHpi. apply H. apply H0.
Qed.

Lemma P_app_proof {a ts rho pi} :
  P rho pi = Some (make_arrow_type ts a) ->
  forall n (pr: n < length ts), P rho (pi ++ (repeat Tgt n ++ [Src])) = Some (nth_ok ts n pr).
Proof.
  intros.
  eapply P_app_split. apply H.
  clear H rho pi.
  revert ts n pr a.
  induction ts.
  - ainv.
  - intros. destruct n.
    + reflexivity.
    + simpl. simpl in pr. pose proof (lt_S_n _ _ pr).
      erewrite nth_ok_proof_irel.
      apply (IHts n H).
Qed.

Lemma P_app_proof_in {rho pi a ts} : P rho pi = Some (make_arrow_type ts a) ->
                                              forall n (pr: n < length ts),
                                              In (pi ++ repeat Tgt n ++ [Src]) (dom_P rho).
Proof.
  intros. apply (@P_app_proof a ts rho pi)  with (n:=n) (pr:=pr) in H.
  apply P_ok_P_ex in H. destruct H as [pr0 _]. assumption.
Qed.

Lemma dom_P_head_Src : forall pi rho, In (Src :: pi) (dom_P rho) -> {rho1 & {rho2 & rho = rho1 ~> rho2 /\ In pi (dom_P rho1)}}.
Proof.
  intros pi [[t1 t2] H.
  - exfalso. ainv.
```

```
  - exists t1. exists t2. apply dom_P_Src in H. split. reflexivity. assumption.
Qed.

Lemma dom_P_head_Tgt : forall pi rho, In (Tgt :: pi) (dom_P rho) -> {rho1 & {rho2 & rho = rho1 ~> rho2 /\ In pi (dom_P rho2)}}.
Proof.
  intros pi [|t1 t2] H.
  - exfalso. ainv.
  - exists t1. exists t2. apply dom_P_Tgt in H. split. reflexivity. assumption.
Qed.

Lemma dom_P_Src_to_Tgt : forall pi rho dir1 dir2, In (pi ++ [dir1]) (dom_P rho) -> In (pi ++ [dir2]) (dom_P rho).
Proof.
  induction pi.
  - intros. destruct rho. ainv. destruct dir2; (asimpl; right; apply in_or_app).
    + left. apply in_map_cons_iff. apply dom_P_nil.
    + right. apply in_map_cons_iff. apply dom_P_nil.
  - simpl. intros. destruct a.
    + apply dom_P_head_Src in H. destruct H as [t1 [t2 [Hrho HIn]]]. subst.
      simpl. right. apply in_or_app. left. apply in_map_cons_iff. eapply IHpi. apply HIn.
    + apply dom_P_head_Tgt in H. destruct H as [t1 [t2 [Hrho HIn]]]. subst.
      simpl. right. apply in_or_app. right. apply in_map_cons_iff. eapply IHpi. apply HIn.
Qed.

Lemma P_ok_Src_to_Tgt : forall pi rho dir1 dir2 pr1 sigma, P_ok rho (pi ++ [dir1]) pr1 = sigma ->
                                                    {pr2 & {tau & P_ok rho (pi ++ [dir2]) pr2 = tau}}.
Proof.
  intros.
  pose proof dom_P_Src_to_Tgt _ _ _ dir2 pr1.
  exists H0. pose proof dom_P_some _ _ H0 as [tau HP]. exists tau.
  apply P_ok_P. assumption.
Qed.

Lemma P_ok_make_arrow : forall ts a, {pr & P_ok (make_arrow_type ts a) (repeat Tgt (length ts)) pr = a}.
Proof.
  intros.
  induction ts.
  - intros. simpl. exists (dom_P_nil _). reflexivity.
  - simpl. destruct IHts as [pr IHts].
    eexists. erewrite P_ok_proof_irl. exact IHts.
    Unshelve.
    right.
    apply in_or_app. right. apply in_map_cons_iff. assumption.
Qed.

Lemma P_Src2 : forall pi rho sigma, P rho (pi ++ [Src]) = Some sigma -> {tau & P rho pi = Some (sigma ~> tau) /\
                                                                          P rho (pi ++ [Tgt]) = Some tau}.
```

```
Proof.
  induction pi.
  - intros. rewrite app_nil_l in H. inversion H. destruct rho eqn:Hrho; try discriminate H1.
    simpl. apply some_eq in H1. subst. exists t2. split; reflexivity.
  - intros. destruct a.
    + simpl. destruct rho eqn:Hrho; try discriminate H. apply IHpi.
      simpl in H. assumption.
    + simpl. destruct rho eqn:Hrho; try discriminate H. apply IHpi.
      simpl in H. assumption.
Qed.

Lemma P_Tgt2 : forall pi rho tau, P rho (pi ++ [Tgt]) = Some tau-> {sigma & P rho pi = Some (sigma ~> tau) /\
                                                                       P rho (pi ++ [Src]) = Some sigma}.
Proof.
  induction pi.
  - intros. rewrite app_nil_l in H. inversion H. destruct rho eqn:Hrho; try discriminate H1.
    simpl. apply some_eq in H1. subst. exists t1. split; reflexivity.
  - intros. destruct a.
    + simpl. destruct rho eqn:Hrho; try discriminate H. apply IHpi.
      simpl in H. assumption.
    + simpl. destruct rho eqn:Hrho; try discriminate H. apply IHpi.
      simpl in H. assumption.
Qed.

Lemma P_path_make_arrow_type {tau pi n rho}: P tau (pi ++ repeat Tgt n)  = Some rho ->
                                             {ts & P tau pi = Some (make_arrow_type ts rho) /\ length ts = n}.
Proof.
  revert pi rho tau.
  induction n; intros pi rho tau.
  - simpl. rewrite app_nil_r. intros. exists []. auto.
  - rewrite repeat_rev. intros. rewrite app_assoc in H. apply P_Tgt2 in H.
    destruct H as [sigma [HP1 HP2]].
    pose proof IHn _ _ _ HP1 as [ts [Hres HLen]].
    exists (ts ++ [sigma]). simpl. rewrite make_arrow_type_last.
    split. assumption. rewrite app_length. simpl. rewrite HLen. omega.
Qed.

Lemma make_arrow_type_dirs {tau ts a n}:
      make_arrow_type ts (? a) = tau ->
      P tau (repeat Tgt n ++ [Src]) = nth_error ts n.
Proof.
  revert ts tau.
  induction n.
  - intros. simpl. destruct tau.
    + pose proof make_arrow_type_ts_is_nil H as [Hts Hrho].
      subst. reflexivity.
    + destruct ts.
      * discriminate H.
      * simpl in H. injection H. intros. subst. reflexivity.
  - intros. asimpl. destruct tau.
```

```
    + pose proof make_arrow_type_ts_is_nil H as [Hts Hrho].
      subst. reflexivity.
    + destruct ts.
      * discriminate H.
      * apply IHn. injection H. intros. assumption.
Qed.

Fixpoint replace_at_path b tau pi {struct pi} : type :=
  match pi with
  | [] ⇒ b
  | Src :: pi' ⇒ match tau with
                 | (? _) ⇒ tau
                 | sigma ⇝ tau' ⇒ replace_at_path b sigma pi' ⇝ tau'
                 end
  | Tgt :: pi'  ⇒ match tau with
                 | (? _) ⇒ tau
                 | sigma ⇝ tau' ⇒ sigma ⇝ replace_at_path b tau' pi'
                  end
  end.
```