Filtr.v

```
Require Import Coq.Arith.PeanoNat.
Require Import Coq.Lists.List.
Require Import Autosubst.Autosubst.

Require Import PrincInh.Terms.
Require Import PrincInh.Types.
Require Import PrincInh.Typing.
Require Import PrincInh.Utils.

Import ListNotations.
Import EqNotations.

(* Given a type derivation D we define the set T(D) = { τ | Γ ⊢ M : τ is a judgement in D} *)

Inductive TD (Gamma : list type) : forall (m : term) (tau : type), ty_T Gamma m tau → type → Type :=
  | TD_refl m tau : forall (proof : ty_T Gamma m tau), TD Gamma m tau proof tau
  | TD_lam s sigma tau :
    forall (tau' : type) (innerproof : ty_T (sigma :: Gamma) s tau),
      TD (sigma :: Gamma) s tau innerproof tau'
      → TD Gamma (\_ s) (sigma ⇝ tau) (Ty_Lam Gamma s sigma tau innerproof) tau'
  | TD_app_l p q sigma tau :
    forall (tau' : type) (leftproof : ty_T Gamma p (sigma ⇝ tau)) (rightproof : ty_T Gamma q sigma),
      TD Gamma p (sigma ⇝ tau) leftproof tau'
         → TD Gamma (p @ q) tau (Ty_App Gamma p q sigma tau leftproof rightproof) tau'
  | TD_app_r p q sigma tau :
    forall (tau' : type) (leftproof : ty_T Gamma p (sigma ⇝ tau)) (rightproof : ty_T Gamma q sigma),
      TD Gamma q sigma rightproof tau'
         → TD Gamma (p @ q) tau (Ty_App Gamma p q sigma tau leftproof rightproof) tau'
.

Lemma app_eq_l {m1 m2 m3 m4} : m1 @ m2 = m3 @ m4 → m1 = m3.
Proof.
  intros. ainv.
Qed.

Lemma app_eq_r {m1 m2 m3 m4} : m1 @ m2 = m3 @ m4 → m2 = m4.
Proof.
  intros. ainv.
Qed.

Lemma lam_eq {s1 s2} : \_ s1 = \_ s2 → s1 = s2.
Proof.
  ainv.
Qed.

Lemma tyrewew {Gamma m1 m2 m1' m2' sigma tau}: forall (eq: m1 @ m2 = m1' @ m2') (proof1: ty_T Gamma m1' (sigma ⇝ tau)) (proof2: ty_T Gamma m2' sigma),
```

```
    Ty_App Gamma m1' m2' sigma tau proof1 proof2 =
    rew[fun m ⇒ ty_T Gamma m tau] eq in
      Ty_App Gamma m1 m2 sigma tau
             (rew  -[fun m ⇒ ty_T Gamma m (sigma ⇝ tau)] app_eq_l eq in proof1)
             (rew  -[fun m ⇒ ty_T Gamma m sigma] app_eq_r eq in proof2).
Proof.
  intros.
  revert proof1 proof2.
  inversion eq.
  revert eq.
  rewrite H1.
  rewrite H0.
  intro eq.
  rewrite ← (Coq.Logic.Eqdep_dec.UIP_dec eq_dec_term eq_refl eq).
  rewrite ← (Coq.Logic.Eqdep_dec.UIP_dec eq_dec_term eq_refl (app_eq_l eq_refl)).
  rewrite ← (Coq.Logic.Eqdep_dec.UIP_dec eq_dec_term eq_refl (app_eq_r eq_refl)).
  unfold eq_rect_r.
  reflexivity.
Qed.

Lemma False_Ty : forall (T : Type), False → T.
Proof.
  intros.
  exfalso.
  apply H.
Qed.

Fixpoint filtration (X : type → bool) (a : var) (rho : type) :=
  match rho with
  | ? b ⇒ ? a
  | sigma ⇝ tau ⇒ if ( andb (X (sigma ⇝ tau)) (X tau)) then
                      (filtration X a sigma) ⇝ (filtration X a tau)
                  else
                    ? a
  end.

Definition repo_filt (X : type → bool) (a : var) : repo → list type :=
  map (filtration X a).

Lemma filt_split : forall sigma tau X a, X tau = true → X (sigma ⇝ tau) = true
                       → filtration X a (sigma ⇝ tau) = (filtration X a sigma) ⇝ (filtration X a tau).
Proof.
  intros.
  unfold filtration. rewrite H. rewrite H0. reflexivity.
Qed.

Lemma repo_filt_split : forall X a rho Gamma, repo_filt X a (rho :: Gamma) = ((filtration X a rho) :: (repo_filt X a (Gamma))).
Proof.
  intros.
  unfold repo_filt.
  reflexivity.
```

```
Qed.

Fixpoint TD_f {Gamma m tau} (proof : ty_T Gamma m tau) : list type ≔
  match proof with
  | Ty_Var _ x rho eqproof ⇒ [rho]
  | Ty_Lam _ s sigma tau' innerproof ⇒ (sigma ⇝ tau') :: TD_f innerproof
  | Ty_App _ s t A B proof1 proof2 ⇒ B::(TD_f proof1 ++ TD_f proof2)
  end.


Lemma TD_last {Gamma m tau}: forall (proof : ty_T Gamma m tau), In tau (TD_f proof).
Proof.
  intros.
  destruct proof.
  - simpl. left. reflexivity.
  - simpl. left. reflexivity.
  - simpl. left. reflexivity.
Qed.



Lemma filter_deriv {m Gamma tau}: forall (X : type → bool) (proof : ty_T Gamma m tau),
    (forall tau', In tau' (TD_f proof) → X tau' = true) → (forall a, ty_T (repo_filt X a Gamma) m (filtration X a tau)).
Proof.
  intros.
  induction proof.
  - constructor. unfold repo_filt.
    apply map_nth_error. assumption.
  - rewrite filt_split.
    + constructor. rewrite ← repo_filt_split. apply IHproof. intros. apply H. simpl. right. assumption.
    + apply H. simpl. right. apply TD_last.
    + apply H. simpl. left. reflexivity.
  - econstructor.
    + instantiate (1≔filtration X a A). rewrite ← filt_split.
      * apply IHproof1. intros. apply H. simpl. right. apply in_or_app. left. assumption.
      * apply H. simpl. left. reflexivity.
      * apply H. simpl. right. apply in_or_app. left. apply TD_last.
    + apply IHproof2. intros. apply H. simpl. right. apply in_or_app. right. assumption.
Qed.

Lemma In_TD_dec {Gamma m tau} : forall (deriv : ty_T Gamma m tau) tau', {In tau' (TD_f deriv)} + {~(In tau' (TD_f deriv))}.
Proof.
  intros.
  apply In_dec.
  apply eq_dec_type.
Defined.

Definition TD_b {Gamma m tau} (deriv : ty_T Gamma m tau) tau' : bool ≔
  if (In_TD_dec deriv tau') then true else false.
```

```
Lemma TD_b_corr {Gamma m tau} {proof : ty_T Gamma m tau}: (forall tau' : type,
 ↪  In tau' (TD_f proof) → TD_b proof tau' = true) .
Proof.
  intros.
  unfold TD_b.
  destruct (In_TD_dec proof tau') eqn:HIn.
  + reflexivity.
  + contradiction.
Qed.

Lemma filt_mtTy :forall a X, repo_filt a X [] = [].
Proof.
 auto.
Qed.

Lemma subformula_subst : forall tau rho Su, subformula rho tau → subformula
 ↪  rho.[Su] tau.[Su].
Proof.
  induction 1.
  - constructor.
  - constructor. assumption.
  - constructor 3. assumption.
Qed.

Lemma subst_subformula : forall sigma tau rho, subformula (sigma ⇝ tau) rho →
 ↪  forall X a Su, rho.[Su] = filtration X a rho →
                                               (sigma ⇝ tau).[Su] = filtration X a
                                                ↪  (sigma ⇝ tau).
Proof.
  intros.
  remember (sigma ⇝ tau) as tau0.
  induction H.
  - ainv.
  - apply IHsubformula.
    + assumption.
    + inversion H0.
      destruct ((X (sigma0 ⇝ tau0) && X tau0)%bool).
      * ainv.
      * ainv.
  - apply IHsubformula.
    + assumption.
    + inversion H0.
      destruct ((X (sigma0 ⇝ tau0) && X tau0)%bool).
      * ainv.
      * ainv.
Qed.

Lemma princ_var : forall A x, princ A (! x) → False.
Proof.
  intros.
  unfold princ in X.
  destruct X.
  inv t.
  rewrite nth_error_nil in H0.
  ainv.
```

```
Qed.

Lemma filter_princ_nec : forall m rho (D : ty_T [] m rho) sigma tau, subformula (sigma ⇝ tau) rho → (TD_b D tau = false) → princ rho m → False.
Proof.
  intros.
  pose proof filter_deriv _ D TD_b_corr (first_fresh_type rho) as filtD.
  simpl in filtD.
  unfold princ in X.
  destruct X as [alsoD Hsubst].
  pose proof Hsubst _ filtD as [Su sucorr].
  pose proof subst_subformula sigma tau rho H (TD_b D) (first_fresh_type rho) Su sucorr.
  simpl in H1. rewrite H0 in H1. rewrite Bool.andb_false_r in H1. inversion H1.
Qed.
```