Typing.v

```
Require Import Coq.Lists.List.
Require Import Coq.Lists.ListSet.
Require Import Coq.Arith.Peano_dec.
Require Import Coq.Classes.EquivDec.
Require Import Coq.Logic.FunctionalExtensionality.

Require Import Autosubst.Autosubst.

Require Import PrincInh.Types.
Require Import PrincInh.Terms.
Require Import PrincInh.NFTerms.
Require Import PrincInh.Utils.

Import ListNotations.


(* Typing relations for terms and nfterms *)
Inductive ty_T (Gamma : repo) : term → type → Type:=
| Ty_Var x A : nth_error Gamma x = Some A →
        ty_T Gamma (Var x) A
| Ty_Lam s A B : ty_T (A :: Gamma) s B →
        ty_T Gamma (Lam s) (Arr A B)
| Ty_App s t A B : ty_T Gamma s (Arr A B) → ty_T Gamma t A →
                   ty_T Gamma (App s t) B.


Inductive nfty (Gamma : repo) : nfterm → type → Type :=
| NFTy_lam s sigma tau : nfty (sigma :: Gamma) s tau → nfty Gamma (\__ s)
 ↪ (sigma ⇝ tau)
| NFTy_var x tau ts ms : nth_error Gamma x = Some (make_arrow_type ts tau) →
                         length ms = length ts →
                         (forall n (pms : n < length ms) (pts : n < length ts),
                             nfty Gamma (nth_ok ms n pms) (nth_ok ts n pts)) →
             nfty Gamma ( !! x @@ ms) tau
.

Definition princ rho m: Type :=
  ty_T [] m rho * forall rho', ty_T [] m rho' → {Su & rho.[Su] = rho'}.


Lemma generation_app_T : forall s t tau (Gamma : repo), ty_T Gamma (s@t) tau →
                        {sigma & prod (ty_T Gamma s (sigma ⇝ tau))
                                  (ty_T Gamma t (sigma)) }.
Proof.
  intros s t tau Gamma H.
  inv H.
  exists A.
  split; assumption.
Qed.

Lemma generation_lam_T : forall s A (Gamma : repo) sigma tau, ty_T Gamma (\_ s)
 ↪ A →
                        A = sigma ⇝ tau →
```

```
                              ty_T (sigma :: Gamma) s tau.
Proof.
  intros.
  ainv.
Qed.

Lemma generation_var_T : forall x A (Gamma : repo), ty_T Gamma (! x) A →
                                nth_error Gamma x = Some A.
Proof.
  intros.
  ainv.
Qed.

Lemma ty_app_ex : forall (Gamma : repo) (B:type) s t, ty_T Gamma (App s t) B → { A & ty_T Gamma t A →
    ty_T Gamma s (A ↝ B) }.
Proof.
    intros. ainv. exists A. ainv.
Qed.

Fixpoint update_list {A} (l1 : list A) (Su : nat → option A) : list A ≔
  match l1 with
  | [] ⇒ []
  | x :: xs ⇒ match Su 0 with
              | Some a ⇒ a
              | None ⇒ x
              end :: update_list xs (fun x ⇒ (Su (S x)))
  end.

Lemma ty_ren_T Gamma s A:
  ty_T Gamma s A → forall Delta xi,
    (forall n, nth_error Gamma n = (xi >>> nth_error Delta) n) →
      ty_T Delta s.[ren xi] A.
Proof.
    induction 1.
    - constructor. subst. rewrite ← e. rewrite (H x). reflexivity.
    - intros. subst. asimpl. econstructor. eapply IHX. intros. simpl. destruct n.
      + ainv.
      + simpl. rewrite H. reflexivity.
    - intros. subst. asimpl. econstructor.
      + eapply IHX1. assumption.
      + eapply IHX2. assumption.
Qed.

Lemma ty_subst_T Gamma s A:
      ty_T Gamma s A → forall sigma Delta,
        (forall x t, nth_error Gamma x = Some t → ty_T Delta (sigma x) (t)) →
          ty_T Delta s.[sigma] A.
Proof.
  induction 1.
    - intros. simpl. apply X. assumption.
    - econstructor. eapply IHX. intros [ ⟧;
      asimpl; eauto using ty_T, ty_ren_T.
    - asimpl. eauto using ty_T.
Qed.
```

```
Definition has_ty (m: term) (tau: type) : Prop :=
    inhabited (ty_T [] m tau).

Example ident_typing : has_ty (\_ !0) (?0 ~> ?0).
Proof.
    unfold has_ty.
    constructor. constructor. constructor. reflexivity.
Qed.


Theorem subst_ty : forall (Gamma: repo) s A, ty_T Gamma s A ->
    forall (Su : var -> type), ty_T Gamma..[Su] s A.[Su].
Proof.
    intros.
    generalize dependent A.
    generalize dependent Gamma.
    induction s.
    - intros Gamma A. constructor. apply subst_repo_some. inversion X. assumption.
    - intros Gamma A. ainv. econstructor.
      + pose proof (IHs1 Gamma (A0 ~> A)). asimpl in X. apply X. assumption.
      + apply IHs2. eassumption.
    - intros Gamma A. ainv. constructor. rewrite subst_repo_cons. eapply IHs. assumption.
Qed.

Definition Typable (t:term) := exists tau Gamma, inhabited ( ty_T Gamma t tau ).

Theorem typable_subterm : forall m t, Typable t -> subterm m t -> Typable m.
Proof.
    intros.
    induction H0.
    - assumption.
    - apply IHsubterm. ainv. unfold Typable. exists (A ~> x). exists x0. constructor. assumption.
    - apply IHsubterm. ainv. unfold Typable. exists A. exists x0. constructor. assumption.
    - apply IHsubterm. ainv. unfold Typable. exists B. exists (A:: x0). constructor. assumption.
Qed.

Lemma mp_gen_T : forall Gamma ms x tau, ty_T Gamma (curry (!x) ms) tau ->
  { sigmas & prod (Forall2_T (ty_T Gamma) ms sigmas) (nth_error Gamma x = Some (make_arrow_type (sigmas) tau)) }.
Proof.
  induction ms using rev_ind_T.
  - intros. ainv. exists []. split.
    + constructor.
    + simpl. assumption.
  - intros. rewrite curry_tail in X. apply generation_app_T in X as [sigma [HArr Hsig]]. apply IHms in HArr as [sigmas0 [HForall HGamma]].
    exists (sigmas0 ++ [sigma]). split.
    + apply Forall2_head_to_last_T. constructor; assumption.
    + unfold make_arrow_type. rewrite fold_right_app.
```

```
      simpl. assumption.
Qed.

Lemma subst_var_is_var_T : forall Su a tau, ? a = tau.[Su] → { b & tau = ? b }.
Proof.
  induction tau.
  - simpl. intros. exists x. reflexivity.
  - simpl. intros. inversion H.
Qed.

Lemma subst_arr_is_arr_or_T : forall x t Su t0, x.[Su] = t ⇝ t0
    → ({st & { st0 &
          x = st ⇝ st0 /\ st.[Su] = t /\ st0.[Su] = t0 } }) +
       ({ a & x = ? a}).
Proof.
  intros. destruct x.
  - right. exists x. auto.
  - left. exists x1. exists x2.
    split.
    + reflexivity.
    + split; ainv.
Qed.
```