Types.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.Utils.

Import ListNotations.

Inductive type :=
| Atom (x: var)
| Arr (A B : type).

Instance Ids_type : Ids type. derive. Defined.
Instance Rename_type : Rename type. derive. Defined.
Instance Subst_type : Subst type. derive. Defined.
Instance SubstLemmas_type : SubstLemmas type. derive. Defined.

Notation "'?' x" := (Atom x) (at level 15).
Notation "a '~>' b" := (Arr a b) (at level 51, right associativity).

Definition repo := list type.

Instance eq_dec_type : EqDec type eq.
Proof.
    unfold EqDec.
    intros x.
    induction x.
    - destruct y.
      + destruct (x == x0); dec_eq.
      + right. intros H. ainv.
    - destruct y.
      + right. intros H. ainv.
      + destruct (IHx1 y1); dec_eq.
        { destruct (IHx2 y2); dec_eq. }
Defined.

Definition mtTy {A} : var -> option A := fun x => None.

Instance eq_dec_option : forall T, EqDec T eq -> EqDec (option T) eq.
Proof.
    unfold EqDec.
    intros.
    destruct x, y.
    - destruct (X t t0); dec_eq.
    - right. isfalse.
    - right. isfalse.
    - left. reflexivity.
Defined.

Lemma is_none_dec {T: Type} : forall (x: option T), {x = None} + { x <> None}.
```

```
Proof.
    intros. destruct x.
    - right. discriminate.
    - left. reflexivity.
Defined.


Definition subst_option (S : var → type) (Gamma : var → option type) (t : var)
  ↪ : option type :=
    match Gamma t with
    | None ⇒ None
    | Some z ⇒ Some (subst S z)
    end.

Definition subst_option_monad (S : var → type) (Gamma : var → option type) :
  ↪ var → option type :=
    Gamma >>= (subst S >>> Some).

Lemma subst_option_def : subst_option = subst_option_monad.
Proof.
    unfold subst_option.
    unfold subst_option_monad.
    unfold kleisli_option. unfold funcomp. reflexivity.
Qed.

Notation "s .?[ sigma ]" := (subst_option sigma s) (at level 2,
    sigma at level 200, left associativity, format "s .?[ sigma ]") :
      ↪ subst_scope.

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

Theorem subst_repo_some : forall (Gamma : repo) (Su : var → type) (a : var)
  ↪ (tau: type),
    nth_error Gamma a = Some tau →
    nth_error Gamma..[Su] a = Some tau.[Su].
Proof.
  intros.
  unfold subst.
  eapply map_nth_error in H.
  exact H.
Qed.

Theorem subst_repo_none : forall (Gamma : repo) (Su : var → type) (a : var),
    nth_error Gamma a = None →
    nth_error Gamma..[Su] a = None.
Proof.
  intros.
  apply nth_error_None in H.
  apply nth_error_None.
  unfold subst.
  erewrite ← map_length in H.
  exact H.
```

```
Qed.

Theorem subst_repo : forall (Gamma : repo) (Su : var → type) (a : var),
    nth_error Gamma..[Su] a = (nth_error Gamma a)..[Su].
Proof.
    intros.
    destruct (nth_error Gamma a) eqn:G.
    - apply subst_repo_some. assumption.
    - apply subst_repo_none. assumption.
Qed.

Theorem subst_repo_cons : forall (Gamma : repo) (Su : var → type)
    (A : type),
    (A.[Su] :: Gamma..[Su]) = (A :: Gamma)..[Su].
Proof.
    autosubst.
Qed.

Inductive subformula : type → type → Prop :=
| subf_refl : forall rho, subformula rho rho
| subf_arrow_l : forall rho sigma tau, subformula rho sigma → subformula rho
 ↪ (sigma ⇝ tau)
| subf_arrow_r : forall rho sigma tau, subformula rho tau → subformula rho
 ↪ (sigma ⇝ tau)
.

Theorem subformula_dec : forall x y, {subformula x y} + {~subformula x y}.
Proof.
    intros.
    destruct (x == y); dec_eq.
    - left. constructor.
    - generalize dependent x. induction y; intros.
      + right. isfalse.
      + destruct (x == y1); dec_eq.
        { left. constructor. constructor. }
        { destruct (IHy1 x); dec_eq.
          - assumption.
          - left. constructor. assumption.
          - destruct (x == y2); dec_eq.
            + left. constructor 3. constructor.
            + destruct (IHy2 x); dec_eq.
              { assumption. }
              { left. constructor 3. assumption. }
              { right. isfalse. } }
Defined.


Definition single_subst (a: var) (tau: type) : var → type :=
    fun (y: var) ⇒ if a == y then tau else ? y.

Definition rel_dom {A B} (ls : list (A * B)) : list A :=
    map fst ls.

Definition rel_codom {A B} (ls : list (A * B)) : list B :=
    map snd ls.

```

```
Definition not_subf (a : var) (tau : type) :=
    ~(subformula (? a) tau).

Theorem not_subf_dec : forall a tau,
    {~subformula (? a) tau} + {~(~subformula (? a) tau) }.
Proof.
    intros.
    destruct (subformula_dec (? a) tau).
    - right. intros F. apply F. assumption.
    - left. assumption.
Defined.

Fixpoint nth_subformula (n:nat) (rho:type) : option type :=
    match (n, rho) with
    | (0, ? x) => Some (? x)
    | (0, sigma ~> tau) => Some sigma
    | (Datatypes.S n', ? x) => None
    | (Datatypes.S n', sigma ~> tau) => nth_subformula n' tau
    end.

Definition mk_arrow_option (left : type) (right : option type) : type :=
    match right with
    | None => left
    | Some x => left ~> x
    end.

Fixpoint type_init (rho: type) : option type :=
    match rho with
    | ? x => None
    | sigma ~> tau => Some (mk_arrow_option sigma (type_init tau))
    end.

Fixpoint type_target (rho: type) : var :=
    match rho with
    | ? x => x
    | sigma ~> tau => type_target tau
    end.

Definition split_type_target (rho: type) : (option type * var) :=
    (type_init rho, type_target rho).

Example nth_subformula_ex : nth_subformula 2 (? 0 ~> (? 1 ~> (? 0 ~> ? 0)) ~> (?
 ↪ 2 ~> ?0) ~> ? 3) = Some (? 2 ~> ?0).
Proof. reflexivity. Qed.

Fixpoint flat_length (rho : type) : nat :=
    match rho with
    | ? x => 1
    | sigma ~> tau => Datatypes.S (flat_length tau)
    end.

Lemma fl_1_iff_var : forall rho, flat_length rho = 1 <-> exists x, rho = ? x.
Proof.
    intros.
    split.
    - intros. destruct rho.
    + exists x. reflexivity.
    + simpl in H. ainv. destruct rho2; simpl in H0; inversion H0.
  - intros. destruct H. subst. reflexivity.
Qed.


Definition make_arrow_type (ts : list type) (a : type) :=
  fold_right Arr a ts.

Lemma make_arrow_type_ts_is_nil {ts rho a}:
  make_arrow_type ts rho = (? a) → ts = [] /\ rho = (? a).
Proof.
  destruct ts.
  - asimpl. auto.
  - asimpl. intros. discriminate H.
Qed.

Lemma pump_type_target : forall sigma tau, type_target tau = type_target (sigma
 ↪  ⇝ tau).
Proof.
    reflexivity.
Qed.

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

Lemma subst_make_arrow : forall Su ts x ss, ss = map (subst Su) ts →
 ↪  make_arrow_type ss (x.[Su])
  = (make_arrow_type ts x).[Su].
Proof.
  induction ts.
  - intros. subst. reflexivity.
  - intros. ainv. simpl. rewrite IHts; reflexivity.
Qed.

Lemma make_arrow_type_last : forall ts t a,
  make_arrow_type (ts ++ [t]) a =
    make_arrow_type (ts) (t ⇝ a).
Proof.
  unfold make_arrow_type.
  intros.
  rewrite ← (rev_involutive ts).
  rewrite ← (rev_head_last).
  rewrite fold_left_rev_right.
  simpl.
  rewrite ← fold_left_rev_right.
  reflexivity.
Qed.

Lemma make_arrow_type_head : forall ts t a,
  make_arrow_type (t :: ts) a =
    t ⇝ make_arrow_type ts a.
```

```
Proof.
  intros. reflexivity.
Qed.

Lemma repo_pump_subst : forall (Gamma : repo) Gamma0 A Su, Gamma = Gamma0..[Su]
 ↪  → (A :: Gamma) = A :: Gamma0..[Su].
Proof.
  intros.
  subst. try rewrite ← subst_repo_cons.
  reflexivity.
Qed.

Lemma repo_subst_exists : forall (Gamma : repo) Su x A, (nth_error Gamma..[Su] x
 ↪  = Some A)
  → exists B, B.[Su] = A /\ nth_error Gamma x = Some B.
Proof.
  intros. destruct (nth_error Gamma x) eqn:Ht.
  + exists t. rewrite subst_repo in H. rewrite Ht in H. ainv. split;
   ↪  reflexivity.
  + rewrite subst_repo in H. rewrite Ht in H. ainv.
Qed.

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

Lemma subst_arr : forall x y Su, x.[Su] ⇝ y.[Su] = (x ⇝ y).[Su].
Proof.
  reflexivity.
Qed.

Lemma add_arr_head : forall A B B0, B = B0 → A ⇝ B = A ⇝ B0.
Proof.
  intros. subst. reflexivity.
Qed.

Lemma mkarrow_subst_exists : forall ts x Su a, x.[Su] = make_arrow_type ts (? a)
 ↪  →
  exists ts0 a0, x = (make_arrow_type ts0 (? a0)).
Proof.
  induction ts.
  - intros. simpl in H. symmetry in H. apply subst_var_is_var in H. exists [].
   ↪  ainv.
  - intros. rewrite make_arrow_type_head in H. apply subst_arr_is_arr_or in H as
   ↪  [[st [st0 [xst [xsu stmkarr]]]] | xvar].
    + apply IHts in stmkarr. inv stmkarr. inv H. exists (st :: x0). exists x.
     ↪  rewrite make_arrow_type_head.
```

```
      reflexivity.
    + ainv. exists []. exists x0. reflexivity.
Qed.


Definition not_in_codom (ls : list (var * type)) (a : var) :=
    Forall (not_subf a) (rel_codom ls).

Theorem not_in_codom_dec : forall ls x, {not_in_codom ls x} + {~not_in_codom ls
 ↪  x}.
Proof.
    intros.
    unfold not_in_codom.
    apply Forall_dec.
    apply not_subf_dec.
Defined.

Theorem not_in_codom_tail (ls : list (var * type)) (c : (var * type)) (a : var)
 ↪  :
    not_in_codom (c :: ls) a → not_in_codom ls a.
Proof.
    ainv.
Qed.

Definition domain_codomain_free ls :=
        Forall (not_in_codom ls) (rel_dom ls).

Theorem domain_codomain_free_dec : forall ls, { domain_codomain_free ls } + { ~
 ↪  domain_codomain_free ls }.
Proof.
    intros ls.
    unfold domain_codomain_free.
    apply Forall_dec.
    apply not_in_codom_dec.
Defined.

Definition unique_domain {A B} (ls : list (A * B)) :=
    NoDup (rel_dom ls).

Theorem unique_domain_dec {A B: Type} {eq_dec: EqDec A eq}: forall (ls : list (A
 ↪  * B)),  {unique_domain ls} + {~unique_domain ls}.
Proof.
    intros ls.
    unfold unique_domain.
    induction ls.
    - left. constructor.
    - destruct IHls.
      + destruct (in_dec eq_dec (fst a) (rel_dom ls)).
        { right. isfalse. }
        { left. unfold rel_dom. rewrite map_cons. constructor; assumption. }
      + right. isfalse.
Defined.

Definition correct_context ls :=
        unique_domain ls /\
        domain_codomain_free ls.
```

```
Theorem correct_context_dec : forall ls, {correct_context ls} + {~correct_context ls}.
Proof.
    intros.
    unfold correct_context.
    destruct (unique_domain_dec ls).
    - destruct (domain_codomain_free_dec ls).
      + left. split; assumption.
      + right. isfalse.
    - right. isfalse.
Defined.

(* Was hat das hier zu suchen?
Fixpoint wrap_lam (n : nat) (m : term) : term :=
  match n with
  | 0 => m
  | S n =>  \_ (wrap_lam n (rename (+1) m) @ !0)
  end. *)


Fixpoint fv_type (tau: type) : set var :=
    match tau with
    | ? a => [a]
    | sigma ~> tau => set_union (nat_eq_eqdec) (fv_type sigma) (fv_type tau)
    end.

Fixpoint subst_len_to_index (ls: list var) (v : var) : var :=
    match ls with
    | [] => v
    | a :: ls' => if v == a then 0 else 1 + subst_len_to_index ls' v
    end.

Definition canon_type_subst (tau : type) := subst_len_to_index (fv_type tau) >>> Atom.

Definition canon_type (tau: type) := tau.[canon_type_subst tau].

Example canon_type_ex : canon_type (? 8 ~> ? 8) = (? 0 ~> ? 0).
Proof.
    reflexivity.
Qed.

Instance Ids_option {T} {ids : Ids T} : Ids (option T) := ids >>> Some.
Instance Rename_option {T} {rename : Rename T} : Rename (option T) := fun xi opterm =>
                                                                    match opterm with
                                                                    | None => None
                                                                    | Some term => Some (rename xi term)
                                                                    end.

```

```
(* Unifikation und so brauchen wir nicht :(
Fixpoint app_unify (Gamma : list type) (sigma : type) (tau : type) : option type :=
  Some tau.

Fixpoint infer_type (Gamma : repo) (depth: nat) (m : term) : option type :=
  match m with
  | !x => nth_error Gamma x
  | \_ s => let otau := infer_type ((? depth) :: Gamma) (depth + 1) s in
            match otau with
            | Some tau => Some (? depth ~> tau)
            | None => None
            end
  | p @ q => let osigma := infer_type Gamma depth q in
             let otau_sigma := infer_type Gamma depth p in
             match (osigma, otau_sigma) with
             | (Some sigma, Some tau) => app_unify Gamma sigma tau
             | _ => None
             end
  end.

Definition upd {A} {B} {eqdec: EqDec A _} (f : A -> B) (upda: A) (updb: B) (a : A) : B :=
  if eqdec upda a then
    updb
  else
    f a.

Fixpoint unify_ (types: nat) (rho1 : type) (rho2 : type)  : option (var -> type) :=
    match types with
    | 0 => None
    | S n =>
      match (rho1, rho2) with
      | (? a, _) => if subformula_dec (? a) rho2 then
                    if (? a) == rho2 then
                      Some ids
                    else
                      None
                 else
                   Some (single_subst a rho2)
      | (_, ? a) => if subformula_dec (? a) rho1 then
                    if (? a) == rho1 then
                      Some ids
                    else
                      None
                 else
                   Some (single_subst a rho1)
      | (sigma1 ~> sigma2, tau1 ~> tau2) => let oSu := unify_ n sigma2 tau2 in
                                           match oSu with
                                           | None => None
                                           | Some Su =>
                                               unify_ n sigma1.[Su] tau1.[Su] >>=
                                                 fun Sbst => Some (Su >> Sbst)
                                           end
      end
    end.

Fixpoint depth_ty rho := match rho with
| ? n ⇒ 1
| sigma ⇝ tau ⇒ 1 + max (depth_ty sigma) (depth_ty tau)
end.

Definition unify rho1 rho2 := unify_
  ((length (fv_type (rho1 ⇝ rho2))) * (depth_ty (rho1⇝rho2))) rho1 rho2.

Definition mgu rho1 rho2 := unify rho1 rho2 ≫=
                                 fun Su ⇒ Some rho1.[Su].

Lemma nat_refl: forall x, (PeanoNat.Nat.eq_dec x x = left eq_refl).
Proof.
  intros.
  induction x.
  - reflexivity.
  - simpl. rewrite IHx. reflexivity.
Defined.

Lemma term_refl: forall x, eq_dec_term x x = left eq_refl.
Proof.
  induction x.
  - simpl. rewrite nat_refl. reflexivity.
  - simpl. rewrite IHx1. rewrite IHx2. reflexivity.
  - simpl. rewrite IHx. reflexivity.
Defined.

Lemma type_refl: forall t, eq_dec_type t t = left eq_refl.
Proof.
  induction t.
  - simpl. rewrite nat_refl. reflexivity.
  - simpl. rewrite IHt1. rewrite IHt2. reflexivity.
Defined.

Lemma notU : (if subformula_dec (? 0) (? 0 ⇝ ? 0) then true else false) = true.
Proof.
    reflexivity.
Qed.

Fixpoint count_app (m: term) : nat :=
  match m with
  | p @ q ⇒ 1 + count_app p
  | _ ⇒ 0
  end.

Fixpoint first_term t :=
  match t with
  | p @ q ⇒ first_term p
  | s ⇒ s
  end.

Fixpoint uncurry (t : term) : term * list term :=
  match t with
  | p @ q ⇒ let (hd, tl) := uncurry p in
              (hd, tl ++ [q])
  | m ⇒ (m , [])
  end.

Lemma uncurry_var_singl t x: (x, []) = uncurry t → t = x.
Proof.
  revert t x.
  induction t.
  + ainv.
  + intros. asimpl in H. destruct (uncurry t1). ainv. destruct l; ainv.
  + ainv.
Qed.

Hint Immediate uncurry_var_singl.
Hint Unfold uncurry.
*)
Fixpoint first_fresh_type (rho: type) : var :=
  match rho with
  | ? x ⇒ (S x)
  | sigma ⇝ tau ⇒ S (Nat.max (first_fresh_type sigma) (first_fresh_type tau))
  end.

Definition fresh_type (rho: type) : type := ? (first_fresh_type rho).
```