Terms.v

```
Require Import Autosubst.Autosubst.
Require Import Nat PeanoNat.
Require Import Coq.Arith.EqNat.
Require Import Coq.Logic.FunctionalExtensionality.
Require Import Coq.Logic.Classical_Pred_Type.
Require Import Coq.Lists.List.
Require Import Coq.Lists.ListSet.
Require Import Coq.Classes.EquivDec.
Require Import Coq.Bool.Sumbool.
Require Import Coq.Classes.DecidableClass.

Require Import PrincInh.Utils.

Import ListNotations.

Inductive term :=
| Var (x : var)
| App (p q : term)
| Lam (s : {bind term}).


Notation "'!' x" := (Var x) (at level 15).
Notation "p '@' q" := (App p q) (at level 31, left associativity).
Notation "'\_' p" := (Lam p) (at level 35, right associativity).

Instance Ids_term : Ids term. derive. Defined.
Instance Rename_term : Rename term. derive. Defined.
Instance Subst_term : Subst term. derive. Defined.
Instance SubstLemmas_term : SubstLemmas term. derive. Qed.

Definition tI := \_ !0.
Definition tK := \_ \_ !0.
Definition tS := \_\_\_((!2@!0)@(!1@!0)).

Fixpoint term_length (m: term) : nat :=
  match m with
  | Var _ ⇒ 1
  | App p q ⇒ 1 + (term_length p) + (term_length q)
  | Lam s ⇒ 1 + (term_length s)
  end.
```

```

Instance eq_dec_term : EqDec term eq.
Proof.
    unfold EqDec.
    unfold equiv.
    induction x.
    - destruct y.
      + destruct (x == x0).
        { left. ainv. }
        { right. unfold complement. intros F. inversion F.
         ↪  contradiction. }
      + right. intros F. inversion F.
      + right. intros F. inversion F.
    - destruct y.
      + right. intros F. inversion F.
      + destruct (IHx1 y1).
        { destruct (IHx2 y2).
          - left. subst. reflexivity.
          - right. intros F. inversion F. contradiction. }
        { right. intros F. inversion F. contradiction. }
      + right. intros F. ainv.
    - destruct y.
      + right. intros F. ainv.
      + right. intros F. ainv.
      + destruct (IHx s0).
        { left. subst. reflexivity. }
        { right. intros F. inversion F. contradiction. }
Defined.

Goal forall sigma,
      (Lam (App (Var 0) (Var 3))).[sigma] = Lam (App (Var 0)
       ↪  (sigma 2).[ren(+1)]).
intros. asimpl. reflexivity. Qed.

Inductive step : term -> term -> Prop :=
| Step_beta (s1 s2 t : term) :
    s1.[t/] = s2 -> step (App (Lam s1) t) s2
| Step_appL (s1 s2 t : term) :
        step s1 s2 -> step (App s1 t) (App s2 t)
| Step_appR (s t1 t2 : term) :
        step t1 t2 -> step (App s t1) (App s t2)
| Step_lam (s1 s2 : term) :
        step s1 s2 -> step (Lam s1) (Lam s2).

```

```
Lemma substitutivity s1 s2 :
       step s1 s2 -> forall sigma, step s1.[sigma] s2.[sigma].
Proof.
    induction 1; constructor; subst; try autosubst.
Qed.

Lemma term_not_rec_appL : forall s t, s <> s @ t.
Proof.
    intros s t F.
    induction s.
    - inversion F.
    - inversion F. subst. contradiction.
    - inversion F.
Qed.

Lemma term_not_rec_appR : forall s t, s <> t @ s.
Proof.
    intros s t F.
    induction s.
    - inversion F.
    - inversion F. subst. contradiction.
    - inversion F.
Qed.

Definition omega_term := \_ !0 @ !0.

Definition Omega_term := omega_term@omega_term.

Example omega_step : step Omega_term Omega_term.
Proof.
    constructor. reflexivity.
Qed.

Inductive subterm : term -> term -> Prop :=
| subterm_refl : forall t, subterm t t
| subterm_appL : forall s s' t, subterm s s' -> subterm s (s' @ t)
| subterm_appR : forall s t t', subterm t t' -> subterm t (s @ t')
| subterm_lam : forall t t', subterm t t' -> subterm t (\_ t').

Theorem subterm_dec : forall t t', (subterm t t') + {~subterm t t'}.
```

```
Proof.
    intros.
    induction t'.
        + destruct (t == (!x)).
          { left. ainv. constructor. }
          { right. intros F. inversion F. subst. apply c. reflexivity. }
        + destruct IHt'1.
          { left. apply subterm_appL. apply s. }
          { destruct IHt'2.
            - left. apply subterm_appR. apply s.
            - destruct (t == (t'1 @ t'2)).
              + ainv. left. constructor.
              + right. intros F. ainv. apply c. reflexivity. }
        + destruct IHt'.
          { left. constructor. assumption. }
          { destruct (t == (\_s)); dec_eq.
            - left. constructor.
            - right. intros F. ainv. dec_eq. }
Defined.

Definition NF (t : term) := forall t', ~step t t'.

Theorem redex_no_NF : forall t, (exists m n, subterm ((\_ m) @ n) t) -> ~NF t.
Proof.
    induction t.
    - ainv.
    - intros. unfold NF. intros F. ainv. inversion H.
      + subst. pose proof (F x.[t2/]). apply H0. constructor. reflexivity.
      + subst. apply IHt1.
        { exists x. exists x0. assumption. }
        { unfold NF. intros. intros Fstep. pose proof (F (t' @ t2)). apply H0.
        constructor. assumption. }
      + subst. apply IHt2.
        { exists x. exists x0. assumption. }
        { unfold NF. intros. intros Fstep. pose proof (F (t1 @ t')). apply H0.
        constructor. assumption. }
    - ainv. intros F. unfold NF in F. eapply IHt.
      + exists x. exists x0. assumption.
```

```
        + unfold NF. intros. intros Fstep. eapply F. constructor.
          ↪ apply Fstep.
Qed.

Theorem NF_no_redex : forall t, NF t → ~(exists m n, subterm
  ↪ ((\_ m) @ n) t).
Proof.
    intros. intros F. apply redex_no_NF in F. contradiction.
Qed.

Theorem no_redex_NF : forall t, ~(exists m n, subterm ((\_m) @
  ↪ n ) t) → NF t.
Proof.
    intros.
    induction t.
    - unfold NF. intros. intros F. ainv.
    - unfold NF. intros. intros F. inversion F.
      + subst. apply H. exists s1. exists t2. constructor.
      + subst. apply IHt1 with s2.
        { intros Fex. ainv. apply H. exists x. exists x0.
          ↪ constructor. assumption. }
        { assumption. }
      + subst. apply IHt2 with t3.
        { intros Fex. ainv. apply H. exists x. exists x0.
          ↪ constructor 3. assumption. }
        { assumption. }
    - unfold NF. intros. intros F. ainv. apply IHt with s2.
      + intros Fex. ainv. apply H. exists x. exists x0.
        ↪ constructor. assumption.
      + assumption.
Qed.

Theorem NF_iff_no_redex : forall t, NF t ↔ ~(exists m n,
  ↪ subterm ((\_m) @ n) t).
Proof.
    intros t. split.
    - apply NF_no_redex.
    - apply no_redex_NF.
Qed.

Theorem exists_redex_dec : forall t ,
    {(exists m n, subterm ((\_ m) @ n) t)} + {~(exists m n,
      ↪ subterm ((\_ m) @ n) t)}.
```

```
Proof.
    intros t.
    simpl.
    induction t.
    - right. intros F. ainv.
    - destruct IHt1.
       + left. ainv. exists x. exists x0. constructor. apply
        ↪ H0.
       + destruct IHt2.
         { left. ainv. exists x. exists x0. constructor 3.
          ↪ assumption. }
         { destruct t1.
           - right. intros F. ainv. inversion H0.
             + subst. ainv.
             + subst. apply n0. exists x0. exists x1.
              ↪ assumption.
           - right. intros F. ainv. inversion H0.
             + subst. apply n. exists x. exists x0. assumption.
             + subst. apply n0. exists x. exists x0.
              ↪ assumption.
           - left. exists s. exists t2. constructor. }
    - destruct IHt.
      + left. ainv. exists x. exists x0. constructor.
       ↪ assumption.
      + right. intros F. apply n. ainv. exists x. exists x0.
       ↪ assumption.
Defined.

Theorem is_NF_dec : forall t, {NF t}+{~(NF t)}.
Proof.
    intros.
    destruct (exists_redex_dec t).
    - right. intros F. apply NF_iff_no_redex in F.
     ↪ contradiction.
    - left. apply NF_iff_no_redex. assumption.
Defined.

Definition curry (x:term) (terms: list term) : term :=
    fold_left App terms x.

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

Lemma curry_tail : forall ms x a, curry x (ms ++ [a]) = curry x
 ↪  ms @ a.
Proof.
    induction ms.
    - reflexivity.
    - simpl. intros. rewrite (IHms (x@a) a0). reflexivity.
Qed.

Example curry_ex : curry tS [tI ; tS ; tK ] = (tS@tI)@tS@tK.
Proof.
    reflexivity.
Qed.

Lemma curry_if_nil : forall ms a x,
   ! x = curry a ms →
   a = (!x) /\ ms = [].
Proof.
    induction ms.
    - simpl in *. ainv. auto.
    - intros. apply IHms in H. ainv.
Qed.

Lemma curry_split : forall x l a s t, curry (! x) (l ++ [a]) =
 ↪  s @ t →
  s = curry (! x) l /\ t = a.
Proof.
  intros.
  rewrite curry_tail in H. ainv. split; reflexivity.
Qed.

Lemma term_app_split : forall m n, term_length (m@n) = 1 +
 ↪  term_length m + term_length n.
Proof.
  intros.
  constructor.
Qed.

Lemma curry_le_cons : forall ms x a, term_length (curry x ms)
 ↪  ⩽ term_length (curry x (a :: ms)).
Proof.
  intros.
  revert x.
  induction ms using rev_ind.
  - simpl. firstorder.
  - intros. rewrite app_comm_cons.
    repeat rewrite curry_tail.
    repeat rewrite term_app_split.
    firstorder.
Qed.

Lemma curry_le_last : forall ms x a, term_length (curry x ms)
 ↪ ≤ term_length (curry x (ms ++ [a])).
Proof.
  intros.
  revert x.
  induction ms.
  - simpl. firstorder.
  - intros. simpl. apply IHms.
Qed.

Lemma curry_le : forall x ms n, term_length (curry x ms) ≤ n
 ↪ →
  Forall (fun m ⇒ term_length m < n) ms.
  Proof.
    intros x ms.
    induction ms using rev_ind.
    - intros; constructor.
    - intros.
      apply Forall_forall. intros.
      eapply (Nat.lt_le_trans); [ | exact H].
    apply in_app_or in H0 as [H1 | H2].
      + simpl. eapply (Nat.lt_le_trans); [ | apply
        ↪ curry_le_last].
      rewrite ← (curry_le_last ms x x0) in H.
        generalize (proj1 (Forall_forall _ _) (IHms
          ↪ (term_length (curry x ms)) (Nat.le_refl _))).
        intros.
        eapply H0. assumption.
      + inversion H2.
        { subst. rewrite curry_tail. simpl. firstorder. }
        { ainv. }
  Qed.


 (* TODO Nicht mehr genutzt *)
```

```
Lemma curry_subst : forall ts t f, (curry t ts).[f] = curry
 ↪  (t.[f]) (map (subst f) ts).
Proof.
  induction ts using rev_ind.
  - reflexivity.
  - intros.
    rewrite map_app.
    simpl.
    repeat rewrite curry_tail.
    simpl.
    rewrite IHts.
    reflexivity.
Qed.

Lemma curry_var : forall x, ! x = curry (! x) [].
Proof.
  auto.
Qed.
```