Contents

Chapter 1

Library B_Unification.terms

```
Require Import Bool.
Require Import Arith.
Require Import List.
Import ListNotations.
```

```
Definition var := nat.
```

```
Definition var_eq_dec := Nat.eq_dec.
```

```
Inductive term: Type :=
  | T0 : term
  | T1 : term
  | VAR : var → term
  | PRODUCT : term → term → term
  | SUM : term → term → term.
```

```
Implicit Types x y z : term.
Implicit Types n m : var.
```

```
Notation "x + y" := (SUM x y).
Notation "x * y" := (PRODUCT x y).
```

```
Axiom sum_comm : ∀ x y, x + y = y + x.
```

```
Axiom sum_assoc : ∀ x y z, (x + y) + z = x + (y + z).
```

```
Axiom sum_id : ∀ x, T0 + x = x.
```

```
Axiom sum_x_x : ∀ x, x + x = T0.
```

```
Axiom mul_comm : ∀ x y, x × y = y × x.
```

```
Axiom mul_assoc : ∀ x y z, (x × y) × z = x × (y × z).
```

```
Axiom mul_x_x : ∀ x, x × x = x.
```

```
Axiom mul_T0_x : ∀ x, T0 × x = T0.
```

```
Axiom mul_id : ∀ x, T1 × x = x.
```

```
Axiom distr : ∀ x y z, x × (y + z) = (x × y) + (x × z).

Hint Resolve sum_comm sum_assoc sum_x_x sum_id distr
        mul_comm mul_assoc mul_x_x mul_T0_x mul_id.

Lemma mul_x_x_plus_T1 :
  ∀ x, x × (x + T1) = T0.
Proof.
intros. rewrite distr. rewrite mul_x_x. rewrite mul_comm.
rewrite mul_id. rewrite sum_x_x. reflexivity.
Qed.

Lemma x_equal_y_x_plus_y :
  ∀ x y, x = y ↔ x + y = T0.
Proof.
intros. split.
- intros. rewrite H. rewrite sum_x_x. reflexivity.
- intros. inversion H.
Qed.

Hint Resolve mul_x_x_plus_T1.
Hint Resolve x_equal_y_x_plus_y.

Fixpoint term_contains_var (t : term) (v : var) : bool :=
  match t with
    | VAR x ⇒ if (beq_nat x v) then true else false
    | PRODUCT x y ⇒ (orb (term_contains_var x v) (term_contains_var y v))
    | SUM x y ⇒ (orb (term_contains_var x v) (term_contains_var y v))
    | _ ⇒ false
  end.
```

GROUND TERM DEFINITIONS AND LEMMAS

```
Fixpoint ground_term (t : term) : Prop :=
  match t with
    | VAR x ⇒ False
    | SUM x y ⇒ (ground_term x) ∧ (ground_term y)
    | PRODUCT x y ⇒ (ground_term x) ∧ (ground_term y)
    | _ ⇒ True
  end.

Example ex_gt1 :
  (ground_term (T0 + T1)).
Proof.
simpl. split.
- reflexivity.
- reflexivity.
Qed.
```

```
Example ex_gt2 :
  (ground_term (VAR 0 × T1)) → False.
Proof.
simpl. intros. destruct H. apply H.
Qed.
```

```
Lemma ground_term_equiv_T0_T1 :
  ∀ x, (ground_term x) → (x = T0 ∨ x = T1).
Proof.
intros. induction x.
- left. reflexivity.
- right. reflexivity.
- contradiction.
- inversion H. destruct IHx1; destruct IHx2; auto. rewrite H2. left. rewrite
mul_T0_x. reflexivity.
rewrite H2. left. rewrite mul_T0_x. reflexivity.
rewrite H3. left. rewrite mul_comm. rewrite mul_T0_x. reflexivity.
rewrite H2. rewrite H3. right. rewrite mul_id. reflexivity.
- inversion H. destruct IHx1; destruct IHx2; auto. rewrite H2. left. rewrite sum_id.
apply H3.
rewrite H2. rewrite H3. rewrite sum_id. right. reflexivity.
rewrite H2. rewrite H3. right. rewrite sum_comm. rewrite sum_id. reflexivity.
rewrite H2. rewrite H3. rewrite sum_x_x. left. reflexivity.
Qed.
```

Chapter 2

Library B_Unification.poly

```
Require Import ListSet.
Require Import Arith.
Require Import List.
Import ListNotations.
Require Import FunctionalExtensionality.

Require Export terms.

Definition mono := set var.

Definition mono_eq_dec := (list_eq_dec var_eq_dec).

Definition is_mono (m : mono) : Prop := NoDup m.

Definition poly := set mono.

Definition polynomial_eq_dec := (list_eq_dec mono_eq_dec).

Definition is_poly (p : poly) : Prop :=
  NoDup p ∧ ∀ (m : mono), In m p → is_mono m.

Definition vars (p : poly) : set var :=
  nodup var_eq_dec (concat p).

Lemma mono_cons : ∀ x m,
  is_mono (x :: m) →
  is_mono m.
Proof.
  unfold is_mono.
  intros x m Hxm.
  apply NoDup_cons_iff in Hxm as [Hx Hm].
  apply Hm.
Qed.

Lemma poly_cons : ∀ m p,
  is_poly (m :: p) →
  is_poly p ∧ is_mono m.
```

```
Proof.
  unfold is_poly.
  intros m p [Hmp HIm].
  split.
  - split.
    + apply NoDup_cons_iff in Hmp as [Hm Hp]. apply Hp.
    + intros. apply HIm, in_cons, H.
  - apply HIm, in_eq.
Qed.

Definition set_symdiff {X:Type} Aeq_dec (x y:set X) : set X :=
  set_diff Aeq_dec (set_union Aeq_dec x y) (set_inter Aeq_dec x y).

Lemma set_add_cons :
  ∀ X (x : set X) (a : X) Aeq_dec,
  ¬ In a x → set_add Aeq_dec a x = x ++ [a].
Proof.
  intros X x a Aeq_dec H. induction x.
  - reflexivity.
  - simpl. destruct (Aeq_dec a a0).
    + unfold not in H. exfalso. apply H. simpl. left. rewrite e. reflexivity.
    + rewrite IHx.
      × reflexivity.
      × unfold not in *. intro. apply H. simpl. right. apply H0.
Qed.

Lemma set_union_cons :
  ∀ X (x : set X) (a : X) Aeq_dec,
  ¬ In a x → set_union Aeq_dec [a] x = a :: x.
Proof.
Admitted.

Lemma set_union_cons_rev :
  ∀ X (x : set X) (a : X) Aeq_dec,
  NoDup x → ¬ In a x → set_union Aeq_dec [a] x = a :: (rev x).
Proof.
  intros X x a Aeq_dec Hn H. induction x.
  - reflexivity.
  - simpl set_union. rewrite IHx.
    + rewrite set_add_cons.
      × simpl. reflexivity.
      × unfold not in *. intro. apply H. destruct H0.
        – rewrite H0. left. reflexivity.
        – apply NoDup_cons_iff in Hn as []. apply In_rev in H0. contradiction.
    + apply NoDup_cons_iff in Hn. apply Hn.
```

```
    + unfold not in *. intro. apply H. right. apply H0.
Qed.

Lemma set_diff_nil :
  ∀ X (x : set X) Aeq_dec,
  NoDup x → set_diff Aeq_dec x [] = (rev x).
Proof.
  intros X x Aeq_dec H. simpl. induction x.
  - reflexivity.
  - simpl. rewrite IHx.
    + apply set_add_cons. apply NoDup_cons_iff in H as []. unfold not in *.
      intro. apply H. apply In_rev in H1. apply H1.
    + apply NoDup_cons_iff in H. apply H.
Qed.

Lemma set_symdiff_app : ∀ X a (x : set X) Aeq_dec,
  NoDup x → ¬ In a x →
  set_symdiff Aeq_dec [a] x = x ++ [a].
Proof.
  intros X a x Aeq_dec Hn H. unfold set_symdiff. simpl.
  replace (set_mem Aeq_dec a x) with (false).
  - rewrite set_diff_nil.
    + rewrite set_union_cons_rev.
      × simpl. rewrite rev_involutive. reflexivity.
      × apply Hn.
      × apply H.
    + apply set_union_nodup.
      × apply NoDup_cons. intro. contradiction. apply NoDup_nil.
      × apply Hn.
  - symmetry. apply set_mem_complete2. unfold set_In. apply H.
Qed.

Lemma set_symdiff_refl : ∀ X (x y : set X) Aeq_dec,
  set_symdiff Aeq_dec x y = set_symdiff Aeq_dec y x.
Proof.
  intros X x y Aeq_dec. unfold set_symdiff.
Admitted.

Lemma set_symdiff_nil : ∀ X (x : set X) Aeq_dec,
  set_symdiff Aeq_dec [] x = x.
Proof.
Admitted.

Lemma set_part_nodup : ∀ X p (x t f : set X),
  NoDup x →
  partition p x = (t, f) →
```

```
  NoDup t ∧ NoDup f.
Proof.
Admitted.
Lemma set_part_no_inter : ∀ X p (x t f : set X) Aeq_dec,
  NoDup x →
  partition p x = (t, f) →
  set_inter Aeq_dec t f = [].
Proof.
Admitted.
Lemma set_part_union : ∀ X p (x t f : set X) Aeq_dec,
  NoDup x →
  partition p x = (t, f) →
  x = set_union Aeq_dec t f.
Proof.
Admitted.
Lemma set_remove_cons : ∀ X (l : set X) x Xeq_dec,
  x :: remove Xeq_dec x l = l.
Proof.
Admitted.
Lemma set_rem_cons_id : ∀ x,
  (fun x0 : list nat ⇒ x :: remove var_eq_dec x x0) = id.
Proof.
  intros.
  apply functional_extensionality.
  intros.
  apply set_remove_cons.
Qed.

Definition addPP (p q : poly) : poly := set_symdiff mono_eq_dec p q.
Definition mulMM (m n : mono) : mono := set_union var_eq_dec m n.
Definition mulMP (m : mono) (p : poly) : poly :=
  fold_left addPP (map (fun n ⇒ [mulMM m n]) p) [].
Definition mulPP (p q : poly) : poly :=
  fold_left addPP (map (fun m ⇒ mulMP m q) p) [].
Lemma mulPP_l_r : ∀ p q r,
  p = q →
  mulPP p r = mulPP q r.
Proof.
  intros p q r H. rewrite H. reflexivity.
Qed.
Lemma mulPP_0 : ∀ p,
```

```
  mulPP [] p = [].
Proof.
  intros p. unfold mulPP. simpl. reflexivity.
Qed.

Lemma addPP_0 : ∀ p,
  addPP [] p = p.
Proof.
  intros p. unfold addPP. simpl. apply set_symdiff_nil.
Qed.

Lemma mulMM_0 : ∀ m,
  mulMM [] m = m.
Proof. Admitted.

Lemma mulMP_0 : ∀ p,
  mulMP [] p = p.
Proof.
  intros p. unfold mulMP. induction p.
  - simpl. reflexivity.
  - simpl.
Admitted.

Lemma mulPP_1 : ∀ p,
  mulPP [[]] p = p.
Proof.
  intros p. unfold mulPP. simpl. rewrite addPP_0. apply mulMP_0.
Qed.

Lemma mulMP_mulPP_eq : ∀ m p,
  mulMP m p = mulPP [m] p.
Proof.
  intros m p. unfold mulPP. simpl. rewrite addPP_0. reflexivity.
Qed.

Lemma addPP_comm : ∀ p q,
  addPP p q = addPP q p.
Proof.
Admitted.

Lemma mulPP_comm : ∀ p q,
  mulPP p q = mulPP q p.
Proof.
  intros p q. unfold mulPP.
Admitted.

Lemma mulPP_addPP_1 : ∀ p q r,
  mulPP (addPP (mulPP p q) r) (addPP [[]] q) =
```

```
  mulPP (addPP [[]] q) r.
Proof.
  intros p q r. unfold mulPP.
Admitted.

Lemma set_part_add : ∀ f p l r,
  NoDup p →
  partition f p = (l, r) →
  p = addPP l r.
Proof.
  intros.
  unfold addPP.
  unfold set_symdiff.
  rewrite (set_part_no_inter _ _ _ _ _ _ H H0).
  assert (NoDup l ∧ NoDup r) as [Hl Hr].
  apply (set_part_nodup _ _ _ _ _ H H0).
  assert (Hndu: NoDup (set_union mono_eq_dec l r)).
  apply (set_union_nodup _ Hl Hr).
  rewrite (set_diff_nil _ _ _ Hndu).
  assert (p = (set_union mono_eq_dec l r)).
  apply (set_part_union _ _ _ _ _ _ H H0).
Admitted.
```

Chapter 3

Library B_Unification.poly_unif

```
Require Import ListSet.
Require Import List.
Import ListNotations.
Require Import Arith.

Require Export poly.

Definition repl := (prod var poly).

Definition subst := list repl.

Definition inDom (x : var) (s : subst) : bool :=
  existsb (beq_nat x) (map fst s).

Fixpoint appSubst (s : subst) (x : var) : poly :=
  match s with
  | [] ⇒ [[x]]
   | (y,p)::s' ⇒ if (x =? y) then p else (appSubst s' x)
  end.

Fixpoint substM (s : subst) (m : mono) : poly :=
  match s with
  | [] ⇒ [m]
  | (y,p)::s' ⇒
    match (inDom y s) with
    | true ⇒ mulPP (appSubst s y) (substM s' m)
    | false ⇒ mulMP [y] (substM s' m)
    end
  end.

Fixpoint substP (s : subst) (p : poly) : poly :=
  match p with
  | [] ⇒ []
  | m :: p' ⇒ addPP (substM s m) (substP s p')
  end.
```

```
Lemma substP_distr_mulPP : ∀ p q s,
  substP s (mulPP p q) = mulPP (substP s p) (substP s q).
Proof.
Admitted.

Definition unifier (s : subst) (p : poly) : Prop :=
  substP s p = [].

Definition unifiable (p : poly) : Prop :=
  ∃ s, unifier s p.

Definition subst_comp (s t u : subst) : Prop :=
  ∀ p,
  is_poly p →
  substP t (substP s p) = substP u p.

Definition more_general (s t : subst) : Prop :=
  ∃ u, subst_comp s u t.

Definition mgu (s : subst) (p : poly) : Prop :=
  unifier s p ∧
  ∀ t,
  unifier t p →
  more_general s t.

Definition reprod_unif (s : subst) (p : poly) : Prop :=
  unifier s p ∧
  ∀ t,
  unifier t p →
  subst_comp s t t.

Lemma reprod_is_mgu : ∀ p s,
  reprod_unif s p →
  mgu s p.
Proof.
Admitted.

Lemma empty_substM : ∀ (m : mono),
  is_mono m →
  substM [] m = [m].
Proof.
  auto.
Qed.

Lemma empty_substP : ∀ (p : poly),
  is_poly p →
  substP [] p = p.
Proof.
  intros.
```

```
  induction p.
  - simpl. reflexivity.
  - simpl.
    apply poly_cons in H as H1.
    destruct H1 as [HPP HMA].
    apply IHp in HPP as HS.
    rewrite HS.
    unfold addPP.
    Admitted.

Lemma empty_mgu : mgu [] [].
Proof.
  unfold mgu, more_general, subst_comp.
  intros.
  simpl.
  split.
  - unfold unifier. apply empty_substP.
    unfold is_poly.
    split.
    + apply NoDup_nil.
    + intros. inversion H.
  - intros.
    ∃ t.
    intros.
    rewrite (empty_substP _ H0).
    reflexivity.
Qed.
```

Chapter 4

Library B_Unification.sve

```
Require Import List.
Import ListNotations.
Require Import Arith.

Require Export poly_unif.

Definition pair (U : Type) : Type := (U × U).

Definition has_var (x : var) := existsb (beq_nat x).

Definition elim_var (x : var) (p : poly) : poly :=
  map (remove var_eq_dec x) p.

Definition div_by_var (x : var) (p : poly) : pair poly :=
  let (qx, r) := partition (has_var x) p in
  (elim_var x qx , r).

Lemma fold_add_self : ∀ p,
  is_poly p →
  p = fold_left addPP (map (fun x ⇒ [x]) p) [].
Proof.
Admitted.

Lemma mulMM_cons : ∀ x m,
  ¬ In x m →
  mulMM [x] m = x :: m.
Proof.
  intros.
  unfold mulMM.
  apply set_union_cons, H.
Qed.

Lemma mulMP_map_cons : ∀ x p q,
  is_poly p →
  is_poly q →
  (∀ m, In m q → ¬ In x m) →
```

```
  p = map (cons x) q →
  p = mulMP [x] q.
Proof.
  intros.
  unfold mulMP.
  assert (map (fun n : mono ⇒ [mulMM [x] n]) q = map (fun n ⇒ [x :: n]) q).
  apply map_ext_in. intros. f_equal. apply mulMM_cons. auto.
  rewrite H3.
  assert (map (fun n ⇒ [x :: n]) q = map (fun n ⇒ [n]) (map (cons x) q)).
  rewrite map_map. auto.
  rewrite H4.
  rewrite ← H2.
  apply (fold_add_self p H).
Qed.
Lemma elim_var_not_in_rem : ∀ x p r,
  elim_var x p = r →
  (∀ m, In m r → ¬ In x m).
Proof.
  intros.
  unfold elim_var in H.
  rewrite ← H in H0.
  apply in_map_iff in H0 as [n []].
  rewrite ← H0.
  apply remove_In.
Qed.
Lemma elim_var_map_cons_rem : ∀ x p r,
  (∀ m, In m p → In x m) →
  elim_var x p = r →
  p = map (cons x) r.
Proof.
  intros.
  unfold elim_var in H0.
  rewrite ← H0.
  rewrite map_map.
  rewrite set_rem_cons_id.
  rewrite map_id.
  reflexivity.
Qed.
Lemma elim_var_mul : ∀ x p r,
  is_poly p →
  is_poly r →
```

```
  (∀ m, In m p → In x m) →
  elim_var x p = r →
  p = mulMP [x] r.
Proof.
  intros.
  apply mulMP_map_cons; auto.
  apply (elim_var_not_in_rem _ _ _ H2).
  apply (elim_var_map_cons_rem _ _ _ H1 H2).
Qed.

Lemma part_fst_true : ∀ X p (x t f : list X),
  partition p x = (t, f) →
  (∀ a, In a t → p a = true).
Proof.
Admitted.

Lemma has_var_eq_in : ∀ x m,
  has_var x m = true ↔ In x m.
Proof.
Admitted.

Lemma div_is_poly : ∀ x p q r,
  is_poly p →
  div_by_var x p = (q, r) →
  is_poly q ∧ is_poly r.
Proof.
Admitted.

Lemma part_is_poly : ∀ f p l r,
  is_poly p →
  partition f p = (l, r) →
  is_poly l ∧ is_poly r.
Proof.
Admitted.

Lemma div_eq : ∀ x p q r,
  is_poly p →
  div_by_var x p = (q, r) →
  p = addPP (mulMP [x] q) r.
Proof.
  intros x p q r HP HD.
  assert (HE := HD).
  unfold div_by_var in HE.
  destruct ((partition (has_var x) p)) as [qx r0] eqn:Hqr.
  injection HE. intros Hr Hq.
  assert (HIH: ∀ m, In m qx → In x m). intros.
```

```
  apply has_var_eq_in.
  apply (part_fst_true _ _ _ _ _ Hqr _ H).
  assert (is_poly q ∧ is_poly r) as [HPq HPr].
  apply (div_is_poly x p q r HP HD).
  assert (is_poly qx ∧ is_poly r0) as [HPqx HPr0].
  apply (part_is_poly (has_var x) p qx r0 HP Hqr).
  apply (elim_var_mul _ _ _ HPqx HPq HIH) in Hq.
  unfold is_poly in HP.
  destruct HP as [Hnd].
  apply (set_part_add (has_var x) _ _ _ Hnd).
  rewrite ← Hq.
  rewrite ← Hr.
  apply Hqr.
Qed.

Definition build_poly (q r : poly) : poly :=
  mulPP (addPP [[]] q) r.

Definition build_subst (s : subst) (x : var) (q r : poly) : subst :=
  let q1 := addPP [[]] q in
  let q1s := substP s q1 in
  let rs := substP s r in
  let xs := (x, addPP (mulMP [x] q1s) rs) in
  xs :: s.

Lemma div_build_unif : ∀ x p q r s,
  is_poly p →
  div_by_var x p = (q, r) →
  unifier s p →
  unifier s (build_poly q r).
Proof.
  unfold build_poly, unifier.
  intros x p q r s HPp HD Hsp0.
  apply (div_eq _ _ _ _ HPp) in HD as Hp.
  assert (∃ q1, q1 = addPP [[]] q) as [q1 Hq1]. eauto.
  assert (∃ sp, sp = substP s p) as [sp Hsp]. eauto.
  assert (∃ sq1, sq1 = substP s q1) as [sq1 Hsq1]. eauto.
  rewrite ← Hsp in Hsp0.
  apply (mulPP_l_r sp [] sq1) in Hsp0.
  rewrite mulPP_0 in Hsp0.
  rewrite ← Hsp0.
  rewrite Hsp, Hsq1.
  rewrite Hp, Hq1.
  rewrite ← substP_distr_mulPP.
```

```
  f_equal.
  rewrite mulMP_mulPP_eq.
  rewrite mulPP_addPP_1.
  reflexivity.
Qed.

Lemma reprod_build_subst : ∀ x p q r s,
  div_by_var x p = (q, r) →
  reprod_unif s (build_poly q r) →
  inDom x s = false →
  reprod_unif (build_subst s x q r) p.
Proof.
Admitted.

Fixpoint sveVars (vars : list var) (p : poly) : option subst :=
  match vars with
  | [] ⇒
      match p with
      | [] ⇒ Some []
      | _ ⇒ None
      end
  | x :: xs ⇒
      let (q, r) := div_by_var x p in
      match sveVars xs (build_poly q r) with
      | None ⇒ None
      | Some s ⇒ Some (build_subst s x q r)
      end
  end.

Definition sve (p : poly) : option subst :=
  sveVars (vars p) p.

Lemma sveVars_correct1 : ∀ (p : poly),
  is_poly p →
  ∀ s, sveVars (vars p) p = Some s →
            mgu s p.
Proof.
  intros.
  induction (vars p) as [|x xs] eqn:HV.
  - simpl in H0.
    destruct p; inversion H0.
    apply empty_mgu.
  - apply IHxs.
Admitted.

Lemma sveVars_correct2 : ∀ (p : poly),
```

```
  is_poly p →
  sveVars (vars p) p = None →
  ¬ unifiable p.
Proof.
Admitted.

Lemma sveVars_correct : ∀ (p : poly),
  is_poly p →
  match sveVars (vars p) p with
  | Some s ⇒ mgu s p
  | None ⇒ ¬ unifiable p
  end.
Proof.
  intros.
  remember (sveVars (vars p) p).
  destruct o.
  - apply sveVars_correct1; auto.
  - apply sveVars_correct2; auto.
Qed.

Theorem sve_correct : ∀ (p : poly),
  is_poly p →
  match sve p with
  | Some s ⇒ mgu s p
  | None ⇒ ¬ unifiable p
  end.
Proof.
  intros.
  apply sveVars_correct.
  auto.
Qed.
```

Chapter 5

Library B_Unification.terms_unif

```
Require Import EqNat.
Require Import Bool.
Require Import List.
```

```
Require Export terms.
```

REPLACEMENT DEFINITIONS AND LEMMAS

```
Definition replacement := (prod var term).
```

```
Implicit Type r : replacement.
```

```
Fixpoint replace (t : term) (r : replacement) : term :=
  match t with
    | T0 ⇒ t
    | T1 ⇒ t
    | VAR x ⇒ if (beq_nat x (fst r)) then (snd r) else t
    | SUM x y ⇒ SUM (replace x r) (replace y r)
    | PRODUCT x y ⇒ PRODUCT (replace x r) (replace y r)
  end.
```

```
Example ex_replace1 :
  (replace (VAR 0 + VAR 1) ((0, VAR 2 × VAR 3))) = (VAR 2 × VAR 3) + VAR 1.
Proof.
simpl. reflexivity.
Qed.
```

```
Example ex_replace2 :
  (replace ((VAR 0 × VAR 1 × VAR 3) + (VAR 3 × VAR 2) × VAR 2) ((2, T0))) = VAR 0
× VAR 1 × VAR 3.
Proof.
simpl. rewrite mul_comm with (x := VAR 3). rewrite mul_T0_x. rewrite mul_T0_x.
rewrite sum_comm with (x := VAR 0 × VAR 1 × VAR 3). rewrite sum_id. reflexivity.
Qed.
```

```
Example ex_replace3 :
  (replace ((VAR 0 + VAR 1) × (VAR 1 + VAR 2)) ((1, VAR 0 + VAR 2))) = VAR 2 × VAR
0.
Proof.
simpl. rewrite sum_assoc. rewrite sum_x_x. rewrite sum_comm.
rewrite sum_comm with (x := VAR 0). rewrite sum_assoc.
rewrite sum_x_x. rewrite sum_comm. rewrite sum_id. rewrite sum_comm.
rewrite sum_id. reflexivity.
Qed.

Lemma replace_distribution :
  ∀ x y r, (replace x r) + (replace y r) = (replace (x + y) r).
Proof.
intros. simpl. reflexivity.
Qed.

Lemma replace_associative :
  ∀ x y r, (replace x r) × (replace y r) = (replace (x × y) r).
Proof.
intros. simpl. reflexivity.
Qed.

Lemma term_cannot_replace_var_if_not_exist :
  ∀ x r, (term_contains_var x (fst r) = false) → (replace x r) = x.
Proof.
intros. induction x.
{ simpl. reflexivity. }
{ simpl. reflexivity. }
{ inversion H. unfold replace. destruct beq_nat.
  inversion H1. reflexivity. }
{ simpl in *. apply orb_false_iff in H. destruct H. apply IHx1 in H.
  apply IHx2 in H0. rewrite H. rewrite H0. reflexivity. }
{ simpl in *. apply orb_false_iff in H. destruct H. apply IHx1 in H.
  apply IHx2 in H0. rewrite H. rewrite H0. reflexivity. }
Qed.

Lemma ground_term_cannot_replace :
  ∀ x, (ground_term x) → (∀ r, replace x r = x).
Proof.
intros. induction x.
- simpl. reflexivity.
- simpl. reflexivity.
- simpl. inversion H.
- simpl. inversion H. apply IHx1 in H0. apply IHx2 in H1. rewrite H0.
rewrite H1. reflexivity.
```

```
- simpl. inversion H. apply IHx1 in H0. apply IHx2 in H1. rewrite H0.
rewrite H1. reflexivity.
Qed.
```

SUBSTITUTION DEFINITIONS AND LEMMAS

```
Definition subst := list replacement.

Implicit Type s : subst.

Fixpoint apply_subst (t : term) (s : subst) : term :=
  match s with
    | nil ⇒ t
    | x :: y ⇒ apply_subst (replace t x) y
  end.

Lemma ground_term_cannot_subst :
  ∀ x, (ground_term x) → (∀ s, apply_subst x s = x).
Proof.
intros. induction s. simpl. reflexivity. simpl. apply ground_term_cannot_replace
with (r := a) in H.
rewrite H. apply IHs.
Qed.

Lemma subst_distribution :
  ∀ s x y, apply_subst x s + apply_subst y s = apply_subst (x + y) s.
Proof.
intro. induction s. simpl. intros. reflexivity. intros. simpl.
apply IHs.
Qed.

Lemma subst_associative :
  ∀ s x y, apply_subst x s × apply_subst y s = apply_subst (x × y) s.
Proof.
intro. induction s. intros. reflexivity. intros. apply IHs.
Qed.

Definition unifies (a b : term) (s : subst) : Prop :=
  (apply_subst a s) = (apply_subst b s).

Example ex_unif1 :
  unifies (VAR 0) (VAR 1) ((0, T0) :: nil) → False.
Proof.
intros. inversion H.
Qed.

Example ex_unif2 :
  unifies (VAR 0) (VAR 1) ((0, T1) :: (1, T1) :: nil).
Proof.
firstorder.
```

```
Qed.
Definition unifies_T0 (a b : term) (s : subst) : Prop :=
  (apply_subst a s) + (apply_subst b s) = T0.
Lemma unifies_T0_equiv :
  ∀ x y s, unifies x y s ↔ unifies_T0 x y s.
Proof.
intros. split.
{ intros. unfold unifies_T0. unfold unifies in H. inversion H.
  rewrite sum_x_x. reflexivity.
}
{ intros. unfold unifies_T0 in H. unfold unifies. inversion H. }
Qed.
Definition unifier (t : term) (s : subst) : Prop :=
  (apply_subst t s) = T0.
Lemma unify_distribution :
  ∀ x y s, (unifies_T0 x y s) ↔ (unifier (x + y) s).
Proof.
intros. split.
{ intros. inversion H. }
{ intros. unfold unifies_T0. unfold unifier in H.
  rewrite ← H. apply subst_distribution. }
Qed.
Definition unifiable (t : term) : Prop :=
  ∃ s, unifier t s.
Example unifiable_ex1 :
  unifiable (T1) → False.
Proof.
intros. inversion H. unfold unifier in H0. rewrite ground_term_cannot_subst in H0.
inversion H0. reflexivity.
Qed.
Example unifiable_ex2 :
  ∀ x, unifiable (x + x + T1) → False.
Proof.
intros. inversion H. unfold unifier in H0. rewrite sum_x_x in H0. rewrite sum_id in
H0.
rewrite ground_term_cannot_subst in H0. inversion H0. reflexivity.
Qed.
```

Chapter 6

Library B_Unification.lowenheim

```
Require Export terms_unif.
```