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 Arith.
Require Import List.
Import ListNotations.
Require Import FunctionalExtensionality.
Require Import Sorting.
Import Nat.
```

```
Require Export terms.
```

2.1 Introduction

Another way of representing the terms of a unification problem is as polynomials and monomials. A monomial is a set of variables multiplied together, and a polynomial is a set of monomials added together. By following the ten axioms set forth in B-unification, we can transform any term to this form.

Since one of the rules is x * x = x, we can guarantee that there are no repeated variables in any given monomial. Similarly, because x + x = 0, we can guarantee that there are no repeated monomials in a polynomial. Because of these properties, as well as the commutativity of addition and multiplication, we can represent both monomials and polynomials as unordered sets of variables and monomials, respectively. This file serves to implement such a representation.

2.2 Monomials and Polynomials

2.2.1 Data Type Definitions

A monomial is simply a list of variables, with variables as defined in terms.v.

```
Definition mono := list var.
```

A polynomial, then, is a list of monomials.

```
Definition poly := list mono.
```

2.2.2 Comparisons of monomials and polynomials

For the sake of simplicity when comparing monomials and polynomials, we have opted for a solution that maintains the lists as sorted. This allows us to simultaneously ensure that there are no duplicates, as well as easily comparing the sets with the standard Coq equals operator over lists.

Ensuring that a list of nats is sorted is easy enough. In order to compare lists of sorted lists, we'll need the help of another function:

```
Fixpoint lex {T : Type} (cmp : T → T → comparison) (l1 l2 : list T)
                  : comparison :=
  match l1, l2 with
  | [], [] ⇒ Eq
  | [], _ ⇒ Lt
  | _, [] ⇒ Gt
  | h1 :: t1, h2 :: t2 ⇒
      match cmp h1 h2 with
      | Eq ⇒ lex cmp t1 t2
      | c ⇒ c
      end
  end.
```

There are some important but relatively straightforward properties of this function that are useful to prove. First, reflexivity:

```
Theorem lex_nat_refl : ∀ (l : list nat), lex compare l l = Eq.
Proof.
  intros.
  induction l.
  - simpl. reflexivity.
  - simpl. rewrite compare_refl. apply IHl.
Qed.
```

Next, antisymmetry. This allows us to take a predicate or hypothesis about the comparison of two polynomials and reverse it. For example, a $<$ b implies b $>$ a.

```
Theorem lex_nat_antisym : ∀ (l1 l2 : list nat),
  lex compare l1 l2 = CompOpp (lex compare l2 l1).
Proof.
  intros l1.
  induction l1.
  - intros. simpl. destruct l2; reflexivity.
  - intros. simpl. destruct l2.
    + simpl. reflexivity.
```

```
  + simpl. destruct (a ?= n) eqn:H;
    rewrite compare_antisym in H;
    rewrite CompOpp_iff in H; simpl in H;
    rewrite H; simpl.
    × apply IHl1.
    × reflexivity.
    × reflexivity.
Qed.
```

```
Lemma lex_eq : ∀ n m,
  lex compare n m = Eq ↔ lex compare m n = Eq.
Proof.
  intros n m. split; intro; rewrite lex_nat_antisym in H; unfold CompOpp in H.
  - destruct (lex compare m n) eqn:H0; inversion H. reflexivity.
  - destruct (lex compare n m) eqn:H0; inversion H. reflexivity.
Qed.
```

```
Lemma lex_lt_gt : ∀ n m,
  lex compare n m = Lt ↔ lex compare m n = Gt.
Proof.
  intros n m. split; intro; rewrite lex_nat_antisym in H; unfold CompOpp in H.
  - destruct (lex compare m n) eqn:H0; inversion H. reflexivity.
  - destruct (lex compare n m) eqn:H0; inversion H. reflexivity.
Qed.
```

Lastly is a property over lists. The comparison of two lists stays the same if the same new element is added onto the front of each list. Similarly, if the item at the front of two lists is equal, removing it from both does not chance the lists' comparison.

```
Theorem lex_nat_cons : ∀ (l1 l2 : list nat) n,
  lex compare l1 l2 = lex compare (n::l1) (n::l2).
Proof.
  intros. simpl. rewrite compare_refl. reflexivity.
Qed.
```

```
Hint Resolve lex_nat_refl lex_nat_antisym lex_nat_cons.
```

2.2.3 Stronger Definitions

Because as far as Coq is concerned any list of natural numbers is a monomial, it is necessary to define a few more predicates about monomials and polynomials to ensure our desired properties hold. Using these in proofs will prevent any random list from being used as a monomial or polynomial.

Monomials are simply sorted lists of natural numbers.

```
Definition is_mono (m : mono) : Prop := Sorted lt m.
```

Polynomials are sorted lists of lists, where all of the lists in the polynomail are monomials.

```
Definition is_poly (p : poly) : Prop :=
  Sorted (fun m n ⇒ lex compare m n = Lt) p ∧ ∀ m, In m p → is_mono m.

Hint Unfold is_mono is_poly.

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

There are a few userful things we can prove about these definitions too. First, every element in a monomial is guaranteed to be less than the elements after it.

```
Lemma mono_order : ∀ x y m,
  is_mono (x :: y :: m) →
  x < y.
Proof.
  unfold is_mono.
  intros.
  apply Sorted_inv in H as [].
  apply HdRel_inv in H0.
  apply H0.
Qed.
```

Similarly, if x :: m is a monomial, then m is also a monomial.

```
Lemma mono_cons : ∀ x m,
  is_mono (x :: m) →
  is_mono m.
Proof.
  unfold is_mono.
  intros.
  apply Sorted_inv in H as [].
  apply H.
Qed.
```

The same properties hold for is_poly as well; any list in a polynomial is guaranteed to be less than the lists after it.

```
Lemma poly_order : ∀ m n p,
  is_poly (m :: n :: p) →
  lex compare m n = Lt.
Proof.
  unfold is_poly.
  intros.
  destruct H.
  apply Sorted_inv in H as [].
  apply HdRel_inv in H1.
```

```
  apply H1.
Qed.
```

And if m :: p is a polynomial, we know both that p is a polynomial and that m is a monomial.

```
Lemma poly_cons : ∀ m p,
  is_poly (m :: p) →
  is_poly p ∧ is_mono m.
Proof.
  unfold is_poly.
  intros.
  destruct H.
  apply Sorted_inv in H as [].
  split.
  - split.
    + apply H.
    + intros. apply H0, in_cons, H2.
  - apply H0, in_eq.
Qed.
```

Lastly, for completeness, nil is both a polynomial and monomial.

```
Lemma nil_is_mono :
  is_mono [].
Proof.
  auto.
Qed.

Lemma nil_is_poly :
  is_poly [].
Proof.
  unfold is_poly. split.
  - auto.
  - intro; contradiction.
Qed.

Hint Resolve mono_order mono_cons poly_order poly_cons nil_is_mono nil_is_poly.
```

2.3 Functions over Monomials and Polynomials

```
Fixpoint addPPn (p q : poly) (n : nat) : poly :=
  match n with
  | 0 ⇒ []
  | S n' ⇒
    match p with
```

```
    | [] => q
    | m::p' =>
      match q with
      | [] => (m :: p')
      | n::q' =>
        match lex compare m n with
        | Eq => addPPn p' q' (pred n')
        | Lt => m :: addPPn p' q n'
        | Gt => n :: addPPn (m::p') q' n'
        end
      end
    end
  end.

Definition addPP (p q : poly) : poly :=
  addPPn p q (length p + length q).

Fixpoint mulMMn (m n : mono) (f : nat) : mono :=
  match f with
  | 0 => []
  | S f' =>
    match m, n with
    | [], _ => n
    | _, [] => m
    | a :: m', b :: n' =>
        match compare a b with
        | Eq => a :: mulMMn m' n' (pred f')
        | Lt => a :: mulMMn m' n f'
        | Gt => b :: mulMMn m n' f'
        end
    end
  end.

Definition mulMM (m n : mono) : mono :=
  mulMMn m n (length m + length n).

Fixpoint mulMP (m : mono) (p : poly) : poly :=
  match p with
  | [] => []
  | n :: p' => addPP [mulMM m n] (mulMP m p')
  end.

Fixpoint mulPP (p q : poly) : poly :=
  match p with
  | [] => []
  | m :: p' => addPP (mulMP m q) (mulPP p' q)
```

```
  end.
Hint Unfold addPP addPPn mulMP mulMMn mulMM mulPP.
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. destruct p; auto.
Qed.
Lemma mulMM_0 : ∀ m,
  mulMM [] m = m.
Proof.
  intros m. unfold mulMM. destruct m; auto.
Qed.
Lemma mulMP_0 : ∀ p,
  is_poly p → mulMP [] p = p.
Proof.
  intros p Hp. induction p.
  - simpl. reflexivity.
  - simpl. rewrite mulMM_0. rewrite IHp.
    + unfold addPP. simpl. destruct p.
      × reflexivity.
      × apply poly_order in Hp. rewrite Hp. auto.
    + apply poly_cons in Hp. apply Hp.
Qed.
Lemma addPP_comm : ∀ p q,
  is_poly p ∧ is_poly q → addPP p q = addPP q p.
Proof.
  intros p q H. generalize dependent q. induction p; induction q.
  - reflexivity.
  - rewrite addPP_0. destruct q; auto.
  - rewrite addPP_0. destruct p; auto.
```

```
- intro. unfold addPP. simpl. destruct (lex compare a a0) eqn:Hlex.
  + apply lex_eq in Hlex. rewrite Hlex. rewrite plus_comm. simpl.
    rewrite ← (plus_comm (S (length p))). simpl. unfold addPP in IHp.
    rewrite plus_comm. rewrite IHp.
    × rewrite plus_comm. reflexivity.
    × destruct H. apply poly_cons in H as []. apply poly_cons in H0 as []. split;
auto.
  + apply lex_lt_gt in Hlex. rewrite Hlex. f_equal. admit.
  + apply lex_lt_gt in Hlex. rewrite Hlex. f_equal. unfold addPP in IHq. simpl
length in IHq. rewrite ← IHq.
    × rewrite ← add_1_l. rewrite plus_assoc. rewrite ← (add_1_r (length p)). reflexivity.
    × destruct H. apply poly_cons in H0 as []. split; auto.
Admitted.

Lemma addPP_is_poly : ∀ p q,
  is_poly p ∧ is_poly q → is_poly (addPP p q).
Proof.
  intros p q Hpoly. inversion Hpoly. unfold is_poly in H, H0. destruct H, H0. split.
  - remember (fun m n : list nat ⇒ lex compare m n = Lt) as comp. generalize dependent
q. induction p, q.
    + intros. apply Sorted_nil.
    + intros. rewrite addPP_0. apply H0.
    + intros. rewrite addPP_comm. rewrite addPP_0. apply H. apply Hpoly.
    + intros. unfold addPP. simpl. destruct (lex compare a m) eqn:Hlex.
      × rewrite plus_comm. simpl. rewrite plus_comm. apply IHp.
        – apply Sorted_inv in H as []; auto.
        – intuition.
        – destruct Hpoly. apply poly_cons in H3 as []. apply poly_cons in H4 as [].
split; auto.
        – apply Sorted_inv in H0 as []; auto.
        – intuition.
      × apply Sorted_cons.
        – rewrite plus_comm. simpl.
Admitted.

Lemma mulPP_1 : ∀ p,
  is_poly p → mulPP [[]] p = p.
Proof.
  intros p H. unfold mulPP. rewrite mulMP_0. rewrite addPP_comm.
  - apply addPP_0.
  - split; auto.
  - apply H.
Qed.

Lemma mulMP_is_poly : ∀ m p,
```

```
  is_mono m ∧ is_poly p → is_poly (mulMP m p).
Proof. Admitted.

Hint Resolve mulMP_is_poly.

Lemma mulMP_mulPP_eq : ∀ m p,
  is_mono m ∧ is_poly p → mulMP m p = mulPP [m] p.
Proof.
  intros m p H. unfold mulPP. rewrite addPP_comm.
  - rewrite addPP_0. reflexivity.
  - split; auto.
Qed.

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.
```

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).

Fixpoint get_var (p : poly) : option var :=
  match p with
  | [] ⇒ None
  | [] :: p' ⇒ get_var p'
  | (x :: m) :: p' ⇒ Some x
  end.

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).

Definition decomp (p : poly) : option (prod var (pair poly)) :=
  match get_var p with
  | None ⇒ None
  | Some x ⇒ Some (x, (div_by_var x p))
  end.

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 decomp_is_poly : ∀ x p q r,
  is_poly p →
  decomp p = Some (x, (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 decomp_eq : ∀ x p q r,
  is_poly p →
  decomp p = Some (x, (q, r)) →
  p = addPP (mulMP [x] q) r.
Proof.
  intros x p q r HP HD.
  assert (HE: div_by_var x p = (q, r)).
  unfold decomp in HD. destruct (get_var p); inversion HD; auto.

  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 (decomp_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 decomp_unif : ∀ x p q r s,
  is_poly p →
  decomp p = Some (x, (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 (decomp_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,
  decomp p = Some (x, (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 bunifyN (n : nat) : poly → option subst := fun p ⇒
  match n with
  | 0 ⇒ None
  | S n' ⇒
      match decomp p with
      | None ⇒ match p with
                 | [] ⇒ Some []
                 | _ ⇒ None
                 end
      | Some (x, (q, r)) ⇒
          match bunifyN n' (build_poly q r) with
          | None ⇒ None
          | Some s ⇒ Some (build_subst s x q r)
          end
      end
  end.
```

```
Definition bunify (p : poly) : option subst :=
  bunifyN (1 + length (vars p)) p.

Lemma bunifyN_correct1 : ∀ (p : poly) (n : nat),
  is_poly p →
  length (vars p) < n →
  ∀ s, bunifyN n p = Some s →
            mgu s p.
Proof.
Admitted.

Lemma bunifyN_correct2 : ∀ (p : poly) (n : nat),
  is_poly p →
  length (vars p) < n →
  bunifyN n p = None →
  ¬ unifiable p.
Proof.
Admitted.

Lemma bunifyN_correct : ∀ (p : poly) (n : nat),
  is_poly p →
  length (vars p) < n →
  match bunifyN n p with
  | Some s ⇒ mgu s p
  | None ⇒ ¬ unifiable p
  end.
Proof.
  intros.
  remember (bunifyN n p).
  destruct o.
  - apply (bunifyN_correct1 p n H H0 s). auto.
  - apply (bunifyN_correct2 p n H H0). auto.
Qed.

Theorem bunify_correct : ∀ (p : poly),
  is_poly p →
  match bunify p with
  | Some s ⇒ mgu s p
  | None ⇒ ¬ unifiable p
  end.
Proof.
  intros.
  apply bunifyN_correct.
  - apply H.
  - 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*.