Contents

Chapter 1

Library B_Unification.terms

```
Require Import Bool.
Require Import Omega.
Require Import EqNat.
Require Import List.
Require Import Setoid.
Import ListNotations.
```

1.1 Introduction

In order for any proofs to be constructed in Coq, we need to formally define the logic and data across which said proofs will operate. Since the heart of our analysis is concerned with the unification of Boolean equations, it stands to reason that we should articulate precisely how algebra functions with respect to Boolean rings. To attain this, we shall formalize what an equation looks like, how it can be composed inductively, and also how substitutions behave when applied to equations.

1.2 Terms

1.2.1 Definitions

We shall now begin describing the rules of Boolean arithmetic as well as the nature of Boolean equations. For simplicity's sake, from now on we shall be referring to equations as terms.

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

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

A term, as has already been previously described, is now inductively declared to hold either a constant value, a single variable, a sum of terms, or a product of terms.

```
Inductive term: Type :=
```

| $T0$: $term$
| $T1$: $term$
| VAR : $var \to term$
| SUM : $term \to term \to term$
| $PRODUCT$: $term \to term \to term$.

For convenience's sake, we define some shorthanded notation for readability.

```
Implicit Types x y z : term.
Implicit Types n m : var.
Notation "x + y" := (SUM x y) (at level 50, left associativity).
Notation "x * y" := (PRODUCT x y) (at level 40, left associativity).
```

1.2.2 Axioms

Now that we have informed Coq on the nature of what a term is, it is now time to propose a set of axioms that will articulate exactly how algebra behaves across Boolean rings. This is a requirement since the very act of unifying an equation is intimately related to solving it algebraically. Each of the axioms proposed below describe the rules of Boolean algebra precisely and in an unambiguous manner. None of these should come as a surprise to the reader; however, if one is not familiar with this form of logic, the rules regarding the summation and multiplication of identical terms might pose as a source of confusion.

For reasons of keeping Coq's internal logic consistent, we roll our own custom equivalence relation as opposed to simply using '='. This will provide a surefire way to avoid any odd errors from later cropping up in our proofs. Of course, by doing this we introduce some implications that we will need to address later.

```
Parameter eqv : term → term → Prop.
Infix " == " := eqv (at level 70).
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).
Axiom term_sum_symmetric :
  ∀ x y z, x == y ↔ x + z == y + z.
```

```
Axiom term_product_symmetric :
  ∀ x y z, x == y ↔ x × z == y × z.

Axiom refl_comm :
∀ t1 t2, t1 == t2 → t2 == t1.

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

Now that the core axioms have been taken care of, we need to handle the implications posed by our custom equivalence relation. Below we inform Coq of the behavior of our equivalence relation with respect to rewrites during proofs.

```
Axiom eqv_ref : Reflexive eqv.
Axiom eqv_sym : Symmetric eqv.
Axiom eqv_trans : Transitive eqv.

Add Parametric Relation : term eqv
  reflexivity proved by @eqv_ref
  symmetry proved by @eqv_sym
  transitivity proved by @eqv_trans
  as eq_set_rel.

Axiom SUM_compat :
  ∀ x x', x == x' →
  ∀ y y', y == y' →
    (x + y) == (x' + y').

Axiom PRODUCT_compat :
  ∀ x x', x == x' →
  ∀ y y', y == y' →
    (x × y) == (x' × y').

Add Parametric Morphism : SUM with
  signature eqv ==> eqv ==> eqv as SUM_mor.
Proof.
exact SUM_compat.
Qed.

Add Parametric Morphism : PRODUCT with
  signature eqv ==> eqv ==> eqv as PRODUCT_mor.
Proof.
exact PRODUCT_compat.
Qed.

Hint Resolve eqv_ref eqv_sym eqv_trans SUM_compat PRODUCT_compat.
```

1.2.3 Lemmas

Since Coq now understands the basics of Boolean algebra, it serves as a good exercise for us to generate some further rules using Coq's proving systems. By doing this, not only do we gain some additional tools that will become handy later down the road, but we also test whether our axioms are behaving as we would like them to.

```
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. apply sum_x_x.
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. rewrite term_sum_symmetric with (y := y) (z := y). rewrite sum_x_x.
  apply H.
Qed.
```

```
Hint Resolve mul_x_x_plus_T1.
Hint Resolve x_equal_y_x_plus_y.
```

These lemmas just serve to make certain rewrites regarding the core axioms less tedious to write. While one could certainly argue that they should be formulated as axioms and not lemmas due to their triviality, being pedantic is a good exercise.

```
Lemma sum_id_sym :
  ∀ x, x + T0 == x.
Proof.
intros. rewrite sum_comm. apply sum_id.
Qed.
```

```
Lemma mul_id_sym :
  ∀ x, x × T1 == x.
Proof.
intros. rewrite mul_comm. apply mul_id.
Qed.
```

```
Lemma mul_T0_x_sym :
  ∀ x, x × T0 == T0.
Proof.
intros. rewrite mul_comm. apply mul_T0_x.
Qed.
```

```
Lemma sum_assoc_opp :
```

$\forall$ x y z, x + (y + z) == (x + y) + z.
Proof.
Admitted.

Lemma *mul_assoc_opp* :
$\forall$ x y z, x $\times$ (y $\times$ z) == (x $\times$ y) $\times$ z.
Proof.
Admitted.

1.3 Variable Sets

Now that the underlying behavior concerning Boolean algebra has been properly articulated to Coq, it is now time to begin formalizing the logic surrounding our meta reasoning of Boolean equations and systems. While there are certainly several approaches to begin this process, we thought it best to ease into things through formalizing the notion of a set of variables present in an equation.

1.3.1 Definitions

We now define a variable set to be precisely a list of variables; additionally, we include several functions for including and excluding variables from these variable sets. Furthermore, since uniqueness is not a property guaranteed by Coq lists and it has the potential to be desirable, we define a function that consumes a variable set and removes duplicate entries from it. For convenience, we also provide several examples to demonstrate the functionalities of these new definitions.

```
Definition var_set := list var.
Implicit Type vars: var_set.

Fixpoint var_set_includes_var (v : var) (vars : var_set) : bool :=
  match vars with
    | nil ⇒ false
    | n :: n' ⇒ if (beq_nat v n) then true else var_set_includes_var v n'
  end.

Fixpoint var_set_remove_var (v : var) (vars : var_set) : var_set :=
  match vars with
    | nil ⇒ nil
    | n :: n' ⇒ if (beq_nat v n) then (var_set_remove_var v n') else n :: (var_set_remove_var
v n')
  end.

Fixpoint var_set_create_unique (vars : var_set) (found_vars : var_set) : var_set :=
  match vars with
    | nil ⇒ nil
    | n :: n' ⇒
```

```
    if (var_set_includes_var n found_vars) then var_set_create_unique n' (n :: found_vars)
    else n :: var_set_create_unique n' (n :: found_vars)
  end.

Fixpoint var_set_is_unique (vars : var_set) (found_vars : var_set) : bool :=
  match vars with
    | nil => true
    | n :: n' =>
    if (var_set_includes_var n found_vars) then false
    else var_set_is_unique n' (n :: found_vars)
  end.

Fixpoint term_vars (t : term) : var_set :=
  match t with
    | T0 => nil
    | T1 => nil
    | VAR x => x :: nil
    | PRODUCT x y => (term_vars x) ++ (term_vars y)
    | SUM x y => (term_vars x) ++ (term_vars y)
  end.

Definition term_unique_vars (t : term) : var_set :=
  (var_set_create_unique (term_vars t) []).
```

1.3.2 Examples

```
Example var_set_create_unique_ex1 :
  var_set_create_unique [0;5;2;1;1;2;2;9;5;3] [] = [0;5;2;1;9;3].
Proof.
simpl. reflexivity.
Qed.

Example var_set_is_unique_ex1 :
  var_set_is_unique [0;2;2;2] [] = false.
Proof.
simpl. reflexivity.
Qed.

Example term_vars_ex1 :
  term_vars (VAR 0 + VAR 0 + VAR 1) = [0;0;1].
Proof.
simpl. reflexivity.
Qed.

Example term_vars_ex2 :
  In 0 (term_vars (VAR 0 + VAR 0 + VAR 1)).
```

```
Proof.
simpl. left. reflexivity.
Qed.
```

1.4 Ground Terms

Seeing as we just outlined the definition of a variable set, it seems fair to now formalize the definition of a ground term, or in other words, a term that has no variables and whose variable set is the empty set.

1.4.1 Definitions

A ground term is a recursively defined proposition that is only True if and only if no variable appears in it; otherwise it will be a False proposition and no longer a ground term.

```
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.
```

1.4.2 Lemmas

Our first real lemma (shown below), articulates an important property of ground terms: all ground terms are equvialent to either 0 or 1. This curious property is a direct result of the fact that these terms possess no variables and additioanlly because of the axioms of Boolean algebra.

```
Lemma ground_term_equiv_T0_T1 :
  forall 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 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.
```

- 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.
Qed.

This lemma, while intuitively obvious by definition, nonetheless provides a formal bridge between the world of ground terms and the world of variable sets.

Lemma *ground_term_has_empty_var_set* :
  $\forall$ *x*, (*ground_term x*) $\rightarrow$ (*term_vars x*) = [].
Proof.
intros. induction *x*.
- simpl. reflexivity.
- simpl. reflexivity.
- *contradiction*.
- firstorder. unfold *term_vars*. unfold *term_vars* in *H2*. rewrite *H2*. unfold *term_vars* in *H1*. rewrite *H1*. simpl. reflexivity.
- firstorder. unfold *term_vars*. unfold *term_vars* in *H2*. rewrite *H2*. unfold *term_vars* in *H1*. rewrite *H1*. simpl. reflexivity.
Qed.

1.4.3 Examples

Here are some examples to show that our ground term definition is working appropriately.

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

Example *ex_gt2* :
  (*ground_term* (*VAR* 0 $\times$ *T1*)) $\rightarrow$ *False*.
Proof.
simpl. intros. destruct *H*. apply *H*.
Qed.

1.5 Substitutions

It is at this point in our Coq development that we begin to officially define the principal action around which the entirety of our efforts are centered: the act of substituting variables with other terms. While substitutions alone are not of great interest, their emergent properties as in the case of whether or not a given substitution unifies an equation are of substantial importance to our later research.

1.5.1 Definitions

Here we define a substitution to be a list of ordered pairs where each pair represents a variable being mapped to a term. For sake of clarity these ordered pairs shall be referred to as replacements from now on and as a result, substitutions should really be considered to be lists of replacements.

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

Definition subst := list replacement.

Implicit Type s : subst.
```

Our first function, find_replacement, is an auxilliary to apply_subst. This function will search through a substitution for a specific variable, and if found, returns the variable's associated term.

```
Fixpoint find_replacement (x : var) (s : subst) : term :=
  match s with
  | nil ⇒ VAR x
  | r :: r' ⇒
      if beq_nat (fst r) x then (snd r)
      else
        (find_replacement x r')
  end.
```

The apply_subst function will take a term and a substitution and will produce a new term reflecting the changes made to the original one.

```
Fixpoint apply_subst (t : term) (s : subst) : term :=
  match t with
  | T0 ⇒ T0
  | T1 ⇒ T1
  | VAR x ⇒ (find_replacement x s)
  | PRODUCT x y ⇒ PRODUCT (apply_subst x s) (apply_subst y s)
  | SUM x y ⇒ SUM (apply_subst x s) (apply_subst y s)
  end.
```

For reasons of completeness, it is useful to be able to generate identity substitutions; namely, substitutions that map the variables of a term's variable set to themselves.

```
Fixpoint build_id_subst (lvar : var_set) : subst :=
  match lvar with
  | nil ⇒ nil
```

```
  | v :: v' ⇒ (cons (v , (VAR v))
                              (build_id_subst v'))
  end.
```

Since we now have the ability to generate identity substitutions, we should now formalize a general proposition for testing whether or not a given substitution is an identity substitution of a given term.

```
Definition subst_equiv (s1 s2: subst) : Prop :=
  ∀ r, In r s1 ↔ In r s2.

Definition subst_is_id_subst (t : term) (s : subst) : Prop :=
  (subst_equiv (build_id_subst (term_vars t)) s).
```

1.5.2 Lemmas

Having now outlined the functionality of a subsitution, let us now begin to analyze some implications of its form and composition by proving some lemmas.

```
Lemma apply_subst_compat : ∀ (t t' : term),
     t == t' → ∀ (sigma: subst), (apply_subst t sigma) == (apply_subst t' sigma).
Proof.
Admitted.

Add Parametric Morphism : apply_subst with
      signature eqv ==> eq ==> eqv as apply_subst_mor.
Proof.
  exact apply_subst_compat.
Qed.

Lemma id_subst_does_not_modify :
  ∀ s x, (subst_is_id_subst x s) → (apply_subst x s) == x.
Proof.
Admitted.
```

An easy thing to prove right off the bat is that ground terms, i.e. terms with no variables, cannot be modified by applying substitutions to them. This will later prove to be very relevant when we begin to talk about unification.

```
Lemma ground_term_cannot_subst :
  ∀ x, (ground_term x) → (∀ s, apply_subst x s == x).
Proof.
intros. induction s.
  - apply ground_term_equiv_T0_T1 in H. destruct H.
  + rewrite H. simpl. reflexivity.
  + rewrite H. simpl. reflexivity.
  - apply ground_term_equiv_T0_T1 in H. destruct H. rewrite H.
    + simpl. reflexivity.
```

+ `rewrite` H. `simpl`. `reflexivity`.
`Qed`.

The last major thing to prove about substitutions is their distributivity and associativity. Again the importance of these proofs will not become apparent until we talk about unification.

`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`. `reflexivity`.
`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`. `simpl`. `reflexivity`.
`Qed`.

`Lemma` *subst_sum_distr_opp* :
  ∀ *s x y*, *apply_subst* (*x* + *y*) *s* == *apply_subst x s* + *apply_subst y s*.
`Proof`.
  `intros`.
  `apply` *refl_comm*.
  `apply` *subst_distribution*.
`Qed`.

`Lemma` *subst_mul_distr_opp* :
  ∀ *s x y*, *apply_subst* (*x* × *y*) *s* == *apply_subst x s* × *apply_subst y s*.
`Proof`.
  `intros`.
  `apply` *refl_comm*.
  `apply` *subst_associative*.
`Qed`.

`Lemma` *var_subst*:
  ∀ (*v* : *var*) (*ts* : *term*) ,
  (*apply_subst* (*VAR v*) (*cons* (*v* , *ts*) *nil*)) == *ts*.
`Proof`.
Admitted.

`Lemma` *id_subst*:
  ∀ (*t* : *term*),
  apply_subst t (*build_id_subst* (*term_unique_vars t*)) == *t*.
`Proof`.
Admitted.

1.5.3 Examples

1.6 Unification

```
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. unfold unifies in H. simpl in H.
Admitted.

Example ex_unif2 :
  unifies (VAR 0) (VAR 1) ((0, T1) :: (1, T1) :: nil).
Proof.
unfold unifies. simpl. reflexivity.
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. rewrite H.
  rewrite sum_x_x. reflexivity.
}
{
  intros. unfold unifies_T0 in H. unfold unifies.
  rewrite term_sum_symmetric with (x := apply_subst x s + apply_subst y s)
  (z := apply_subst y s) in H. rewrite sum_id in H.
  rewrite sum_comm in H.
  rewrite sum_comm with (y := apply_subst y s) in H.
  rewrite ← sum_assoc in H.
  rewrite sum_x_x in H.
  rewrite sum_id in H.
  apply H.
}
Qed.

Definition unifier (t : term) (s : subst) : Prop :=
  (apply_subst t s) == T0.

Example unifier_ex1 :
```

```
  ~(unifier (VAR 0) ((1, T1) :: nil)).
Proof.
unfold unifier. simpl. intuition.
Admitted.

Example unifier_ex2 :
  ~(unifier (VAR 0) ((0, VAR 0) :: nil)).
Proof.
unfold unifier. simpl. intuition.
Admitted.

Example unifier_ex3 :
  (unifier (VAR 0) ((0, T0) :: nil)).
Proof.
unfold unifier. simpl. reflexivity.
Qed.

Lemma unifier_distribution :
  ∀ x y s, (unifies_T0 x y s) ↔ (unifier (x + y) s).
Proof.
intros. split.
{
  intros. unfold unifies_T0 in H. unfold unifier.
  rewrite ← H. symmetry. apply subst_distribution.
}
{
  intros. unfold unifies_T0. unfold unifier in H.
  rewrite ← H. apply subst_distribution.
}
Qed.

Lemma unifier_subset_imply_superset :
  ∀ s t r, unifier t s → unifier t (r :: s).
Proof.
intros. induction s.
{
  unfold unifier in *. simpl in *.
Admitted.

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.
Admitted.
```

```
Example unifiable_ex2 :
  ∀ x, unifiable (x + x + T1) → False.
Proof.
intros. unfold unifiable in H. unfold unifier in H.
Admitted.
```

```
Example unifiable_ex3 :
  ∃ x, unifiable (x + T1).
Proof.
∃ (T1). unfold unifiable. unfold unifier.
∃ nil. simpl. rewrite sum_x_x. reflexivity.
Qed.
```

1.7 Most General Unifier

```
Definition subst_compose (s s' delta : subst) (t : term) : Prop :=
    apply_subst t s' == apply_subst (apply_subst t s) delta.
```

```
Definition more_general_subst (s s': subst) (t : term) : Prop :=
  ∃ delta, subst_compose s s' delta t.
```

```
Notation "u1 <_u2 { t }" := (more_general_subst u1 u2 t) (at level 51, left associativity).
```

```
Definition mgu (t : term) (s : subst) : Prop :=
  (unifier t s) ∧ (∀ (s' : subst), unifier t s' → (more_general_subst s s' t) ).
```

```
Definition reprod_unif (t : term) (s : subst) : Prop :=
  unifier t s ∧
  ∀ u,
  unifier t u →
  subst_compose s u u t.
```

```
Lemma reprod_is_mgu : ∀ (t : term) (u : subst),
  reprod_unif t u →
  mgu t u.
Proof.
Admitted.
```

```
Example mgu_ex1 :
  mgu (VAR 0 × VAR 1) ((0, VAR 0 × (T1 + VAR 1)) :: nil).
Proof.
unfold mgu. unfold unifier. simpl. unfold more_general_subst. simpl. split.
{
  rewrite distr. rewrite mul_comm with (y := T1). rewrite mul_id.
  rewrite mul_comm. rewrite distr. rewrite mul_comm with (x := VAR 0).
  rewrite ← mul_assoc with (x := VAR 1) (y := VAR 1). rewrite mul_x_x.
```

```
  rewrite sum_x_x. reflexivity.
}
{
  intros. unfold subst_compose.
Admitted.
```

1.8 Auxilliary Computational Operations and Simplifications

```
Fixpoint identical (a b: term) : bool :=
  match a , b with
    | T0, T0 => true
    | T0, _ => false
    | T1 , T1 => true
    | T1 , _ => false
    | VAR x , VAR y => if beq_nat x y then true else false
    | VAR x, _ => false
    | PRODUCT x y, PRODUCT x1 y1 => if ((identical x x1) && (identical y y1)) then
true
                                        else false
    | PRODUCT x y, _ => false
    | SUM x y, SUM x1 y1 => if ((identical x x1) && (identical y y1)) then true
                                        else false
    | SUM x y, _ => false
  end.

Definition plus_one_step (a b : term) : term :=
  match a, b with
    | T0, _ => b
    | T1, T0 => T1
    | T1, T1 => T0
    | T1 , _ => SUM a b
    | VAR x , T0 => a
    | VAR x , _ => if identical a b then T0 else SUM a b
    | PRODUCT x y , T0 => a
    | PRODUCT x y, _ => if identical a b then T0 else SUM a b
    | SUM x y , T0 => a
    | SUM x y, _ => if identical a b then T0 else SUM a b
  end.

Definition mult_one_step (a b : term) : term :=
  match a, b with
```

```
    | T0, _ ⇒ T0
    | T1 , _ ⇒ b
    | VAR x , T0 ⇒ T0
    | VAR x , T1 ⇒ a
    | VAR x , _ ⇒ if identical a b then a else PRODUCT a b
    | PRODUCT x y , T0 ⇒ T0
    | PRODUCT x y , T1 ⇒ a
    | PRODUCT x y, _ ⇒ if identical a b then a else PRODUCT a b
    | SUM x y , T0 ⇒ T0
    | SUM x y , T1 ⇒ a
    | SUM x y, _ ⇒ if identical a b then a else SUM a b
  end.

Fixpoint simplify (t : term) : term :=
  match t with
    | T0 ⇒ T0
    | T1 ⇒ T1
    | VAR x ⇒ VAR x
    | PRODUCT x y ⇒ mult_one_step (simplify x) (simplify y)
    | SUM x y ⇒ plus_one_step (simplify x) (simplify y)
  end.

Fixpoint Simplify_N (t : term) (counter : nat): term :=
  match counter with
    | O ⇒ t
    | S n' ⇒ (Simplify_N (simplify t) n')
  end.
```

Chapter 2

Library B_Unification.lowenheim_formula

```
Require Export terms.
Require Import List.
Import ListNotations.

Fixpoint build_on_list_of_vars (list_var : var_set) (s : term) (sig1 : subst) (sig2 : subst) :
subst :=
  match list_var with
   | nil ⇒ nil
   | v' :: v ⇒
      (cons (v' , (s + T1) × (apply_subst (VAR v') sig1 ) + s × (apply_subst (VAR v' )
sig2 ) )
            (build_on_list_of_vars v s sig1 sig2) )
  end.

Definition build_lowenheim_subst (t : term) (tau : subst) : subst :=
  build_on_list_of_vars (term_unique_vars t) t (build_id_subst (term_unique_vars t)) tau.
```

2.2 Lowenheim's algorithm

```
Definition update_term (t : term) (s' : subst) : term :=
  (simplify (apply_subst t s' ) ).

Definition term_is_T0 (t : term) : bool :=
  (identical t T0).

Inductive subst_option: Type :=
    | Some_subst : subst → subst_option
    | None_subst : subst_option.

Fixpoint rec_subst (t : term) (vars : var_set) (s : subst) : subst :=
```

```
  match vars with
    | nil ⇒ s
    | v' :: v ⇒
        if (term_is_T0
              (update_term (update_term t (cons (v' , T0) s) )
                           (rec_subst (update_term t (cons (v' , T0) s) )
                                      v (cons (v' , T0) s)) )
                          )
          then
                (rec_subst (update_term t (cons (v' , T0) s) )
                                      v (cons (v' , T0) s))
         else
          if (term_is_T0
               (update_term (update_term t (cons (v' , T1) s) )
                            (rec_subst (update_term t (cons (v' , T1) s) )
                                       v (cons (v' , T1) s)) ) )
          then
                (rec_subst (update_term t (cons (v' , T1) s) )
                                      v (cons (v' , T1) s))
          else
                (rec_subst (update_term t (cons (v' , T0) s) )
                                      v (cons (v' , T0) s))
     end.

Compute (rec_subst ((VAR 0) × (VAR 1)) (cons 0 (cons 1 nil)) nil) .

Fixpoint find_unifier (t : term) : subst_option :=
  match (update_term t (rec_subst t (term_unique_vars t) nil) ) with
    | T0 ⇒ Some_subst (rec_subst t (term_unique_vars t) nil)
    | _ ⇒ None_subst
  end.

Compute (find_unifier ((VAR 0) × (VAR 1))).
Compute (find_unifier ((VAR 0) + (VAR 1))).
Compute (find_unifier ((VAR 0) + (VAR 1) + (VAR 2) + T1 + (VAR 3) × ( (VAR 2) + (VAR
0)) )).

Definition Lowenheim_Main (t : term) : subst_option :=
  match (find_unifier t) with
    | Some_subst s ⇒ Some_subst (build_lowenheim_subst t s)
    | None_subst ⇒ None_subst
  end.

Compute (find_unifier ((VAR 0) × (VAR 1)) ) .

Compute (Lowenheim_Main ((VAR 0) × (VAR 1))).
```

```
Compute (Lowenheim_Main ((VAR 0) + (VAR 1)) ).
Compute (Lowenheim_Main ((VAR 0) + (VAR 1) + (VAR 2) + T1 + (VAR 3) × ( (VAR 2) +
(VAR 0)) ) ).
Compute (Lowenheim_Main (T1)).
Compute (Lowenheim_Main (( VAR 0) + (VAR 0) + T1)).
```

2.3 Lowenheim testing

```
Definition Test_find_unifier (t : term) : bool :=
  match (find_unifier t) with
    | Some_subst s ⇒
      (term_is_T0 (update_term t s))
    | None_subst ⇒ true
  end.

Compute (Test_find_unifier (T1)).
Compute (Test_find_unifier ((VAR 0) × (VAR 1))).
Compute (Test_find_unifier ((VAR 0) + (VAR 1) + (VAR 2) + T1 + (VAR 3) × ( (VAR 2) +
(VAR 0)) )).

Definition apply_lowenheim_main (t : term) : term :=
  match (Lowenheim_Main t) with
  | Some_subst s ⇒ (apply_subst t s)
  | None_subst ⇒ T1
  end.

Compute (Lowenheim_Main ((VAR 0) × (VAR 1) )).
Compute (apply_lowenheim_main ((VAR 0) × (VAR 1) ) ).

Compute (Lowenheim_Main ((VAR 0) + (VAR 1) )).
Compute (apply_lowenheim_main ((VAR 0) + (VAR 1) ) ).
```

Chapter 3

Library B_Unification.lowenheim_proof

```
Require Export terms.
Require Export lowenheim_formula.
Require Export EqNat.
Require Import List.
Import ListNotations.
```

3.1 Declarations and their lemmas useful for the proof

```
Definition sub_term (t : term) (t' : term) : Prop :=
  ∀ (x : var ),
  (In x (term_unique_vars t) ) → (In x (term_unique_vars t')) .
Lemma sub_term_id :
  ∀ (t : term),
  sub_term t t.
 Proof.
Admitted.
```

3.2 Proof that Lownheim's algorithm unifes a given term

```
Lemma helper_1:
∀ (t' s : term) (v : var) (sig1 sig2 : subst),
  sub_term (VAR v) t' →
  apply_subst (VAR v) (build_on_list_of_vars (term_unique_vars t') s sig1 sig2)
  ==
  apply_subst (VAR v) (build_on_list_of_vars (term_unique_vars (VAR v)) s sig1 sig2).
Proof.
Admitted.
Lemma helper_2a:
```

```
  ∀ (t1 t2 t' : term),
  sub_term (t1 + t2) t' → sub_term t1 t'.
Proof.
Admitted.

Lemma helper_2b:
  ∀ (t1 t2 t' : term),
  sub_term (t1 + t2) t' → sub_term t2 t'.
Proof.
Admitted.

Lemma subs_distr_vars_ver2 :
  ∀ (t t' : term) (s : term) (sig1 sig2 : subst),
  (sub_term t t') →
  apply_subst t (build_on_list_of_vars (term_unique_vars t') s sig1 sig2)
    ==
  (s + T1) × (apply_subst t sig1) + s × (apply_subst t sig2).
Proof.
 intros. generalize dependent t'. induction t.
  - intros t'. repeat rewrite ground_term_cannot_subst.
     + rewrite mul_comm with (x := s + T1). rewrite distr. repeat rewrite mul_T0_x.
rewrite mul_comm with (x := s).
       rewrite mul_T0_x. repeat rewrite sum_x_x. reflexivity.
     + unfold ground_term. reflexivity.
     + unfold ground_term. reflexivity.
     + unfold ground_term. reflexivity.
  - intros t'. repeat rewrite ground_term_cannot_subst.
     + rewrite mul_comm with (x := s + T1). rewrite mul_id. rewrite mul_comm with
(x := s). rewrite mul_id. rewrite sum_comm with (x := s).
       repeat rewrite sum_assoc. rewrite sum_x_x. rewrite sum_comm with (x := T1).
rewrite sum_id. reflexivity.
     + unfold ground_term. reflexivity.
     + unfold ground_term. reflexivity.
     + unfold ground_term. reflexivity.
  - intros. rewrite helper_1 .
     + unfold term_unique_vars. unfold term_vars. unfold var_set_create_unique. unfold
var_set_includes_var. unfold build_on_list_of_vars.
     rewrite var_subst. reflexivity.
     + apply H.
  - intros. specialize (IHt1 t'). specialize (IHt2 t'). repeat rewrite subst_sum_distr_opp.
       rewrite IHt1. rewrite IHt2.
     + rewrite distr. rewrite distr. repeat rewrite sum_assoc. rewrite sum_comm with
(x := (s + T1) × apply_subst t2 sig1)
       (y := (s × apply_subst t1 sig2 + s × apply_subst t2 sig2)). repeat rewrite sum_assoc.
```

```
      rewrite sum_comm with (x := s × apply_subst t2 sig2 ) (y := (s + T1) × apply_subst
t2 sig1).
      repeat rewrite sum_assoc. reflexivity.
    + pose helper_2b as H2. specialize (H2 t1 t2 t'). apply H2. apply H.
    + pose helper_2a as H2. specialize (H2 t1 t2 t'). apply H2. apply H.
  - intros. specialize (IHt1 t'). specialize (IHt2 t'). repeat rewrite subst_mul_distr_opp.
rewrite IHt1. rewrite IHt2.
    + rewrite distr. rewrite mul_comm with (y := ((s + T1) × apply_subst t2 sig1)).
    rewrite distr. rewrite mul_comm with (y := (s × apply_subst t2 sig2)). rewrite
distr.
    repeat rewrite mul_assoc. repeat rewrite mul_comm with (x := apply_subst t2
sig1).
    repeat rewrite mul_assoc.
    rewrite mul_assoc_opp with (x := (s + T1)) (y := (s + T1)) . rewrite mul_x_x.
    rewrite mul_assoc_opp with (x := (s + T1)) (y := s). rewrite mul_comm with (x :=
(s + T1)) (y := s).
    rewrite distr. rewrite mul_x_x. rewrite mul_id_sym. rewrite sum_x_x. rewrite
mul_T0_x.
    repeat rewrite mul_assoc. rewrite mul_comm with (x := apply_subst t2 sig2 ).
    repeat rewrite mul_assoc. rewrite mul_assoc_opp with (x := s ) (y := (s + T1)).
    rewrite distr. rewrite mul_x_x. rewrite mul_id_sym. rewrite sum_x_x. rewrite
mul_T0_x.
    repeat rewrite sum_assoc. rewrite sum_assoc_opp with (x := T0) (y := T0). rewrite
sum_x_x. rewrite sum_id.
    repeat rewrite mul_assoc. rewrite mul_comm with (x := apply_subst t2 sig2) (y :=
s × apply_subst t1 sig2).
    repeat rewrite mul_assoc. rewrite mul_assoc_opp with (x := s). rewrite mul_x_x.
reflexivity.
    + pose helper_2b as H2. specialize (H2 t1 t2 t'). apply H2. apply H.
    + pose helper_2a as H2. specialize (H2 t1 t2 t'). apply H2. apply H.
Qed.

Lemma specific_sigmas_unify:
  ∀ (t : term) (tau : subst),
  (unifier t tau) →
  (apply_subst t (build_on_list_of_vars (term_unique_vars t) t (build_id_subst (term_unique_vars
t)) tau )
  ) == T0 .
  Proof.
  intros.
  rewrite subs_distr_vars_ver2.
  - rewrite id_subst. rewrite mul_comm with (x := t + T1). rewrite distr. rewrite
mul_x_x. rewrite mul_id_sym. rewrite sum_x_x.
```

```
    rewrite sum_id.
    unfold unifier in H. rewrite H. rewrite mul_T0_x_sym. reflexivity.
  - apply sub_term_id.
Qed.

Lemma lownheim_unifies:
  ∀ (t : term) (tau : subst),
  (unifier t tau) →
  (apply_subst t (build_lowenheim_subst t tau)) == T0.
Proof.
intros. unfold build_lowenheim_subst. apply specific_sigmas_unify. apply H.
Qed.
```

3.3 Proof that Lownheim's algorithm produces a most general unifier
3.3.a Proof that Lownheim's algorithm produces a reproductive unifier

```
Definition reproductive_unifier (t : term) (sig : subst) : Prop :=
  unifier t sig →
  ∀ (tau : subst) (x : var),
  unifier t tau →
  (apply_subst (apply_subst (VAR x) sig ) tau) == (apply_subst (VAR x) tau).

Lemma term_ident_prop :
 ∀ (t1 t2 : term),
  match identical t1 t2 with
   | true ⇒ True
   | false ⇒ False
  end.
 Proof.
Admitted.

Lemma distr_opp :
 ∀ x y z, x × y + x × z == x × ( y + z).
Proof.
Admitted.

Lemma lowenheim_rephrase1 :
  ∀ (t : term) (tau : subst) (x : var),
  (unifier t tau) →
  (In x (term_unique_vars t)) →
  (apply_subst (VAR x) (build_lowenheim_subst t tau)) ==
  (t + T1) × (VAR x) + t × (apply_subst (VAR x) tau).
  Proof.
intros.
induction t.
  - unfold build_lowenheim_subst. unfold term_unique_vars. unfold term_vars. unfold
var_set_create_unique.
```

unfold build_id_subst. unfold build_on_list_of_vars. rewrite *mul_comm* with ($y :=$ VAR x). rewrite *distr*.

rewrite mul_T0_x_sym. rewrite *sum_id*. rewrite *mul_T0_x*. rewrite mul_id_sym. rewrite sum_id_sym. unfold apply_subst. reflexivity.

- unfold term_unique_vars in $H0$. unfold term_vars in $H0$. unfold var_set_create_unique in $H0$. unfold In in $H0$. destruct $H0$.

- unfold build_lowenheim_subst. unfold term_unique_vars. unfold term_vars. unfold var_set_create_unique.

unfold var_set_includes_var. unfold term_unique_vars in $H0$. unfold term_vars in $H0$. unfold var_set_create_unique in $H0$.

unfold var_set_includes_var in $H0$. unfold In in $H0$. simpl in $H0$. destruct $H0$.

+ rewrite $H0$. unfold build_id_subst. unfold build_on_list_of_vars. simpl. destruct beq_nat.

{ reflexivity. }

{ rewrite *mul_comm* with ($y :=$ VAR x). rewrite *distr*. rewrite *mul_x_x*. rewrite mul_id_sym. rewrite *sum_x_x*. rewrite *sum_id*.

rewrite $H0$ in H. unfold unifier in H. rewrite H. rewrite mul_T0_x_sym. pose *proof* *term_ident_prop* as $H1$. specialize ($H1$ (VAR x) T0).

simpl in $H1$. destruct $H1$. }

+ destruct $H0$.

- unfold unifier in H. rewrite subst_sum_distr_opp in H. pose *proof* unifies_T0_equiv as $H5$. specialize ($H5$ $t1$ $t2$ tau).

unfold unifies in $H5$. unfold unifies_T0 in $H5$. rewrite $\leftarrow$ $H5$ in H.

Admitted.

Lemma lowenheim_rephrase2 :

$\forall$ (t : **term**) (tau : subst) (x : var),

(unifier t tau) $\rightarrow$

$\neg$ (In x (term_unique_vars t)) $\rightarrow$

(apply_subst (VAR x) (build_lowenheim_subst t tau)) ==

(VAR x).

Proof.

Admitted.

@@ dd - this will be hard to prove! need a detour into decidability, etc... Trt to avoid it!

Advice for now: don't for now try to prove "reproductive", just prove "mgu". THAT IS, only prove the sub condition for variables in t. Then (I think) you don't need var_in_out_list

Lemma var_in_out_list:

$\forall$ (x : var) ($lvar$: **list** var),

(In x $lvar$) $\vee$ $\neg$ (In x $lvar$).

Proof.

Admitted.

Lemma lowenheim_reproductive:

```
  ∀ (t : term) (tau : subst),
  (unifier t tau) →
  reproductive_unifier t (build_lowenheim_subst t tau) .
Proof.
 intros. unfold reproductive_unifier. intros.
  pose proof var_in_out_list. specialize (H2 x (term_unique_vars t)). destruct H2.
  {
  rewrite lowenheim_rephrase1.
  - rewrite subst_sum_distr_opp. rewrite subst_mul_distr_opp. rewrite subst_mul_distr_opp.
    unfold unifier in H1. rewrite H1. rewrite mul_T0_x. rewrite subst_sum_distr_opp.
    rewrite H1. rewrite ground_term_cannot_subst.
    + rewrite sum_id. rewrite mul_id. rewrite sum_comm. rewrite sum_id. reflexivity.
    + unfold ground_term. intuition.
  - apply H.
  - apply H2.
  }
  { rewrite lowenheim_rephrase2.
    - reflexivity.
    - apply H.
    - apply H2.
  }
Qed.
```

3.3.b lowenheim builder gives a most general unifier

```
Definition substitution_composition (s s' delta : subst) (t : term) : Prop :=
  ∀ (x : var), apply_subst (apply_subst (VAR x) s) delta == apply_subst (VAR x) s' .

Definition more_general_substitution (s s': subst) (t : term) : Prop :=
  ∃ delta, substitution_composition s s' delta t.

Definition most_general_unifier (t : term) (s : subst) : Prop :=
  (unifier t s) → (∀ (s' : subst), unifier t s' → more_general_substitution s s' t ).

Lemma reproductive_is_mgu : ∀ (t : term) (u : subst),
  reproductive_unifier t u →
  most_general_unifier t u.
Proof.
 intros. unfold most_general_unifier. unfold reproductive_unifier in H.
  unfold more_general_substitution . unfold substitution_composition.
  intros. specialize (H H0). ∃ s' . intros. specialize (H s' x). specialize (H
H1). apply H.
Qed.

Lemma lowenheim_most_general_unifier:
  ∀ (t : term) (tau : subst),
  (unifier t tau) →
```

```
  most_general_unifier t (build_lowenheim_subst t tau) .
Proof.
intros. apply reproductive_is_mgu. apply lowenheim_reproductive. apply H.
Qed.
```

3.4 extension to include Main function and subst_option

```
Definition subst_option_is_some (so : subst_option) : bool :=
  match so with
  | Some_subst s ⇒ true
  | None_subst ⇒ false
  end.

Definition convert_to_subst (so : subst_option) : subst :=
  match so with
  | Some_subst s ⇒ s
  | None_subst ⇒ nil
  end.

Lemma find_unifier_is_unifier:
 ∀ (t : term),
  (unifiable t) → (unifier t (convert_to_subst (find_unifier t))).
Proof.
Admitted.

Lemma builder_to_main:
 ∀ (t : term),
(unifiable t) → most_general_unifier t (build_lowenheim_subst t (convert_to_subst (find_unifier
t))) →
 most_general_unifier t (convert_to_subst (Lowenheim_Main t)) .
Proof.
Admitted.

Lemma lowenheim_main_most_general_unifier:
 ∀ (t: term),
 (unifiable t) → most_general_unifier t (convert_to_subst (Lowenheim_Main t)).
Proof.
 intros. apply builder_to_main.
 - apply H.
 - apply lowenheim_most_general_unifier. apply find_unifier_is_unifier. apply H.
Qed.
```

Chapter 4

Library B_Unification.poly

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

```
Require Export terms.
```

4.1 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.

4.2 Monomials and Polynomials

4.2.1 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.
```

4.2.2 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.
```

4.2.3 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.
```

4.3 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.

Lemma part_add_eq : ∀ f p l r,
  is_poly p →
  partition f p = (l, r) →
  p = addPP l r.
Proof.
Admitted.
```

Chapter 5

Library B_Unification.poly_unif

```
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_unifier : unifier [] [].
Proof.
Admitted.
Lemma empty_mgu : mgu [] [].
Proof.
  unfold mgu, more_general, subst_comp.
  intros.
  simpl.
  split.
  - apply empty_unifier.
  - intros.
    ∃ t.
    intros.
    rewrite (empty_substP _ H0).
    reflexivity.
Qed.
```

Chapter 6

Library B_Unification.sve

6.1 Intro

Here we implement the algorithm for successive variable elimination. The basic idea is to remove a variable from the problem, solve that simpler problem, and build a solution from the simpler solution. The algorithm is recursive, so variables are removed and problems generated until we are left with either of two problems; 1 =B 0 or 0 =B 0. In the former case, the whole original problem is not unifiable. In the latter case, the problem is solved without any need to substitute since there are no variables. From here, we begin the process of building up substitutions until we reach the original problem.

6.2 Eliminating Variables

This section deals with the problem of removing a variable x from a term t. The first thing to notice is that t can be written in polynomial form p. This polynomial is just a set of monomials, and each monomial a set of variables. We can now seperate the polynomials into two sets qx and r. The term qx will be the set of monomials in p that contain the variable x. The term q, or the quotient, is qx with the x removed from each monomial. The term r, or the remainder, will be the monomials that do not contain x. The original term can then be written as $x \times q + r$.

Implementing this procedure is pretty straightforward. We define a function div_by_var that produces two polynomials given a polynomial p and a variable x to eliminate from it. The first step is dividing p into qx and r which is performed using a partition over p with the predicate has_var. The second step is to remove x from qx using the helper elim_var which just maps over the given polynomial removing the given variable.

```
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) : prod poly poly :=
```

```
let (qx, r) := partition (has_var x) p in
(elim_var x qx , r).
```

We would also like to prove some lemmas about varaible elimination that will be helpful in proving the full algorithm correct later. The main lemma below is div_eq, which just asserts that after eliminating x from p into q and r the term can be put back together as in $p = x \times q + r$. This fact turns out to be rather hard to prove and needs the help of 10 or so other sudsidiary lemmas.

```
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.
Admitted.

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.
Admitted.

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.
```

Admitted.

```
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.
```

As explained earlier, given a polynomial p decomposed into a variable x, a quotient q, and a remainder r, div_eq asserts that $p = x \times q + r$.

```
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.
  apply (part_add_eq (has_var x) _ _ _ HP).
  rewrite ← Hq.
  rewrite ← Hr.
  apply Hqr.
Qed.
```

The second main lemma about varaible elimination is below. Given that a term p has been decomposed into the form $x \times q + r$, we can define $p' = (q + 1) \times r$. The lemma div_build_unif states that any unifier of $p =B\ 0$ is also a unifier of $p' =B\ 0$. Much of this proof relies on the axioms of polynomial arithmetic.

This helper function build_poly is used to construct $p' = (q + 1) \times r$ given the quotient and remainder as inputs.

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

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.
  assert (HMx: is_mono [x]). auto.
  apply (div_is_poly x p q r HPp) in HD.
  destruct HD as [HPq HPr].
  assert (is_mono [x] ∧ is_poly q). auto.
  rewrite (mulMP_mulPP_eq _ _ H).
  rewrite mulPP_addPP_1.
  reflexivity.
Qed.
```

6.3 Building Substitutions

This section handles how a solution is built from subproblem solutions. Given that a term p has been decomposed into the form $x \times q + r$, we can define $p' = (q + 1) \times r$. The lemma reprod_build_subst states that if some substitution s is a reproductive unifier of $p' =B\ 0$, then we can build a substitution s' which is a reproductive unifier of $p =B\ 0$. The way s' is built from s is defined in build_subst. Another replacement is added to s of the form $x \to x \times (s(q) + 1) + s(r)$ to construct s'.

```
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 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.
```

6.4 Recursive Algorithm

Now we define the actual algorithm of successive variable elimination. Built using five helper functions, the definition is not too difficult to construct or understand. The general idea, as mentioned before, is to remove one variable at a time, creating simpler problems. Once the simplest problem has been reached, to which the solution is already known, every solution to each subproblem can be built from the solution to the successive subproblem. Formally, given the polynomials $p = x \times q + r$ and $p' = (q + 1) \times r$, the solution to $p =B\ 0$ is built from the solution to $p' =B\ 0$. If s solves $p' =B\ 0$, then $s' = s\ U\ (x \to x \times (s(q) + 1) + s(r))$ solves $p =B\ 0$.

The function sve is the final result, but it is sveVars which actually has all of the meat. Due to Coq's rigid type system, every recursive function must be obviously terminating. This means that one of the arguments must decrease with each nested call. It turns out that Coq's type checker is unable to deduce that continually building polynomials from the quotient and remainder of previous ones will eventually result in 0 or 1. So instead we add a fuel argument that explicitly decreases per recursive call. We use the set of variables in the polynomial for this purpose, since each subsequent call has one less variable.

```
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.
```

6.5 Correctness

Finally, we must show that this algorithm is correct. As discussed in the beginning, the correctness of a unification algorithm is proven for two cases. If the algorithm produces a solution for a problem, then the solution must be most general. If the algorithm produces no solution, then the problem must not be unifiable. These statements have been formalized in the theorem sve_correct with the help of the predicates mgu and unifiable as defined in the library *poly_unif.v*. The two cases of the proof are handled seperately by the lemmas sveVars_some and sveVars_none.

```
Lemma sveVars_some : ∀ (p : poly),
  is_poly p →
  ∀ s, sveVars (vars p) p = Some s →
              mgu s p.
Proof.
Admitted.

Lemma sveVars_none : ∀ (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_some; auto.
  - apply sveVars_none; 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.
```